



Community Experience Distilled

# Building an E-Commerce Application with MEAN

Develop an end-to-end, real-time e-commerce application using the MEAN stack

Adrian Mejia

**[PACKT]** open source\*  
PUBLISHING community experience distilled

# Building an E-Commerce Application with MEAN

Develop an end-to-end, real-time e-commerce application using the MEAN stack

**Adrian Mejia**



BIRMINGHAM - MUMBAI

# Building an E-Commerce Application with MEAN

Copyright © 2015 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 author(s), nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be 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.

First published: November 2015

Production reference: 1161115

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

ISBN 978-1-78528-655-1

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

# Credits

**Author**

Adrian Mejia

**Project Coordinator**

Izzat Contractor

**Reviewers**

Carlos De la Rosa

Dan Shreim

**Proofreader**

Safis Editing

**Commissioning Editor**

Nadeem Bagban

**Indexer**

Monica Ajmera Mehta

**Acquisition Editor**

Reshma Raman

**Production Coordinator**

Conidon Miranda

**Content Development Editor**

Nikhil Potdukhe

**Cover Work**

Conidon Miranda

**Technical Editor**

Parag Topre

**Copy Editor**

Sonia Mathur

# About the Author

**Adrian Mejia** is a software engineer, full stack web developer, writer, and blogger. He has worked for more than 6 years in the fields of web development and software engineering. He has worked on a variety of projects and platforms, ranging from start-ups to enterprises and from embedded systems to web and e-commerce applications.

Adrian loves contributing to open source web-related projects and blogging at <http://adrianmejia.com/>. He started his blog as a reminder to himself of how to solve certain software- and programming-related problems. Later, Adrian noticed that many people found it useful as well, and his blog now gets around 75,000 pageviews every month.

He holds a master's degree in software engineering from Rochester Institute of Technology (RIT) in Rochester, New York. Adrian has worked at ADTRAN, Inc. as a software engineer since 2012.

---

I would like to thank my girlfriend, Nathalie, for understanding my late-night writing sessions. I would also like to give deep thanks to my mother, Reina, and Avelino for encouraging me to step into the writing world.

---

# About the Reviewers

**Carlos De la Rosa** is a developer who constantly looks for new challenges in the realm of software technologies and paradigms. Residing in the Dominican Republic, he chose to venture into the software development field because he realized that information is the most important asset in the majority of companies today and that our generation continuously benefits from it. These days, our generation has more resources to change the world, improve in order to achieve personal growth, and get more educated. Becoming an outstanding developer requires a lot of effort, but Carlos believes that being part of a group of people who constantly bring about change is worth it. He has been working with Java EE for around 2 years. Currently, Carlos is getting the experience required to master the MEAN stack.

You can find him on Github at <https://bitbucket.org/chaarlie/>.

**Dan Shreim** worked as a frontend web developer for over 15 years, specializing in AngularJS, user experience, and interface design. He worked in conjunction with multiple award-winning agencies on projects for numerous household name brands around the world in the entertainment, finance, security, and business sectors.

Previously, Dan worked in Toronto, Canada, designing loyalty platforms and London, UK, building prototypes for usability testing. Currently, he works as an interface developer for a leading cyber security company in Colorado.

For more information on Dan, you can visit his site at <http://snapjay.com>.

---

I'd like to thank Jasper for his compassion and support over the past few years.

---

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit [www.PacktPub.com](http://www.PacktPub.com).

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

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



<https://www2.packtpub.com/books/subscription/packtlib>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read Packt's entire library of books.

## Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

## Free access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

# Table of Contents

<b>Preface</b>	<b>vii</b>
<b>Chapter 1: Getting Started with the MEAN Stack</b>	<b>1</b>
<b>Introducing the MEAN stack</b>	<b>2</b>
NodeJS	3
ExpressJS	4
MongoDB	4
AngularJS	4
<b>Installing the MEAN component</b>	<b>4</b>
Installing NodeJS	4
Installing ExpressJS	5
Installing MongoDB	6
Installing AngularJS tools	6
The AngularJS debugger	8
<b>Understanding the project structure</b>	<b>9</b>
The file structure	9
Components	10
Testing	10
Tools	11
Package managers	11
Bower packages	12
<b>Previewing the final e-commerce app</b>	<b>12</b>
Homepage	13
Marketplace	14
Backoffice	15
<b>Understanding the requirements for e-commerce applications</b>	<b>17</b>
Minimum Viable Product for an e-commerce site	17
Defining the requirements	17
<b>Summary</b>	<b>19</b>



---

<b>Chapter 2: Building an Amazing Store Frontend with AngularJS</b>	<b>21</b>
<b>Understanding AngularJS and the client directory structure</b>	<b>22</b>
Client-side structure	23
Directives	24
Modules	25
Routing with AngularUI router	26
Controllers and scopes	28
Templates	29
<b>Laying out the e-commerce MVP</b>	<b>30</b>
Products	30
Factories and services	31
Creating the products factory	32
Creating the marketplace	33
Filters	34
<b>CRUD-ing products with AngularJS</b>	<b>35</b>
Services	35
Controllers	37
Routes	38
Templates	39
Partial forms	40
Product New	41
Product edit	42
The product view	42
The product list	43
Styling the main page	44
<b>Summary</b>	<b>46</b>
<b>Chapter 3: Building a Flexible Database with MongoDB</b>	<b>47</b>
<b>Understanding MongoDB</b>	<b>48</b>
MongoDB daemons and CLI	48
Mapping SQL knowledge to MongoDB	49
Basics concepts	49
Queries	49
Aggregators	51
<b>CRUDing with Mongoose</b>	<b>52</b>
Schemas	52
Create	54
Read	55
Update	56
Delete	56
<b>Exploring a few advanced features in Mongoose</b>	<b>57</b>
Instance methods	57
The static methods	57

---

Virtuals	58
Validations	58
Built-in validations	59
Custom validations	59
Middleware	60
<b>Reviewing models and server-side structure</b>	<b>61</b>
The server folder	61
Current Mongoose models	62
CommonJS Modules	62
The user model	64
<b>Summary</b>	<b>66</b>
<b>Chapter 4: Creating a RESTful API with NodeJS and ExpressJS</b>	<b>67</b>
<b>Getting started with REST</b>	<b>68</b>
<b>Scaffolding RESTful APIs</b>	<b>69</b>
<b>Bootstrapping ExpressJS</b>	<b>69</b>
<b>Understanding routes in ExpressJS</b>	<b>72</b>
<b>Testing, TDD, BDD, and NodeJS</b>	<b>74</b>
<b>Creating the product model</b>	<b>75</b>
Testing the products model	75
Product model implementation	76
<b>Implementing the Product API</b>	<b>77</b>
Testing the API	77
Index action tests	77
Show action tests	78
Creating action tests	78
Deleting action tests	78
Product controller	78
<b>Summary</b>	<b>79</b>
<b>Chapter 5: Wiring AngularJS with ExpressJS REST API</b>	<b>81</b>
<b>Implementing a RESTful product service</b>	<b>81</b>
Building the marketplace	83
<b>Wiring the product controller with new RESTful methods</b>	<b>83</b>
<b>Uploading product images</b>	<b>85</b>
Uploading files in Angular	85
Handling file upload on Node	89
Seeding products	91
<b>Testing RESTful APIs in AngularJS</b>	<b>92</b>
Unit testing	92
ngMock	92
Setting up testing	93
Understanding the Services tests	93

Testing all \$resource methods	94
Testing the Product Controller	94
<b>End-to-end testing</b>	<b>95</b>
Cleaning the database on each e2e run	96
<b>Summary</b>	<b>97</b>
<b>Chapter 6: Managing User Authentication and Authorization</b>	<b>99</b>
<b>Getting started with authentication strategies</b>	<b>100</b>
Session-based authentication	100
Token-based authentication – using JWT	101
OAuth authentication	103
<b>Understanding client-side authentication</b>	<b>104</b>
Authentication management	105
The signing up process	108
<b>Understanding server-side authentication</b>	<b>111</b>
Authentication with PassportJS	112
Initializing PassportJS	112
The user model	114
Authentication strategies and routes	116
Local authentication	117
End-to-end tests for local authentication	118
<b>Authenticating with Facebook, Google, and Twitter</b>	<b>118</b>
Facebook	119
Twitter	121
Google	122
<b>Summary</b>	<b>124</b>
<b>Chapter 7: Checking Out Products and Accepting Payment</b>	<b>125</b>
<b>Setting up the shopping cart</b>	<b>125</b>
Installing ngCart	126
Making use of ngCart directives	127
Add/remove to cart	127
The cart's summary	128
<b>The checkout page and Braintree integration</b>	<b>129</b>
<b>Setting up Braintree endpoint and authentication</b>	<b>132</b>
The API keys	132
Gateway	133
Controller	133
Router	135
<b>Creating an order</b>	<b>135</b>
Modifying the order model	136

---

Testing the order model	138
Using the sandbox account	140
<b>Summary</b>	<b>140</b>
<b>Chapter 8: Adding Search and Navigation</b>	<b>141</b>
<hr/>	
<b>Adding search to the navigation bar</b>	<b>141</b>
<b>Adding product categories</b>	<b>143</b>
Adding the sidebar	143
Improving product models and controllers	146
Catalog controller and routes	147
The catalog model	148
Seeding products and categories	150
<b>Implementing the search and navigation functionality</b>	<b>151</b>
Adding custom \$resource methods	152
Setting up routes and controllers	152
<b>Wrapping it up</b>	<b>153</b>
How navigation works on the client side	153
How search works on the client side	154
<b>Summary</b>	<b>155</b>
<b>Chapter 9: Deploying a Production-ready e-Commerce App</b>	<b>157</b>
<hr/>	
<b>Building for production</b>	<b>157</b>
Application environments	158
Optimizations for production environments	158
<b>Scaling web applications</b>	<b>159</b>
Scaling out vertically – one server	160
Scaling out horizontally – multiple servers	160
<b>Deploying the application to the cloud</b>	<b>162</b>
Platform as a Service	162
Heroku	162
Virtual Private Servers and Cloud Servers	163
Digital Ocean	163
<b>Deploying applications in a multi-server environment</b>	<b>165</b>
Setting up the app server – NodeJS application	166
Setting up web server – Nginx server	168
<b>Performing stress tests</b>	<b>169</b>
HTTP benchmarking tools	169
ApacheBench	169
Benchmarking Heroku deployment	169
Benchmarking VPS multi-server deployment	170

<b>Production architecture for scaling NodeJS</b>	<b>170</b>
Phase 0 – one server	170
Phase 1 – multiple application instances in one server	171
Phase 2 – multiple servers	172
Phase 3 – Micro-services	173
<b>Next steps on security</b>	<b>173</b>
<b>Summary</b>	<b>174</b>
<b>Chapter 10: Adding Your Own Features with High Quality</b>	<b>175</b>
<b>Planning a new feature</b>	<b>175</b>
Wire framing	176
Implementing the solution	177
The HTML page	177
<b>Testing the new feature</b>	<b>179</b>
AngularJS testing	179
<b>Features backlog</b>	<b>182</b>
<b>Deploying a new version of the app</b>	<b>182</b>
Zero-downtime deployments	183
Setting up the zero-downtime production server	184
Getting started with Capistrano	185
Installing Capistrano	185
Understanding Capistrano	185
Preparing the server	187
Setting up Capistrano variables	187
Capistrano tasks	188
Adding new tasks	189
Preparing Nginx	191
Load balancing	193
Static file server	193
WebSockets	193
<b>Summary</b>	<b>194</b>
<b>Index</b>	<b>195</b>

---

# Preface

E-commerce platforms are widely available these days. However, it is common to invest a significant amount of time in learning to use a specific tool and realize later that it does not fit your unique e-commerce needs. So, the greatest advantage of building your own application with an agile framework is that you can quickly meet your immediate and future needs with a system that you fully understand.

The MEAN stack (MongoDB, ExpressJS, AngularJS, and NodeJS) is a killer JavaScript full stack combination. It provides agile development without compromising on performance and scalability. It is ideal to build responsive applications with a large user base, such as e-commerce applications.

This book will teach you how to create your own e-commerce application using the MEAN stack. It will take you step by step through the process of learning and building parallelly. Using this guide, you will be able to show a product catalog, add products to shopping carts, and perform checkouts. It will cover product categorization, search, scalable deployment, server setups, and other features.

At the end of this book, you will have a complete e-commerce application that is fully tested, scalable, and alive. Additional topics on how to scale the application, server deployment, and security will also be discussed.

## What this book covers

*Chapter 1, Getting Started with the MEAN Stack*, is an introductory chapter presenting a list of the main features of an e-commerce application. It also covers the basics of each of the MEAN components – MongoDB, ExpressJS, AngularJS, and NodeJS – and explains how to install them.

*Chapter 2, Building an Amazing Store Frontend with AngularJS*, is a jumpstart on AngularJS. It goes over the file organization used throughout the project and the layout of the first version of the application.

*Chapter 3, Building a Flexible Database with MongoDB*, focuses on MongoDB. It covers how to set up models with Mongoose and CRUD operations and review our code base for the server side.

*Chapter 4, Creating a RESTful API with NodeJS and ExpressJS*, gets the reader familiarized with REST and APIs. It covers the basics of ExpressJS routing and implements the product API as an example.

*Chapter 5, Wiring AngularJS with ExpressJS REST API*, builds upon all the concepts learned in previous chapters and puts them to work together. It covers the basics of testing APIs and end-to-end testing.

*Chapter 6, Managing User Authentication and Authorization*, explains the difference between authorization and authentication. It implements a user login with an e-mail ID and password as well as social logins with Facebook, Twitter, and Google+.

*Chapter 7, Checking Out Products and Accepting Payment*, is a complete, hands-on chapter. It deals with the creation of shopping carts, orders, and API calls. It also takes a look at how to receive payments using PayPal.

*Chapter 8, Adding Search and Navigation*, is about improving the UI/UX by adding navigation menus, categories, and a search bar.

*Chapter 9, Deploying a Production-ready e-Commerce App*, is all about making the application suitable for a production environment. It deals with the setup of a server and scalable architectures. It also explains step by step how to deploy the server to cloud servers as well as multiserver environments.

*Chapter 10, Adding Your Own Features with High Quality*, is a closing chapter explaining how to continue extending and scaling the application. It also covers the process of automating deployments so that new features can be introduced to the application effortlessly and with zero downtime.

## What you need for this book

For this book, you need a basic understanding of JavaScript, HTML, and CSS. You will also need the following tools:

- A text editor or IDE (for example, Sublime Text, Cloud9, Eclipse, Notepad++, Vim, and so on)

- A command-line terminal
- NodeJS (<https://nodejs.org/>)
- MongoDB (<https://www.mongodb.org/>)
- Accounts on social network (for example, Facebook, Twitter, or Google+) and, optionally, accounts on PayPal.
- Access to a cloud server, either Heroku, Digital Ocean, or similar.

## Who this book is for

If you are a web developer with experience in JavaScript and wish to make a profit with an e-commerce application, then this book is for you. It will take you through the steps to understand and implement the main features of an e-commerce site. It will also build the foundations to add new features at will with high-quality trough testing and automated deployment. This book is meant for JavaScript developers looking to develop a flexible e-commerce site that they can fully understand with the technology they love.

## Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "Take a look into `package.json` and `bower.json`."

A block of code is set as follows:

```
angular.module('meanstackApp')
  .config(function ($stateProvider) {
    $stateProvider
      .state('main', {
        url: '/',
        templateUrl: 'app/main/main.html',
        controller: 'MainCtrl'
      });
  });
```



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


```
angular.module('meanstackApp')
  .controller('MainCtrl', function ($scope, $http, socket) {
    $scope.awesomeThings = [];


    $http.get('/api/things').success(function(awesomeThings) {
      $scope.awesomeThings = awesomeThings;
      socket.syncUpdates('thing', $scope.awesomeThings);
    });
  });
```

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

```
# Install the tools
$ npm install -g generator-angular-fullstack
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Furthermore, you need to go to **API & auth** | **APIs**, and enable **Google+ API**."

[  Warnings or important notes appear in a box like this. ]

[  Tips and tricks appear like this. ]

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files from your account at <http://www.packtpub.com> for all the Packt Publishing books you have purchased. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to <https://www.packtpub.com/books/content/support> and enter the name of the book in the search field. The required information will appear under the **Errata** section.

## Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

## **Questions**

If you have a problem with any aspect of this book, you can contact us at [questions@packtpub.com](mailto:questions@packtpub.com), and we will do our best to address the problem.

# 1

## Getting Started with the MEAN Stack

In order to build a powerful and interactive e-commerce application, business logic has to move closer to the users, also known as thick clients. JavaScript is well-suited for this job since it is the native language of the web. Any browser on any device can support it without any plugins. Furthermore, we can dramatically increase the interactivity and speed of a web application using JavaScript and HTML5. Instead of rendering the whole web page every time the user clicks something (as with the traditional server side languages), we can use asynchronous calls to quickly fetch data from the server and render only what's required.

The **MEAN** stack (**M**ongoDB, **E**xpressJS, **A**ngularJS, and **N**odeJS), as you might already know, is a JavaScript full stack web development framework. Using only one main programming language from the client side all the way back to the server and the database has many advantages, as we are going to discuss later.

Some companies like Walmart, Groupon, Netflix, and PayPal have moved from traditional *enterprise* web frameworks (like Java's Spring MVC and Ruby on Rails) to NodeJS and thick JavaScript clients. This improved not just their productivity but also their performance! They have reported an increase in the requests per second as well as a decrease in their development time.

I hope you are as excited as I am to build an e-commerce app using state-of-the-art technology like the MEAN stack with the help of this book. By the end of this book, you will not only know more about MEAN e-commerce apps but also about following the best practices. You will get familiar with concepts like Agile methodologies, sprint/iterations, continuous integration, **Test Driven Development (TDD)**, and production deployment.

We are going to cover some ground in this first chapter, and then deepen our knowledge on each of the MEAN stack components. The topics that will be covered in this chapter are as follows:

- Installing the MEAN components
- Understanding the project structure
- Previewing the final app built-in this book
- Understanding the requirements for e-commerce applications

## Introducing the MEAN stack

MEAN is more than an acronym; it offers the following advantages over its traditional alternatives such as **LAMP** (Linux, Apache, MySQL, and PHP) and other solution stacks:

- Asynchronous programming (evented non-blocking I/O) which is translated into high throughput and scalability
- One single programming language across the whole project which is translated into fast paced development and is easy to learn
- Vibrant community. NPM packages have grown faster than any other programming language community yet
- Excellent for JSON APIs, **Single-Page Applications (SPAs)**, and soft real-time applications

As for any other solution, there are some disadvantages as well, but they can be mitigated as follows:

- **CPU/thread-intensive apps:** Applications that have to deal with heavy CPU usage and low I/O are not very well suited for NodeJS. Some examples of such applications are video encoding and artificial intelligence, which are probably better handled in C/C++. NodeJS supports C++ add-ons, so seamless integrations are an option.

In general, this is how SPAs (and the MEAN stack) work: when a client request is received from the user's browser, it hits the ExpressJS web server first. ExpressJS runs on top of the NodeJS platform, and it connects to the MongoDB database as needed. Finally, the client's request is answered with an AngularJS application. From that point on, all subsequent requests are made behind the scenes through AJAX to the ExpressJS API. AngularJS takes care of rendering any new data instantly and avoiding unnecessary full-page refreshes.

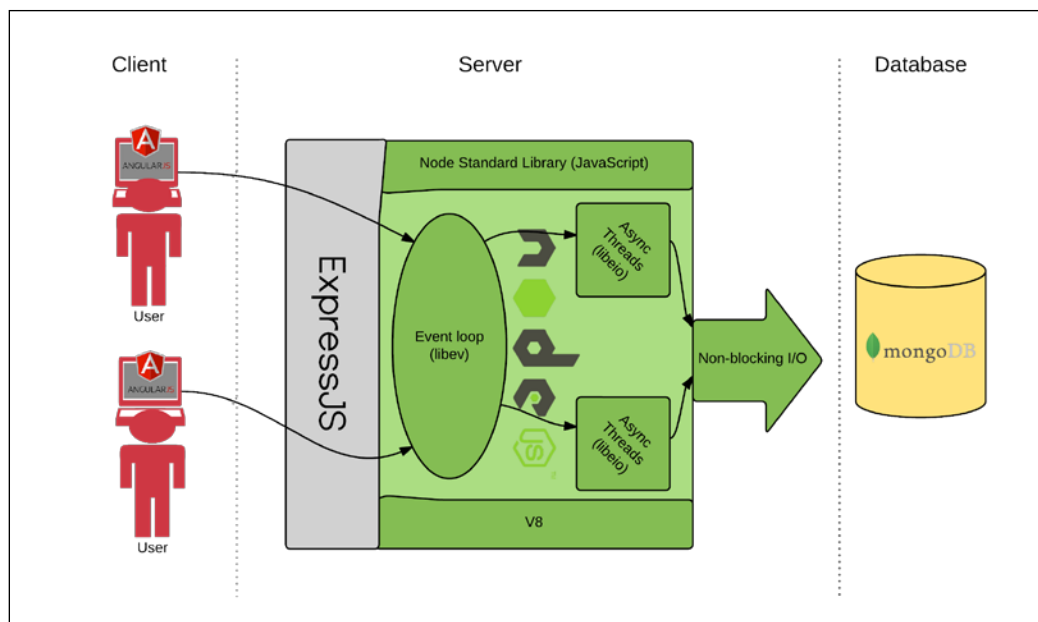


Figure 1: MEAN stack components

Now, let's go through each component individually.

## NodeJS

NodeJS is a platform that allows running of the JavaScript code outside of the browser. It's built on top of Google Chrome's V8 JavaScript runtime environment. Thanks to its non-blocking I/O and event-driven model, it is very fast and uses each CPU cycle optimally.

The JavaScript event-loop is sometimes confusing for developers who are new to JavaScript. In many popular languages (Java, Ruby, Python), each thread executes only one line of code at a time (everything else is blocked). However, JavaScript was conceived to be responsive at all times, so it makes heavy use of callbacks and the event-loop. The event-loop is an internal loop that executes all the code, and does not wait for anything in particular. The callbacks are called when a process is done. For instance, for blocking the I/O, a database query will make the thread sit and wait for the result. On the other hand, in JavaScript, it will continue executing the successive lines of code. When the database result is ready, it will be handled by the callback. Thus, non-blocking I/O.

NodeJS will be our server-side platform, and we will need it to run our web server, ExpressJS.

## ExpressJS

ExpressJS is a web framework for NodeJS. It is a very transparent webserver that not only allows us to build web applications but also to expose RESTful JSON APIs. Express is very modular, so we can process/modify any requests using something called middleware. There are middlewares to provide authentication, handling cookies, logging, error handling and other such functions.

## MongoDB

MongoDB is an open source document-oriented database, one of the most popular in the NoSQL database world. It favors a JSON-like document structure rather than the more traditional table-based structures found in relational databases. Its query language is very powerful and expressive. Later in this book, we are going to compare the equivalent SQL queries to MongoDB queries.

MongoDB stores all the users' data, products, orders, and anything that needs to be persistent in the e-commerce application.

## AngularJS


AngularJS is an open source JavaScript MVC framework (it might also be referred to as MV\* or MVW[Whatever]). This framework enhances the HTML markups with new *directives*, properties, and tags, and always keeps the views and models in sync. AngularJS facilitates the creation of data-driven and **Single-Page Applications (SPAs)**.

## Installing the MEAN component

All right. Enough theory, let's get our hands dirty and execute some command lines.

## Installing NodeJS

The node community has been very busy lately and has created more packages for it than any other existing platform. Not just packages but the core development too has been forked (for example, ioJS) and merged back to the main line. Since we are aiming for production, we are going with the current stable version v0.12. We are going to use **NVM (Node Version Manager)** to switch easily between versions when newer versions become stable.

[  If you are a Windows user, you can see the instruction on the NVM site at <https://github.com/creationix/nvm>. ]


For most \*nix users, this is how you install NVM:

```
$ curl https://raw.githubusercontent.com/creationix/nvm/v0.24.1/install.sh | bash
$ source ~/.nvm/nvm.sh
$ nvm install 0.12
$ nvm use 0.12
```

The NodeJS installation comes with the **NPM (Node Package Manager)**. You can verify if node and npm were installed correctly by checking the versions, as follows:

```
$ node -v
# v0.12.7

$ npm -v
# 2.11.3
```

[  In case you have multiple versions of NodeJS, you might want to set v0.12 as the default. So when you open a new terminal and it has the right version, use the `nvm alias default 0.12` command to do so. ]

## Installing ExpressJS

You can download Express using npm by typing the following in your terminal:

```
npm install -g express@4.13.3
```

Note that we are downloading a specific version using `@4.13.3`; the `-g` argument means that we are installing the package globally.



## Installing MongoDB

If you are using a Mac, I'd recommend installing `brew` first:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Then you can install MongoDB:

```
brew update && brew install mongodb 3.0.2
```

Ubuntu:

```
sudo apt-get -y install mongodb=3.0.2
```



For other operating systems, follow the instructions given at the official site at <https://www.mongodb.org/downloads>.

## Installing AngularJS tools

You don't need to install the AngularJS tools like the other components so far.

We can just download the Angular JavaScript files from its website at <https://angularjs.org> and source it in our HTML. The other option is to source it directly from a **Content Delivery Network (CDN)**. However, if you notice, every time you start a new web app, you have to execute the same steps over and over again, for example, type HTML5 doctype, copy initial code, create the directory structure, and so on. All this boilerplate can be automated using a tool called Yeoman. There are many generators in the Yeoman repository. They can scaffold most of the JavaScript frameworks out there: AngularJS, BackboneJS, and the like; if it doesn't exist, you can also create your own. The best ones that suit our needs are `angular-fullstack` and `meanjs`. We are going to use the first one since it provides out-of-the-box functionality that is closer to our needs. Let's go ahead and install it:

```
# Install all the tools
```

```
$ npm install -g generator-angular-fullstack@3.0.0-rc4
```

```
# Make a new directory 'meanshop'
```

```
$ mkdir meanshop && cd $_
```

```
# Execute the AngularJS scaffold
```

```
$ yo angular-fullstack meanshop
```



You will see a lot of things going on when you run the commands. But don't worry, we are going to explain all the magic in the next few chapters. For now, play along and get everything installed.

The command line has some kind of wizard that asks you about your preferences. The following are the ones that we are going to use throughout the examples in this book:

#### # Client

```
? What would you like to write scripts with? JavaScript + Babel
? What would you like to write markup with? HTML
? What would you like to write stylesheets with? Sass
? What Angular router would you like to use? uiRouter
? Would you like to include Bootstrap? Yes
? Would you like to include UI Bootstrap? Yes
```

#### # Server

```
? What would you like to use for data modeling? Mongoose (MongoDB)
? Would you scaffold out an authentication boilerplate? Yes
? Would you like to include additional OAuth strategies? Google,
Facebook, Twitter
? Would you like to use socket.io? Yes
```

#### # Project

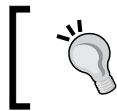
```
? What would you like to write tests with? Mocha + Chai + Sinon
? What would you like to write Chai assertions with? Expect
```

It takes a while, because it installs a number of packages for you. Take a look into `package.json` and `bower.json`. Bower is another package manager like `npm` but for the frontend libraries. After the installation is complete, you can run the example app with these commands:

```
# Build
$ grunt

# Preview the app in development mode
$ grunt serve

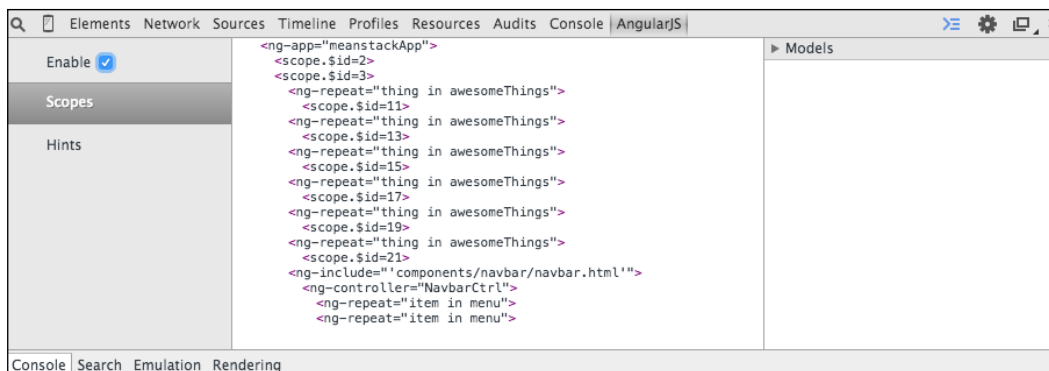
# Preview the app in production mode
$ grunt serve:dist
```



You can preview the scaffolded app at: `http://localhost:9000`.  
If you get errors, remember that you need to have `mongod` running.

## The AngularJS debugger

Most of the modern browsers provide debugging tools for JavaScript. There's a Google Chrome plugin called **Batarang**, which is a web inspector extension. I'd recommend getting it installed since it adds more context and useful information and hints:



Example of the output of Batarang AngularJS extension for Chrome's web inspector.



You can find more information about Batarang at <https://github.com/angular/angularjs-batarang>.

## Understanding the project structure

Applications built with the `angular-fullstack` generator have many files and directories. Some code goes into the client, some executes in the backend, and another portion is just needed for development, such as the tests suites. It's important to understand the layout to keep the code organized.

Yeoman generators are time savers. They are created and maintained by the community following the current best practices. It creates many directories and a lot of boilerplate code to get you started. It might be a bit overwhelming at first to see the number of (possibly) unknown files there. Do not panic, we are going to cover them here and in the next few chapters.

Review the directory structure that was created. There are three main directories: `client`, `e2e`, and `server`:

- The `client` folder contains the AngularJS files and assets
- The `server` directory contains the NodeJS files, which handle ExpressJS and MongoDB
- Finally, the `e2e` files contain the AngularJS end-to-end tests

## The file structure

The following is an overview of the file structure of this project:

```
meanshop
├── client
│   ├── app          - App specific components
│   ├── assets       - Custom assets: fonts, images, etc...
│   └── components   - Non-app specific/reusable components
├── e2e              - Protractor end to end tests
└── server
    ├── api          - Apps server API
    ├── auth         - Authentication handlers
    ├── components   - App-wide/reusable components
    ├── config       - App configuration
    │   ├── local.env.js - Environment variables
    │   └── environment - Node environment configuration
    └── views        - Server rendered views
```

Zooming into `clients/app`, we will find that each folder has the name of the component (main page, products page), and that inside each folder are all the files related to that component. For instance, if we look inside `main`, we will find the AngularJS files, CSS (scss), and HTML:

```
meanshop/client/app/main
├─ main.js                - Routes
├─ main.controller.js     - Controller
├─ main.controller.spec.js - Test
├─ main.html              - View
└─ main.scss              - Styles
```

Similarly, for our back-end, we have folders named after the components with all the related files inside. We will find NodeJS files, ExpressJS routes, SocketIO events, and mocha tests:

```
meanshop/server/api/thing
├─ index.js               - Routes
├─ thing.controller.js    - Controller
├─ thing.model.js         - Database model
├─ thing.socket.js        - Socket events
└─ thing.spec.js          - Test
```

## Components

There are a number of tools used in this project that you might be already familiar with. If that's not the case, read the brief description given, and when needed, we will describe it more thoroughly.

## Testing

AngularJS comes with a default test runner called **Karma**, and we are going to leverage its default choices:

- **Karma**: This is the JavaScript unit test runner.
- **Jasmine**: This is the BDD framework for testing the JavaScript code, which is executed with Karma.
- **Protractor**: This is used for end-to-end tests with AngularJS. This is the highest level of testing, which runs in the browser and simulates user interactions with the app.

## Tools

The following are some tools/libraries that we are going to use for increasing our productivity:

- **GruntJS**: This tool serves to automate repetitive tasks such as CSS/JS minification, compilation, unit testing, and JS linting.
- **Yeoman (yo)**: This is a CLI tool for scaffolding web projects. It automates the creation of directories and files through generators, and also provides command lines for common tasks.
- **Travis CI**: This is a **Continuous Integration (CI)** tool that runs your tests suite every time you commit to the repository.
- **EditorConfig**: This is an IDE plugin, which loads the configuration from a file, `.editorconfig`. For example, you can set `indent_size = 2`, indents with spaces or tabs, and so on. It's a time saver and maintains consistency across multiple IDEs/teams.
- **SocketIO**: This is a library that enables real-time bidirectional communication between the server and the client.
- **Bootstrap**: This is a frontend framework for web development. We are going to use it for building the theme throughout this project.
- **AngularJS full-stack**: This is the generator for Yeoman that will provide useful command lines to quickly generate server/client code and deploy to Heroku or OpenShift.
- **BabelJS**: This is the `js-to-js` compiler that allows the use of features from the next generation JavaScript (*ECMAScript 6*) instantly, without waiting for browser support.
- **Git**: This is a distributed code versioning control system.

## Package managers

AngularJS comes with package managers for third-party backend and frontend modules:

- **NPM**: This is the default package manager for NodeJS.
- **Bower**: This is the frontend package manager that can be used to handle versions and dependencies of the libraries and assets used in a web project. The file `bower.json` contains the packages and versions to install, and the file `.bowerrc` contains the path for the location where those packages need to be installed. The default directory is `./bower_components`.

## Bower packages

If you have followed the exact steps for scaffolding our app, you will have the following frontend components installed:

- angular
- angular-cookies
- angular-mocks
- angular-resource
- angular-sanitize
- angular-scenario
- angular-ui-router
- angular-socket-io
- angular-bootstrap
- bootstrap
- es5-shim
- font-awesome
- json3
- jquery
- lodash

In the next chapter, we will dive deeper into AngularJS and our file structure. The second part of this chapter is about the functionality of our final app.

## Previewing the final e-commerce app

Let's take a break from the terminal. In any project, before starting coding, we need to spend some time planning and visualizing what we are aiming for. That's exactly what we are going to do: draw some wireframes that walk us through the app. Our e-commerce app—**MEANshop**—will have three main sections:

- Homepage
- Marketplace
- Backoffice

# Homepage

The homepage will contain the featured products, navigation, menus, and some basic information, as you can see in the following image:

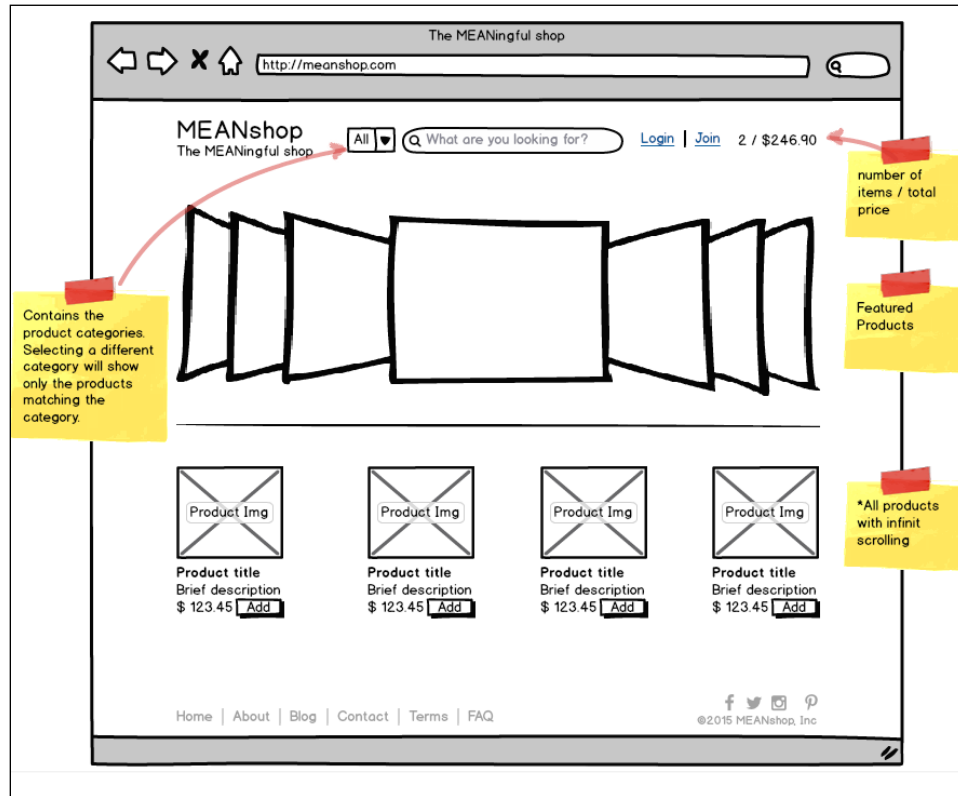


Figure 2: Wireframe of the homepage



## Marketplace

This section will show all the products, categories, and search results:

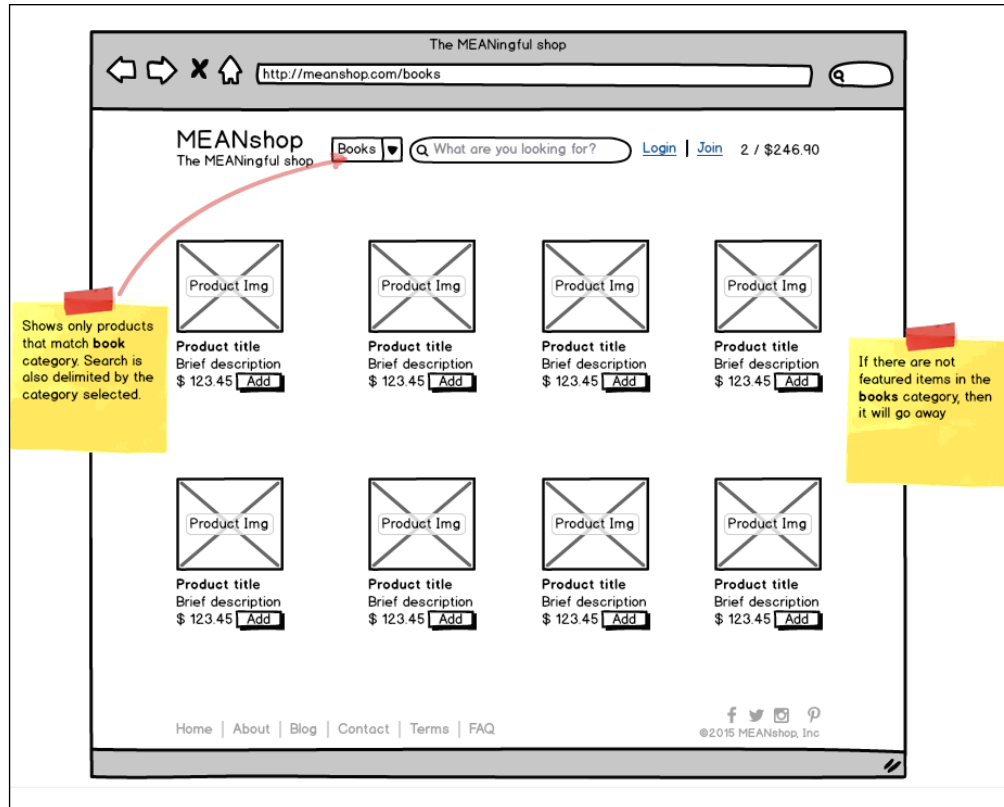


Figure 3: Wireframe of the products page

## Backoffice

You need to be a registered user for accessing the back office section.

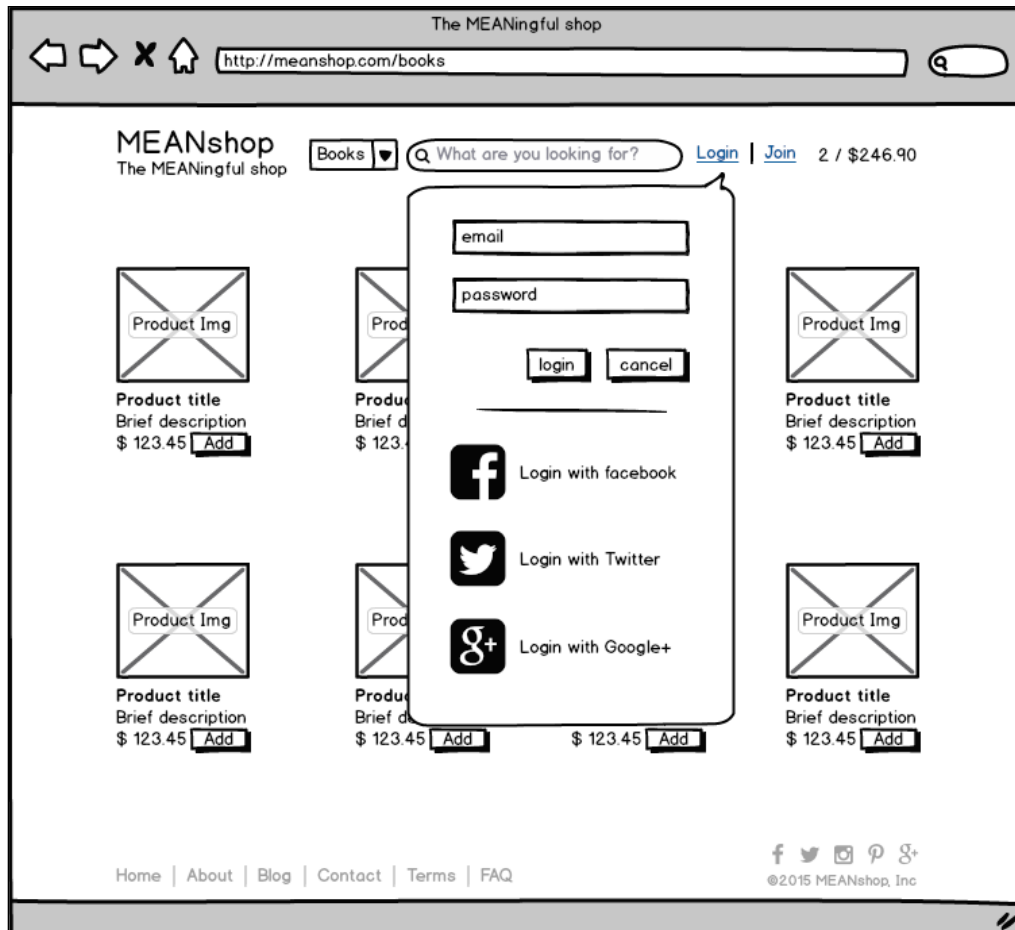


Figure 4: Wireframe of the login page

After you log in, the app will present you with different options depending on your role. If you are the seller, you can create new products, as shown in the following image:

The wireframe shows a web browser window titled "The MEANingful shop" with the URL "http://meanshop.com/products/new". The page header includes the "MEANshop" logo, a "Books" category dropdown, a search bar with the placeholder "What are you looking for?", and a user greeting "Hello Adrian" with a cart icon showing "0 / \$0.00". The main content area is for creating a new product. It features a large image placeholder (a square with an 'X') and three smaller image placeholders below it. To the right of the image placeholders are input fields for "Title", "Brief description", and "Extended Product description". Below these are input fields for "price" and "stock". At the bottom of this section are three buttons: "Upload", "Create", and "Cancel". Below the "Create" button is an "Options" section with three rows, each containing a "key" input field and a "values" input field. The footer includes a navigation menu with links for "Home", "About", "Blog", "Contact", "Terms", and "FAQ", and social media icons for Facebook, Twitter, Instagram, and Pinterest, along with the copyright notice "©2015 MEANshop, Inc".

Figure 5: Wireframe of the Product creation page

If you are an admin, you can do everything that a seller does (create products), and you can manage all the users and delete/edit products.

## Understanding the requirements for e-commerce applications

There's no better way to learn new concepts and technologies than developing something useful with it. This is why we are building a realtime e-commerce application from scratch. However, there are many kinds of e-commerce apps. In the next section, we are going to delimit what we are going to do.

### Minimum Viable Product for an e-commerce site

Even the largest applications that we see today started small and built their way up. The **Minimum Viable Product (MVP)** is the strict minimum that an application needs to work. In the e-commerce example, it will be the following:


- Add products with their title, price, description, photo, and quantity
- Guest checkout page for products
- One payment integration (for example, PayPal)

This is the minimum requirement for getting an e-commerce site working. We are going to start with these, but by no means will we stop there. We will keep adding features as we go, and build a framework that will allow us to extend the functionality along with high quality.

### Defining the requirements

We are going to capture our requirements for the e-commerce application with user stories. A user story is a brief description of a feature told from the perspective of a user, where he expresses his desire and benefit in the following format:


*As a <role>, I want <desire> [so that <benefit>]*

[  User stories and many other concepts were introduced with the Agile Manifesto. Learn more about this concept at [https://en.wikipedia.org/wiki/Agile\\_software\\_development](https://en.wikipedia.org/wiki/Agile_software_development). ]

The following are the features, captured as user stories, that we are planning to develop through this book:

- As a seller, I want to **create products**.
- As a user, I want to **see all the published products and their details** when I click on them.
- As a user, I want to **search for a product** so that I can find what I'm looking for quickly.
- As a user, I want to have a **category navigation menu** so that I can narrow down the search results.
- As a user, I want to have **realtime information** so that I can know immediately if a product just got sold out or became available.
- As a user, I want to **check out products as a guest user** so that I can quickly purchase an item without registering.
- As a user, I want to **create an account** so that I can save my shipping addresses, see my purchase history, and sell products.
- As an admin, I want to **manage user roles** so that I can create new admins, sellers, and remove seller permissions.
- As an admin, I want to **manage all the products** so that I can ban them if they are not appropriate.
- As an admin, I want to see a **summary of the activities** and order statuses.

All these stories might seem verbose, but they are useful for capturing the requirements in a consistent way. They are also handy for developing test cases.

[  Learn more about user stories at [https://en.wikipedia.org/wiki/User\\_story](https://en.wikipedia.org/wiki/User_story). ]

The technical requirements about deployment, scalability, and performance will be discussed in the final chapters.

## Summary

In this chapter, we discussed the reasons for using the MEAN stack to build our e-commerce application, and got it installed. This is not just some kind of trend which some companies are migrating to. It gives a tremendous performance boost to the apps, and eases the learning curve using one main language for both, the frontend and the backend. We also described the file structure that we are going to use for organizing the code. Finally, we explored the features that the final app will have, and the way it's going to look and behave. In the next series of chapters, we are going to work with each piece of the MEAN stack individually. Later, we will start integrating all the components and make them work together. The next chapter will cover the most visible part: building the marketplace with AngularJS.



# 2

## Building an Amazing Store Frontend with AngularJS

The tagline for AngularJS is: *HTML enhanced for web apps!*, and it does exactly that. It extends the HTML code with 'directives' like attributes, new tags, classes, and comments. These directives bind the HTML elements to AngularJS code and change the default behavior of the element. AngularJS differs from unobtrusive frameworks such as jQuery and BackboneJS in that the bindings are through HTML extensions (directives) rather than ID and CSS selectors. This approach alleviates the code complexity and readability that comes with the more traditional JS frameworks. The AngularJS framework first appeared in 2009 and quickly became one of the most popular JavaScript frameworks. Finally, AngularJS has many built-in features that make it an excellent candidate for developing **Single Page Applications (SPA)** and **Data-Driven Applications**.

A couple of decades ago, JavaScript was mainly used for form validations, some animations, and visual effects. Back then, most of the JS testing was done manually. The JS code was tightly coupled to the DOM, which made it hard to test. Over the years, in order to minimize the response time and network delays, more of the business logic started migrating from the backend to the client. JS needed more than ever to be *testable*. AngularJS excels in testability. Tests and test runners were created for it since its architectural conception. The Karma test runner and the end-to-end tests with Protractor are good examples of this.

AngularJS is a very capable **Model-View-Controller (MVC)** framework. MVC is a UI architectural pattern which separates the view from the data models and binds them through controllers. Angular 1.x implements a two-way data binding that always keeps the view (**HTML**) and models (**JavaScript**) in sync. Thus, AngularJS models are the single source of truth. Another nice feature is that AngularJS provides XHR services, which make it very easy to integrate with the RESTful APIs.



This chapter will focus only on AngularJS. In later chapters, we will show you the ways to connect AngularJS to the rest of the MEAN components.

Without further ado, let's dive in and discuss the following topics in this chapter:

- Understanding AngularJS and the client directory structure
- Laying out the e-commerce site
- CRUD-ing products

## Understanding AngularJS and the client directory structure

This section will help you get started with AngularJS. We are going to focus on the client part of our project and the AngularJS concepts will be explained as we walk through our app.

Let's first get familiarized with the functionality of our application out-of-the-box:

- Go to the project root, and run `grunt serve` in the command line (remember, you need `mongod` running as well).
- Interact with the app for a few minutes (`http://localhost:9000`). Open another window with the application. Add a new item and notice that this too gets updated in all the browser windows in real-time. Pretty cool, huh? We are going to use that feature (SocketIO) to make our e-commerce site real-time as well.

We are going to build a single page application (SPA), so we only need a single HTML file: `index.html`. AngularJS will take control of it and manage the transitions and the different states in such a way that the user will feel like he/she is navigating through different pages. This HTML file contains all the necessary boilerplate to render well in most browsers and mobile devices.

## Client-side structure

We have three main directories in our meanshop project: `client`, `server`, and `e2e`. In this chapter we are going to focus solely on the `client` folder. It contains our AngularJS code and assets such as images and fonts. This is an overview of the `client` directory:

`client`

```
|— app          - All of our app specific components go in here
|— assets       - Custom assets: fonts, images, and so on
└— components  - Reusable components, non-specific to our app
```

Mostly, we are going to be using the `app` folder to write our AngularJS code. Let's take a look at an example to see how each of our components will be organized:

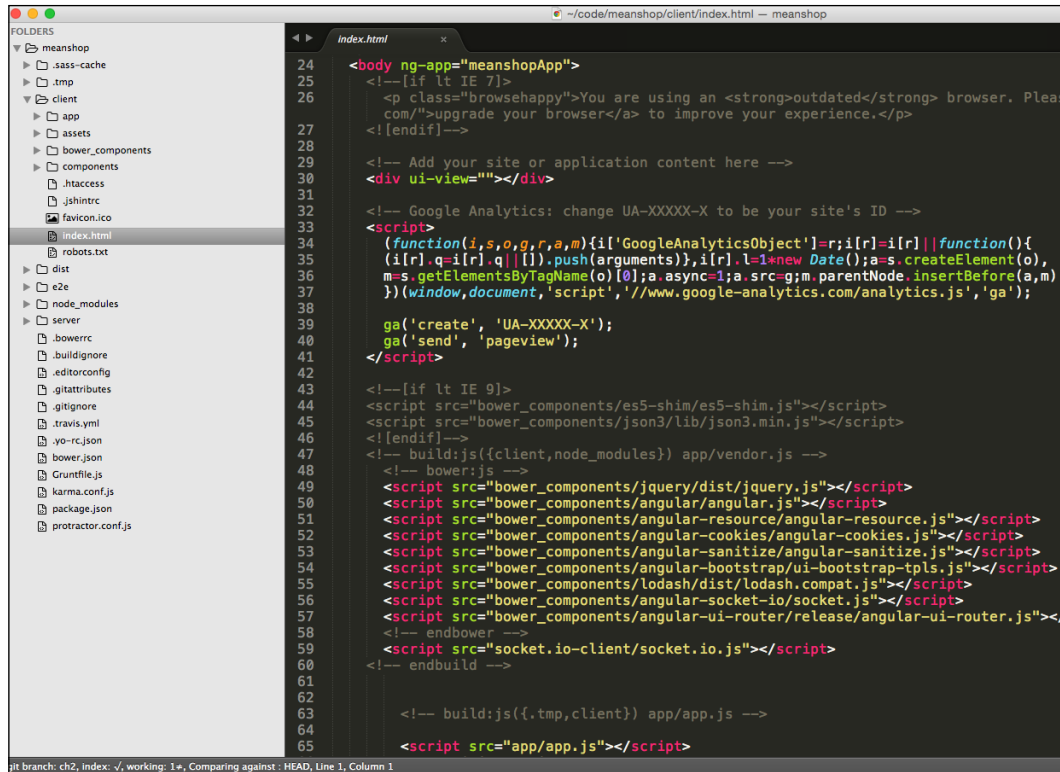
`meanshop/client/app/main`

```
|— main.js           - Routes
|— main.controller.js - Controller
|— main.controller.spec.js - Test
|— main.html         - View
└— main.scss         - Styles
```

As you can see, we are going to have routes, controllers, views and styles properly named in each app subfolder.

Now we are going to examine the code in the order of execution.

Open the `meanshop` folder and inside that, open the `client` folder. This is where we are going to set all the client code such as AngularJS files, HTML, CSS, and images. Open `index.html`. The following screenshot displays the project file structure and the AngularJS directives in the `index.html` file:



## Directives

Directives are HTML extensions in the form of attributes, tags, CSS classes, and even HTML comments. Directives are a key component in AngularJS, and the way in which it extends the HTML functionality. We can also define our own directives, as we are going to see in further chapters.

Take a look at `index.html`. The first thing to notice are all the non-standard HTML attributes. These are the directive attributes. In the body tag, we have `ng-app` defining the name of the module.

```
<!-- client/index.html -->
<body ng-app="meanshopApp">
```

The `ng-app` directive is necessary for bootstrapping AngularJS automatically. It defines the root module of the application:



Directive attributes are not part of the HTML standard; they are extensions to AngularJS. So, they will fail when the HTML validators are run against them. On the other hand, HTML5 does allow custom attribute names using the prefix `data-*`. All that we need to do to make the AngularJS directive's HTML valid is prefixing it with `data`. For instance: `data-ng-app=""` and `data-ui-view=""`.

There's another directive to be aware of called `ui-view`:

```
<!-- client/index.html -->
<div ui-view=""></div>
```

This `div` element is where a module called **ui-router** pushes the HTML code. More on modules and routing can be learnt in the following sections.

Moving back to the `index.html` file, notice some JS files being referenced at the bottom of the screen: `Angular-resource`, `angular-ui-router` to mention a few, as well as third party JS libraries, such as `jQuery`, `SocketIO`, `AngularJS`, and so on. Below that, we can see our application files starting with `app/app.js`. Let's now get into the `client/app` folder. The `app.js` file is the entry point of the application.

## Modules

Modules are the preferred way for organizing code in AngularJS. Even though we could get away without them, it makes the code much easier to maintain. AngularJS provides a global variable that contains many functions, and one of them is `module`. We are going to use `module` extensively through our application to set and get modules and to manage dependencies. The `angular.module` can work both as a getter and a setter. The getter is just the name of the module without any other parameter. It will load that module if it exists:

```
angular.module('meanshopApp');
```

On the other hand, the setter has two parameters: one to define the name of the module as defined in the `ng-app` directive, and the other to load the dependencies if any. In case there are no dependencies, an empty array must be passed.

In the following code, `angular.module` is being used as a setter. It creates the module and has a second parameter for dependencies:

```
/* client/app/app.js */

angular.module('meanshopApp', [
  'ngCookies',
  'ngResource',
  'ngSanitize',
  'btford.socket-io',
  'ui.router',
  'ui.bootstrap'
]);
```

We are creating a new module called `meanshopApp` in the preceding code. The second argument is an array of third-party AngularJS modules such as `ui.router`, `cookies` (`ngCookies`), and so on. Furthermore, notice that we have chained the methods to the module right after creating it.

When AngularJS sees `ng-app="meanshopApp"` in `index.html`, it knows it has to execute the module with the matching name and make all the listed dependencies available.

The `angular.module` getter has a number of other methods that can be chained to it. For instance, in the `app.js`, we can see the `config`, `factory`, and `run` methods.

- **Config:** This executes at the time of loading of a module.
- **Factory:** This returns an object or a function closure
- **Run:** This executes when all the modules are done loading

At a higher level, we use the `config` function to bootstrap the routes, which we are going to explain next.

## Routing with AngularUI router

Routing allows the user to have a URL that reflects the current state of the application. **Single Page Applications (SPAs)** could have all the different pages with just one unchangeable URL. Conversely, the user will not have the functionality to go back in history or to access a page/state by typing the URL directly. To fix that, we are going to use a module called **ui-router**, which is going to pair URLs and states with HTML views and controllers.

Back in `index.html`, remember the `ui-view` directive; this is where `ui-router` pushes the content of the page. Furthermore, besides loading `app/app.js`, there are other files being loaded such as `main.js`. Let's take a look at that one.

```
/* client/app/main/main.js */

angular.module('meanshopApp')
  .config(function ($stateProvider) {
    $stateProvider
      .state('main', {
        url: '/',
        templateUrl: 'app/main/main.html',
        controller: 'MainCtrl'
      });
  });
```

The AngularJS UI router allows us to define views and states. The way we wire up routes, controllers, and states to HTML templates is through `$stateProvider`. `Main.js` sets up the root URL (`/`) to a template (`main.html`) and a controller (`MainCtrl`). A similar pattern is found in the following files:

- `client/app/admin/admin.js`
- `client/app/account/account.js`

We can list multiple routes at once, like in `account.js`. Now, let's talk more about the controllers that these routes are using.



### Single Page Applications (SPA), Routing, and SEO

In an SPA, every time an internal link is clicked, it does not pull it from the server. Instead, it renders the predefined HTML template and does API calls in case it needs some data. This is a performance boost and a smarter utilization of the bandwidth. However, there is a caveat: since a single URL could have a lot of different pages and states, the history of the browser cannot detect it. To fix that, SPA adds dynamic routes through JS using hash like this: `/#products` or `/#/orders`. In terms of SEO, some web crawlers still don't render AJAX pages. They need the hash-bang URL convention to recognize AJAX pages, for example, `/#!products` or `/#!/orders`. Modern browsers have HTML5 `history.pushstate`, which allows us to use a URL without the hash or hash-bang, and still make it work like the conventional pages. For more info, refer to <https://developers.google.com/webmasters/ajax-crawling/docs/learn-more>.

## Controllers and scopes

As in any MVC framework, a controller interacts with Views and Modules. Controllers are the ones that are responsible for loading the data and representing it in the HTML templates (Views).

Going back to the `main.js` file, we see that the root URL (`/`) is managed by `MainCtrl`. Let's open `main.controller.js`:

```
/* meanshop/client/app/main/main.controller.js */

angular.module('meanshopApp')
  .controller('MainCtrl', function ($scope, $http, socket) {
    $scope.awesomeThings = [];

    $http.get('/api/things').success(function(awesomeThings) {
      $scope.awesomeThings = awesomeThings;
      socket.syncUpdates('thing', $scope.awesomeThings);
    });

    $scope.addThing = function() {
      if($scope.newThing === '') {
        return;
      }
      $http.post('/api/things', { name: $scope.newThing });
      $scope.newThing = '';
    };

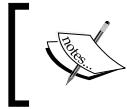
    $scope.deleteThing = function(thing) {
      $http.delete('/api/things/' + thing._id);
    };

    $scope.$on('$destroy', function () {
      socket.unsyncUpdates('thing');
    });
  });
```

In AngularJS, a controller is the glue between the models/services and the views. The data is retrieved using `$http`, and is made accessible to the view through `$scope`.

What is `$http`? It is a core AngularJS service that facilitates the communication with remote servers. Behind the scenes, it handles the `XMLHttpRequest` or `jsonp` JSONP requests. We are going to use it in the next few chapters to communicate with the ExpressJS server.

What is `$scope`? It is an object that glues the controller to the views. It provides two-way data binding. Every time we update a variable in `$scope`, it automatically rerenders the HTML view. Similarly, every time a change is made in the HTML, its value representation gets updated. In the preceding code, notice that we cannot just set variables like `awesomeThings`, but we can also make functions available and listen for events.



All AngularJS core identifiers and built-in services are distinguished by the dollar sign prefix with the name. For example, `$scope`, `$http`.

Next we are going to see how we can use these scoped variables and functions in the templates.

## Templates

Templates are the HTML files mixed with AngularJS enhancements. They execute the JS expressions, reference variables, and functions in the `$scope`.

Let's see an example. Open `main.html`:

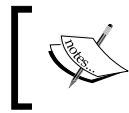
```
<!-- client/main/main.html *excerpt -->

<div class="container">
  <div class="row">
    <div class="col-lg-12">
      <h1 class="page-header">Features:</h1>
      <ul class="nav nav-tabs nav-stacked col-md-4 col-lg-4 col-sm-6"
        ng-repeat="thing in awesomeThings">
        <li><a href="#" tooltip="{{thing.info}}">{{thing.name}}<button
          type="button" class="close" ng-click="deleteThing(thing)">
            &times;</button></a></li>
      </ul>
    </div>
  </div>
</div>
```

In this small fragment of the `main.html` file, we can see new directives such as:

- `ng-click`: This is a directive that allows specifying a custom behavior when it is clicked. For example, in the code snippet, when the button is clicked, it executes `'deleteThing()'`, which is a function defined in `MainCtrl`'s `$scope`.
- `ng-repeat`: This is an iterator for a collection. For instance, `'awesomeThings'` is an array set in the controller, and `thing` represents each of the elements in turn.





There are many other directives available for use in templates. Take a look at the entire list at <https://docs.angularjs.org/api/ng/directive>.

It has been a great walk-through so far. Now, it's coding time!

## Laying out the e-commerce MVP

Now we are going to take leave of the boilerplate code and start writing our own. In this first section, we are going to build the marketplace.

### Products

So, let's create the page that will hold all the products. We will use the Yeoman generator again to automate the tasks of creating the new files, and focus on customizing it to our needs. Go to the terminal and execute the following command:

```
$ yo angular-fullstack:route products
```

```
? Where would you like to create this route? client/app/
```

```
? What will the url of your route be? /products
```

```
create client/app/products/products.js
```

```
create client/app/products/products.controller.js
```

```
create client/app/products/products.controller.spec.js
```

```
create client/app/products/products.html
```

```
create client/app/products/products.scss
```

Go with the defaults. This will create the route, controller, template, CSS, and test file with this single command. Let's run `grunt serve`, and check that the route is working as `http://localhost:9000/products`.

It will also be nice to add it to the main menu. Let's do that inside navbar:

```
/* client/components/navbar/navbar.controller.js *excerpt */
$scope.menu = [{
  'title': 'Home',
  'state': 'main'
}, {
  'title': 'Products',
  'state': 'products'
}];
```

Get familiar with the `navbar.html` file. Notice some directives like `ng-click`, `ng-href`, `ng-show`, `ng-hide`, `ng-class`, and specially `ng-repeat`:

```
<!-- client/components/navbar/navbar.html *excerpt -->

<li ng-repeat="item in menu" ui-sref-active="active">
  <a ui-sref="{{item.state}}">{{item.title}}</a>
</li>
```

Let's take a look into the functions of these directives:

- The `Ng-repeat` directive loops through the `$scope.menu` collection, where we previously added the product link.
- `Ng-click` is the AngularJS version of click. It intercepts the click events and call functions defined in the controller, or executes an expression.
- `Ng-show`/`Ng-hide` hides/shows an element if the expression is equivalent to `true`.
- `Ng-class` sets a class to an element if the expression is `true`.
- `Ui-sref` generates the link based on the route state name. You can see the states names associated to a route in `client/products/products.js` and `client/main/main.js`.
- `Ui-sref-active` adds a class to an element when the `ui-sref` is active and removes the class when it is inactive.

## Factories and services

The AngularJS services are singleton objects or functions that are bound to controllers or other components using the **Dependency Injection (DI)**. Services are great for the separation of concerns. The controller's job is to bind the data with the view using `$scope`. On the other hand, services handle the business logic to fetch that data. Bear in mind that we can call these services not just inside the controllers, but from anywhere: directives, filters, and so on.

Using a service or a factory is more about code style. The following is the main difference between the two:

- **Services:** This gets the instance of the function (`this` keyword is available). They return a constructor function, so we need to use the `new` operator.
- **Factories:** This gets the value returned by invoking the function. It allows to create closures.

## Creating the products factory

We are going to use factories just for the sake of simplicity. Let's create a factory for products with:

```
$ yo angular-fullstack:factory products
```

```
? Where would you like to create this factory? client/app/  
  create client/app/products/products.service.js  
  create client/app/products/products.service.spec.js
```

Since we do not have the web server ready yet, we are going to use an array to hold the data. Replace the content in `products.service.js` to look like the following code:

```
/* client/app/products/products.service.js */  
  
angular.module('meanshopApp')  
  .factory('Product', function () {  
    return [  
      {_id: 1, title: 'Product 1', price: 123.45, quantity: 10,  
description: 'Lorem ipsum dolor sit amet'},  
      {_id: 2, title: 'Product 2', price: 123.45, quantity: 10,  
description: 'Lorem ipsum dolor sit amet'},  
      {_id: 3, title: 'Product 3', price: 123.45, quantity: 10,  
description: 'Lorem ipsum dolor sit amet'},  
      {_id: 4, title: 'Product 4', price: 123.45, quantity: 10,  
description: 'Lorem ipsum dolor sit amet'},  
      {_id: 5, title: 'Product 5', price: 123.45, quantity: 10,  
description: 'Lorem ipsum dolor sit amet'}  
    ];  
  });
```

Notice that we changed the name from `products` to `Product`. Later we are going to do the same in `products.service.spec.js`.

This factory is going to simulate the product data in a web server for now. It is time to inject the product factory into the controller:

```
// client/app/products/products.controller.js  
  
angular.module('meanstackApp')  
  .controller('ProductsCtrl', function ($scope, Products) {  
    $scope.products = Products;  
  });
```

Notice that we just need to match the name of our factory (products) and it will become available in our controller. Moving on, we created a new variable `$scope.products`. This way, `products` will become available in the view as well.


## Creating the marketplace

To recap, we get our products' data from our `Products` factory. This one is injected into the `ProductsCtrl` controller and made available through `$scope` (`$scope.products`). The last remaining part is to use the `products` variable exposed in `$scope` in our template, and do some styling on `products.html`:

```
<!-- clients/app/products/products.html -->

<div ng-include="'components/navbar/navbar.html'"></div>

<div class="container">
  <div class="row">
    <div class="col-sm-6 col-md-4" ng-show="products.length < 1">
      No products to show.</div>
    <div class="col-sm-6 col-md-4" ng-repeat="product in products">
      <div class="thumbnail">
        
        <div class="caption">
          <h3>{{product.title}}</h3>
          <p>{{product.description | limitTo: 100}} ...</p>
          <p>{{product.price | currency }}</p>
          <p>
            <a href="#" class="btn btn-primary" role="button">Buy</a>
            <a ui-sref="viewProduct({id: product._id})"
              class="btn btn-default" role="button">Details</a>
          </p>
        </div>
      </div>
    </div>
  </div>
</div>
```

[  Run `grunt serve` and go to: `http://localhost:9000/products`. ]

In each of the highlighted lines, we can see new directives:

- `ng-include`: This will insert the HTML code from the indicated path.
- `ng-repeat`: As its name implies, it repeats wherever it is declared for each element in the collection. In this case, it is going to create a `div` with everything inside it for each product. We can use the Angular expression syntax `{{ }}` to evaluate the products' scope properties such as `_id`, `title`, `description`, and `price`.
- In the third highlighted line, notice a vertical line (or pipe) dividing the `product.price` and `product.description`. They are called **filters**; more on that soon.
- `ui-sref`: This one invokes the state specified, and passes the parameters that are going to match the URL defined in the routes, in this case `/products/:id`. The `:id` is going to be replaced by `product._id` as indicated in the parameter. We haven't created the `viewProduct` state yet, but we will soon.

## Filters

Filters allow modifying the output of an expression into another one. We can also chain multiple filters. They can be used either in the DOM with the pipe `|` inside an expression, or using the `$filter` service. Most of the time, we will just use it in the DOM.

Even though we can define our own, it is worth taking a look into the built-in filters since they are time savers. Some of them are as follows:

- `limitTo`: This truncates the string or array to a specified number of characters.
- `currency`: This is the same as `Number` and adds a currency symbol in front. The default symbol is `$`, but the other ones can be specified as a parameter.
- `number`: This adds comma-separated thousands and two decimal places by default. The default settings can be modified through the parameters.
- `json`: This converts the JavaScript objects to JSON strings.
- `lowercase/uppercase`: This converts strings to lowercase/uppercase.
- `date`: This takes the Unix timestamp and transforms it into a format that we specified.
- Other search filters are `orderBy` and `filter`, to name a few.

## CRUD-ing products with AngularJS

So far, we have all the products showing in the `/products` path. If we click on details, nothing happens. Let's fix that, and also add the rest of the **CRUD** (**create-read-update-delete**) functionality.

### Services

We will build a server-less CRUD; the data will be stored in memory. Nonetheless, when we build the RESTful API in the next chapter, we just need to replace our in-memory storage for the `$http` service. The rest of the application would remain working the same way.

For that, let's refactor our product service to return public methods: `query` to return all the products and `get` to get a single product from the collection. `Edit`, `create`, and `update` are self-explanatory:

```
/* clients/app/products/products.service.js */

angular.module('meanshopApp')
.factory('Product', function () {
  var last_id = 5;
  var example_products = [
    {_id: 1, title: 'Product 1', price: 123.45, quantity: 10,
     description: 'Lorem ipsum dolor sit amet'},
    {_id: 2, title: 'Product 2', price: 123.45, quantity: 10,
     description: 'Lorem ipsum dolor sit amet'},
    {_id: 3, title: 'Product 3', price: 123.45, quantity: 10,
     description: 'Lorem ipsum dolor sit amet'},
    {_id: 4, title: 'Product 4', price: 123.45, quantity: 10,
     description: 'Lorem ipsum dolor sit amet'},
    {_id: 5, title: 'Product 5', price: 123.45, quantity: 10,
     description: 'Lorem ipsum dolor sit amet'}
  ];

  return {
    query: function(){
      return example_products;
    },

    get: function(product){
      var result = {};
      angular.forEach(example_products, function (product) {
        if(product._id == params.id)
```

```
        return this.product = product;
      }, result);
      return result.product;
    },

    delete: function(params) {
      angular.forEach(example_products, function (product, index) {
        if(product._id == params._id){
          console.log(product, index);
          example_products.splice(index, 1);
          return;
        }
      });
    },

    create: function(product) {
      product.id = ++last_id;
      example_products.push(product);
    },

    update: function(product) {
      var item = this.get(product);
      if(!item) return false;

      item.title = product.title;
      item.price = product.price;
      item.quantity = product.quantity;
      item.description = product.description;
      return true
    }
  };
});
```

We needed five main public methods in our service: create, update, delete, and read (get and query). We are using a simple array of objects as a poor man's database. Notice that nothing from outside the factory will be able to access `example_products`. This variable is private, while all the methods in the returned object are public. This technique is called closure. Finally, notice that we are using `angular.forEach()` to iterate through the list of products.



By convention, the names of services/factories are capitalized. For example, `Products`.

We also need to fix the unit test name:

```
/* client/app/products/products.service.spec.js */

describe('Service: Product', function () {

  // load the service's module
  beforeEach(module('meanshopApp'));

  // instantiate service
  var Product;
  beforeEach(inject(function (_Product_) {
    Product = _Product_;
  }));

  it('should do something', function () {
    expect(!!Product).toBe(true);
  });

});
```

In further chapters, we are going to explain and expand more on unit testing. For now, let's just keep all tests passing: `grunt test`.

## Controllers

The next step is to set up the controllers that are going to use the Products service methods:

```
/* clients/app/products/products.controller.js */

angular.module('meanshopApp')
  .controller('ProductsCtrl', function ($scope, Product) {
    $scope.products = Product.query();
  })

  .controller('ProductViewCtrl', function ($scope, $state,
    $stateParams, Product) {
    $scope.product = Product.get({id: $stateParams.id});

    $scope.deleteProduct = function(){
      Products.delete($scope.product);
      $state.go('products');
    }
  })
```



```
    })

    .controller('ProductNewCtrl', function ($scope, $state, Product) {
        $scope.product = {}; // create a new instance
        $scope.addProduct = function(product) {
            Product.create($scope.product);
            $state.go('products');
        }
    })

    .controller('ProductEditCtrl', function ($scope, $state,
    $stateParams, Product) {
        $scope.product = Product.get({id: $stateParams.id});

        $scope.editProduct = function(product) {
            Product.update($scope.product);
            $state.go('products');
        }
    });
```

We have added controllers for each one of the CRUD operations, except for delete. On the products details page, we are going to provide the button for deleting the product. That is why `deleteProduct` is displayed inside the Product View controller.

Notice that we are injecting new dependencies besides the `$scope` and `Products` service, such as `$state` and `$stateParams`. The first one allows us to redirect to a different state or route, while `$stateParams` is an object that contains all the variables from the URL (for example, product id).

## Routes

Now that we have the controllers and services in place, we need a route that links a URL to the controllers and templates.

For instance, suppose we want to show the detail of each product that we click on. For that, we need to set a new route like `/products/:id` and a template. We will use the controller that we created to fetch the products' detailed data, and inject the needed variables into the template. For adding new products and editing the existing ones, the URL will be unique as well:

```
/* meanshop/clients/app/products/products.js */

angular.module('meanshopApp')
```

---

```
.config(function ($stateProvider) {
  $stateProvider
    .state('products', {
      url: '/products',
      templateUrl: 'app/products/templates/product-list.html',
      controller: 'ProductsCtrl'
    })

    .state('newProduct', {
      url: '/products/new',
      templateUrl: 'app/products/templates/product-new.html',
      controller: 'ProductNewCtrl'
    })

    .state('viewProduct', {
      url: '/products/:id',
      templateUrl: 'app/products/templates/product-view.html',
      controller: 'ProductViewCtrl'
    })

    .state('editProduct', {
      url: '/products/:id/edit',
      templateUrl: 'app/products/templates/product-edit.html',
      controller: 'ProductEditCtrl'
    })
  });
});
```

We do not need the delete route because it is a button, and its result will be redirected to the marketplace. Thus, no template is to be shown on deletion.

It is important to notice that each URL parameter might contain a placeholder using colons. The parameter passed to the `ui-sref` in templates will map to the matching URL parameter. For instance, `:id` is going to be replaced later by a real value such as `product._id`.

## Templates

Templates are the actual HTML code. Based on our `products.js` routes file, we will need to create the following product-\* templates: `list`, `view`, `new`, and `edit`. The New product and edit product templates will look almost identical: similar form but different actions. Instead of repeating ourselves, we are going to create a partial template file called `_product_form.html` that we will include in the edit and new templates.

## Partial forms

To keep the files organized, we are going to create all the templates in their own directories, and we will rename `products.html` as `products-list.html`. Create the template directory and template files, as follows:

```
$ mkdir client/app/products/templates/
$ mv client/app/products/products.html client/app/products/templates/
  product-list.html
$ touch client/app/products/templates/_product_form.html
$ touch client/app/products/templates/product-{new,edit,view}.html
```

Now we can add the following to our template:

```
<!-- clients/app/products/templates/_product_form.html -->

<div class="form-group">
  <label for="title" class="col-sm-2 control-label">Title</label>
  <div class="col-sm-10">
    <input type="text" ng-model="product.title" class="form-control"
      id="title" placeholder="product title"/>
  </div>
</div>
<div class="form-group">
  <label for="year" class="col-sm-2 control-label">Description</label>
  <div class="col-sm-10">
    <input type="text" ng-model="product.description"
      class="form-control" id="description"
      placeholder="Describe your product..."/>
  </div>
</div>
<div class="form-group">
  <label for="director" class="col-sm-2 control-label">Price</label>
  <div class="col-sm-10">
    <input type="number" step="any" min="0" ng-model="product.price"
      class="form-control" id="price" placeholder="0.00"/>
  </div>
</div>
<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <input type="submit" class="btn btn-primary" value="Save"/>
  </div>
</div>
```

Notice the `ng-model` directive. It binds the `product` to `$scope.product`. A route relates a controller to a template, and as we saw earlier, the controllers define `$scope.product` which is made available in the template.



By convention, the names of partial templates start with an underscore.

Now we can create two more views (edit and new) that reference the created form.

## Product New

Add the following code to the `product-new.html` file. It will reference the partial form, `_product_form.html`, that we created earlier:

```
<!-- clients/app/products/templates/product-new.html -->

<navbar></navbar>

<div class="container">
  <h1>Create new Product</h1>
  <form class="form-horizontal" role="form" ng-submit="addProduct()">
    <div ng-include="'app/products/templates/_product_form.html'"></div>
  </form>
</div>

<footer></footer>
```

Once the form is submitted (`ng-submit`), the `addProduct` method in `ProductNewCtrl` will perform the creation task.

The `footer` and `navbar` elements are directives defined in the `components` folder. Take a look into the following:

- `client/components/navbar/navbar.directive.js`
- `client/components/footer/footer.directive.js`

## Product edit

The edit form is very similar to the new form. They both reference the partial form.

```
<!-- clients/app/products/templates/product-edit.html -->

<navbar></navbar>

<div class="container">
  <h1>Edit Product</h1>
  <form class="form-horizontal" role="form" ng-submit="editProduct()">
    <div ng-include="'app/products/templates/_product_form.html'">
    </div>
  </form>
</div>

<footer></footer>
```

Again, once submitted, the `editProduct()` method in `ProductEditCtrl` is going to execute the `updateProduct()` method that was made available through the Product factory.

## The product view

It will be nice to have an edit/delete link inside the product details view, so let's add it:

```
<!-- clients/app/products/product-view.html -->

<navbar></navbar>

<div class="container">
  <div class="row">
    <div class="col-sm-6 col-md-4 col-md-offset-2">
      
      <div class="caption">
        <h3>{{product.title}}</h3>
        <p>{{product.description}}</p>
        <p>{{product.price | currency }}</p>
        <p>
```

---

```

        <a href="#" class="btn btn-primary" role="button">Buy</a>
        <a ui-sref="products" class="btn btn-default"
          role="button">Back</a>
        <a ui-sref="editProduct({id: product._id})"
          class="btn btn-default" role="button">Edit</a>
        <a class="btn btn-danger"
          ng-click="deleteProduct()">Delete</a>
      </p>
    </div>
  </div>
</div>

<footer></footer>

```

## The product list

Finally, we have the product list template. This will show all the products and contain the links to create new ones:

```

/* client/app/products/templates/product-list.html */

<navbar></navbar>

<div class="container">
  <div class="row">
    <div class="col-sm-6 col-md-4">
      <h1>Products</h1>
      <p ng-show="products.length < 1">No products to show.</p>
      <a ui-sref="newProduct">New Product</a>
    </div>
  </div>

  <div class="row">
    <div class="col-sm-6 col-md-4" ng-repeat="product in products">
      <div class="thumbnail">
        
        <div class="caption">
          <h3>{{product.title}}</h3>

```

```
<p>{{product.description | limitTo: 100}} ...</p>
<p>{{product.price | currency }}</p>
<p>
  <a href="#" class="btn btn-primary" role="button">Buy</a>
  <a ui-sref="viewProduct({id: product._id})"
    class="btn btn-default" role="button">Details</a>
</p>
</div>
</div>
</div>
</div>
</div>

<footer></footer>
```

That is all we need to get our CRUD working. Run `grunt serve` and test it out.

## Styling the main page

We still have the demo data showing in our main page. Let's personalize our front page by showing the last products as featured, and by adding the link to add new products:

```
<!-- client/app/main/main.html -->

<navbar></navbar>

<header class="hero-unit" id="banner">
  <div class="container">
    <h1>MEANshop</h1>
    <p class="lead">Your mean stack one stop shop</p>
    
  </div>
</header>

<div class="container">
  <div class="row">
    <div class="col-lg-12">
      <h1 class="page-header">Featured Products:</h1>
      <ul class="nav nav-tabs nav-stacked col-md-4 col-lg-4 col-sm-6"
ng-repeat="product in products">
```

---

```

        <li><a ui-sref="viewProduct({id: product._id})"
            tooltip="{{product.description}}">
                {{product.title}} -
                <span class="text-muted">{{product.price | currency}}</span>
            </a></li>
    </ul>
</div>
</div>

<form class="thing-form">
    <label>Sell your Products</label>
    <p class="input-group">
        <span class="input-group-btn">
            <button class="btn btn-primary"
                ui-sref="newProduct()">Add New</button>
        </span>
    </p>
</form>
</div>

<footer></footer>

```

Finally, we need to inject the products using the Factory. Replace the `main.controller.js` with the following:

```

/* client/app/main/main.controller.js */
angular.module('meanshopApp')
    .controller('MainCtrl', function($scope, $http, socket, Product) {
        $scope.products = Product.query().slice(3);
    });

```

You might want to change these files to meet your needs:

- `client/assets/images/logo.png`
- `client/components/navbar/navbar.html`
- `client/components/footer/footer.html`



## Summary

In this chapter, you got familiar with the directory structure while going through the main concepts of AngularJS. We learnt that directives are the way in which AngularJS extends the HTML code. Modules are preferred for keeping the code organized. Routers allow us to define URLs and bind them to controllers and templates. Controllers are the glue between services/factories (model) and the templates (view). Templates are the HTML codes where we can use expressions and variables defined through `$scope` in the controller. Finally, filters are used in templates for modifying the output of strings.

We have created the first part of our frontend e-commerce app while explaining each of the components used. We now have the marketplace which shows all the products and also provides the CRUD functionality to add, remove, and edit products.

In this chapter, we provided the CRUD functionality using data in a predefined array from the product factory. However, in the next chapter, we are going to interact with data from a real database. We are going to use our understanding of the CRUD methods to bind them with the REST verbs, and create a RESTful API to be consumed by our Angular app.

# 3

## Building a Flexible Database with MongoDB

A crucial step in building any web application is to choose the location for storing data. Databases are all the more critical for e-commerce applications. A database needs to be flexible enough to adapt to the ever-growing needs of the future while fitting the current products catalog. Scalability is another feature of paramount importance that can allow any business to grow or shrink. How many orders per minute does the application handle? What's the average response time of users' load peaks and valleys? How long does it take to find a product in a large pool? Will today's schema meet tomorrow's demands? These are a few of the many questions that affect the choice of database.

For around 40 years, relational database technologies (RDBMS) have governed the enterprise world. However, traditional databases often fall short in terms of scaling web applications to millions of users around the world. Today, leading internet companies (Amazon, Google, Facebook, and others) have developed NoSQL technologies to overcome the limitations of the RDBMS. NoSQL is focused on providing scalability, performance, and high availability. It has a growing number of test cases such as big data, big users, cloud computing, and the Internet of Things (IoT).

MongoDB is one of the most popular NoSQL databases. It has been adopted by a number of major websites such as Craigslist, eBay, Foursquare, The New York Times, and others. MongoDB provides a flexible data model and an expressive query language, and is highly scalable. We are going to use MongoDB to build our application. This chapter will get us started with Mongoose, which is a NodeJS driver for MongoDB. In further chapters, we are going to explore more advanced features to scale MongoDB. The following topics will be covered in this chapter:

- Understanding MongoDB
- CRUDing with Mongoose
- Exploring the advanced features of Mongoose
- Reviewing models and the server-side structure

## Understanding MongoDB

MongoDB is an open source, cross-platform, document-oriented database. Instead of using the traditional table-based rigid schemas, MongoDB favors JSON-based documents with dynamic schemas.

## MongoDB daemons and CLI

In the first chapter, we installed MongoDB. There are two main executables that come with the installation:

- `mongod`: This is the database server and daemon
- `mongo`: This is the database client and shell



The `mongod` server has to be running to use the `mongo` client.

Execute the `mongo` shell, and you will see the Command Prompt. It is similar to the NodeJS Command Prompt in the sense that it is a JavaScript environment. We can define custom JS functions apart from the core functions that are also available. For instance, the following `Math` module and `parseInt` are available:

```
mongo> db.products.insert({
...   title: 'Cellphone',
...   stock: parseInt(Math.random()*100),
...   price: Math.random()*100
... })
WriteResult({ "nInserted" : 1 })
```

## Mapping SQL knowledge to MongoDB

For those of us who are already familiar with the SQL world, this section will be useful for translating this knowledge into the MongoDB way of performing similar actions.

### Basics concepts

The following basic concepts and terms will help us speak the same language through this chapter:

SQL	MongoDB	Description
Column	Field	This is a label that represents a kind of data
Row	Document	This is the basic unit of data organized in columns/fields
Table	Collection	This is the collection of rows/documents and columns/fields
Database	Database	This is the group of tables/collections
Index	Index	This is the data structure to speed up queries
Table joins	Embedded docs	These are the link related rows/documents
Primary key	Primary key	This is the unique identifier of a row/column. Usually <code>id</code> or <code>_id</code> .

### Queries

One of the advantages of MongoDB is its powerful querying language. Here are a few examples of common SQL queries and the way they translate to MongoDB:

SQL	MongoDB	Description
<pre>CREATE TABLE products (   id INTEGER PRIMARY   KEY AUTOINCREMENT,   title TEXT,   stock INTEGER,   price REAL );</pre>	(Implicit)	<p>Creates a table/collection.</p> <p>RDBMS needs the schema to be defined upfront whereas MongoDB does not. It has a flexible data storage that can change.</p>

SQL	MongoDB	Description
<pre>INSERT INTO products (title, stock, price) VALUES ('Product 1', 15, 112.36);</pre>	<pre>db.products.insert({   title: 'Product1',   stock: 15,   price: 112.36 })</pre>	Creates a row/ document.
<pre>SELECT * FROM products;</pre>	<pre>db.products.find()</pre>	Retrieves all data from the table/ collection.
<pre>SELECT * FROM products WHERE stock &gt; 10 AND stock &lt; 100;</pre>	<pre>db.products.find({stock: {\$lt: 100, \$gt: 10}})</pre>	Finds all products where the stock is less than 100 and greater than 10.
<pre>UPDATE products SET title = 'MEANshop';</pre>	<pre>db.products.update({}, { \$set: {title: 'MEANshop'} }, { multi: true} )</pre>	Updates a row/document from a table/ collection.
<pre>DELETE FROM products;</pre>	<pre>db.products.remove({})</pre>	Deletes all data from the table/ collection.
<pre>ALTER TABLE products ADD last_update TIMESTAMP;</pre>	(Implicit)	<b>SQL(ite):</b> This adds a new column to the table.  <b>MongoDB:</b> These documents are schema-less so that new fields can be added at any time.

Being schema-less gives much flexibility, since we can add new fields without having to change the previous data.

## Aggregators

**Aggregators** perform a calculation on the data, and usually reduce it to numeric values or filter/sort out data.

SQL	MongoDB	Description
SUM(column) / COUNT(*)	\$sum	This sums/counts all rows/fields
WHERE / HAVING	\$match	This filters documents based on the query passed
GROUP BY	\$group	This groups rows/documents by an expression.
ORDER BY	\$sort	This sorts the data
<b>Examples</b>		
SELECT SUM(price) FROM products AS total;	db.products.aggregate([ \$group: { _id: null, total: { \$sum: "\$price" } } ])	This adds the price of all products.
SELECT title, SUM(price) FROM products GROUP BY title HAVING title == 'Product';	db.products.aggregate([ { \$match: {title: 'Product' }}, { \$group: { _id: '\$title', total: { \$sum: "\$price" } } } ])	This adds all products by title and shows only the one where the title is Product.

[



Try running these MongoDB code snippets on the mongo console, and verify the output.

]

## CRUDing with Mongoose

MongoDB features are great, but how do we use it in NodeJS? There are multiple libraries such as `mongodb-native` and `mongoose`. We are going to use the latter, because it provides validation, casting, and ORM.

Being schema-free could be handy at times; however, there are cases when we want to maintain the consistency of the data. Mongoose offers such an opportunity to enjoy both strict and loose schema. We are going to learn how to CRUD data on MongoDB through Mongoose.

## Schemas

Schemas are the way through which Mongoose provides data types and validations to the MongoDB documents.

We are going to do the examples in the NodeJS shell directly. Go to the `meanshop` folder and run `node` (and run `mongod` if you have not yet done so). Once in the node shell, run this code, and we'll explain the details subsequently:

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema;

mongoose.connect('mongodb://localhost/meanshop_cli');

var ProductSchema = new Schema({
  description : String,
  published   : Boolean,
  stock       : { type: Number, min: 0 },
  title       : { type: String, required: true },
  price       : { type: Number, required: true },
  updated_at  : { type: Date, default: Date.now },
});

var Product = mongoose.model('Product', ProductSchema);
```

Keep the terminal window open while we explain what we just did. In order to connect to MongoDB, we need a URI. We can even have a URI with multiple MongoDB servers in a production environment. Since we are in localhost and using the default port, we do not require a password, username, or port. However, the full URI looks like the following code:

```
// single servers
var uri = 'mongodb://username:password@hostname:port/database'
```

```
// multiple servers in replica sets
var uri = 'mongodb://user:pass@localhost:port/database,mongodb://
anotherhost:port,mongodb://yetanother:port';

mongoose.connect(uri);
```

We pass the URI to `mongoose.connect` in order to connect to MongoDB. We can specify a database name that does not exist yet, and it will be created on the fly.

The next step is to define the new schema. As we can see, we have labels and schema types. The following are the valid schema types that we have at our disposal:

- `String`: This stores a string value encoded in UTF-8
- `Number`: This stores a number value
- `Date`: This stores a date and time object as an `ISODate`
- `Buffer`: This stores binary information, for example, images, files, and so on
- `Boolean`: This stores either `true` or `false`
- `Mixed`: This stores a JSON object which could contain any kind of element
- `ObjectId`: This is a unique identifier of a record, usually used to refer other documents
- `Array`: This can hold a collection of any of the other data types described in this list


Depending on the type that we define our property, the data will be cast to the specified schema type. Besides the data types, we can set up defaults, validations, setters, and getters as we saw in the preceding example.

The final step is compiling the schema into a model using `mongoose.model`. The model is a constructor from where we can create new documents. Go back to the terminal and type the following:

```
> var glass = new Product({title: 'MEANglass', price: 415.20});
> glass.save();
> console.log(glass);
{ __v: 0,
  title: 'MEANglass',
  price: 415.2,
  _id: 55c643b50ce1ad8f5bac5642,
  updated_at: Sat Aug 08 2015 14:00:21 GMT-0400 (AST) }
```



We just saved a new product! There are some more advanced features that we are going to see later. In the following sections, we are going to use that `Product` instance and learn how to CRUD it.

[  Use **Robomongo** (<http://robomongo.org/>) to see the data in MongoDB. ]

## Create

Next, we want to create a new product. There are multiple methods in Mongoose for doing so:

- `Model#save([options], product, )`: This is the instance method that saves a document
- `Model.create(doc(s), [fn])`: This is the class method to automatically create and save the data


`Product.create` is preferable since it is a shortcut; we can use it like this:

```
Product.create({title: 'MEANPhone', price: 523.12}, function(err, data){
  if(err) console.log(err);
  else console.log(data);
});
```

All the mongoose functions follow an asynchronous pattern like most of the functions in JavaScript. We pass a callback method that tells us when the product was saved or if an error occurred. The other method for creating new data is by using `product.save`. We have to first create the product, and then save it:

```
> var product = new Product({title: 'MEANBook', price: 29.99});

> product.save(function(err){
  if(err) console.log(err);
  else console.log(product);
});
```

[  Notice the capitalization convention. Instances are in the lower case: `product.save()`, while the constructors and classes are capitalized: `Product.create()`. ]

For convenience's sake, let's create a generic callback method that we can use from now on:

```
> var callback = function callback(err, docs) {  
  if (err) console.log('----> Errors: ', err);  
  else console.log('----> Docs: ', docs);  
}  
  
// Now, we can rewrite the save function as follows:  
> product.save(callback);
```

## Read

It is time to retrieve our data. Again, Mongoose provides multiple methods for doing so:

- `Model.find(conditions, [fields], [options], [callback])`
- `Model.findById(id, [fields], [options], [callback])`
- `Model.findOne(conditions, [fields], [options], [callback])`

We are going to use `Product.find`; the other two, `findById` and `findOne`, are shortcuts. If we want to get all the products, we can omit the condition in the parameter:

```
Product.find(callback);
```

This will get all our products. We can perform searches if we pass conditions, as follows:

```
Product.find({title: 'MEANBook'}, callback);  
Product.find({title: /mean/i}, callback);  
Product.findOne({title: /^mean/i}, callback);  
Product.find({price: {$gte: 100 }}, callback);
```

The conditions object is very expressive. We can have regular expressions, and even use aggregators like the following:

- `$gte`: greater than or equal; and `$lte`: less than or equal
- `$eq`: equal; and `$ne`: not equal
- `$gt`: greater than; and `$lt`: less than
- `$in`: matches in an array; and `$nin`: not in array

## Update

Similar to find methods, we have three methods to update values in a model:

- `Model.update(conditions, update, [options], [callback])`
- `Model.findByIdAndUpdate(id, [update], [options], [callback])`
- `Model.findOneAndUpdate([conditions], [update], [options], [callback])`

However, unlike `find`, the update methods are destructive. If we are not careful, we can update (unknowingly) all the documents in a collection. Thus, the callback for multi-document updates is different:

```
Product.update({}, { price: 0.00 }, { multi: true }, function (err,
  numberAffected, raw) {
  if (err) return handleError(err);
  console.log('The number of updated docs was: %d', numberAffected);
  console.log('The raw response from Mongo was: ', raw);
});
```

Oops, we just set all the products' prices to 0.00! Run `Product.find(callback)` and verify the prices. Copy and paste one of the `_id` values to update using `findByIdAndUpdate`:

```
Product.findByIdAndUpdate('55567c61938c50c9a339bf86', {price: 100.12},
  callback);
```

## Delete

Finally, the delete methods are as follows:

- `Model.remove(conditions, [callback])`
- `Model.findByIdAndRemove(id, [options], [callback])`
- `Model.findOneAndRemove(conditions, [options], [callback])`

They are almost identical to the update methods; they are also destructive, so pay attention to the queries/conditions passed. Let's get rid of the first product that has a price of 0.00:

```
Product.findOneAndRemove({price: {$eq: 0.00 }}, {price: 149.99},
  callback);
```

By now, we are familiar with all the methods that we need to create a RESTful API in the next chapter. However, before that, let's take a look at some more Mongoose features.

## Exploring a few advanced features in Mongoose

Mongoose not only provides convenience methods for CRUDing documents, but it also offers a number of convenient functionalities. We will now use modified snippets from the `meanshop/server/api/user/user.model.js` project to explain some concepts.

### Instance methods

Instance methods are custom methods that we can add to our schemas. They become available at the instance level. For example:

```
UserSchema.methods.authenticate = function(plainText) {  
  return this.encryptPassword(plainText) === this.hashPassword;  
}
```

The keyword `this` allows us to access the schema properties and other instance methods. We can use instance methods only in the instances of the model, such as the following code:

```
var User = mongoose.model('User', UserSchema);  
var user = new User(); // creates instance of the model  
user.authenticate(password);
```

### The static methods

Static methods are available at the constructor level, as shown in the following code:

```
UserSchema.statics.findByName = function(name, callback){  
  return this.find({name: new RegExp(name, 'i')}, callback);  
}
```

As mentioned earlier, we can use it directly on the constructor, that is, `User` in this case:

```
var User = mongoose.model('User', UserSchema);  
User.findByName('adrian', callback);
```

## Virtuals

**Virtuals** are setters and getters that don't persist in MongoDB. They are useful for composing data from multiple fields using `get`. Furthermore, using `set` can break down data to save it in separate fields.

For instance, let's say that we want to save the user's password in a hash instead of plain text. We can use virtual methods to abstract that:

```
// meanshop/server/api/user/user.model.js *modified excerpt */

var UserSchema = new Schema({
  name: String,
  email: { type: String, lowercase: true },
  hashedPassword: String,
  salt: String
});

UserSchema
  .virtual('password')
  .set(function(password) {
    this._password = password;
    this.salt = this.makeSalt();
    this.hashedPassword = this.encryptPassword(password);
  })
  .get(function() {
    return this._password;
  });
```

We only want the encrypted password and salt to be in the database. Moreover, we want to just deal with the plain text and hide the complexity. The virtual property `password` is doing exactly that. We are setting and getting a plain text with the virtual `password` while setting `salt` and `hashedPassword` behind the scenes.

## Validations

Validations occur when a document attempts to be saved or updated. There are two kinds of validations: **built-in** and **custom** validations.

## Built-in validations

We have already used built-in validations in our `ProductSchema` to validate the required fields and `min` values. To summarize, the following are all the built-in validations:

- All the `SchemaTypes` have the `required` validator
- The `min` and `max` validators are only for the `Number` `SchemaTypes`
- The `enum` from array and `match` `regexp` validators are only for the `String` `SchemaTypes`

## Custom validations

Sometimes, the built-in validators are not enough. That is when custom validators come to the rescue. They are used by passing the property we want to validate in the `path` method, and passing the validator function on the method `validate`, as follows:

```
// meanshop/server/api/user/user.model.js *modified excerpt */

UserSchema
  .path('email')
  .validate(function(value, respond) {
    var self = this;
    this.constructor.findOne({email: value}, function(err, user) {
      if(err) throw err;
      if(user) {
        if(self.id === user.id) return respond(true);
        return respond(false);
      }
      respond(true);
    });
  }, 'The specified email address is already in use.');
```

In the preceding code, we are validating that the e-mail is not already in use. We used `path` to set the property we want to validate. Later, we passed a callback function to perform the validation and show an error message. We can respond with either `respond(true)` or `respond(false)`, or just return a `Boolean`.

## Middleware

Middleware in Mongoose are hooks that execute before (pre) or after (post) certain actions like initialization, validations, update, save, or remove. Middlewares are usually used for triggering custom events, performing asynchronous tasks, and for performing complex validations.

We can chain multiple middleware hooks, and they will execute one after another. Here's an example of many of the possible middleware combinations:

```
schema.post('init', function (doc) {
  console.log('%s has been initialized from the db', doc._id);
});

schema.post('validate', function (doc) {
  console.log('%s has been validated (but not saved yet)', doc._id);
});

schema.pre('save', function (doc) {
  console.log('%s has NOT been saved yet', doc._id);
});

schema.pre('update', function() {
  console.log(this instanceof mongoose.Query); // true
  console.log('%s has NOT been updated yet', this._id);
});

schema.post('update', function() {
  console.log(this instanceof mongoose.Query); // true
  console.log('%s has been updated', this._id);
});

schema.post('remove', function (doc) {
  console.log('%s has been removed', doc._id);
});
```

In the preceding example, we can see multiple pre and post combinations with init, validate, save, remove, and update.

## Reviewing models and server-side structure

So far, we have gained real knowledge of the Mongoose models and MongoDB. Now it is time to see them in action in our application, and to get familiarized with the server directories.

### The server folder

We have already covered in detail the main folders (`client`, `server`, and `e2e`) of the `client` directory in the previous chapter. In this chapter, we are going to focus solely on the `server` directory. Here is an overview what it looks like:

```
meanshop/server
├── api           - Server API components
├── auth         - Authentication handlers
├── components   - App-wide/reusable components
├── config       - App configuration
│   ├── local.env.js - Environment variables
│   └── environment - Node environment configuration
├── views        - Server rendered views
└── app.js       - Bootstrap the application
```

The `app.js` script is the main script. It loads all the other scripts and bootstraps ExpressJS. Take a look on your own and follow the referenced files, just to get familiarized with them. We are going to explain them thoroughly in later chapters.

For the rest of this chapter, we are going to concentrate mainly on the `server/api` directory. Let's take a look at an example to understand what an API resource will look like:

```
meanshop/server/api/thing
├── index.js      - ExpressJS Routes
├── thing.controller.js - ExpressJS Controller
├── thing.model.js - Mongoose model
├── thing.socket.js - SocketIO events
└── thing.spec.js - Controller tests
```

Each API component in our system will have a similar naming convention.



## Current Mongoose models

Take a look under `meanshop/server/api`. Notice the `user` and `thing` folders. Now, take a look at `server/api/thing/thing.model.js`:

```
var mongoose = require('mongoose'),
    Schema = mongoose.Schema;

var ThingSchema = new Schema({
  name: String,
  info: String,
  active: Boolean
});

module.exports = mongoose.model('Thing', ThingSchema);
```

So far, we had explained how the schema works and even defined a `ProductSchema` ourselves. Let us now explain the `module.exports` and `require` methods.

## CommonJS Modules

We have seen that AngularJS keeps the files organized in modules with `angular.module`. On the server side, however, we will do something similar using CommonJS's `module.exports` and `require`.

The purpose of CommonJS is to provide modules in the server-side JavaScript. It provides a way for handling dependencies, and to solve scope issues using the following:

- `require`: This function allows the importing of a module into the current scope.
- `module.exports`: This object allows exporting functionality from the current module. Everything attached to it (functions/attributes) will be available when the `require` function is invoked.
- `exports`: This is the `module.exports` helper. Modules ultimately return `module.exports`, not `exports`. So, everything attached to `exports` is collected and passed to `module.exports` if and only if `module.exports` have not been assigned to anything yet.

These concepts might seem a little abstract. Let's do a couple of examples to drive these home. Create a new file `user.js` as follows:

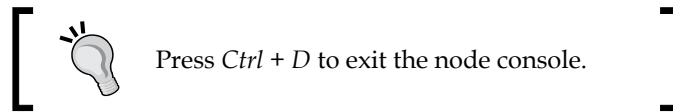
```
// user.js

exports.name = function(name){
  return 'My name is ' + name;
}
```

Go to node shell:

```
$ node
> var user = require('./user');
undefined
> user.name('Adrian');
'My name is Adrian'
```

You were able to access the method name that was exported in `user.js`.



Now, let's try to assign something to `module.export`, and see what happens. Edit `user.js` as follows:

```
// user.js

module.exports = 'Mejia';

exports.name = function(){
  return 'My name is Adrian';
}
```

Run the console again:

```
node
> var user = require('./user');
undefined
> user.name()
TypeError: Object Mejia has no method 'name' ...
> user
'Mejia'
>
```

What went wrong? We just verified that `exports` gets ignored if `module.exports` gets assigned to something, regardless of the order of assignment.



#### Exports versus `module.exports`

The rule of thumb is to use only `exports` to assign multiple properties or functions. On the other hand, when we want to export a single object or a single function, we can do it directly to `module.exports`. Remember never to use both since `module.exports` is going to override everything attached to `exports`.

## The user model

Since we are learning about Mongoose in this chapter, let's see what we have learned so far in our project. Let's take a look at the `user.model.js`. It is a little bit long, so we are just going to highlight certain fragments as examples of what we have learned in this chapter:

```
/* server/api/user/user.model.js *modified excerpt */

var UserSchema = new Schema({
  name: String,
  email: { type: String, lowercase: true },
  role: {
    type: String,
    default: 'user'
  },
  hashedPassword: String,
  provider: String,
  salt: String,
  facebook: {},
  twitter: {}
});
```

We learned that in Mongoose that everything starts with a schema, which has properties and data types. It might also contain default values, middleware, virtual attributes, some built-in validations, and preprocessors (lowercase, trim, and so on).

```
// Virtual attributes

UserSchema
  .virtual('profile')
```

---

```

    .get(function() {
      return {
        'name': this.name,
        'role': this.role
      };
    });
  });

```

Virtual attributes are not stored in MongoDB, but allow us to make composites of different properties with `get`. Furthermore, we can break down the composed values and store them in separate properties using `set`:

```

// Validate empty password

UserSchema
  .path('hashedPassword')
  .validate(function(hashedPassword) {
    if (authTypes.indexOf(this.provider) !== -1) return true;
    return hashedPassword.length;
  }, 'Password cannot be blank');

```

Validations are necessary to keep the data in a healthy state. They are run before save/update, and display user error messages to the user when the conditions are not met. We use `path` to specify the property that we want to validate and `validate` to provide the validator function.

Sometimes, we may want to provide the users the ability to log in with social networks and also with their e-mail ID and passwords. Thus, users only need to offer a password if they do not have a social provider associated. This is a perfect job for a pre-save hook:

```

/* server/api/user/user.model.js *excerpt */

UserSchema
  .pre('save', function(next) {
    if (!this.isNew) return next();

    if (!validatePresenceOf(this.hashedPassword) &&
        authTypes.indexOf(this.provider) === -1)
      next(new Error('Invalid password'));
    else
      next();
  });

```

Finally, we have instance methods:

```
/* server/api/user/user.model.js *simplified excerpt */

UserSchema.methods.authenticate = function(plainText) {
  return this.encryptPassword(plainText) === this.hashPassword;
}
```

In this case, we can attach methods to the instance of the user model using `Schema.methods`. It can be used only in an instance of `User`. For example:

```
/* server/api/user/user.controller.js *excerpt */

User.findById(userId, function (err, user) {
  if (user.authenticate(oldPass)) {
    user.password = newPass;
    user.save(function(err) {
      if (err) return validationError(res, err);
      res.send(200);
    });
  } else {
    res.send(403);
  }
});
```

## Summary

In this chapter, we explored one of the most popular NoSQL databases: **MongoDB**. We also compared the main SQL queries to their counterparts in MongoDB. We further explored the process of using MongoDB in NodeJS using Mongoose. Finally, we rounded off by covering the basics and some advanced features of Mongoose, and highlighted how we are already using it in our project.

While we were going through this chapter, we had a brief overview of another main directory in our project: the `server` folder. Previously, we went through the `client` folder, and in the next chapter, you will learn how to make a RESTful API and how to test it with `e2e` tests.

# 4

## Creating a RESTful API with NodeJS and ExpressJS

**REST (REpresentational State Transfer)** has become the modern standard for building scalable web services. It is fast replacing older alternatives such as **SOAP (Simple Object Access Protocol)** and **WSDL (Web Services Description Language)**. RESTful APIs have earned a widespread acceptance across the Internet because of their simplicity, performance, and maintainability.

On the other hand, ExpressJS is one of the most popular web servers for NodeJS. It comes with support for building RESTful APIs over HTTP and JSON out-of-the-box. ExpressJS not only provides endpoints for APIs, but is also suitable for building single-page, multi-page, and hybrid applications.

Finally, using NodeJS as an API platform comes with many advantages thanks to its non-blocking, event-driven I/O paradigm. Those features make it suitable for building realtime applications that scale well. NodeJS and SocketIO facilitate developers in moving from the traditional, stateless, one-way connection applications to have realtime, two-way connection web applications. The server and clients can initiate communication asynchronously and exchange data instantly. The server can *push* data to the client as soon as it is available. This is in contrast to the typical applications, where only the clients can initiate connections and have to poll the servers periodically for any new information.

Without further ado, let's cover the following topics in this chapter:

- Getting started with REST
- Scaffolding the RESTful APIs
- Bootstrapping ExpressJS
- Understanding Routes in ExpressJS
- Testing, TDD, BDD, and NodeJS
- Creating the product model
- Implementing the product API

## Getting started with REST

REST is a stateless, cacheable, and uniform interface that provides client-server communication. It leverages the HTTP protocol. REST uses the HTTP verb methods such as GET, POST, PUT, PATCH, and DELETE. These methods are accompanied by a **URI (Uniform Resource Identifier)**, which has a protocol, domain, and a path. The media type is also specified in the header of the HTTP request, such as HTML, JSON, XML, Images, and Atom to name a few. For our case, we are going to be using only JSON.

The following table shows an example of our future RESTful API for products:

URI	GET	PUT/PATCH	POST	DELETE
/products	Get all the products	N/A	Creates new product	N/A
/products/1	Get single product	Update product	N/A	Delete product

The GET method is considered safe or nullipotent since it does not have any side effects on the data (it just reads the data). The PUT and DELETE methods are considered idempotent, since their call produces the same results no matter how many times they are called. The POST method is neither nullipotent nor idempotent because it creates new data every time it is called, and every POST request produces changes in the data. Notice that all the bulk update (PUT) and deletions (DELETE) are not implemented, since they are considered unsafe when modifying more than one resource. More practical examples will be given in later sections of this chapter.

**PUT versus PATCH**

PUT is used to replace an existing resource entirely, while PATCH is use for partial updates of a resource.

## Scaffolding RESTful APIs

Our Yeoman generator helps us in scaffolding our endpoint APIs. It creates most of the files that we are going to need.

Let's go ahead and create our products' API endpoint. In the next chapter, we are going to use what we build here, and connect it to our AngularJS app. The Yeoman generator has a command for that:

```
$ yo angular-fullstack:endpoint product
```

```
? What will the url of your endpoint be? /api/products
```

```
create server/api/product/product.controller.js
create server/api/product/product.events.js
create server/api/product/product.integration.js
create server/api/product/product.model.js
create server/api/product/product.socket.js
create server/api/product/index.js
create server/api/product/index.spec.js
```

This command will create some files needed for our API. Run `grunt test` (and `mongodb`); verify that all the tests are passed. That is all we need to have a working products API backed-up in MongoDB. Don't worry, we are going to unveil the magic of each file in turn.

## Bootstrapping ExpressJS

ExpressJS is a web server composed mainly of routes, middlewares, and views. However, since we are aiming to build a **Single Page Applications (SPA)**, we are going to use ExpressJS as a REST endpoint and Angular's views. So from ExpressJS, we are only going to use the routes and middleware.



When the server receives a request, it goes through all the registered middlewares one by one. The middlewares are functions with three parameters; request, response, and next. If no error is found, it hands over the request to the next middleware in the chain. Middlewares are very diverse: they can log information, process cookies, sessions, do authentication, and so on. Similarly, routes too are functions that process requests, but they have only two parameters: request and response (there is no next). Routes are executed only when the URL matches theirs. For example: /products and /products/1222.

Let's shift gears and review the existing file structure in the server folder:

meanshop/server

```
|— api
|   |— product                - Product API
|   |   |— index.js           - Routes
|   |   |— index.spec.js      - Routes tests
|   |   |— product.controller.js - Controller
|   |   |— product.integration.js - Controller tests
|   |   |— product.model.js   - Model
|   |   |— product.model.spec.js - Model tests
|   |   |— product.socket.js   - SocketIO config
|   |   |— product.events.js   - Model event emitter
|   |— thing/*                - Thing API (demo)
|   |— user/*                  - User API
|— config
|   |— environment
|   |   |— development.js     - Development config
|   |   |— index.js           - Loads env scripts
|   |   |— production.js      - Sets production config
|   |   |— test.js            - Sets testing config
|   |— express.js             - Express middleware
|   |— local.env.js           - Environment variables
|   |— seed.js                - Sample data (seeds)
|   |— socketio.js            - (Web)Socket config
|— auth/*
|— components/*
|— views/*
|— app.js                     - bootstrap ExpressJS
|— routes.js                   - Loads all the routes
```

Now, let's go through each file in turn. Everything starts on the server/app.js:

```
/* meanshop/server/app.js */

// Set default node environment to development
process.env.NODE_ENV = process.env.NODE_ENV || 'development';

var express = require('express');
var mongoose = require('mongoose');
var config = require('./config/environment');

// Connect to MongoDB
mongoose.connect(config.mongo.uri, config.mongo.options);
mongoose.connection.on('error', function(err) {
  console.error('MongoDB connection error: ' + err);
  process.exit(-1);
});

// Populate databases with sample data
if (config.seedDB) { require('./config/seed'); }

// Setup server
var app = express();
var server = require('http').createServer(app);
var socketio = require('socket.io')(server, {
  serveClient: config.env !== 'production',
  path: '/socket.io-client'
});
require('./config/socketio')(socketio);
require('./config/express')(app);
require('./routes')(app);

// Start server
function startServer() {
  server.listen(config.port, config.ip, function() {
    console.log('Express server listening on %s:%d, in %s mode',
      config.ip, config.port, app.get('env'));
  });
}

setImmediate(startServer);

// Expose app
exports = module.exports = app;
```

We have three main environments: production, development, and test. Based on the environment, a different MongoDB database is used for each case.

Notice that we require the file `./config/express.js`. This is where all the middleware is set up. *ExpressJS is essentially a chain of middleware calls that ends in a route.* There are middlewares for processing cookies, sessions, logging, authentication, and soon. When the requests hit the server, a middleware processes it and then hands it over to the next registered middleware, till it finally reaches the router. The router passes it on to the matching controller depending on the URL and HTTP verb.

## Understanding routes in ExpressJS

In this section, we are going to explore the routes and controllers and the way in which the Product API is built.

Let's take a look at another generated file—`product.controller.js`:

```
/* meanshop/server/api/product/product.controller.js *excerpt */

// Gets a list of Products
exports.index = function(req, res) {
  Product.findAsync()
    .then(responseWithResult(res))
    .catch(handleError(res));
};

// Gets a single Product from the DB
exports.show = function(req, res) {
  Product.findByIdAsync(req.params.id)
    .then(handleEntityNotFound(res))
    .then(responseWithResult(res))
    .catch(handleError(res));
};

// Creates a new product in the DB.
exports.create = function(req, res) { ... }

// Updates an existing product in the DB.
exports.update = function(req, res) { ... }

// Deletes a product from the DB.
exports.destroy = function(req, res) { ... }
```

The preceding code has five actions: `index`, `show`, `update`, `create`, and `delete`. The `exports` keyword allows making these actions public and accessible when this file is required. The `show`, `update`, and `destroy` actions have a `req.params.id`. The `id` allows access to the product ID from the URL. The rest of this controller is an excellent review of our previous chapter about CRUDing with Mongoose. Thus, all these actions should be pretty familiar to you.



Notice that the Mongoose methods have an additional suffix `-Async`. This was added from the **BluebirdJS** promise package. For more details, go to <https://github.com/petkaantonov/bluebird/blob/master/API.md#promisepromisifyallobject-target-object-options---object>.

Now, let's take a look at `product/index.js`. This is the route that calls each of the actions that we defined in `product.controller.js`.

```
/* meanshop/server/api/product/index.js */

var express = require('express');
var controller = require('./product.controller');

var router = express.Router();

router.get('/', controller.index);
router.get('/:id', controller.show);
router.post('/', controller.create);
router.put('/:id', controller.update);
router.patch('/:id', controller.update);
router.delete('/:id', controller.destroy);

module.exports = router;
```

Since `product.controller` is required under the `controller` variable, this means that we can make use of the five actions defined in there. In the `index.js` file, each one of the actions is associated with a route. Notice that some of them have the `:id` parameter; this will make it accessible to the controller using `req.params.id`:

Finally, let's take a look at `server/routes.js`:

```
/* server/routes.js */

app.use('/api/products', require('./api/product'));
app.use('/api/things', require('./api/thing'));
app.use('/api/users', require('./api/user'));
```

`app.use` mounts our routes created in the `api/product/index.js` on top of the route `/api/products`. The end result is something like this:

Route	GET	PUT/PATCH	POST	DELETE
<code>/api/products</code>	index		create	
<code>/api/products/:id</code>	show	update		delete

Now that we have the controllers and routing ready, let's implement the product models and test them.



#### Cleanup note

Remove the `server/api/thing` directory and the references to it in `server/routes.js` and `server/config/seed.js`.

## Testing, TDD, BDD, and NodeJS

From this chapter on, all the code is considered production code. Thus, having a test suite early on is of paramount importance. We need to ensure high quality and guarantee that when we add new features, all the old functionality keeps working. Furthermore, we want to refactor our code as we go without being afraid that we broke something. Let's introduce **TDD (Test Driven Development)** and **BDD (Behavior Driven Development)**.

In software terms, TDD is like the double-entry bookkeeping in accounting. Every feature has a double entry, one in the production code and another one in the testing code. It helps us in detecting mistakes quickly and reducing the debug time. It has a very short development cycle:

1. Write a test and it should fail since nothing has been implemented yet.
2. Develop just enough production code to implement the test.
3. Finally, refactor the code (and tests), and make sure the tests still succeed.

BDD builds on top of TDD; the tests are usually referred as *specs* (specifications) or scenarios, and the test suite provides a functional documentation of the code. In the following section, we are going to see this combo in action. In NodeJS, there are many tools that we can use for writing our tests. One of them, which is also very popular, is **Mocha**. Mocha was created by TJ Hollowaychuk, who has also created ExpressJS. Take some time to go through the examples at <http://mochajs.org/#getting-started>.

## Creating the product model

In the previous chapter, we learned all that we need to know about Mongoose, MongoDB, and Schemas. We are going to use that knowledge widely to build our Product schema and test it.

We do not want to leave the test till the very end, because sometimes, we tend to skip things that are left for the end. With the test-first approach, our tests will serve as guidelines for things to be implemented. We are going to use Mocha, which is very modular and supports multiple assertion libraries. For testing the models, we are going to use a BDD style assertion `should.js` (also developed for TJ). You can study all the possible assertions at <https://github.com/tj/should.js>.

## Testing the products model

Now, create the `product.model.spec.js` file using the following command:

```
touch server/api/product/product.model.spec.js
```

Following TDD, we should perform the following steps:

1. Create a single test, and watch it fail by running `grunt test:server`.
2. Implement the code and run `grunt test:server` again; it should pass this time.
3. Refactor the tests and code if possible, and the test should still succeed.
4. Write another test, and repeat this cycle until all tests are completed.

However, it will be very extensive for this book to go on developing and testing one by one. So, we are going to copy the full `product.model.spec.js` file from <https://raw.githubusercontent.com/amejiarosario/meanshop/ch4/server/api/product/product.model.spec.js> and paste it.

Let's stop here, and explain what's going on. The first thing that you will notice on opening `product.model.spec.js` is that we include `ProductSchema`, which is a generic schema created by the `yo` generator. We will modify the model later to make all the tests pass.

Most of the BDD style tests are grouped in the `describe` blocks, and inside these blocks there are `it` blocks. They are both JavaScript functions that take two parameters: one is the plain English text string, and the other one is a callback function. The `it` blocks are the ones that define the tests in their callback functions. Notice that some of the callback functions have the `done` parameter. The `done()` function helps us to test asynchronous functions, and when called, it also returns errors with `done(err)`. The `beforeEach` blocks are executed before each `it` block inside their current `describe` block. In our case, we are going to remove all the products before starting each test using `Product.remove(done)`. The block `beforeEach` helps us to test each block in isolation so that the products created in the previous tests do not interfere with the tests that follow.

Run `grunt test:server` in the console, and watch how all the tests fail. Here's a challenge: Can you make modifications to `product.model.js` to make all of them pass without seeing the solution?

## Product model implementation

The following changes are needed to meet our (test) requirements:

```
/* server/api/product/product.model.js *excerpt */

var ProductSchema = new Schema({
  title: { type: String, required: true, trim: true },
  price: { type: Number, required: true, min: 0 },
  stock: { type: Number, default: 1 },
  description: String
});
```

Now all the tests should succeed and will look similar to this:

```
Product
  ✓ should not create without title
  ✓ should remove trailing spaces from title
  ✓ should default stock to 1
  ✓ should not create without price
  ✓ should not allow negative price
  ✓ should save a description
```

The tests have now documented our Product schema describing what our products do and what it validates.

Running `grunt test:server`, made the Products tests succeed but it broke the `product.integration.js` tests. Let's implement the API and fix that.

## Implementing the Product API

The Product API will require all the knowledge that we acquired about Mongoose, ExpressJS routes, and RESTful APIs.

Our generator has already generated a good starter code for the Product API. So, we just need to tweak some variables to make it work with our new product model.

## Testing the API

The final goal of the tests is to ensure that our API replies with the right status code and responses to different scenarios. We are going to test the following scenarios:

```
Product API:
GET /api/products
  ✓ should respond with JSON array
POST /api/products
  ✓ should respond with the newly created product
GET /api/products/:id
  ✓ should respond with the requested product
PUT /api/products/:id
  ✓ should respond with the updated product
DELETE /api/products/:id
  ✓ should respond with 204 on successful removal
  ✓ should respond with 404 when product does not exist
```

We are going to use **SuperTest** (also created by TJ) to provide a high-level abstraction for testing the HTTP routes for our controllers. Check out <https://github.com/visionmedia/supertest> for more details.

Copy the adjusted tests from <https://raw.githubusercontent.com/amejiarosario/meanshop/ch4/server/api/product/product.integration.js>

## Index action tests

Open the `product.integration.js` file and take a look at the tests inside the `GET /api/products` block. The format is BDD, using the `describe` and `it` blocks. What is new is the use of the SuperTest requests and expectations. We used `expect()` to assert the status code and match response headers, such as `content-type JSON`. The `end()` function will perform the request, catch any errors, and display the response.



## Show action tests

The next test will be the show action `GET /api/products/:id`, which returns a single product with the matching `ObjectId`.

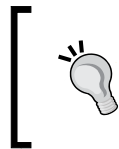
These tests have a `beforeEach()` function which runs before each it block. It cleans the database and creates two valid products. It stores the results of the last product in the `existing_product` variable. Notice that MongoDB generates `_id` automatically from a special type called `ObjectId`. It should have 24 hex characters else it will throw a casting error.

## Creating action tests

The next test is the create product function. These tests attempt to create a valid product and an invalid product. The latter is expected to receive validation errors since the price is negative.

## Deleting action tests

Finally, we have the update and destroy actions. Both are similar in that they can validate and update/delete valid products. Additionally, they validate that the not found products are handled properly.



The full `product.integration.js` file can be found at <https://raw.githubusercontent.com/amejiarosario/meanshop/ch4/server/api/product/product.integration.js>.

## Product controller

The product controller provides the implementation for each of the five actions: `index`, `show`, `create`, `update`, and `delete`.

Notice that all the highlighted code are the changes from the original auto-generated file. All the actions that are required to handle the product ID (`show`, `update`, and `delete`) use a function called `findByIdAsync`.

## Summary

This chapter has been a big jump start into ExpressJS routes, middleware, and controllers. It built on what you learned in the previous chapters about MongoDB for creating a Product model and controller. We covered some supporting topics needed for implementing a high quality RESTful API such as testing and REST basics.

In the next chapter, we are going to use the product API created here and connect it with the AngularJS side. That will complete the CRUD for products from end to end!



# 5

## Wiring AngularJS with ExpressJS REST API

In this chapter, we are going to wire the Angular app we built in *Chapter 2, Building an Amazing Store Frontend with AngularJS* with the web server we built in *Chapter 4, Creating a RESTful API with NodeJS and ExpressJS*. In other words, we are going to connect the front store with the NodeJS RESTful API and introduce testing for AngularJS. Furthermore, we are going to improve the e-commerce app.

We will cover the following topics in this chapter:

- Implementing a RESTful service
- Wiring the Product Controller with new RESTful methods
- Uploading product images
- Testing RESTful APIs in AngularJS
- End-to-end testing

### Implementing a RESTful product service

We are going to implement the following user story that we introduced in *Chapter 1, Getting Started with the MEAN Stack*:

*As a seller, I want to **create products**.*

This time we are going to use the products that the API built in *Chapter 4, Creating a RESTful API with NodeJS and ExpressJS* and replace the content in `products.service.js`. For that we are going to use `ngResource`, which is an Angular service for communicating with RESTful APIs.

We just need to inject `ngResource` as a dependency:

```
/* client/app/app.js */

angular.module('meanshopApp', [
  'ngCookies',
  'ngResource',
  'ngSanitize',
  'btford.socket-io',
  'ui.router',
  'ui.bootstrap'
])
```

Use it as follows:

```
/* client/app/products/products.service.js */

angular.module('meanshopApp')
  .factory('Product', function ($resource) {
    return $resource('/api/products/:id', null, {
      'update': { method: 'PUT' }
    });
  });
```


That's it! That's all we need to connect with the backend. Notice that we use `$resource` to pass the URL of the API endpoint, in our case `/api/products/:id`. The second parameter is the default value for the URL. Finally, the last parameter defines the custom actions that extend the default set of actions. We added the `update` action, because the default actions do not include it. Here's a list of the default actions:

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

What we need to do next is to adjust our product controller to use the new methods. Back in *Chapter 2, Building an Amazing Store Frontend with AngularJS*, we added a button to create new products in the homepage. Make sure you can create new products.

## Building the marketplace

Continuing with the user stories, next we have the following:

[  As a user, I want to see all published products and their details when I click on them. ]

For that, let's change the main controller to show a summary of all the products:

```
/* client/app/main/main.controller.js */

angular.module('meanshopApp')
  .controller('MainCtrl', function($scope, $http, socket, Product) {
    $scope.products = Product.query();
  });
```

Now, after running `grunt serve`, we can see the seeded products on display, and when we click on a product, we can see its details as well.

## Wiring the product controller with new RESTful methods

Now, let's enable the edit/delete buttons for the products. Our product controller will work pretty much in the same way as it did in *Chapter 2, Building an Amazing Store Frontend with AngularJS*. But this time, we will make use of the callbacks:

```
/* client/app/products/products.controller.js */

var errorHandler;

angular.module('meanshopApp')
  .controller('ProductsCtrl', function ($scope, Products) {
    $scope.products = Products.query();
  })

  .controller('ProductViewCtrl', function ($scope, $state,
    $stateParams, Products) {
    $scope.product = Product.get({id: $stateParams.id});
    $scope.deleteProduct = function() {
      Product.delete({id: $scope.product._id},
        function success(/* value, responseHeaders */) {
          $state.go('products');
        }
      );
    };
  });
```

```
    }, errorHandler($scope));
  };
})

.controller('ProductNewCtrl', function ($scope, $state, Product) {
  $scope.product = {}; // create a new instance
  $scope.addProduct = function(){
    Product.save($scope.product,
      function success(value /*, responseHeaders*/) {
        $state.go('viewProduct', {id: value._id});
      }, errorHandler($scope));
  };
})

.controller('ProductEditCtrl', function ($scope, $state,
  $stateParams, Product) {
  $scope.product = Product.get({id: $stateParams.id});
  $scope.editProduct = function(){
    Product.update({id: $scope.product._id,
      $scope.product, function success(value /*, responseHeaders*/) {
        $state.go('viewProduct', {id: value._id});
      }, errorHandler($scope));
  };
});

errorHandler = function ($scope){
  return function error(httpResponse){
    $scope.errors = httpResponse;
  };
};
```

The following new additions were made in the code:

- Callbacks for asynchronous responses when products are created/edited/deleted
- Redirect to the product view when a product is created/edited
- Use of `$scope.errors` to show errors

To view the backend errors, add the following code snippet to the `_product.form.html` file:

```
<!-- meanshop/client/app/products/_product.form.html *excerpt -->

<div ng-show="errors" class="alert alert-danger" role="alert">
  <strong>Error(s) :</strong>
  <p>{{errors}}</p>
</div>
```

## Uploading product images

For the time being, we have been using a placeholder for the product images. Let's again wire the client with the server to handle image upload. For the frontend, we are going to use `ng-file-upload`, and for the backend, we are going to use a middleware called **connect-multiparty**.

## Uploading files in Angular

File upload is not very straightforward in Angular. We cannot just bind to an input file HTML tag and expect it to work. Fortunately, there are plenty of libraries out there, like the one we are using, to solve the problem. Let's install `ng-file-upload`:

```
bower install ng-file-upload --save
bower install ng-file-upload-shim --save
```

This library has useful features such as drag and drop, validation, file upload progress, cancel ongoing upload, and more.

After downloading the package, we need to list it as a dependency at the root of our Angular dependencies:

```
/* client/app/app.js *excerpt */

angular.module('meanshopApp', [
  'ngCookies',
  'ngResource',
  'ngSanitize',
  'btford.socket-io',
  'ui.router',
  'ui.bootstrap',
  'ngFileUpload'
])
```



Now let's add it to the product form. We are going to change the layout a little bit to make room for the image upload:

```
/* client/app/products/templates/_product_form.html */

<div ng-show="errors" class="alert alert-danger" role="alert">
  <strong>Error(s) :</strong>
  <p>{{errors}}</p>
</div>

<div class="row">
  <div class="col-md-2">
    <div ngf-select="upload($file)" ngf-drop="upload($file)"
      class="drop-box"
      ngf-drag-over-class="dragover" ngf-multiple="true"
      ngf-pattern="'image/*'" ngf-max-size="15MB"
      accept="image/*">Drop Images here or click to upload</div>
    <div ngf-no-file-drop>File Drag/Drop is not supported for this
      browser</div>

    <button accept="image/*" ngf-max-size="15MB" type="file"
      class="btn btn-primary" ngf-select="upload($file)">Upload on
      file select</button>

    <h2 ng-show="file">Preview</h2>
    
    <div ng-show="file">{{file.name}} {{file.$error}}
    {{file.$errorParam}}</div>
  </div>

  <div class="col-md-10">
    <div class="form-group">
      <label for="title" class="col-sm-2 control-label">Title</label>
      <div class="col-sm-10">
        <input type="text" ng-model="product.title"
          class="form-control" id="title" placeholder="product title"/>
      </div>
    </div>
    <div class="form-group">
      <label for="year" class="col-sm-2 control-label">Description
      </label>
      <div class="col-sm-10">
        <input type="text" ng-model="product.description" class="form-
        control" id="description" placeholder="Describe your product...">
      </div>
    </div>
  </div>
</div>
```

```
</div>
<div class="form-group">
  <label for="director" class="col-sm-2 control-label">Price
</label>
  <div class="col-sm-10">
    <input type="number" step="any" min="0" ng-model="product.
      price" class="form-control" id="price" placeholder="0.00"/>
  </div>
</div>

<div class="form-group">
  <div class="col-sm-offset-2 col-sm-10">
    <input type="submit" class="btn btn-primary" value="Save"/>
  </div>
</div>
</div>
</div>
```

Code highlights:

- We validate that the uploads are images only, and that the images are not larger than 15 MB
- We get an immediate preview when the image is uploaded

Next, we are going to add some styling to make it look good:

```
/* client/app/products/products.scss */

.drop-box {
  background: #F8F8F8;
  border: 5px dashed #DDD;
  width: 170px;
  height: 170px;
  display: flex;
  justify-content: center;
  align-items: center;
  margin-bottom: 25px;
}

.drop-box.dragover {
  border: 5px dashed blue;
}

.drop-box.dragover-err {
  border: 5px dashed red;
}
```

Finally, we bring the template actions to life in the controller:

```
/* client/app/products/products.controller.js *excerpt */

.controller('ProductEditCtrl', function ($scope, $state,
    $stateParams, Product, Upload, $timeout) {
    $scope.product = Product.get({id: $stateParams.id});
    $scope.editProduct = function(){
        Product.update({id: $scope.product._id}, $scope.product,
            function success(value /*, responseHeaders*/) {
                $state.go('viewProduct', {id: value._id});
            }, errorHandler($scope));
    };

    $scope.upload = uploadHandler($scope, Upload, $timeout);
});

errorHandler = function ($scope){
    return function error(httpResponse){
        $scope.errors = httpResponse;
    };
};

uploadHandler = function ($scope, Upload, $timeout) {
    return function(file) {
        if (file && !file.$error) {
            $scope.file = file;
            file.upload = Upload.upload({
                url: '/api/products/'+$scope.product._id+'/upload',
                file: file
            });

            file.upload.then(function (response) {
                $timeout(function () {
                    file.result = response.data;
                });
            }, function (response) {
                if (response.status > 0){
                    console.log(response.status + ': ' + response.data);
                    errorHandler($scope)(response.status + ': ' + response.
                        data);
                }
            });
        }
    };
};
```

```

        file.upload.progress(function (evt) {
            file.progress = Math.min(100, parseInt(100.0 *
                evt.loaded / evt.total));
        });
    }
};

```

## Handling file upload on Node

When the client side sends a POST request with an image, we need to add the code to handle that. Let's start by installing a file handler:

```
npm install connect-multiparty --save
```

The first thing we need to do is to define the route. It's should match the `/api/products/:id/upload` pattern. Let's add the upload route to the products:

```

/* server/api/product/index.js *excerpt*/

var uploadOptions = { autoFile: true,
                      uploadDir: 'client/assets/uploads/'
}
var multiparty = require('connect-multiparty');

router.post('/:id/upload', multiparty(uploadOptions), controller.
upload);

```

Notice that for the upload action, we have added a middleware called `multiparty` and also passed some options. In the upload options we can specify where we want to store the files; in this case, we have specified the local file system (`client/assets/uploads/`). We could upload it directly to a cloud service such as Amazon S3, but we will talk about it later, in the upcoming chapters.

Now that we have the route defined, we need to handle the upload providing the corresponding action:

```

/* server/api/product/product.controller.js *excerpt */

var path = require('path');

function saveFile(res, file) {
    return function(entity){
        var newPath = '/assets/uploads/' + path.basename(file.path);
        entity.imageUrl = newPath;
        return entity.saveAsync().spread(function(updated) {
            console.log(updated);
        });
    };
}

```

```
        return updated;
      });
    }
  }

  // Uploads a new Product's image in the DB
  exports.upload = function(req, res) {
    var file = req.files.file;
    if(!file){
      return handleError(res)('File not provided');
    };

    Product.findByIdAsync(req.params.id)
      .then(handleEntityNotFound(res))
      .then(saveFile(res, file))
      .then(responseWithResult(res))
      .catch(handleError(res));
  };
};
```

The upload action invokes the `saveFile` function, which is going to update the `imageUrl` attribute in the product. Finally, we add the attributes in the product model:

```
/* server/api/product/product.model.js */

var ProductSchema = new Schema({
  title: { type: String, required: true, trim: true },
  price: { type: Number, required: true, min: 0 },
  stock: { type: Number, default: 1 },
  description: String,
  imageBin: { data: Buffer, contentType: String },
  imageUrl: String
});
```

In the preceding snippet, we added `imageUrl` to store the path in the system or the cloud, where the image is hosted. We also added `imageBin` to store the binary of the image; we could save the image in the database itself. However, this is not advisable, since it could have some scalability issues in the future.

We can automate the process of creating mock products using seeds. Let's do that next.

## Seeding products

Now that we have everything in place, let's create some mock products to work with. Feel free to use your creativity here:

```
/* server/config/seed.js */

var Product = require('../api/product/product.model');
Product.find({}).removeAsync()
  .then(function() {
    Product.createAsync({
      title: 'MEAN eCommerce Book',
      imageUrl: '/assets/uploads/meanbook.jpg',
      price: 25,
      stock: 250,
      description: 'Build a powerful e-commerce...'
    }, {
      title: 'tshirt',
      imageUrl: '/assets/uploads/meantshirt.jpg',
      price: 15,
      stock: 100,
      description: 'tshirt with the MEAN logo'
    }, {
      title: 'coffee mug',
      imageUrl: '/assets/uploads/meanmug.jpg',
      price: 8,
      stock: 50,
      description: 'Convert coffee into MEAN code'
    })
    .then(function() {
      console.log('finished populating products');
    });
  });
```

You can change `imageUrl` to whatever filename you want as long as it is present in the uploads directory.

## Testing RESTful APIs in AngularJS

Testing is crucial to ensure the quality of any software, and this e-commerce app is no exception. The developers can either waste most of the development time chasing bugs, or investing that time in what really matters: algorithms, business logic, and UX/UI improvements. We are going to use three tools to test our client-side code: **Karma** and **Mocha/Chai/SinonJS** for unit testing, and **Protractor** for end-to-end testing.

### Unit testing


In the previous chapter, we implemented some unit testing for Angular and for Node controllers and models.

We are going to continue our unit testing/implementation with the Product service. As we saw in the previous chapters, the service/factory is the module in charge of retrieving, saving, updating, and deleting the Product data (CRUD-ing products) from the database. In the service unit test, we just test the factory logic with no dependencies on other parts such as a database or RESTful API. For that, we are going to send mock HTTP calls with `$httpBackend`, and inject data into the controllers using `ngMock`.

### ngMock

**ngMock** is a mocking module from the AngularJS core that helps us in injecting variables into the tests and mock AngularJS services. It also provides the ability to inspect them. Some of the services that ngMock provides are:

- `$httpBackend`: This is a fake HTTP backend which can reply to requests with predefined responses. For example, `$httpBackend.expectGET('/products').respond({title: 'book'})`.
- `$controller`: This is useful for testing controllers and directives. For example, `var instanceController = $controller('ProductsCtrl', {$scope: scope});`.
- Other mock services: `$timeout`, `$interval`, `$log`, and `$exceptionHandler`.

[  You can read more about ngMock at <https://docs.angularjs.org/api/ngMock>. ]

## Setting up testing

All the tests/implementations that we are going to develop are under `meanshop/client/app/products`. We will now test the Product Factory. Previously, in *Chapter 2, Building an Amazing Store Frontend with AngularJS*, we implemented a temporary factory that stores the data in memory. This time we are going to implement a `Factory` that uses REST to communicate with the ExpressJS web server and database. Later, we are going to unit test the Product Controller. There's no need to test the Product Factory again in the controller's unit tests. That is because, the `Factory` methods are mocked with the Jasmine spies in the controller's unit tests. Finally, we are going to do end-to-end tests using Protractor.

The commands that we are going to use are the following. Run all client and server unit tests:

```
grunt test
```

Before running `e2e`, we need to update the protractor web driver if we have not yet done so, by running the following command:

```
node node_modules/grunt-protractor-runner/node_modules/protractor/bin/webdriver-manager update
```

Also, let's create a new directory under `e2e` folder; create the product spec:

```
$ mkdir -p e2e/products
$ touch e2e/products/product{s.spec,.po}.js
```

Finally, we can run all `e2e` tests:

```
grunt test:e2e
```

## Understanding the Services tests

The first action is to get a list of the products. We need to make an AJAX GET request to `/api/products`. Copy the content of <https://github.com/amejiarosario/meanshop/blob/ch5/client/app/products/products.service.spec.js> to your `products.service.spec.js` file. Let us now explain each element:

First, let's focus on the `#index` test. Most of the tests will look the same: we expect an `$http` call with a specific URL and HTTP verb to be made. In this case, the URL is `/api/products` and the verb is GET. The real `$http` is synchronous; however, the `$httpBackend` is not for easing the testing scenarios. It uses `$httpBackend.flush()` to check the replies to the HTTP requests synchronously.

Run `grunt test:client`.



## Testing all \$resource methods

\$resource uses the \$http service behind the scenes to make the calls to a REST endpoint. It provides a good starting number of methods, but we can extend our own ones like in the update method.

Let's continue taking a look at the #update tests. This time we pass a parameter with the ID of 123 to fetch the product. Later, we pass `updated_attributes`, which simulates the new products' data and expects it back. We use `expectPUT` to match the verb. Also, `expectPUT` and `expectDELETE` are available.

Run the tests again, and verify that everything gets passed. If we had not previously added the update method to \$resource, this last test would not pass.

The delete, create, and show tests are very similar, and the full test suite can be viewed at <https://raw.githubusercontent.com/amejiarosario/meanshop/ch5/client/app/products/products.service.spec.js>.

## Testing the Product Controller

Wait a moment! This controller uses our previously tested Product service. So, should we mock the HTTP requests as we did in the service test, or should we mock the service methods? Yes, since we've already tested the service, we can safely mock it in the controllers' tests at <https://raw.githubusercontent.com/amejiarosario/meanshop/ch5/client/app/products/products.controller.spec.js>.

Some highlights:

- In AngularJS, all controller's scope inherit from \$rootScope. We can create sub-scopes using the \$new method. We create a new scope on each controller, and pass it using the \$controller function.
- We mock the Products service with SinonJS. Check out <http://sinonjs.org/> for full details.

## End-to-end testing

This is where things start to get more interesting! So far, we have been stubbing the HTTP requests, but not in end-to-end tests. This kind of test does not stub anything. It hits the database and the web server, and even simulates user interactions (clicks, keystrokes, and so on) with the browser. Let's go ahead and get started with Protractor and our tests.

```
$ grunt test:e2e
```

When the tests are run, the browser opens up, and if we observe carefully, we will see text boxes, filling out forms, and clicks. Since we have changed things in the UI, some tests might be broken. The updated versions are available at the following links:

- <https://github.com/amejiarosario/meanshop/blob/ch5/e2e/main/main.spec.js>
- <https://raw.githubusercontent.com/amejiarosario/meanshop/ch5/e2e/products/product.po.js>
- <https://raw.githubusercontent.com/amejiarosario/meanshop/ch5/e2e/products/products.spec.js>

As we can notice, we can fill out forms (`sendKeys`), click on buttons and links (`click`), get URLs (`browser.getCurrentUrl`), and perform many more other actions. The e2e tests take more time than unit tests. Thus, instead of each one being independent of each other, each test prepares the next one: *create product | read product | update product | delete product*.



Find out more about the protractor syntax at <https://angular.github.io/protractor/#/api> and about Page Object Pattern at <http://bit.ly/AngularPageObject>.

## Cleaning the database on each e2e run

There's one issue with the e2e tests. The data is not cleaned up on each run. To fix this, we can add new tasks in our Grunt task runner:

```
/* meanshop/Gruntfile.js *excerpt */

grunt.registerTask('db', function (target) {
  if(target === 'clean'){
    var done = this.async();
    var config = require('./server/config/environment');
    var mongoose = require('mongoose');

    mongoose.connect(config.mongo.uri, config.mongo.options,
      function(err){
        if(err) {
          done(err);
        } else {
          mongoose.connection.db.dropDatabase(function (err) {
            if(err) {
              console.log('Connected to ' + config.mongo.uri);
              done(err);
            } else {
              console.log('Dropped ' + config.mongo.uri);
              done();
            }
          });
        }
      });
  }
});
```

These new tasks will allow us to clean up the database by calling the `db:clean` function. We do not want to invoke it manually, so let's add it to the `e2e` workflow in the same file:

```
/* meanshop/Gruntfile.js *excerpt */

else if (target === 'e2e') {
  return grunt.task.run([
    'clean:server',
    'env:all',
    'env:test',
    'db:clean',
    'injector:sass',
    'concurrent:test',
    'injector',
    'wiredep',
    'autoprefixer',
    'express:dev',
    'protractor'
  ]);
}
```

Now every time we run `test:e2e`, it will clean up the database.

## Summary

In this chapter, we completed the CRUD for the Products from the database, and the web server, all the way to the client side. We connected the frontend to the backend, and implemented the most basic features. You learnt how to test the application to ensure that every new feature that we add from here on doesn't break the existing functionality.

Hold on tight, the upcoming chapters are going to be exciting and fun! We are going to add user authentication, search and check out products, and so on. Chapter by chapter, our app will turn into a production-ready ecommerce app.



# 6

## Managing User Authentication and Authorization

One of the most important features of web applications is the ability to allow users to authenticate. The application should keep track of the authenticated users; however, web requests are stateless. Each request is treated as an unrelated transaction to any previous request. There are multiple workarounds to keep track of logged users such as sessions, cookies, and tokens. Once we solve the authentication needs, another important feature is authorization. It dictates what the users can see and do inside the app. Authorization assigns roles to the users. For instance, a seller should be able to edit his/her own products but not anyone else's. However, administrators should be able to edit/delete any product that they find inappropriate.

Modern applications allow logins not only through usernames and passwords, but also through social networks. This latter strategy of authentication usually uses OAuth, which is an open protocol that enables secure authentication through third-party applications.

In this context, we are going to cover the following topics in this chapter:

- Getting started with authentication strategies
- Understanding client-side authentication
- Understanding server-side authentication
- Authenticating with Facebook, Google, and Twitter

## Getting started with authentication strategies

Generally, the task of implementing user authentication can be time-consuming and repetitive. Thankfully, the Yeoman generator gets us started with all the boilerplate code. We will start with the most common authentication strategies. Later, we will make them functional using the Facebook, Google, and Twitter API keys. Then, we will go through the backend authentication and routes. And finally, we are going to create end-to-end tests to make sure the users can log in using their username/password and social authentications.

Let's first take a necessary detour, and explain how the different authentication mechanisms work.

### Session-based authentication

Session-based authentication is one of the most common methods for providing authentication. It uses cookies to save a session ID that is usually related to the user ID. Once the user is logged in, the session ID is passed on each request:

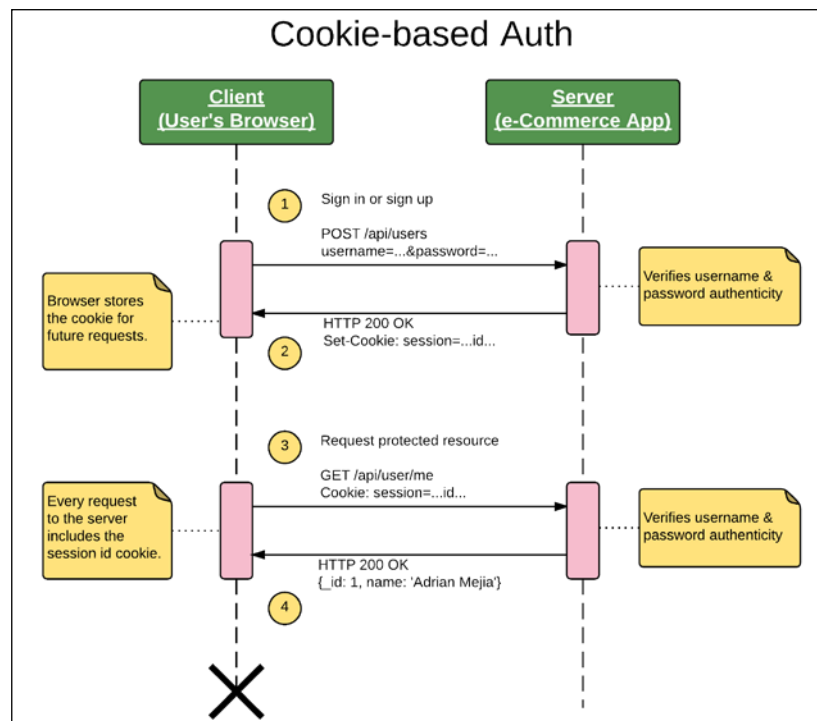


Figure 1: Session-based authentication sequence diagram

How it works:

1. The browser sends a `POST` request to the server with the username and password.
2. If the username/password combination matches the records, the server responds with a *cookie* containing the *session ID*, and it is stored locally in the browser's cookie, session, or local storage.
3. From that time onwards, the browser will include the cookie session at each request.
4. The server verifies the authenticity of the *cookie's session ID* and replies accordingly.

## Token-based authentication – using JWT

Token-based authentication uses JSON web tokens instead of cookies. Once the user is logged in, a token is added to the HTTP header of each request to validate the user.

The JSON Web Token authentication diagram might look similar to the session-based authentication, but it brings some advantages that we are going to discuss later.

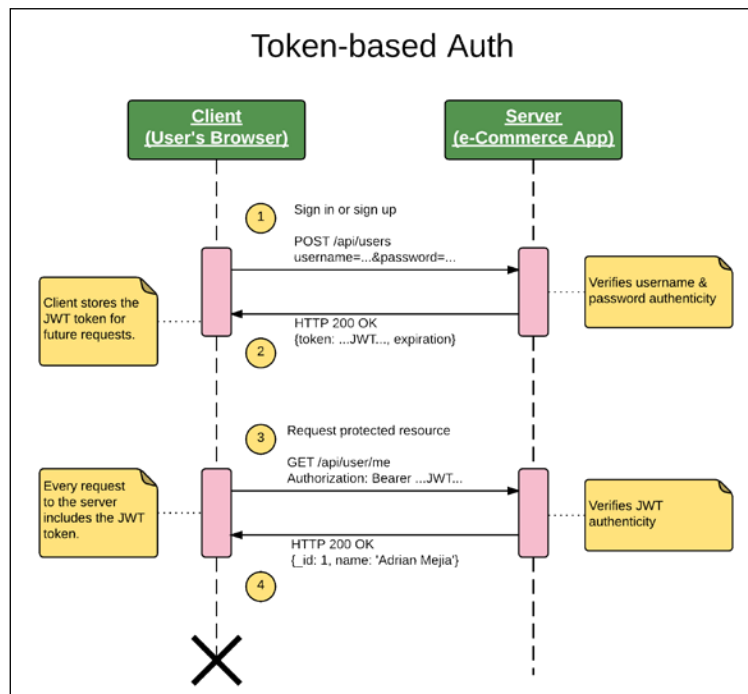


Figure 2: Token-based authentication sequence diagram



The main difference is that instead of relying on cookies, it uses an HTTP header to send the authentication token.

This is how it works:

1. The browser sends a POST request to the server with the username and password.
2. If the username/password combination matches the records, the server responds with a **JSON Web Token (JWT)** containing the *user data*, and it is stored locally in the browser's cookie, session, or local storage.
3. From that time onwards, the client needs to add the HTTP header, `Authentication: Bearer TOKEN`.
4. The server verifies the authenticity of the *authentication token in the HTTP header* and replies accordingly. Note that the cookie is not used for authentication purposes in this strategy.

There seems to be only one subtle difference between these two kinds of authentication, but there are great advantages of using JWT over session-based authentication such as the following:

- **Cross-Origin Resource Sharing (CORS):** Cookie-based authentication and AJAX don't play very well across multiple domains. On the other hand, token-based authentication works seamlessly.
- **Scalability:** Session IDs need to be stored in the server which implies some challenges when using distributed applications, and it also adds some lookup latency. JWTs are self-signed, and any instance of the server can validate its authenticity. The servers just need to share the private key.
- **Mobile:** Cookies are not ideal for working with mobile devices and secured APIs. The token approach simplifies this a lot.
- **Security:** There is no need to worry about **CSRF (Cross-site request forgery)**, since no cookies are used.
- **Others advantages:** JWT is performance and standard-based.



For a deeper understanding about how JWT works, take a look at <http://jwt.io/>.



## OAuth authentication

OAuth-based authentication is popular on social networks for allowing third-party applications to provide a single sign-on. For instance, you might want to login to an app using your Twitter account information without compromising your Twitter account credentials (username/password). That's what OAuth enables us to do, and it is useful for both registering and signing in as a user with just one click.

The following diagram shows the interactions between the client (the customer's browser), server (the e-commerce application), and the OAuth provider (for example, Twitter). This is called a three-legged OAuth:

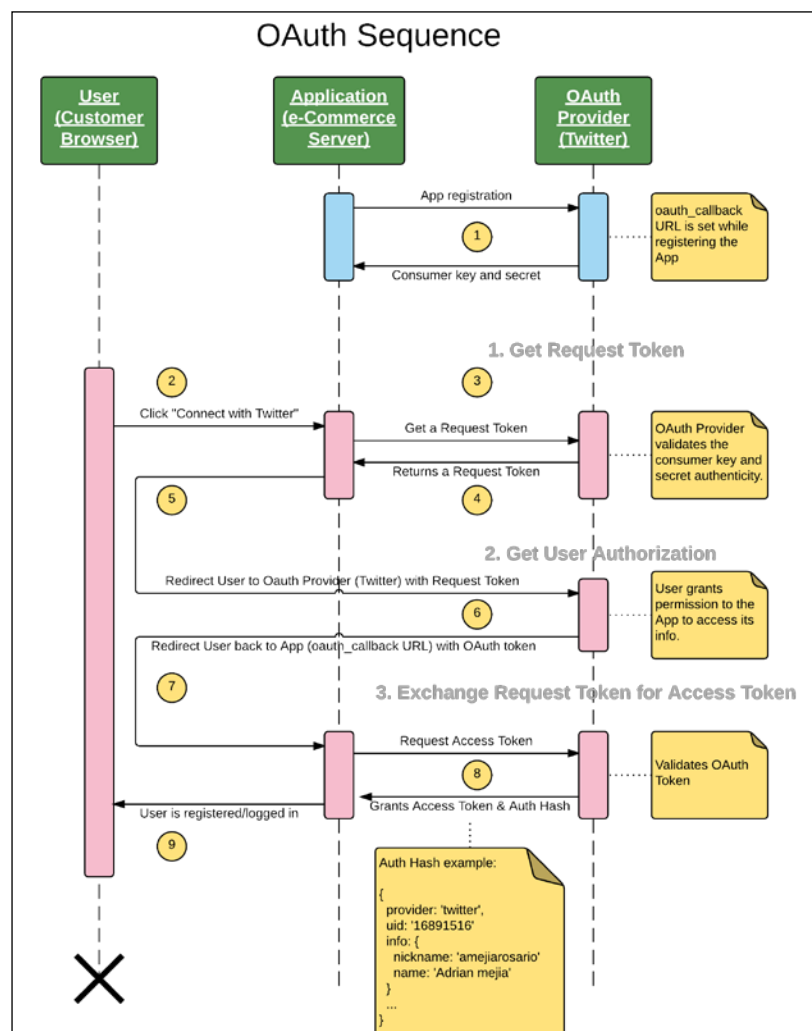


Figure 3: Three-legged OAuth authentication sequence diagram

This is how it works:

1. Ahead of time, the application needs to get registered with the OAuth Provider (in this case, Twitter) to get its API credentials: **consumer key** and **secret**. Moreover, it needs to set a *callback URL* to redirect the users back to the application.
2. The user has to log in/register by clicking the **Connect with Twitter** button.
3. The app asks for a **Request Token** from the OAuth provider. Along with the request, the app sends its consumer key and secret to get properly identified.
4. The provider (Twitter) verifies the consumer key, secret and request URL; if everything matches with the registered records, it grants a Request Token.
5. The app saves the Request Token, and redirects the user to the provider website.
6. The provider website prompts the user to provide authorization.
7. Once the user has authorized the OAuth provider to share data with the app, the provider redirects the user back to the application using the callback URL along with the `oauth_verifier`.
8. The app uses the `oauth_verifier` to exchange the Request Token for an **Access Token**. The provider grants the Access Token along with the user's profile data.
9. At this point, the user is registered/logged in the app with the profile data fetched from the OAuth Provider, and a session ID is created.
10. From that time on, all the requests from the user to the app will use the session ID.

This is one of the most common scenarios for OAuth, and the one that we are going to use in this project.

## Understanding client-side authentication

First of all, we are going to use token-based authentication for both local users (e-mail/password) and social logins (OAuth). If the users choose to log in through a social network, it is done through OAuth. However, once the identity of the user is confirmed, the rest of the process is token-based.

Authentication is one of the most important parts of our e-commerce application. It not only allows the users to log in, sign up, sign in and sign out, but also allows the app to keep a track of sellers and buyers of each product. Next, we are going to explore how the Angular client exercises the API to save new users/products to the database and to interact with the backend authentication mechanisms.

On the client-side, the bulk of the authentication code is contained in the folders `app/account` and `app/admin`.

## Authentication management

The `account` folder contains all the login and signup forms with its controllers. The pages that require authentication are marked with `authenticate: true`:

```
/* client/app/account/account.js *excerpt */

angular.module('meanstackApp')
  .config(function ($stateProvider) {
    $stateProvider
      .state('login', {
        url: '/login',
        templateUrl: 'app/account/login/login.html',
        controller: 'LoginCtrl'
      })
      .state('signup', {
        url: '/signup',
        templateUrl: 'app/account/signup/signup.html',
        controller: 'SignupCtrl'
      })
      .state('settings', {
        url: '/settings',
        templateUrl: 'app/account/settings/settings.html',
        controller: 'SettingsCtrl',
        authenticate: true
      });
  });
```

By itself, `authenticate: true` does nothing. In order to check if the user is authenticated, we need to intercept the route change event just before rendering the template. We are doing just that at the very bottom of `app.js`:

```
/* client/app/app.js */

.run(function($rootScope, $state, Auth) {
  // Redirect to login if route requires auth and the user is not
  logged in
  $rootScope.$on('$stateChangeStart', function(event, next) {
    if (next.authenticate) {
      Auth.isLoggedIn(function(loggedIn) {
        if (!loggedIn) {
          event.preventDefault();
          $state.go('login');
        }
      });
    }
  });
});
```

The other folder, `admin`, has the views where the administrator can delete other users from the system:

```
/* client/app/admin/admin.js *excerpt */

angular.module('meanstackApp')
.config(function ($stateProvider) {
  $stateProvider
    .state('admin', {
      url: '/admin',
      templateUrl: 'app/admin/admin.html',
      controller: 'AdminController'
    });
});
```

Finally, as we discussed in the section on token-based authentication, having a cookie with the token is not what matters; what is required is having the token in the HTTP header. Instead of doing that manually on every `$http` call, we can do it in `app.js` using an Angular interceptor, as shown in the following code:

```
/* client/app/app.js *excerpt */

.factory('authInterceptor', function ($rootScope, $q, $cookieStore,
  $location) {
```

---

```

return {
  // Add authorization token to headers
  request: function (config) {
    config.headers = config.headers || {};
    if ($cookieStore.get('token')) {
      config.headers.Authorization = 'Bearer ' + $cookieStore.
        get('token');
    }
    return config;
  },

  // Intercept 401s and redirect you to login
  responseError: function(response) {
    if(response.status === 401) {
      $location.path('/login');
      // remove any stale tokens
      $cookieStore.remove('token');
      return $q.reject(response);
    }
    else {
      return $q.reject(response);
    }
  }
};
})
.config(function ($stateProvider, $urlRouterProvider,
  $locationProvider, $httpProvider) {
  $urlRouterProvider
    .otherwise('/');

  $locationProvider.html5Mode(true);
  $httpProvider.interceptors.push('authInterceptor');
})

```

The highlights of the preceding code are as follows:

- If there's a cookie with a token, we use it to set the authentication HTTP header
- If the token used for authentication is invalid, we discard the cookie with the token and redirect to the login page
- We use interceptors for handling the authentication before they are sent to the server and intercept the responses before they are handled by the application



In-depth discussions about interceptors and promises are outside the scope of this book. More information is available at [https://docs.angularjs.org/api/ng/service/\\$q](https://docs.angularjs.org/api/ng/service/$q) and [https://docs.angularjs.org/api/ng/service/\\$http#interceptors](https://docs.angularjs.org/api/ng/service/$http#interceptors).

## The signing up process

When we click on register or go to the path `/signup`, the template is displayed with the form, and `signup.controller.js` injects the values:

```
/* client/app/account/signup/signup.controller.js */

angular.module('meanstackApp')
  .controller('SignupCtrl', function ($scope, Auth, $location,
    $window) {
    $scope.user = {};
    $scope.errors = {};

    $scope.register = function(form) {
      $scope.submitted = true;

      if(form.$valid) {
        Auth.createUser({
          name: $scope.user.name,
          email: $scope.user.email,
          password: $scope.user.password
        })
        .then( function() {
          // Account created, redirect to home
          $location.path('/');
        })
        .catch( function(err) {
          err = err.data;
          $scope.errors = {};

          // Update validity of form fields that match the mongoose
          errors
          angular.forEach(err.errors, function(error, field) {
            form[field].$setValidity('mongoose', false);
            $scope.errors[field] = error.message;
          });
        });
      }
    };
  });
```

```

    }
  };

  $scope.loginOAuth = function(provider) {
    $window.location.href = '/auth/' + provider;
  };
});

```

This controller, along with the `signup.html` form, allows users to register either by using the name/e-mail/password form or through a social network provider.

Sign up/in is possible thanks to the `user` and `auth` services that define all the methods related to authentication on the frontend. They are located in the `client/components` directory:

```

/* client/components/auth/auth.service.js *excerpt */

angular.module('meanstackApp')
  .factory('Auth', function Auth($location, $rootScope, $http,
    User, $cookieStore, $q) {
    var currentUser = {};
    if($cookieStore.get('token')) {
      currentUser = User.get();
    }
    return {
      login: function(user, callback) {
        var cb = callback || angular.noop;
        var deferred = $q.defer();

        $http.post('/auth/local', {
          email: user.email,
          password: user.password
        }).
        success(function(data) {
          $cookieStore.put('token', data.token);
          currentUser = User.get();
          deferred.resolve(data);
          return cb();
        }).
        error(function(err) {
          this.logout();
          deferred.reject(err);
          return cb(err);
        }).bind(this);
      }
    };
  });

```



```
    return deferred.promise;
  },
  logout: function() {
    $cookieStore.remove('token');
    currentUser = {};
  },
  createUser: function(user, callback) {
    var cb = callback || angular.noop;

    return User.save(user,
      function(data) {
        $cookieStore.put('token', data.token);
        currentUser = User.get();
        return cb(user);
      },
      function(err) {
        this.logout();
        return cb(err);
      }).bind(this)).$promise;
  },
  changePassword: function...,
  getCurrentUser: function...,
  isLoggedIn: function...,
  isAdmin: function...,
  getToken: function...
}
```

A few things to highlights from this service are as follows:

- `$cookieStore.get/set` is used to manipulate the cookies and to store the JSON Web Token in it
- Both, `Login` and `createUser`, set the `$cookieStore` parameter of the logged user, while `logout` removes the created cookie

Finally, let's take a look at the user factory:

```
/* client/components/auth/user.service.js */

angular.module('meanstackApp')
  .factory('User', function ($resource) {
    return $resource('/api/users/:id/:controller', {
      id: '@_id'
    },
    {
    }
  });
```

```
    changePassword: {
      method: 'PUT',
      params: {
        controller: 'password'
      }
    },
    get: {
      method: 'GET',
      params: {
        id: 'me'
      }
    }
  }
});
```

`User.get()` is very often called in the Auth service. This method sets the ID to `me` for generating a called `GET /api/users/me`. The server identifies the user `me` by the information contained in the token.

Now, let's go to the server side and complete our understanding of the authentication process.

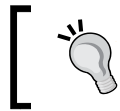
## Understanding server-side authentication

Some of the **node modules (npm)** that we are using in this chapter are as follows:

- `Passport`: Authentication module for NodeJS
- `Passport-local`: Username and password authentication strategy for Passport and Node.js
- `Passport-facebook`: Facebook authentication strategy
- `Passport-twitter`: Twitter authentication strategy
- `Passport-google-oauth`: Google (OAuth) authentication strategies
- `Jsonwebtoken`: Implementation of the JSON Web Token (JWT) standard. It is used to send encrypted JSON with sensitive data
- `Express-jwt`: Express middleware to authenticate HTTP requests using **JWT (JSON Web Token)** and sets `req.user`

## Authentication with PassportJS

PassportJS is an npm module that can be added to the ExpressJS middlewares. It uses sessions to provide the user login functionality, and supports more than 300 authentication strategies. It also supports single sign-on through OAuth, SAML, and JWT to mention a few. The use of authentication strategies extends from the most popular social networks to sports and health portals such as Fitbit and Human API to art galleries like deviantArt and federated authentication using OpenID. In case you do not find a strategy, PassportJS gives you the option to create your own.



It's recommended to check out Passport to get a deeper understanding of the module at <http://passportjs.org/docs>.

## Initializing PassportJS

If you run `grunt serve` again, you will see that we already have the sign up and login links. We can sign up, sign in, and log out with an e-mail/password. However, social logins won't work. Let's get through the functionality and fix the single sign-on strategies.

When we run `grunt serve` or `npm start`, it executes `node server/app.js`. As we have seen before, this script is the one that bootstraps the rest of the application. The most significant portions are where the ExpressJS app is set up and the routes are defined:

```
/* server/app.js *excerpt */

var app = express();
var server = require('http').createServer(app);
// ...
require('./config/express')(app);
require('./routes')(app);
```

The `express.js` file initializes the PassportJS, and sets up the session:

```
/* server/config/express.js *excerpt */

var session = require('express-session');
var mongoStore = require('connect-mongo')(session);
var mongoose = require('mongoose');
var passport = require('passport');

app.use(passport.initialize());
```

---

```
// Persist sessions with mongoStore / sequelizeStore
// We need to enable sessions for passport twitter because
// its an oauth 1.0 strategy

app.use(session({
  secret: config.secrets.session,
  resave: true,
  saveUninitialized: true,
  store: new mongoStore({
    mongooseConnection: mongoose.connection,
    db: 'meanshop'
  })
}));
```

The first step for enabling Passport is to add it to the middleware chain using `passport.initialize()`.

Next, we use the `express-session` middleware to set up our sessions. Only the session ID is stored in a cookie and not the session data. The ID is used to retrieve the data stored on the server side. Let us take a look at the meaning of the parameters passed:

- **Secret:** This is a string that is used to generate the session ID cookie.
- **Resave:** This forces the session to always save back to the server even if it has not been modified.
- **SaveUninitialized:** This is a session which is uninitialized when it is new. This option forces the session to be saved even when it is new and unmodified.
- **Store:** This is where the session data is saved. By default, it is stored in the memory (`MemoryStore`). However, we are using MongoDB instead with `connect-mongo`. Another popular storage option is (`connect-redis`) Redis.



`MemoryStore` and `MongoStore` are two different strategies for saving sessions. In the first one, the sessions data is stored in the memory of the server. That can cause scaling across multiple servers, since memory is separate on each server. Furthermore, it can consume a significant amount of memory if there are many concurrent users. On the other hand, `MongoStore`, is a better solution. Sessions can scale as we scale the database, and it is shared across multiple servers.

Now that we have properly initialized PassportJS, we need to map it to the routes, and add strategies to make it functional. Let's now explore how we use the User model to provide authentication.

## The user model

The following is an excerpt of the user model. It comes preloaded with all the methods that we need for authentication and validations:

```
/* server/api/user/user.model.js *excerpt */

var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var crypto = require('crypto');
var authTypes = ['github', 'twitter', 'facebook', 'google'];

var UserSchema = new Schema({
  name: String,
  email: { type: String, lowercase: true },
  role: {
    type: String,
    default: 'user'
  },
  hashedPassword: String,
  provider: String,
  salt: String,
  facebook: {},
  twitter: {},
  google: {},
  github: {}
});

UserSchema
  .virtual('password')
  .set(function(password) {
    this._password = password;
    this.salt = this.makeSalt();
    this.hashedPassword = this.encryptPassword(password);
  })
```

```
.get(function() {
  return this._password;
});

UserSchema.methods = {
  authenticate: function(plainText) {
    return this.encryptPassword(plainText) === this.hashPassword;
  },

  makeSalt: function() {
    return crypto.randomBytes(16).toString('base64');
  },

  encryptPassword: function(password) {
    if (!password || !this.salt) return '';
    var salt = new Buffer(this.salt, 'base64');
    return crypto.pbkdf2Sync(password, salt, 10000, 64).
      toString('base64');
  }
};
```

A plain text password should never be saved to the database. Thus, we are always saving the encrypted version of the password. Furthermore, a `salt` parameter is added to the password encryption mechanism for extra security.



#### Password salt

The purpose of adding a salt is to protect the users with a simple password (dictionary words). Salt is random data generated to be used along with the password in the one-way hashing function. The end result is an encrypted password, which is the one that is stored in the database.

## Authentication strategies and routes

We are going to use four strategies for our application: local (e-mail/password), Facebook, Twitter, and Google. The following screenshot displays the file structure organization:

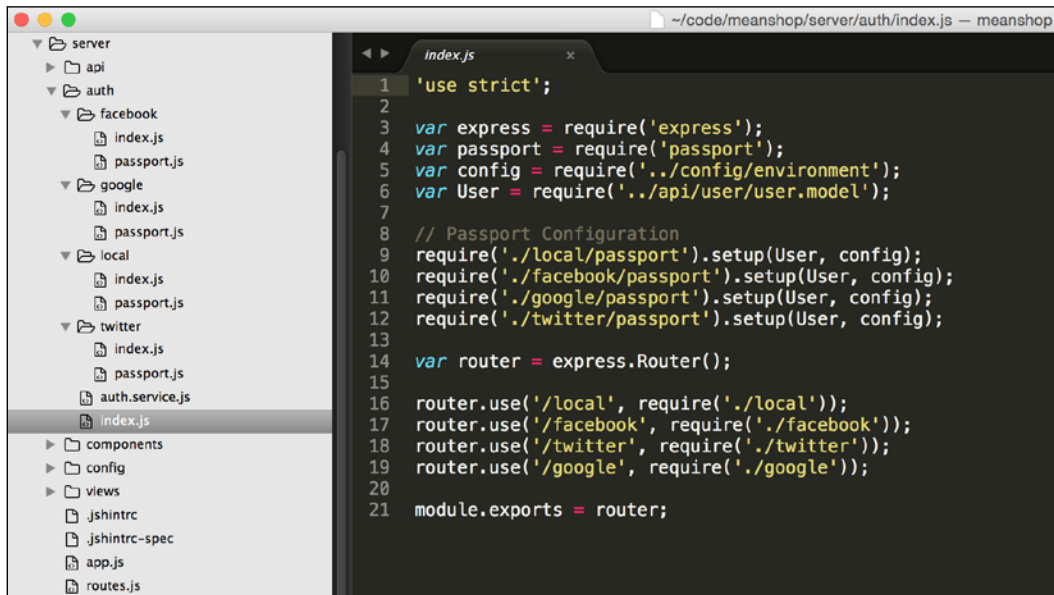


Figure 4: File structure to organize authentication strategies

They are all contained inside the auth folder, and have two files inside:

- `index.js`: This file defines the routes
- `passport.js`: This file defines the strategies

To begin with the routes, open `routes.js`. All the authentication strategies will be mapped under the `/auth` path:

```
/* server/routes.js *excerpt */  
  
app.use('/api/products', require('./api/product'));  
app.use('/api/users', require('./api/user'));  
app.use('/auth', require('./auth'));
```

When we expand `./auth` routes, we see that we have one for any strategy that we want to support: local, Facebook, Twitter, and Google:

```
/* server/auth/index.js *excerpt */

router.use('/local', require('./local'));
router.use('/facebook', require('./facebook'));
router.use('/twitter', require('./twitter'));
router.use('/google', require('./google'));
```

To sum up, the preceding code will yield the following routes:

- **Local authentication:** `/auth/local`
- **Facebook authentication:** `/auth/facebook`
- **General path:** `/auth/:provider`

We are going to focus only on the local strategy first to drive the concepts home, and then explain the social strategies.

## Local authentication

Setting up local authentication or any other kind of authentication involves two steps:

1. Defining the strategy with `passport.use`:

```
/* server/auth/local/passport.js *excerpt */

exports.setup = function(User, config) {
  passport.use(new LocalStrategy({
    usernameField: 'email',
    passwordField: 'password' // this is the virtual field
                             on the model
  }, function(email, password, done) {
    return localAuthenticate(User, email, password, done);
  }));
};
```



2. Adding the strategy to the route with `passport.authenticate`:

```
/* server/auth/local/index.js *excerpt */

router.post('/', function(req, res, next) {
  passport.authenticate('local', function (err, user, info) {
    var error = err || info;
    if (error) return res.json(401, error);
    if (!user) return res.json(404, {message: 'Something went
      wrong, please try again.'});

    var token = auth.signToken(user._id, user.role);
    res.json({token: token});
  })(req, res, next)
});
```

`passport.use` defines the method to validate that the e-mail/password combination is correct. This is done by finding the user by e-mail, and then encrypting the plain text password and verifying if it matches with the one in the database. If the e-mail is found and there's a match in the password, then the user is authenticated; otherwise, it throws an error.

`passport.authenticate` is used to associate an authentication strategy with a route. In this particular case, we pull out the `local` strategy that we defined previously in the previous step.

## End-to-end tests for local authentication

To make sure that users can sign in and out at all times, we are going to add automated testing. Our Yeoman generator gets us started quickly. Review the following directories: `e2e/account/signup/signup.spec.js`, `e2e/account/login/login.spec.js`, and `e2e/account/logout/logout.spec.js`. In this set of tests, we are also testing that we can sign up, log in, and log out.

## Authenticating with Facebook, Google, and Twitter

Adding new social strategies is very similar to what we did with the local strategy. The main difference with the social strategies is that we need to register our app on each of the platforms, and of course, have an account on each one of them.

meanshop

Dashboard

Settings

Status & Review

App Details

Roles

Open Graph

Alerts

Localize

Canvas Payments

Audience Network

Test Apps

Analytics

BasicAdvancedMigrations

App ID

App Secret

Display Name

Namespace

App Domains

Contact Email

Website

Site URL

+ Add Platform

Delete App

Discard

Save Changes

Once you have the ID and secret, go to `local.env.js`, and fill it out:

- [ 119 ] -

```
    TWITTER_ID: 'app-id',
    TWITTER_SECRET: 'secret',

    GOOGLE_ID: 'app-id',
    GOOGLE_SECRET: 'secret',
  };
```

Now, restart the server and try it out. It should work but the release **3.0.0-rc4** has a bug. Let's fix it by removing `JSON.stringify` in *line 81*:

```
/* server/auth/auth.service.js:81 *excerpt */
function setTokenCookie(req, res) {
  if (!req.user) {
    return res.status(404).send('Something went wrong, please try
      again.');
```

```
    again.');
```

```
  }
```

```
  var token = signToken(req.user._id, req.user.role);
```

```
  res.cookie('token', token);
```

```
  res.redirect('/');
```

```
}
```

In the `facebook/passport.js` (`passport.use`), this time we look up the users by `facebook.id` instead of e-mail:

```
passport.use(new FacebookStrategy({
  clientID: config.facebook.clientID,
  clientSecret: config.facebook.clientSecret,
  callbackURL: config.facebook.callbackURL
}),
function(accessToken, refreshToken, profile, done) {
  User.findOne({
    'facebook.id': profile.id
  },
  function(err, user) {
    if (err) {
      return done(err);
    }
    if (!user) {
      user = new User({
        name: profile.displayName,
        email: profile.emails[0].value,
        role: 'user',
        username: profile.username,
        provider: 'facebook',
        facebook: profile._json
      });
    }
  });
});
```

```

    user.save(function(err) {
      if (err) done(err);
      return done(err, user);
    });
  } else {
    return done(err, user);
  }
}
}
));

```

We get the Facebook data in the `profile` variable, and use it to populate all of the user's information.

Finally, we define the route using `passport.authenticate('facebook', ...)`. The scope and callback were not defined in the local strategy. The scope refers the pieces of Facebook data that we want to get from the users. The callback is the route that handles the data once the user is redirected from the Facebook login form.

## Twitter

Let's go to the Twitter Developers' site at <https://apps.twitter.com>, and get our app registered. This time, use the following callback URL: `http://127.0.0.1:9000/auth/twitter/callback`.

The screenshot shows the Twitter Developer Console for an application named "meanshop". The interface includes tabs for "Details", "Settings", "Keys and Access Tokens", and "Permissions". The "Settings" tab is active, showing the application's profile picture (a blue Twitter bird), name "the meaningful shop", and callback URL "http://127.0.0.1:9000". Below this is the "Organization" section, which is currently empty. The "Application Settings" section shows the "Access level" as "Read and write", the "Consumer Key (API Key)" as a redacted value, the "Callback URL" as "http://127.0.0.1:9000/auth/twitter/callback", and the "Sign in with Twitter" option as "Yes".

Organization	
Organization	None
Organization website	None

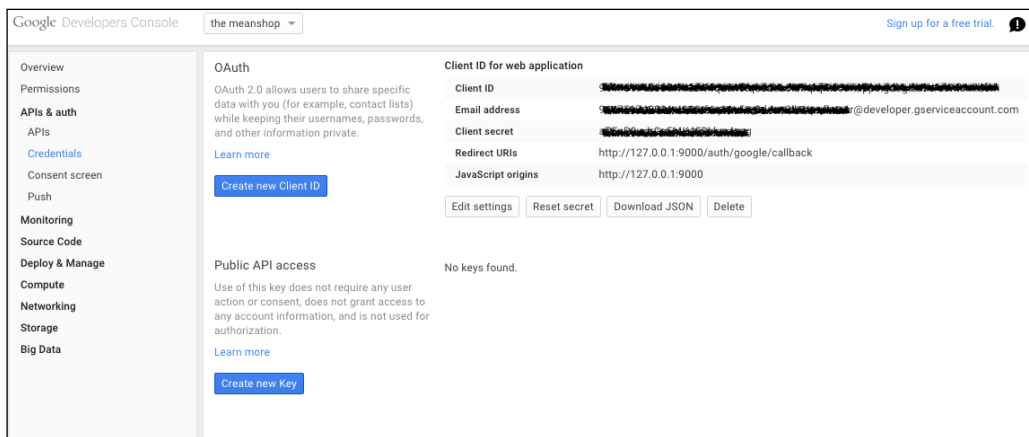
Application Settings	
Access level	Read and write (modify app permissions)
Consumer Key (API Key)	[REDACTED] (manage keys and access tokens)
Callback URL	http://127.0.0.1:9000/auth/twitter/callback
Sign in with Twitter	Yes

Furthermore, you need to go to the settings tab and enable **Sign in with Twitter**.

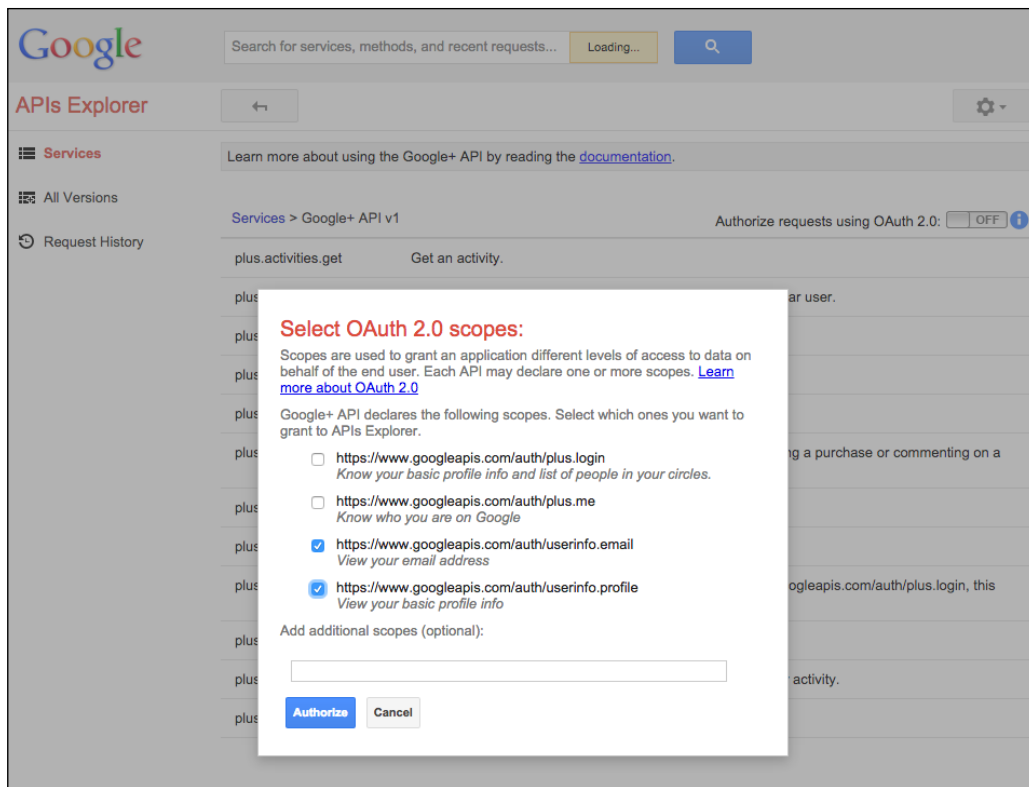
Then, fill out the App ID and secret in `local.env.js`. Take a look at `twitter/passport.js` and `twitter/index.js`. They are almost identical to the ones in Facebook. The main difference is that Twitter doesn't have scopes.

## Google

Create our app at <https://console.developers.google.com/project>. Use the URL and callback using `127.0.0.1` or `localhost` as shown in the following image:



1. Furthermore, you need to go to **API & auth | APIs**, and enable **Google+ API**.
2. Later, click on **Explore this API**, and enable **Authorize requests using OAuth 2.0**, as shown in the following screenshot:



Again, fill out the `local.env.js` with it, and take a look at the `google/passport.js` and `google/index.js`.

That's it! We just learnt how to set up the app with several social network providers.

## Summary

In this chapter, we have learnt how to allow users to log in using the traditional username and password method and also through their social networks. We explained the difference between authentication and authorization. Authentication dictates if a user can log in or not into the system, while authorization defines what a user is allowed to do or see while he/she are logged into the system.

This chapter is very important, and it will enable us to add more interesting features such as product checkout, order creation, and history by user. Read on, that's what we are doing next!

# 7

## Checking Out Products and Accepting Payment

Shopping carts and checkouts are the most important feature of an e-commerce website. In this chapter, we are going to integrate a shopping cart, and allow users to checkout their products.

Furthermore, we will go through the process of accepting payments securely. We are going to explain how PayPal works and how it can be integrated for accepting payments.

The following topics will be covered in this chapter:

- Setting up the shopping cart
- Creating an order
- Checking out products

### Setting up the shopping cart

One of the main advantages of developing on open source technology is that we don't have to reinvent the wheel with every feature. We first look if there's any module that already does what we want to accomplish, before we start it ourselves, from scratch. That's true in the case of the shopping cart; we are going to use the already available module: **ngCart**.



## Installing ngCart

The ngCart module provides the following directives and services that we need to get started with the shopping cart:

1. It renders an **Add to cart** button: `<ngcart-addtocart id="{{item.id}}" name="{{item.name}}" price="{{item.price}}"></ngcart-addtocart>`.
2. Renders a shopping cart: `<ngcart-cart></ngcart-cart>`.
3. Shows the cart summary: `<ngcart-summary></ngcart-summary>`.
4. Renders the checkout buttons for PayPal and other HTTP services: `<ngcart-checkout service="http" settings="{url:'/checkout'}"></ngcart-checkout>`.

We are going to use bower to quickly install ngCart in our project:

```
bower install ngcart#1.0.0 --save
```

There is no need to worry about adding it to `index.html`; a grunt task (`grunt-wiredep`) will inject it for us. Let's go ahead and load it with the rest of our AngularJS dependencies in `app.js`:

```
/* client/app/app.js *excerpt */

angular.module('meanshopApp', [
  'ngCookies',
  'ngResource',
  'ngSanitize',
  'btford.socket-io',
  'ui.router',
  'ui.bootstrap',
  'ngFileUpload',
  'ngCart'
])
```

The ngCart directives require some templates for its directives. We need to copy them to our project:

```
cp -R client/bower_components/ngcart/template/ngCart client/components/ngcart
```

Every time we use any of the aforementioned directives, they are going to be translated to their matching template:

- `client/components/ngcart/addtocart.html`
- `client/components/ngcart/cart.html`
- `client/components/ngcart/checkout.html`
- `client/components/ngcart/summary.html`

So every time we want to customize the look and feel, this is the place to go.

## Making use of ngCart directives

We are going to make use of four directives/templates in our project, and create a new checkout page that shows the shopping cart and adds the checkout strategies.

### Add/remove to cart

First, let's replace the **Buy** button for the **Add to cart** button in the products listing page (`product-list.html` line 14). Replace the following code:

```
<a href="#" class="btn btn-primary" role="button">Buy</a>
```

With the following code:

```
<!-- client/app/products/templates/product-list.html:14 -->

<ngcart-addtocart id="{{product._id}}" quantity="1" quantity-
max="9" name="{{product.title}}" price="{{product.price}}" template-
url="components/ngcart/addtocart.html" data="product">Add to Cart</
ngcart-addtocart>
```

Do the same in `product-view.html`; replace the **buy** button for the `ngcart-addtocart` directive. Now, let's change the default template:

```
<!-- client/components/ngcart/addtocart.html -->

<span ng-hide="attrs.id">
  <a class="btn btn-primary" ng-disabled="true" ng-transclude></a>
</span>
<span ng-show="attrs.id">
  <span ng-show="!inCart()">
    <span ng-show="quantityMax">
      <select name="quantity" id="quantity" ng-model="q" ng-options="
v for v in qtyOpt"></select>
    </span>
```

```
        <a class="btn btn-primary" ng-click="ngCart.addItem(id, name,
price, q, data)" ng-transclude></a>
    </span>
    <mark ng-show="inCart()">
        <a class="btn btn-success" ng-click="ngCart.
removeItemById(id)">Added</a>
    </mark>
</span>
```

With this new template, we only need one button. If you click once, we add the items to the cart, and if we click again, we remove it from the cart. Pretty simple! Moreover, we have a dropdown menu to select the number of items we want at a time.

## The cart's summary

Next, let's add a summary of the checkout cart in `navbar.html`, just before line 26:

```
<!-- client/components/navbar/navbar.html:26 | excerpt -->

<li>
    <a ui-sref="checkout">
        <ngcart-summary template-url="components/ngcart/summary.html"></ngcart-summary>
    </a>
</li>
```

Notice that we are linking the cart summary to the *checkout* state. Let's create that state:

```
/* client/app/products/products.js:30 */

.state('checkout', {
  url: '/checkout',
  templateUrl: 'app/products/templates/products-checkout.html',
  controller: 'ProductCheckoutCtrl'
});
```

The next step is to adjust the `summary.html` template for a better look and feel. Replace the whole content of the template for these few lines:

```
/* client/components/ngcart/summary.html */

<span class="fa fa-shopping-cart"></span>
{{ ngCart.getTotalItems() }} /
{{ ngCart.totalCost() | currency }}
```



You can run `grunt serve` to verify that you can add/remove products from the cart. Items should be reflected in the summary at the top-right corner.

## The checkout page and Braintree integration

We are going to receive payments from two sources: PayPal and credit/debit cards. We are going to use the Braintree services to collect payments from both sources. For that, we are going to use `braintree-angular`. Let's install it:

```
bower install braintree-angular#1.3.1 --save
```

Then add it to our app dependencies:

```
angular.module('meanshopApp', [
  'ngCookies',
  'ngResource',
  'ngSanitize',
  'btford.socket-io',
  'ui.router',
  'ui.bootstrap',
  'ngFileUpload',
  'ngCart',
  'braintree-angular'
])
```

Now that we have the route, let's create the template and the controller. The following is the content for the template:

```
<!-- client/app/products/templates/products-checkout.html -->

<navbar></navbar>

<div class="container">
  <div class="row">
    <h1>Cart</h1>
    <ngcart-cart template-url="components/ngcart/cart.html"></ngcart-
cart>
  </div>

  <div class="row">
    <h1>Choose a payment method</h1>
```


```
<p class="help-block">{{ errors }}</p><br>
<div class="col-md-6">
  <form>
    <braintree-dropin options="paymentOptions">Loading...</
braintree-dropin>
    <input type="submit" value="Purchase" />
  </form>
</div>
</div>
</div>

<footer></footer>
```

We are using the `braintree-dropin` directive to render the default card forms and the PayPal button. This form is going to take care of the validations and security.

Braintree has a workflow different from the other payment platforms. This is how it is going to work in our app:

1. Angular app will request a *client token* from our node backend.
2. The node backend will use the *Braintree SDK* to generate a client token and reply with it.
3. The Angular app will use the client token and the PayPal/card information to get authorization from the Braintree servers. The Braintree servers will generate a *nonce*, which is sent back to the Angular app.
4. The Angular app will then send the *nonce* to the node backend to finish executing the payment operation.

[  Refer to <https://developers.braintreepayments.com/javascript+node/start/overview> for more detailed information. ]

All the Angular steps are going to be added in `products.controller.js`:

```
/* client/app/products/products.controller.js:35 */

.constant('clientTokenPath', '/api/braintree/client_token')
```

```
.controller('ProductCheckoutCtrl',
  function($scope, $http, $state, ngCart){
    $scope.errors = '';

    $scope.paymentOptions = {
      onPaymentMethodReceived: function(payload) {
        angular.merge(payload, ngCart.toObject());
        payload.total = payload.totalCost;
        console.error(payload);
        $http.post('/api/orders', payload)
          .then(function success () {
            ngCart.empty(true);
            $state.go('products');
          }, function error (res) {
            $scope.errors = res;
          });
      }
    };
  });
```

We add a constant with the path to get the client token. We are going to create that controller in our backend later.

In the scope, we defined a couple of things that we are already using in the checkout page such as errors and paymentOptions. In paymentOptions, we are adding a callback that is going to be executed when the user provides his/her credit card number or PayPal authorization. If the authorization is successful, the payload parameter gets a nonce parameter, which will be used to execute the payment on the NodeJS side. Notice that we are also serializing the shopping cart, and adding/merging it to the payload. Finally, we create an order in our backend using the payload information. If we are able to collect the payment in the backend, we redirect the user to the marketplace and empty the cart; otherwise, we present an error message.

If you run the application (`grunt serve`), you will notice that we can add and remove items from the shopping cart. However, when we enter our card data, we are unable to complete the transaction, as the route `/api/orders` is not created yet. Let's move to the backend and work on that next.

## Setting up Braintree endpoint and authentication

The generator has already created the required order routes and controller for use. We are going to roll with that. If you remember from our Angular app, we are using a route called `/api/braintree/client_token`, which we haven't created yet. For that we are going to use the Braintree SDK. Let's get that set up:

```
npm install braintree@1.29.0 --save
```

Next, let's generate the endpoint:

```
yo angular-fullstack:endpoint braintree
? What will the url of your endpoint be? /api/braintree
...
```

## The API keys

We need to get the API keys in order to use the Braintree services. Go to <https://www.braintreepayments.com/get-started> and create a sandbox account.

Once you have the private/public keys and the merchant ID, add it to `local.env.js`:

```
/* server/config/local.env.js */

BRAINTREE_ID: '7hh... public key ...rq',
BRAINTREE_SECRET: 'f1c... private key ...028',
BRAINTREE_MERCHANT: 'gwz... merchant ID ...g3m',
```

Later, let's create the config files to make use of it:

```
/* server/config/environment/index.js *excerpt */

braintree: {
  clientID: process.env.BRAINTREE_ID || 'id',
  clientSecret: process.env.BRAINTREE_SECRET || 'secret',
  clientMerchant: process.env.BRAINTREE_MERCHANT || 'merchant'
}
```

We have been using the same pattern for other services that need keys such as Facebook and Twitter. In the next section, we are going to make use of it.

## Gateway

Open `braintree.model.js`, and replace the content with the following code:

```
/* server/api/braintree/braintree.model.js */

var braintree = require('braintree');
var config = require('../../config/environment');
var isProduction = config.env === 'production';

var gateway = braintree.connect({
  environment: isProduction ? braintree.Environment.Production :
    braintree.Environment.Sandbox,
  merchantId: config.braintree.clientMerchant,
  publicKey: config.braintree.clientID,
  privateKey: config.braintree.clientSecret
});

module.exports = gateway;
```

This model is very different, since it doesn't store anything in MongoDB like the others. Instead, it sets up communication with a remote service in the Braintree servers.

## Controller

Now we can use the Braintree gateway in the controllers:

```
/*server/api/braintree/braintree.controller.js */

var _ = require('lodash');
var Braintree = require('./braintree.model');

function handleError(res, statusCode) {
  statusCode = statusCode || 500;
  return function(err) {
    res.status(statusCode).send(err);
  };
}

function handleResponse (res) {
  return function (err, result) {
    if(err) {
```



```
        return handleError(res)(err);
      }
      responseWithResult(res)(result);
    }
  }

function responseWithResult(res, statusCode) {
  statusCode = statusCode || 200;
  return function(entity) {
    if (entity) {
      res.status(statusCode).json(entity);
    }
  };
}

exports.clientToken = function(req, res){
  Braintree.clientToken.generate({}, function (err, data) {
    return handleResponse(res)(err, data.clientToken);
  });
}

exports.checkout = function(req, res){
  Braintree.transaction.sale({
    amount: req.body.total,
    paymentMethodNonce: req.body.nonce,
  }, function callback (err, result) {
    if(err) {
      return handleError(res)(err);
    }
    if(result.success){
      responseWithResult(res)(result);
    } else {
      handleError(res)(result.errors);
    }
  });
}
```

## Router

We are using only two actions: `POST checkout` and `GET clientToken`. Let's define them:

```
/* server/api/braintree/index.js */

var express = require('express');
var controller = require('./braintree.controller');

var router = express.Router();

router.get('/client_token', controller.clientToken);
router.post('/checkout', controller.checkout);

module.exports = router;
```

Go ahead and delete everything else that the generator did for the Braintree endpoint; we are not going to use that. The only three required files are the Braintree model, controller, and index.

## Creating an order

So far, users can add/remove products from the cart, and go to a checkout page. Now we need to implement the actual checkout functionality. Using `HTTP POST`, we can send all the cart information to the backend to create an order. Let's create a new API call for the orders:

```
yo angular-fullstack:endpoint order

? What will the url of your endpoint be? /api/orders
  create server/api/order/order.controller.js
  create server/api/order/order.events.js
  create server/api/order/order.integration.js
  create server/api/order/order.model.js
  create server/api/order/order.socket.js
  create server/api/order/index.js
  create server/api/order/index.spec.js
```

The preceding command will create all the scaffolding code needed to CRUD orders.

## Modifying the order model

Let's modify the order model with the fields that we need:

```
/* server/api/order/order.model.js */

var _ = require('lodash');
var mongoose = require('bluebird').promisifyAll(require('mongoose'));
var Schema = mongoose.Schema;
var Braintree = require('../braintree/braintree.model');

var OrderDetailsSchema = new Schema({
  product: { type: Schema.Types.ObjectId, ref: 'Product' },
  quantity: Number,
  total: { type: Number, get: getPrice, set: setPrice }
});

var OrderSchema = new Schema({
  // buyer details
  name: String,
  user: { type: Schema.Types.ObjectId, ref: 'User' },
  shippingAddress: String,
  billingAddress: String,
  // price details
  items: [OrderDetailsSchema],
  shipping: { type: Number, get: getPrice, set: setPrice, default: 0.0 },
  tax: { type: Number, get: getPrice, set: setPrice, default: 0.0 },
  discount: { type: Number, get: getPrice, set: setPrice, default: 0.0 },
  subTotal: { type: Number, get: getPrice, set: setPrice },
  total: { type: Number, get: getPrice, set: setPrice, required: true },
  // payment info
  status: { type: String, default: 'pending' }, // pending, paid/
    failed, delivered, canceled, refunded.
  paymentType: { type: String, default: 'braintree' },
  paymentStatus: Schema.Types.Mixed,
  nonce: String,
  type: String
});

// execute payment
OrderSchema.pre('validate', function (next) {
  if(!this.nonce) { next(); }
```

---

```

    executePayment(this, function (err, result) {
      this.paymentStatus = result;
      if(err || !result.success){
        this.status = 'failed. ' + result.errors + err;
        next(err || result.errors);
      } else {
        this.status = 'paid';
        next();
      }
    }).bind(this));
  });

function executePayment(payment, cb){
  Braintree.transaction.sale({
    amount: payment.total,
    paymentMethodNonce: payment.nonce,
  }, cb);
}

function getPrice(num){
  return (num/100).toFixed(2);
}

function setPrice(num){
  return num*100;
}

module.exports = mongoose.model('Order', OrderSchema);

```

There's a lot going on here. But, let's explain it step by step:

- Notice that the fields `product` and `user` use the `ObjectId` type with a `ref` attribute. That way, we can reference the objects in other collections. We are also using sub-documents, which is a feature of `mongoose`.
- `Items` has a direct reference to `OrderDetailSchema`. We are not creating a new collection for it; we are just embedding the order details into the order collection.
- Unfortunately, `mongoose` lacks a currency type, so we simulate our own. Using `get` and `set`, we can store the prices in integers rather than floating points, and avoid round-up errors.
- We use the Braintree services to execute the payment before saving the order.



Refer to <http://mongoosejs.com/docs/subdocs.html> to find more information about working with sub-documents.

## Testing the order model

Our order model has some required fields such as the total of the order. We will test whether all the other attributes can be saved correctly as well:

```
/* server/api/order/order.model.spec.js */

var Order = require('./order.model');
var Product = require('../product/product.model');
var User = require('../user/user.model');

describe('Order', function() {
  beforeEach(cleanDb);
  after(cleanDb);

  describe('#create', function() {
    var products,
        products_attributes = [
          {title: 'Product 1', price: 111.11 },
          {title: 'Product 2', price: 2222.22 },
        ],
        user = new User({
          provider: 'local',
          name: 'Fake User',
          email: 'test@test.com',
          password: 'password'
        });

    beforeEach(function (done) {
      Product.create(products_attributes, function (err, data) {
        if(err) return done(err);
        products = data;
        return user.save();
      }).then(function () {
        done();
      }, done);
    });
  });
});
```

---

```

    it('should create an order with valid attributes', function(done)
    {
        var attributes = {
            products: products.map(function(p){ return p._id; }),
            user: user._id,
            total: products.reduce(function(p, c) { return p.price +
                c.price; }),
        };

        Order.create(attributes).then(function (results) {
            return Order.findOne({}).populate(['products', 'user']);
        }).then(function(order) {
            order.products.length.should.be.equal(2);
            order.total.should.be.equal(111.11+2222.22);
            order.shipping.should.be.equal(0.0);
            order.tax.should.be.equal(0.0);
            order.discount.should.be.equal(0.0);
            done();
        }).then(null, done);
    });

    it('should not create an Order without total', function(done)
    {
        var invalid_attributes = {
            items: products.map(function(p){ return p._id; }),
            user: user._id,
        };

        Order.createAsync(invalid_attributes)
        .then(function (res) {
            done(new Error('Validation failed'));
        })
        .catch(function(err){
            err.should.not.be.null;
            err.message.should.match(/validation\ failed/);
            done();
        });
    });

    });

    function cleanDb(done){
        Order.remove().then(function () {

```

---

```
        return Product.remove();
    }).then(function () {
        return User.remove();
    }).then(function () {
        done();
    }).then(null, done);
}
```

In these tests, we verified that we can save the order with valid data, and that we cannot create orders without the total price amount.

## Using the sandbox account

Finally, we can test the workflow of order creation by running the application (grunt serve), as follows:

1. Add multiple products to the shopping cart.
2. Checkout the products using some valid credit card numbers for testing such as 4111 1111 1111 1111 or 4242 4242 4242 4242.
3. Any expiration date in the future will work.

You can review all these changes at <https://github.com/amejiarosario/meanshop/compare/ch6...ch7>.

## Summary

In this chapter, we learnt how to set up a shopping cart in AngularJS, leveraging open source packages such as ngCart. We learnt how to use Braintree SDK for NodeJS to accept payments, and to set up the Angular app to render payment forms. We went through the different stages and redirections that Braintree payment requires.

In the next chapter, we are going to add navigation and search for the products.

# 8

## Adding Search and Navigation

As an e-commerce application grows, the number of products becomes too large for users to find what they are looking for. Navigation allows users to filter products, while search allows them to find exactly what they are looking for.

In this chapter, we are going to focus on some UX/UI enhancements such as adding categories to the products and search capabilities. So, we are going to cover the following topics in this chapter:

- Adding search to the navigation bar
- Adding product categories
- Building navigation menus
- Implementing search functionality

### Adding search to the navigation bar

One of the quickest ways to find exactly what the user is looking for is through the search textbox. We are going to add one to the main navigation bar inside `navbar.html`, after line 20:

```
<!-- client/components/navbar/navbar.html:20 - excerpt -->

<!-- search bar -->
<form class="navbar-form navbar-left" role="search">
  <div class="form-group">
    <div class="input-group">
      <span class="input-group-addon"><span class="fa fa-search"></
span></span>
```



```
        <input id="searchBox" type="text" class="form-control
            input-large" placeholder="Search" ng-model="searchTerm"
            ng-change="search()" ng-focus="redirect()">
    </div>
</div>
</form>
```

We would like to broadcast an event when anything on the search form is typed. So, we add that in the navigation bar controller:

```
/* client/components/navbar/navbar.controller.js */
angular.module('meanshopApp')
.controller('NavbarCtrl', function ($scope, Auth, $rootScope,
    $state, $window, $timeout) {
    $scope.menu = [{
        'title': 'Home',
        'state': 'main'
    }, {
        'title': 'Products',
        'state': 'products'
    }
    ];

    $scope.isCollapsed = true;
    $scope.isLoggedIn = Auth.isLoggedIn;
    $scope.isAdmin = Auth.isAdmin;
    $scope.getCurrentUser = Auth.getCurrentUser;

    $scope.search = function () {
        $rootScope.$broadcast('search:term', $scope.searchTerm);
    };

    $scope.redirect = function () {
        $state.go('products');
        // timeout makes sure that it is invoked after any other event
        // has been triggered.
        $timeout(function () {
            // focus on search box
            var searchBox = $window.document.getElementById('searchBox');
            if(searchBox){ searchBox.focus(); }
        })
    };
});
```

After we add the `$rootScope` dependency, we can catch any change in the event on the search form and broadcast it. Later in this chapter, we are going to listen for such events, and perform the search within the categories.

Notice that `ng-focus` redirects to the products page as soon as the user focuses on the search bar. When the user starts typing the product name, the matching word shows up.

## Adding product categories

Product categories allow us to filter the products based on certain criteria, such as books, clothing, and so on. Hierarchies are another important feature to be added so that we can group the categories into sub-categories. For example, within the book category, we can have multiple sub-categories like non-fiction, novels, self-help, and so on.

## Adding the sidebar

We are now going to add a new sidebar to the marketplace that shows all the products. This sidebar will list all the products categories in the `product-list` page:

```
/* client/app/products/templates/product-list.html *excerpt */

<navbar></navbar>

<div class="container">
  <div class="row">
    <div class="col-md-3"
      ng-include="'components/sidebar/sidebar.html'"></div>
    <div class="col-sm-6 col-md-9"
      ng-show="products.length < 1"> No products to show.</div>
    <div class="col-sm-6 col-md-9" ng-repeat="product in products">
      <div class="thumbnail">
        <h3>{{product.title}}</h3>
        
        <figcaption class="figure-caption">
          {{product.description | limitTo: 100}} ...</figcaption>
        <div class="caption">
          <p>{{product.price | currency }}</p>
          <p>
```

```
        <ngcart-addtocart id="{{product._id}}"
            quantity="1" quantity-max="9" name="{{product.title}}"
            price="{{product.price}}" template-url="components/
ngcart/addtocart.html" data="product">Add to Cart</ngcart-addtocart>

        <a ui-sref="viewProduct({id: product._id})"
            class="btn btn-default" role="button">Details</a>
    </p>
</div>
</div>
</div>
</div>
</div>

<footer></footer>
```

We haven't defined it as yet, but the sidebar will contain the products listed according to their categories. Let's create the new sidebar components:

```
mkdir client/components/sidebar
touch client/components/sidebar/sidebar.{html,scss,controller.js,service.js}
```

Here is the content for the sidebar.html:

```
/* client/components/sidebar/sidebar.html */

<div ng-controller="SidebarCtrl" class="sidebar">
  <ul class="nav nav-sidebar">
    <li ng-repeat="category in catalog" ng-class="{active:
      isActive(category.slug)}">
      <a ui-sref="productCatalog
        ({slug: category.slug})">{{category.name}}</a>
    </li>
  </ul>
</div>
```

We will need some CSS styling for the sidebar:

```
/* client/components/sidebar/sidebar.scss */

/* Hide for mobile, show later */
.sidebar {
  display: none;
}
```

---

```

@media (min-width: 768px) {
  .sidebar {
    display: block;
    padding: 20px;
    overflow-x: hidden;
    overflow-y: auto;
  }
}

/* Sidebar navigation */
.nav-sidebar {
  margin-right: -21px; /* 20px padding + 1px border */
  margin-bottom: 20px;
  margin-left: -20px;
}
.nav-sidebar > li > a {
  padding-right: 20px;
  padding-left: 20px;
}
.nav-sidebar > .active > a,
.nav-sidebar > .active > a:hover,
.nav-sidebar > .active > a:focus {
  color: #fff;
  background-color: #428bca;
}

```

The `SidebarCtrl` controller is going to pull out all the product categories from products:

```

/* client/components/sidebar/sidebar.controller.js */

angular.module('meanshopApp')
  .controller('SidebarCtrl', function ($scope, Catalog, $location) {
    $scope.catalog = Catalog.query();

    $scope.isActive = function(route) {
      return $location.path().indexOf(route) > -1;
    };
  });

```

And finally, we need a service that will retrieve the categories from the database. We do that as follows:

```
/* client/components/sidebar/sidebar.service.js */

angular.module('meanshopApp')
  .factory('Catalog', function ($resource) {
    return $resource('/api/catalogs/:id');
  });
```

Now, it's time to move to the backend and create the `/api/catalogs` route. In the next section, we are going to set up the backend to add categories to the products. We will also create 'slugs'—human friendly URLs—that will be linked to the categories.



#### URL slugs

Slugs are human and SEO-friendly URLs. Instead of having a page with a URL identified by an ID such as `/categories/561bcb1cf387488206202ab1`, it is better to have a URL with a unique and meaningful name, such as `/categories/books`.

## Improving product models and controllers

Let's move to the server side. We will now provide the routes and URLs for filtering the products by categories and will allow us to search for products. For the search, we are going to add MongoDB's full-text indexes, and for the categories, we are going to create a new model:

```
/* server/api/product/product.model.js *excerpt */

var ProductSchema = new Schema({
  title: { type: String, required: true, trim: true },
  price: { type: Number, required: true, min: 0 },
  stock: { type: Number, default: 1 },
  description: String,
  imageBin: { data: Buffer, contentType: String },
  imageUrl: String,
  categories: [{ type: Schema.Types.ObjectId,
    ref: 'Catalog', index: true }]
}).index({
  'title': 'text',
  'description': 'text'
});
```

We haven't created the Catalog model, but we will soon. Notice that we added two text indexes on `title` and `description`. That will allow us to search on those fields.

## Catalog controller and routes

In order to provide filtering by category and searching, we need to create new routes as follows:

```
/* server/api/product/index.js *excerpt */

router.get('/', controller.index);
router.get('/:id', controller.show);
router.get('/:slug/catalog', controller.catalog);
router.get('/:term/search', controller.search);
router.post('/', controller.create);
router.put('/:id', controller.update);
router.patch('/:id', controller.update);
router.delete('/:id', controller.destroy);
```

Now that we are referencing the `catalog` and `search` actions in the controller, we need to create them:

```
/* server/api/product/product.controller.js *excerpt */

var Catalog = require('../../catalog/catalog.model');

exports.catalog = function(req, res) {
  Catalog
    .findOne({ slug: req.params.slug })
    .then(function (catalog) {
      var catalog_ids = [catalog._id].concat(catalog.children);
      console.log(catalog_ids, catalog);
      return Product
        .find({'categories': { $in: catalog_ids } })
        .populate('categories')
        .exec();
    })
    .then(function (products) {
      res.json(200, products);
    })
    .then(null, function (err) {
      handleError(res, err);
    });
};
```

```
exports.search = function(req, res) {
  Product
    .find({ $text: { $search: req.params.term } })
    .populate('categories')
    .exec(function (err, products) {
      if(err) { return handleError(res, err); }
      return res.json(200, products);
    });
};
```

For the catalog action, we are performing the following two steps:

1. Finding the category ID by the slug
2. Finding all the products that match the category's ID and the IDs of the category's children.

For the search action, we are using MongoDB's `$text $search`; this is going to work on all the fields which have text indexes, such as title and description. Now, let's create the catalog model.

## The catalog model

In our product catalog, we would like to modify the URL based on the category we are showing. So, for instance, to show all the products under the book category, we would like to show a URL like `/products/books`. For that, we will use a slug.

Let's create the Product catalog and library to help us with the slug:

```
npm install mongoose-url-slugs@0.1.4 --save
```

```
yo angular-fullstack:endpoint catalog
```

```
? What will the url of your endpoint be? /api/catalogs
```

```
create server/api/catalog/catalog.controller.js
create server/api/catalog/catalog.events.js
create server/api/catalog/catalog.integration.js
create server/api/catalog/catalog.model.js
create server/api/catalog/catalog.socket.js
create server/api/catalog/index.js
create server/api/catalog/index.spec.js
```

Now let's modify the catalog model as follows:

```

/* server/api/catalog/catalog.model.js */

var mongoose = require('bluebird').promisifyAll(require('mongoose'));
var Schema = mongoose.Schema;
var slugs = require('mongoose-url-slugs');

var CatalogSchema = new Schema({
  name: { type: String, required: true },
  parent: { type: Schema.Types.ObjectId, ref: 'Catalog' },
  ancestors: [{ type: Schema.Types.ObjectId, ref: 'Catalog' }],
  children: [{ type: Schema.Types.ObjectId, ref: 'Catalog' }]
});

CatalogSchema.methods = {
  addChild: function (child) {
    var that = this;
    child.parent = this._id;
    child.ancestors = this.ancestors.concat([this._id]);
    return this.model('Catalog').create(child).addCallback(
      (function (child) {
        that.children.push(child._id);
        that.save();
      })
    );
  }
}

CatalogSchema.plugin(slugs('name'));

module.exports = mongoose.model('Catalog', CatalogSchema);

```

With this catalog model, we can not only add nested categories, but also keep track of the categories' ancestors and children. Also, notice that we are adding a plugin to generate the slugs based on the name.

One way to test that everything is working as intended is through the unit tests; for more information, refer to <https://raw.githubusercontent.com/amejiarosario/meanshop/ch8/server/api/catalog/catalog.model.spec.js>.

From this unit, we can see that we can find products based on `catalog._id`; we can also find multiple ones using `$in`.



## Seeding products and categories

In order to have a predefined list of products and categories, it will be a good idea to seed the development database with it. Replace the previous product's seed with the following:

```
/* server/config/seed.js *excerpt */

var Catalog = require('../api/catalog/catalog.model');
var mainCatalog, home, books, clothing;

Catalog
  .find({})
  .remove()
  .then(function () {
    return Catalog.create({ name: 'All' });
  })
  .then(function (catalog) {
    mainCatalog = catalog;
    return mainCatalog.addChild({ name: 'Home' });
  })
  .then(function (category) {
    home = category._id;
    return mainCatalog.addChild({ name: 'Books' });
  })
  .then(function (category) {
    books = category._id;
    return mainCatalog.addChild({ name: 'Clothing' });
  })
  .then(function (category) {
    clothing = category._id;
    return Product.find({}).remove({});
  })
  .then(function () {
    return Product.create({
      title: 'MEAN eCommerce Book',
      imageUrl: '/assets/uploads/meanbook.jpg',
      price: 25,
      stock: 250,
      categories: [books],
    });
  });
```

```
        description: 'Build a powerful e-commerce application ...'
      }, {
        title: 'tshirt',
        imageUrl: '/assets/uploads/meantshirt.jpg',
        price: 15,
        stock: 100,
        categories: [clothing],
        description: 'tshirt with the MEAN logo'
      }, {
        title: 'coffee mug',
        imageUrl: '/assets/uploads/meanmug.jpg',
        price: 8,
        stock: 50,
        categories: [home],
        description: 'Convert coffee into MEAN code'
      }
    ]
  });
})
.then(function () {
  console.log('Finished populating Products with categories');
})
.then(null, function (err) {
  console.error('Error populating Products & categories: ', err);
});
```

We use promises to avoid the so-called *callback hell*. We create each one of the categories first and save them in variables. Later, we create each of the products and associate them with its corresponding category. If any errors occur in the process, they are *catch'ed* at the very end.

Now when we run `grunt serve`, we see that the new categories and products are being created.

## Implementing the search and navigation functionality

Now that we have the backend ready to support the products categories and search, we can move to AngularJS and prepare the navigation and search.

## Adding custom \$resource methods

We are going to add some new actions to our product service to match the ones created at the backend:

```
/* client/app/products/products.service.js */

angular.module('meanshopApp')
.factory('Product', function ($resource) {
  return $resource('/api/products/:id/:controller', null, {
    'update': { method: 'PUT' },
    'catalog': { method: 'GET', isArray: true,
      params: {
        controller: 'catalog'
      }
    },
    'search': { method: 'GET', isArray: true,
      params: {
        controller: 'search'
      }
    }
  });
});
```

We have modified the URL with a new attribute; `controller`, which can be either `search` or `catalog`. We pass that parameter explicitly using `params`.

## Setting up routes and controllers

If you remember, earlier in the chapter we broadcast an event every time the search form changed. Now is the time to listen for those events and act on them:

```
/* client/app/products/products.controller.js *excerpt */

.controller('ProductsCtrl', function ($scope, Product) {
  $scope.products = Product.query();

  $scope.$on('search:term', function (event, data) {
    if(data.length) {
      $scope.products = Product.search({id: data});
    } else {

```

---

```

        $scope.products = Product.query();
    }
    });
})

.controller('ProductCatalogCtrl', function ($scope, $stateParams,
Product) {
    $scope.product = Products.catalog({id: $stateParams.slug});
});

```

We have added the products catalog controller here, so we need to add the route/state as well. For the product catalog, we need to add the state `productCatalog` in the routes:

```

/* client/app/products/products.js *excerpt */

.state('productCatalog', {
    url: '/products/:slug',
    templateUrl: 'app/products/products.html',
    controller: 'ProductCatalogCtrl'
})

```

This is all that we need to complete search and navigation. Run `grunt serve`, and verify that search and navigation are working as expected.

## Wrapping it up

All the changes can be found at <https://github.com/amejia/rosario/meanshop/compare/ch7...ch8> along with some UI styling improvements. Now let's sum up how search and navigation works.

## How navigation works on the client side

This is a summary about how navigation works on the AngularJS side.

Navigation starts in `sidebar.html`, where we render each one of the results from `Catalog.query()` (see `sidebar.service.js` and `sidebar.controller.js`). Each one of the rendered catalog elements are linked to a state (`ui-sref="productCatalog({slug: category.slug})"`).

The `productCatalog` (in `product.js`) state invokes the controller, `ProductCatalogCtrl`. This last one invokes the product service with the catalog action (`Product.catalog`). Finally, the request is made to the Node API, which returns the products matching the given category.

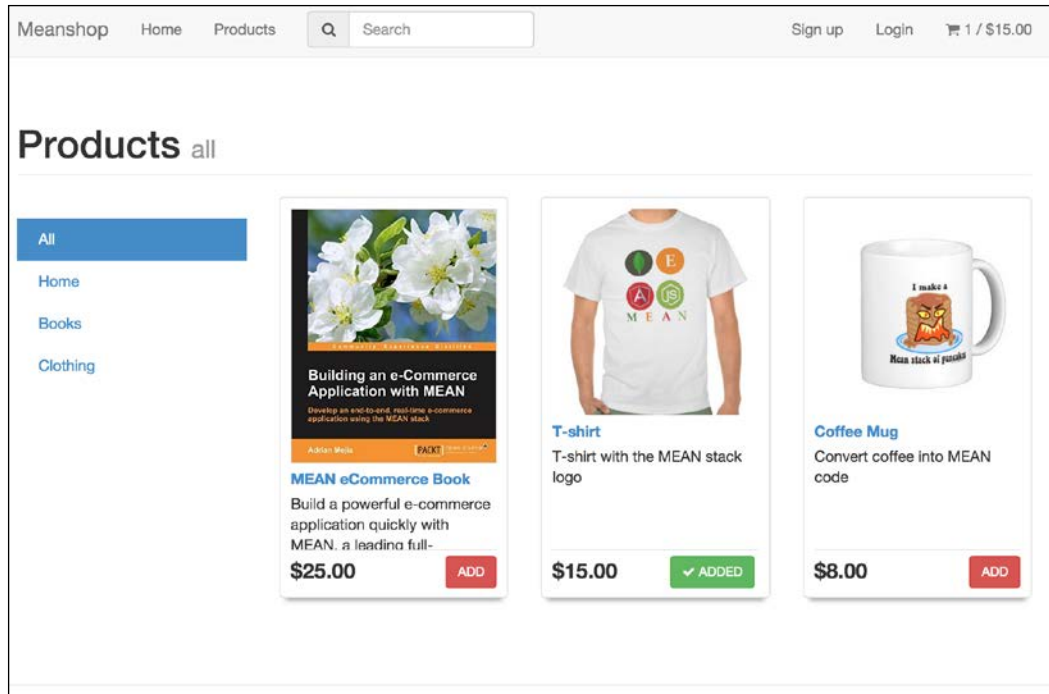


Figure 1: Products filtered by the category "all", which matches all the products

## How search works on the client side

Again, this is a summary about how we implement search in our app.

Search starts in `navbar.html`, where we added the search textbox. As soon as the user focuses on the search box, he/she will be redirected to the products page. The navbar broadcasts the event `'search:term'`, so any service interested in it can subscribe to it.

The `ProductsCtrl` controller renders all the products by default. It also listens for the `search:term` events. If there's a search term, then instead of rendering all the products, it will render only the elements matching the search words. This is done through the `Product.search` service, which invokes the Node API, and returns all the matching products given in the search term.

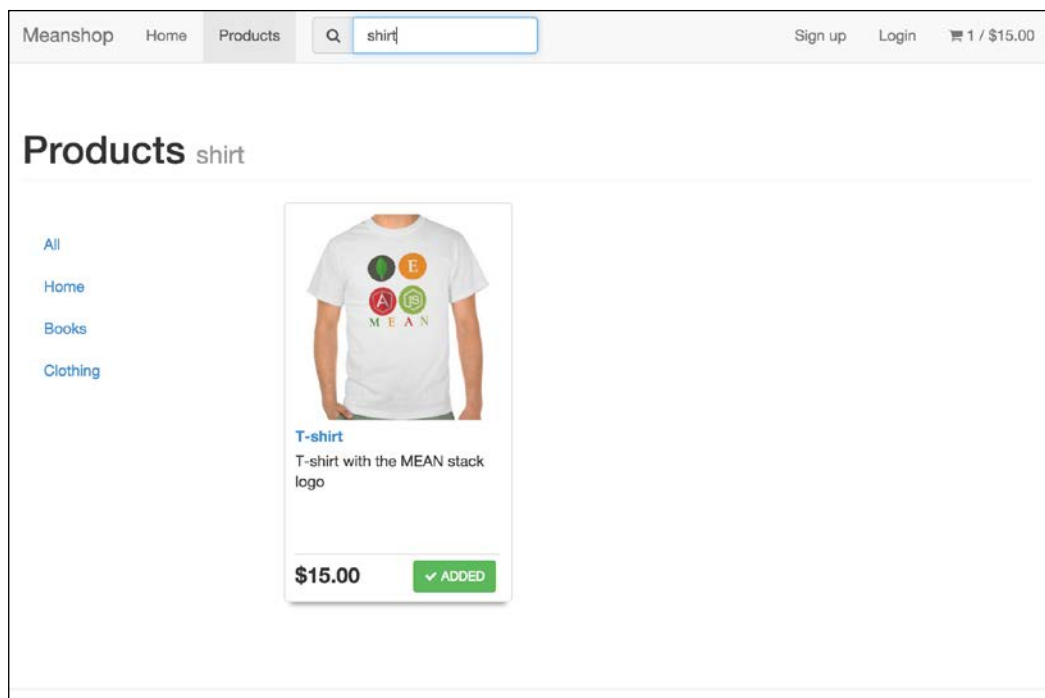


Figure 2: Products matching the search term "shirt"

## Summary

We made major improvements to the UI of our application such as navigation menus, search functionality, and better looks. By now, we have a functional e-commerce site, where we can add new products, and later, filter and search them. We also completed another user story:

*As a user, I want to search for a product so that I can find what I'm looking for quickly.*

The next chapter will get very exciting, since we are going to prepare our site for production and introduce it to the world!



# 9

## Deploying a Production-ready e-Commerce App

Regardless of the kind of application you are building, the deployment phase is key to the success of the application. It dictates the number of users that can access the application simultaneously, the application loading time, and so on. These numbers cannot be taken lightly. Users are not very forgiving towards sluggish applications. They will just move to another one. In this chapter, we are going to explore a number of different deployment setups, application environments, and stress testing the servers.

The following topics will be covered in this chapter:

- Building for production
- Deploying the application to the cloud
- Deploying applications in a multi-server environment
- Performing stress tests

### Building for production

Before deploying the application to the *wild* world web, there are a number of optimizations that need to be done for saving time and bandwidth as well as security issues that need to be addressed. The production environment should be the fastest it can possibly be. It doesn't need to recompile assets, because the file changes are not as frequent as in the development mode.



## Application environments

Usually, applications have four environments: development, testing, acceptance, and production (**DTAP**). Each one has its own independent database, and the main differences are listed as follows:

- **Development:** The application is rebuilt in this stage to reflect any code change in the UI.
- **Testing:** This is used to exercise the full test suite. It usually runs in **Continuous Integration (CI)** servers or the developers run it on-demand in their environments.
- **Production:** All the files and images are optimized to a reduced size and bandwidth utilization. The code is considered stable and ready to serve the end users.
- **Acceptance (triage, staging):** This is identical to the production environment, but the developers interact with new features before releasing it to real users in production. It might have a snapshot of the real production database.

Up to this point, we have been running our application in a development environment (`grunt serve`). It allows us to see our code changes immediately in the UI. However, for production, we don't need constant re-building, since the files don't change frequently. Instead, we need to serve files as fast as possible.

## Optimizations for production environments

There are certain optimizations to maximize performance:

- **Minification:** This is done to remove comments and unnecessary blank spaces from the CSS and JavaScript files. A further minification for the JS files is to rename the variables as a letter. The smaller the size, the larger the bandwidth savings.
- **Concatenation:** The aim of concatenation is to reduce the number of server requests. All CSS files are concatenated into one single file; similarly, all JS files are concatenated into a single file. That way, the server only serves a few files instead of dozens or even hundreds in large projects. The fewer the requests, the faster the browser loads the page.
- **Using CDN for assets:** The idea behind using CDN for assets is to load resources from different servers in a browser in parallel, while resources from the same server are loaded one at a time. Using CDN not only allows the browser to load files in parallel, but it also leverages the browser's cache. Since CDNs are often used across many websites, they are often cached in the browser for future use.

Minification and concatenation are taken care of by our Grunt tasks; those tasks allow us to automate the process, and focus on the meat of our app. Now, we are going to discuss some common setups for production environments.

## Scaling web applications

As the number of users grow over time, the application needs to scale up to be able to keep up with the load. There are usually two main ways to scale up applications:

- **Vertical scale:** This is the simplest scaling. It involves upgrading the server with more resources such as CPU, RAM, HDD, I/O, and more.
- **Horizontal scale:** This is more complicated, but also better in the long run. It involves distributing the load across multiple servers instead of just one.

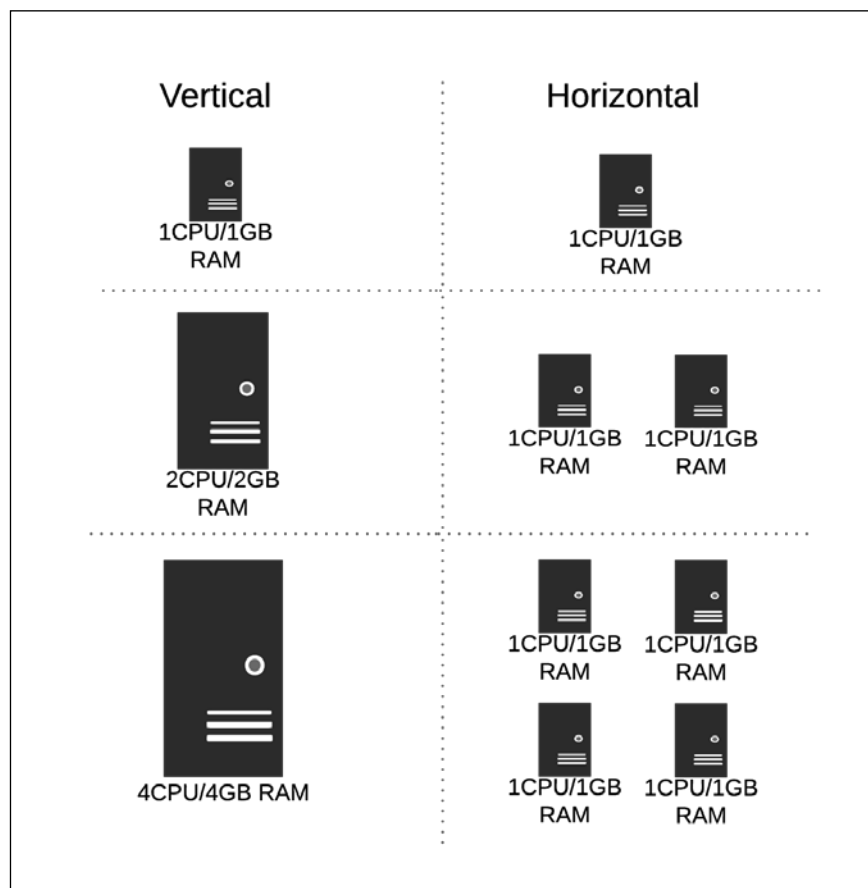


Figure 1: Example of vertical versus horizontal scaling

## Scaling out vertically – one server

The simplest way to deploy an application is to put everything into a single server, that is, all of the services, such as databases and webservers are in one server.

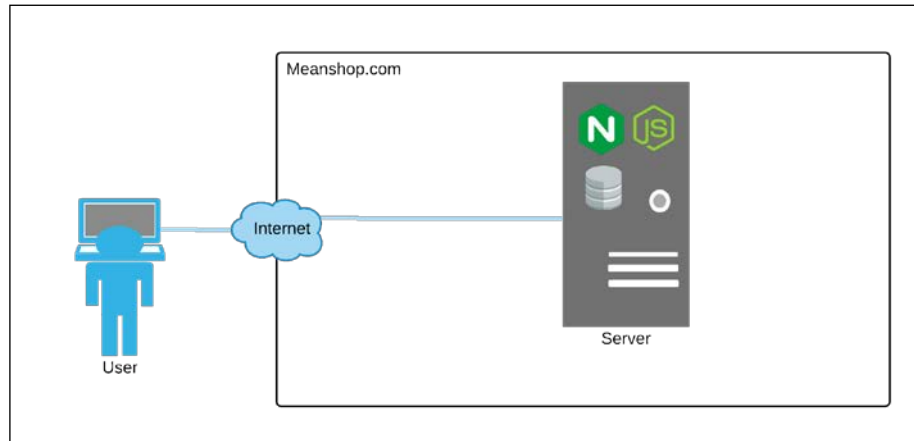


Figure 2: Single server deployment setup

All the processes such as the database, web server, and application are going to compete for the I/O, CPU, and RAM. As the number of users grows significantly, we can scale the single server by adding more and faster resources to it. For example, by adding SSD, more CPU, and RAM. This is called vertical scaling. However, this has limits to how cost-effective it can be, rather than scaling horizontally.

## Scaling out horizontally – multiple servers

Splitting the application into multiple servers has proven to be a more cost-effective way of scaling applications. As a matter of fact, companies like Google, Facebook, Amazon, and many others use multiple clusters of servers to serve millions of concurrent users.

The following diagram shows a way to scale in a multi-server deployment:

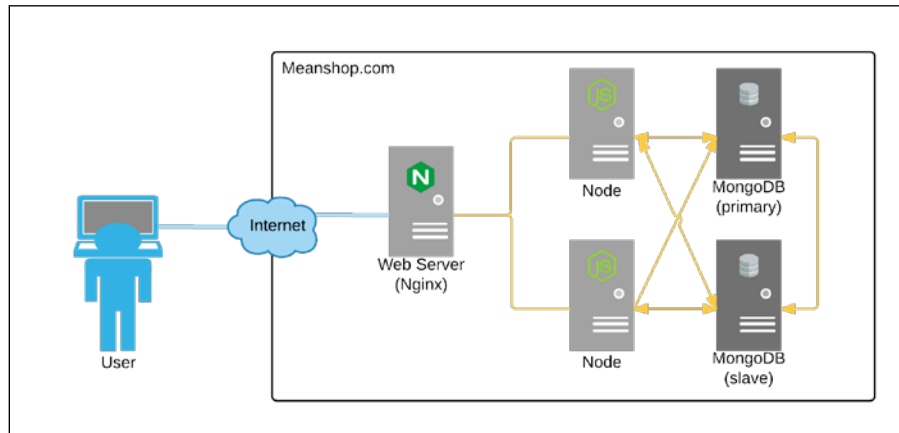


Figure 3: Multi-server deployment setup

There are many ways of splitting applications, but the following are the main components:

- **Database server(s):** The application and database no longer compete for CPU, RAM, and I/O. Now, each tier can have its own servers. The bottleneck could be network bandwidth and latency, so the network should be optimized for the required transfer rates.
- **Load balancer server(s):** Now that we have multiple servers, we can use a load balancer to distribute the application load into multiple instances. This provides protection against DDoS attacks. However, the load balancer should be configured properly, and have enough resources or it can become a bottleneck for performance.
- **Caching/reverse proxy server(s):** Static files, images, and some HTTP requests can be cached to serve them quicker and to reduce CPU usage. For example, Nginx is good at this.
- **Database replication server(s):** Very often, a web application requires many more reads (show/search products) than writes (create new products). We can add multiple-read databases (slaves) while having one read-write database (master).
- **Additional services server(s):** These are not essential, but they can be useful for monitoring, logging, and taking backups, and so on. Monitoring helps in detecting when a server is down or reaching max capacity. Having centralized logging aids debugging. Finally, backups are very useful for restoring the site in case of failures.

## Deploying the application to the cloud

Deploying an application is the process of moving the current code base to a production server. There are many options depending on our needs. There are cloud application platforms such as Heroku and Openshift that abstract all the complexity of setting up all the servers, and allow us to scale on-demand with just a few clicks. On the other hand, there are also **Virtual Private Servers (VPS)**, which grant you access to cloud resources, and you have to set up all the servers yourself. In the first one (**Platform as a Service** or **PaaS**), we only need to worry about the application, while the platform manages the servers in the background. VPS, on the other hand, give us full access to cloud servers where we need to do everything ourselves.

## Platform as a Service

**Platform as a service (PaaS)** is a convenient type of cloud computing. It allows us to quickly deploy applications without having to spend time setting up servers. The platform is configured to support a number of different types of applications. There are a couple of PaaS that are free to try such as Heroku and Openshift. Let's try to get our app deployed!

## Heroku

Heroku requires installing a command-line program called `heroku-toolbelt`. Follow the instructions on <https://toolbelt.heroku.com> to install it. We also need to create an account in Heroku. Then, we will be able to log in through the command-line by typing: `heroku login`.

Our `yo` generator already supports deploying to Heroku. So, we just need to build the project and deploy it:

```
$ grunt build
$ yo angular-fullstack:heroku
```

The last command is going to ask you a few questions like application name, and so on. After this is done, you need to set up the MongoDB database:

```
$ cd dist && heroku addons:create mongolab:sandbox
```

We also need to set the environment variables. Set `NODE_ENV` and all variables that you have on `local.env.js` to heroku:

```
$ heroku config:set NODE_ENV=production

# add all the social networks IDs and secrets.e.g.:
$ heroku config:set FACEBOOK_ID=appId FACEBOOK_SECRET=secret

# visualize all the set variables with
$ heroku config
```

Finally, you can open the application by running it with the following command:

```
$ heroku open
```

Any other update can be refreshed on Heroku by typing the following command:

```
$ grunt buildcontrol:heroku
```

PaaS offers the ability to scale easily on demand. However, you will never have full control of the server's configurations. In case you want to do that, you need to go with **Virtual Private Servers (VPS)** or your own servers. In the next section, we are going to explain how to set it up.

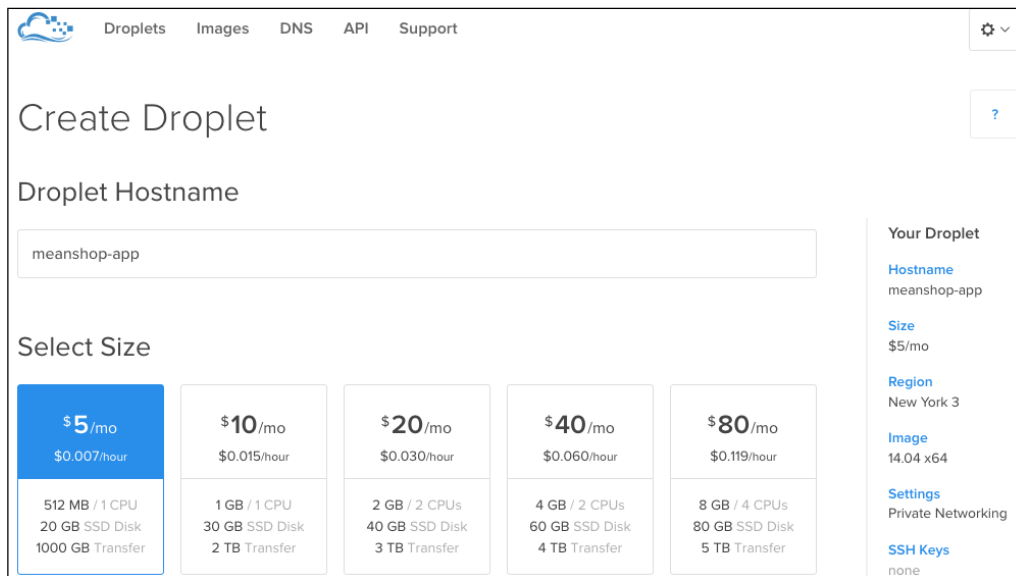
## Virtual Private Servers and Cloud Servers

There are a number of **Virtual Private Server (VPS)** providers, such as Amazon AWS, Digital Ocean, Rackspace, and many more. We will go through the process of deploying our app in Digital Ocean, but the same applies to any other provider or even our own (bare metal) servers.

### Digital Ocean

Again, the steps that follow are more or less the same for any server using Ubuntu 14.04 x64. You can go to <https://www.digitalocean.com/?refcode=dd388adf295f> to get a \$10 credit and follow along with this section, or use any other similar provider that you are familiar with.

After you sign up, go ahead and create a 512 MB/1 CPU droplet (server instance) using Ubuntu 14.04 x64:



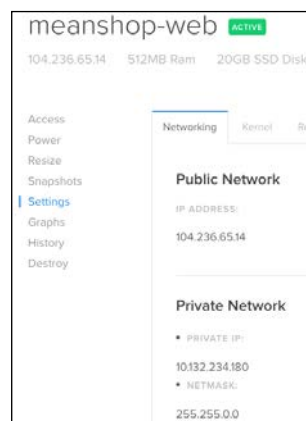
The screenshot shows the 'Create Droplet' page in the DigitalOcean dashboard. The 'Droplet Hostname' field is filled with 'meanshop-app'. Under 'Select Size', the '\$5/mo' option (512 MB / 1 CPU, 20 GB SSD Disk, 1000 GB Transfer) is selected. On the right, the 'Your Droplet' summary shows: Hostname: meanshop-app, Size: \$5/mo, Region: New York 3, Image: 14.04 x64, Settings: Private Networking, and SSH Keys: none.

Size	Price	Hourly Price	RAM	CPU	SSD Disk	Transfer
\$5/mo	\$0.007/hour	512 MB / 1 CPU	20 GB SSD Disk	1000 GB Transfer		
\$10/mo	\$0.015/hour	1 GB / 1 CPU	30 GB SSD Disk	2 TB Transfer		
\$20/mo	\$0.030/hour	2 GB / 2 CPUs	40 GB SSD Disk	3 TB Transfer		
\$40/mo	\$0.060/hour	4 GB / 2 CPUs	60 GB SSD Disk	4 TB Transfer		
\$80/mo	\$0.119/hour	8 GB / 4 CPUs	80 GB SSD Disk	5 TB Transfer		

Figure 4: Creating a server instance in Digital Ocean

Choose the region that is closest to you, in my case, New York. In the settings, select **enable private network**. Once you click, you create a droplet. After a minute or so, you will receive an e-mail with the password for the root username.

Create another droplet, this time called meanshop-web, with exactly the same settings. Next, go to the **droplet** menu, and get the **Public** and **Private IP** addresses for each one under settings.



The screenshot shows the 'Networking' tab for a droplet named 'meanshop-web'. It displays the public IP address 104.236.65.14 and the private IP address 10.132.234.180 with a netmask of 255.255.0.0.

Network Type	IP Address	Netmask
Public Network	104.236.65.14	
Private Network	10.132.234.180	255.255.0.0

Figure 5: Public and Private networks on Digital Ocean settings

Once you log in as the root user, you can create a new user with `sudo` privileges as follows:

```
adduser YOUR_USERNAME
passwd -a YOUR_USERNAME sudo
sudo - YOUR_USERNAME
```

Now, we are ready to deploy our app! Again, the steps that follow are the same regardless of the provider you choose, or even with your own personal servers with Ubuntu 14.04.

## Deploying applications in a multi-server environment

In this section, we are going to learn how to deploy our app and scale it out vertically. We are going to use only two servers this time: one for the Reverse Proxy/Load Balancer and another for the node application. We can scale this out by adding more node application servers and database servers, and by referencing it in the load balancer.

For this section's sake, we are going to deploy our app in two servers: one server containing Nginx that serves as the load balancer and another server that contains the NodeJS app and the database.

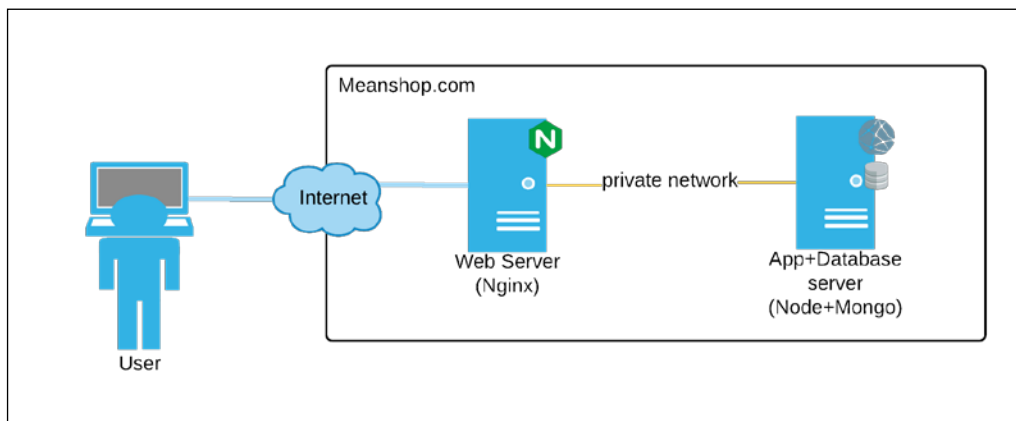


Figure 6: Two-server deployment



## Setting up the app server – NodeJS application

Once you ssh into the app server, install the project dependencies and NodeJS through nvm:

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential openssl libssl-dev pkg-config
git-core mongodb ruby
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.29.0/
install.sh | bash
$ source ~/.bashrc
$ nvm install 0.12.7
$ nvm use 0.12.7
$ nvm alias default 0.12.7
$ node -v
```

The last command should verify that you have NodeJS installed. Now, we can proceed to pull out our project from our repository. If you have been using GitHub for your code changes, you can do something like this:

```
cd && git clone https://github.com/amejia Rosario/meanshop.git
```

Once in the directory, install all the packages, as well as some other dependencies like SASS and grunt-cli, and run the app:

```
npm install -g grunt-cli bower pm2
npm install
bower install
npm install grunt-contrib-imagemin
sudo gem install sass
# and finally:
grunt build
```

To run our application in the production mode, we need to set the production environment, and run the app from the dist folder:

```
NODE_ENV=production node dist/server/app.js
```

We can go to our browser, and type `http://ip-address:8080` to see our application! Yay!

Let's permanently set our server to production. Add `NODE_ENV` to the very end of this file:

```
sudo vi /etc/environment
```

Also, add the following variables into the environment:

```
NODE_ENV=production
IP=10.132.234.0 #set to APP_PRIVATE_IP_ADDRESS
PORT=8080
DOMAIN=104.236.0.0 # PUBLIC_IP_ADDRESS or domain if you have it
FACEBOOK_ID=...
FACEBOOK_SECRET=...
TWITTER_ID=...
TWITTER_SECRET=...
GOOGLE_ID=...
GOOGLE_SECRET=...
PAYPAL_ID=...
PAYPAL_SECRET=...
```

Load the environment values by relogging into the shell:

```
sudo su - $USER
echo $NODE_ENV
```

You should see the word `production`. A further improvement could be to use `pm2` to daemonize the application:

```
pm2 start ~/meanshop/dist/server/app.js
```

We can also add it to the start-up phase, so every time the server boots up, it starts the application immediately:

```
pm2 startup ubuntu
# run the generated command line and then:
pm2 save
```

Other useful commands are as follows:

```
pm2 list
pm2 monit
pm2 logs
```

## Setting up web server – Nginx server

Let's go to our second server and install Nginx. Nginx is a high performance web server. We are going to use it as our reverse proxy and load balancer:

```
sudo apt-get update
sudo apt-get install -y nginx
```

If we go to our browser and type the public IP address of the second server, we can see that Nginx is already serving a default page on port 80. The configuration file is at the following location:

```
sudo vi /etc/nginx/sites-available/default
```

Replace all the content with this new configuration:

```
# /etc/nginx/sites-available/default

upstream node_apps {
    server APP_PRIVATE_IP_ADDRESS:8080;
}

server {
    listen 80;

    server_name localhost; # or your hostname.com

    location / {
        proxy_pass http://node_apps;
        proxy_http_version 1.1;
        # server context headers
        proxy_set_header HOST $host;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        # headers for proxying a WebSocket
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection 'upgrade';
        proxy_set_header Host $host;
        proxy_cache_bypass $http_upgrade;
    }
}
```

In the upstream block, we can add as many node applications as we want. Nginx will act as the load balancer, and it will forward the request to one of the servers listed in turn. However, if we add a new application server instance, we will need to move the database to a different server so that all node instances reference the same data.

## Performing stress tests

An important part of the production environment is to be able to assess performance. In this section, we are going to discuss some tools that we can use for doing that.

## HTTP benchmarking tools

The HTTP benchmarking tools are designed to generate a number of concurrent requests, evaluate how long the server takes to process it, and to capture failures. One common tool that works great for this case is ApacheBench.

### ApacheBench

**ApacheBench** (**ab**) not only generates concurrent requests, but it also generates a report on the time taken to process the requests, errors, request per seconds, and so on.

We can install it on a Mac using brew:

```
brew install ab
```

In Ubuntu, we can do it using the following command:

```
sudo apt-get install apache2-utils
```

## Benchmarking Heroku deployment

Let's see how long it takes to process 10,000 requests at a rate of 100 connections at a time at <https://gist.github.com/amejiarosario/b5aa655522f2776379f6>.

That's a lot of information. But we can make out that this configuration can handle around 95 req/sec, and that the meantime is 1 sec, on each request.

## Benchmarking VPS multi-server deployment

Let's run the same command, this time hitting the Digital Ocean deployment at <https://gist.github.com/amejiarosario/c6f45e6cc046871f9ef5>.

In this case, we can see a huge difference. All requests were processed in 28s. vs 105s. It was able to handle almost four times the requests (400 req/sec.), and the mean time for processing the request is five times less (279 ms).

Using a similar method, we can try out different deployment setups, and compare them.

## Production architecture for scaling NodeJS

So far, we have seen some different server setups and scaling concepts. In this section, we are going to explain how to incrementally scale a production application.

### Phase 0 – one server

This server is usually used in development and low-traffic production applications. It starts with all the components in the same server:

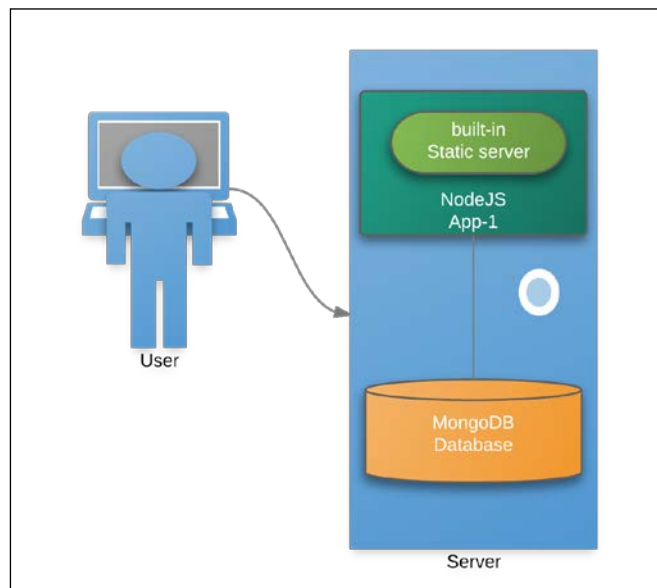


Figure 7: Single server setup

This is the simplest setup possible, where everything (database, webserver, or an application) is on the same server.

## Phase 1 – multiple application instances in one server

The next step is to add an additional application instance to be able to perform zero-downtime upgrades. This way, we can roll the application updates one instance at a time, without taking the site down for maintenance.

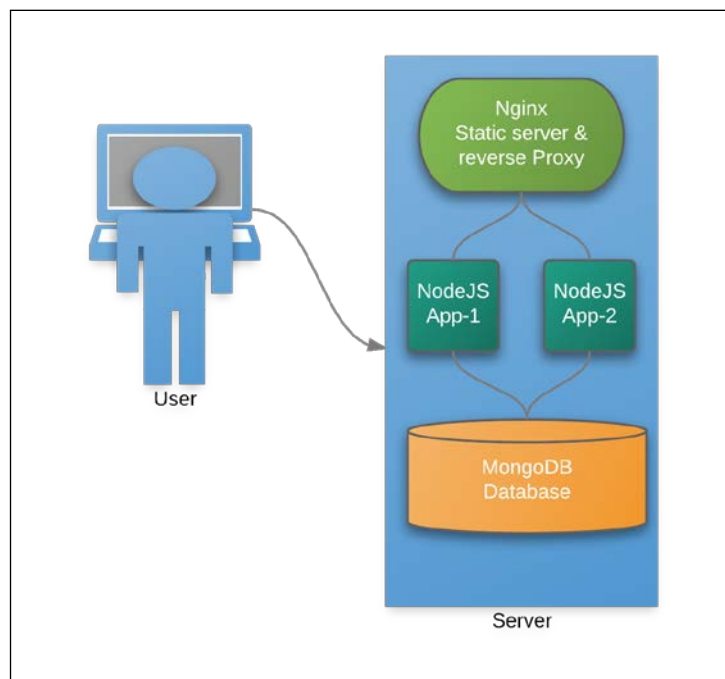


Figure 8: Multiple app instances in one server

For managing the two application instances, we are using Nginx as a load balancer and reverse proxy. We are also serving static files directly from Nginx without touching NodeJS for better performance. Notice that the number of app instances are given by the number of CPUs available in the server (two CPUs in our case). This is vertical scaling, since we increased the performance of the server (two CPUs) while keeping everything in one server.

## Phase 2 – multiple servers

As the number of users starts to grow, the database and application instances start fighting for the server's resources. At this point, we need to have each service in its own server as that makes more sense.

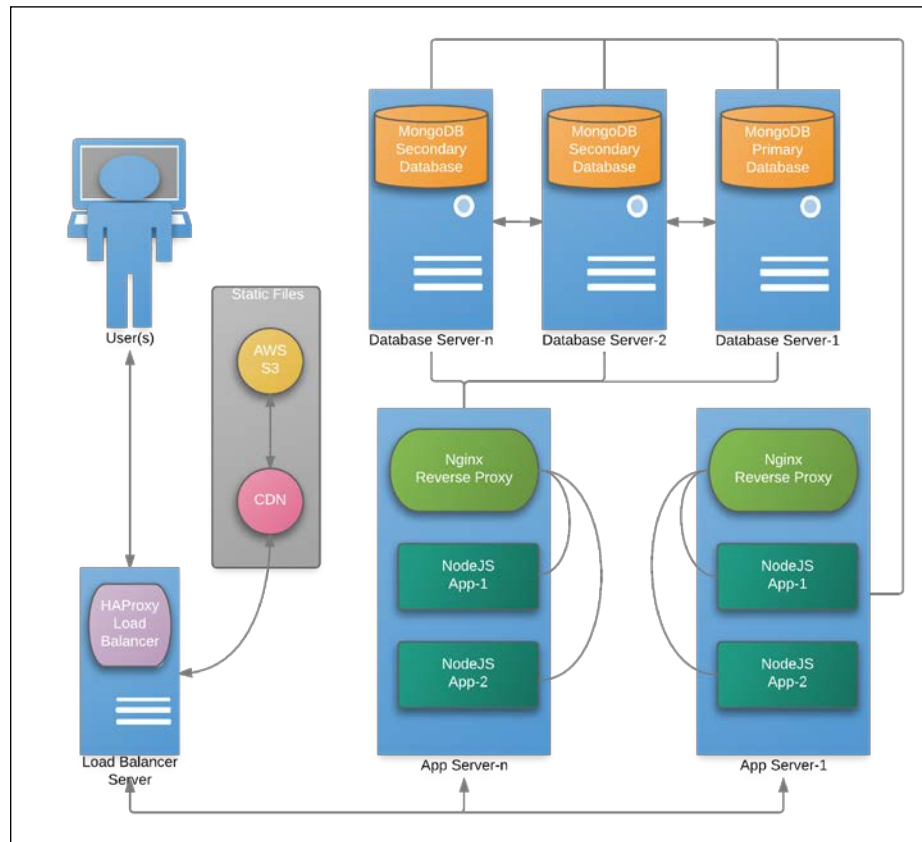


Figure 9: Horizontal scaling with multiple servers

The following are the guidelines for splitting up the services into multiple servers:

- Instead of serving static files with Nginx, delegate static files to different servers or services such as CDNs and Amazon S3. CDNs have the advantage of being distributed data centers on multiple locations (closer to the user).
- Split the database's instances to its own servers in a primary/secondary setup. There you can scale up or down to many database replicas as needed.
- Add a load balancer so that you can scale up or down with *app servers* as needed.

## Phase 3 – Micro-services

Micro-services (specialization of **Service Oriented Architecture**) aim to divide (monolithic) applications into small and independent processes that communicate through an API. This way, we can scale each one as needed. Let's imagine that the setup in phase two is for the *store*. When we need to add new features into the same code base, we should consider adding them as a separate service, as it makes more sense:

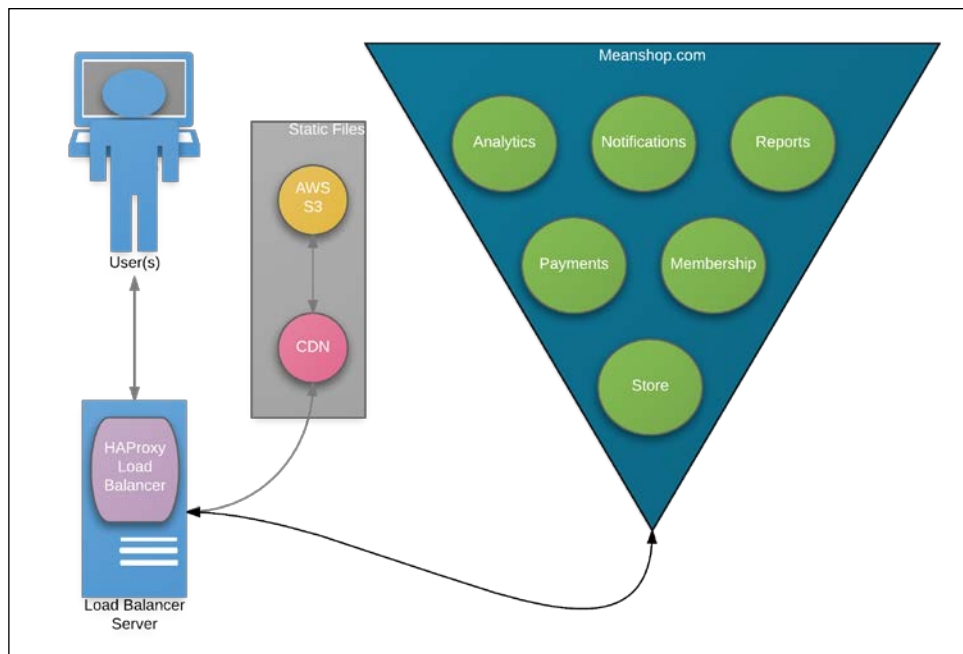


Figure 10: Micro-service architecture

Each one of the micro services can be deployed independently, and scaled up or down following the description in previous phases.

## Next steps on security

In this chapter, it is not possible to cover all the aspects of production servers, since it is a very broad topic. Another extensive topic is security; we are going to provide some resources to continue exploring that at; [https://www.owasp.org/index.php/Web\\_Application\\_Security\\_Testing\\_Cheat\\_Sheet](https://www.owasp.org/index.php/Web_Application_Security_Testing_Cheat_Sheet).





**Open Web Application Security Project (OWASP)** is a non-profit international organization that provides resources so that applications can be made secure and trusted.

## Summary

This chapter is key for turning any web project into a real-world application. In this chapter, we examined multiple server setups, and you learned how to scale them vertically and horizontally. Vertical scaling implies adding more CPU and RAM to a single server, while horizontal scaling implies splitting up the work among multiple servers.

You also learned how to benchmark different application deployments. This is a valuable tool when we need to scale our application, and also find bottlenecks by profiling our application. Finally, we explored scaling up our application gradually, and making use of micro-services for increasing performance and maintainability.

In the next and last chapter, we are going to explore how you can continue extending this e-commerce application to meet your particular needs. Stay tuned!

# 10

## Adding Your Own Features with High Quality

Web applications and software in general are never *done*. It doesn't matter how hard we try, there is always some defect to fix, some new features to add, room for improvements, and new ideas to implement. Instead of being discouraged by this fact, it is better to embrace it and plan ahead.

There will always be some kind of bugs in the software, but we can mitigate them significantly by adding a test suite. In some seasons, (for example, holidays), the e-commerce application might be stricken by a large number of users at once. For those times, we need to know how to scale our application to supply the demand. There will be new ideas and unique demands; for that, we need our application to be extensible.

We will cover the following topics in this chapter:

- Planning a new feature
- Testing the new feature
- Deploying a new version of the application

### Planning a new feature

Planning a new feature before writing the code can save us a lot of time. It's better to make all changes in a mock-up/ wireframe, than in an application's code. In this section, we are going to explore some tips for extending our application with new features.

With reference to the user stories that we mentioned in *Chapter 1, Getting Started with the MEAN Stack*, we have implemented everything but the back office. We are going to do that in this chapter:

- As an admin, I want to *manage user roles* so that I can create new admins and sellers, and remove seller permissions
- As an admin, I want to *manage all the products* so that I can ban them if they are not appropriate

## Wire framing

Wireframes allow us to make changes quickly while experimenting with different ideas. They don't have to be implemented exactly, but are more like a general guideline. Refer again to *Chapter 1, Getting Started with the MEAN Stack*, and review the wireframes that we have already created. For managing users and products, we plan to have the following interface:

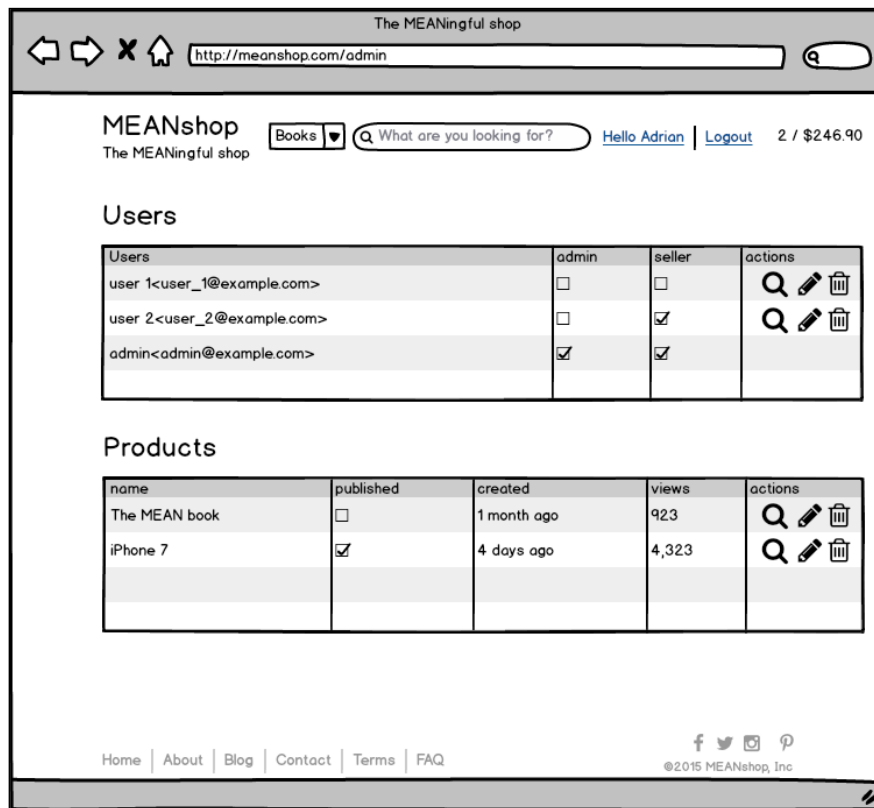


Figure 1: Wireframe for the back office

Once we have the wireframe, we can tell what we will need beforehand.

## Implementing the solution

This is a pretty straightforward implementation. We are going to add the tables to the `admin.html` page. Additionally, we are going to fetch the product and user data, and implement the actions in `admin.controller.js`.

### The HTML page

We already have some user data displayed on the page, but we are going to format it as a table, and add the products data as well:

```
<!-- client/app/admin/admin.html -->

<navbar></navbar>

<div class="container">
  <h1>Users</h1>
  <div class="table-responsive">
    <table class="table table-hover">
      <tr>
        <th>User</th>
        <th>Role</th>
        <th>Provider</th>
        <th>Actions</th>
      </tr>
      <tr ng-repeat="user in users">
        <td>{{user.name}} <span class="text-muted">{{user.email}}</span></td>
        <td class="success" ng-class="{ 'success' : user.role == 'admin', 'info': user.role == 'user' }">{{user.role}}</td>
        <td>{{user.provider}}</td>
        <td>
          <a ng-click="deleteUser(user)"><i class="fa fa-2x fa-trash-o"></i></a>
        </td>
      </tr>
    </table>
  </div>

  <h1>Products</h1>
  <div class="table-responsive">
    <table class="table table-hover">
      <tr>
        <th>Name</th>
        <th>Price</th>
      </tr>
    </table>
  </div>
</div>
```

```
        <th>Stock</th>
        <th>Categories</th>
        <th>Actions</th>
    </tr>
    <tr ng-repeat="product in products">
        <td>{{product.title}}</td>
        <td>{{product.price | currency}}</td>
        <td>{{product.stock}}</td>
        <td>
            <span ng-repeat="category in product.categories">
                <a ui-sref="productCatalog
                    ({slug: category.slug})">{{category.name}}</a>
            </span>
        </td>
        <td>
            <a ng-click="showProduct(product)">
                <i class="fa fa-2x fa-eye"></i></a>
            <a ng-click="editProduct(product)">
                <i class="fa fa-2x fa-pencil"></i></a>
            <a ng-click="deleteProduct(product)">
                <i class="fa fa-2x fa-trash-o"></i></a>
        </td>
    </tr>
</table>
</div>
</div>
```

Now we need to define the methods used on this view by its controller:

```
/* client/app/admin/admin.controller.js */

angular.module('meanshopApp')
    .controller('AdminCtrl', function($scope, $http, Auth, User,
        Product, $state) {

        // Use the User $resource to fetch all users
        $scope.users = User.query();

        $scope.deleteUser = function(user) {
            User.remove({ id: user._id });
            $scope.users.splice(this.$index, 1);
        };

        $scope.products = Product.query();
```

---

```

$scope.showProduct = function(product){
    $state.go('viewProduct', {id: product._id});
}

$scope.editProduct = function(product){
    $state.go('editProduct', {id: product._id});
}

$scope.deleteProduct = function(product){
    Product.remove({ id: product._id });
    $scope.products.splice(this.$index, 1);
}
});

```

For the edit and show product actions, we are just redirecting the user to the form that we have previously created.

## Testing the new feature

Running the tests is a great way to minimize the number of bugs introduced in the application along with the new feature. Let's go ahead and test the implementation of the administrator page.

## AngularJS testing

Like all other tests, we are testing our Angular controllers with the Mocha and Karma runners. Since we modified `admin.controller.js`, we need to test that all is working as intended. Add this new file:

```

/* client/app/admin/admin.controller.spec.js */

describe('AdminController', function() {
    beforeEach(module('meanshopApp'));

    var Product, User, $state, $controller, controller, $scope;

    var productAttributes = [
        {_id: 1, title: 'Product1', price: 100.10, stock: 10},
        {_id: 2, title: 'Product2', price: 200.00, stock: 20}
    ];

    var userAttributes = [
        {_id: 1, name: 'User1', email: 'user1@example.com', provider:
            'local'},
    ];

```

```
    {_id: 2, name: 'User2', email: 'user2@example.com', provider:
      'facebook'}
  ];

  beforeEach(inject(function (_$controller_, $rootScope, _User_,
    _Product_) {
    $controller = _$controller_;
    $scope = $rootScope.$new();

    User = _User_;
    Product = _Product_;

    sinon.stub(User, 'query').returns(userAttributes);
    sinon.stub(User, 'remove');
    sinon.stub(Product, 'query').returns(productAttributes);
    sinon.stub(Product, 'remove');

    $state = { go: sinon.stub() };
  }));

  describe('$scope.users', function() {
    beforeEach(function () {
      controller = $controller('AdminCtrl', {
        $scope: $scope,
        User: User
      });
    });

    it('loads the users', function() {
      expect($scope.users).toEqual(userAttributes);
    });

    it('deletes users', function() {
      var user1 = userAttributes[0];
      var user2 = userAttributes[1];
      $scope.deleteUser(user1);
      assert(User.remove.calledOnce);
      expect(angular.equals($scope.users, [user2])).toEqual(true);
    });
  });
});
```

```
describe('$scope.products', function() {
  var product1 = productAttributes[0];
  var product2 = productAttributes[1];

  beforeEach(function () {
    controller = $controller('AdminCtrl', {
      $scope: $scope,
      $state: $state,
      Product: Product,
    });
  });

  it('loads the products', function() {
    expect($scope.products).toEqual(productAttributes);
  });

  it('deletes products', function() {
    $scope.deleteProduct(product1);
    assert(Product.remove.calledOnce);
    expect(angular.equals($scope.products,
      [product2])).toEqual(true);
  });

  it('redirects to edit form', function() {
    $scope.editProduct(product1);
    $state.go.should.have.been.calledWith('editProduct',
      {id: product1._id});
  });

  it('redirects to product show', function() {
    $scope.showProduct(product2);
    $state.go.should.have.been.calledWith('viewProduct',
      {id: product2._id});
  });
});
```

We are using Sinon stubs to test that the `Products` and `User` service methods are invoked properly. The same applies to `$state.go`.



## Features backlog

There are so many more features that we could add to our e-commerce application. However, we cannot add all of them in this book. We will, instead, cover the basics, and provide the tools to build any feature that you need to fulfill your needs. Some ideas are as follows:

- **Wishlist:** Similar to a shopping cart, but a list of items that the user would like to buy in the future.
- **Notifications:** This sends e-mails/UI alerts to the users and admins based on events of their interest.
- **Statistics:** This will add charts and tables for approximately 10 of the top most searched/sold items.
- **Reports:** This can be used for generating sales reports, seller sold items, and so on.
- **Monitoring:** This is a service that automatically checks for server resources, and notifies when they are about to reach the max performance limit.
- **Authorization:** This requires the user to be logged in to perform certain actions such as creating a new product. Similarly, it allows the user to modify a product only if he/she is the owner.

The preceding list is a sample backlog of the features to implement. However, we are going to focus on how to redeploy the application every time we add a new feature to it.

## Deploying a new version of the app

Deploying can be a repetitive and error-prone task. However, we are going to automate it to make it more efficient. We are going to use a deployment tool called Capistrano to ease the task of deploying new versions of our app. We are also aiming for zero-downtime upgrades.

## Zero-downtime deployments

It's not convenient for users to see a message, such as *Site down for maintenance*, so we are going to avoid that at all costs. We would also like to be able to update our app as often as needed without the users even noticing. This can be accomplished with a zero-downtime architecture. Using two node applications, we can update one first while the other is still serving new requests. Then, we update the second app while the updated first app starts serving clients. That way, there's always an instance of the application serving the clients.

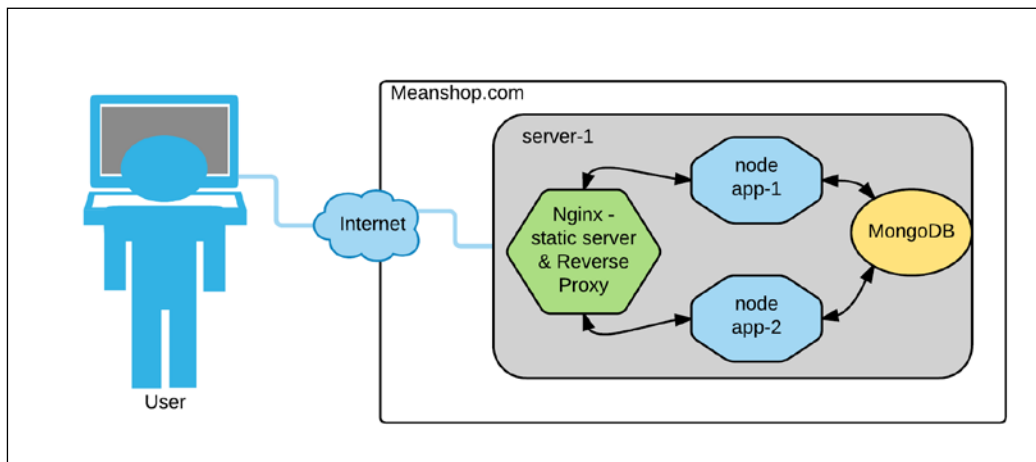


Figure 2: Zero-downtime deployment setup

Now that we have the architecture plan in place, let's go ahead and automate the process.

## Setting up the zero-downtime production server

At this point, you should create a server with at least two CPUs, with the help of the instructions given in the previous chapter (using the \$10 bonus), or you can follow along with any other server that you prefer. Our setup might look like this:

The screenshot shows the DigitalOcean 'Create Droplet' page. At the top, there are navigation links: Droplets, Images, DNS, API, and Support. Below the navigation bar, the title 'Create Droplet' is displayed. Underneath, there's a section for 'Droplet Hostname' with a text input field containing 'meanshop'. To the right of this section is a settings icon and a help icon. Below the hostname section is the 'Select Size' section, which contains a grid of size options. The '\$20/mo' option is highlighted in blue. To the right of the size selection is a sidebar titled 'Your Droplet' which lists the configuration details: Hostname (meanshop), Size (\$20/mo), Region (New York 3), Image (14.04 x64), Settings (Private Networking), and SSH Keys (laptop).

Size	Price	Hourly Price	Memory	CPU	Disk	Transfer
\$5/mo	\$0.007/hour	512 MB / 1 CPU	20 GB SSD Disk	1000 GB Transfer		
\$10/mo	\$0.015/hour	1 GB / 1 CPU	30 GB SSD Disk	2 TB Transfer		
<b>\$20/mo</b>	<b>\$0.030/hour</b>	<b>2 GB / 2 CPUs</b>	<b>40 GB SSD Disk</b>	<b>3 TB Transfer</b>		
\$40/mo	\$0.060/hour	4 GB / 2 CPUs	60 GB SSD Disk	4 TB Transfer		
\$80/mo	\$0.119/hour	8 GB / 4 CPUs	80 GB SSD Disk	5 TB Transfer		
\$160/mo	\$0.238/hour	16 GB / 8 CPUs	160 GB SSD Disk	6 TB Transfer		
\$320/mo	\$0.476/hour	32 GB / 12 CPUs	320 GB SSD Disk	7 TB Transfer		
\$480/mo	\$0.714/hour	48 GB / 16 CPUs	480 GB SSD Disk	8 TB Transfer		
\$640/mo	\$0.952/hour	64 GB / 20 CPUs	640 GB SSD Disk	9 TB Transfer		

Figure 3: Creating VM with two CPUs

Write down the private and public IP addresses. NodeJS applications use the private address to bind to ports 8080 and 8081, while Nginx will bind to the public IP address on port 80.

## Getting started with Capistrano

Capistrano is a remote multi-server automation tool that will allow us to deploy our app in different environments such as Staging/QA and production. Also, we can update our app as often as needed without worrying about the users getting dropped.

## Installing Capistrano

Capistrano is a Ruby program, so we need to install Ruby (if you haven't done so yet).

For Windows, go to: <http://rubyinstaller.org/>.

For Ubuntu, we are going to install a **Ruby version manager (rvm)**:

```
sudo apt-get install ruby
```

Or for MacOS:

```
brew install ruby
```

We can install Capistrano as follows:

```
gem install Capistrano -v 3.4.0
```

Now we can bootstrap it in the `meanshop` folder:

```
cap install
```

## Understanding Capistrano

The way Capistrano works is through tasks (rake tasks). Those tasks perform operations on servers such as installing programs, pulling code from a repository, restarting a service, and much more. Basically, we can automate any action that we can perform through a remote shell (SSH). We can scaffold the basic files running `cap install`.


During the installation process, a number of files and directories are added to the project, which are as follows:

- `Capfile`: This loads the Capistrano tasks, and can also load predefined tasks made by the community
- `config/deploy.rb`: This sets the variables that we are going to use through our tasks such as repository, application name, and so on
- `config/deploy/{production.rb, staging.rb}`: While `deploy.rb` sets the variables that are common for all environments, `production/staging.rb` set the variables specific to the deployment stage, for example, `NODE_ENV`, servers IP addresses, and so forth
- `lib/capistrano/tasks/*.rake`: This contains all the additional tasks, and can be invoked from the `deploy.rb` script

Capistrano comes with a default task called `cap production deploy`. This task executes the following sequence:

- `deploy:starting`: This starts a deployment, making sure everything is ready
- `deploy:started`: This is the started hook (for custom tasks)
- `deploy:updating`: This updates server(s) with a new release (for example, `git pull`)
- `deploy:updated`: This is the updated hook (for custom tasks)
- `deploy:publishing`: This publishes the new release (for example, create symlinks)
- `deploy:published`: This is the published hook (for custom tasks)
- `deploy:finishing`: This finishes the deployment, cleans up temp files
- `deploy:finished`: This is the finished hook (for custom tasks)

This `deploy` task pulls the code, and creates a release directory where the last five are kept. The most recent release has a symlink to `current` where the app lives.

[  Full documentation on Capistrano can be found at <http://capistranorb.com>. ]

## Preparing the server

Now, we need a deployer user that we can use in Capistrano. Let's ssh into the server where we just created the user:

```
root@remote $ adduser deployer
```

Optionally, to avoid typing the password every time, let's add the remote keys. In Ubuntu and MacOS you can do the following:

```
root@local $ ssh-copy-id deployer@remote
```

## Setting up Capistrano variables

Set the variables in `config/deploy.rb`, for instance:

```
/* config/deploy.rb */

# config valid only for current version of Capistrano
lock '3.4.0'

set :application, 'meanshop'
set :repo_url, 'git@github.com:amejiasrosario/meanshop.git'
set :user, 'deployer'
set :node_version, '0.12.7'
set :pty, true
set :forward_agent, true
set :linked_dirs, %w{node_modules}

namespace :deploy do
  # after :deploy, 'app:default'
  # after :deploy, 'nginx:default'
  # before 'deploy:reverted', 'app:default'end
```

The production server settings are done as follows:

```
/* config/deploy/production.rb */

server '128.0.0.0', user: 'deployer', roles: %w{web app db},
  private_ip: '10.0.0.0', primary: true

set :default_env, {
  NODE_ENV: 'production',
  path: "/home/#{fetch(:user)}/.nvm/versions/node/#{fetch
    (:node_version)}/bin:$PATH"
}
```

The next step is to forward our SSH keys to our server by running:

```
ssh-add ~/.ssh/id_rsa
ssh-add -L
```

Finally, you can deploy the application code to the server by running:

```
cap production deploy
```

If everything goes well, the application will be deployed to `/var/www/meanshop/current`.

Note: Refer to the previous chapter to install NodeJS, MongoDB, pm2, grunt-cli, and all the required components in only one server:

```
$ sudo apt-get update
$ sudo apt-get install -y build-essential openssl libssl-dev pkg-config
git-core mongodb ruby
$ curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.29.0/
install.sh | bash
$ source ~/.bashrc
$ nvm install 0.12.7
$ nvm use 0.12.7
$ nvm alias default 0.12.7
$ npm install grunt-contrib-imagemin
$ npm install -g grunt-cli bower pm2
$ sudo gem install sass
```

## Capistrano tasks

Time to automate the deployment! Capistrano has a default task which does the following:

1. Create a file structure on the remote server.
2. Set up ENV variables, create symlinks, release version, and so on.
3. Check out the Git repository.
4. Clean up.

We can use hooks to add tasks to this default workflow. For instance, we can run `npm install`, `build assets`, and `start servers` after Git is checked out.

## Adding new tasks

Let's add a new task, `app.rake`, which prepares our application for serving the client, and also updates the servers one by one (zero-downtime upgrade). First, let's uncomment the scripts in `config/deploy` that invoke the `app:default` task (in the `app.rake` script). And now, let's add `app.rake`:

```
# lib/capistrano/tasks/app.rake

namespace :app do
  desc 'Install node dependencies'
  task :install do
    on roles :app do
      within release_path do
        execute :npm, 'install', '--silent', '--no-spin'
        execute :bower, 'install', '--config.interactive=false',
          '--silent'
        execute :npm, :update, 'grunt-contrib-imagemin'
        execute :grunt, 'build'
      end
    end
  end

  desc 'Run the apps and also perform zero-downtime updates'
  task :run do
    on roles :app do |host|
      null, app1, app2 = capture(:pm2, 'list', '-m').split('+---')
      if app1 && app2 && app1.index('online') && app2.index('online')
        execute :pm2, :restart, 'app-1'
        sleep 15
        execute :pm2, :restart, 'app-2'
      else
        execute :pm2, :kill
        template_path = File.expand_path('../templates/pm2.json.erb',
          __FILE__)
        host_config = ERB.new
          (File.new(template_path).read).result(binding)
        config_path = "/tmp/pm2.json"
        upload! StringIO.new(host_config), config_path
        execute "IP=#{host.properties.private_ip}", "pm2",
          "start", config_path
      end
    end
  end

  task default: [:install, :run]
end
```



Don't worry too much if you don't understand everything that's going on here. The main points about `app.rake` are:

- `app:install`: This downloads the `npm` and the `bower` package, and builds the assets.
- `app:run`: This checks if the app is running and if it is going to update one node instance at a time at an interval of 15 seconds (zero-downtime). Otherwise, it will start both the instances immediately.



More information about other things that can be done with Rake tasks can be found at <https://github.com/ruby/rake>, as well as the Capistrano site at <http://capistranorb.com/documentation/getting-started/tasks/>.

Notice that we have a template called `pm2.json.erb`; let's add it:

```
/* lib/capistrano/tasks/templates/pm2.json.erb */

{
  "apps": [
    {
      "exec_mode": "fork_mode",
      "script": "<%= release_path %>/dist/server/app.js",
      "name": "app-1",
      "env": {
        "PORT": 8080,
        "NODE_ENV": "production"
      },
    },
    {
      "exec_mode": "fork_mode",
      "script": "<%= release_path %>/dist/server/app.js",
      "name": "app-2",
      "env": {
        "PORT": 8081,
        "NODE_ENV": "production"
      },
    },
  ]
}
```

## Preparing Nginx

This time we are using Nginx as a load balancer between our two node instances and the static file server. Similar to `app.rake`, we are going to add new tasks that install Nginx, set up the config file, and restart the service:

```
# lib/capistrano/tasks/nginx.rake

namespace :nginx do
  task :info do
    on roles :all do |host|
      info "host #{host}:#{host.properties.inspect}"
      (#{host.roles.to_a.join}): #{capture(:uptime)}"
    end
  end

  desc 'Install nginx'
  task :install do
    on roles :web do
      execute :sudo, 'add-apt-repository', '-y', 'ppa:nginx/stable'
      execute :sudo, 'apt-get', '-y', 'update'
      execute :sudo, 'apt-get', 'install', '-y', 'nginx'
    end
  end

  desc 'Set config file for nginx'
  task :setup do
    on roles :web do |host|
      template_path = File.expand_path('../templates/nginx.conf.erb',
        __FILE__)
      file = ERB.new(File.new(template_path).read).result(binding)
      file_path = '/tmp/nginx.conf'
      dest = "/etc/nginx/sites-available/#{fetch(:application)}"
      upload! StringIO.new(file), file_path
      execute :sudo, :mv, file_path, dest
      execute :chmod, '0655', dest
      execute :sudo, :ln, '-fs', dest,
        "/etc/nginx/sites-enabled/#{fetch(:application)}"
    end
  end

  task :remove do
    on roles :web do
      execute :sudo, 'apt-get', :remove, '-y', :nginx
    end
  end
end
```

```
%w[start stop restart status].each do |command|
  desc "run #{command} on nginx"
  task command do
    on roles :web do
      execute :sudo, 'service', 'nginx', command
    end
  end
end

desc 'Install nginx and setup config files'
task default: [:install, :setup, :restart]
end
```

We also need to add the new template for Nginx config:

```
# lib/capistrano/tasks/templates/nginx.conf.erb

upstream node_apps {
  ip_hash;
  server <%= host.properties.private_ip %>:8080;
  server <%= host.properties.private_ip %>:8081;
}

server {
  listen 80;
  server_name localhost; # or your hostname.com
  root <%= release_path %>/dist/public;
  try_files $uri @node;

  location @node {
    proxy_pass http://node_apps;
    proxy_http_version 1.1;
    # server context headers
    proxy_set_header Host $host;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    # headers for proxying a WebSocket
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection 'upgrade';
    proxy_set_header Host $host;
    proxy_cache_bypass $http_upgrade;
  }
}
```

Run `cap production deploy`, and after it finishes, you will see the app running in the public IP!

There are three main points of interest in this configuration: load balancing, static file server, and WebSockets.

## Load balancing

Nginx has different strategies for load balancing applications:

- **Round-robin:** The requests are distributed on the servers in the same order as defined. When one request reaches the last server, the next one goes to the first server.
- **Least-connected:** The requests are distributed on the basis of the number of connections. The next request is sent to the server with the least connections.
- **IP-hash:** The requests are distributed based on the IP address. This ensures that the client always uses the same server on each request. This is useful for sticky clients using WebSockets.

We are going to use `ip_hash` because of our WebSocket/SocketIO requirements.

## Static file server

Static assets such as images, JS, and CSS files are not changed very often in production environments. So, they can be safely cached and served directly from Nginx without having to hit the node instances:

```
root /home/deployer/meanshop/current/dist/public;  
try_files $uri @node;
```

Nginx will look for a static file in the file system first (root path). If it doesn't find it, Nginx will assume that it is a dynamic request, and hand it off to the node instances.

## WebSockets

We are using **WebSockets (WS)** to establish a bidirectional communication between the servers and the clients. This allows our store application to have realtime updates. For that, we have headers in the configuration that advises the clients to upgrade from HTTP 1.0 to HTTP 1.1 to enable the WS connections.

## Summary

In this last chapter, we implemented the last set of user stories for managing all products and users. Moreover, we provided the framework to continue adding new features to our application in production without interrupting the service (zero-downtime upgrades).

We have now reached the end of the book. It has been a fun ride! I wish you all the best with your e-commerce application!

# Index

## A

**Access Token** 104

**Agile Manifesto**

URL 17

**Angular**

product image files, uploading 85-87

**AngularJS**

about 4, 22

client-side structure 23

controllers 28, 29

CRUD functionality, adding for

products 35

debugger 8

directives 24

modules 25

RESTful APIs, testing 92

routing, with AngularUI router 26, 27

scopes 28, 29

templates 29, 30

tools, installing 6-8

unit testing 92

**AngularUI router**

used, for routing 26, 27

**ApacheBench (ab)** 169

**application**

deploying, in multi-server environment 165

environments 158

**application, deploying to cloud**

about 162

Cloud Servers 163

Platform as a service (PaaS) 162

Virtual Private Server (VPS) 163

**applications, deploying in multi-server environment**

about 165

Nginx server 168, 169

NodeJS application 166, 167

**applications, splitting**

additional services server(s) 161

caching/reverse proxy server(s) 161

database replication server(s) 161

database server(s) 161

load balancer server(s) 161

**authentication**

Facebook, using 119-121

Google, using 122, 123

Twitter, using 121

**authentication, strategies**

about 100

OAuth authentication 103, 104

session-based authentication 100, 101

token-based authentication 101, 102

## B

**Batarang**

about 8

URL 8

**Behavior Driven Development (BDD)**

testing 74

**BluebirdJS promise package**

URL 73

**Bower**

packages 12

## **Braintree**

- API keys 132
- API keys, URL 132
- authentication, setting up 132
- endpoint, setting up 132
- gateway 133
- gateway, using in controllers 133
- GET clientToken 135
- integrating, with checkout page 129-131
- POST checkout 135
- URL 130

## **C**

### **Capistrano**

- about 185, 186
- files and directories 186
- installing 185
- load balancing applications 193
- new tasks, adding 189, 190
- Nginx, preparing 191-193
- static file server 193
- tasks 188
- URL 185
- variables, setting up 187, 188
- WebSockets (WS) 193

### **checkout page**

- integrating, with Braintree 129-131

### **client-side authentication**

- about 104
- management 105-107
- signing up process 108-110

### **client-side structure**

- in meanshop project 23

### **CommonJS Modules**

- about 62
- exports 62
- module.exports object 62
- require function 62

### **components, project structure**

- about 10
- Bower packages 12
- package managers 11
- testing 10
- tools/libraries 11

### **config method 26**

### **connect-multiparty 85**

### **consumer key 104**

### **Content Delivery Network (CDN) 6**

### **controllers, AngularJS 28, 29**

### **Cross-site request forgery (CSRF) 102**

### **CRUD (create-read-update-delete) functionality 35**

### **CRUDing, with Mongoose**

- about 52
- data, retrieving 55
- delete methods 56
- product, creating 54
- schemas 52, 53
- values, updating 56

## **D**

### **Data-Driven Applications 21**

### **Dependency Injection (DI) 31**

### **development, testing, acceptance, and production (DTAP)**

- about 158
- acceptance (triage, staging) 158
- development 158
- production 158
- testing 158

### **Digital Ocean**

- about 163-165
- deployment, URL 170
- URL 163

### **directives, AngularJS**

- about 24
- URL 30

## **E**

### **e-commerce applications**

- back office 15, 16
- home page 13
- marketplace 14
- previewing 12
- requisites 17

### **e-commerce MVP**

- factories, using 31
- filters, using 34
- laying out 30
- marketplace, creating 33, 34
- products, creating 30
- services, using 31

## **end-to-end testing**

- about 95
- database, cleaning 96, 97
- for local authentication 118
- URL 95

## **exports**

- versus module.exports 64

## **ExpressJS**

- about 4
- bootstrapping 69-72
- installing 5
- routes, exploring 72, 73

## **F**

### **Facebook**

- URL 119
- used, for authentication 119-121

### **factories**

- products factory, creating 32, 33
- using 31

### **factory method 26**

### **filters**

- using 34

## **G**

### **Google**

- used, for authentication 118-123

## **H**

### **Heroku**

- URL 162

### **HTTP benchmarking tools**

- about 169
- ApacheBench (ab) 169

## **I**

### **ideas, features**

- authorization 182
- monitoring 182
- notifications 182
- reports 182
- statistics 182
- wishlist 182

## **installations**

- AngularJS debugger 8
- AngularJS tools 6-8
- ExpressJS 5
- MEAN component 4
- MongoDB 6
- NodeJS 4, 5

## **instance methods 57**

## **interceptors**

- URL 108

## **J**

### **Jasmine 10**

### **JSON Web Token (JWT)**

- about 102, 111
- advantages 102
- URL 102

## **K**

### **Karma 10, 92**

## **L**

### **Linux, Apache, MySQL, and PHP (LAMP) 2**

### **load balancing applications strategies,**

#### **Nginx**

- IP-hash 193
- least-connected 193
- round-robin 193

### **local authentication**

- about 117, 118
- end-to-end tests 118

## **M**

### **MEAN component**

- AngularJS debugger 8
- AngularJS tools, installing 6-8
- ExpressJS, installing 5
- installing 4
- MongoDB, installing 6
- NodeJS, installing 4, 5

### **meanshop**

- URL 140

### **MEANshop. See e-commerce applications**



## **MEAN stack**

- about 1
- AngularJS 4
- disadvantages 2
- ExpressJS 4
- MongoDB 4
- NodeJS 3

## **middleware 4, 60**

## **Minimum Viable Product (MVP) 17**

## **Mocha**

- about 74
- URL 74

## **Model-View-Controller (MVC) 21**

## **modules, AngularJS 25, 26**

## **MongoDB**

- about 4, 48
- and CLI 48
- installing 6
- mongo 48
- mongod 48
- SQL knowledge, mapping to 49
- URL 6

## **MongoDB, ExpressJS, AngularJS, and NodeJS stack. *See* MEAN stack**

## **Mongoose**

- used, for CRUDing 52

## **Mongoose, advanced features**

- exploring 57
- instance methods 57
- middleware 60
- static methods 57
- validations 58
- virtuals 58

## **Mongoose models**

- about 61
- commonJS modules 62, 63
- current models 62
- server folder 61
- user model 64, 65

## **multi-server environment**

- applications, deploying 165

# **N**

## **navigation bar**

- search textbox, adding 141, 142

## **navigation functionality**

- controllers, setting up 152
- custom \$resource methods, adding 152
- implementing 151
- routes, setting up 152
- working 153
- working, on client-side 153

## **new feature**

- AngularJS testing 179-181
- backlog 182
- HTML page, implementing 177, 178
- planning 175
- solution, implementing 177
- testing 179
- wire framing 176

## **new version, app**

- Capistrano 185
- Capistrano tasks 188
- deploying 182
- server, preparing 187
- zero-downtime deployments 183
- zero-downtime production server, setting up 184

## **ngCart**

- about 125
- directives, using 127
- installing 126

## **ngCart directives**

- Add to cart button 127
- summary, adding to cart 128

## **ngMock**

- about 92
- URL 92

## **node**

- file upload, handling 89, 90

## **NodeJS**

- about 3
- installing 4
- scaling, with production architecture 170
- testing 74

## **node modules (npm) 111**

## **Node Version Manager (NVM)**

- about 4
- URL 5

## O

### OAuth authentication

- about 103
- working 104

### Open Web Application Security Project (OWASP) 174

### optimizations, for production environments

- CDN, using for assets 158
- concatenation 158
- minification 158

### order

- creating 135
- model, modifying 136, 137
- model, testing 138-140
- sandbox account, using 140

## P

### package managers, AngularJS

- Bower 11
- NPM 11

### PassportJS

- initializing 112, 113
- URL 112
- used, for authentication 112

### Platform as a service (PaaS)

- about 162
- Heroku 162, 163

### Product API

- create action, testing 78
- delete action, testing 78
- implementing 77
- index action tests, performing 77
- product controller 78
- show action, testing 78
- testing 77

### product categories

- adding 143
- catalog controller 147, 148
- catalog model 148, 149
- categories, seeding 150, 151
- controllers, improving 146
- product models, improving 146
- products, seeding 150, 151
- routes 147, 148
- sidebar, adding 143-145
- URL 149

### product images

- files, uploading in Angular 85-87
- file upload, handling on Node 89, 90
- mock products, creating 91
- uploading 85

### product.integration.js file

- URL 78

### production architecture

- application environments 158
- building for 157
- micro-services 173
- multiple application instances, in one server 171
- multiple servers 172
- one server 170
- production environments, optimizations 158, 159
- services, splitting into multiple servers 172
- used, for scaling NodeJS 170

### production environments

- optimizations 158

### product model

- creating 75
- implementing 76
- testing 75, 76
- URL 75

### products

- controllers, setting up 37, 38
- CRUD functionality, adding with AngularJS 35
- routes, setting up 38, 39
- services, refactoring 35, 36
- templates, creating 39

### project structure

- about 9
- client folder 9
- components 10
- e2e files 9
- file structure 9, 10
- server directory 9

### Protractor

- about 10, 92
- URL 95

### PUT

- versus PATCH 69

## R

### Rake tasks

URL 190

### REpresentational State Transfer (REST) 67, 68

### Request Token 104

### requisites, e-commerce applications

about 17

defining 18

Minimum Viable Product (MVP) 17

user stories, URL 18

### RESTful APIs

\$resource methods, testing 94

Product Controller, testing 94

scaffolding 69

Services tests 93

testing, in AngularJS 92

### RESTful product service

implementing 81, 82

marketplace, building 83

product controller, adding 83, 84

### Robomongo

URL 54

### routes, ExpressJS

exploring 72, 73

### run method 26

## S

### scopes, AngularJS 28, 29

### search functionality

custom \$resource methods, adding 152

implementing 151

working 153

working, on client-side 154

### search textbox

adding, to navigation bar 141-143

### security 173, 174

### server-side authentication

about 111

authentication strategies

and routes 116, 117

local authentication 117, 118

local authentication, end-to-end tests 118

PassportJS, initializing 112, 113

PassportJS, using 112

user model 114, 115

### Service Oriented Architecture 173

### services

using 31

### session-based authentication

working 101

### shopping carts

about 125

ngCart directives, using 127

ngCart, installing 126

setting up 125

### Simple Object Access Protocol (SOAP) 67

### Single Page Applications (SPAs)

about 2, 4, 21, 26, 69

URL 27

### SinonJS

URL 94

### SQL knowledge, mapping to MongoDB

aggregators 51

collection 49

database 49

document 49

embedded docs 49

field 49

index 49

primary key 49

SQL queries 49, 50

### static methods 57

### stress tests

Heroku deployment, benchmarking 169

HTTP benchmarking tools 169

performing 169

VPS multi-server deployment,

benchmarking 170

### sub-documents

URL 138

### SuperTest

URL 77

using 77

## T

### templates, AngularJS 29, 30

### templates, products

creating 39

main page, styling 44, 45

partial forms, creating 40, 41

product edit, creating 42

- product list, creating 43, 44
- product new, creating 41
- product view, creating 42

### **Test Driven Development (TDD)**

- about 1
- testing 74

### **token-based authentication**

- about 101
- working 102

### **tools/libraries**

- AngularJS full-stack 11
- BabelJS 11
- Bootstrap 11
- EditorConfig 11
- Git 11
- GruntJS 11
- SocketIO 11
- Travis CI (Continuous Integration ) 11
- Yeoman (yo) 11

### **Twitter**

- URL 121
- used, for authentication 118-122

## **U**

**ui-router** 25, 26

**Uniform Resource Identifier (URI)** 68

### **unit testing**

- about 92
- ngMock 92
- setting up 93

**URL slugs** 146

## **V**

### **validations, Mongoose**

- about 58
- built-in validations 59
- custom validations 59

### **Virtual Private Server (VPS)**

- about 162, 163
- and Cloud Servers 163
- Digital Ocean 163-165

**virtuals** 58

## **W**

### **web application**

- scaling 159
- scaling, horizontal scale 159-161
- scaling, vertical scale 159, 160

### **Web Application Security Testing Cheat Sheet**

- URL 173

**Web Services Description Language  
(WSDL)** 67

**wireframes** 176





## Thank you for buying **Building an E-Commerce Application with MEAN**

### About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at [www.packtpub.com](http://www.packtpub.com).

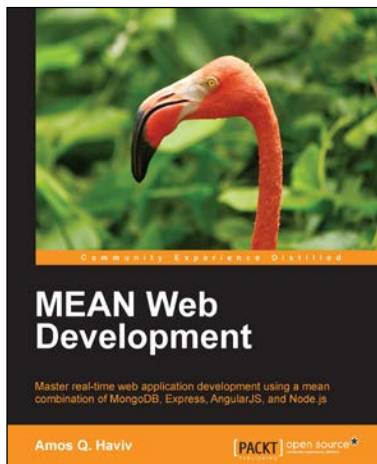
### About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



## MEAN Web Development

ISBN: 978-1-78398-328-5

Paperback: 354 pages

Master real-time MEAN web application development and learn how to construct a MEAN application using a combination of MongoDB, Express, AngularJS, and Node.js

1. Learn how to construct a fully functional MEAN application by using its components along with the best third-party modules.
2. Harness the power of the JavaScript ecosystem to effectively run, build, and test your MEAN application.
3. Gain a deep, practical understanding of real-time web application development through real-world examples.



## Building E-commerce Sites with VirtueMart Cookbook

ISBN: 978-1-78216-208-7

Paperback: 310 pages

Over 90 recipes to help you build an attractive, profitable, and fully-featured e-commerce store with VirtueMart

1. Get to grips with VirtueMart and build an attractive store powered by Joomla!
2. Increase the visibility of your store with SEO and product descriptions.
3. Keep your store profitable by configuring tax, shipping and orders.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



## WooCommerce Cookbook

ISBN: 978-1-78439-405-9

Paperback: 248 pages

Create, design, and manage your own personalized online store with WooCommerce, the fastest growing e-commerce platform

1. Get your online store up and running in no time.
2. Dozens of simple recipes to setup and manage your store.
3. Easy to understand code samples that can help you customize every tiny detail and take your store to the next level.



## Mastering MEAN Web Development [Video]

ISBN: 978-1-78439-506-3

Duration: 03:36 hours

Everything you need to know to build flawless and robust websites with the MEAN stack

1. Combine Node.js and MongoDB with HTML, CSS, and JavaScript to write stunning frontend code.
2. Write automated unit, integration, and end-to-end tests to see your code work the way it should.
3. Use reusable components such as Angular directives to write great code in no time and deploy a live website to prove it!.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles