

Tomasz Dyl, Kamil Przeorski

Mastering Full-Stack React Web Development

Dynamic and forward-thinking JavaScript web
development



Packt>

Mastering Full-Stack React Web Development

Dynamic and forward-thinking JavaScript web development

Tomasz Dyl
Kamil Przeorski



BIRMINGHAM - MUMBAI

Mastering Full-Stack React Web Development

Copyright © 2017 Packt Publishing

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

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, 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: April 2017

Production reference: 1240417

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

ISBN 978-1-78646-176-6

www.packtpub.com

Credits

Authors

Tomasz Dyl

Kamil Przeorski

Reviewer

Samer Buna

Commissioning Editor

Amarabha Banerjee

Acquisition Editor

Smeet Thakkar

Content Development Editor

Parshva Sheth

Technical Editor

Prajakta Mhatre

Copy Editors

Safis Editing

Madhusudan Uchil

Project Coordinator

Sheeja Shah

Proofreader

Safis Editing

Indexer

Tejal Daruwale Soni

Production Coordinator

Melwyn D'sa

About the Authors

Tomasz Dyl has worked with React since 2014. He's the technical lead developer for React Poland. The team focuses on cross-platform full-stack development using React and Node (<http://reactpoland.com>). He has worked on over 20 different React and Node projects for his clients since 2014. React Poland also works on the React Community platform (<https://reactjs.co>).

Kamil Przeorski has worked with React since 2014. Along with Tomasz, he is the co-founder of React Poland--the leading company for React developers. In the meantime, besides building the best React team in Poland, he runs two different Facebook groups called Node.js Poland (<https://www.facebook.com/groups/nodejsPL/>) and ReactJS, React Native, GraphQL (<https://web.facebook.com/groups/reactjs.co/>)--both groups have around 2000 members.

About the Reviewer

Samer Buna is a coder, mentor, and technical content author. He has a master's degree in Information Security, and years of progressive experience and success creating tailored solutions for businesses in many industries.

Samer is passionate about everything JavaScript, and he loves exploring new libraries. His favorite technical stack is Node.js for the backend and React.js for the frontend.

Samer has authored a few books and online courses about React and GraphQL. You can follow him on Twitter at <https://twitter.com/samerbuna>, and you can read more of what he writes at <https://edgecoders.com/>.

www.PacktPub.com

For support files and downloads related to your book, please visit 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 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 for more details.

At 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://www.packtpub.com/mapt>

Get the most in-demand software skills with Mapt. Mapt gives you full access to all Packt books and video courses, as well as industry-leading tools to help you plan your personal development and advance your career.

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

Customer Feedback

Thanks for purchasing this Packt book. At Packt, quality is at the heart of our editorial process. To help us improve, please leave us an honest review on this book's Amazon page at <https://www.amazon.com/dp/1786461765>.

If you'd like to join our team of regular reviewers, you can e-mail us at customerreviews@packtpub.com. We award our regular reviewers with free eBooks and videos in exchange for their valuable feedback. Help us be relentless in improving our products!

For Kate and Anna

- Kamil Przeorski

Table of Contents

Preface	1
Chapter 1: Configuring Full-Stack with Node.js, Express.js, MongoDB, Mongoose, Falcor, and Redux	6
More about our technical stack	7
Environment preparation	8
NVM and Node installation	8
MongoDB installation	9
Robomongo GUI for MongoDB	9
Running MongoDB and viewing our collections in the Robomongo GUI	10
Importing the first example collection into the database	11
Importing the articles to MongoDB	13
Server setup with Node.js and Express.js	13
Working on our server (server.js)	16
Mongoose and Express.js	17
A summary of how to run the project	18
Redux basic concepts	21
The single immutable state tree	21
Immutability - actions and state tree are read-only	22
Pure and impure functions	23
The reducer function	23
First reducer and webpack config	23
The rest of the important dependencies installation and npm dev script	27
Working on src/app.js and src/layouts/PublishingApp.js	28
Wrapping up React-Redux application	29
Finishing our first static publishing app	31
Falcor's basic concepts	33
What is Falcor and why do we need it in our full-stack publishing app?	33
Tight coupling and latency versus one model everywhere	34
No more tight coupling on client and server side	35
Client-side Falcor	35
A summary of client-side Falcor + Redux	42
Moving Falcor's model to the backend	42
Configuring Falcor's router (Express.js)	46
Second route for returning our two articles from the backend	47

Final touch to make full-stack Falcor run	49
Adding MongoDB/Mongoose calls based on Falcor's routes	51
Double-check with the server/routes.js and package.json	53
Our first working full-stack app	55
Summary	56
Chapter 2: Full-Stack Login and Registration for Our Publishing App	57
Structure of JWT token	59
New MongoDB users collection	60
Explanation	61
Importing the initPubUsers.js file into MongoDB	61
Working on the login's falcor-route	62
Creating a falcor-router's login (backend)	63
How the call routes work	64
Separating the DB configs - configMongoose.js	65
Explanation	66
Improving the routes.js file	66
Explanation	67
Checking to see if the app works before implementing JWT	67
Creating a Mongoose users' model	68
Explanation	68
Implementing JWT in the routesSession.js file	68
Explanation	70
Successful login on falcor-route	71
Explanation	71
Frontend side and Falcor	72
The CoreLayout component	72
The LoginView component	74
A root's container for our app	74
Remaining configuration for configureStore and rootReducer	76
Last tweaks in layouts/PublishingApp.js before running the app	78
Last changes in src/app.js before running the app	79
Screenshots of our running app	80
Home page	80
Login view	81
Working on the login form that will call the backend in order to authenticate	81
Working on LoginForm and DefaultInput components	82
Explanation	83
LoginForm and making it work with LoginView	84

Improving the src/views/LoginView.js	85
Making DashboardView's component	87
Finishing the login's mechanism	89
Handling successful logins in the LoginView's component	90
Explanation	90
A few important notes about DashboardView and security	91
Starting work on the new editor's registration	91
Adding register's falcor-route	92
Explanation	94
Frontend implementation (RegisterView and RegisterForm)	95
RegisterView	96
Summary	101
Chapter 3: Server-Side Rendering	102
When the server side is worth implementing	102
Mocking the database response	103
The handleServerSideRender function	106
Double-check server/server.js	110
Frontend tweaks to make the server-side rendering work	112
Summary	117
Chapter 4: Advanced Redux and Falcor on the Client Side	118
Focusing on the app's frontend	118
Backend wrap-up before frontend improvement	119
Improving handleServerSideRender	120
Changing routes in Falcor (frontend and backend)	121
Our website header and articles list need improvements	121
New ArticleCard component	123
Dashboard - adding an article button, logout, and header improvements	128
Important note before creating a frontend add article feature	130
The AddArticleView component	131
Modifying DashboardView	135
Starting work on our WYSIWYG	137
Stylesheet for the draft-js WYSIWYG	138
Coding a draft-js skeleton	140
Improving the views/articles/AddArticleView component	143
Adding more formatting features to our WYSIWYG	145
Pushing a new article into article reducer	150
MapHelpers for improving our reducers	151
The CoreLayout improvements	154
Why Maps over a JS object?	155
Improving PublishingApp and DashboardView	156

Tweaks to AddArticleView	157
The ability to edit an article (the EditArticleView component)	160
Let's add a dashboard link to an article's edition	162
Creating a new action and reducer	162
Edit mode in src/components/articles/WYSIWYGeditor.js	163
Improvements in EditArticleView	165
EditArticleView's render improvements	167
Deleting an article's feature implementation	169
Summary	175
Chapter 5: Falcor Advanced Concepts	176
The problem that Falcor aims to solve	177
Virtual JSON - one model everywhere	178
Falcor versus Relay/GraphQL	178
Big-picture similarities	179
Technical differences - overview	179
Improving our application and making it more reliable	181
Securing the auth required routes	181
JSON Graph and JSON envelopes in Falcor	181
Improving our Falcor code on the frontend	182
Improving server.js and routes.js	184
Falcor's sentinel implementation	188
The \$ref sentinel	188
Detailed example of the \$ref sentinel	189
Improving our articles' numberOfLikes with \$ref	191
Practical use of \$ref in our project	192
Mongoose config improvements	192
The server/routes.js improvements	193
JSON Graph atoms	196
Improving the articles[<code>{integers}</code>] route	197
New route in server/routes.js: articles.add	199
Frontend changes in order to add articles	201
Important note about route returns	204
Full-stack - editing and deleting an article	205
Deleting an article	206
Frontend - edit and delete	207
Securing the CRUD routes	209
The \$error sentinel basics	209
DRY error management on the client side	210
Tweaks - FalcorModel.js on the frontend	213

Backend implementation of the \$error sentinel	215
Testing our \$error-related code	215
Cleaning up \$error after a successful test	218
Wrapping up the routes' security	218
What routes to secure	219
Summary	223
Chapter 6: AWS S3 for Image Upload and Wrapping Up Key Application Features	224
AWS S3 - an introduction	225
Generating keys (access key ID and secret key)	225
IAM	227
Setting up S3 permissions for the user	230
Coding the image upload feature in the AddArticleView	237
Environment variables in Node.js	238
Improving our Mongoose article schema	239
Adding routes for S3's upload	240
Creating the ImgUploader component on the frontend	243
Wrapping up the ImgUploader component	244
AddArticleView improvements	247
Some remaining tweaks for PublishingApp, ArticleCard, and DashboardView	252
Improving the ArticleCard component	254
Improving the DashboardView component	257
Editing an article's cover photo	258
Adding the ability to add/edit the title and subtitle of an article	263
AddArticleView improvements	264
Ability to edit an article title and subtitle	267
ArticleCard and PublishingApp improvements	271
Dashboard improvement (now we can strip the remaining HTML)	274
Summary	276
Chapter 7: The MongoDB Deployment on mLab	278
mLab overview	278
Replica set connections and high availability	280
MongoDB failover	280
Free versus paid plan in mLab	281
The new mLab's account and node	281
Creating the database's user/password and other configurations	284
Config wrap up	286
Summary	287

Chapter 8: Docker and the EC2 Container Service	288
Docker installation with Docker Toolbox	289
Docker Hub - an hello world example	291
Dockerfile example	294
Modifications to our codebase in order to create it	294
Working on the publishing app Docker image	297
Building the publishing app container	299
Running the publishing app container locally	301
Debugging a container	302
Pushing a Docker container to a remote repository	303
A summary of useful Docker commands	306
Introduction to Docker on AWS EC2	307
Manual approach - Docker on EC2	307
Basics - launching an EC2 instance	308
SSH access via PuTTY - Windows users only	316
Connecting to an EC2 instance via SSH	318
Basics of ECS - AWS EC2	323
Working with ECS	323
Step 1 - creating a task definition	324
Step 2 - configuring the service	327
Step 3 - configuring the cluster	329
Step 4 - reviewing	331
Launch status	332
Finding your load balancer address	333
AWS Route 53	337
Summary	346
Chapter 9: Continuous Integration with Unit and Behavioral Tests	347
When to write unit and behavioral tests	347
React conventions	348
Karma for testing	348
How to write unit and behavioral tests	349
What is Mocha and why do you need it?	352
Testing CoreLayout step-by-step	353
Continuous integration with Travis	356
Summary	356
Index	358

Preface

The innovations in the JavaScript programming language in recent years have been vital. For example, since 2009 the rise of Node granted developers the ability to use the same programming language in the browser and on the backend. The environmental changes haven't slowed down in 2017 as well. This book will teach you about some new and hot concepts that speed up the full-stack development process even more.

In 2016 and 2017, there has been an even greater demand for making apps even faster with full-stack technologies such as Falcor or GraphQL. This book isn't just a guide on how to expose the API endpoints in Node and start consuming them with your client-side application. You will learn how to use the newest technology from Netflix, called Falcor. Beside that, you will learn how to set up a project with the use of React and Redux. In this book, you will find a huge tutorial for building a full-stack app from scratch with the use of the Node.js, Express.js, MongoDB, Mongoose, Falcor, and Redux libraries. You will also learn how to deploy your application with the use of Docker and Amazon's AWS services.

What this book covers

Chapter 1, Configuring Full-Stack with Node.js, Express.js, MongoDB, Mongoose, Falcor, and Redux, takes you through the initial setup of the application from scratch. It helps you understand how different libraries from npm make a usable full-stack React starter kit.

Chapter 2, Full-Stack Login and Registration for Our Publishing App, guides you through how to set up the JWT token in order to implement a basic full-stack authentication mechanism.

Chapter 3, Server-Side Rendering, teaches you how to add server-side rendering to the application, which is helpful for quicker application execution and search engine optimization.

Chapter 4, Advanced Redux and Falcor on the Client Side, shows you how to add more advanced features to your application such as an integrated WYSIWYG editor and the Material-UI components that extend the application from the app user's perspective.

Chapter 5, Falcor Advanced Concepts, takes you through a more detailed backend-related development guide to Falcor and its backend, Falcor-Router.

Chapter 6, *AWS S3 for Image Upload and Wrapping Up Key Application Features*, guides you through the publishing app's articles cover photos uploading process.

Chapter 7, *The MongoDB Deployment on mLab*, teaches you how to prepare a remote database for your application.

Chapter 8, *Docker and the EC2 Container Service*, teaches you AWS/Docker setup.

Chapter 9, *Continuous Integration with Unit and Behavioral Tests*, shows you what you will need to use in order to prepare your CI and tests for the published application.

What you need for this book

This book was written with use of the macOS El Capitan and Sierra. It was tested on Linux Ubuntu and a Windows 10 machine (some additional comments have been added concerning the differences between these three operating systems).

The rest of the toolset installation is shown in Chapter 1, *Configuring Full-Stack with Node.js, Express.js, MongoDB, Mongoose, Falcor, and Redux*.

Who this book is for

Do you want to build and understand the full-stack development from the ground up? Then this book is for you.

If you are a React developer who is looking for a way to improve your full-stack development skill sets, then you will be right at home as well. You will use the newest technologies to build your next full-stack publishing application from scratch. Make your first full-stack application with end-to-end guidance.

We assume in the book that you already have basic knowledge of the React library.

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: "In the project's directory, create a file called `initData.js`."

A block of code is set as follows:

```
[
  {
    articleId: '987654',
    articleTitle: 'Lorem ipsum - article one',
    articleContent: 'Here goes the content of the article'
  },
  {
    articleId: '123456',
    articleTitle: 'Lorem ipsum - article two',
    articleContent: 'Sky is the limit, the content goes here.'
  }
]
```

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

```
mkdir server
cd server
touch index.js
```

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: "Create a connection with defaults by clicking the **Create** link."



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 it.

To send us general feedback, simply e-mail 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.

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 for this book from your account at <http://www.packtpub.com>. 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.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

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

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

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Full-Stack-React-Web-Development>. We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output.

You can download this file from

https://www.packtpub.com/sites/default/files/downloads/MasteringFullStackReactWebDevelopment_ColorImages.pdf.

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 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, and we will do our best to address the problem.

1

Configuring Full-Stack with Node.js, Express.js, MongoDB, Mongoose, Falcor, and Redux

Welcome to *Mastering Full-Stack React Web Development*. In this book, you will create a universal full-stack application in JavaScript. The application that we are going to build is a publishing platform similar to those that are currently popular on the market, for example:

- Medium (<https://medium.com/>)
- WordPress (<https://wordpress.com/>)
- issuu (<https://issuu.com/>)

There are many smaller publishing platforms, and, of course, our application will have fewer features than the ones listed in the aforementioned list because we will only focus on the main features, such as publishing an article, editing an article, or deleting an article (the core features that you can use to implement your own ideas). Besides that, we will focus on building a robust application that can be built on because one of the most important things about these kinds of applications is scalability. Sometimes, a single article gets much more web traffic than the whole site put together (10,000 percent times more traffic is normal in the industry because, for instance, a single article could gain insane traction through social media).

The first chapter of this book is all about setting up the project's main dependencies.

Our focus for this chapter will include the following topics:

- Installation of **Node Version Manager (NVM)** for easier Node management
- Installation of Node and NPM
- Preparing MongoDB in our local environment
- Robomongo for Mongo's GUI
- Express.js setup
- Mongoose installation and configuration
- Initial React Redux structure for our client-side app
- Netflix Falcor on the backend and frontend as a glue and replacement for the old RESTful approach

We will use very modern app stacks that gained a lot of traction in 2015 and 2016—I am sure that the stack that you are going to learn throughout the book will be even more popular in years to come, as we in our company, *MobileWebPro.pl*, see huge spikes of interest in the technologies that are listed in the previous bullets. You will gain a lot from this book, and will catch up with the newest approaches to building robust, full-stack applications.

More about our technical stack

In this book, we assume that you are familiar with JavaScript (ES5 and ES6) and we will also introduce you to some mechanisms from ES7 and ES8.

For the client side, you will use React.js, which you must already be familiar with, so we won't discuss React's API in detail.

For data management on the client side, we will use Redux. We will also show you how to set up the server-side rendering with Redux.

For the database, you will learn how to use MongoDB alongside Mongoose. The second one is an object data modeling library that provides a rigorous modeling environment for your data. It enforces a structure, and at the same time it also allows you to keep the flexibility that makes MongoDB so powerful.

Node.js and Express.js are standard choices for a frontend developer to start a full-stack development. Express's framework has the best support for the innovative client backend data fetching mechanism created by **Netflix-Falcor.js**. We believe you will love Falcor because of its simplicity and the fact that it will save you so much time when doing full-stack development. We will explain in detail later in the book why it is so efficient to use this data fetching library instead of the standard process of building a RESTful API.

Generally, we will use an object notation (JSON) pretty much everywhere--with React as the library, JSON is heavily used for diffing the Virtual DOM (under the hood). Redux uses a JSON tree for its single state tree container as well. Netflix Falcor's library also uses an advanced concept called a virtual JSON graph (we will describe it in detail later). Finally, MongoDB is also a document-based database.

JSON everywhere--this setup will improve our productivity drastically, mainly because of Falcor, which is binding everything together.

Environment preparation

For starting up, you're going to need the following tools installed on your operating system:

- MongoDB
- Node.js
- NPM--installed automatically with Node.js

We strongly recommend using either Linux or OS X for development. For Windows users, we'd recommend setting up a virtual machine and doing the development part within it. For doing so, you can either use **Vagrant** (<https://www.vagrantup.com/>), which creates a virtual env process in the background with development taking place almost natively on Windows, or you can use Oracle's **VirtualBox** (<https://www.virtualbox.org/>) directly, and work within a virtual desktop, however the performance here is significantly lower than working native.

NVM and Node installation

NVM is a very handy tool for keeping different Node versions on your machine during development. Go to <https://github.com/creationix/nvm> for instructions if you don't have NVM installed on your system yet.

After you have NVM on your system, you can type the following:

```
$ nvm list-remote
```

This command lists all possible Node versions that are available. We will use Node v4.0.0 in our case, so you need to type the following in your terminal:

```
$ nvm install v4.0.0
$ nvm alias default v4.0.0
```

These commands will install Node version 4.0.0. and set it as default. We use NPM 2.14.23 during the book so you can check your version with the following command:

```
$ npm -v  
2.14.23
```

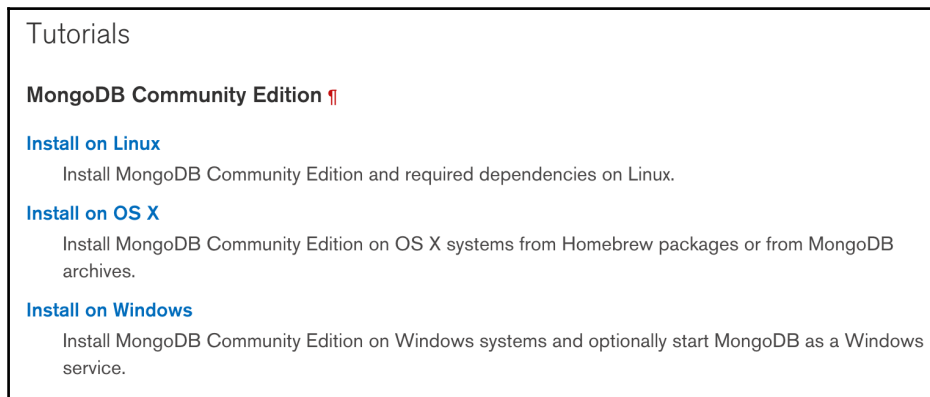
After you have the same versions of Node and NPM on your local machine, then we can start to set up the rest of the tools that we are going to use.

MongoDB installation

You can find all the MongoDB instructions at

<https://docs.mongodb.org/manual/installation/> under the **Tutorials** section.

The following is a screenshot from the MongoDB website:



The instructions and prepared packages for installing Node.js can be found at <https://nodejs.org>.

Robomongo GUI for MongoDB

Robomongo is a cross-platform desktop client that may be compared to MySQL or PostgreSQL for SQL databases.

When developing an app, it's good to have a GUI and be able to quickly review collections in our database. This is an optional step if you feel familiar with using shell for DB management, but it's helpful if it's your first step in working with databases.

To obtain Robomongo (for all operating systems), visit <https://robomongo.org/> and install one on your machine.

In our case, we will use version 0.9.0 RC4 of Robomongo.

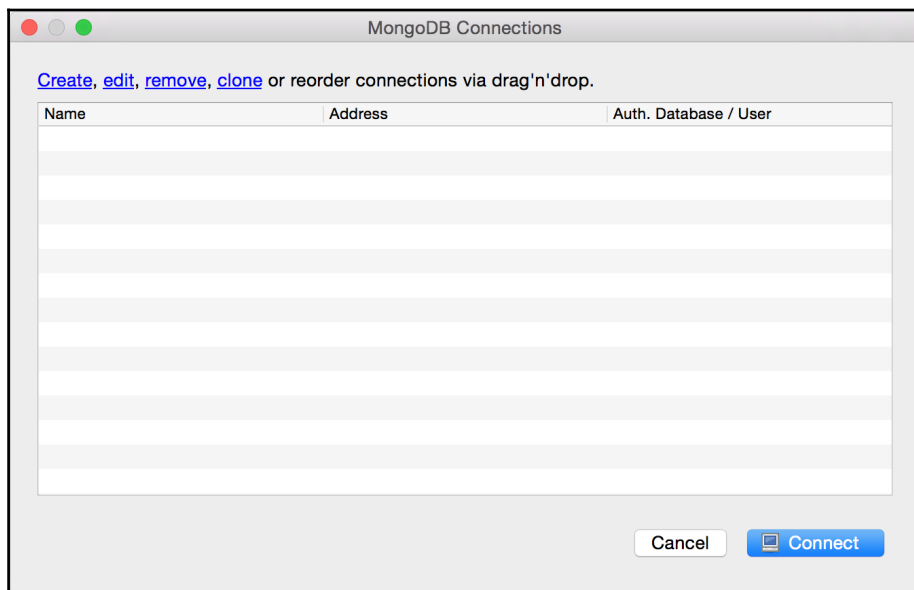
Running MongoDB and viewing our collections in the Robomongo GUI

After you have installed MongoDB and Robomongo on your machine, you need to run its daemon process, which listens to connections and delegates them to the database. To run the Mongo daemon process in your terminal, use the following command:

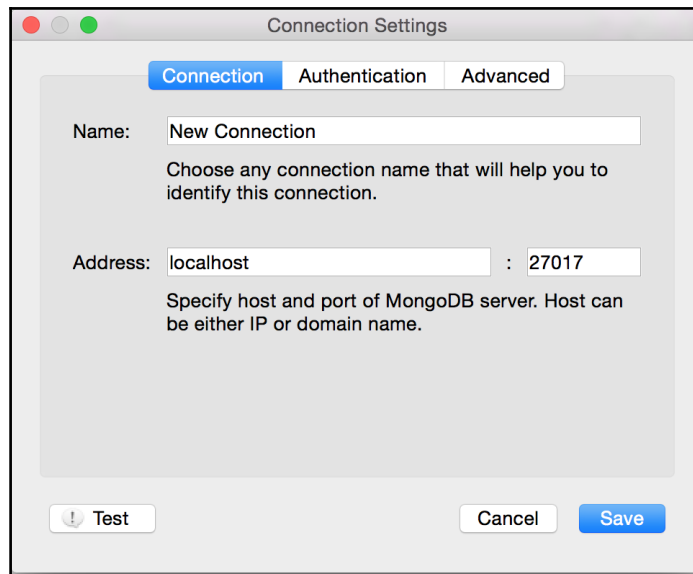
```
mongod
```

Then perform the following steps:

1. Open Robomongo's client--the following screen will appear:



2. Create a connection with defaults by clicking the **Create** link:



3. Pick a name for your connection and use port 27017, which is the default for databases, and click on **Save**.

At this point, you have a localhost database setup finished, and you can preview its content using the GUI client.

Importing the first example collection into the database

In the project's directory, create a file called `initData.js`:

```
touch initData.js
```

In our case, we are building the publishing app so it will be a list of articles. In the following code, we have an example collection of two articles in a JSON format:

```
[
  {
    articleId: '987654',
    articleTitle: 'Lorem ipsum - article one',
    articleContent: 'Here goes the content of the article'
  },
  {
    articleId: '123456',
    articleTitle: 'Lorem ipsum - article two',
    articleContent: 'Sky is the limit, the content goes here.'
  }
]
```

In general, we start from a mocked collection of articles--later we will add a feature to add more articles into MongoDB's collection, but for now we will stick with only two articles for the sake of brevity.

To list your localhost databases, open the Mongo shell by typing:

```
$ mongo
```

While in the Mongo shell, type:

```
show dbs
```

See the following for a full example:

```
Welcome to the MongoDB shell.
For interactive help, type "help".
For more comprehensive documentation, see
  http://docs.mongodb.org/
Questions? Try the support group
  http://groups.google.com/group/mongodb-user
Server has startup warnings:
2016-02-25T13:31:05.896+0100 I CONTROL [initandlisten]
2016-02-25T13:31:05.896+0100 I CONTROL [initandlisten] ** WARNING: soft
rlimits too low. Number of files is 256, should be at least 1000
> show dbs
local  0.078GB
>
```

In our example, it shows that we have one database in the localhost called `local`.

Importing the articles to MongoDB

In the following, we will use Terminal (the command prompt) in order to import the articles into the database. Alternatively, you can use Robomongo to do it via the GUI as well:

```
mongoimport --db local --collection articles --jsonArray initData.js --host=127.0.0.1
```



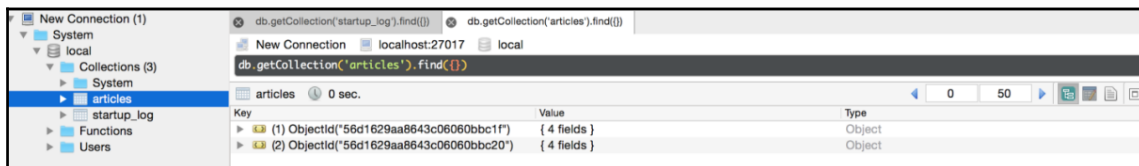
Remember that you need a new tab in your Terminal and `mongo import` will work while you are in the Mongo shell (Don't confuse it with the `mongod` process).

Then you shall see the following information in your terminal:

```
connected to: 127.0.0.1
imported 2 documents
```

In case you get the error `Failed: error connecting to db server: no reachable servers`, then make sure you have `mongod` running on the given host IP (127.0.0.1).

After importing those articles via the command line, you will also see this reflected in Robomongo:



Server setup with Node.js and Express.js

Once we have our article collection in MongoDB, we can start working on our Express.js server in order to work on the collection.

First, we need an NPM project in our directory:

```
npm init --yes
```

The `--yes` flag means that we will use the default settings for `package.json`.

Next, let's create an `index.js` file in the `server` directory:

```
mkdir server
cd server
touch index.js
```

In `index.js`, we need to add a `Babel/register` in order to get better coverage of the ECMAScript 2015 and 2016 specification. This will enable us to support such structures as `async` and `generator` functions, which are not available in the current version of Node.js by default.

See the following for the `index.js` file content (we will install Babel's dev dependencies later):

```
// babel-core and babel-polyfill to be installed later in that
//chapter
require('babel-core/register');
require('babel-polyfill');
require('./server');
```

Installing `express` and other initial dependencies:

```
npm i express@4.13.4 cors@2.7.1 body-parser@1.15.0--save
```

In the command, you can see `@4.13.4` after `express` and others. These are the versions of the libraries we're going to install, and we've picked it intentionally to make sure that it works well along side Falcor, but most probably you can skip these, and newer versions should work just as well.

We also need to install dev dependencies (we have spilled all `npm install` commands into separate for better readability):

```
npm i --save-dev babel@6.5.2
npm i --save-dev babel-core@6.6.5
npm i --save-dev babel-polyfill@6.6.1
npm i --save-dev babel-loader@6.2.4
npm i --save-dev babel-preset-es2015@6.6.0
npm i --save-dev babel-preset-react@6.5.0
npm i --save-dev babel-preset-stage-0@6.5.0
```

We need the `babel-preset-stage-0` is for ES7 features. The `babel-preset-es2015` and `babel-preset-react` are required for JSX and ES6 support.

Also, note that we install Babel to give our Node's server the ability to use ES6 features. We need to add the `.babelrc` file, so create the following:

```
$ [[you are in the main project's directory]]
$ touch .babelrc
```

Then open the `.babelrc` file and fill it with the following content:

```
{
  'presets': [
    'es2015',
    'react',
    'stage-0'
  ]
}
```

Remember that the `.babelrc` is a hidden file. Probably the best way to edit the `.babelrc` is to open the whole project in a text editor such as Sublime Text. Then you should be able to see all hidden files.

We also need the following libraries:

- `babel` and `babel-core/register`: This is the library for transpiling new ECMAScript functions into the existing version
- `cors`: This module is responsible for creating cross-origin requests to our domain in an easy way
- `body-parser`: This is the middleware for parsing the request's body

After this, your project's file structure should look like the following:

```
&boxvr;&boxh;&boxh; node_modules
&boxv;  &boxvr;&boxh;&boxh; ***
&boxvr;&boxh;&boxh; initData.js
&boxvr;&boxh;&boxh; package.json
&boxur;&boxh;&boxh; server
      &boxur;&boxh;&boxh; index.js
```

The `***` is a wildcard, which means that there are files required for our project, but we don't list them here as it would be too long.

Working on our server (server.js)

We will start working on the `server/server.js` file, which is new to our project, so we need to create it first with the following commands, in the `server` directory of your project:

```
touch server.js
```

The content for the `server/server.js` file is as follows:

```
import http from 'http';
import express from 'express';
import cors from 'cors';
import bodyParser from 'body-parser';

const app = express();
app.server = http.createServer(app);

// CORS - 3rd party middleware
app.use(cors());

// This is required by falcor-express middleware
//to work correctly with falcor-browser
app.use(bodyParser.json({extended: false}));

app.get('/', (req, res) => res.send('Publishing App Initial
Application!'));

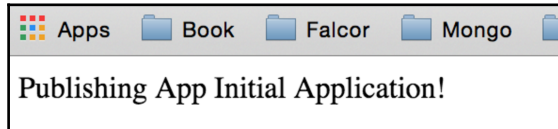
app.server.listen(process.env.PORT || 3000);
console.log(`Started on port ${app.server.address().port}`);
export default app;
```

These files use the `babel/register` library so that we can use ES6 syntax in our code. In the `index.js` file, we have an `http` module which comes from Node.js (https://nodejs.org/api/http.html#http_http). Next, we have `express`, `cors`, and `body-parser`.

Cors is middleware for dynamically or statically enabling **cross-origin resource sharing (CORS)** in Express applications--it will be useful in our development environment (we will delete it later for our production server).

Body-parser is middleware for HTTP's body parsing. It has some fancy settings that help us build the app faster.

This how our app looks at this stage of our development:



Mongoose and Express.js

At the moment, we have a simple working Express.js server. Now we have to add Mongoose to our project:

```
npm i mongoose@4.4.5 --save
```

Once we have installed Mongoose and a running MongoDB database in the background, we can import it to our `server.js` file and do the coding:

```
import http from 'http';
import express from 'express';
import cors from 'cors';
import bodyParser from 'body-parser';
import mongoose from 'mongoose';

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

const articleSchema = {
  articleTitle:String,
  articleContent:String
};

const Article = mongoose.model('Article', articleSchema, 'articles');
const app = express();
app.server = http.createServer(app);

// CORS - 3rd party middleware
app.use(cors());

// This is required by falcor-express middleware to work correctly
```

```
//with falcor-browser
app.use(bodyParser.json({extended: false}));

app.use(express.static('dist'));

app.get('/', (req, res) => {
  Article.find( (err, articlesDocs) => {
    const ourArticles = articlesDocs.map((articleItem) => {
      return &grave;<h2>${articleItem.articleTitle}</h2>
        ${articleItem.articleContent}&grave;;
    }).join('<br/>');

    res.send(&grave;<h1>Publishing App Initial Application!</h1>
      ${ourArticles}&grave;);
  });
});

app.server.listen(process.env.PORT || 3000);
console.log(&grave;Started on port ${app.server.address().port}&grave;);
export default app;
```

A summary of how to run the project

Make sure that you have MongoDB running in the background on your machine using the following command:

```
mongod
```

After you run the `mongod` command in your terminal (or PowerShell on Windows), then you should see something like the following in your console:

```
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] MongoDB starting : pid=3462 port=27017 dbpath=/data/db
64-bit host=Kamils-MBP
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] db version v3.2.6
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] git version: 05552b562c7a0b3143a729aaa0838e558dc49b25
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] OpenSSL version: OpenSSL 1.0.2h 3 May 2016
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] allocator: system
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] modules: none
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] build environment:
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] distarch: x86_64
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] target_arch: x86_64
2016-06-17T12:01:04.482+0200 I CONTROL [initandlisten] options: {}
2016-06-17T12:01:04.484+0200 I - [initandlisten] Detected data files in /data/db created by the 'mmapv1'
storage engine, so setting the active storage engine to 'mmapv1'.
2016-06-17T12:01:04.484+0200 W - [initandlisten] Detected unclean shutdown - /data/db/mongod.lock is not
empty.
2016-06-17T12:01:04.498+0200 I JOURNAL [initandlisten] journal dir=/data/db/journal
2016-06-17T12:01:04.499+0200 I JOURNAL [initandlisten] recover begin
2016-06-17T12:01:04.499+0200 I JOURNAL [initandlisten] recover lsn: 8153110
2016-06-17T12:01:04.499+0200 I JOURNAL [initandlisten] recover /data/db/journal/j._0
2016-06-17T12:01:04.500+0200 I JOURNAL [initandlisten] recover skipping application of section seq:40 < lsn:81
53110
2016-06-17T12:01:04.500+0200 I JOURNAL [initandlisten] recover skipping application of section seq:308215 < ls
n:8153110
2016-06-17T12:01:04.500+0200 I JOURNAL [initandlisten] recover skipping application of section seq:354946 < ls
n:8153110
2016-06-17T12:01:04.500+0200 I JOURNAL [initandlisten] recover skipping application of section seq:399361 < ls
n:8153110
2016-06-17T12:01:04.500+0200 I JOURNAL [initandlisten] recover skipping application of section seq:544991 < ls
n:8153110
2016-06-17T12:01:04.500+0200 I JOURNAL [initandlisten] recover skipping application of section seq:4142804 < l
sn:8153110
2016-06-17T12:01:04.500+0200 I JOURNAL [initandlisten] recover skipping application of section seq:4332201 < l
sn:8153110
2016-06-17T12:01:04.501+0200 I JOURNAL [initandlisten] recover skipping application of section seq:4853604 < l
sn:8153110
2016-06-17T12:01:04.501+0200 I JOURNAL [initandlisten] recover skipping application of section seq:4868074 < l
sn:8153110
2016-06-17T12:01:04.501+0200 I JOURNAL [initandlisten] recover skipping application of section more...
2016-06-17T12:01:04.502+0200 I JOURNAL [initandlisten] recover final skipped journal section had sequence numb
er 7857140
2016-06-17T12:01:04.502+0200 I JOURNAL [initandlisten] recover applying initial journal section with sequence
number 8153110
```

Before you run the server, make sure the `devDependencies` in your `package.json` file look like the following:

```
"devDependencies": {
  "babel": "6.5.2",
  "babel-core": "6.6.5",
  "babel-loader": "6.2.4",
  "babel-polyfill": "6.6.1",
  "babel-preset-es2015": "6.6.0",
  "babel-preset-react": "6.5.0",
  "babel-preset-stage-0": "6.5.0"
}
```

Before you run the server, make sure that the dependencies in your `package.json` look like the following:

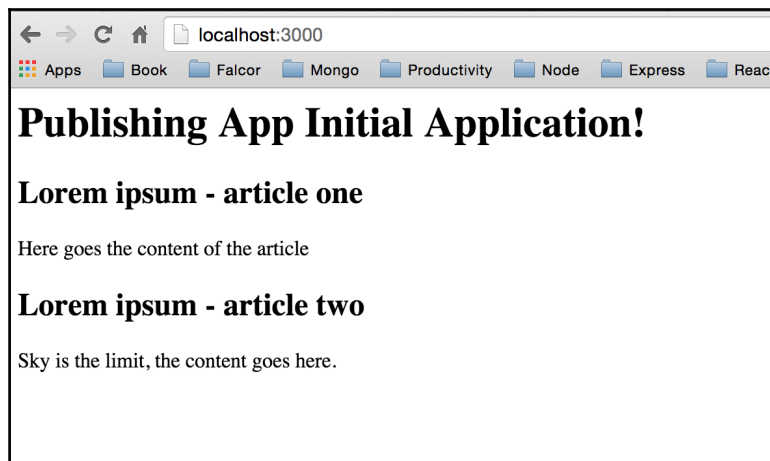
```
"dependencies": {  
  "body-parser": "1.15.0",  
  "cors": "2.7.1",  
  "express": "4.13.4",  
  "mongoose": "4.4.5"  
}
```

In the main directory, run Node with the following:

```
node server/index.js
```

After that, your terminal should show something like the following:

```
$ node server/index.js  
Started on port 3000
```



Redux basic concepts

In this section, we will cover only the most basic concepts of Redux that will help us make our simple publishing app. The app will only be in *read-only* mode for this chapter; later in the book we will add more functionality such as adding/editing an article. You will discover all the important rules and principles about Redux in the later chapters.

Basic topics covered are:

- What is a state tree?
- How immutability works in Redux
- The concept and basic use of reducers

Let's start with the basics.

The single immutable state tree

The most important principle of Redux is that you are going to represent the whole state of your application as a single JavaScript object.

All changes (actions) in Redux are explicit, so you can track a history of all your actions through the application with a dev tool.



The preceding screenshot is a simple, example dev tool use case that you will use in your development environment. It will help you to track the changes of state in your app. The example shows how we have incremented the counter value in our state by +1, three times. Of course, our publishing app structure will be much more complicated than this example. You will learn more about that dev tool later in the book.

Immutability - actions and state tree are read-only

As Redux bases its concepts on functional programming paradigms, you cannot modify/mutate the values in your state tree in the same way that you can for Facebook's (and other) FLUX implementations.

As with other FLUX implementations, an action is a plain object that describes the change--like adding an article (in the following code we mock the payload for the sake of brevity):

```
{
  type: 'ADD_ARTICLE',
  payload: '_____HERE_GOES_INFORMATION_ABOUT_THE_CHANGE_____'
}
```

An action is a minimal representation of the change for our app state tree. Let's prepare actions for our publishing app.

Pure and impure functions

A **pure function** is a function that doesn't have any side effects, such as for example, I/O (reading a file or an HTTP request). **Impure functions** have side effects so, for example, if you make a call to the HTTP request, it can return different values for exactly the same arguments Y, Z ($function(X, Y)$) because an endpoint is returning us a random value, or could be down because of a server error.

Pure functions are always predictable for the same X, Y arguments. In Redux, we use only pure functions in reducers and actions (otherwise Redux's `lib` won't work properly).

In this book, you will learn the whole structure and where to make API calls. So if you follow the book, then you won't have to worry too much about that principle in Redux.

The reducer function

Reducers from Redux can be compared to a single store from Facebook's Flux. What is important is that a reducer always takes a previous state and returns a new reference to a new object (with the use of `Object.assign` and others like that), so we can have immutable JS helping us to build a more predictable state of our application in comparison to older Flux implementations that mutate variables in the store.

Thus, creating a new reference is optimal because Redux uses old references to values from reducers that didn't change. This means that even if each action creates a whole new object via a reducer then the values that don't change have a previous reference in the memory so we don't overuse the computation power of the machine. Everything is fast.

In our app, we will have an article reducer that will help us to list, add, edit, and delete our articles from the view layer.

First reducer and webpack config

First, let's create a reducer for our publication app:

```
mkdir src
cd src
mkdir reducers
cd reducers
touch article.js
```

So, our first reducer's location is `src/reducers/article.js` and the content of our `reducers/article.js` is as follows:

```
const articleMock = {
  '987654': {
    articleTitle: 'Lorem ipsum - article one',
    articleContent: 'Here goes the content of the article'
  },
  '123456': {
    articleTitle: 'Lorem ipsum - article two',
    articleContent: 'Sky is the limit, the content goes here.'
  }
};

const article = (state = articleMock, action) => {
  switch (action.type) {
    case 'RETURN_ALL_ARTICLES':
      return Object.assign({}, state);
    default:
      return state;
  }
}

export default article;
```

In the preceding code, we have our `articleMock` kept in the browser memory (it's the same as in `initData.js`)--later, we will fetch this data from our backend database.

The arrow function, `const article` is getting `action.type` which will come from constants (we will create them later) in the same way that Facebook's FLUX implementation works.

For the default return in the `switch` statement, we provide the state from `state = articleMock` (`return state;` part above). This will return the initial state of our publishing app at first startup before any other action occurs. To be exact, the default in our case will do exactly the same as the `RETURN_ALL_ARTICLES` action before we start fetching data from the backend (after the articles' fetching mechanism from backend is implemented; then the default will return an empty object).

Because of our webpack configuration (described here), we need `index.html` in `dist`. Let's create a `dist/index.html` file:

```
pwd
/Users/przeor/Desktop/React-Convention-Book/src/reducers
cd ../../
mkdir dist
cd dist
touch index.html
```

The `dist/index.html` file's content is as follows:

```
<!doctype html>
<html lang="en">
<head>
<title>Publishing App</title>
<meta charset="utf-8">

</head>
<body>
<div id="publishingAppRoot"></div>

<script src="app.js"></script>
</body>
</html>
```

We have an article reducer and `dist/index.html`, but before we start building our Redux's publishing app, we need to configure webpack for our built automation.

Install webpack first (you may need `sudo` root access for it):

```
npm i --save-dev webpack@1.12.14 webpack-dev-server@1.14.1
```

Then, in the main directory next to the `package.json` and `initData.js` files, input the following:

```
touch webpack.config.js
```

Then create webpack configs:

```
module.exports = {
  entry: ['babel-polyfill', './src/app.js'],
  output: {
    path: './dist',
    filename: 'app.js',
    publicPath: '/'
  },
  devServer: {
    inline: true,
    port: 3000,
    contentBase: './dist'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /(node_modules|bower_components)/,
        loader: 'babel',
        query: {
          presets: ['es2015', 'stage-0', 'react']
        }
      }
    ]
  }
}
```

Simply, webpack config says that the entry of the CommonJS module is at entry `./src/app.js`. webpack builds a whole app following all imports from the `app.js` and the final output is located at path `./dist`. Our app that is located at `contentBase: './dist'` will live at port 3000. We also configure the use of ES2015 and React so that webpack will compile ES2015 into ES5 and React's JSX into JavaScript for us. If you are interested in webpack's configuration options, then read its documentation.

The rest of the important dependencies installation and npm dev script

Install the Babel tools that are used by webpack (check the config file):

```
npm i --save react@0.14.7 react-dom@0.14.7 react-redux@4.4.0 redux@3.3.1
```

We also need to update our `package.json` file (add scripts):

```
"scripts": {  
  "dev": "webpack-dev-server"  
},
```

Our complete `package.json` should look like the following, with all frontend dependencies:

```
{  
  "name": "project",  
  "version": "1.0.0",  
  "description": "",  
  "scripts": {  
    "dev": "webpack-dev-server"  
  },  
  "dependencies": {  
    "body-parser": "1.15.0",  
    "cors": "2.7.1",  
    "express": "4.13.4",  
    "mongoose": "4.4.5",  
    "react": "0.14.7",  
    "react-dom": "0.14.7",  
    "react-redux": "4.4.0",  
    "redux": "3.3.1"  
  },  
  "devDependencies": {  
    "babel": "6.5.2",  
    "babel-core": "6.6.5",  
    "babel-loader": "6.2.4",  
    "babel-polyfill": "6.6.1",  
    "babel-preset-es2015": "6.6.0",  
    "babel-preset-react": "6.5.0",  
    "babel-preset-stage-0": "6.5.0",  
    "webpack": "1.12.14",  
    "webpack-dev-server": "1.14.1"  
  }  
}
```



As you may realize, the mentioned `package.json` doesn't have the `^` signs as we want to use the exact versions of each package in order to make sure that all our packages are installed with the correct and exact version given in the package. Otherwise, you may have some difficulties, for example, if you add `"mongoose": "4.4.5"`, with the `^` then it will install a newer version that causes some additional warnings in the console. Let's stick to the versions mentioned in the book in order to avoid unnecessary problems with the app that we are building. We want to avoid NPM dependencies hell at all cost.

Working on `src/app.js` and `src/layouts/PublishingApp.js`

Let's create our `app.js` file, where the main part of our app will live at `src/app.js`:

```
//[[your are in the main directory of the project]]
cd src
touch app.js
```

The content of our new `src/app.js` file is the following:

```
import React from 'react';
import { render } from 'react-dom';
import { Provider } from 'react-redux';
import { createStore } from 'redux';
import article from '../reducers/article';
import PublishingApp from '../layouts/PublishingApp';

const store = createStore(article);

render(
  <Provider store={store}>
    <PublishingApp />
  </Provider>,
  document.getElementById('publishingAppRoot')
);
```

The new part is the `store = createStore(article)` part--this utility from Redux lets you keep an application state object, dispatch an action, and allows you to give a reducer as an argument that tells you how the app is updated with actions.

The `react-redux` is a useful binding of Redux into React (so we will write less code and be more productive):

```
<Provider store>
```

The `Provider store` helps us to make the Redux store available to the `connect()` calls in the child components (as shown here):

```
connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])
```

`connect` will be used in any component that has to listen to the reducer's changes in our app. You will see how to use it later in this chapter.

For the store, we use `const store = createStore(article)`--just for the sake of brevity, I will mention that there are several methods in the store that we will use in the next steps of building our app from scratch:

```
store.getState();
```

The `getState` function gives you the current state of the application:

```
store.dispatch({ type: 'RETURN_ALL_ARTICLES' });
```

The `dispatch` function can help you change the state of your app:

```
store.subscribe(() => {  
  
});
```

`Subscribe` allows you register a callback that Redux will call each time an action has been dispatched, so the view layer can learn about the change in the application state and refresh its view.

Wrapping up React-Redux application

Let's finish our first React-Redux app. For a summary, let's see our current directory structure:

```
&boxvr;&boxh;&boxh; dist  
&boxv;  &boxur;&boxh;&boxh; index.html  
&boxvr;&boxh;&boxh; initData.js  
&boxvr;&boxh;&boxh; node_modules  
&boxv;  &boxvr;&boxh;&boxh; ***** (A LOT OF LIBRARIES HERE)  
&boxvr;&boxh;&boxh; package.json  
&boxvr;&boxh;&boxh; server
```

```
&boxv;    &boxvr;&boxh;&boxh; index.js
&boxv;    &boxur;&boxh;&boxh; server.js
&boxvr;&boxh;&boxh; src
&boxv;    &boxvr;&boxh;&boxh; app.js
&boxv;    &boxur;&boxh;&boxh; reducers
&boxv;        &boxur;&boxh;&boxh; article.js
&boxur;&boxh;&boxh; webpack.config.js
```

Now we need to create the main view of our app. We will put this into the layout directory in our first version:

```
pwd
/Users/przeor/Desktop/React-Convention-Book/src
mkdir layouts
cd layouts
touch PublishingApp.js
```

The content of `PublishingApp.js` is:

```
import React from 'react';
import { connect } from 'react-redux';

const mapStateToProps = (state) => ({
  ...state
});

const mapDispatchToProps = (dispatch) => ({
});

class PublishingApp extends React.Component {
  constructor(props) {
    super(props);
  }
  render () {
    console.log(this.props);
    return (
<div>
      Our publishing app
</div>
    );
  }
}
export default connect(mapStateToProps, mapDispatchToProps)(PublishingApp);
```

The preceding introduces the ES7 syntax `...` next to `...`:

```
const mapStateToProps = (state) => ({
  ...state
});
```

`...` is a spread operator that is well described in Mozilla's documentation as; *an expression to be expanded in places where multiple arguments (for function calls) or multiple elements (for array literals) are expected*. In our case, this `...` operator spreads one object state into a second one (in our case, empty object `{ }`). It's written like this here because, in future, we will specify multiple reducers that have to be mapped from our app's state into the `this.props` component.

Finishing our first static publishing app

The last thing to do in our static app is to render the articles that come from `this.props`.

Thanks to Redux, the object mocked in the reducer is available, so if you check `console.log(this.props)` in the render function of `PublishingApp.js`, then you will be able to access our `articles` object:

```
const articleMock = {
  '987654': {
    articleTitle: 'Lorem ipsum - article one',
    articleContent: 'Here goes the content of the article'
  },
  '123456': {
    articleTitle: 'Lorem ipsum - article two',
    articleContent: 'Sky is the limit, the content goes here.'
  }
};
```

In our case, we need to change the React's render function, as follows (in `src/layouts/PublishingApp.js`):

```
render () {
  let articlesJSX = [];

  for(let articleKey in this.props) {
    const articleDetails = this.props[articleKey];
    const currentArticleJSX = (
      <div key={articleKey}>
        <h2>{articleDetails.articleTitle}</h2>
        <h3>{articleDetails.articleContent}</h3>
      </div>);
```



```
        articlesJSX.push(currentArticleJSX);
    }

    return (
        <div>
            <h1>Our publishing app</h1>
            {articlesJSX}
        </div>
    );
}
```

In the preceding code snippet, we are iterating `for(let articleKey in this.props)` over the article Mock object (passed from the reducer's state in `this.props`) and creating an array of articles (in JSX) with `articlesJSX.push(currentArticleJSX);`. After it is created, then we will have added the `articlesJSX` into the return statement:

```
<div>
<h1>Our publishing app</h1>
    {articlesJSX}
</div>
```

This comment will start your project on port 3000:

```
npm run dev
```

After you check `localhost:3000`, the new static Redux app should look as shown in the following screenshot:



Great, so we have a static app in Redux! It's time to fetch data from our MongoDB database using Falcor.

Falcor's basic concepts

Falcor is like a glue between:

- Backend and its database structure (remember importing `initData.js` into MongoDB)
- Frontend Redux single state tree container

It glues the pieces in a way that is much more effective than building an old-fashioned REST API for a single-page application.

Like the *Redux basic concepts* section, in this one we will learn only the most basic concepts of Falcor and they will help us build a simple full-stack application in *read-only* mode. Later in the book, you will learn how to make an add/edit article with Falcor.

We will focus on the most important aspects:

- What is Falcor's model?
- Retrieving values from Falcor (frontend and backend)
- Concepts and basic use of JSON graphs
- Concepts and basic use of sentinels
- How to retrieve data from the backend
- How to configure our first route with middleware for Express.js called `falcor-router`

What is Falcor and why do we need it in our full-stack publishing app?

Let's first consider what the difference is between web pages and web applications:

- When the **World Wide Web (WWW)** was invented, web pages served small amounts of large resources (such as HTML, PDF, and PNG files). For example, you could request a PDF, video, or text file from a server.
- Since *circa 2008*, the development of web apps has been getting more and more popular. Web applications serve large amounts of small resources. What does it mean for us? You have a lot of small REST API calls to the server using AJAX calls. The old approach of many API requests creates latency, which slows down the mobile/web app.

Why do we use old REST API requests (as we did in 2005) in apps written in 2016 and later? This is where Falcor shines; it solves the problem of latency and tight coupling of backend to frontend.

Tight coupling and latency versus one model everywhere

If you are familiar with frontend development, you know how to make requests to an API. This old way of doing things always forces you to tight-couple the backend API with frontend API utilities. It's always like that:

1. You make an API endpoint like
`https://applicationDomain.com/api/recordDetails?id=92.`
2. You consume the data with HTTP API requests on the frontend:

```
{
  id: '92',
  title: 'example title',
  content: 'example content'
}
```

In large applications, it's hard to maintain real DRY RESTful APIs, and this problem causes plenty of endpoints that are not optimized, so the frontend sometimes has to do many round trips in order to fetch the data required for a certain view (and sometimes it fetches much more than it needs, which causes even more latency for the end user of our application).

Imagine that you have a large application with over 50 different API endpoints. After the first version of your application is finished, your client or boss finds a better way to structure the user flow in the app. What does this mean? That you have to work on changing both frontend and backend endpoints in order to satisfy the changes in the user interface layer. This is called tight coupling between frontend and backend.

What does Falcor bring to this situation to improve on those two areas that cause the inefficiency in working with RESTful APIs? The answer is one model everywhere.

It would be super easy to build your web applications if all your data was accessible in memory, on the client.

Falcor provides utilities that help you feel that all your data is at your fingertips without coding backend API endpoints and client-side consuming utilities.

No more tight coupling on client and server side

Falcor helps you represent all of your app's data as one virtual JSON model on the server.

When programming client side, Falcor makes you feel as if the whole JSON model of your application is reachable locally, and allows you to read data the same way as you would from an in-memory JSON--you will learn it very soon!

Because of Falcor's library for browsers and the `falcor-express` middleware, you can retrieve your data from the model on-demand, from the cloud.

Falcor transparently handles all the network communication and keeps your client-side app in sync with the server and databases.

In this chapter, we will also learn how to use `falcor-router`.

Client-side Falcor

Let's install Falcor from NPM first:

```
pwd
/Users/przeor/Desktop/React-Convention-Book
npm i --save falcor@0.1.
16 falcor-http-datasource@0.1.3
```

The `falcor-http-datasource` helps us to retrieve data from server to client side, out-of-the-box (without worrying about HTTP API requests)--we will use this later when moving the client-side model to the backend.

Let's create our app's Falcor model on the client side:

```
cd src
touch falcorModel.js
```

Then the content of the `falcorModel.js` will be as follows:

```
import falcor from 'falcor';
import FalcorDataSource from 'falcor-http-datasource';

let cache = {
  articles: [
    {
      id: 987654,
      articleTitle: 'Lorem ipsum - article one',
      articleContent: 'Here goes the content of the article'
```

```
    },
    {
      id: 123456,
      articleTitle: 'Lorem ipsum - article two from backend',
      articleContent: 'Sky is the limit, the content goes here.'
    }
  ]
};

const model = new falcor.Model({
  'cache': cache
});
export default model;
```

In this code, you can find a well-known, brief, and readable model of our publishing application with two articles in it.

Now we will fetch that data from the frontend Falcor's model in our `src/layouts/PublishingApp.js` React component, we will add a new function called `_fetch()` which will be responsible for fetching all articles on our application start.

We need to import our Falcor model first, so at the top of the `PublishingApp.js` file, we need to add the following:

```
import falcorModel from '../falcorModel.js';
```

In our `PublishingApp` class, we need to add the following two functions; `componentWillMount` and `_fetch` (more explanation follows):

```
class PublishingApp extends React.Component {
  constructor(props) {
    super(props);
  }

  componentWillMount() {
    this._fetch();
  }

  async _fetch() {
    const articlesLength = await falcorModel.
      getValue('articles.length').
      then((length) => length );

    const articles = await falcorModel.
      get(['articles', {from: 0, to: articlesLength-1},
        ['id','articleTitle', 'articleContent']])
      .then((articlesResponse) => articlesResponse.json.articles);
```

```
}  
// below here are next methods o the PublishingApp
```

Here, you see the asynchronous function called `_fetch`. This is a special syntax that allows you to use the `await` keyword like we do when using `let articlesLength = await falcorModel` and `let articles = await falcorModel`.

Using `async await` over Promises means our code is more readable and avoids callback hell situations where nesting multiple callbacks one after the other makes code very hard to read and extend.

The `async/await` feature is taken from ECMAScript 7 inspired by C#. It allows you to write functions that appear to be blocked at each asynchronous operation that is waiting for the result before continuing to the next operation.

In our example, the code will execute as follows:

1. First it will call Falcor's mode for an article count with the following:

```
const articlesLength = await falcorModel.  
  getValue('articles.length').  
  then( (length) => length );
```

2. In the article's `Length` variable, we will have a count of `articles.length` from our model (in our case it will be number two).
3. After we know that we have two articles in our model, then the next block of code executes the following:

```
let articles = await falcorModel.  
  get(['articles', {from: 0, to: articlesLength-1},  
    ['id','articleTitle', 'articleContent']]).  
  then( (articlesResponse) => articlesResponse.json.articles);
```

The `get` method on `falcorModel.get(['articles', {from: 0, to: articlesLength-1}, ['id','articleTitle', 'articleContent']])` is also an asynchronous operation (in the same way as `http request`). In the `get` method's parameter, we provide the location of our articles in our model (in `src/falcorModel.js`), so we are providing the following path:

```
falcorModel.get(  
  ['articles', {from: 0, to: articlesLength-1}, ['id','articleTitle',  
    'articleContent']]  
)
```

The explanation of the preceding Falcor path is based on our model. Let's call it again:

```
{
  articles: [
    {
      id: 987654,
      articleTitle: 'Lorem ipsum - article one',
      articleContent: 'Here goes the content of the article'
    },
    {
      id: 123456,
      articleTitle: 'Lorem ipsum - article two from backend',
      articleContent: 'Sky is the limit, the content goes here.'
    }
  ]
}
```

What we are saying to Falcor:

1. First we want to get data from `articles` within our object using:

```
['articles']
```

2. Next from `articles` collection select subset of all the articles it has with a range `{from: 0, to: articlesLength-1}` (the `articlesLength` we have fetched earlier) with the following path:

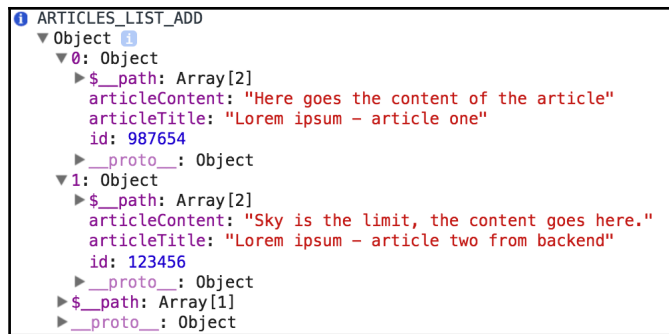
```
['articles', {from: 0, to: articlesLength-1}]
```

3. The last step explains to Falcor, which properties from the object you want to fetch from our model. So the complete path in that `falcorModel.get` query is the following:

```
['articles', {from: 0, to: articlesLength-1},
['id', 'articleTitle', 'articleContent']]
```

4. The array of `['id', 'articleTitle', 'articleContent']` says that you want those three properties out of every article.

5. In the end, we receive an array of article objects from Falcor:



After we have fetched the data from our Falcor model, we need to dispatch an action that will change the article's reducer accordingly and ultimately re-render our list of articles from our Falcor model from the `const articleMock` (in `src/reducers/article.js`) instead.

But before we will be able to dispatch an action, we need to do the following:

Create the actions directory with `article.js`:

```
pwd
$ /Users/przeor/Desktop/React-Convention-Book
cd src
mkdir actions
cd actions
touch article.js
```

Create the content for our `src/actions/article.js` file as follows:

```
export default {
  articlesList: (response) => {
    return {
      type: 'ARTICLES_LIST_ADD',
      payload: { response: response }
    }
  }
}
```

There isn't too much in that `actions/article.js` file . If you are familiar with FLUX already then it's very similar. One important rule for actions in Redux is that it has to be a pure function. For now, we will hardcode a constant called `ARTICLES_LIST_ADD` into `actions/article.js`.

In the `src/layouts/PublishingApp.js` file we need to add a new import code at the top of the file:

```
import {bindActionCreators} from 'redux';
import articleActions from '../actions/article.js';
```

When you have added the preceding two in our `PublishingApp`, then modify our existing function in the same file from the following:

```
const mapDispatchToProps = (dispatch) => ({
  });
```

Add `articleActions: bindActionCreators(articleActions, dispatch)` so that we are able to bind the articles' actions into our `this.props` component:

```
const mapDispatchToProps = (dispatch) => ({
  articleActions: bindActionCreators(articleActions, dispatch)
});
```

Thanks to the mentioned changes (`articleActions: bindActionCreators(articleActions, dispatch)`) in our component, we will be able to dispatch an action from props because now, when you use `this.props.articleActions.articlesList(articles)` then the `articles` object fetched from Falcor will be available in our reducer (and from there, there is only one step to make our app fetch data work).

Now, after you are done with these changes, add an action into our component in the `_fetch` function:

```
this.props.articleActions.articlesList(articles);
```

Our whole function for fetching will look as follows:

```
async _fetch() {
  const articlesLength = await falcorModel.
    getValue('articles.length').
    then( (length) => length);

  let articles = await falcorModel.
    get(['articles', {from: 0, to: articlesLength-1},
      ['id', 'articleTitle', 'articleContent']]).
    then( (articlesResponse) => articlesResponse.json.articles);

  this.props.articleActions.articlesList(articles);
}
```

Also, don't forget about calling `_fetch` from `ComponentWillMount`:

```
componentWillMount() {  
  this._fetch();  
}
```

At this point, we shall be able to receive an action in our Redux's reducer. Let's improve our `src/reducers/article.js` file:

```
const article = (state = {}, action) => {  
  switch (action.type) {  
    case 'RETURN_ALL_ARTICLES':  
      return Object.assign({}, state);  
    case 'ARTICLES_LIST_ADD':  
      return Object.assign({}, action.payload.response);  
    default:  
      return state;  
  }  
}  
export default article
```

As you can see, we don't need `articleMock` anymore, so we have deleted it from the `src/reducers/article.js`.

We have added a new case, `ARTICLES_LIST_ADD`:

```
case 'ARTICLES_LIST_ADD':  
  let articlesList = action.payload.response;  
  return Object.assign({}, articlesList);
```

It returns a new `articlesList` object (with a new reference in the memory, thanks to `Object.assign`).



Don't confuse the two files with the same name and other locations, such as:

`reducers/article.js`

`actions/article.js`

You need to make sure that you are editing the correct file, otherwise the app won't work.

A summary of client-side Falcor + Redux

If you run `http://localhost:3000/index.html`, you will see that, currently we have two separate applications:

- One at the frontend using Redux and client-side Falcor
- One at the backend using MongoDB, Mongoose, and Express

We need to stick both together so we have one source of state for our applications (that comes from MongoDB).

Moving Falcor's model to the backend

We also need to update our `package.json` file:

```
"scripts": {  
  "dev": "webpack-dev-server",  
  "start": "npm run webpack; node server",  
  "webpack": "webpack --config ./webpack.config.js"  
},
```

Because we are starting the full-stack development part, we need to add `npm start` to our scripts in `package.json`--this will help compile client side, put them into the `dist` folder (generated via webpack), and create static files in `dist`, and then use this folder as the source of static files (check `server/server.js` for `app.use(express.static('dist'))`).

The next important thing is to install new dependencies that are required for Falcor on the backend:

```
npm i --save falcor-express@0.1.2 falcor-router@0.2.12
```

When you have finally installed new dependencies and configured the basic scripts for running the backend and frontend on the same port, then edit the `server/server.js` as follows:

1. On top of our file, import new libraries in the `server/server.js`:

```
import falcor from 'falcor';  
import falcorExpress from 'falcor-express';
```

2. Then between the following two:

```
app.use(bodyParser.json({extended: false}));
app.use(express.static('dist'));
```

3. Add new code for managing Falcor at the backend:

```
app.use(bodyParser.json({extended: false}));

let cache = {
  articles: [
    {
      id: 987654,
      articleTitle: 'Lorem ipsum - article one',
      articleContent: 'Here goes the content of the article'
    },
    {
      id: 123456,
      articleTitle: 'Lorem ipsum - article two from
backend',
      articleContent: 'Sky is the limit, the content goes
here.'
    }
  ]
};

var model = new falcor.Model({
  cache: cache
});

app.use('/model.json', falcorExpress.dataSourceRoute((req,
res) => {
  return model.asDataSource();
}));
app.use(express.static('dist'));
```

4. The preceding code is almost the same as the one in the `src/falcorModel.js` file. The only difference is that now Falcor will fetch data from the backend's mocked object, called `cache` in `server.js`.

5. The second part is to change our data source on the frontend, so in the `src/falcorModel.js` file, you change the following old code:

```
import falcor from 'falcor';
import FalcorDataSource from 'falcor-http-datasource';

let cache = {
  articles: [
    {
      id: 987654,
      articleTitle: 'Lorem ipsum - article one',
      articleContent: 'Here goes the content of the article'
    },
    {
      id: 123456,
      articleTitle: 'Lorem ipsum - article two from backend',
      articleContent: 'Sky is the limit, the content goes here.'
    }
  ]
};

const model = new falcor.Model({
  'cache': cache
});

export default model;
```

6. Change it to the following updated code:

```
import falcor from 'falcor';
import FalcorDataSource from 'falcor-http-datasource';

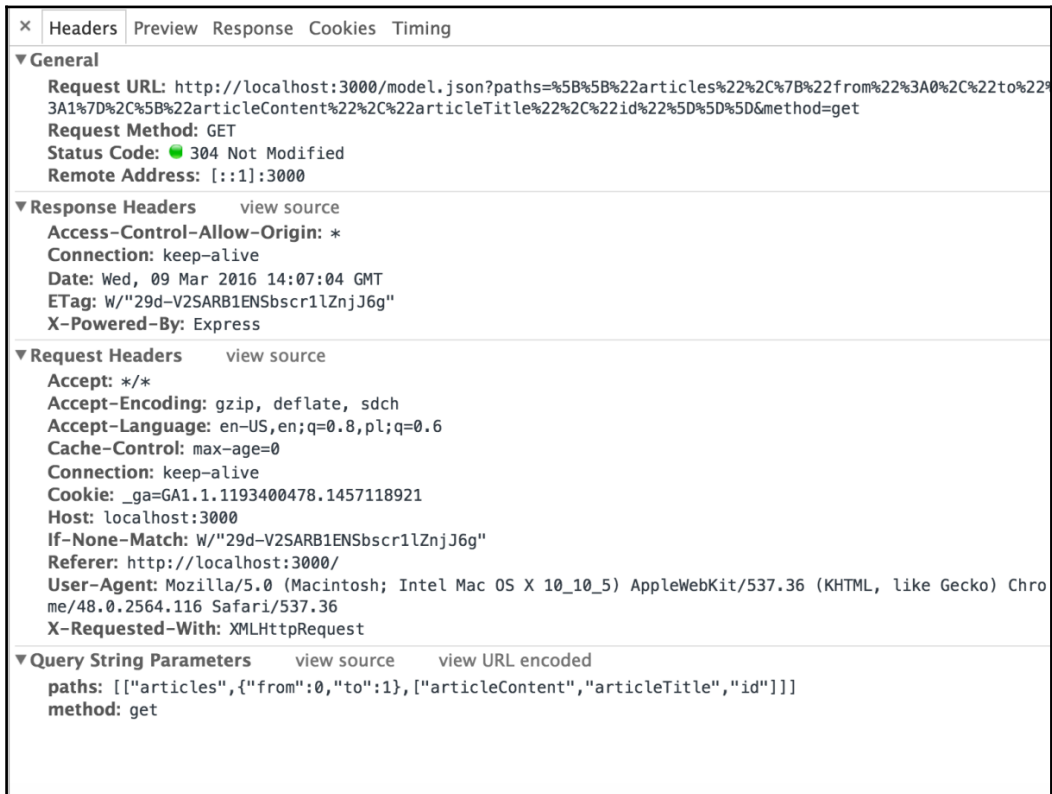
const model = new falcor.Model({
  source: new FalcorDataSource('/model.json')
});

export default model;
```

7. Run your app with the following:

npm start

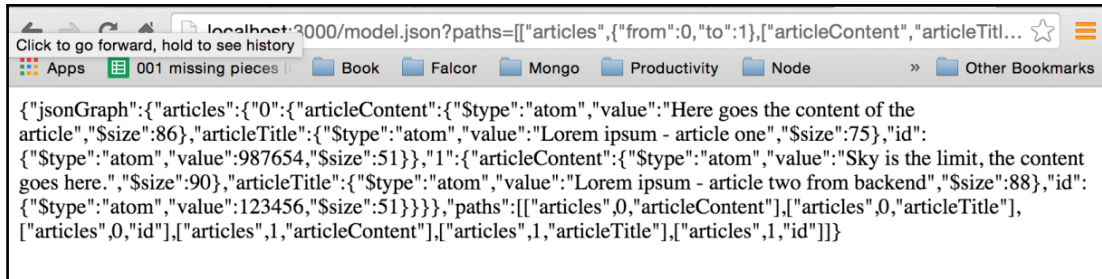
8. You will see in your browser's dev tools a new HTTP request made by Falcor--for example, in our case:



If you follow all the instructions correctly, then you can also make a request to your server directly from your browser by executing this:

```
http://localhost:3000/model.json?paths=[["articles", {"from": 0, "to": 1}, ["articleContent", "articleTitle", "id"]]]&method=get.
```

Then you shall see a `jsonGraph` in the response:



You don't have to worry about those preceding two screenshots. They are just an example of how Falcor is communicating between the backend and frontend in Falcor's language. You don't have to worry anymore about exposing API endpoints and programming frontend to understand what data the backend is providing. Falcor is doing all of this out-of-the-box and you will learn more details while making this publishing application.

Configuring Falcor's router (Express.js)

Currently, our model on the backend is hardcoded, so that it stays in the RAM memory of a server. We need to add the ability to read the data from our MongoDB's articles collection--this is where the `falcor-router` comes in handy.

We need to create our routes definition files that will be consumed by `falcor-router` lib:

```
$ pwd
/Users/przeor/Desktop/React-Convention-Book
$ cd server
$ touch routes.js
```

We have created the `server/routes.js` file; the content for that router will be as follows:

```
const PublishingAppRoutes = [{
  route: 'articles.length',
  get: () => {
    const articlesCountInDB = 2; // hardcoded for example
    return {
      path: ['articles', 'length'],
      value: articlesCountInDB
    };
  }
}];
export default PublishingAppRoutes;
```

As you can see, we have created our first route that will match the `articles.length` from our `_fetch` function (in `layouts/PublishingApp.js`).

We have hardcoded the number two in `articlesCountInDB`, later we will make a query to our database there.

The new stuff here is `route: 'articles.length'`, this is simply a route for matching by Falcor.

To be more precise, the Falcor routes' paths are exactly the same stuff that you have provided in your `src/layouts/PublishingApp.js` (`_fetch` function) for example, to match this frontend call:

```
// location of that code snippet: src/layouts/PublishingApp.js
const articlesLength = await falcorModel.
  getValue('articles.length').
  then((length) => length);
```

- `path: ['articles', 'length']`: This property tells us Falcor's path (it's consumed by Falcor at the backend and frontend). We need to provide that because sometimes, one route can return many different objects as server articles (you will see it in the next route we create).
- `value: articlesCountInDB`: This is a return value. In this case, it is an integer number, but it can also be an object with several properties, as you will learn later.

Second route for returning our two articles from the backend

Our second route (and last one in this chapter) will be the following:

```
{
  route: 'articles[{integers}]["id","articleTitle","articleContent"]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];
    const articlesArrayFromDB = [{
      'articleId': '987654',
      'articleTitle': 'BACKEND Lorem ipsum - article one',
      'articleContent': 'BACKEND Here goes the content of the article'
    }, {
      'articleId': '123456',
      'articleTitle': 'BACKEND Lorem ipsum - article two',
      'articleContent': 'BACKEND Sky is the limit, the content goes here.'
    }
  ]
}
```



```
    }); // That are our mocked articles from MongoDB

    let results = [];
    articlesIndex.forEach((index) => {
      const singleArticleObject = articlesArrayFromDB[index];
      const falcorSingleArticleResult = {
        path: ['articles', index],
        value: singleArticleObject
      };
      results.push(falcorSingleArticleResult);
    });

    return results;
  }
}
```

The new thing in the second route is `pathSet`, if you log that into the console, then you will see, in our case (when trying to run our full-stack app) the following:

```
[
  'articles',
  [ 0, 1 ],
  [ 'articleContent', 'articleTitle', 'id' ]
]
```

`pathSet` tells us what indexes are requested from the client side (`[0, 1]`, in our example).

Because, in this case, we are returning an array of articles (multiple articles), we need to create a result variable:

```
let results = [];
```

Iterate over the requested indexes:

```
articlesIndex.forEach((index) => {
  const singleArticleObject = articlesArrayFromDB[index];
  const falcorSingleArticleResult = {
    path: ['articles', index],
    value: singleArticleObject
  };
  results.push(falcorSingleArticleResult);
});
```

In the preceding code snippet, we iterate over an array of requested indexes (do you remember `{from: 0, to: articlesLength-1}` in `PublishingApp.js`?). Based on the indexes (`[0, 1]`) we fetch mocked data via `const singleArticleObject = articlesArrayFromDB[index];`. Later we put into the path and index (`path: ['articles', index],`) so Falcor knows to what path in our JSON graph object, the value `singleArticleObject` belongs to.

Return that array of articles:

```
console.info(results)
return results;
```

`console.info` will show us what has been returned by that path:

```
[{
  path: ['articles', 0],
  value: {
    articleId: '987654',
    articleTitle: 'BACKEND Lorem ipsum - article one',
    articleContent: 'BACKEND Here goes the content of the article'
  }
}, {
  path: ['articles', 1],
  value: {
    articleId: '123456',
    articleTitle: 'BACKEND Lorem ipsum - article two',
    articleContent: 'BACKEND Sky is the limit, the content goes here.'
  }
}]
```

Final touch to make full-stack Falcor run

Currently, we still have mocked data in our routes, but before we start making calls to MongoDB, we need to wrap up the current setup so you will be able to see it running in your browser.

Open your `server/server.js` and make sure you import the following two things:

```
import falcorRouter from 'falcor-router';
import routes from './routes.js';
```

Now that we have imported our `falcor-router` and `routes.js`--we need to use them, so modify this old code:

```
// This is old code, remove it and replace with new
app.use('/model.json', falcorExpress.dataSourceRoute((req, res) => {
  return model.asDataSource();
}));
```

Replace the preceding code with:

```
app.use('/model.json', falcorExpress.dataSourceRoute((req, res) => {
  return new falcorRouter(routes);
}));
```

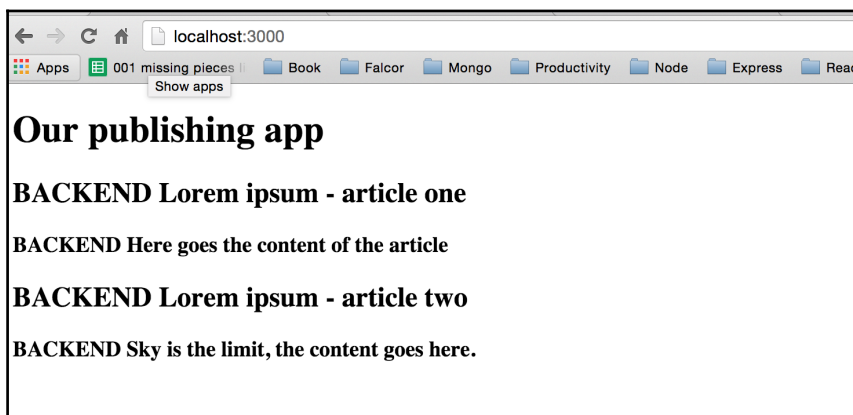
This will work only when the `falcor-router` has been already installed and imported in the `server.js` file. This is a library for `DataSource` that creates a virtual JSON graph document on your app server. As you can see in `server.js` so far we have `DataSource` provided by our hardcoded model, `return model.asDataSource();`. The router here will make the same, but now you will be able to match routes based on your app requirements.

Also, as you can see, the new `falcorRouter` takes an argument of our routes `return new falcorRouter(routes);`.

If you have followed the instructions correctly, you will be able to run the project:

npm start

On port 3000, you will see the following:



Adding MongoDB/Mongoose calls based on Falcor's routes

Let's get back to our `server/routes.js` file. We need to move over (delete from `server.js` and move into `routes.js`) this following code:

```
// this goes to server/routes.js
import mongoose from 'mongoose';

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

const articleSchema = {
  articleTitle:String,
  articleContent:String
};
const Article = mongoose.model('Article', articleSchema, 'articles');
```

In the first route `articles.length`, you need to replace the mocked number two (the articles count) into Mongoose's `count` method:

```
route: 'articles.length',
get: () => {
  return Article.count({}, (err, count) => count)
    .then ((articlesCountInDB) => {
      return {
        path: ['articles', 'length'],
        value: articlesCountInDB
      }
    })
}
```



We are returning a Promise in `get` (Mongoose, by its asynchronous nature, always returns a Promise while making any database's request, as in the example, `Article.count`).

The method `Article.count` simply retrieves the integer number of articles' count from our *Article* model (that was prepared at the beginning of this book in MongoDB/Mongoose sub-chapter).

The second route route:

'articles[{integers}]["id","articleTitle","articleContent"]', has to be changed as follows:

```
{
  route: 'articles[{integers}]["id","articleTitle","articleContent"]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];

    return Article.find({}, (err, articlesDocs) => articlesDocs)
      .then ((articlesArrayFromDB) => {
        let results = [];
        articlesIndex.forEach((index) => {
          const singleArticleObject =
            articlesArrayFromDB[index].toObject();
          const falcorSingleArticleResult = {
            path: ['articles', index],
            value: singleArticleObject
          };
          results.push(falcorSingleArticleResult);
        });
        return results;
      })
  }
}
```

We return a Promise again with `Article.find`. Also, we have deleted the mocked response from the database and instead we are using the `Article.find` method.

The array of articles is returned in `}).then ((articlesArrayFromDB) => {` and next we simply iterate and create a results array.

Note that on `const singleArticleObject = articlesArrayFromDB[index].toObject();` we use a method `.toObject`. This is very important for making this work.

Double-check with the server/routes.js and package.json

In order to save you time in case the app doesn't run, we can double-check that the backend's Falcor routes are prepared correctly:

```
import mongoose from 'mongoose';

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

const articleSchema = {
  articleTitle:String,
  articleContent:String
};

const Article = mongoose.model('Article', articleSchema, 'articles');

const PublishingAppRoutes = [
  {
    route: 'articles.length',
    get: () => Article.count({}, (err, count) => count)
      .then ((articlesCountInDB) => {
        return {
          path: ['articles', 'length'],
          value: articlesCountInDB
        };
      })
  },
  {
    route: 'articles[{integers}]
[id,"articleTitle","articleContent"]',
    get: (pathSet) => {
      const articlesIndex = pathSet[1];

      return Article.find({}, (err, articlesDocs) =>
articlesDocs);
      .then ((articlesArrayFromDB) => {
        let results = [];

        articlesIndex.forEach((index) => {
          const singleArticleObject =
articlesArrayFromDB[index].toObject();
          const falcorSingleArticleResult = {
            path: ['articles', index],
            value: singleArticleObject
          };
        });
      });
    }
  }
];
```

```
        results.push(falcorSingleArticleResult);
    });

    return results;
  })
}
}
];

export default PublishingAppRoutes;
```

Check that your `server/routes.js` file looks similar to the preceding code and the other code elements that you have used.

Also, check that your `package.json` look likes the following one:

```
{
  "name": "project",
  "version": "1.0.0",
  "scripts": {
    "dev": "webpack-dev-server",
    "start": "npm run webpack; node server",
    "webpack": "webpack --config ./webpack.config.js"
  },
  "dependencies": {
    "body-parser": "^1.15.0",
    "cors": "^2.7.1",
    "express": "^4.13.4",
    "falcor": "^0.1.16",
    "falcor-express": "^0.1.2",
    "falcor-http-datasource": "^0.1.3",
    "falcor-router": "0.2.12",
    "mongoose": "4.4.5",
    "react": "^0.14.7",
    "react-dom": "^0.14.7",
    "react-redux": "^4.4.0",
    "redux": "^3.3.1"
  },
  "devDependencies": {
    "babel": "^6.5.2",
    "babel-core": "^6.6.5",
    "babel-loader": "^6.2.4",
    "babel-polyfill": "^6.6.1",
    "babel-preset-es2015": "^6.6.0",
    "babel-preset-react": "^6.5.0",
    "babel-preset-stage-0": "^6.5.0",
    "webpack": "^1.12.14",
    "webpack-dev-server": "^1.14.1"
  }
}
```

```
}  
}
```

The important thing to notice about `package.json` is that we have removed the `^` from `"mongoose": "4.4.5"`. We did this because if NPM installs any higher version than `4.4.5`, then we get a warning in the bash/command line.

Our first working full-stack app

After that, you should have a complete full-stack version of the app working:



At almost every step, the UI part of our app is identical. The preceding screenshot is the publishing app, which does the following:

1. Fetches data from the DB using `Falcor-Express` and `Falcor-Router`.
2. The data moves from the backend (the source is MongoDB) to the frontend. We populate Redux's `src/reducers/article.js` state tree.
3. We render the DOM elements based on our single state tree.
4. All these steps allow us to take all of the full-stack app's data from the database, to the user's browser (so a user can see an article).

Summary

We haven't started to work on the app design, but in our book, we will use the Material Design CSS for React (<http://material-ui.com>). In the next chapter, we will start using it for user registration and login. After that, we will re-style the main page of our application using Material Design's components.

In order to give you a teaser of the goal (while working through the book), here is a screenshot of the app and how the publishing app will improve in the following chapters:



In the preceding screenshot, there is an example article from our application. We are using several Material Design components in order to make our work easier and the publishing app look more professional. You will learn it later.

Are you ready to work on the full-stack login and registration for our publishing app in the next chapter? Let's continue the fun.

2

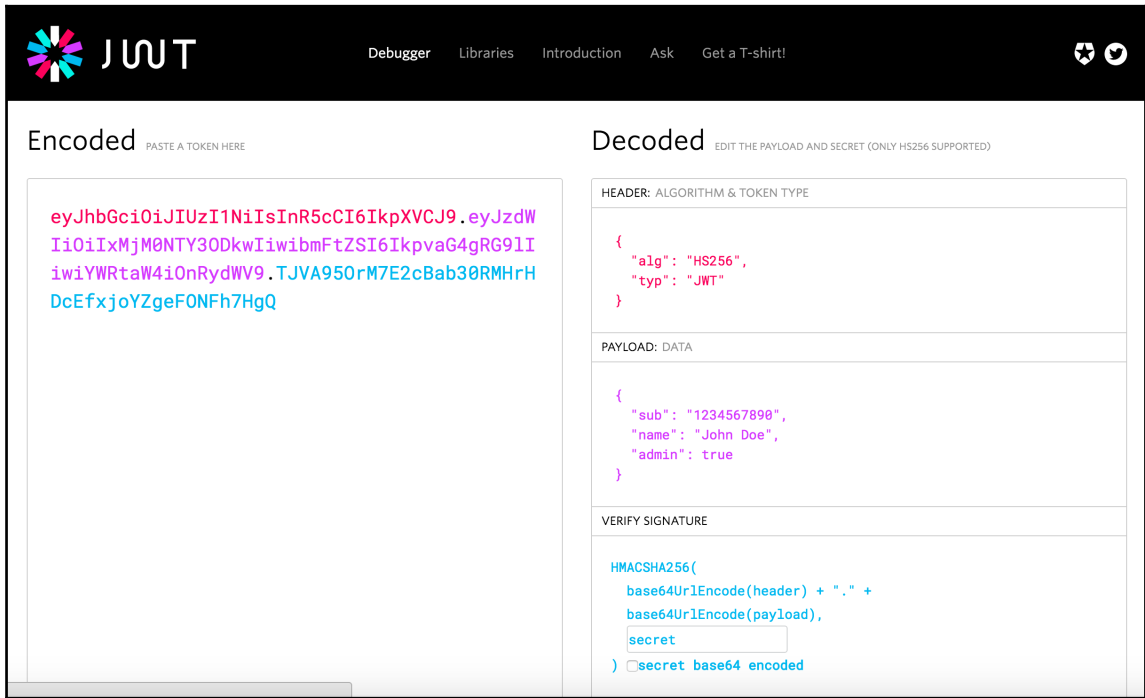
Full-Stack Login and Registration for Our Publishing App

JSON Web Token (JWT) is a security tokens format, which is relatively new, but works very well. It's an open standard (RFC 7519) that improves OAuth2 and OpenID connection when dealing with the problem of passing claims between parties in a web application environment.

In practice, the flow is as follows:

- The server assigns an encoded JSON object
- After client has been alerted, it sends that encoded token with every request to the server
- Based on that token, the server knows who is sending a request

It's worth visiting the <http://jwt.io/> website and playing with it before you start working with it:



The screenshot shows the JWT.io website interface. On the left, under the 'Encoded' tab, a JWT token is pasted: `eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG91IiwiaWF0IjoiYWRtaW4iOnRydWV9.TJVA950rM7E2cBab30RMHrHDcEfxjoYZgeFONfh7HgQ`. On the right, under the 'Decoded' tab, the token is decoded. The header shows `{ "alg": "HS256", "typ": "JWT" }`. The payload shows `{ "sub": "1234567890", "name": "John Doe", "admin": true }`. The signature verification section shows the HMACSHA256 formula and a text input field containing the word 'secret'.

After successful login, the JWT's solution provides an object to our frontend application that tells us about a current user's authorization:

```
{ 'iss': 'PublishginAppIssuer', 'name': 'John Doe', 'admin': true }
```

The `iss` is an issuer property--in our case it will be our publishing app's backend application. The name of the logged user is obvious--John Doe has logged in successfully. The `admin` property is just saying that an identified user (logged into our backend's app with the correct login and password) is an admin (`'admin': true` flag). You will learn how to use it in this chapter.

Besides what has been said in the preceding example, the JWT's response also contains information about subjects/claims, a signed SHA256's generated token, and an expiration date. The important rule here is that you must be sure about the issuer of your token. You need to trust the content provided along with the response. It may sound complicated, but it is very simple in real-life applications.

The important thing is that you need to keep the token generated by JWT protected---this will be elaborated upon later in this chapter.

The flow is as follows:

1. Our client's publishing app requests a token from our express's server.
2. The publishing backed app issues a token to the frontend Redux's app.
3. After that, each time we fetch data from the backend, we check if a user has access to the requested resources on the backend--the resource consumes the token.

In our case, the resource is a falcor-router's route, which has a close relationship with the backend, but this may work as well in more distributed platforms.

Remember that the JWT tokens are similar to private keys--you must keep them secure!

Structure of JWT token

The header has information that is required on the backend for recognizing what cryptographic operation to do based on that information (metadata, the algorithms, and keys being used):

```
{
  'typ': 'JWT',
  'alg': 'HS256'
}
```

In general, that part is done 100 percent out of the box for us, so we don't have to care about headers while implementing it.

The second part consists of claims provided in the JSON format, such as:

- **Issuer:** This lets us know who has issued the token
- **Audience:** This lets us know that this token has to be consumed by our application
- **Issue date:** This lets us know when the token has been created
- **Expiration date:** This lets us know when the token is expiring so we have to generate a new one
- **Subject:** This lets us know which part of the app can use the token (useful in bigger applications)

Besides these claims, we can create custom claims that are specifically defined by the app's creator:

```
{
  'iss': 'http://theIssuerAddress',
  'exp': '1450819372',
  'aud': 'http://myAppAddress',
  'sub': 'publishingApp',
  'scope': ['read']
}
```

New MongoDB users collection

We need to create a users' collection in our database. The users will have privileges allowing them to:

- Add new articles in our publishing application
- Edit existing articles in our publishing application
- Delete articles in our publishing application

The first step is that we need to create a collection. You can do this from the GUI in Robomongo (introduced at the beginning of the book), but we will use the command line.

First of all we need to create a file called `initPubUsers.js`:

```
$ [[you are in the root directory of your project]]
$ touch initPubUsers.js
```

Then add the following content to `initPubUsers.js`:

```
[
  {
    'username' : 'admin',
    'password' :
    'c5a0df4e293953d6048e78bd9849ec0ddce811f0b29f72564714e474615a7852',
    'firstName' : 'Kamil',
    'lastName' : 'Przeorski',
    'email' : 'kamil@mobilewebpro.pl',
    'role' : 'admin',
    'verified' : false,
    'imageUrl' : 'http://lorempixel.com/100/100/people/'
  }
]
```

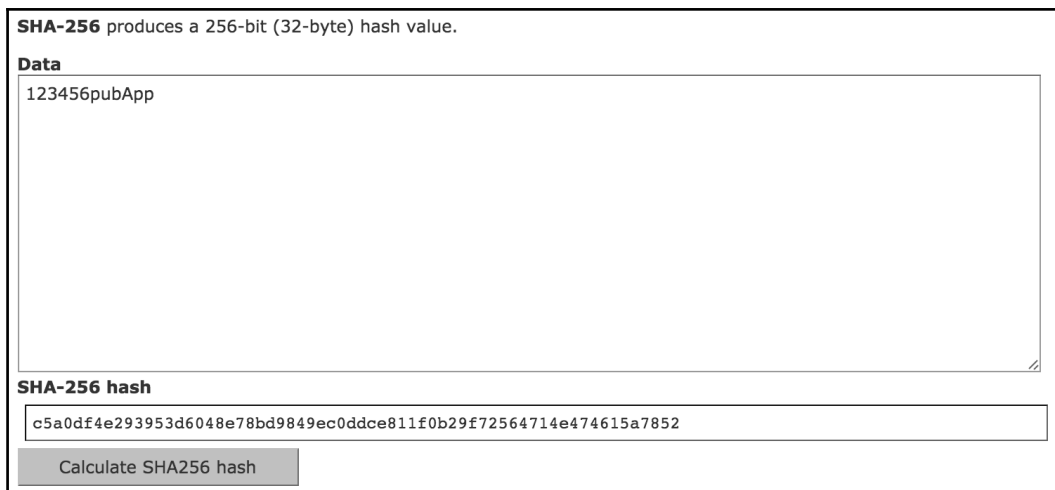
Explanation

The SHA256 string,

c5a0df4e293953d6048e78bd9849ec0ddce811f0b29f72564714e474615a7852, is the equivalent of a password, 123456, with a salt's string equal to pubApp.

If you want to generate this salted password hash yourself, then go to

<http://www.xorbin.com/tools/sha256-hash-calculator> and type 123456pubApp on their website. You will get the following screen:



The screenshot shows a web interface for calculating a SHA-256 hash. At the top, it states "SHA-256 produces a 256-bit (32-byte) hash value." Below this is a section labeled "Data" with a text input field containing "123456pubApp". Underneath the input field is the label "SHA-256 hash" and a corresponding text box displaying the resulting hash: "c5a0df4e293953d6048e78bd9849ec0ddce811f0b29f72564714e474615a7852". At the bottom of the interface is a button labeled "Calculate SHA256 hash".

These steps are required only at the beginning. Later we need to program a registration form that is salting the password for our own.

Importing the initPubUsers.js file into MongoDB

After we have the correct content in our `initPubUsers.js` file, we can run a command line as follows in order to import the new `pubUsers` collection to our database:

```
mongoimport --db local --collection pubUsers --jsonArrayinitPubUsers.js --host=127.0.0.1
```

You will get the same terminal output as what we we got after importing the article in Chapter 1, *Configuring Full-Stack with Node.js, Express.js, MongoDB, Mongoose, Falcor, and Redux*, looking similar to this:

```
2009-04-03T11:36:00.566+0200  connected to: 127.0.0.1
2009-04-03T11:36:00.569+0200  imported 1 document
```

Working on the login's falcor-route

Now we need to start working with the falcor-router in order to create a new endpoint that will use the JWT library to provide a unique token for the client-side app.

The first thing that we need to do is to provide `secret` on the backend.

Let's create that `secret` endpoint's config file:

```
$ cd server
$ touch configSecret.js
```

Now we need to put in the content of this `secret`:

```
export default {
  'secret': process.env.JWT_SECRET || 'devSecretGoesHere'
}
```

In future, we will use environment variables on the production server, so the notation `process.env.JWT_SECRET || 'devSecretGoesHere'` means that the environment variable of `JWT_SECRET` doesn't exist so use default `secret` endpoint's string, `devSecretGoesHere`. At this point we don't need any development environment variables.

Creating a falcor-router's login (backend)

In order to make our codebase more organized, instead of adding one more route to our `server/routes.js` file, we will make a new file called `routesSession.js` and in that file we will keep all the endpoints related to the current logged user's session.

Make sure you are in the `server` directory:

```
$ cd server
```

First open the `server.js` file in order to add one line of code that will allow you to post usernames and passwords to the backend. Add this:

```
app.use(bodyParser.urlencoded({extended: false}));
```

This has to be added under `app.use(bodyParser.json({extended: false}));` so you will end up with `server.js` code that begins as follows:

```
import http from 'http';
import express from 'express';
import cors from 'cors';
import bodyParser from 'body-parser';
import mongoose from 'mongoose';
import falcor from 'falcor';
import falcorExpress from 'falcor-express';
import Router from 'falcor-router';
import routes from './routes.js';

var app = express();
app.server = http.createServer(app);

// CORS - 3rd party middleware
app.use(cors());

// This is required by falcor-express middleware to work correctly with
falcor-browser
app.use(bodyParser.json({extended: false}));
app.use(bodyParser.urlencoded({extended: false}));
```


The last line is a new line that has to be added in order to make it work. Then create a new file in the same directory with:

```
$ touch routesSession.js
```

And put this initial content into the `routesSession.js` file:

```
export default [
  {
    route: ['login'],
    call: (callPath, args) => {
      const { username, password } = args[0];

      const userStatementQuery = {
        $and: [
          { 'username': username },
          { 'password': password }
        ]
      }
    }
  }
];
```

How the call routes work

We have just created an initial call login route in the `routesSession.js` file. Instead of using a GET method, we are going to use a 'call' (**call: async (callPath, args) => **). That is the equivalent to POST in the old RESTful approach.

The difference between the call and get methods in Falcor's routes is that we can provide arguments with `args`. That allows us to get from the client-side the username and the password:

The plan is that after we receive credentials with this:

```
const { username, password } = args[0];
```

Then we will check them against our database with one user admin. A user will need to know that the real plaintext password is 123456 in order to get a correct login JWTtoken:

We also have prepared in this step a `userStatementQuery`---this will be used later when querying a database:

```
const userStatementQuery = {
  $and: [
    { 'username': username },
    { 'password': password }
  ]
}
```

Separating the DB configs - `configMongoose.js`

We need to separate DB configs from `routes.js`:

```
$ [[we are in the server/ directory]]
$ touch configMongoose.js
```

And its new content:

```
import mongoose from 'mongoose';

const conf = {
  hostname: process.env.MONGO_HOSTNAME || 'localhost',
  port: process.env.MONGO_PORT || 27017,
  env: process.env.MONGO_ENV || 'local',
};

mongoose.connect(`mongodb://${conf.hostname}:
${conf.port}/${conf.env}&grave;`);

const articleSchema = {
  articleTitle:String,
  articleContent:String
};

const Article = mongoose.model('Article', articleSchema,
'article');

export default {
  Article
};
```

Explanation

We have just introduced the following new env variables: `MONGO_HOSTNAME`, `MONGO_PORT`, and `MONGO_ENV`. We will use them when preparing a production environment.

The `mongodb://${conf.hostname}:${conf.port}/${conf.env}` expression is using a templating feature available since EcmaScript6.

The rest of `configMongoose.js` will be known to you, as we have introduced it in Chapter 1, *Configuring Full-Stack with Node.js, Express.js, MongoDB, Mongoose, Falcor, and Redux*.

Improving the routes.js file

After we have created two new files, `configMongoose.js` and `routesSession.js`, we have to improve our `server/routes.js` file in order to make everything work together. The first step is to delete from `routes.js` the following code:

```
import mongoose from 'mongoose';

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

const articleSchema = {
  articleTitle:String,
  articleContent:String
};

const Article = mongoose.model('Article', articleSchema,
  'articles');
```

Replace it with the following new code:

```
import configMongoose from './configMongoose';
import sessionRoutes from './routesSession';
const Article = configMongoose.Article;
```

Also, we need to spread `sessionRoutes` into our current `PublishingAppRoutes` as follows:

```
const PublishingAppRoutes = [
  ...sessionRoutes,
  {
    route: 'articles.length',
```

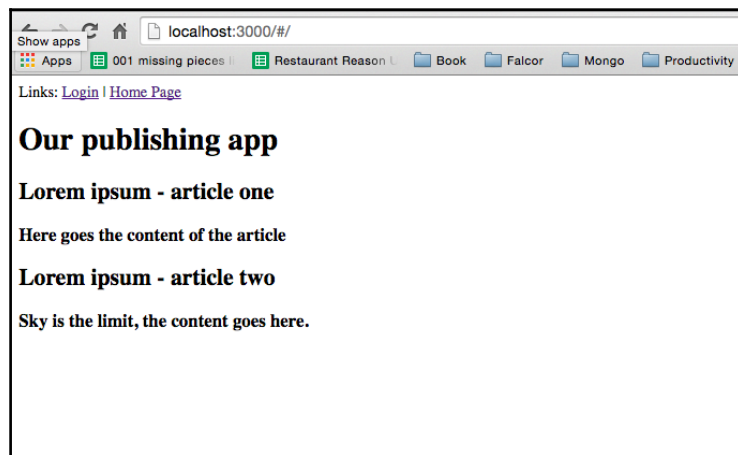
At the beginning of `PublishingAppRoutes` you need to spread `...sessionRoutes, routes,` so the login route will be available to use across the Falcor's routes.

Explanation

We got rid of the old code that was helping us to run the first Mongoose query that was fetching the articles, and we moved everything to `configMongoose` so that we can use it in different files around our project. We have also imported the session routes and later spread them with the `... spread` operation into the array called `PublishingAppRoutes`.

Checking to see if the app works before implementing JWT

At this point, when doing `npm start`, the app should be working and showing the list of articles:



When running with `npm start` you should get the following information, validating that everything works correctly:

```
Hash: eeeb09711c820a7978d5
Version2, : webpack 1.12.14
Time: 2609ms
  Asset      Size  Chunks             Chunk Names
app.js  1.9 MB          0  [emitted]  main
    [0] multi main 40 bytes {0} [built]
```

```
+ 634 hidden modules
Started on port 3000
```

Creating a Mongoose users' model

In the file `configMongoose.js` we need to create and export a `User` model. Add the following code to that file:

```
const userSchema = {
  'username' : String,
  'password' : String,
  'firstName' : String,
  'lastName' : String,
  'email' : String,
  'role' : String,
  'verified' : Boolean,
  'imageUrl' : String
};

const User = mongoose.model('User', userSchema, 'pubUsers');

export default {
  Article,
  User
};
```

Explanation

The `userSchema` describes our user's JSON model. The `user` is our Mongoose's model that is pointing to the `pubUsers` collection in our MongoDB. At the end, we are exporting the `User` model by adding it to the `export default`'s object.

Implementing JWT in the `routesSession.js` file

The first step is to export our `User` model into `routesSession` scope by adding at the top of that file an `import` statement:

```
import configMongoose from './configMongoose';
const User = configMongoose.User;
```

Install jsonwebtoken and crypto (for SHA256):

```
$ npm install --save jsonwebtoken crypto
```

After you have installed jsonwebtoken, we need to import it to routesSession.js:

```
import jwt from 'jsonwebtoken';
import crypto from 'crypto';
import jwtSecret from './configSecret';
```

After you have imported everything in the routesSession, continue on working with the route: ['login'].

You need to improve userStatementQuery, so it will have the saltedPassword instead of plain text:

```
const saltedPassword = password+'pubApp';
// pubApp is our salt string
const saltedPassHash = crypto
.createHash('sha256')
.update(saltedPassword)
.digest('hex');
const userStatementQuery = {
  $and: [
    { 'username': username },
    { 'password': saltedPassHash }
  ]
}
```

So instead of plain text, we will query a salted SHA256 password.

Under this userStatementQuery, return Promise, with the following details:

```
return User.find(userStatementQuery, function(err, user) {
  if (err) throw err;
}).then((result) => {
  if(result.length) {
    return null;
    // SUCCESSFUL LOGIN mocked now (will implement next)
  } else {
    // INVALID LOGIN
    return [
      {
        path: ['login', 'token'],
        value: "INVALID"
      },
      {
        path: ['login', 'error'],
```

```
        value: "NO USER FOUND, incorrect login  
        information"  
      }  
    ];  
  }  
  return result;  
});
```

Explanation

The `User.find` is a Promise that comes from Mongoose user's model (that we created in `configMongoose.js`)--this is a standard method. Then as a first argument we provide `userStatementQuery` which is that filter's object with the username and password in it: `(*{ username, password } = args[0];)`.

Next, we provide a function that is a callback when the query is done: `(function(err, user) {})`. We count the amount of results with `if(result.length) {}`.

If `result.length=== 0` then we have mocked return statement, and we are getting the else code running with the following return:

```
return [  
  {  
    path: ['login', 'token'],  
    value: "INVALID"  
  },  
  {  
    path: ['login', 'error'],  
    value: 'NO USER FOUND, incorrect login  
    information'  
  }  
];
```

As you will learn later, we will ask for that token's path on the frontend, `['login', 'token']`. In this case, we haven't found the correct username and the password provided so we return the "INVALID" string, instead of a JWT token. The path `['login', 'error']` is describing the error's type in more detail so that message can be shown to a user that has provided the invalid login credentials.

Successful login on falcor-route

We need to improve the successful login path. We have a case for handling an invalid login; we need to make a case that will handle a successful login, so replace this code:

```
return null; // SUCCESSFUL LOGIN mocked now (will implement next)
```

With this code that is returning the successful login's details:

```
const role = result[0].role;
const userDetailsToHash = username+role;
const token = jwt.sign(userDetailsToHash, jwtSecret.secret);
return [
  {
    path: ['login', 'token'],
    value: token
  },
  {
    path: ['login', 'username'],
    value: username
  },
  {
    path: ['login', 'role'],
    value: role
  },
  {
    path: ['login', 'error'],
    value: false
  }
];
```

Explanation

As you can see, the only thing that we fetch from DB right now is the role value === `result[0].role`. We need to add this to hash, because we don't want our app to be vulnerable so a normal user can get an admin role with some hacking. The value of the token is calculated based on `userDetailsToHash = username+role`---that's enough for now.

After we are fine with this, the only thing that needs to be done on the backend is returning the paths with values:

- The login token with `['login', 'token']`
- The username with `['login', 'username']`

- The logged user's role with ['login', 'role']
- Information that there were no errors at all with ['login', 'error']

The next step is to use this route on the frontend. Run the app and if everything is working fine, we can start coding on the frontend side.

Frontend side and Falcor

Let's create a new route for the login in our Redux application. In order to do that, we need to introduce the `react-router`:

```
$ npm install --save react-router@1.0.0redux-simple-router@0.0.10redux-thunk@1.0.0
```



It's important to use the correct NPM's version otherwise things get broke!

After we have installed them, we need to add routes in `src`:

```
$ cd src
$ mkdir routes
$ cd routes
$ touch index.js
```

Then make the content of this `index.js` file as follows:

```
import React from 'react';
import {Route, IndexRoute} from 'react-router';
import CoreLayout from '../layouts/CoreLayout';
import PublishingApp from '../layouts/PublishingApp';
import LoginView from '../views/LoginView';

export default (
  <Route component={CoreLayout} path='/'>
    <IndexRoute component={PublishingApp} name='home' />
    <Route component={LoginView} path='login' name='login' />
  </Route>
);
```

At this point, we are missing two components for our app called `CoreLayout` and `LoginView` (we will implement them in a minute).

The CoreLayout component

The `CoreLayout` component is the wrapper for our whole application. Create it by executing the following:

```
cd ../layouts/  
touch CoreLayout.js
```

Then, populate it with the following content:

```
import React from 'react';  
import {Link} from 'react-router';  
  
class CoreLayout extends React.Component {  
  static propTypes = {  
    children : React.PropTypes.element  
  }  
  
  render () {  
    return (  
      <div>  
        <span>  
          Links: <Link to='/login'>Login</Link> |  
          <Link to='/'>Home Page</Link>  
        </span>  
        <br/>  
        {this.props.children}  
      </div>  
    );  
  }  
}  
  
export default CoreLayout;
```

As you probably know, all the content of a current route will go into the `{this.props.children}` target (that is a basic `React.JS` concept that you must know beforehand). We also created two links to our routes as a header.

The LoginView component

For the time being, we will create a mocked LoginView component. Let's create the views directory:

```
$ pwd
$ [[you shall be at the src folder]]
$ mkdir views
$ cd views
$ touch LoginView.js
```

The content of the LoginView.js file is shown in the following code with the FORM GOES HERE placeholder:

```
import React from 'react';
import Falcor from 'falcor';
import falcorModel from '../falcorModel.js';
import {connect} from 'react-redux';
import {bindActionCreators} from 'redux';

const mapStateToProps = (state) => ({
  ...state
});

// You can add your reducers here
const mapDispatchToProps = (dispatch) => ({});

class LoginView extends React.Component {
  render () {
    return (
      <div>
        <h1>Login view</h1>
        FORM GOES HERE
      </div>
    );
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(LoginView);
```

We are done with all the missing pieces for the routes/index.js, but there is some other outstanding stuff to do before our app with the routing will start working.

A root's container for our app

Because our app is getting more complicated, we need to create a container that it will live in. In order to do that, let's do the following in the `src` location:

```
$ pwd
$ [[you shall be at the src folder]]
$ mkdir containers
$ cd containers
$ touch Root.js
```

The `Root.js` is going to be our main root file. The content of this file is as follows:

```
import React from 'react';
import {Provider} from 'react-redux';
import {Router} from 'react-router';
import routes from '../routes';
import createHashHistory from 'history/lib/createHashHistory';

const noQueryKeyHistory = createHashHistory({
  queryKey: false
});

export default class Root extends React.Component {
  static propTypes = {
    history : React.PropTypes.object.isRequired,
    store    : React.PropTypes.object.isRequired
  }

  render () {
    return (
      <Provider store={this.props.store}>
      <div>
      <Router history={noQueryKeyHistory}>
        {routes}
      </Router>
      </div>
      </Provider>
    );
  }
}
```

For now it's only a simple container, but later we will implement into it more features for debugging, hot reloading, and so on. The `noQueryKeyHistory` is saying to the router, that we don't want to have any random strings in our URL so our routes will be looking nicer (not a big deal, you can change the false flag to true, to see what I am talking about).

Remaining configuration for configureStore and rootReducer

Let's create `rootReducer` first. Why do we need it? Because in bigger applications you always end up with many different reducers; for example, in our app we will have reducers such as:

- **Article's reducer:** Which keeps stuff related to articles (`RETURN_ALL_ARTICLES` and so on)
- **Session's reducer:** Which will be related to our users' sessions (`LOGIN`, `REGISTER`, and so on)
- **Editor's reducer:** Which will be related to the editor's actions (`EDIT_ARTICLE`, `DELETE_ARTICLE`, `ADD_NEW_ARTICLE`, and so on)
- **Routing's reducer:** This will manage the state of our routes (out-of-the-box, because it is managed by the `redux-simple-router`'s external lib)

Let's create an `index.js` file in our `reducers` directory:

```
$ pwd
$ [[you shall be at the src folder]]
$ cd reducers
$ touch index.js
```

The content for the `index.js` is as follows:

```
import {combineReducers} from 'redux';
import {routeReducer} from 'redux-simple-router';
import article from './article';

export default combineReducers({
  routing: routeReducer,
  article
});
```

The new thing here is that we are introducing a `combineReducers` function from Redux. This is exactly what I've written before. We will have more than one reducer---in our case, we have also introduced the `routeReducer` from a `redux-simple-router`'s library.

The next step is to create the `configureStore` that will be managing our stores and also in order to implement a server rendering later in this book:

```
$ pwd
$ [[you shall be at the src folder]]
$ mkdir store
$ cd store
$ touch configureStore.js
```

The content for the `configureStore.js` file is as follows:

```
import rootReducer from '../reducers';
import thunk from 'redux-thunk';
import {applyMiddleware, compose, createStore} from 'redux';

export default function configureStore (initialState, debug =
false) {
  let createStoreWithMiddleware;
  const middleware = applyMiddleware(thunk);

  createStoreWithMiddleware = compose(middleware);

  const store = createStoreWithMiddleware(createStore)(
    rootReducer, initialState
  );
  return store;
}
```

In the preceding code, we are importing the `rootReducer` that we've created recently. We also import the `redux-thunk` lib which is very useful for server-side rendering (described later in the book).

At the end, we export a store which is composed of many different reducers (currently routing and the article's reducer that you can find in `reducer/index.js`) and is able to handle the server-rendering initial state.

Last tweaks in layouts/PublishingApp.js before running the app

The last thing that changed in our app is that we have out-of-date code in our publishing app.

Why is it outdated? Because we have introduced `rootReducer` and `combineReducers`. So if you check your code in the rendering of `PublishingApp` here, it won't work:

```
let articlesJSX = [];  
  
for(let articleKey in this.props) {  
  const articleDetails = this.props[articleKey];  
  
  const currentArticleJSX = (  
    <div key={articleKey}>  
      <h2>{articleDetails.articleTitle}</h2>  
      <h3>{articleDetails.articleContent}</h3>  
    </div>);  
  
  articlesJSX.push(currentArticleJSX);  
}
```

You need to change it to this:

```
let articlesJSX = [];  
  
for(let articleKey in this.props.article) {  
  const articleDetails = this.props.article[articleKey];  
  
  const currentArticleJSX = (  
    <div key={articleKey}>  
      <h2>{articleDetails.articleTitle}</h2>  
      <h3>{articleDetails.articleContent}</h3>  
    </div>);  
  
  articlesJSX.push(currentArticleJSX);  
}
```

Do you see the difference? The old `for(let articleKey in this.props)` has changed into `for(let articleKey in this.props.article)` and `this.props[articleKey]` has changed to `this.props.article[articleKey]`. Why? I will recall again: now every new reducer will be available in our app via its name created in `routes/index.js`. We have named our reducer `article`, so we now had to add this into `this.props.article` to make this stuff work together.

Last changes in src/app.js before running the app

The last thing is to improve the `src/app.js` so it will use the root's container. We need to change the old code:

```
// old codebase, to improve:
import React from 'react'
import { render } from 'react-dom'
import { Provider } from 'react-redux'
import { createStore } from 'redux'
import article from './reducers/article'
import PublishingApp from './layouts/PublishingApp'

const store = createStore(article)

render(
  <Provider store={store}>
    <PublishingApp store={store} />
  </Provider>,
  document.getElementById('publishingAppRoot')
);
```

We need to change the preceding code to the following:

```
import React from 'react';
import ReactDOM from 'react-dom';
import createBrowserHistory from 'history/lib/createBrowserHistory';
import {syncReduxAndRouter} from 'redux-simple-router';
import Root from './containers/Root';
import configureStore from './store/configureStore';

const target = document.getElementById('publishingAppRoot');
const history = createBrowserHistory();

export const store = configureStore(window.__INITIAL_STATE__);

syncReduxAndRouter(history, store);

const node = (
  <Root
    history={history}
    store={store} />
);

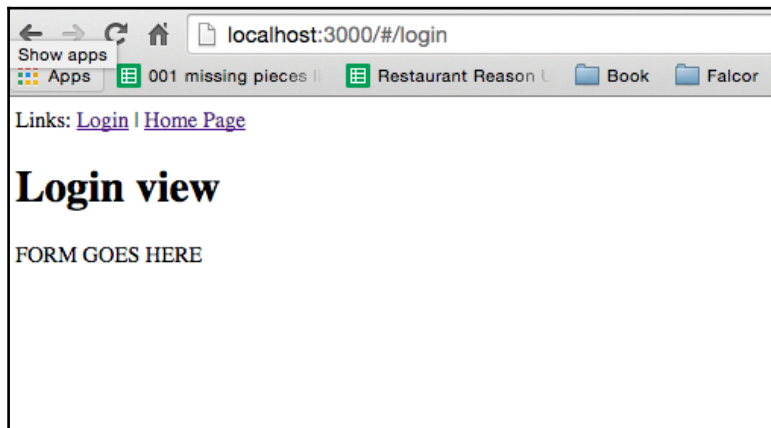
ReactDOM.render(node, target);
```


We start using the `Root` instead of the `Provider` directly, and we need to send the `store` and `history`'s props to the `Root` component. The `***export const store = configureStore(window.__INITIAL_STATE__)***` part is here for the server-side rendering which we will add in one of the following chapters. We also use the `history`'s library to manage the browser's history with the `JavaScript`.

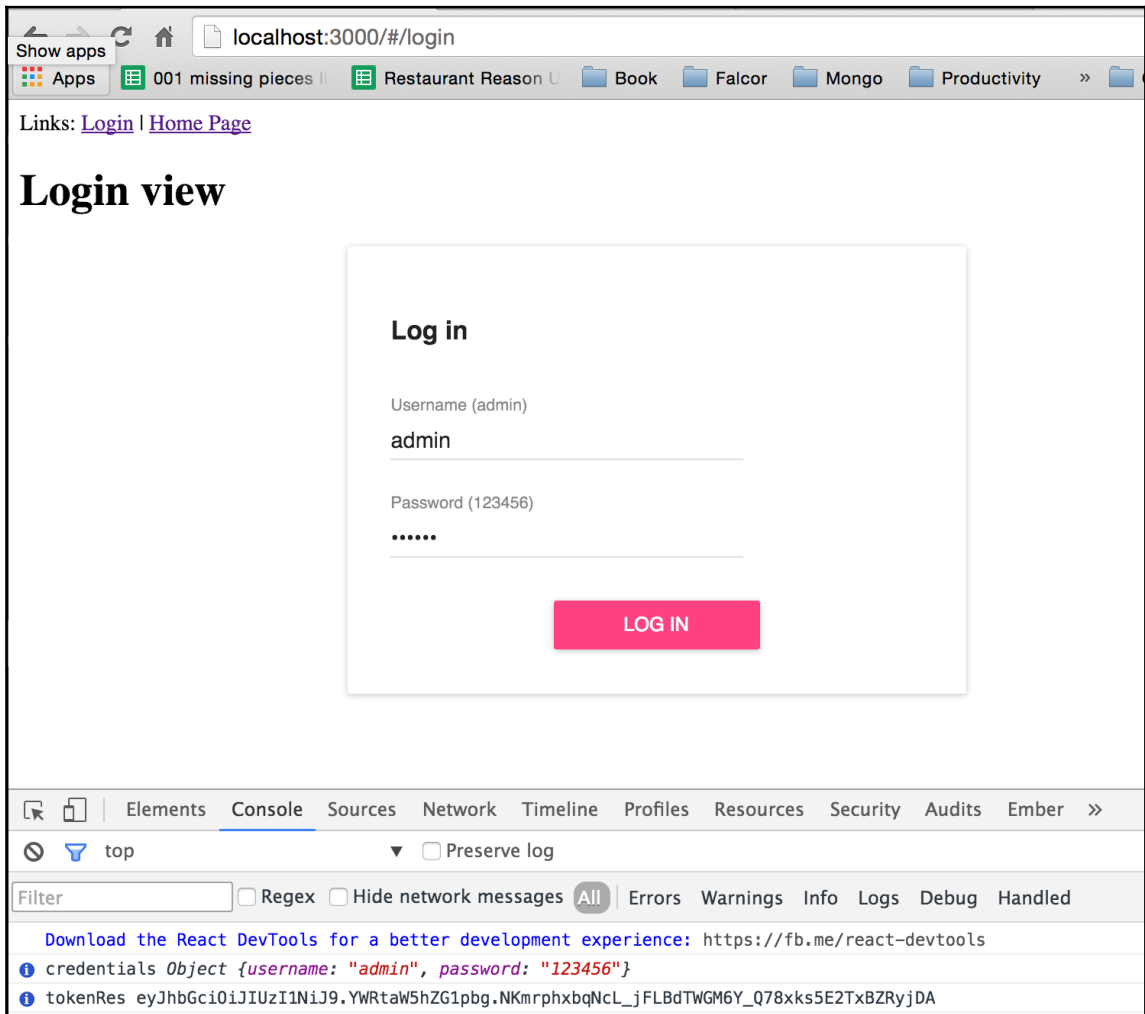
Screenshots of our running app

Currently when you do `npm start` then you will see the following two routes.

Home page



Login view



Working on the login form that will call the backend in order to authenticate

Okay, so we have done a lot of preparation in terms of having an extensible project structure (routes, rootReducer, configStores, and so on).

In order to make our app nicer from a user perspective, we will start using Material Design CSS. For making our work easier with forms, we will start using a `formsy-react` library. Let's install it:

```
$ npm i --save material-ui@0.14.4formsy-react@0.17.0
```

At the time of writing this book, the version .20.14.4 of Material UI is the best choice; I used this version because the ecosystem is changing so quickly that it's better to mark the used version in here so you won't have any surprises when following the instructions in this book.

The `formsy-react` library is a very handy library which will help us to validate our forms in the publishing app. We will use it on pages like login and registration as you will see on the next pages.

Working on LoginForm and DefaultInput components

After we are done with installing our new dependencies, let's create a folder that will keep files related to dumb components (the components that don't have access to any stores; they communicate with the other parts of our application with the help of callbacks---you will learn more about this later):

```
$ pwd
$ [[you shall be at the src folder]]
$ mkdir components
$ cd components
$ touch DefaultInput.js
```

Then make the content of this file as follows:

```
import React from 'react';
import {TextField} from 'material-ui';
import {HOC} from 'formsy-react';

class DefaultInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {currentText: null}
  }

  handleChange(e) {
    this.setState({currentText: e.target.value})
  }
}
```

```
this.props.setValue(e.target.value);
this.props.onChange(e);
}

render() {
  return (
<div>

<TextField
  ref={this.props.name}
  floatingLabelText={this.props.title}
  name={this.props.name}
  onChange={this.handleChange}
  required={this.props.required}
  type={this.props.type}
  value={this.state.currentText ?
    this.state.currentText : this.props.value}
  defaultValue={this.props.defaultValue} />
    {this.props.children}
</div>);
}
};

export default HOC(DefaultInput);
```

Explanation

The {HOC} from `formsy-react` is another way of decorating the component (aka `mixin` in React's ECMAScript5) with `export default HOC (DefaultInput)` --you can find more information about this at

<https://github.com/christianalfoni/formsy-react/blob/master/API.md#formsyhoc>.

We are also using the `TextField` from the `material-ui`; then it takes different properties. The following are the properties:

- `ref`: We want `ref` for each input with its name (username and e-mail).
- `floatingLabelText`: This is a nice looking floating text (known as label).
- `onChange`: This tells the function's name that has to be called when someone is typing into the `TextField`.

- `required`: This helps us to manage the required inputs in our form.
- `value`: This is, of course, the current value of our `TextField`.
- `defaultValue`: This is a value that is initial. It is very important to remember that it's called just once when a component is calling a constructor of the component.

The current text (`this.state.currentText`) is the value of the `DefaultInput` component---it changes with the new value on every `changeValue` event called by the callback given in the `TextField` `onChange` prop.

LoginForm and making it work with LoginView

The next step is to create `LoginForm`. This will use the `DefaultInput` component with the following commands:

```
$ pwd
$ [[you shall be at the components folder]]
$ touch LoginForm.js
```

Then the content of our `src/components/LoginForm.js` file is as follows:

```
import React from 'react';
import Formsy from 'formsy-react';
import {RaisedButton, Paper} from 'material-ui';
import DefaultInput from './DefaultInput';

export class LoginForm extends React.Component {
  constructor() {
    super();
  }

  render() {
    return (
      <Formsy.Form onSubmit={this.props.onSubmit}>
        <Paper zDepth={1} style={{padding: 32}}>
          <h3>Log in</h3>
          <DefaultInput
            onChange={this.handleChange} => {}
            name='username'
            title='Username (admin)'
            required />

          <DefaultInput
            onChange={this.handleChange} => {}
```

```
type='password'
name='password'
title='Password (123456)'
required />

<div style={{marginTop: 24}}>
  <RaisedButton
    secondary={true}
    type="submit"
    style={{margin: '0 auto', display: 'block', width:
      150}}
    label={'Log in'} />
</div>
</Paper>
</Formsy.Form>
  );
}
}
```

In the preceding code, we have our `LoginForm` component that is using the `DefaultInput`'s component. It's a simple `React.js` form that after being submit calls the `this.props.onSubmit`--this `onSubmit` function will be defined in `src/views/LoginView.js` smart component in a moment. I won't talk too much about attached styles on the component because it's up to you how you will style it--you will see a screenshot of applied styles of our app in a moment.

Improving the `src/views/LoginView.js`

The last part at our development at this stage before running our application is to improve the `LoginView` component.

In `src/views/LoginView.js` make the following changes. Import our new `LoginForm` component:

```
import {LoginForm} from '../components/LoginForm.js';
Add a new constructor of that component:
constructor(props) {
  super(props);
  this.login = this.login.bind(this);
  this.state = {
    error: null
  };
}
```

Then after you are done with imports and constructors, you need a new function called `login`:

```
async login(credentials) {
  console.info('credentials', credentials);

  await falcorModel
    .call(['login'], [credentials])
    .then((result) => result);

  const tokenRes = await falcorModel.getValue('login.token');
  console.info('tokenRes', tokenRes);
  return;
}
```

At this point, the `login` function only prints our new JWT token to the console--it's enough for now; later we will build more on top of it.

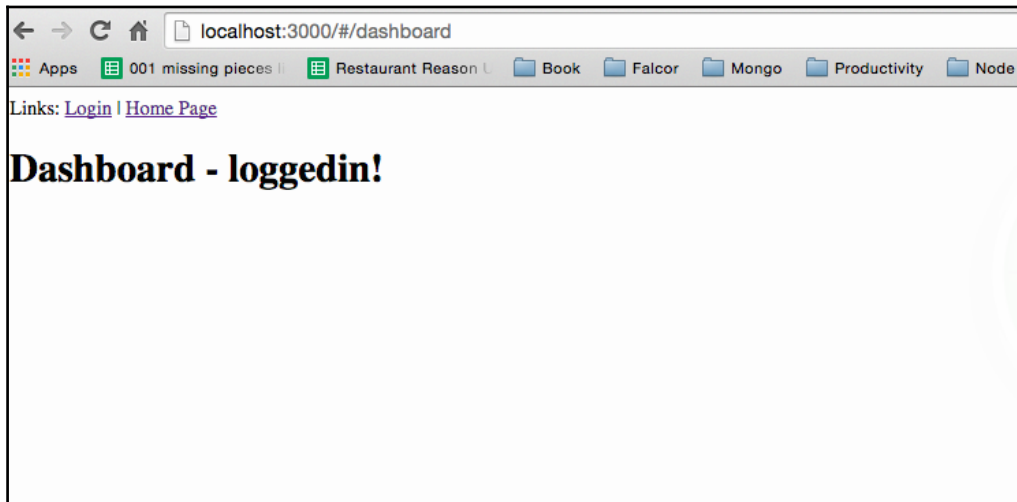
The last step here is to improve our `render` function from:

```
render () {
  return (
    <div>
    <h1>Login view</h1>
      FORM GOES HERE
    </div>
  );
}
```

To the new one, as follows:

```
render () {
  return (
    <div>
    <h1>Login view</h1>
    <div style={{maxWidth: 450, margin: '0 auto'}}>
    <LoginForm
    onSubmit={this.login} />
    </div>
    </div>
  );
}
```

Great! Now we are done! The following is what you will see after running `npm start` and running it in your browser:



As you can see in the browser's console, we can see the submitted credential's object (credentials Object {username: "admin", password: "123456"}) and also a token that has been fetched from the backend (tokenRes eyJhbGciOiJIUzI1NiJ9.YWRtaW5hZG1pbGg.NKmrphxbqNcL_jFLBdTWGM6Y_Q78xks5E2TxBZRyjDA). All this tells us that we are on track in order to implement the login's mechanism in our publishing application.



Important

If you get an error, then make sure that you have used the 123456 password while creating the hash. Otherwise, type in the custom password that is valid to your case.

Making DashboardView's component

At this point, we have a login feature that is not finished, but before continuing the work on it, let's create a simple `src/views/DashboardView.js` component that will be shown after a successful login:

```
$ pwd
$ [[you shall be at the views folder]]
$ touch DashboardView.js
```


Add some simple content as follows:

```
import React from 'react';
import Falcor from 'falcor';
import falcorModel from '../falcorModel.js';
import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';
import { LoginForm } from '../components/LoginForm.js';

const mapStateToProps = (state) => ({
  ...state
});

// You can add your reducers here
const mapDispatchToProps = (dispatch) => ({});

class DashboardView extends React.Component {
  render () {
    return (
      <div>
        <h1>Dashboard - loggedin!</h1>
      </div>
    );
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(DashboardView);
```

This is a simple component, which is static at this point. Later, we will build more features into it.

The last thing regarding the dashboards that we need to create is a new route in the `src/routes/index.js` file:

```
import DashboardView from '../views/DashboardView';

export default (
  <Route component={CoreLayout} path='/'>
    <IndexRoute component={PublishingApp} name='home' />
    <Route component={LoginView} path='login' name='login' />
    <Route component={DashboardView} path='dashboard' name='dashboard' />
  </Route>
);
```

We've just added second route using the react-router's config. It uses `DashboardView` component located in `../views/DashboardView` file.

Finishing the login's mechanism

The last improvements for login at this point of our publishing app remain at the `src/views/LoginView.js` location:

First of all, let's add handling an invalid login:

```
console.info('tokenRes', tokenRes);

if(tokenRes === 'INVALID') {
  const errorRes = await falcorModel.getValue('login.error');
  this.setState({error: errorRes});
  return;
}

return;
```

We have added this `if(tokenRes === 'INVALID')` in order to update the error state with `this.setState({error: errorRes})`.

The next step is to add into the render function `Snackbar` that will show to the user a type of error. At the top of the `LoginView` component add this import:

```
import { Snackbar } from 'material-ui';
```

Then you need to update the render function as follows:

```
<Snackbar
  autoHideDuration={4000}
  open={!!this.state.error}
  message={this.state.error || ''}
  onRequestClose={() => null} />
```

So after adding it, the render function will look like this:

```
render () {
  return (
    <div>
      <h1>Login view</h1>
      <div style={{maxWidth: 450, margin: '0 auto'}}>
        <LoginForm
          onSubmit={this.login} />
      </div>
```

```
<Snackbar autoHideDuration={4000}
  open={!this.state.error}
  message={this.state.error || ''}
onRequestClose={() => null} />
</div>
);
}
```

The `SnackBar` `onRequestClose` is required here otherwise you will get a warning in the developer's console from the Material UI. Okay, so we are handling login's error, now let's work on successful logins.

Handling successful logins in the LoginView's component

For handling a successful token's backend responses add the login function:

```
if(tokenRes === 'INVALID') {
  const errorRes = await falcorModel.getValue('login.error');
  this.setState({error: errorRes});
  return;
}
```

A new code for handling correct responses, as follows:

```
if(tokenRes) {
  const username = await falcorModel.getValue('login.username');
  const role = await falcorModel.getValue('login.role');

  localStorage.setItem('token', tokenRes);
  localStorage.setItem('username', username);
  localStorage.setItem('role', role);

  this.props.history.pushState(null, '/dashboard');
}
```

Explanation

After we know that the `tokenRes` is not `INVALID` and it's not undefined (otherwise shows a fatal error to the user), then we follow certain steps:

We are fetching the username from the Falcor's model (`await falcorModel.getValue('login.username')`). We are fetching the user's role (`await falcorModel.getValue('login.role')`). Then we save all the known variables from the backend into `localStorage` with:

```
localStorage.setItem('token', tokenRes);
localStorage.setItem('username', username);
localStorage.setItem('role', role);
```

At the same end we are sending our user to the `/dashboard` route with the use of `this.props.history.pushState(null, '/dashboard')`.

A few important notes about DashboardView and security

At this point, we won't secure `DashboardView` as there isn't any important stuff to secure---we will do it later when we put more assets/features into this route, which at the end of our book will be an editor's dashboard that will give control over all articles in the system.

The only remaining step for us is to make it a `RegistrationView` component. This route will also be available for everyone at this point. Later in the book, we will make a mechanism so that only the main admin will be able to add new editors into the system (and manage them).

Starting work on the new editor's registration

In order to wrap up the registration, let's first make some changes in our user's scheme from Mongoose's config file at the location `server/configMongoose.js`:

```
const userSchema = {
  'username' : String,
  'password' : String,
  'firstName' : String,
  'lastName' : String,
  'email' : String,
  'role' : String,
  'verified' : Boolean,
  'imageUrl' : String
};
```

To the new scheme as follows:

```
const userSchema = {
  'username' : { type: String, index: {unique: true, dropDups: true }},
  'password' : String,
  'firstName' : String,
  'lastName' : String,
  'email' : { type: String, index: {unique: true, dropDups: true }},
  'role' : { type: String, default: 'editor' },
  'verified' : Boolean,
  'imageUrl' : String
};
```

As you can see, we have added unique indexes to the username and the email fields. Also, we have added a default value for a role, as any next user in our collection will be an editor (not an admin).

Adding register's falcor-route

In the file located at `server/routesSession.js`, you need to add a new route (next to the login's route):

```
{
  route: ['register'],
  call: (callPath, args) =>
  {
    const newUserObj = args[0];
    newUserObj.password = newUserObj.password+'pubApp';
    newUserObj.password = crypto
      .createHash('sha256')
      .update(newUserObj.password)
      .digest('hex');
    const newUser = new User(newUserObj);
    return newUser.save((err, data) => { if (err) return err; })
      .then ((newRes) => {
        /*
          got new obj data, now let's get count:
        */
        const newUserDetail = newRes.toObject();

        if(newUserDetail._id) {
          return null; // Mocked for now
        } else {
          // registration failed
          return [
            {

```

```
        path: ['register', 'newUserId'],
        value: 'INVALID'
      },
      {
        path: ['register', 'error'],
        value: 'Registration failed - no id has been
        created'
      }
    ];
  }
  return;
}).catch((reason) => console.error(reason));
}
```

What this code is actually doing is simply receiving the new user's object from the frontend via `const newUserObj = args[0]`.

Then we are salting the password that we will store in our database:

```
newUserObj.password = newUserObj.password+'pubApp';
newUserObj.password = crypto
  .createHash('sha256')
  .update(newUserObj.password)
  .digest('hex');
```

Then we are creating a new user model from Mongoose via `const newUser = new User(newUserObj)`, because the `newUser` variable is a new model (not saved yet) of the user. Next we need to save it with this code:

```
return newUser.save((err, data) => { if (err) return err; })
```

After it's saved into the db and the Promise has been resolved, we are managing an invalid entry to the db first by making the Mongoose result's object into a simple JSON structure with `const newUserDetail = newRes.toObject();`.

And after we are done with it, then we are returning an `INVALID` information to the Falcor's model:

```
// registration failed
return [
  {
    path: ['register', 'newUserId'],
    value: 'INVALID'
  },
  {
    path: ['register', 'error'],
```

```
        value: 'Registration failed - no id has been created'
    }
}
```

So, we are done with handling an invalid user registration from Falcor. The next step is to replace this:

```
// you shall already have this in your codebase, just a recall
if(newUserDetail._id) {
    return null; // Mocked for now
}
The preceding code needs to be replaced with:
if(newUserDetail._id) {
    const newUserId = newUserDetail._id.toString();

    return [
        {
            path: ['register', 'newUserId'],
            value: newUserId
        },
        {
            path: ['register', 'error'],
            value: false
        }
    ];
}
```

Explanation

We need to cast our new user's ID into the string, `newUserId = newUserDetail._id.toString()` (otherwise it will break the code).

As you can see, we have a standard return statement that complements the model in Falcor.

To quickly recall, after it returns correctly on the backend, we will be able to request this value on the frontend as follows: `const newUserId = await falcorModel.getValue(['register', 'newUserId']);` (this is just an example of how to fetch this new `UserId` on the client-side--don't write it into your code, we will do it in a minute).

You will get used to it after few more examples.

Frontend implementation (RegisterView and RegisterForm)

Let's first create a component that will manage on the frontend, the register's form with the following actions:

```
$ pwd
$ [[you shall be at the components folder]]
$ touch RegisterForm.js
```

The content of that file will be:

```
import React from 'react';
import Formsy from 'formsy-react';
import {RaisedButton, Paper} from 'material-ui';
import DefaultInput from './DefaultInput';

export class RegisterForm extends React.Component {
  constructor() {
    super();
  }

  render() {
    return (
      <Formsy.FormonSubmit={this.props.onSubmit}>
      <Paper zDepth={1} style={{padding: 32}}>
      <h3>Registration form</h3>
      <DefaultInput
        onChange={ (event) => {}}
        name='username'
        title='Username'
        required />

      <DefaultInput
        onChange={ (event) => {}}
        name='firstName'
        title='Firstname'
        required />

      <DefaultInput
        onChange={ (event) => {}}
        name='lastName'
        title='Lastname'
        required />

      <DefaultInput
```



```
      onChange={ (event) => {}}
      name='email'
      title='Email'
      required />

    <DefaultInput
      onChange={ (event) => {}}
      type='password'
      name='password'
      title='Password'
      required />

    <div style={{marginTop: 24}}>
      <RaisedButton
        secondary={true}
        type="submit"
        style={{margin: '0 auto', display:
          'block', width: 150}}
        label={'Register'} />

    </div>
  </Paper>
</Formsy.Form>
  );
}
}
```

The preceding registration component is creating a form exactly the same way as on `LoginForm`. After a user clicks the `Register` button, it sends a callback to the `src/views/RegisterView.js` component (we will create this in a moment).

Remember that in the components' directory we keep only DUMB components so all the communication with the rest of the app must to be done via callbacks like in this example.

RegisterView

Let's create a `RegisterView` file:

```
$ pwd
$ [[you shall be at the views folder]]
$ touch RegisterView.js
```

Its content is:

```
import React from 'react';
import falcorModel from '../falcorModel.js';
import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';
import { Snackbar } from 'material-ui';
import { RegisterForm } from '../components/RegisterForm.js';

const mapStateToProps = (state) => ({
  ...state
});
const mapDispatchToProps = (dispatch) => ({});
```

These are standard things that we use in our smart components (we need `falcorModel` in order to communicate with the backend and `mapStateToProps` and `mapDispatchToProps` in order to communicate with our Redux's store/reducer).

Okay, that's not all for the register view; next let's add a component:

```
const mapDispatchToProps = (dispatch) => ({});

class RegisterView extends React.Component {
  constructor(props) {
    super(props);
    this.register = this.register.bind(this);
    this.state = {
      error: null
    };
  }

  render () {
    return (
      <div>
        <h1>Register</h1>
        <div style={{maxWidth: 450, margin: '0 auto'}}>
          <RegisterForm
            onSubmit={this.register} />
        </div>
      </div>
    );
  }
}

export default connect(mapStateToProps, mapDispatchToProps)(RegisterView);
```

As you can see in the preceding code snippet, we are missing the `register` function, so between the constructor and the render function add the function, as follows:

```
async register (newUserModel) {console.info("newUserModel",  newUserModel);

  await falcormodel
    .call(['register'],[newUserModel])
    .then((result) =>result);

  const newUserId = await falcormodel.getValue(['register',
    'newUserId']);

  if(newUserId === 'INVALID') {
    const errorRes = await falcormodel.getValue('register.error');

    this.setState({error: errorRes});
    return;
  }

  this.props.history.pushState(null, '/login');
}
```

As you can see, the `async register (newUserModel)` function is asynchronous and friendly to the awaits. Next we are just logging into the console what a user has submitted with `console.info("newUserModel", newUserModel)`. After that, we query the falcormodel router with a call:

```
await falcormodel
  .call(['register'],[newUserModel])
  .then((result) => result);
```

After we have called the router, we fetch the response with:

```
const newUserId = await falcormodel.getValue(['register', 'newUserId']);
```

Depending on the response from the backend, we do the following:

- For `INVALID` we are fetching and setting error message into the component's state (`this.setState({error: errorRes})`)
- If the user has registered correctly, then we have their new ID and we are asking the user to login with the history's push state (`this.props.history.pushState(null, '/login');`)

We didn't create a route inside `routes/index.js` for `RegisterView` and there is no link in `CoreLayout` so our user is unable to use it. Add new imports in `routes/index.js`:

```
import RegisterView from '../views/RegisterView';
```

Then add a route, so the export default from `routes/index.js` will look like this:

```
export default (  
  <Route component={CoreLayout} path="/" />  
  <IndexRoute component={PublishingApp} name='home' />  
  <Route component={LoginView} path='login' name='login' />  
  <Route component={DashboardView} path='dashboard' name='dashboard' />  
  <Route component={RegisterView} path='register' name='register' />  
  </Route>  
);
```

And finally, add a link inside the `src/layouts/CoreLayout.js` file's render method:

```
render () {  
  return (  
    <div>  
      <span>  
        Links:<Link to='/register'>Register</Link>  
        <Link to='/login'>Login</Link>  
        <Link to="/">Home Page</Link>  
      </span>  
      <br/>  
      {this.props.children}  
    </div>  
  );  
}
```

At this point ,we should be able to register with this form:

The screenshot shows a web browser window with the address bar displaying 'localhost:3000/#/register'. The browser's tab bar shows several tabs: 'Apps', '001 missing pieces', 'Restaurant Reason U', 'Book', and 'Other Bo'. Below the address bar, there are links: 'Links: Register | Login | Home Page'. The main heading of the page is 'Register'. Below this heading is a 'Registration form' box. Inside this box, there are five input fields: 'Username', 'Firstname', 'Lastname', 'Email', and 'Password'. At the bottom of the form box is a pink button labeled 'REGISTER'.

Summary

In the next chapter, we will start working on the server-side rendering of our app. This means that on each request to our Express's server, we will generate the HTML markup based on the request from the client side. That feature is very useful for apps like ours where the speed of web loading is very important for such users as ours.

You can imagine that most of the news sites are for entertainment and that means a short attention span from our potential users. The speed of loading is important. There are also some opinions that the server-side rendering also helps for search engine optimization reasons.

The crawlers have easier ways to *read* the text from our article as they don't need to execute the JavaScript in order to fetch it from the server (in comparison to non-server-side rendering single-page apps).

At least one thing is certain: if you have a server-side rendering on your articles' publishing app then Google may see that you care about the fast loading of your app and so it will probably give you some disadvantage over full single-page websites that don't care about server-side rendering.

3

Server-Side Rendering

Universal JavaScript, or isomorphic JavaScript, are different names for a feature that we are going to implement in this chapter. To be more exact, we will develop our app and render the app's pages on both the server and client side. It will be different to **Angular1** or Backbone single-page apps which are mainly rendered on the client side. Our approach is more complicated in technological terms as you need to deploy your full-stack skills which work on server-side rendering, but having this experience will make you a more desirable programmer so you can advance your career to the next level--you will be able to charge more for your skills on the market.

When the server side is worth implementing

Server-side rendering is a very useful feature in text content (like news portals) start-ups/companies, because it helps achieve better indexing by different search engines. It's an essential feature for any news and content-heavy website, because it helps grow organic traffic. In this chapter, we will also run our app with server-side rendering. Other companies where server-side rendering may be useful are entertainment businesses where users have less patience and they might close the browser if a webpage is loading slowly. In general, all **B2C** (consumer facing) apps should use server-side rendering to improve the experience for the people who visit their websites.

Our focus in this chapter will include the following:

- Making the whole server-side code rearrangements to prepare for the server-side rendering
- Starting to use react-dom/server and its `renderToString` method

- `RoutingContext` and `match` for the react-router working on the server side
- Improving the client-side application so it will be optimized for an isomorphic JavaScript application

Are you ready? Our first step is to mock the database's response on the backend (we will create a real DB query after whole server-side rendering works correctly on the mocked data).

Mocking the database response

First of all, we will mock our database response on the backend in order to prepare to go into server-side rendering directly; we will change it later in this chapter:

```
$ [[you are in the server directory of your project]]
$ touch fetchServerSide.js
```

The `fetchServerSide.js` file will consist of all functions that will fetch data from our database in order to make the server side work.

As mentioned earlier, we will mock it for now, with the following code in `fetchServerSide.js`:

```
export default () => {
  return {
    'article':{
      '0': {
        'articleTitle': 'SERVER-SIDE Lorem ipsum - article one',
        'articleContent': 'SERVER-SIDE Here goes the content of the
          article'
      },

      '1': {
        'articleTitle': 'SERVER-SIDE Lorem ipsum - article two',
        'articleContent': 'SERVER-SIDE Sky is the limit, the
          content goes here.'
      }
    }
  }
}
```


The goal of making this mocked object is that we will be able to see if our server-side rendering works correctly after implementation because, as you have probably already spotted, we have added `SERVER-SIDE` at the beginning of each title and content--so it will help us to learn if our app is getting the data from server-side rendering. Later, this function will be replaced with a query to MongoDB.

The next thing that will help us implement the server-side rendering is to make a `handleServerSideRender` function that will be triggered each time a request hits the server.

In order to make the `handleServerSideRender` trigger every time the frontend calls our backend, we need to use Express middleware using `app.use`. So far, we have used some external libraries such as:

```
app.use(cors());
app.use(bodyParser.json({extended: false}))
```

For the first time in this book, we will write our own, small, middleware function that behaves in a similar way to `cors` or `bodyParser` (the external libs that are also middleware).

Before doing so, let's import the dependencies that are required in React's server-side rendering (`server/server.js`):

```
import React from 'react';
import {createStore} from 'redux';
import {Provider} from 'react-redux';
import {renderToStaticMarkup} from 'react-dom/server';
import ReactRouter from 'react-router';
import {RoutingContext, match} from 'react-router';
import * as hist from 'history';
import rootReducer from '../src/reducers';
import reactRoutes from '../src/routes';
import fetchServerSide from './fetchServerSide';
```

So, after adding all those imports of `server/server.js`, the file will look as follows:

```
import http from 'http';
import express from 'express';
import cors from 'cors';
import bodyParser from 'body-parser';
import falcor from 'falcor';
import falcorExpress from 'falcor-express';
import falcorRouter from 'falcor-router';
import routes from './routes.js';
import React from 'react'
```

```
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import { renderToStaticMarkup } from 'react-dom/server'
import { Router } from 'react-router';
import { RouterContext, match } from 'react-router';
import * as hist from 'history';
import rootReducer from '../src/reducers';
import reactRoutes from '../src/routes';
import fetchServerSide from './fetchServerSide';
```

Most of that stuff explained here is similar to client-side development in previous chapters. What is important is to import history in the given way, as in the example: `import * as hist from 'history'`. The `RouterContext`, `match` is a way of using `React-Router` on the server side. The `renderToStaticMarkup` function is going to generate an HTML markup for us on the server side.

After we have added the new imports, then under Falcor's middleware setup:

```
// this already exists in your codebase
app.use('/model.json', falcorExpress.dataSourceRoute((req, res) => {
  return new falcorRouter(routes); // this already exists in your
    codebase
}));
```

Under that `model.json` code, add the following:

```
let handleServerSideRender = (req, res) =>
{
  return;
};

let renderFullHtml = (html, initialState) =>
{
  return;
};
app.use(handleServerSideRender);
```

The `app.use(handleServerSideRender)` event is fired each time the server side receives a request from a client's application. Then we will prepare the empty functions that we will use:

- `handleServerSideRender`: It will use `renderToString` in order to create a valid server-side HTML markup
- `renderFullHtml`: It is a helper function that will embed our new React's HTML markup into a whole HTML document as we will see later

The `handleServerSideRender` function

First, we are going to create a new Redux store instance that will be created on every call to the backend. The main goal of this is to give the initial state information to our application so it can create a valid markup based on the current request.

We will use the `Provider` component that we have already used in our client-side's app that will be wrapping the `Root` component. That will make the store available to all our components.

The most important part here is `ReactDOMServer.renderToString()` to render the initial HTML markup of our application, before we send the markup to the client side.

The next step is to get the initial state from the Redux store by using the `store.getState()` function. The initial state will be passed along in our `renderFullHtml` function, as you will learn in a moment.

Before we work on the two new functions (`handleServerSideRender` and `renderFullHtml`), replace this in `server.js`:

```
app.use(express.static('dist'));
```

Replace with the following:

```
app.use('/static', express.static('dist'));
```

That's everything in our `dist` project. It will be available as a static file under the localhost address (`http://localhost:3000/static/app.js*`). This will help us make a single-page app after initial server-side rendering.

Also make sure that `app.use('/static', express.static('dist'));` is placed directly under `app.use(bodyParser.urlencoded({extended: false }));`. Otherwise it may not work if you misplace this in the `server/server.js` file.

After you are done with the preceding work of `express.static`, let's make this function more complete:

```
let renderFullHtml = (html, initialState) =>
{
  return; // this is already in your codebase
};
```

Replace the preceding empty function with the following improved version:

```
let renderFullPage = (html, initialState) =>
{
  return &grave;
  <!doctype html>
  <html>
  <head>
  <title>Publishing App Server Side Rendering</title>
  </head>
  <body>
  <h1>Server side publishing app</h1>
  <div id="publishingAppRoot">${html}</div>
  <script>
  window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}
  </script>
  <script src="/static/app.js"></script>
  </body>
  </html>
  &grave;
};
```

In short, this HTML code will be sent by our server when a user hits the website for the first time so we need to create the HTML markup with a body and head in order to make it work. The server-side publishing app's header is here just temporarily, to check if we are fetching the server-side HTML template correctly. Later you can find `$html` with the following command:

`${html}`



Notice that we are using ES6 templates (Google ES6 template literals) syntax with ```.

In this, we will later put the value that is generated by the `renderToStaticMarkup` function. The last step in the `renderFullPage` function is to give the initial, server-side rendering state in the window with `window.INITIAL_STATE = ${JSON.stringify(initialState)}` so the app can work correctly on the client-side with data fetched on the backend when the first request to the server has been made.

Okay, next let's focus on the `handleServerSideRender` function by replacing the following:

```
let handleServerSideRender = (req, res) =>
{
  return;
};
```

Replace with the more complete version of the function as follows:

```
let handleServerSideRender = (req, res, next) => {
  try {
    let initMOCKstore = fetchServerSide(); // mocked for now

    // Create a new Redux store instance
    const store = createStore(rootReducer, initMOCKstore);
    const location = hist.createLocation(req.path);

    match({
      routes: reactRoutes,
      location: location,
    }, (err, redirectLocation, renderProps) => {
      if (redirectLocation) {
        res.redirect(301, redirectLocation.pathname +
          redirectLocation.search);
      } else if (err) {
        console.log(err);
        next(err);
        // res.send(500, error.message);
      } else if (renderProps === null) {
        res.status(404)
          .send('Not found');
      } else {
        if (typeof renderProps === 'undefined') {
          // using handleServerSideRender middleware not required;
          // we are not requesting HTML (probably an app.js or other
          file)
          return;
        }

        let html = renderToStaticMarkup(
          <Provider store={store}>
            <RoutingContext {...renderProps}/>
          </Provider>
        );
      }
    });
  }
};
```

```
const initialState = store.getState()

let fullHTML = renderFullPage(html, initialState);
res.send(fullHTML);
}
});
} catch (err) {
  next(err)
}
}
```

The `let initMOCKstore = fetchServerSide();` expression is fetching data from MongoDB (mocked for now, to be improved later). Next, we create a server-side's Redux story with `store = createStore(rootReducer, initMOCKstore)`. We also need to prepare a correct location for our app's user consumable by the react-router with `location = hist.createLocation(req.path)` (in `req.path` there is a simple path which is in the browser; `/register` or `/login` or simply `main page /`). The function `match` is provided by the react-router in order to match the correct route on the server side.

When we have matched the route on the server side, we will see the following:

```
// this is already added to your codebase:
let html = renderToStaticMarkup(
  <Provider store={store}>
    <RoutingContext {...renderProps}/>
  </Provider>
);

const initialState = store.getState();

let fullHTML = renderFullPage(html, initialState);
res.send(fullHTML);
```

As you can see here, we are creating the server-side HTML markup with `renderToStaticMarkup`. Inside this function, there is a `Provider` with the store that has previously been fetched with the `let initMOCKstore = fetchServerSide()`. Inside the `Redux Provider` we have `RoutingContext` which simply passes all required props down into our app so we can have a correctly-created markup server side.

After all that, we only need to prepare `initialState` of our `Redux Store` with `const initialState = store.getState();` and later `let fullHTML = renderFullPage(html, initialState);` to have everything we need to send it to the client with `res.send(fullHTML)`.

We are done with server-side preparations.

Double-check server/server.js

Before we go forward with the client-side development, we will double-check `server/server.js` as the order of our code is important and this is one of the error-prone files:

```
import http from 'http';
import express from 'express';
import cors from 'cors';
import bodyParser from 'body-parser';
import falcor from 'falcor';
import falcorExpress from 'falcor-express';
import falcorRouter from 'falcor-router';
import routes from './routes.js';
import React from 'react'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import { renderToStaticMarkup } from 'react-dom/server'
import ReactRouter from 'react-router';
import { RoutingContext, match } from 'react-router';
import * as hist from 'history';
import rootReducer from '../src/reducers';
import reactRoutes from '../src/routes';
import fetchServerSide from './fetchServerSide';

const app = express();

app.server = http.createServer(app);
// CORS - 3rd party middleware
app.use(cors());
// This is required by falcor-express middleware to work correctly
// with falcor-browser
app.use(bodyParser.json({extended: false}));

app.use(bodyParser.urlencoded({extended: false}));

app.use('/static', express.static('dist'));

app.use('/model.json', falcorExpress.dataSourceRoute(function(req, res) {
  return new falcorRouter(routes);
}));

let handleServerSideRender = (req, res, next) => {
  try {
    let initMOCKstore = fetchServerSide(); // mocked for now
    // Create a new Redux store instance
```

```
const store = createStore(rootReducer, initMOCKstore);
const location = hist.createLocation(req.path);
match({
  routes: reactRoutes,
  location: location,
}, (err, redirectLocation, renderProps) => {
  if (redirectLocation) {

    res.redirect(301, redirectLocation.pathname +
      redirectLocation.search);
  } else if (err) {

    next(err);
    // res.send(500, error.message);
  } else if (renderProps === null) {

    res.status(404)
      .send('Not found');
  } else {
    if (typeof renderProps === 'undefined') {
      // using handleServerSideRender middleware not
      // required;
      // we are not requesting HTML (probably an app.js or
      // other file)

      return;
    }
    let html = renderToStaticMarkup(
      <Provider store={store}>
        <RoutingContext {...renderProps}/>
      </Provider>
    );

    const initialState = store.getState()
    let fullHTML = renderFullPage(html, initialState);
    res.send(fullHTML);
  }
});
} catch (err) {
  next(err)
}
}

let renderFullPage = (html, initialState) =>
{
  return &grave;
  <!doctype html>
  <html>
```



```
<head>
<title>Publishing App Server Side Rendering</title>
</head>
<body>
<h1>Server side publishing app</h1>
<div id="publishingAppRoot">${html}</div>
<script>
window.__INITIAL_STATE__ = ${JSON.stringify(initialState)}
</script>
<script src="/static/app.js"></script>
</body>
</html>
&grave;
};

app.use(handleServerSideRender);

app.server.listen(process.env.PORT || 3000);
console.log(&grave;Started on port ${app.server.address().port}&grave;);

export default app;
```

Here you have everything you need for the server-side rendering on the backend. Let's move on to the frontend side improvements.

Frontend tweaks to make the server-side rendering work

We need some tweaks to the frontend. First of all, go to the file in `src/layouts/CoreLayout.js` and add the following:

```
import React from 'react';
import { Link } from 'react-router';

import themeDecorator from 'material-ui/lib/styles/theme-decorator';
import getMuiTheme from 'material-ui/lib/styles/getMuiTheme';

class CoreLayout extends React.Component {
  static propTypes = {
    children :React.PropTypes.element
  }
}
```

From the preceding code, the new thing to add is:

```
import themeDecorator from 'material-ui/lib/styles/theme-decorator';
import getMuiTheme from 'material-ui/lib/styles/getMuiTheme';
```

Besides this, improve the render function and export default to:

```
render () {
  return (
    <div>
      <span>
        Links:  <Link to='/register'>Register</Link> |
                <Link to='/login'>Login</Link> |
                <Link to='/'>Home Page</Link>
      </span>
      <br/>
      {this.props.children}
    </div>
  );
}

export default themeDecorator(getMuiTheme(null, { userAgent: 'all'
})) (CoreLayout);
```

We need the changes in the `CoreLayout` component because the Material UI design by default is checking on what browser you run it in and as you can predict, there is no browser on the server side so we need to provide the information in our app on whether { `userAgent: 'all' }` is set to `all`. It will help avoid warnings in the console about the server-side HTML markup being different from the one generated by the client-side browser.

We also need to improve our component `WillMount/_fetch` function in the publishing app's component, so it will be fired only on the frontend. Go to the `src/layouts/PublishingApp.js` file then replace this old code:

```
componentWillMount() {
  this._fetch();
}
```

Replace it with the new improved code:

```
componentWillMount() {
  if(typeof window !== 'undefined') {
    this._fetch(); // we are server side rendering, no fetching
  }
}
```

That `if(typeof window !== 'undefined')` statement checks if there is a window (on the server-side, the window will be undefined). If yes then it starts fetching data via Falcor (when on the client side).

Next, go to the `containers/Root.js` file and change it to the following:

```
import React from 'react';
import {Provider} from 'react-redux';
import {Router} from 'react-router';
import routes from '../routes';
import createHashHistory from 'history/lib/createHashHistory';

export default class Root extends React.Component {
  static propTypes = {
    history : React.PropTypes.object.isRequired,
    store    : React.PropTypes.object.isRequired
  }

  render () {
    return (
      <Provider store={this.props.store}>
      <div>
      <Router history={this.props.history}>
      {routes}
      </Router>
      </div>
      </Provider>
    );
  }
}
```

As you can see, we have deleted this part of the code:

```
// deleted code from Root.js
const noQueryKeyHistory = createHashHistory({
  queryKey: false
});
```

And we have changed this:

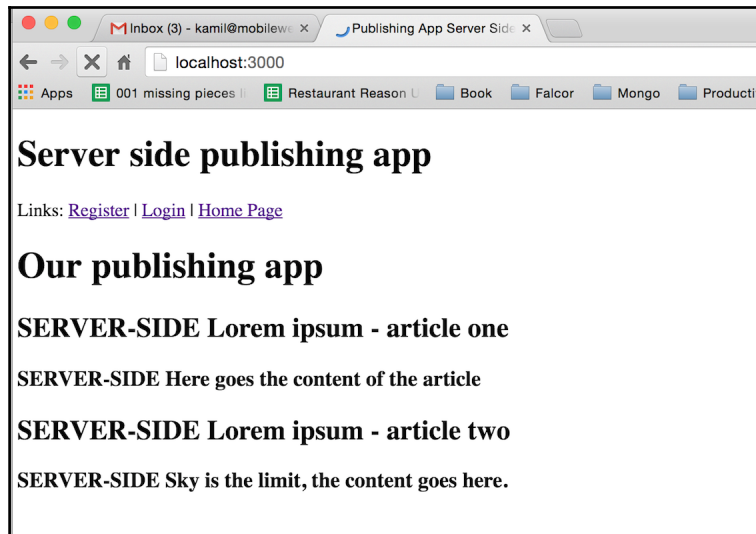
```
<Router history={noQueryKeyHistory}>
```

To this:

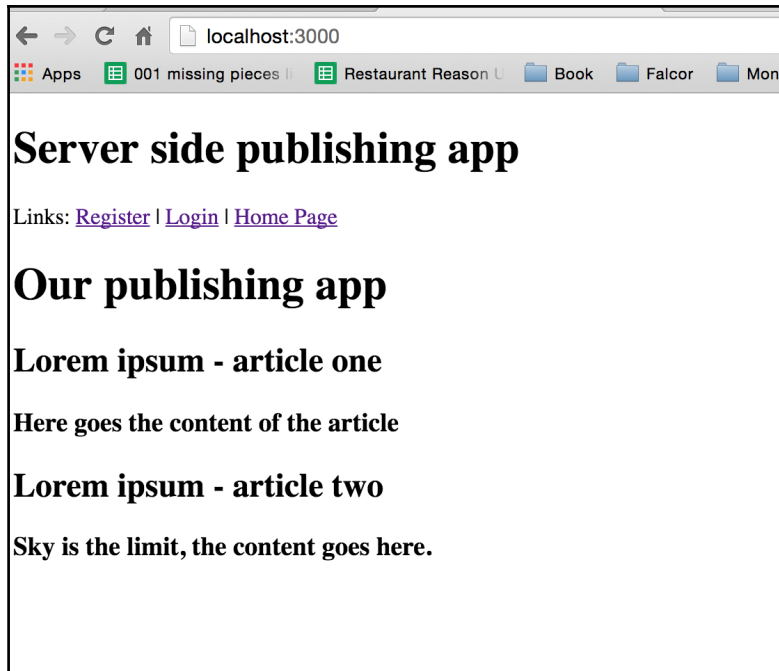
```
<Router history={this.props.history}>
```

Why do we need to do all this? It helps us to get rid of the `/#/` sign from our client-side browser's URL so next time when we hit, for example, `http://localhost:3000/register` then our `server.js` can see the user's current URL with the `req.path` (in our case when hitting the `http://localhost:3000/register` the `req.path` is then equal to `/register`) that we use in the `handleServerSideRender` function.

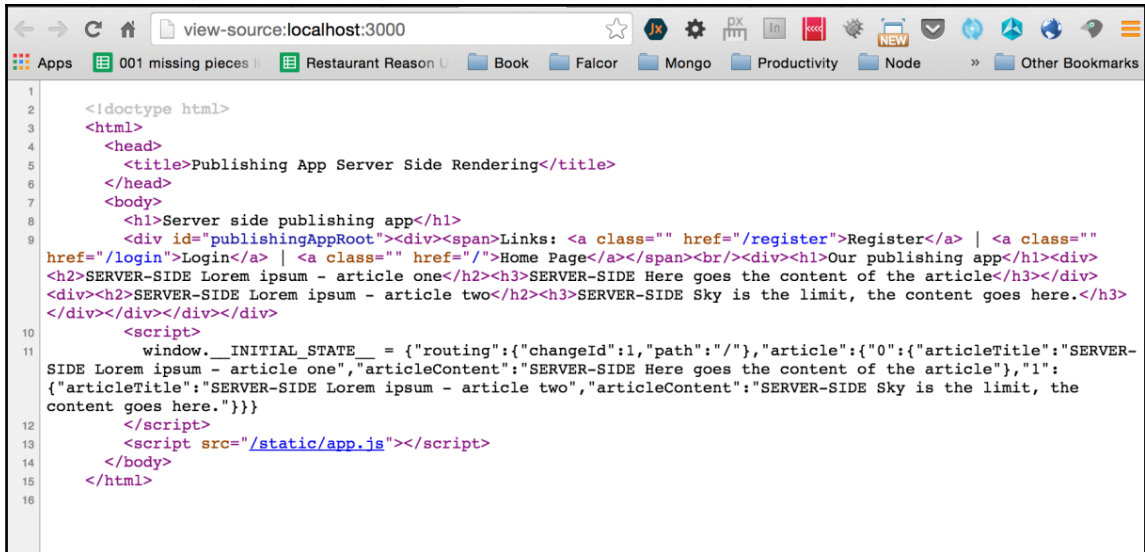
After all that, you will then be able to see the following in your client browser:



After 1-2 seconds it will change to the following because of firing the real `this._fetch()` function in the `PublishingApp.js`:



Of course, you can see the server-rendered markup when you go to the page's HTML source:



Summary

We have done the basic server-side rendering, as you can see in the screenshots. The only missing piece in the server-side rendering is to fetch real data from our MongoDB--that will be implemented in the next chapter (we will unlock this fetching in `server/fetchServerSide.js`).

After unmocking the server side's database query, we will start working on improving the whole look of our app and implement some key features that are important for us such as adding/editing/deleting an article.

4

Advanced Redux and Falcor on the Client Side

Redux is our app's state container which keeps the information about how the React view layer shall render in the browser. On the other hand, Falcor differs from Redux, because it is the full-stack toolset that replaces the outdated approach of API endpoints data communication. In the next pages, we will work with Falcor on the client side, but you need to remember that Falcor is a full-stack library. That means, we need to use it on both sides (where on backend we use an additional library called Falcor-Router). Starting from Chapter 5, *Falcor Advanced Concepts*, we will work with full-stack Falcor. While in the current chapter, we will focus only on the client side.

Focusing on the app's frontend

Currently, our app is a simple starter kit, which is a skeleton for its further development. We need to focus more on the customer-facing frontend because it's important to have a good-looking frontend in the current age. Thanks to Material UI, we can reuse many things to make our app look prettier.

It's important to note that responsive web design is not in the scope of this book at this point (and overall), so you need to find out how all the styles can be improved for mobile. The app we are going to work on will look fine on tablets, but small mobile screens may not look so good.

In this chapter, we will focus our efforts on the following:

- Unmocking `fetchServerSide.js`
- Adding a new `ArticleCard` component, which will make our main page more professional for our users
- Improving the general look of our application
- Implementing the ability to logout
- Adding a WYSIWYG editor in `Draft.js` which is a rich text-editor framework for React created by the Facebook team
- Adding the ability to create new articles in our Redux frontend application

Backend wrap-up before frontend improvement

In the previous chapter, we performed a server-side rendering that will affect our users such that they will see their articles quicker and will improve our website's SEO as the whole HTML markup is being rendered on the server side.

The last thing to make our server-side rendering work 100% is to unmock the server-side article fetching in `/server/fetchServerSide.js`. The new code for fetching is as follows:

```
import configMongoose from './configMongoose';
const Article = configMongoose.Article;

export default () => {
  return Article.find({}, function(err, articlesDocs) {
    return articlesDocs;
  }).then ((articlesArrayFromDB) => {
    return articlesArrayFromDB;
  });
}
```

As you can find in the preceding code snippet, this function returns a promise with `Article.find` (the `find` function comes from Mongoose). You can also find that we are returning an array of articles that are fetched from our MongoDB.

Improving `handleServerSideRender`

The next step is to tweak the `handleServerSideRender` function, which is currently kept in the `/server/server.js` file. The current function is as shown in the following code snippet:

```
// the following code should already be in your codebase:
let handleServerSideRender = (req, res, next) => {
  try {
    let initMOCKstore = fetchServerSide(); // mocked for now

    // Create a new Redux store instance
    const store = createStore(rootReducer, initMOCKstore)
    const location = hist.createLocation(req.path);
```

We need to replace it with this improved one:

```
// this is an improved version:
let handleServerSideRender = async (req, res, next) => {
  try {
    let articlesArray = await fetchServerSide();
    let initMOCKstore = {
      article: articlesArray
    }

    // Create a new Redux store instance
    const store = createStore(rootReducer, initMOCKstore)
    const location = hist.createLocation(req.path);
```

What is new in our improved `handleServerSideRender`? As you can see, we have added `async await`. Recall that it is helping us make our code less painful with asynchronous calls such as queries to the database (synchronous-looking generator-style code). This ES7 feature helps us write asynchronous calls as if they're synchronous ones--under the hood, `async await` is much more complicated (after it's transpiled into ES5 so that it can be run in any modern browser), but we won't get into details of how `async await` works because it's not in the scope of this chapter.

Changing routes in Falcor (frontend and backend)

You also need to change the two ID variable names to `_id` (`_id` is a default name for the ID of a document in a Mongo collection).

Look in `server/routes.js` for this old code:

```
route: 'articles[{integers}]["id","articleTitle","articleContent"]',
```

Change it into the following:

```
route: 'articles[{integers}]["_id","articleTitle","articleContent"]',
```

The only change is that we will return `_id` instead of `id`. We need to fetch the `_id` value in `src/layouts/PublishingApp.js`, so find the following code snippet:

```
get(['articles', {from: 0, to: articlesLength-1}, ['id','articleTitle',  
  'articleContent']]).
```

Change it into the new one with `_id`:

```
get(['articles', {from: 0, to: articlesLength-1}, ['_id','articleTitle',  
  'articleContent']]).
```

Our website header and articles list need improvements

Since we've finished wrapping up the server-side rendering and fetching articles from the DB, let's start with the frontend.

First, delete the following header from `server/server.js`; we don't need it anymore:

```
<h1>Server side publishing app</h1>
```

You can also delete this header in `src/layouts/PublishingApp.js`:

```
<h1>Our publishing app</h1>
```

Delete the `h1` markup in the registration and login view (`src/LoginView.js`):

```
<h1>Login view</h1>
```

Delete registration in `src/RegisterView.js`:

```
<h1>Register</h1>
```

All these `h1` lines are not needed as we want to have a nice-looking design instead of an outdated one.

After this, go to `src/CoreLayout.js` and import a new `AppBar` component and two button components from the Material UI:

```
import AppBar from 'material-ui/lib/app-bar';
import RaisedButton from 'material-ui/lib/raised-button';
import ActionHome from 'material-ui/lib/svg-icons/action/home';
```

Add this `AppBar` together with inline styles into `render`:

```
render () {
  const buttonStyle = {
    margin: 5
  };
  const homeIconStyle = {
    margin: 5,
    paddingTop: 5
  };

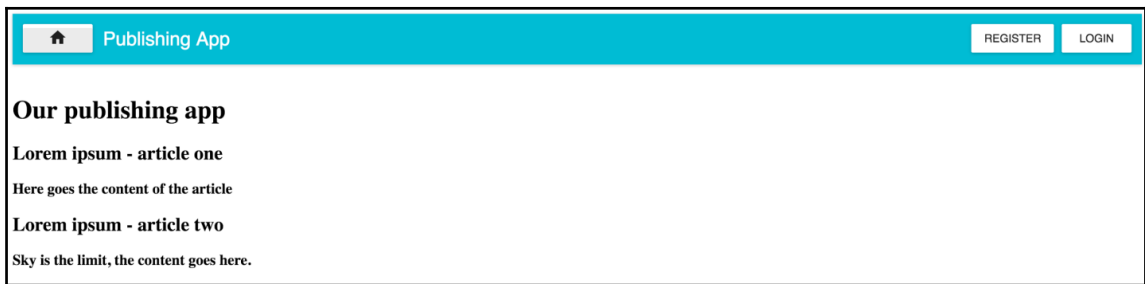
  let menuLinksJSX = (
    <span>
      <Link to='/register'>
        <RaisedButton label='Register' style={buttonStyle} />
      </Link>
      <Link to='/login'>
        <RaisedButton label='Login' style={buttonStyle} />
      </Link>
    </span>);

  let homePageButtonJSX = (
    <Link to='/'>
      <RaisedButton label={<ActionHome />}
        style={homeIconStyle} />
    </Link>);

  return (
    <div>
      <AppBar
        title='Publishing App'
        iconElementLeft={homePageButtonJSX}
        iconElementRight={menuLinksJSX} />
```

```
        <br/>
        {this.props.children}
      </div>
    );
  }
}
```

We have added the inline styles for `buttonStyle` and `homeIconStyle`. The `menuLinksJSX` and `homePageButtonJSX`'s visual output will improve. This is how your app will be looking after those `AppBar` changes:



New ArticleCard component

The next step in order to improve the look of our home page is to make article cards based on the Material Design CSS as well. Let's create a component's file first:

```
$ [[you are in the src/components/ directory of your project]]
$ touch ArticleCard.js
```

Then, in the `ArticleCard.js` file, let's initialize the `ArticleCard` component with the following content:

```
import React from 'react';
import {
  Card,
  CardHeader,
  CardMedia,
  CardTitle,
  CardText
} from 'material-ui/lib/card';
import {Paper} from 'material-ui';

class ArticleCard extends React.Component {
  constructor(props) {
    super(props);
  }
}
```

```
    }

    render() {
      return <h1>here goes the article card</h1>;
    }
  };
  export default ArticleCard;
```

As you can find in the preceding code, we have imported the required components from `material-ui/card` that will help our home page's articles list look nice. The next step is to improve our article card's `render` function with the following:

```
render() {
  let title = this.props.title || 'no title provided';
  let content = this.props.content || 'no content provided';

  const paperStyle = {
    padding: 10,
    width: '100%',
    height: 300
  };

  const leftDivStyle = {
    width: '30%',
    float: 'left'
  };

  const rightDivStyle = {
    width: '60%',
    float: 'left',
    padding: '10px 10px 10px 10px'
  };

  return (
    <Paper style={paperStyle}>
      <CardHeader
        title={this.props.title}
        subtitle='Subtitle'
        avatar='/static/avatar.png'
      />

      <div style={leftDivStyle}>
        <Card >
          <CardMedia
            overlay={<CardTitle title={title}
              subtitle='Overlay subtitle' />}>
            <img src='/static/placeholder.png' height="190" />
          </CardMedia>
```

```
      </Card>
    </div>
    <div style={rightDivStyle}>
      {content}
    </div>
  </Paper>);
}
```

As you can find in the preceding code, we have created an article card, and there are some inline styles for the `Paper` component and left and right `div`. Feel free to change the styles if you want.

In general, we are missing two static images in the previous render function, which are `src= '/static/placeholder.png'` and `avatar='/static/avatar.png'`. Let's add them using the following steps:

1. Make a PNG file with the name `placeholder.png` in the `dist` directory. In my case, this is what my `placeholder.png` file looks like:



2. Also create an `avatar.png` file in the `dist` directory that will be exposed in `/static/avatar.png`. I am not providing the screenshot here, as it has my personal photo in it.

The `/static/` file in `express.js` is exposed in the `/server/server.js` file with `codeapp.use('/static', express.static('dist'))`; (you will already have it in there as we have added this in the previous chapter).

The last thing is that you need to import `ArticleCard` and modify the render of `layouts/PublishingApp.js` from the old simple view to the new one.

Add import to the top of the file:

```
import ArticleCard from '../components/ArticleCard';
```

Then, replace the render with this new one:

```
render () {

  let articlesJSX = [];
  for(let articleKey in this.props.article) {
    const articleDetails = this.props.article[articleKey];

    const currentArticleJSX = (
      <div key={articleKey}>
        <ArticleCard
          title={articleDetails.articleTitle}
          content={articleDetails.articleContent} />
      </div>
    );

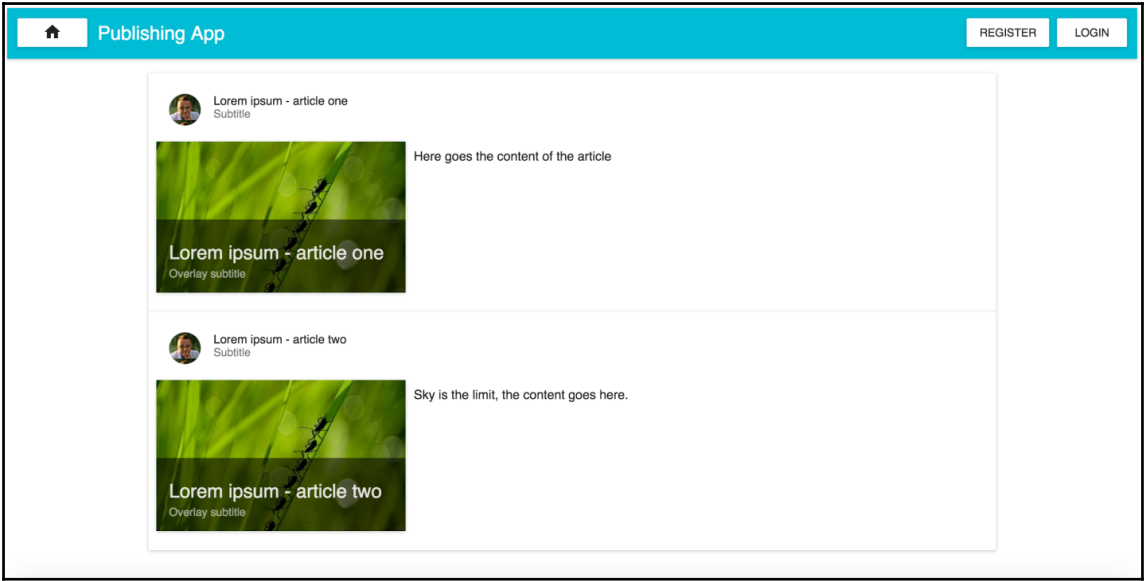
    articlesJSX.push(currentArticleJSX);
  }
  return (
    <div style={{height: '100%', width: '75%', margin: 'auto'}}>
      {articlesJSX}
    </div>
  );
}
```

The preceding new code only differs in this new `ArticleCard` component:

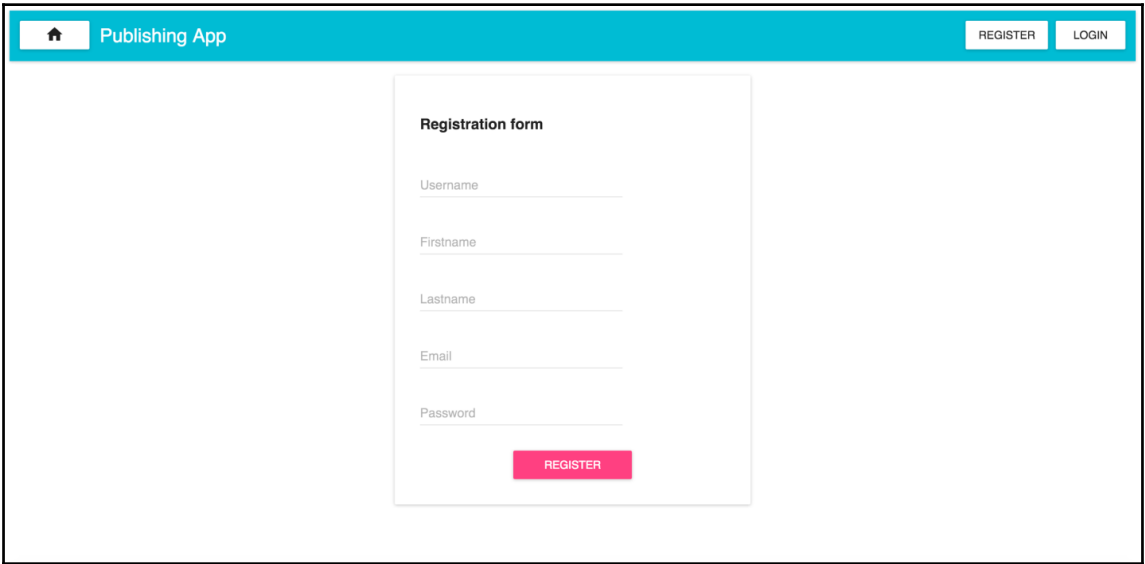
```
<ArticleCard
  title={articleDetails.articleTitle}
  content={articleDetails.articleContent} />
```

We also have added some styles to `div style={{height: '100%', width: '75%', margin: 'auto'}}.`

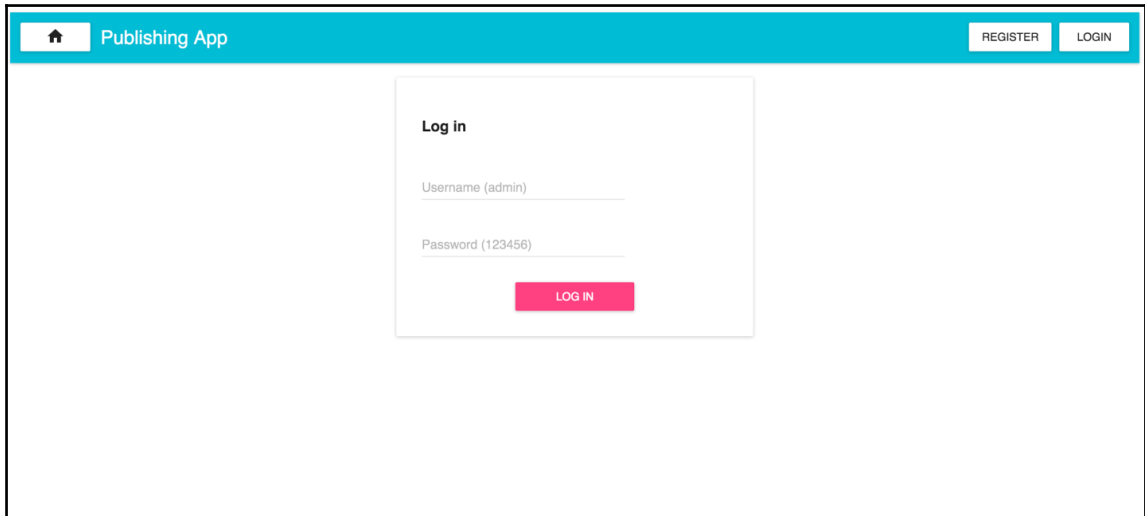
On following all these steps exactly in terms of styles, this is what you will see:



This is the register user view:



This is the login user view:



The mockup shows a web application interface. At the top is a teal header bar. On the left of the header is a home icon and the text 'Publishing App'. On the right are two buttons: 'REGISTER' and 'LOGIN'. The main content area is white and contains a centered 'Log in' form. The form has two input fields: 'Username (admin)' and 'Password (123456)'. Below the password field is a red 'LOG IN' button.

Dashboard - adding an article button, logout, and header improvements

Our plan for now is to create a logout mechanism, make our header aware whether a user is logged in or not, and based on that information show different buttons in the header (**Login/Register** when a user is not logged in and **Dashboard/Logout** when a user is logged in) We will create an **Add Article** button in our dashboard and create a mocked view with a mocked WYSIWYG (we will unmock it later).



WYSIWYG stands for **what you see is what you get**, of course.

The WYSIWYG mockup will be located in `src/components/articles/WYSIWYGeditor.js`, so you need to create a new directory and file in components with the following commands:

```
$ [[you are in the src/components/ directory of your project]]
$ mkdir articles
$ cd articles
$ touch WYSIWYGeditor.js
```

Then our `WYSIWYGeditor.js` mock content will be as follows:

```
import React from 'react';

class WYSIWYGeditor extends React.Component {
  constructor(props) {
    super(props);
  }

  render() {
    return <h1>WYSIWYGeditor</h1>;
  }
};
export default WYSIWYGeditor;
```

The next step is to create a logout view at `src/views/LogoutView.js`:

```
$ [[you should be at src/views/ directory of your project]]
$ touch LogoutView.js
```

The `src/views/LogoutView.js` file's content is as follows:

```
import React from 'react';
import {Paper} from 'material-ui';

class LogoutView extends React.Component {
  constructor(props) {
    super(props);
  }

  componentWillMount() {
    if (typeof localStorage !== 'undefined' && localStorage.token) {
      delete localStorage.token;
      delete localStorage.username;
      delete localStorage.role;
    }
  }

  render () {
```

```
    return (
      <div style={{width: 400, margin: 'auto'}}>
        <Paper zDepth={3} style={{padding: 32, margin: 32}}>
          Logout successful.
        </Paper>
      </div>
    );
  }
}
export default LogoutView;
```

The `logout` view mentioned here is a simple view without a connecting function to Redux (in comparison with `LoginView.js`). We are using some styling to make it nice, with the `Paper` component from Material UI.

The `componentWillMount` function is deleted from the `localStorage` information when the user lands on the logout page. As you can see, it also checks whether there is `localStorage` with `**if(typeof localStorage !== 'undefined' && localStorage.token) **` because, as you can imagine, when you perform server-side rendering, `localStorage` is undefined (the server side doesn't have `localStorage` and window like the client side).

Important note before creating a frontend add article feature

We've come to the point where you need to delete all documents from your articles collection, or you may have some trouble performing the next steps as we are going to use a `draft-js` library and some other stuff that will need a new schema on the backend. We will create that backend's schema in the next chapter as this chapter is focused on the frontend.

Delete all documents in your MongoDB articles collection right now, but keep the user collection as it was (don't delete users from the database).

The AddArticleView component

After creating the LogoutView and the WYSIWYGeditor components, let's create the final missing component in our process: the `src/views/articles/AddArticleView.js` file. So let's create a directory and file now:

```
$ [[you are in the src/views/ directory of your project]]
$ mkdir articles
$ cd articles
$ touch AddArticleView.js
```

As a result, you'll have that file in your `views/articles` directory. We need to put content into it:

```
import React from 'react';
import {connect} from 'react-redux';
import WYSIWYGeditor from '../../components/articles/WYSIWYGeditor.js';

const mapStateToProps = (state) => ({
  ...state
});

const mapDispatchToProps = (dispatch) => ({

});

class AddArticleView extends React.Component {
  constructor(props) {
    super(props);
  }

  render () {
    return (
      <div style={{height: '100%', width: '75%', margin: 'auto'}}>
        <h1>Add Article</h1>
        <WYSIWYGeditor />
      </div>
    );
  }
}

export default connect(mapStateToProps,
mapDispatchToProps)(AddArticleView);
```

As you can see here, it's a simple React view, and it imports the WYSIWYGeditor component that we created a moment ago (`import WYSIWYGeditor from '../../components/articles/WYSIWYGeditor.js'`). We have some inline styles in order to make the view look nicer for our user.

Let's create two new routes for a logout and for an add article feature by modifying the routes file at the `**src/routes/index.js` location:

```
import React from 'react';
import {Route, IndexRoute} from 'react-router';
import CoreLayout from '../layouts/CoreLayout';
import PublishingApp from '../layouts/PublishingApp';
import LoginView from '../views/LoginView';
import LogoutView from '../views/LogoutView';
import RegisterView from '../views/RegisterView';
import DashboardView from '../views/DashboardView';
import AddArticleView from '../views/articles/AddArticleView';

export default (
  <Route component={CoreLayout} path='/'>
    <IndexRoute component={PublishingApp} name='home' />
    <Route component={LoginView} path='login' name='login' />
    <Route component={LogoutView} path='logout' name='logout' />
    <Route component={RegisterView} path='register'
      name='register' />
    <Route component={DashboardView} path='dashboard'
      name='dashboard' />
    <Route component={AddArticleView} path='add-article'
      name='add-article' />
  </Route>
);
```

As explained in our `src/routes/index.js` file, we have added two routes:

- `<Route component={LogoutView} path='logout' name='logout' />`
- `<Route component={AddArticleView} path='add-article' name='add-article' />`

Don't forget to import those two views' components with the following:

```
import LogoutView from '../views/LogoutView';
import AddArticleView from '../views/articles/AddArticleView';
```

Now, we have created the views and created routes into that view. The last piece is to show links into those two routes in our app.

First let's create the `src/layouts/CoreLayout.js` component so it will have a login/logout-type login so that a logged-in user will see different buttons than a user who isn't. Modify the render function in the `CoreLayout` component to this:

```
render () {
  const buttonStyle = {
    margin: 5
  };
  const homeIconStyle = {
    margin: 5,
    padding: 5
  };

  let menuLinksJSX;
  let userIsLoggedIn = typeof localStorage !== 'undefined' &&
    localStorage.token && this.props.routes[1].name !== 'logout';

  if (userIsLoggedIn) {
    menuLinksJSX = (
      <span>
        <Link to='/dashboard'>
          <RaisedButton label='Dashboard' style={buttonStyle} />
        </Link>
        <Link to='/logout'>
          <RaisedButton label='Logout' style={buttonStyle} />
        </Link>
      </span>);
  } else {
    menuLinksJSX = (
      <span>
        <Link to='/register'>
          <RaisedButton label='Register' style={buttonStyle} />
        </Link>
        <Link to='/login'>
          <RaisedButton label='Login' style={buttonStyle} />
        </Link>
      </span>);
  }

  let homePageButtonJSX = (
    <Link to='/'>
      <RaisedButton label={<ActionHome />} style={homeIconStyle} />
    </Link>);

  return (
    <div>
```

```
      <AppBar
        title='Publishing App'
        iconElementLeft={homePageButtonJSX}
        iconElementRight={menuLinksJSX} />
      <br/>
      {this.props.children}
    </div>
  );
}
```

You can see that the new part in the preceding code is as follows:

```
let menuLinksJSX;
let userIsLoggedIn = typeof localStorage !==
'undefined' && localStorage.token && this.props.routes[1].name
!== 'logout';

if (userIsLoggedIn) {
  menuLinksJSX = (
    <span>
      <Link to='/dashboard'>
        <RaisedButton label='Dashboard' style={buttonStyle} />
      </Link>
      <Link to='/logout'>
        <RaisedButton label='Logout' style={buttonStyle} />
      </Link>
    </span>);
} else {
  menuLinksJSX = (
    <span>
      <Link to='/register'>
        <RaisedButton label='Register' style={buttonStyle} />
      </Link>
      <Link to='/login'>
        <RaisedButton label='Login' style={buttonStyle} />
      </Link>
    </span>);
}
```

We have added `let userIsLoggedIn = typeof localStorage !== 'undefined' && localStorage.token && this.props.routes[1].name !== 'logout';`. The `userIsLoggedIn` variable is found if we are not on the server side (then it doesn't have `localStorage` as mentioned earlier). Then, it checks whether `localStorage.token` is yes, and also checks whether a user didn't click on the logout button with the `this.props.routes[1].name !== 'logout'` expression. The `this.props.routes[1].name` value/information is provided by the `redux-simple-router` and `react-router`. This is always the name of our current route on the client side, so we can render the proper buttons based on that information.

Modifying DashboardView

As you will find, we have added the `if (userIsLoggedIn)` statement, and the new part is the dashboard and logout `RaisedButton` entities with links to the correct routes.

The last piece to finish at this stage is to modify the `src/views/DashboardView.js` component. Add link to the `/add-article` route using the `{Link}` component imported from `react-router`. Also, we need to import new Material UI components in order to make `DashboarView` nicer:

```
import {Link} from 'react-router';
import List from 'material-ui/lib/lists/list';
import ListItem from 'material-ui/lib/lists/list-item';
import Avatar from 'material-ui/lib/avatar';
import ActionInfo from 'material-ui/lib/svg-icons/action/info';
import FileFolder from 'material-ui/lib/svg-icons/file/folder';
import RaisedButton from 'material-ui/lib/raised-button';
import Divider from 'material-ui/lib/divider';
```

After you have imported all this in your `src/views/DashboardView.js` file, then we need to start work on improving the render function:

```
render () {

  let articlesJSX = [];
  for(let articleKey in this.props.article) {
    const articleDetails = this.props.article[articleKey];
    const currentArticleJSX = (
      <ListItem
        key={articleKey}
        leftAvatar={<img
          src='/static/placeholder.png'
          width='50'
```



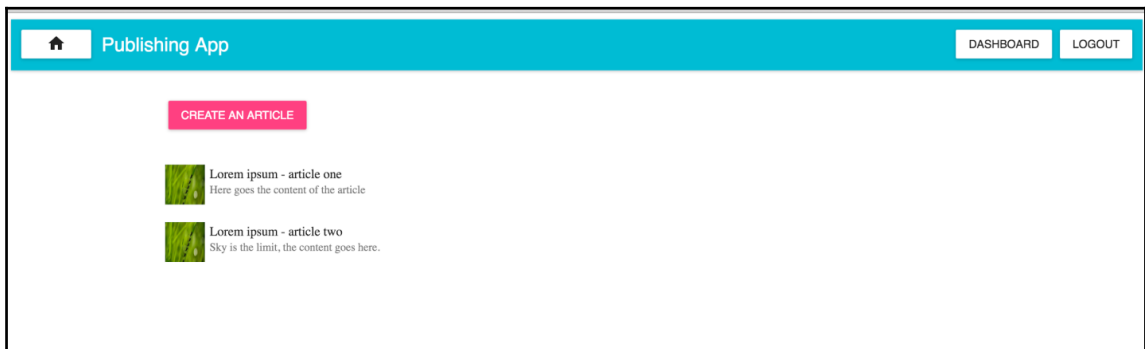
```
        height='50' />}
        primaryText={articleDetails.articleTitle}
        secondaryText={articleDetails.articleContent}
      />
    );

    articlesJSX.push(currentArticleJSX);
  }
  return (
    <div style={{height: '100%', width: '75%', margin: 'auto'}}>
      <Link to='/add-article'>
        <RaisedButton
          label='Create an article'
          secondary={true}
          style={{margin: '20px 20px 20px 20px'}} />
      </Link>

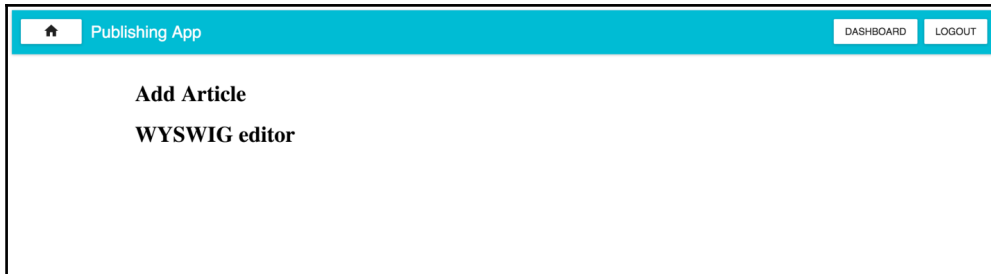
      <List>
        {articlesJSX}
      </List>
    </div>
  );
}
```

Here, we have our new render function for `DashboardView`. We are using the `List` component to make our nice lists. We have also added the link and button to the `/add-article` routes. There are some inline styles, but feel free to style this app on your own.

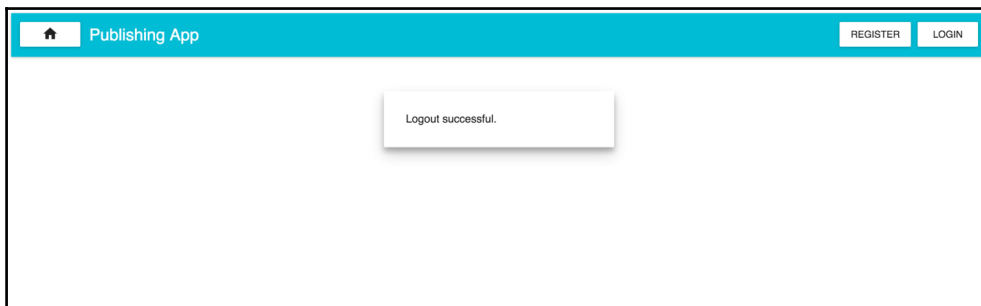
Let's look at a few screenshots of how the app looks after all these changes after adding the **CREATE AN ARTICLE** button with a new view of articles:



After having mocked WYSIWYG on the `/add-article` view:



Our new logout view page will look like this:



Starting work on our WYSIWYG

Let's install a `draft-js` library, which is "a framework for building rich text editors in React, powered by an immutable model and abstracting over cross-browser differences," as stated on their website.

In general, `draft-js` is made by friends from Facebook, and it helps us make powerful WYSIWYG tools. It will be useful in our publishing app as we want to provide good tools for our editors in order to create interesting articles on our platform.

Let's install it first:

```
npm i --save draft-js@0.5.0
```

We will use version 0.5.0 of draft-js in our book. Before we start coding, let's install one more dependency that will be helpful later in fetching the articles from the DB via Falcor. Execute the following command:

```
npm i --save falcor-json-graph@1.1.7
```

In general, the `falcor-json-graph@1.1.7` syntax provides us with the ability to use different sentinels provided via the Falcor helper library (which will be described in detail in the next chapter).

Stylesheet for the draft-js WYSIWYG

In order to style draft-js editor, we need to create a new CSS file in the `dist` folder located at `dist/styles-draft-js.css`. It's the only place where we will put a CSS stylesheet:

```
.RichEditor-root {
  background: #fff;
  border: 1px solid #ddd;
  font-family: 'Georgia', serif;
  font-size: 14px;
  padding: 15px;
}

.RichEditor-editor {
  border-top: 1px solid #ddd;
  cursor: text;
  font-size: 16px;
  margin-top: 10px;
  min-height: 100px;
}

.RichEditor-editor .RichEditor-blockquote {
  border-left: 5px solid #eee;
  color: #666;
  font-family: 'Hoefler Text', 'Georgia', serif;
  font-style: italic;
  margin: 16px 0;
  padding: 10px 20px;
}

.RichEditor-controls {
  font-family: 'Helvetica', sans-serif;
  font-size: 14px;
  margin-bottom: 5px;
  user-select: none;
}
```

```
}

.RichEditor-styleButton {
  color: #999;
  cursor: pointer;
  margin-right: 16px;
  padding: 2px 0;
}

.RichEditor-activeButton {
  color: #5890ff;
}
```

After you have created this file at `dist/styles-draft-js.css`, we need to import it to `server/server.js`, where we have been creating the HTML header, so the following code is already present in the `server.js` file:

```
let renderFullPage = (html, initialState) =>
{
  return &grave;
    <!doctype html>
    <html>
      <head>
        <title>Publishing App Server Side Rendering</title>
        <link rel="stylesheet" type="text/css"
          href="/static/styles-draft-js.css" />
      </head>
      <body>
        <div id="publishingAppRoot">${html}</div>
        <script>
          window.__INITIAL_STATE__ =
            ${JSON.stringify(initialState)}
        </script>
        <script src="/static/app.js"></script>
      </body>
    </html>
    &grave;
};
```

Then you need to include the link to the stylesheet with this:

```
<link rel="stylesheet" type="text/css" href="/static/styles-draft-
js.css" />
```

Nothing fancy so far. After we are done with the styles for our rich text WYSIWYG editor, let's have some fun.

Coding a draft-js skeleton

Let's get back to the `src/components/articles/WYSIWYGeditor.js` file. It's currently mocked, but we will improve it now.

Just to make you aware, we will make a skeleton of the WYSIWYG right now. We will improve it later in the book. At this point, the WYSIWYG won't have any functionalities such as making text bold or creating lists with OL and UL elements.

```
import React from 'react';
import {
  Editor,
  EditorState,
  ContentState,
  RichUtils,
  convertToRaw,
  convertFromRaw
} from 'draft-js';

export default class WYSIWYGeditor extends React.Component {
  constructor(props) {
    super(props);

    let initialEditorFromProps =
      EditorState.createWithContent(
        ContentState.createFromText(''));

    this.state = {
      editorState: initialEditorFromProps
    };

    this.onChange = (editorState) => {
      var contentState = editorState.getCurrentContent();

      let contentJSON = convertToRaw(contentState);
      props.onChangeTextJSON(contentJSON, contentState);
      this.setState({editorState})
    };
  }

  render() {
    return <h1>WYSIWYGeditor</h1>;
  }
}
```

Here, we have created only a constructor of our new draft-js file's WYSIWYG. The `let initialEditorFromProps = EditorState.createWithContent(ContentState.createFromText(''));` expression is simply creating an empty WYSIWYG container. Later, we will improve it so we are able to receive `ContentState` from the database when we would like to edit our WYSIWYG.

The `editorState: initialEditorFromProps` is our current state. Our `**this.onChange = (editorState) => {` `**` line is firing on each change, so our view component at `src/views/articles/AddArticleView.js` will know about any changes in the WYSIWYG.

Anyway, you can check the documentation of draft-js at <https://facebook.github.io/draft-js/>.

This is just the beginning; the next step is to add two new functions under `onChange`:

```
this.focus = () => this.refs['refWYSIWYGeditor'].focus();
this.handleKeyCommand = (command) => this._handleKeyCommand(command);
```

And add a new function in our `WYSIWYGeditor` class:

```
_handleKeyCommand(command) {
  const {editorState} = this.state;
  const newState = RichUtils.handleKeyCommand(editorState,
    command);

  if (newState) {
    this.onChange(newState);
    return true;
  }
  return false;
}
```

After all these changes, this is how your construction of the `WYSIWYGeditor` class should look:

```
export default class WYSIWYGeditor extends React.Component {
  constructor(props) {
    super(props);

    let initialEditorFromProps =
      EditorState.createWithContent
        (ContentState.createFromText(''));

    this.state = {
```

```
    editorState: initialEditorFromProps
  };

  this.onChange = (editorState) => {
    var contentState = editorState.getCurrentContent();

    let contentJSON = convertToRaw(contentState);
    props.onChangeTextJSON(contentJSON, contentState);
    this.setState({editorState});
  };

  this.focus = () => this.refs['refWYSIWYGeditor'].focus();
  this.handleKeyCommand = (command) =>
    this._handleKeyCommand(command);
}
```

And the rest of this class is as follows:

```
_handleKeyCommand(command) {
  const {editorState} = this.state;
  const newState = RichUtils.handleKeyCommand(editorState,
    command);

  if (newState) {
    this.onChange(newState);
    return true;
  }
  return false;
}

render() {
  return <h1> WYSIWYGeditor</h1>;
}
}
```

The next step is to improve the render function with the following code:

```
render() {
  const { editorState } = this.state;
  let className = 'RichEditor-editor';
  var contentState = editorState.getCurrentContent();

  return (
    <div>
      <h4>{this.props.title}</h4>
      <div className='RichEditor-root'>
        <div className={className} onClick={this.focus}>
          <Editor
```

```
        editorState={editorState}
        handleKeyCommand={this.handleKeyCommand}
        onChange={this.onChange}
        ref='refWYSIWYGeditor' />
      </div>
    </div>
  </div>
);
}
```

Here, what we have done is simply use the draft-js API in order to make a simple rich editor; later, we will make it more functional, but for now, let's focus on something simple.

Improving the views/articles/AddArticleView component

Before we move forward with adding all the WYSIWYG features, such as bolding, we need to improve the `views/articles/AddArticleView.js` component with a few things. Install a library that will convert the draft-js state into plain HTML with the following:

```
npm i --save draft-js-export-html@0.1.13
```

We will use this library to save read-only plain HTML for our regular readers. Next, import this into `src/views/articles/AddArticleView.js`:

```
import { stateToHTML } from 'draft-js-export-html';
```

Improve `AddArticleView` by changing a constructor and adding a new function called `_onDraftJSChange`:

```
class AddArticleView extends React.Component {
  constructor(props) {
    super(props);
    this._onDraftJSChange = this._onDraftJSChange.bind(this);

    this.state = {
      contentJSON: {},
      htmlContent: ''
    };
  }

  _onDraftJSChange(contentJSON, contentState) {
    let htmlContent = stateToHTML(contentState);
    this.setState({contentJSON, htmlContent});
  }
}
```


We need to save on each change a state of `this.setState({contentJSON, htmlContent})`; . This is because `contentJSON` will be saved into the database in order to have immutable information about our WYSIWYG and `htmlContent` will be the server for our readers. Both `htmlContent` and `contentJSON` variables will be kept in the articles collection. The last thing in the `AddArticleView` class is to modify `render` to new code, as follows:

```
render () {
  return (
    <div style={{height: '100%', width: '75%', margin: 'auto'}}>
      <h1>Add Article</h1>
      <WYSIWYGeditor
        initialValue=''
        title='Create an article'
        onChangeTextJSON={this._onDraftJSChange} />
    </div>
  );
}
```

After all these changes, the new view that you will see is this:

Add Article

Create an article

This is a WYSIWIG
where you can put TEXT
but you can't format that text, YET

The only feature is a new line and paragraphs :-> ... all the abstractions are get away
from you and everything is handled by the Draft-JS' library

Adding more formatting features to our WYSIWYG

Let's start working on version two of our WYSIWYG, with more options, as in the following example:

Add Article

Create an article

H1 H2 Blockquote UL OL

Bold Italic Underline

HEADER 2

We can make lists

1. first item
2. second item
3. We can **make text BOLD**

UNDERLINE

After you follow the steps mentioned here, you will be able to format your text as follows and extract the HTML markup from it as well so that we can save both the JSON state of our WYSIWYG and plain HTML in our MongoDB articles collection.

In the following new file, called `WYSIWYGbuttons.js`, we will export two different classes, and we will import them into `components/articles/WYSIWYGeditor.js` using the following:

```
// don't write it, this is only an example:
import { BlockStyleControls, InlineStyleControls } from
  './wysiwyg/WYSIWYGbuttons';
```

In general, that new file will have three different React components, as follows:

- **StyleButton**: This will be a generic-style button that will be used in both `BlockStyleControls` and `InlineStyleControls`. Don't get confused by the fact that in the `WYSIWYGbuttons` file, you are creating the `StyleButton` React component first.

- **BlockStyleControls**: This is an exported component, which will be used for block controls such as H1, H2, Blockquote, UL, and OL.
- **InlineStyleControls**: This component is used for bold, italics, and underline.

Now we are aware that in the new file, you will create three separate React components.

First, we need to create the WYSIWIG buttons in the

`src/components/articles/wysiwyg/WYSIWYGbuttons.js` location:

```
$ [[you are in the src/components/articles directory of your project]]
$ mkdir wysiwyg
$ cd wysiwyg
$ touch WYSIWYGbuttons.js
```

The content of this file will be the buttons component:

```
import React from 'react';

class StyleButton extends React.Component {
  constructor() {
    super();
    this.onToggle = (e) => {
      e.preventDefault();
      this.props.onToggle(this.props.style);
    };
  }

  render() {
    let className = 'RichEditor-styleButton';
    if (this.props.active) {
      className += ' RichEditor-activeButton';
    }

    return (
      <span className={className} onMouseDown={this.onToggle}>
        {this.props.label}
      </span>
    );
  }
}
```

The preceding code is giving us a reusable button with a certain label at `this.props.label`. As mentioned before, don't get confused with `WYSIWYGbuttons`; it's a generic button component that will be reused in the inline and block type button controls.

Next, under that component, you can put the following object:

```
const BLOCK_TYPES = [
  {label: 'H1', style: 'header-one'},
  {label: 'H2', style: 'header-two'},
  {label: 'Blockquote', style: 'blockquote'},
  {label: 'UL', style: 'unordered-list-item'},
  {label: 'OL', style: 'ordered-list-item'}
];
```

This object is block type, which we can create in our draft-js WYSIWYG. It is used in the following component:

```
export const BlockStyleControls = (props) => {
  const {editorState} = props;
  const selection = editorState.getSelection();
  const blockType = editorState
    .getCurrentContent()
    .getBlockForKey(selection.getStartKey())
    .getType();

  return (
    <div className='RichEditor-controls'>
      {BLOCK_TYPES.map((type) =>
        <StyleButton
          key={type.label}
          active={type.style === blockType}
          label={type.label}
          onToggle={props.onToggle}
          style={type.style}
        />
      )}
    </div>
  );
};
```

The preceding code is a whole bunch of buttons for block-style formatting. We will import them into WYSIWYGeditor in a while. As you can see, we are exporting it with `export const BlockStyleControls = (props) => {`.

Put the next object under the `BlockStyleControls` component, but this time, for inline styles such as **Bold**:

```
var INLINE_STYLES = [
  {label: 'Bold', style: 'BOLD'},
  {label: 'Italic', style: 'ITALIC'},
  {label: 'Underline', style: 'UNDERLINE'}
];
```

As you can see, in our WYSIWYG, an editor will be able to use bold, italics, and underline.

The last component for those inline styles that you can put under all this is the following:

```
export const InlineStyleControls = (props) => {
  var currentStyle = props.editorState.getCurrentInlineStyle();
  return (
    <div className='RichEditor-controls'>
      {INLINE_STYLES.map(type =>
        <StyleButton
          key={type.label}
          active={currentStyle.has(type.style)}
          label={type.label}
          onToggle={props.onToggle}
          style={type.style}
        />
      )}
    </div>
  );
};
```

As you can see, this is very simple. We are mapping the defined styles in the blocks and inline styles each time, and based on each iteration, we are creating `StyleButton`.

The next step is to import both `InlineStyleControls` and `BlockStyleControls` in our WYSIWYGeditor component (`src/components/articles/WYSIWYGeditor.js`):

```
import { BlockStyleControls, InlineStyleControls } from
'../wysiwyg/WYSIWYGbuttons';
```

Then, in the WYSIWYGeditor constructor, include the following code:

```
this.toggleInlineStyle = (style) =>
this._toggleInlineStyle(style);
this.toggleBlockType = (type) => this._toggleBlockType(type);
```

Bind to `toggleInlineStyle` and `toggleBlockType` two arrow functions, which will be the callbacks when someone chooses the toggle in order to use inline or block type in our `WYSIWYGEditor` (we will create those functions in a moment).

Create these two new functions:

```
_toggleBlockType(blockType) {this.onChange(  
  RichUtils.toggleBlockType(  
    this.state.editorState,  
    blockType  
  )  
});  
}  
  
_toggleInlineStyle(inlineStyle) {  
  this.onChange(  
    RichUtils.toggleInlineStyle(  
      this.state.editorState,  
      inlineStyle  
    )  
  );  
}
```

Here, both functions are using the draft-js `RichUtils` in order to set flags inside our `WYSIWYG`. We are using certain formatting options from `BLOCK_TYPES` and `INLINE_STYLES` that we have defined in the `import { BlockStyleControls, InlineStyleControls } from './wysiwg/WYSIWGbuttons';`

After we are done improving our `WYSIWYGEditor` construction and the `_toggleBlockType` and `_toggleInlineStyle` functions, then we can start improving our render function:

```
render() {  
  const { editorState } = this.state;  
  let className = 'RichEditor-editor';  
  var contentState = editorState.getCurrentContent();  
  
  return (  
    <div>  
      <h4>{this.props.title}</h4>  
      <div className='RichEditor-root'>  
        <BlockStyleControls  
          editorState={editorState}  
          onToggle={this.toggleBlockType} />  
  
        <InlineStyleControls
```

```
        editorState={editorState}
        onToggle={this.toggleInlineStyle} />

    <div className={className} onClick={this.focus}>
      <Editor
        editorState={editorState}
        handleKeyCommand={this.handleKeyCommand}
        onChange={this.onChange}
        ref='refWYSIWYGeditor' />
    </div>
  </div>
</div>
);
}
```

As you may notice, in the preceding code, we have only added the `BlockStyleControls` and `InlineStyleControls` component. Also notice that we are using callbacks with `onToggle={this.toggleBlockType}` and `onToggle={this.toggleInlineStyle}`; this is for communicating between our `WYSIWYGbuttons` and the `draft-js RichUtils` about what a user has clicked on and in which mode they are currently using (such as bold, header1, and UL or OL).

Pushing a new article into article reducer

We need to create a new action called `pushNewArticle` in the `src/actions/article.js` location:

```
export default {
  articlesList: (response) => {
    return {
      type: 'ARTICLES_LIST_ADD',
      payload: { response: response }
    }
  },
  pushNewArticle: (response) => {
    return {
      type: 'PUSH_NEW_ARTICLE',
      payload: { response: response }
    }
  }
}
```

The next step is to improve the `src/components/ArticleCard.js` component by improving the render function in it:

```
return (
  <Paper style={paperStyle}>
    <CardHeader
      title={this.props.title}
      subtitle='Subtitle'
      avatar='/static/avatar.png'
    />

    <div style={leftDivStyle}>
      <Card >
        <CardMedia
          overlay={<CardTitle title={title} subtitle='Overlay'
            subtitle' />>
          <img src='/static/placeholder.png' height='190' />
        </CardMedia>
      </Card>
    </div>
    <div style={rightDivStyle}>
      <div dangerouslySetInnerHTML={{__html: content}} />
    </div>
  </Paper>);
}
```

Here, we have replaced the old `{content}` variable (which was receiving a plain text value in the content's variable) to a new one that shows all of the HTML using `dangerouslySetInnerHTML` in the article card:

```
<div dangerouslySetInnerHTML={{__html: content}} />
```

This will help us show our readers the HTML code generated by our WYSIWYG.

MapHelpers for improving our reducers

In general, all reducers *must* return a new reference to an object when it has changed. In our first example, we used `Object.assign`:

```
// this already exists in your codebase case 'ARTICLES_LIST_ADD':
let articlesList = action.payload.response;
return Object.assign({}, articlesList);
```


We will replace this `Object.assign` approach with a new one, with ES6's Maps:

```
case 'ARTICLES_LIST_ADD':
  let articlesList = action.payload.response;
  return mapHelpers.addMultipleItems(state, articlesList);
```

In the preceding code, you can find a new `ARTICLES_LIST_ADD` with `mapHelpers.addMultipleItems(state, articlesList)`.

In order to make our map helpers, we need to create a new directory called `utils` and a file called `mapHelpers.js` (`src/utils/mapHelpers.js`):

```
$ [[you are in the src/ directory of your project]]
$ mkdir utils
$ cd utils
$ touch mapHelpers.js
```

And then, you can enter this first function into that `src/utils/mapHelpers.js` file:

```
const duplicate = (map) => {
  const newMap = new Map();
  map.forEach((item, key) => {
    if (item['_id']) {
      newMap.set(item['_id'], item);
    }
  });
  return newMap;
};

const addMultipleItems = (map, items) => {
  const newMap = duplicate(map);

  Object.keys(items).map((itemIndex) => {
    let item = items[itemIndex];
    if (item['_id']) {
      newMap.set(item['_id'], item);
    }
  });

  return newMap;
};
```

The duplicate simply creates a new reference in memory in order to make our immutability a requirement in Redux applications. We also are checking, with `if (key === item['_id'])`, whether there is an edge case that the key is different from our object ID (the `_` in `_id` is intentional as this is how Mongoose marks the ID from our DB). The `addMultipleItems` function adds items to the new duplicated map (for example, after a successful fetch of articles).

The next code change that we require is in the same file at `src/utils/mapHelpers.js`:

```
const addItem = (map, newKey, newItem) => {
  const newMap = duplicate(map);
  newMap.set(newKey, newItem);
  return newMap;
};

const deleteItem = (map, key) => {
  const newMap = duplicate(map);
  newMap.delete(key);

  return newMap;
};

export default {
  addItem,
  deleteItem,
  addMultipleItems
};
```

As you can see, we have added an `add` function and `delete` function for a single item. After that, we are exporting all that from `src/utils/mapHelpers.js`.

The next step is that we need to improve the `src/reducers/article.js` reducer in order to use the map utilities in it:

```
import mapHelpers from '../utils/mapHelpers';

const article = (state = {}, action) => {
  switch (action.type) {
    case 'ARTICLES_LIST_ADD':
      let articlesList = action.payload.response;
      return mapHelpers.addMultipleItems(state, articlesList);
    case 'PUSH_NEW_ARTICLE':
      let newArticleObject = action.payload.response;
      return mapHelpers.addItem(state, newArticleObject['_id'],
        newArticleObject);
    default:
```

```
        return state;
    }
}
export default article
```

What's new in the `src/reducers/article.js` file? As you can see, we have improved `ARTICLES_LIST_ADD` (already discussed). We have added a new `PUSH_NEW_ARTICLE`; case; this will push a new object into our reducer's state tree. It's similar to pushing an item to an array, but we use our reducer and maps instead.

The CoreLayout improvements

Because we are switching to the ES6's Map in the frontend, we also need to make sure that after we receive an object with server-side rendering, it is also a Map (not a plain JS object). Check out the following code:

```
// The following is old codebase:
import React from 'react';
import { Link } from 'react-router';
import themeDecorator from 'material-ui/lib/styles/theme-
decorator';
import getMuiTheme from 'material-ui/lib/styles/getMuiTheme';
import RaisedButton from 'material-ui/lib/raised-button';
import AppBar from 'material-ui/lib/app-bar';
import ActionHome from 'material-ui/lib/svg-icons/action/home';
```

In the following new code snippet, you can find all the imports required in the CoreLayout component:

```
import React from 'react';
import {Link} from 'react-router';
import themeDecorator from 'material-ui/lib/styles/theme-
decorator';
import getMuiTheme from 'material-ui/lib/styles/getMuiTheme';
import RaisedButton from 'material-ui/lib/raised-button';
import AppBar from 'material-ui/lib/app-bar';
import ActionHome from 'material-ui/lib/svg-icons/action/home';
import {connect} from 'react-redux';
import {bindActionCreators} from 'redux';
import articleActions from '../actions/article.js';

const mapStateToProps = (state) => ({
  ...state
});

const mapDispatchToProps = (dispatch) => ({
```

```
    articleActions: bindActionCreators(articleActions, dispatch)
  });
```

Above the `CoreLayout` component, we have added the Redux tools, so we will have a state tree and the actions available in the `CoreLayout` component.

Also, in the `CoreLayout` component, add the `componentWillMount` function:

```
componentWillMount() {
  if (typeof window !== 'undefined' && !this.props.article.get)
  {
    this.props.articleActions.articlesList(this.props.article);
  }
}
```

This function is responsible for checking whether an article's properties are an ES6 Map or not. If not, then we send an action to `articlesList` that makes the job done, and after that, we have maps in `this.props.article`.

The last thing is to improve export in the `CoreLayout` component:

```
const muiCoreLayout = themeDecorator(getMuiTheme(null, {
  userAgent: 'all' }))(CoreLayout);
export default connect(mapStateToProps,
  mapDispatchToProps)(muiCoreLayout);
```

The preceding code helps us connect to the Redux single-state tree and the actions it allows.

Why Maps over a JS object?

In general, an ES6 Map has some features for easy data manipulation---functions such as `.get` and `.set` which make programming more pleasurable. It also helps to have a simpler code to be able to keep our immutability as required by Redux.

Map methods are much easier to use than `slice/c-oncat/Object.assign`. I am sure that there are always some cons/pros to each approach, but in our app, we will use an ES6 Map-wise approach in order to keep things simpler after we are completely set up with it.

Improving PublishingApp and DashboardView

In the `src/layouts/PublishingApp.js` file, we need to improve our render function:

```
render () {

  let articlesJSX = [];

  this.props.article.forEach((articleDetails, articleKey) => {
    const currentArticleJSX = (
      <div key={articleKey}>
        <ArticleCard
          title={articleDetails.articleTitle}
          content={articleDetails.articleContent} />
        </div>
      );

    articlesJSX.push(currentArticleJSX);
  });

  return (
    <div style={{height: '100%', width: '75%', margin: 'auto'}}>
      {articlesJSX}
    </div>
  );
}
```

As you can see in the preceding code, we switched the old `for(let articleKey in this.props.article) {` code into `this.props.article.forEach` because we have switched from objects to using Maps.

We need to do the same in the `src/views/DashboardView.js` file's render function:

```
render () {

  let articlesJSX = [];
  this.props.article.forEach((articleDetails, articleKey) => {
    const currentArticleJSX = (
      <ListItem
        key={articleKey}
        leftAvatar={<img src='/static/placeholder.png'
          width='50'
          height='50' />}
        primaryText={articleDetails.articleTitle}
        secondaryText={articleDetails.articleContent}
      />
    );
  });
}
```

```
    articlesJSX.push(currentArticleJSX);
  });

  return (
    <div style={{height: '100%', width: '75%', margin: 'auto'}}>
      <Link to='/add-article'>
        <RaisedButton
          label='Create an article'
          secondary={true}
          style={{margin: '20px 20px 20px 20px'}} />
      </Link>

      <List>
        {articlesJSX}
      </List>
    </div>
  );
}
```

For the same reason as in the `PublishingApp` component, we switched to using ES6's new `Map`, and we will be also using the new ES6 `forEach` method:

```
this.props.article.forEach((articleDetails, articleKey) => {
```

Tweaks to AddArticleView

After we are finished preparing our app to save a new article into the article's reducer, we need to tweak the `src/views/articles/AddArticleView.js` component. New imports in the `AddArticleView.js` are as follows:

```
import {bindActionCreators} from 'redux';
import {Link} from 'react-router';
import articleActions from '../../actions/article.js';
import RaisedButton from 'material-ui/lib/raised-button';
```

As you can see in the preceding code, we are importing `RaisedButton` and `Link`, which will be useful for redirecting an editor to the dashboard's view after a successful article addition. Then, we import `articleActions` because we need to make the `this.props.articleActions.pushNewArticle(newArticle);` action on article submit. The `bindActionCreators` will already be imported in your `AddArticleView` if you followed instructions from previous chapters.

Use `bindActionCreators` in order to have `articleActions` in the `AddArticleView` component by replacing this code snippet:

```
// this is old code, you shall have it already
const mapDispatchToProps = (dispatch) => ({
});
```

Here is the new `bindActionCreators` code:

```
const mapDispatchToProps = (dispatch) => ({
  articleActions: bindActionCreators(articleActions, dispatch)
});
```

The following is an updated constructor of the `AddArticleView` component:

```
constructor(props) {
  super(props);
  this._onDraftJSChange = this._onDraftJSChange.bind(this);
  this._articleSubmit = this._articleSubmit.bind(this);

  this.state = {
    title: 'test',
    contentJSON: {},
    htmlContent: '',
    newArticleID: null
  };
}
```

The `_articleSubmit` method will be required after an editor wants to add an article. We have also added some default states for our `title`, `contentJSON` (we will keep the draft-js article state there), `htmlContent`, and the `newArticleID`. The next step is to create the `_articleSubmit` function:

```
_articleSubmit() {
  let newArticle = {
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON
  }

  let newArticleID = 'MOCKEDRandomid' + Math.floor(Math.random()
    * 10000);

  newArticle['_id'] = newArticleID;
  this.props.articleActions.pushNewArticle(newArticle);
  this.setState({ newArticleID: newArticleID });
}
```

As you can see here, we get the state of our current writing with `this.state.title`, `this.state.htmlContent`, and `this.state.contentJSON`, and based on that, we then create a new `Article` model:

```
let newArticle = {
  articleTitle: this.state.title,
  articleContent: this.state.htmlContent,
  articleContentJSON: this.state.contentJSON
}
```

Then we mock the new article's ID (later, we will save it to the DB) with `newArticle['_id'] = newArticleID`; and push it into our article's reducer with `this.props.articleActions.pushNewArticle(newArticle)`; The only thing is to set up `newArticleID` with `this.setState({ newArticleID: newArticleID })`; The last step is to update our render method in the `AddArticleView` component:

```
render () {
  if (this.state.newArticleID) {
    return (
      <div style={{height: '100%', width: '75%', margin:
        'auto'}}>
        <h3>Your new article ID is
          {this.state.newArticleID}</h3>
        <Link to='/dashboard'>
          <RaisedButton
            secondary={true}
            type='submit'
            style={{margin: '10px auto', display: 'block',
              width: 150}}
            label='Done' />
        </Link>
      </div>
    );
  }

  return (
    <div style={{height: '100%', width: '75%', margin: 'auto'}}>
      <h1>Add Article</h1>
      <WYSIWYGeditor
        name='addarticle'

        onChangeTextJSON={this._onDraftJSChange} />
      <RaisedButton
        onClick={this._articleSubmit}
        secondary={true}
        type='submit'
        style={{margin: '10px auto', display: 'block', width:
```



```
        150}}
        label={'Submit Article'} />
    </div>
  );
}
```

Here in the `render` method, we have one statement that checks whether an article's editor has already created an article (clicked on the **Submit Article** button) with `if(this.state.newArticleID)`. If yes, then the editor will see his new article's ID and a button that links to the dashboard (link is `to='/dashboard'`).

The second return is in case an editor is in edit mode; if yes, then he can submit it by clicking on the `RaisedButton` component with the `onClick` method's called `_articleSubmit`.

The ability to edit an article (the `EditArticleView` component)

We can add an article, but we can't edit it yet. Let's implement that feature.

The first thing to do is to create a route in `src/routes/index.js`:

```
import EditArticleView from '../views/articles/EditArticleView';
```

Then edit the routes:

```
export default (
  <Route component={CoreLayout} path="/">
    <IndexRoute component={PublishingApp} name='home' />
    <Route component={LoginView} path='login' name='login' />
    <Route component={LogoutView} path='logout' name='logout' />
    <Route component={RegisterView} path='register'
      name='register' />
    <Route component={DashboardView}
      path='dashboard' name='dashboard' />
    <Route component={AddArticleView}
      path='add-article' name='add-article' />
    <Route component={EditArticleView}
      path='/edit-article/:articleID' name='edit-article' />
  </Route>
);
```

As you can see, we have added the `EditArticleViews` route with `path='/edit-article/:articleID'`; as you should know already, the `articleID` will be sent to us with props as `this.props.params.articleID` (this is a default feature of `redux-router`).

The next step is to create the `src/views/articles/EditArticleView.js` component, which is a new component (mocked for now):

```
import React from 'react';
import Falcor from 'falcor';
import {Link} from 'react-router';
import falcorModel from '../../falcorModel.js';
import {connect} from 'react-redux';
import {bindActionCreators} from 'redux';
import articleActions from '../../actions/article.js';
import WYSIWYGEditor from '../../components/articles/WYSIWYGEditor';
import {stateToHTML} from 'draft-js-export-html';
import RaisedButton from 'material-ui/lib/raised-button';

const mapStateToProps = (state) => ({
  ...state
});

const mapDispatchToProps = (dispatch) => ({
  articleActions: bindActionCreators(articleActions, dispatch)
});

class EditArticleView extends React.Component {
  constructor(props) {
    super(props);
  }

  render () {
    return <h1>An edit article MOCK</h1>
  }
}

export default connect(mapStateToProps,
  mapDispatchToProps)(EditArticleView);
```

Here, you can find a standard view component with a `render` function that returns a mock (we will improve it later). We have already put all the required imports in place (we will use all of them in the next iteration of the `EditArticleView` component).

Let's add a dashboard link to an article's edition

Make a small tweak in `src/views/DashboardView.js`:

```
let articlesJSX = [];
this.props.article.forEach((articleDetails, articleKey) => {
  let currentArticleJSX = (
    <Link to={`&grave;/edit-article/${articleDetails['_id']}&grave;`}
      key={articleKey}>
      <ListItem
        leftAvatar={<img
          src='/static/placeholder.png'
          width='50'
          height='50' />}
        primaryText={articleDetails.articleTitle}
        secondaryText={articleDetails.articleContent}
      />
    </Link>
  );

  articlesJSX.push(currentArticleJSX);
});
```

Here, we have two things that need to be changed: adding a `Link` attribute to `to={``/edit-article/${articleDetails['_id']}``}`. This will redirect a user to the article's edition view after clicking on `ListItem`. We also need to give a `Link` element a unique `key` property.

Creating a new action and reducer

Modify the `src/actions/article.js` file and add this new action called `EDIT_ARTICLE`:

```
export default {
  articlesList: (response) => {
    return {
      type: 'ARTICLES_LIST_ADD',
      payload: { response: response }
    }
  },
  pushNewArticle: (response) => {
    return {
      type: 'PUSH_NEW_ARTICLE',
      payload: { response: response }
    }
  },
};
```

```
    editArticle: (response) => {
      return {
        type: 'EDIT_ARTICLE',
        payload: { response: response }
      }
    }
  }
}
```

The next step is to improve our reducer at `src/reducers/article.js`:

```
import mapHelpers from '../utils/mapHelpers';

const article = (state = {}, action) => {
  switch (action.type) {
    case 'ARTICLES_LIST_ADD':
      let articlesList = action.payload.response;
      return mapHelpers.addMultipleItems(state, articlesList);
    case 'PUSH_NEW_ARTICLE':
      let newArticleObject = action.payload.response;
      return mapHelpers.addItem(state, newArticleObject['_id'],
        newArticleObject);
    case 'EDIT_ARTICLE':
      let editedArticleObject = action.payload.response;
      return mapHelpers.addItem(state, editedArticleObject['_id'],
        editedArticleObject);
    default:
      return state;
  }
};export default article;
```

As you can find here, we have added a new switch case for `EDIT_ARTICLE`. We use our `mapHelpers.addItem`; in general, if `_id` does exist in Map, then it replaces a value (this works great for editing actions).

Edit mode in `src/components/articles/WYSIWYGeditor.js`

Let's now implement the ability to use the edit mode in our `WYSIWYGeditor` components by improving our construction in the `WYSIWYGeditor.js` file:

```
export default class WYSIWYGeditor extends React.Component {
  constructor(props) {
    super(props);

    let initialEditorFromProps;
```

```
if (typeof props.initialValue === 'undefined' || typeof
  props.initialValue !== 'object') {
  initialEditorFromProps =
    EditorState.createWithContent
      (ContentState.createFromText(''));
} else {
  let isInvalidObject = typeof props.initialValue.entityMap
    === 'undefined' || typeof props.initialValue.blocks ===
      'undefined';

  if (isInvalidObject) {
    alert('Invalid article-edit error provided, exit');
    return;
  }
  let draftBlocks = convertFromRaw(props.initialValue);
  let contentToConsume =
    ContentState.createFromBlockArray(draftBlocks);

  initialEditorFromProps =
    EditorState.createWithContent(contentToConsume);
}

this.state = {
  editorState: initialEditorFromProps
};

this.focus = () => this.refs['refWYSIWYGeditor'].focus();
this.onChange = (editorState) => {
  var contentState = editorState.getCurrentContent();

  let contentJSON = convertToRaw(contentState);
  props.onChangeTextJSON(contentJSON, contentState);
  this.setState({editorState})
};

this.handleKeyCommand = (command) =>
  this._handleKeyCommand(command);
this.toggleInlineStyle = (style) =>
  this._toggleInlineStyle(style);
this.toggleBlockType = (type) =>
  this._toggleBlockType(type);
}
```

Here you can find out how your constructor will look after making changes.

As you already know, draft-js is required to be an object, so we check in the first `if` statement whether it is one. Then, if not, we put an empty WYSIWYG as default (check `if(typeof props.initialValue === 'undefined' || typeof props.initialValue !== 'object'))`).

In the `else` statement, we put the following:

```
let isInvalidObject = typeof props.initialValue.entityMap ===
  'undefined' || typeof blocks === 'undefined';
if (isInvalidObject) {
  alert('Error: Invalid article-edit object provided, exit');
  return;
}
let draftBlocks = convertFromRaw(props.initialValue);
let contentToConsume =
  ContentState.createFromBlockArray(draftBlocks);
let initialEditorFromProps =
  EditorState.createWithContent(contentToConsume);
```

Here we check whether we have a valid draft-js JSON object; if not, we need to throw a critical error and return, because otherwise, the error can crash the whole browser (we need to handle that edge case with `withif(isInvalidObject)`).

After we have a valid object, we recover the state of our WYSIWYG editor with the use of the `convertFromRaw`, `ContentState.createFromBlockArray`, and `EditorState.createWithContent` functions provided by the draft-js library.

Improvements in EditArticleView

The last improvement before finishing the article edit mode is improving `src/views/articles/EditArticleView.js`:

```
class EditArticleView extends React.Component {
  constructor(props) {
    super(props);
    this._onDraftJSChange = this._onDraftJSChange.bind(this);
    this._articleEditSubmit = this._articleEditSubmit.bind(this);
    this._fetchArticleData = this._fetchArticleData.bind(this);

    this.state = {
      articleFetchError: null,
      articleEditSuccess: null,
      editedArticleID: null,
      articleDetails: null,
      title: 'test',
    };
  }
}
```

```
        contentJSON: {},
        htmlContent: ''
    };
}
```

This is our constructor; we will have some states variables such as `articleFetchError`, `articleEditSuccess`, `editedArticleID`, `articleDetails`, `title`, `contentJSON`, and `htmlContent`.

In general, all these variables are self-explanatory. Regarding the `articleDetails` variable here, we will keep the whole object fetched from a `reducer/mongoDB`. Things such as `title`, `contentHTML`, and `contentJSON` are kept in the `articleDetails` state (as you will find in a moment).

After you are done with the `EditArticleView` constructor, add some new functions:

```
componentWillMount() {
    this._fetchArticleData();
}

_fetchArticleData() {
    let articleID = this.props.params.articleID;
    if (typeof window !== 'undefined' && articleID) {
        let articleDetails = this.props.article.get(articleID);
        if(articleDetails) {
            this.setState({
                editedArticleID: articleID,
                articleDetails: articleDetails
            });
        } else {
            this.setState({
                articleFetchError: true
            })
        }
    }
}

onDraftJSChange(contentJSON, contentState) {
    let htmlContent = stateToHTML(contentState);
    this.setState({contentJSON, htmlContent});
}

_articleEditSubmit() {
    let currentArticleID = this.state.editedArticleID;
    let editedArticle = {
        _id: currentArticleID,
        articleTitle: this.state.title,
```

```
        articleContent: this.state.htmlContent,
        articleContentJSON: this.state.contentJSON
      }

      this.props.articleActions.editArticle(editedArticle);
      this.setState({ articleEditSuccess: true });
    }
  }
```

On `componentWillMount`, we will fetch data about the article with `_fetchArticleData`. The `_fetchArticleData` is getting the article's ID from props via `react-redux` (`let articleID = this.props.params.articleID;`). Then, we check whether we are not on the server side with `if(typeof window !== 'undefined' && articleID)`. After this, we use the `.get` Map function in order to get details from a reducer (`let articleDetails = this.props.article.get(articleID);`), and based on the case, we set the state of our component with the following:

```
if (articleDetails) {
  this.setState({
    editedArticleID: articleID,
    articleDetails: articleDetails
  });
} else {
  this.setState({
    articleFetchError: true
  })
}
```

Here you can find that in the `articleDetails` variable, we keep all data fetched from reducer/DB. In general, now we only have the frontend side because a backend side fetching an edited article will be introduced later in this book.

The `_onDraftJSChange` function is similar to the one in the `AddArticleView` component.

The `_articleEditSubmit` is quite standard, so I will leave it to you to read the code. I will only mention that `_id: currentArticleID` is very important, because it's used later in our `reducer/mapUtils` in order to update the article correctly in the article's reducer.

EditArticleView's render improvements

The last part is to improve our render function in the `EditArticleView` component:

```
render () {
  if (this.state.articleFetchError) {
    return <h1>Article not found (invalid article's ID
```



```
    {this.props.params.articleID})</h1>;
  } else if (!this.state.editedArticleID) {
    return <h1>Loading article details</h1>;
  } else if (this.state.articleEditSuccess) {
    return (
      <div style={{height: '100%', width: '75%', margin:
        'auto'}}>
        <h3>Your article has been edited successfully</h3>
        <Link to='/dashboard'>
          <RaisedButton
            secondary={true}
            type='submit'
            style={{margin: '10px auto', display: 'block',
              width: 150}}
            label='Done' />
        </Link>
      </div>
    );
  }
}

let initialWYSIWYGValue =
  this.state.articleDetails.articleContentJSON;

return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Edit an existing article</h1>
    <WYSIWYGeditor
      initialValue={initialWYSIWYGValue}
      name='editarticle'
      title='Edit an article'
      onChangeTextJSON={this._onDraftJSChange} />
    <RaisedButton
      onClick={this._articleEditSubmit}
      secondary={true}
      type='submit'
      style={{margin: '10px auto', display: 'block',
        width: 150}}
      label={'Submit Edition'} />
  </div>
);
}
```

We are managing different states of our component with

`if(this.state.articleFetchError), else if(!this.state.editedArticleID), and else if(this.state.articleEditSuccess), as follows:`

```
<WYSIWYGeditor
  initialValue={initialWYSIWYGValue}
  name='editarticle'
  title='Edit an article'
  onChangeTextJSON={this._onDraftJSChange} />
```

In this part, the major change is adding a new property called `initialValue`, which is passed down to the `WYSIWYGeditor`, the `draft-js` JSON object.

Deleting an article's feature implementation

Let's create a new action for deletion at `src/actions/article.js`:

```
deleteArticle: (response) => {
  return {
    type: 'DELETE_ARTICLE',
    payload: { response: response }
  }
}
```

Next, let's add a `DELETE_ARTICLE` switch case into `src/reducers/article.js`:

```
import mapHelpers from '../utils/mapHelpers';

const article = (state = {}, action) => {
  switch (action.type) {
    case 'ARTICLES_LIST_ADD':
      let articlesList = action.payload.response;
      return mapHelpers.addMultipleItems(state, articlesList);
    case 'PUSH_NEW_ARTICLE':
      let newArticleObject = action.payload.response;
      return mapHelpers.addItem(state, newArticleObject['_id'],
        newArticleObject);
    case 'EDIT_ARTICLE':
      let editedArticleObject = action.payload.response;
      return mapHelpers.addItem(state, editedArticleObject['_id'],
        editedArticleObject);
    case 'DELETE_ARTICLE':
      let deleteArticleId = action.payload.response;
      return mapHelpers.deleteItem(state, deleteArticleId);
    default:
```

```
        return state;
    }
    export default article
```

The last step in implementing a delete button is to modify `src/views/articles/EditArticleView.js` component. Import `Popover` (it will ask a second time whether you are sure about deleting an article):

```
import Popover from 'material-ui/lib/popover/popover';
Improve the constructor of EditArticleView:
class EditArticleView extends React.Component {
  constructor(props) {
    super(props);
    this._onDraftJSChange = this._onDraftJSChange.bind(this);
    this._articleEditSubmit = this._articleEditSubmit.bind(this);
    this._fetchArticleData = this._fetchArticleData.bind(this);
    this._handleDeleteTap = this._handleDeleteTap.bind(this);
    this._handleDeletion = this._handleDeletion.bind(this);
    this._handleClosePopover =
      this._handleClosePopover.bind(this);

    this.state = {
      articleFetchError: null,
      articleEditSuccess: null,
      editedArticleID: null,
      articleDetails: null,
      title: 'test',
      contentJSON: {},
      htmlContent: '',
      openDelete: false,
      deleteAnchorEl: null
    };
  }
}
```

The new things here are `_handleDeleteTap`, `_handleDeletion`, `_handleClosePopover`, and state (`htmlContent`, `openDelete`, `deleteAnchorEl`). Then, add three new functions to `EditArticleView`:

```
_handleDeleteTap(event) {
  this.setState({
    openDelete: true,
    deleteAnchorEl: event.currentTarget
  });
}

_handleDeletion() {
  let articleID = this.state.editedArticleID;
```

```
    this.props.articleActions.deleteArticle(articleID);

    this.setState({
      openDelete: false
    });
    this.props.history.pushState(null, '/dashboard');
  }

  _handleClosePopover() {
    this.setState({
      openDelete: false
    });
  }
}
```

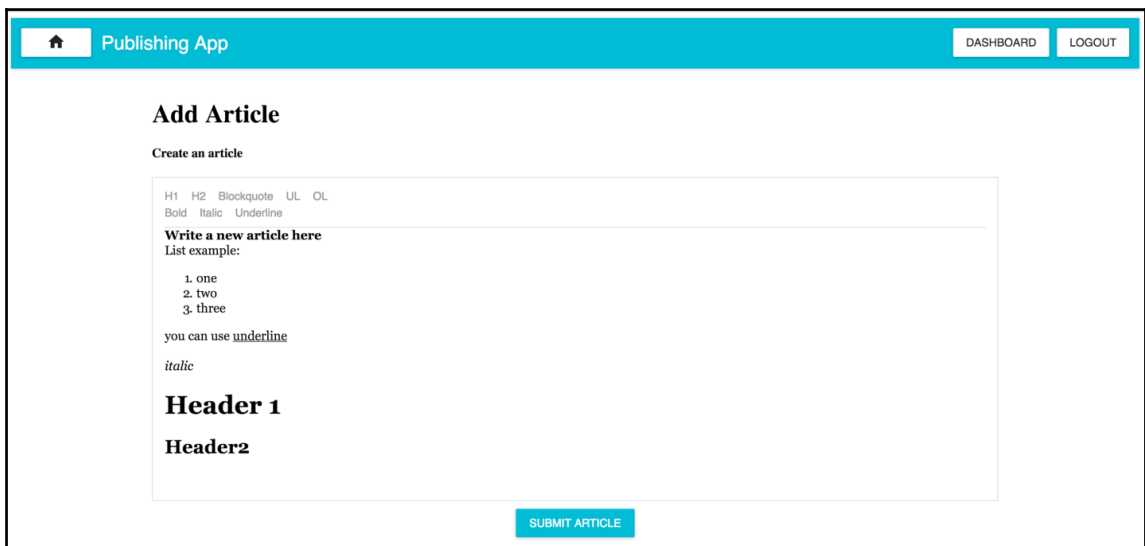
Improve the return in the render function:

```
let initialWYSIWYGValue =
  this.state.articleDetails.articleContentJSON;

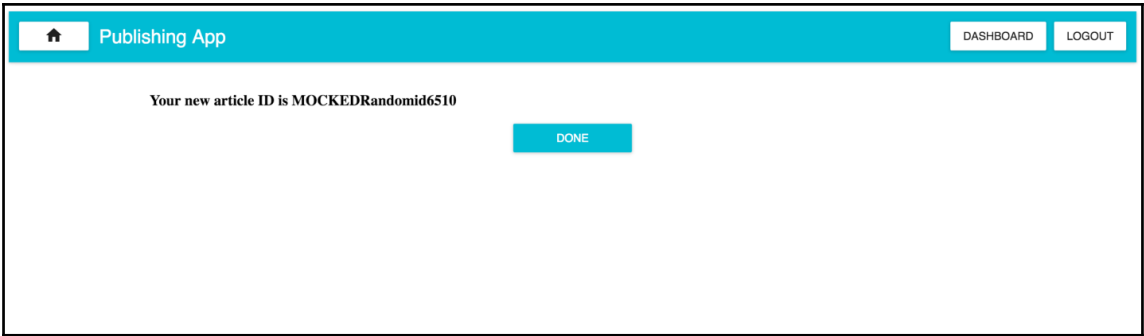
return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Edit an existng article</h1>
    <WYSIWYGeditor
      initialValue={initialWYSIWYGValue}
      name='editarticle'
      title='Edit an article'
      onChangeTextJSON={this._onDraftJSChange} />
    <RaisedButton
      onClick={this._articleEditSubmit}
      secondary={true}
      type='submit'
      style={{margin: '10px auto', display: 'block',
        width: 150}}
      label={'Submit Edition'} />
    <hr />
    <h1>Delete permanently this article</h1>
    <RaisedButton
      onClick={this._handleDeleteTap}
      label='Delete' />
    <Popover
      open={this.state.openDelete}
      anchorEl={this.state.deleteAnchorEl}
      anchorOrigin={{horizontal: 'left', vertical:
        'bottom'}}
      targetOrigin={{horizontal: 'left', vertical: 'top'}}
      onRequestClose={this._handleClosePopover}>
      <div style={{padding: 20}}>
        <RaisedButton
```

```
        onClick={this._handleDeletion}
        primary={true}
        label="Permanent delete, click here"/>
      </div>
    </Popover>
  </div>
);
```

Regarding render, new things are all under the new `hr` tag: `<h1>`: Delete permanently this article `<h1>`. `RaisedButton`: `DeletePopover` is a component from Material-UI. You can find more documentation of this component at <http://www.material-ui.com/v0.15.0-alpha.1/#/components/popover>. You can find in the following screenshots how it should look in the browser `RaisedButton`: Permanent delete, click here label. The `AddArticleView` component:



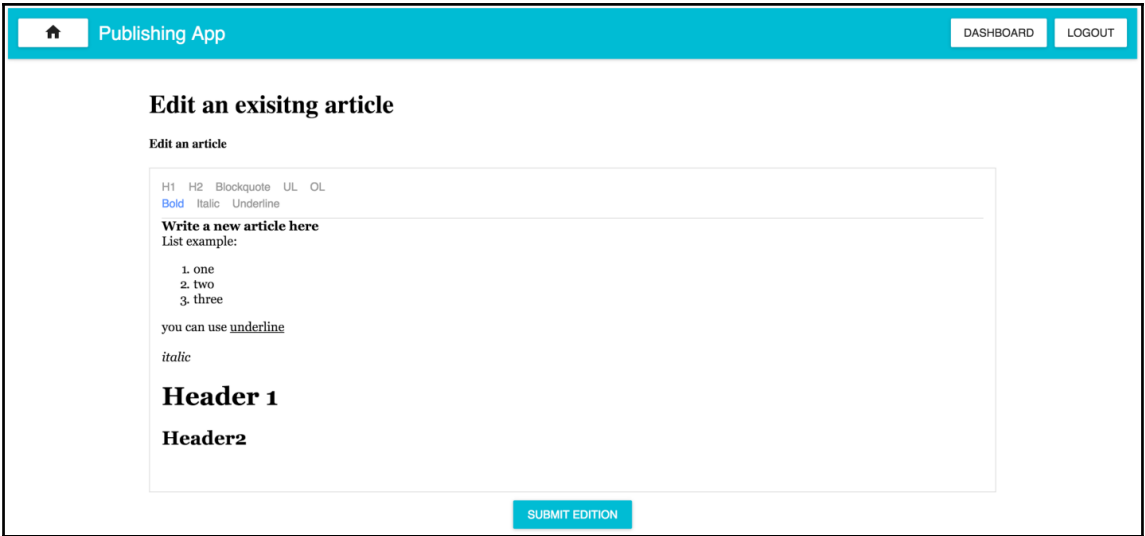
The AddArticleView component after a `SUBMIT ARTICLE` button has been hit:



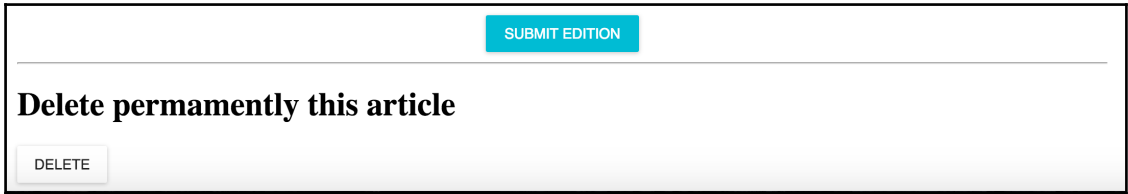
The dashboard component:



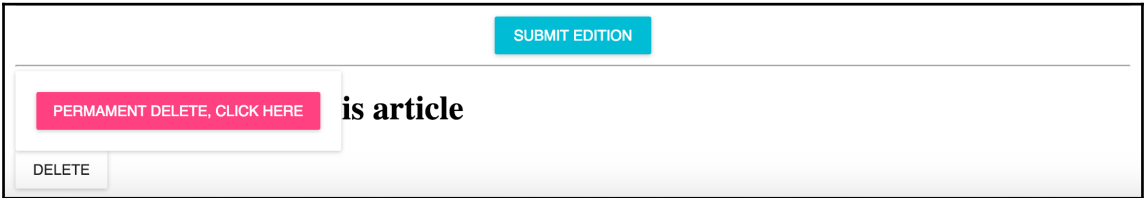
The EditArticleView component:



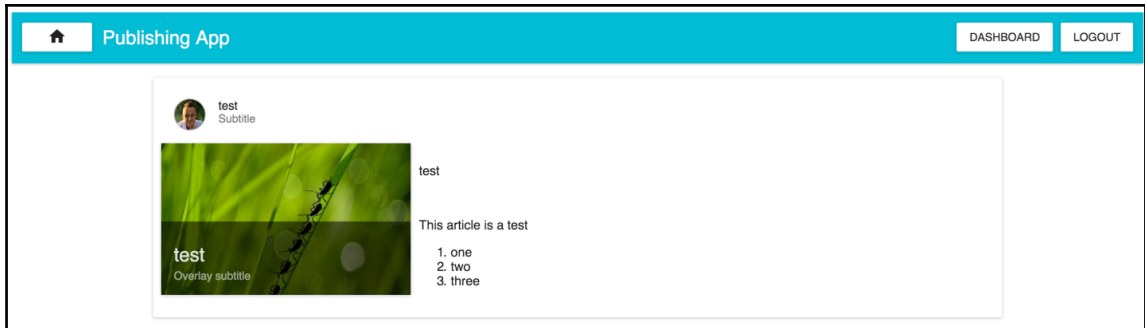
A DELETE button on the EditArticleView component:



A DELETE button on the EditArticleView component after first click (popover component):



A PublishingApp component (main page):



Summary

Currently, we have made a lot of progress on the frontend using Redux to store the state of our application in its single-state tree. The important drawback is that after you hit refresh, all the data disappears.

In the next chapter, we will start implementing the backend in order to store the articles in our database.

As you already know, Falcor is our glue that replaces the old popular RESTful approach; you will master stuff related to Falcor very soon. You will also learn what the differences between Relay/GraphQL and Falcor are. Both are trying to solve similar problems, but in very different ways.

Let's go even more in depth into our full-stack Falcor application. We will make it even more awesome for our end users.

5

Falcor Advanced Concepts

Currently, our app has the ability to add, edit, and delete articles, but only on the frontend with the help of Redux's reducers. We need to add some full-stack mechanism to make this able to perform CRUD operations on the database. We will also need to add some security features on the backend so non-authenticated users won't be able to perform CRUD operations on the MongoDB collections.

Let's pause coding for a moment. Before we start developing the full-stack Falcor mechanism, let's discuss our React, Node, and Falcor setup in more detail.

It's important to understand why we have chosen Falcor in our technical stack. In general, at the custom software development company where I work (you can find more at www.ReactPoland.com), we use Falcor as it has many great advantages for our clients in terms of the productivity of developing full-stack mobile/web applications. Some of them are as follows:

- The simplicity of the concept
- A speedup of over by 30 percent in development in comparison to a RESTful approach
- Shallow learning curve, so a developer who learns Falcor can become effective very quickly
- An effective way of fetching data (from backend to the client side) that is quite astounding

I will keep these four points short and sweet, for now. Later in the chapter, you will learn more about problems that you may face when using Falcor and Node.

Currently, we have assembled a kind of full-stack starter kit with React, Redux, Falcor, Node, Express, and MongoDB. It's not perfect yet. We will make it our focus for this chapter, which will include the following topics:

- Better understanding the big picture of *RESTless data fetching* solutions and the similarities and differences between Falcor and Relay/GraphQL
- How to secure routes in order to authenticate users on the backend
- How to handle errors on the backend and send them seamlessly to the frontend with the help of `errorSelectors`
- A detailed look at Falcor's sentinels and how exactly `$ref`, `$atom`, and `$error` work in Falcor
- What a JSON graph is and how it works
- What the virtual JSON concept is in Falcor

The problem that Falcor aims to solve

Before the era of single-page applications, there weren't problems with fetching data on the client, as all of the data was always fetched on the server, and even then, the server would send the HTML markup to the client. Each time someone clicked on a URL (`href`), our browser requested totally new HTML markup from the server.

Based on the preceding principles of non-SPA applications, Ruby on Rails became the king of web development's technical stack, but later things changed. Since 2009-2010, we've been creating more and more JavaScript client applications, which are more likely fetched once from the backend as, for example, a `bundle.js` file. They're called SPAs.

Because of this SP Apps trend, some new problems emerged that weren't known to non-SP Apps developers, such as fetching data from the API endpoint on the backend in order to consume that JSON data on the client side.

In general, the old-fashioned workflow for RESTful applications was as follows:

1. Create endpoints on the backend.
2. Create the fetching mechanism on the frontend.
3. Fetch data from the backend by coding POST/GET requests on the frontend based on the API's specification.
4. When you fetch the JSON from the backend to the frontend, you can consume the data and use it in order to create the UI view based on a certain use case.

This process is kind of frustrating if someone such as a client or boss changes their mind, because you were implementing the entire code on the backend and frontend. Later the backend API endpoints become irrelevant, so you need to start working on them from scratch based on the changed requirements.

Virtual JSON - one model everywhere

For Falcor, one model everywhere is the main tagline of this great library. In general, the main purpose of using it is to create a single JSON model that is exactly the same on the frontend and backend. What does this mean for us? It means that if anything changes, we need to change the model, which is exactly the same on the backend and frontend--so in case of any changes, we need to tweak our model without worrying about how the data is provided on the backend and fetched on the frontend.

Falcor's innovation is to introduce a new concept called virtual JSON (analogical to virtual DOM for React). This lets you represent all your remote data sources (for example, MongoDB in our case) as a single domain model. The whole idea is that you code the same way without caring where your data is: is it on the client-side memory cache or on the server? You don't need to care, as Falcor, with its innovative approach, does a lot of the job (for example, querying with `xhr` requests) for you.

Data fetching is a problem for developers. Falcor is here to help to make it simpler. You can fetch data from the backend to the frontend, writing fewer lines of code than ever!

It's May 2016, and the only viable competitors that I see on the horizon are the Facebook libraries called Relay (on the client side) and GraphQL (on the backend).

Let's try to compare both.

Falcor versus Relay/GraphQL

As with any tool, there are always pros and cons.

For certain, Falcor is always better than Relay/GraphQL in small/mid-sized projects, at least unless you have master developers (or you are a master yourself) who know Relay/GraphQL very well. Why is that?

In general, Relay (for the frontend) and GraphQL (for the backend) are two different tools and you must be efficient in order to use properly.

Very often in commercial environments, you don't have too much time to learn things from scratch. This is also a reason behind the success of React.

Why has React succeeded? React is much easier to grasp in order to be an efficient frontend developer. A CTO or technical director hires a newbie developer who knows jQuery (for example), and then the CTO can easily project that this junior developer will be effective in React in 7 to 14 days; I was teaching junior frontend developers with basic knowledge of JavaScript/jQuery, and I found out that they quite quickly become efficient in creating client-side apps with React.

We can find the same situation with Falcor. Falcor, in comparison to Relay + GraphQL, is like the simplicity of React compared to the monolithic framework of Angular.

This single factor described in the previous few paragraphs means that Falcor is better for small/mid-size projects with a limited budget.

You may find some opportunities to learn Relay/GraphQL in bigger companies with much bigger budgets, such as Facebook, when you have 6 months to master a technology.

FalcorJS can be mastered effectively in two weeks, but GraphQL + Relay cannot.

Big-picture similarities

Both these tools are trying to solve the same problem. They are efficient by design for both developers and the network (trying to optimize the number of queries in comparison to a RESTful approach).

They have the ability to query the backend server in order to fetch data and also have batching ability (so you can fetch more than two different sets of data with one network request). Both have some caching abilities.

Technical differences - overview

With a technical overview, we can find out that in general, Relay allows you to query an undefined number of items from the GraphQL server. In Falcor, for comparison, you need to first ask the backend how many items it has before being able to query for the collection objects' details (such as articles, in our book's case).

In general, the biggest difference here is that GraphQL/Relay is a query language tool and Falcor is not. What is a query language? It's one with which you can make queries from the frontend similar to SQL, like this:

```
post: () => Relay.QL
  fragment on Articles {
    title,
    content
  }
```

The preceding code can be made a query from the frontend via `Relay.QL`, and then GraphQL processes the query in the same way as SQL, like this:

```
SELECT title, content FROM Articles
```

Things may get harder if there are, for example, a million articles in the DB and you didn't expect so many on the frontend.

In Falcor, you do it differently, as you've already learned:

```
const articlesLength = await falcorModel.
  getValue('articles.length').
  then((length) => length);

const articles = await falcorModel.
  get(['articles', {from: 0, to: articlesLength-1},
    ['_id', 'articleTitle', 'articleContent']]).
  then((articlesResponse) => articlesResponse.json.articles);
```

In the preceding Falcor example, you must first know how many records there are in the MongoDB instance.

This is one of most important differences and creates some challenges for both sides.

For GraphQL and Relay, the question is whether the power of those query languages is worth the complexity created in the learning curve, because that complexity may not be worth it for small/mid-sized projects.

Now that the basic differences have been discussed, let's focus on Falcor and improving our current publishing app.

Improving our application and making it more reliable

We need to improve things such as the following:

- After a login, we shall send user details in each request (the token, username, and a role; you can find a screenshot later in the section *Improving our Falcor code on the frontend*)
- The backend needs to be secured so that authorization is checked before running add/edit/delete operations on the backend
- We need to provide the ability to catch errors on the backend and give a notification to the user on the frontend about something not working correctly

Securing the auth required routes

Currently, our app has the ability to add/edit/delete a route. The problem with our current implementation is that we don't check whether a client making a CRUD operation has the privileges to do so.

The solution of securing Falcor routes requires some changes to our current implementation, so for each request, before performing the operation, we will check whether we have got the correct token from the client and whether the user making the call has the ability to edit (in our case, it means that if anyone has an editor role and is authenticated correctly with his username and password, then he can add/edit/delete an article).

JSON Graph and JSON envelopes in Falcor

As the Falcor documentation states, "JSON Graph is a convention for modeling graph information as a JSON object. Applications that use Falcor represent all their domain data as a single JSON Graph object."

In general, JSON Graph in Falcor is valid JSON with some new features. To be more precise, JSON Graph introduces a new types of data besides strings, numbers, and Booleans. The new data type in Falcor is called a **sentinel**. I will try to explain it later in the chapter.

Generally, the second most important thing to understand in Falcor are JSON envelopes. The great thing is that they work out of the box, so you don't have to worry too much about them. But if you want to know what the short and sweet answer is, JSON envelopes help send JSON's model via HTTP's protocol. It's a way of transferring data from frontend to backend (with the `.call`, `.set`, and `.get` methods). In the same way, before the backend (after processing a request's details), before sending the improved model's details to the client side, Falcor puts it into an *envelope* so that it can be easily transferred via a network.

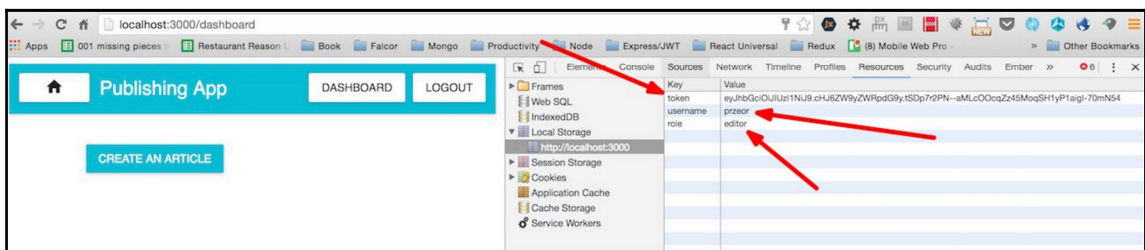
A good (but not perfect) analogy for JSON envelopes is that you put a written list into an envelope because you don't want to send some valuable information over from point *A* to point *B*; the network doesn't care what you send in that envelope. The most important thing is that the sender and the receiver know the context of the application model.

You can find more information about the JSON Graph and envelopes at <http://netflix.github.io/falcor/documentation/jsongraph.html>.

Improving our Falcor code on the frontend

Currently, after a user authorizes himself, all the data is saved into local storage. We need to close the loop by sending that data--token, username, and role--back to the backend with each request so we can check again whether a user is authenticated correctly. If not, then we need to send an authentication error with the request and show it back on the frontend.

The arrangement in the following screenshot is specifically important for security reasons so that no unauthorized user can add/edit/delete an article in our database:



In the screenshot, you can find out where you can get information about the `localStorage` data.

The following is our current code in `src/falcorModel.js`:

```
// this code is already in the codebase
const falcor = require('falcor');
const FalcorDataSource = require('falcor-http-datasource');

const model = new falcor.Model({
  source: new FalcorDataSource('/model.json')
});
export default model;
```

We need to change this to a new, improved version:

```
import falcor from 'falcor';
import FalcorDataSource from 'falcor-http-datasource';

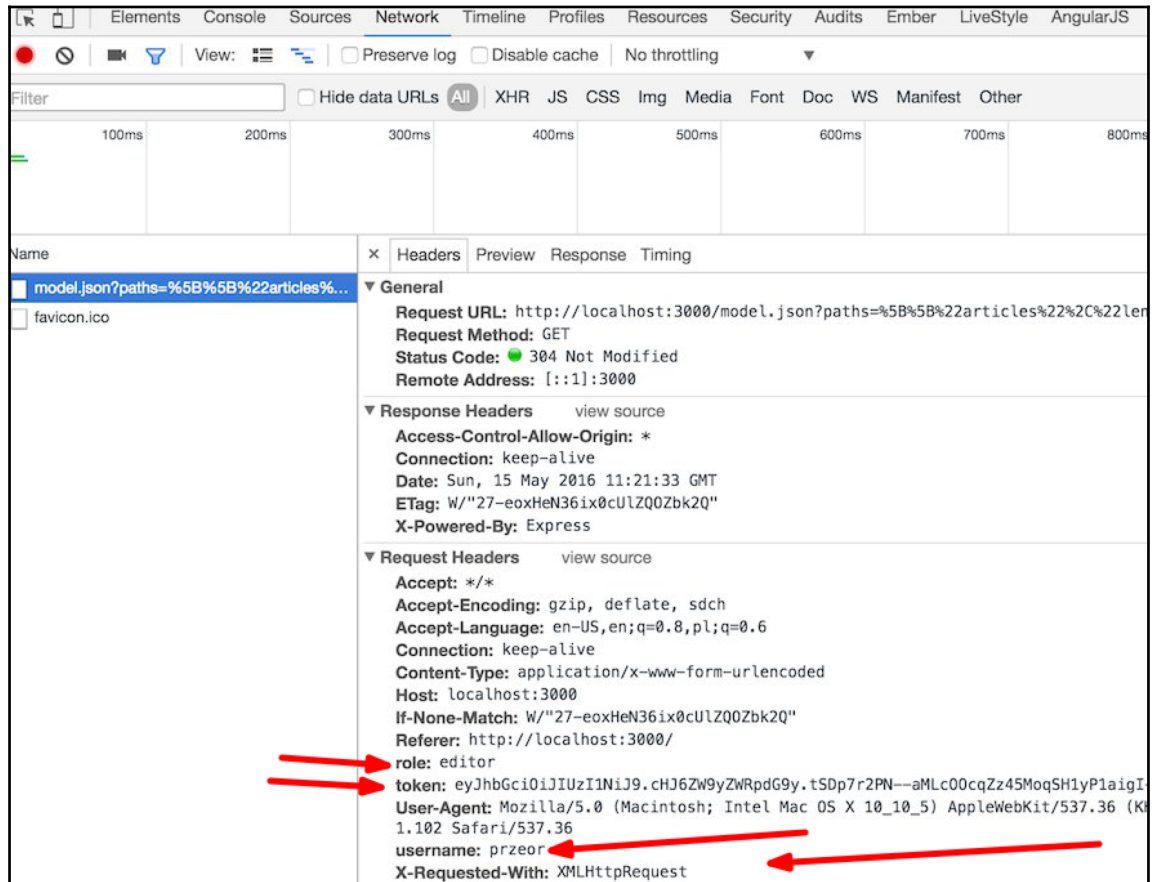
class PublishingAppDataSource extends FalcorDataSource {
  onBeforeRequest ( config ) {
    const token = localStorage.token;
    const username = localStorage.username;
    const role = localStorage.role;

    if (token && username && role) {
      config.headers['token'] = token;
      config.headers['username'] = username;
      config.headers['role'] = role;
    }
  }
}

const model = new falcor.Model({
  source: new PublishingAppDataSource('/model.json')
});
export default model;
```

What have we done in the previous code snippet? The `extends` keyword from ECMAScript6 shows an example of where the simplicity of the class syntax shines. Extending the `FalcorDataSource` means that `PublishingAppDataSource` inherits everything that the `FalcorDataSource` has and it makes the `onBeforeRequest` method have our custom behavior (by mutating `config.headers`). The `onBeforeRequest` method is checking the configuration mutated by us before our `xhr` instance is created. This helps us modify the `XMLHttpRequest` with a token, username, and role--in case our app's user logs out in the meantime, we can send that information to the backend.

After you implement the previous code in `falcorModel.js` and a user is logged, those variables will be added to each request:



Improving server.js and routes.js

In general, we currently export an array of objects from the `server/routes.js` file. We need to improve it, so we will return a function that will modify our array of objects so we have control over which route is returned to which user, and in case a user does not have a valid token or enough privileges, we will return an error. This will improve the security of our whole app.

In the `server/server.js` file, find this old code:

```
// this shall be already in your codebase
app.use('/model.json', falcorExpress.dataSourceRoute((req, res)
=> {
  return new falcorRouter(routes);
}));
```

Replace it with this improved one:

```
app.use('/model.json', falcorExpress.dataSourceRoute((req, res)
=> {
  return new falcorRouter(
    []
    .concat(routes(req, res))
  );
}));
```

In our new version, we assume that the `routes` variable is a function with the `req` and `res` variables.

Let's improve the `routes` itself so we won't return an array anymore, but a function that returns an array (so we end up with more flexibility).

The next step is to improve the `server/routes.js` file in order to make a function that receives the `currentSession` object, which stores all the information about a request. We need to change this in `routes.js`:

```
// this code is already in your codebase:
const PublishingAppRoutes = [
  ...sessionRoutes,
  {
    route: 'articles.length',
    get: () => {
      return Article.count({}, function(err, count) {
        return count;
      }).then ((articlesCountInDB) => {
        return {
          path: ['articles', 'length'],
          value: articlesCountInDB
        }
      })
    }
  },
  //
  // ..... There is more code between, it has been truncated in
  //order to save space

```

```
//  
export default PublishingAppRoutes;
```

Instead of exporting an array of routes, we need to export a function that will return routes based on a current request's header details.

The top part of the `server/routes.js` file (with imports) is as follows:

```
import configMongoose from './configMongoose';  
import sessionRoutes from './routesSession';  
import jsonGraph from 'falcor-json-graph';  
import jwt from 'jsonwebtoken';  
import jwtSecret from './configSecret';  
  
let $atom = jsonGraph.atom; // this will be explained later  
                           //in the chapter  
const Article = configMongoose.Article;
```

Follow this by exporting a new function:

```
export default ( req, res ) => {  
  let { token, role, username } = req.headers;  
  let userDetailsToHash = username+role;  
  let authSignToken = jwt.sign(userDetailsToHash,  
    jwtSecret.secret);  
  let isAuthorized = authSignToken === token;  
  let sessionObject = {isAuthorized, role, username};  
  
  console.info(`&grave;The ${username} is authorized === &grave;,,  
    isAuthorized);  
  
  const PublishingAppRoutes = [  
    ...sessionRoutes,  
    {  
      route: 'articles.length',  
      get: () => {  
        return Article.count({}, function(err, count) {  
          return count;  
        }).then ((articlesCountInDB) => {  
          return {  
            path: ['articles', 'length'],  
            value: articlesCountInDB  
          }  
        })  
      })  
    }  
  ]};
```

```
    return PublishingAppRoutes;
  }
```

First of all, we receive the `req` (request details) and `res` (object that represents the HTTP response) variables into the arrow functions. Based on the information provided by `req`, we get the header details (`let { token, role, username } = req.headers;`). Next, we have `userDetailsToHash` and then we check what will be the correct `authToken` with `let authSignToken = jwt.sign(userDetailsToHash, jwtSecret.secret);`. Afterward, we check whether the user is authorized with `let isAuthorized = authSign === token`. Then we create a `sessionObject`, which will be reused across all the Falcor routes later (`let sessionObject = {isAuthorized, role, username};`).

Currently, we have one route (`articles.length`), which was described in Chapter 2, *Full-Stack Login and Registration for Our Publishing App* (so there's nothing new so far).

As you can see in the previous code, instead of exporting `PublishingAppRoutes` directly, we are exporting with the arrow function `export default (req, res)`.

We need to re-add (under `articles.length`) the second route, called `articles[{{integers}}]["_id", "articleTitle", "articleContent"]`, with the following code in the `server/routes`:

```
{
  route:
    'articles[{{integers}}]["_id", "articleTitle", "articleContent"]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];

    return Article.find({}, function(err, articlesDocs) {
      return articlesDocs;
    }).then ((articlesArrayFromDB) => {
      let results = [];
      articlesIndex.forEach((index) => {
        const singleArticleObject =
          articlesArrayFromDB[index].toObject();

        const falcorSingleArticleResult = {
          path: ['articles', index],
          value: singleArticleObject
        };

        results.push(falcorSingleArticleResult);
      });
      return results;
    })
  }
}
```

```
}
```

This is the route that fetches articles from databases and returns `falcor-route` for it. It's exactly the same as introduced before; the only different is that now it's part of the function (`export default (req, res) => { ... }`).

Before we start to implement add/edit/delete on the backend with `falcor-router`, we need to introduce ourselves to the concept of sentinels, as it will be very important for the well-being of our full-stack application, the reason for which will be explained in a moment.

Falcor's sentinel implementation

Let's understand what sentinels are. They are required to make Fullstack's Falcor application work. It's a set of tools you have to learn.

They are new primitive value types created exclusively for making data transportation between the backend and client side much easier and out of the box (examples of new Falcor primitive values are `$error` and `$ref`). Here's an analogy: you have types in a regular JSON such as `string`, `number`, and `object` and. On the other hand, in Falcor's virtual JSON, you can additionally use sentinels such as `$error`, `$ref`, or `$atom` alongside the standard JSON types listed previously.



Additional information about sentinels is available at <https://netflix.github.io/falcor/documentation/model.html#sentinel-metadata>.

At this stage, it's important to understand how Falcor's sentinels are working. The different types of sentinel in Falcor are explained in the following sections.

The `$ref` sentinel

According to the documentation, "a reference is a JSON object with a `$type` key that has a value of `ref` and a `value` key that has a `Path` array as its value."

"A reference is like a symbolic link in the UNIX filesystem," as the documentation states, and this comparison is very good.

An example of `$ref` is as follows:

```
{ $type: 'ref', value: ['articlesById', 'STRING_ARTICLE_ID_HERE'] }
```



If you use `$ref(['articlesById', 'STRING_ARTICLE_ID_HERE'])`, it's equal to the preceding example. The `$ref` sentinel is a function that changes the array's details into that `$type` and value's notation object.

You can find both approaches in order to deploy/use `$ref` in any Falcor-related project, but in our project, we will stick to the `$ref(['articlesById', 'STRING_ARTICLE_ID_HERE'])` convention.

Just to make it clear, this is how to import a `$ref` sentinel in our codebase:

```
// wait, this is just an example, don't code this here:
import jsonGraph from 'falcor-json-graph';
let $ref = jsonGraph.ref;
// now you can use $ref([x, y]) function
```

After you import `falcor-json-graph`, you can use the `$ref` sentinel. You will already have installed the `falcor-json-graph` library as the installation has been described in the previous chapter; if not, use this:

```
npm i --save falcor-json-graph@1.1.7
```

But what does `articlesById` mean in that whole `$ref` gig? And what does `STRING_ARTICLE_ID_HERE` mean in the preceding example? Let's look at an example from our project that might make it clearer for you.

Detailed example of the \$ref sentinel

Let's assume that we have two articles in our MongoDB instance:

```
// this is just explanation example, don't write this here
// we assume that _id comes from MongoDB
[
  {
    _id: '987654',
    articleTitle: 'Lorem ipsum - article one',
    articleContent: 'Here goes the content of the article'
  },
  {
    _id: '123456',
    articleTitle: 'Lorem ipsum - article two',
    articleContent: 'Sky is the limit, the content goes here.'
  }
]
```

So based on our array's example with mocked articles (IDs 987654 and 123456), the `$ref` will look as follows:

```
// JSON envelope is an array of two $refs
// The following is an example, don't write it
[
  $ref([ articlesById, '987654' ]),
  $ref([ articlesById, '123456' ])
]
```

An even more detailed answer is this one:

```
// JSON envelope is an array of two $refs (other notation than
//above, but the same end effect)
[
  { $type: 'ref', value: ['articlesById', '987654'] },
  { $type: 'ref', value: ['articlesById', '123456'] }
]
```



An important thing to note is that `articlesById` is a new route that hasn't been created yet (we will do so in a moment).

But why do we need those `$ref` in our articles?

In general, you can keep a reference (like a symbolic link in Unix) in many places to one object in the database. In our case, it's an article with a certain `_id` in the article's collection.

When do `$ref` sentinels come in handy? Imagine that in our publishing app's model, we add a *recently visited* articles feature and provide the ability to like an article (like on Facebook).

Based on these two new features, our new model will look as follows (this is just an example; don't code it):

```
// this is just explanatory example code:
let cache = {
  articles: [
    {
      id: 987654,
      articleTitle: 'Lorem ipsum - article one',
      articleContent: 'Here goes the content of the article'
      numberOfLikes: 0
    },
    {
      id: 123456,
```

```
      articleTitle: 'Lorem ipsum - article two from backend',
      articleContent: 'Sky is the limit, the content goes
        here.',
      numberOfLikes: 0
    }
  ],
  recentlyVisitedArticles: [
    {
      id: 123456,
      articleTitle: 'Lorem ipsum - article two from backend',
      articleContent: 'Sky is the limit, the content goes
        here.',
      numberOfLikes: 0
    }
  ]
};
```

Based on our preceding example's model, if someone likes an article with ID 123456, we will need to update the model in two places. That's exactly where `$ref` comes in handy.

Improving our articles' numberOfLikes with \$ref

Let's improve our example to the following:

```
let cache = {
  articlesById: {
    987654: {
      _id: 987654,
      articleTitle: 'Lorem ipsum - article one',
      articleContent: 'Here goes the content of the article'
      numberOfLikes: 0
    },
    123456: {
      _id: 123456,
      articleTitle: 'Lorem ipsum - article two from backend',
      articleContent: 'Sky is the limit, the content goes
        here.',
      numberOfLikes: 0
    }
  },
  articles: [
    { $type: 'ref', value: ['articlesById', '987654'] },
    { $type: 'ref', value: ['articlesById', '123456'] }
  ],
  recentlyVisitedArticles: [
    { $type: 'ref', value: ['articlesById', '123456'] }
  ]
};
```



```
  ]  
};
```

In our new improved `$ref` example, you can find the notation where you need to tell Falcor the ID of the article you want to have in `articles` or `recentlyVisitedArticles`. Falcor on its own will follow the `$ref` sentinel, knowing the route name (the `articlesById` route in this case) and ID of the object we are looking for (in our example, `123456` or `987654`). We will use it in practice in a moment.

Understand that this is a simplified version of how it works, but the best analogy to use in order to understand `$ref` is UNIX's symbolic links.

Practical use of \$ref in our project

Okay, that was a lot of theory--let's start coding! We will improve our Mongoose model.

Then we'll add the `$ref` sentinels described before into the `server/routes.js` file:

```
// example of ref, don't write it yet:  
let articleRef = $ref(['articlesById', currentMongoID]);
```

We will also add two Falcor routes, `articlesById` and `articles.add`. On the frontend, we will make some improvements to `src/layouts/PublishingApp.js` and `src/views/articles/AddArticleView.js`.

Let's start the fun.

Mongoose config improvements

First thing we will do is open the Mongoose model at `server/configMongoose.js`:

```
// this is old codebase, you already shall have it:  
import mongoose from 'mongoose';  
  
const conf = {  
  hostname: process.env.MONGO_HOSTNAME || 'localhost',  
  port: process.env.MONGO_PORT || 27017,  
  env: process.env.MONGO_ENV || 'local',  
};  
  
mongoose.connect(&grave;mongodb://${conf.hostname}:${conf.port}/${  
  ${conf.env}&grave;);
```

```
const articleSchema = {
  articleTitle:String,
  articleContent:String
}
```

We'll improve it to this version:

```
import mongoose from 'mongoose';
const Schema = mongoose.Schema;

const conf = {
  hostname: process.env.MONGO_HOSTNAME || 'localhost',
  port: process.env.MONGO_PORT || 27017,
  env: process.env.MONGO_ENV || 'local',
};

mongoose.connect(&grave;mongodb://${conf.hostname}:${conf.port}/${
  ${conf.env}&grave;);

const articleSchema = new Schema({
  articleTitle:String,
  articleContent:String,
  articleContentJSON: Object
},
{
  minimize: false
}
);
```

In the preceding code, you'll find we import new `const Schema = mongoose.Schema`. Later, we improve our `articleSchema` with `articleContentJSON: Object`. This is required, because the state of draft-js will be kept in a JSON object. This will be useful if a user creates an article, saves it to the database, and later would like to edit the article. In such a case, we'll use this `articleContentJSON` in order to restore the content state of the draft-js editor.

The second thing is providing options with `{ minimize: false }`. This is required because by default Mongoose gets rid of all empty objects, such as `{ emptyObject: {}, nonEmptyObject: { test: true } }`, so if `minimize: false` isn't set up then we would get incomplete objects in our database (it's a very important step to have this flag here). There are some draft-js objects that are required, but by default are empty (specifically the `entityMap` property of a draft-js object).

The server/routes.js improvements

In the `server/routes.js` file, we need to start using the `$ref` sentinel. Your import in that file should look as follows:

```
import configMongoose from './configMongoose';
import sessionRoutes from './routesSession';
import jsonGraph from 'falcor-json-graph'; // this is new
import jwt from 'jsonwebtoken';
import jwtSecret from './configSecret';

let $ref = jsonGraph.ref; // this is new
let $atom = jsonGraph.atom; // this is new
const Article = configMongoose.Article;
```

In the preceding code snippet, the only new thing is that we import `jsonGraph` from `'falcor-json-graph'`; and then add `let $ref = jsonGraph.ref;` and `let $atom = jsonGraph.atom.`

We have added the `$ref` sentinel in our `routes.js` scope. We need to prepare a new route, `articlesById[{keys}]` ["_id", "articleTitle", "articleContent", "articleContentJSON"], as follows:

```
{
  route: 'articlesById[{keys}] ["_id", "articleTitle",
    "articleContent", "articleContentJSON"]',
  get: function(pathSet) {
    let articlesIDs = pathSet[1];
    return Article.find({
      '_id': { $in: articlesIDs }
    }, function(err, articlesDocs) {
      return articlesDocs;
    }).then ((articlesArrayFromDB) => {
      let results = [];

      articlesArrayFromDB.map((articleObject) => {
        let articleResObj = articleObject.toObject();
        let currentIdString = String(articleResObj['_id']);

        if (typeof articleResObj.articleContentJSON !==
          'undefined') {
          articleResObj.articleContentJSON =
            $atom(articleResObj.articleContentJSON);
        }

        results.push({
```

```
        path: ['articlesById', currentIdString],
        value: articleResObj
      });
    });
    return results;
  });
},
```

The `articlesById[{keys}]` route is defined, and the keys are the IDs of the request URL that we need to return in the request, as you can see with `const articlesIDs = pathSet[1];`.

To be more specific regarding `pathSet`, check out this example:

```
// just an example:
[
  { $type: 'ref', value: ['articlesById', '123456'] },
  { $type: 'ref', value: ['articlesById', '987654'] }
]
```

In this case, the `falcor-router` will follow `articlesById`, and in the `pathSet`, you will get this (you can see the exact value of the `pathSet`):

```
['articlesById', ['123456', '987654']]
```

The value of the `articlesIDs` from `const articlesIDs = pathSet[1];` you can find here:

```
['123456', '987654']
```

As you will find later, we use this `articlesIDs` value next:

```
// this is already in your codebase:
return Article.find({
  '_id': { $in: articlesIDs }
}, function(err, articlesDocs) {
```

As you can see in `'_id': { $in: articlesIDs }`, we are passing an array of `articlesIDs`. Based on those IDs, we will receive an array of certain articles found by IDs (the SQL `WHERE` equivalent). The next step here is iterating over received articles:

```
// this already is in your codebase:
articlesArrayFromDB.map((articleObject) => {
```

Push the object into the results array:

```
// this already is in your codebase:
let articleResObj = articleObject.toObject();
let currentIdString = String(articleResObj['_id']);

if (typeof articleResObj.articleContentJSON !== 'undefined') {
  articleResObj.articleContentJSON =
    $atom(articleResObj.articleContentJSON);
}

results.push({
  path: ['articlesById', currentIdString],
  value: articleResObj
});
```

Almost nothing is new in the preceding code snippet. The only new thing is this statement:

```
// this already is in your codebase:
if (typeof articleResObj.articleContentJSON !== 'undefined') {
  articleResObj.articleContentJSON =
    $atom(articleResObj.articleContentJSON);
}
```

We are explicitly using the `$atom` sentinel from Falcor here:

```
$atom(articleResObj.articleContentJSON);
```

JSON Graph atoms

The `$atom` sentinel is metadata attached to values, which has to be handled differently by the model. You can very simply return a value of a number type or a value of a string type with Falcor. It's more tricky for Falcor to return an object. Why?

Falcor is diffing with heavy usage of JavaScript's objects and arrays, and when we tell that an object/array is wrapped by an `$atom` sentinel (such as `$atom(articleResObj.articleContentJSON)` in our example), then Falcor knows that it shouldn't go deeper into that array/object. It's made that way by design for performance reasons.

What performance reasons? For example, if you return an array of 10,000 very deep objects without wrapping the array, it may take a very, very long time to build and diff the model. Generally, for performance reasons, any objects and arrays that you want to return via `falcor-router` to the frontend have to be wrapped by an `$atom` before doing so; otherwise, you will get an error like this (if you don't wrap by `$atom` this object):

```
Uncaught MaxRetryExceededError: The allowed number of retries
have been exceeded.
```

This error will be shown on the client side while Falcor tries to fetch those deeper objects without being wrapped by an `$atom` sentinel beforehand on the backend.

Improving the `articles[{integers}]` route

We now need to return a `$ref` sentinel to `articlesById` instead of all of the articles' details, so we need to change this old code:

```
// this already shall be in your codebase:
{
  route:
    'articles[{integers}][_id,"articleTitle","articleContent"]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];

    return Article.find({}, function(err, articlesDocs) {
      return articlesDocs;
    }).then ((articlesArrayFromDB) => {
      let results = [];
      articlesIndex.forEach((index) => {
        const singleArticleObject =
          articlesArrayFromDB[index].toObject();

        const falcorSingleArticleResult = {
          path: ['articles', index],
          value: singleArticleObject
        };

        results.push(falcorSingleArticleResult);
      });
      return results;
    })
  }
}
```

We'll improve that to this new code:

```
{
  route: 'articles[{integers}]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];

    return Article.find({}, '_id', function(err, articlesDocs) {
      return articlesDocs;
    }).then ((articlesArrayFromDB) => {
      let results = [];
      articlesIndex.forEach((index) => {
        let currentMongoID =
          String(articlesArrayFromDB[index]['_id']);
        let articleRef = $ref(['articlesById', currentMongoID]);

        const falcorSingleArticleResult = {
          path: ['articles', index],
          value: articleRef
        };

        results.push(falcorSingleArticleResult);
      });
      return results;
    })
  }
},
```

What has been changed? Look at the route in the old codebase:

`articles[{integers}]["_id", "articleTitle", "articleContent"]`. Currently, our `articles[{integers}]` route doesn't directly return (in the new version) the `for["_id", "articleTitle", "articleContent"]` data, so we had to delete it in order to get Falcor know about this fact (the `articlesById` is returning detailed information now).

The next thing that has been changed is that we create a new `$ref` sentinel with the following:

```
// this is already in your codebase:
let currentMongoID = String(articlesArrayFromDB[index]['_id']);
let articleRef = $ref(['articlesById', currentMongoID]);
```

As you see, by doing this, we are informing (with `$ref`) falcor-router that if the frontend requests any more information about `article[{integers}]`, then the falcor-router should follow the `articlesById` route in order to retrieve that data from the database.

After this, look at this old path's value:

```
// old version
const singleArticleObject = articlesArrayFromDB[index].toObject();

const falcorSingleArticleResult = {
  path: ['articles', index],
  value: singleArticleObject
};
```

You'll find that it has been replaced by the value of `articleRef`:

```
// new improved version
let articleRef = $ref(['articlesById', currentMongoID]);

const falcorSingleArticleResult = {
  path: ['articles', index],
  value: articleRef
};
```

As you can probably spot, in the old version we were returning all of the information about an article (the `singleArticleObject` variable), but in the new version we return only the `$ref` sentinel (`articleRef`).



The `$ref` sentinels make `falcor-router` automatically follow on the backend, so if there are any refs in the first route, Falcor resolves all the `$ref` sentinels until it gets all the pending data; after that, it returns the data in a single request, which saves a lot of latency (instead of performing several HTTP requests, everything followed with `$refs` is fetched in one browser-to-backend call).

New route in `server/routes.js`: `articles.add`

The only thing left that we need to do is add into the router a new `articles.add` route:

```
{
  route: 'articles.add',
  call: (callPath, args) => {
    const newArticleObj = args[0];
    var article = new Article(newArticleObj);

    return article.save(function (err, data) {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
    });
  }
}
```



```
    }
    else {
      return data;
    }
  }).then ((data) => {
    return Article.count({}, function(err, count) {
      }).then((count) => {
        return { count, data };
      });
  }).then ((res) => {
    //
    // we will add more stuff here in a moment, below
    //
    return results;
  });
}
```

As you can see here, we receive from the frontend a new article's details with `const newArticleObj = args[0];`, and later we create a new `Article` model with `var article = new Article(newArticleObj);`. After that, the `article` variable has a `.save` method, which is called in the following query. We perform two queries that return a promise from Mongoose. Here's the first:

```
return article.save(function (err, data) {
```

This `.save` method simply helps us insert the document into the database. After we have saved the article, we need to count how many there are in our database, so we run a second query:

```
return Article.count({}, function(err, count) {
```

After we have saved the article and counted it, we return that information (`return { count, data };`). The last thing is to return the new article ID and the count number from the backend to the frontend with the help of `falcor-router`, so we replace this comment:

```
//
// we will add more stuff here in a moment, below
//
```

In its place, we'll have this new code that helps us make things happen:

```
let newArticleDetail = res.data.toObject();
let newArticleID = String(newArticleDetail['_id']);
let NewArticleRef = $ref(['articlesById', newArticleID]);
let results = [
  {
```

```
    path: ['articles', res.count-1],
    value: NewArticleRef
  },
  {
    path: ['articles', 'newArticleID'],
    value: newArticleID
  },
  {
    path: ['articles', 'length'],
    value: res.count
  }
];
return results;
```

As you can see in the preceding code snippet, we get the `newArticleDetail` details here. Next, we take the new ID with `newArticleID` and make sure that it's a string. After all that, we define a new `$ref` sentinel with `let NewArticleRef = $ref(['articlesById', newArticleID]);`.

In the `results` variable, you can find three new paths:

- `path: ['articles', res.count-1]`: This path builds up the model, so we can have all the information in the Falcor model after we receive the response on the client side
- `path: ['articles', 'newArticleID']`: This helps us quickly fetch the new ID on the frontend
- `path: ['articles', 'length']`: This, of course, updates the length of our `articles` collections, so the frontend's Falcor model can have up-to-date information after we have added a new article

We just have made a backend route for adding an article. Let's now start working on the frontend so that we will be able to push all our new articles into the database.

Frontend changes in order to add articles

In the `src/layouts/PublishingApp.js` file, find this code:

```
get(['articles', {from: 0, to: articlesLength-1}, ['_id', 'articleTitle', 'articleContent'])).
```

Change it to an improved version with `articleContentJSON`:

```
get(['articles', {from: 0, to: articlesLength-1}, ['_id', 'articleTitle', 'articleContent', 'articleContentJSON'])).
```

The next step is to improve our `_submitArticle` function in `src/views/articles/AddArticleView.js` and add a `falcorModel` import:

```
// this is old function to replace:
_articleSubmit() {
  let newArticle = {
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON
  }

  let newArticleID = 'MOCKEDRandomid' + Math.floor(Math.random() *
    10000);

  newArticle['_id'] = newArticleID;
  this.props.articleActions.pushNewArticle(newArticle);
  this.setState({ newArticleID: newArticleID});
}
```

Replace this code with the following improved version:

```
async _articleSubmit() {
  let newArticle = {
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON
  }

  let newArticleID = await falcorModel
    .call(
      'articles.add',
      [newArticle]
    )
    .then((result) => {
      return falcorModel.getValue(
        ['articles', 'newArticleID']
      ).then((articleID) => {
        return articleID;
      });
    });

  newArticle['_id'] = newArticleID;
  this.props.articleActions.pushNewArticle(newArticle);
  this.setState({ newArticleID: newArticleID});
}
```

Also, at the top of the `AddArticleView.js` file, add this import; otherwise, `async_articleSubmit` won't work:

```
import falcorModel from '../../falcorModel.js';
```

As you can see, we have added the `async` keyword before the function name (`async_articleSubmit()`). The new thing is this request:

```
// this already is in your codebase:
let newArticleID = await falcorModel
  .call(
    'articles.add',
    [newArticle]
  ).
  then((result) => {
    return falcorModel.getValue(
      ['articles', 'newArticleID']
    ).then((articleID) => {
      return articleID;
    });
  });
```

Here, we wait for `falcorModel.call`. In the `.call` arguments, we add `newArticle`. Then, after the promise is resolved, we check what the `newArticleID` is with the following:

```
// this already is in your codebase:
return falcorModel.getValue(
  ['articles', 'newArticleID']
).then((articleID) => {
  return articleID;
});
```

Later, we simply use exactly the same stuff as in the old version:

```
newArticle['_id'] = newArticleID;
this.props.articleActions.pushNewArticle(newArticle);
this.setState({ newArticleID: newArticleID});
```

This simply pushes the updated `newArticle` with a real ID from MongoDB via the `articleActions` into the article's reducer. We also use `setState` with the `newArticleID` so you can see that the new article has been created correctly with a real Mongo ID.

Important note about route returns

You should be aware that in every route, we return an object or an array of an object; both approaches are fine even with one route to return. Take this, for example:

```
// this already is in your codebase (just an example)
{
  route: 'articles.length',
  get: () => {
    return Article.count({}, function(err, count) {
      return count;
    }).then ((articlesCountInDB) => {
      return {
        path: ['articles', 'length'],
        value: articlesCountInDB
      }
    })
  }
},
```

This can also return an array with one object, as follows:

```
get: () => {
  return Article.count({}, function(err, count) {
    return count;
  }).then ((articlesCountInDB) => {
    return [
      {
        path: ['articles', 'length'],
        value: articlesCountInDB
      }
    ]
  })
}
```

As you can see, even with one `articles.length`, we are returning an array (instead of a single object), and this will also work.

For the same reason as described previously, this is why, in `articlesById`, we have pushed multiple routes into the array:

```
// this is already in your codebase
let results = [];

articlesArrayFromDB.map((articleObject) => {
  let articleResObj = articleObject.toObject();
  let currentIdString = String(articleResObj['_id']);
```

```
if (typeof articleResObj.articleContentJSON !== 'undefined') {
  articleResObj.articleContentJSON =
    $atom(articleResObj.articleContentJSON);
}
// pushing multiple routes
results.push({
  path: ['articlesById', currentIdString],
  value: articleResObj
});
});
return results; // returning array of routes' objects
```

This is one thing that may be worth mentioning in the Falcor chapter.

Full-stack - editing and deleting an article

Let's create a route in the `server/routes.js` file for updating an existing document (edit feature):

```
{
  route: 'articles.update',
  call: async (callPath, args) => {
    {
      let updatedArticle = args[0];
      let articleID = String(updatedArticle._id);
      let article = new Article(updatedArticle);
      article.isNew = false;

      return article.save(function (err, data) {
        if (err) {
          console.info('ERROR', err);
          return err;
        }
      }).then ((res) => {
        return [
          {
            path: ['articlesById', articleID],
            value: updatedArticle
          },
          {
            path: ['articlesById', articleID],
            invalidate: true
          }
        ];
      });
    }
  }
}
```

```
},
```

As you can see here, we still use the `article.save` approach similar to the `articles.add` route. The important thing to note is that Mongoose requires the `isNew` flag to be `false` (`article.isNew = false;`). If you don't give this flag, then you will get a Mongoose error similar to this:

```
{ "error": { "name": "MongoError", "code": 11000, "err": "insertDocument
:: caused by :: 11000 E11000 duplicate key error index:
staging.articles.$_id _ dup key: { :
ObjectId('1515b34ed65022ec234b5c5f') }" } }
```

The rest of the code is quite simple; we save the article's model and then return the updated model via `falcor-router` with the following:

```
// this is already in your code base:
return [
  {
    path: ['articlesById', articleID],
    value: updatedArticle
  },
  {
    path: ['articlesById', articleID],
    invalidate: true
  }
];
```

The new thing is the `invalidate` flag. As it states in the documentation, "invalidate method synchronously removes several Paths or PathSets from a Model cache." In other words, you need to tell the Falcor model on the frontend that something has been changed in the `["articlesById", articleID]` path so that you will have synced data on both backend and frontend.



For more stuff about `invalidate` in Falcor, you can go to <https://netflix.github.io/falcor/doc/Model.html#invalidate>.

Deleting an article

In order to implement the `delete` feature, we need to create a new route:

```
{
  route: 'articles.delete',
  call: (callPath, args) =>
```

```
{
  const toDeleteArticleId = args[0];
  return Article.find({ _id: toDeleteArticleId }).
    remove((err) => {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
    }).then((res) => {
      return [
        {
          path: ['articlesById', toDeleteArticleId],
          invalidate: true
        }
      ]
    });
}
```

This also uses `invalidate`, but this time, this is the only thing that we return here, as the document has been deleted, so the only thing we need to do is to inform the browser's cache that the old article has been invalidated and there is nothing to replace it as in the update example.

Frontend - edit and delete

We have implemented the update and delete routes on the backend. Next, in the `src/views/articles/EditArticleView.js` file, you need to find this code:

```
// this is old already in your codebase:
_articleEditSubmit() {
  let currentArticleID = this.state.editedArticleID;
  let editedArticle = {
    _id: currentArticleID,
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON
  }

  this.props.articleActions.editArticle(editedArticle);
  this.setState({ articleEditSuccess: true });
}
```


Replace it with this `async _articleEditSubmit` function:

```
async _articleEditSubmit() {
  let currentArticleID = this.state.editedArticleID;
  let editedArticle = {
    _id: currentArticleID,
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON
  }

  let editResults = await falcorModel
    .call(
      ['articles', 'update'],
      [editedArticle]
    )
    .then((result) => {
      return result;
    });

  this.props.articleActions.editArticle(editedArticle);
  this.setState({ articleEditSuccess: true });
}
```

As you can see here, the most important thing is that we implemented the `.call` function in the `_articleEditSubmit` function that sends details of an edited object with the `editedArticle` variable.

In the same file, find the `_handleDeletion` method:

```
// old version
_handleDeletion() {
  let articleID = this.state.editedArticleID;
  this.props.articleActions.deleteArticle(articleID);

  this.setState({
    openDelete: false
  });
  this.props.history.pushState(null, '/dashboard');
}
```

Change it to the new improved version:

```
async _handleDeletion() {
  let articleID = this.state.editedArticleID;

  let deletionResults = await falcorModel
    .call(
```

```
        ['articles', 'delete'],
        [articleID]
      ).
      then((result) => {
        return result;
      });

      this.props.articleActions.deleteArticle(articleID);
      this.setState({
        openDelete: false
      });
      this.props.history.pushState(null, '/dashboard');
    }
  }
}
```

Similar to the deletion, the only difference is that we only send `articleID` of a deleted article with `.call`.

Securing the CRUD routes

We need to implement a way to secure all add/edit/delete routes and also make a universal **DRY (don't repeat yourself)** way of informing the user of errors that occurred on the backend. For example, errors that may occur on the frontend, and we need to inform the user with an error message in our React instance's client-side app:

- **Auth error:** You are not authorized to perform the action
- **Timeout error:** For example, you use an external API's service; we need to inform the user of any potential errors
- **Data doesn't exist:** There may be a case where a user will call for the ID of an article that doesn't exist in our DB, so let's inform him

In general, our goal for now is to create one universal way of moving all potential error messages on the backend to the client side so that we can improve the general experience of using our application.

The `$error` sentinel basics

There is the `$error` sentinel (variable type related to Falcor), which is generally an approach to returning errors.

Generally, as you should already know, Falcor batches requests. Thanks to them, you can fetch data from different falcor-routes in one HTTP request. The following example is what you can fetch in one go:

- **One dataset:** Complete and ready to retrieve
- **Second dataset:** Second dataset, may contain an error

We don't want to influence the fetching process of one dataset when there is an error in the second dataset (you need to remember that the two datasets from our example are fetched in one request).



Useful parts from the documentation that may help you understand error handling in Falcor are available here:

<https://netflix.github.io/falcor/doc/Model.html#~errorSelector>

<https://netflix.github.io/falcor/documentation/model.html#error-handling>

<http://netflix.github.io/falcor/documentation/router.html> (search for `$error` on this page to find more examples from the documentation)

DRY error management on the client side

Let's start with improvements to the `CoreLayout` (`src/layouts/CoreLayout.js`). Under `AppBar`, import a new `snackbar` component with this:

```
import AppBar from 'material-ui/lib/app-bar';
import Snackbar from 'material-ui/lib/snackbar';
```

Then, under the imports, outside the `CoreLayout`, create a new function and export it:

```
let errorFuncUtil = (errMsg, errPath) => {
}
export { errorFuncUtil as errorFunc };
```

Then find the `CoreLayout` constructor to change it to use the exported function called `errorFuncUtil` as a callback in the base in case of an error returned by the Falcor `$error` sentinel:

```
// old constructor
constructor(props) {
  super(props);
}
```

Here's the new one:

```
constructor(props) {
  super(props);
  this.state = {
    errorValue: null
  }

  if (typeof window !== 'undefined') {
    errorFuncUtil = this.handleFalcorErrors.bind(this);
  }
}
```

As you can find here, we have introduced a new `errorValue` state (the default state is `null`). Then, on the frontend only (because of `if(typeof window !== 'undefined')`), we assign `this.handleErrors.bind(this)` to our `errorFuncUtil`.

As you will find in a moment, this is so because the exported `errorFuncUtil` will be imported in our `falcorModel.js`, where we will use the best possible DRY way to inform our `CoreLayout` about any error occurring on the backend with Falcor. The great thing about this is that we will implement it just once, but it will be a universal way of informing our client-side app users of any errors (and it will also save us development effort in the future, as any error will be handled by the approach that we are implementing now).

We need to add a new function to our `CoreLayout` called `handleFalcorErrors`:

```
handleFalcorErrors(errMsg, errPath) {
  let errorValue = &grave;Error: ${errMsg} (path
  ${JSON.stringify(errPath)})&grave;;
  this.setState({errorValue});
}
```

The `handleFalcorErrors` function is setting the new state of our error. We will compose our error for the user with an `errMsg` (we create this on the backend, as you will learn in a moment) and the `errPath` (optional, but this is the `falcor-route` path where the error has occurred).

Okay, we have everything in place; the only thing missing from the `CoreLayout` function is the improved render. The new render of the `CoreLayout` is as follows:

```
render () {
  let errorSnackbarJSX = null;
  if (this.state.errorValue) {
    errorSnackbarJSX = <Snackbar
      open={true}
```

```
      message={this.state.errorValue}
      autoHideDuration={8000}
      onRequestClose={ () => console.log('You can add custom
        onClose code')} } />;
    }

    const buttonStyle = {
      margin: 5
    };
    const homeIconStyle = {
      margin: 5,
      paddingTop: 5
    };

    let menuLinksJSX;
    let userIsLoggedIn = typeof localStorage !== 'undefined' &&
      localStorage.token && this.props.routes[1].name !== 'logout';

    if (userIsLoggedIn) {
      menuLinksJSX = (
        <span>
          <Link to='/dashboard'>
            <RaisedButton label='Dashboard' style={buttonStyle} />
          </Link>
          <Link to='/logout'>
            <RaisedButton label='Logout' style={buttonStyle} />
          </Link>
        </span>);
    } else {
      menuLinksJSX = (
        <span>
          <Link to='/register'>
            <RaisedButton label='Register' style={buttonStyle} />
          </Link>
          <Link to='/login'>
            <RaisedButton label='Login' style={buttonStyle} />
          </Link>
        </span>);
    }

    let homePageButtonJSX = (
      <Link to='/'>
        <RaisedButton label={<ActionHome />}
          style={homeIconStyle} />
      </Link>);
    return (
      <div>
```

```
        {errorSnackbarJSX}
      <AppBar
        title='Publishing App'
        iconElementLeft={homePageButtonJSX}
        iconElementRight={menuLinksJSX} />
      <br/>
      {this.props.children}
    </div>

  );
}
```

As you can find here, the new parts are related to the Material-UI `snackbar` component. Take a look at this:

```
let errorSnackbarJSX = null;
if (this.state.errorValue) {
  errorSnackbarJSX = <Snackbar
    open={true}
    message={this.state.errorValue}
    autoHideDuration={8000} />;
}
```

This code snippet is preparing our `erroSnackbarJSX` and the following:

```
<div>
  {errorSnackbarJSX}
  <AppBar
    title='Publishing App'
    iconElementLeft={homePageButtonJSX}
    iconElementRight={menuLinksJSX} />
  <br/>
  {this.props.children}
</div>
```

Make sure `{errorSnackbarJSX}` is placed exactly the same way as in this example. Otherwise, you may find some problems during the app's test run. You now have completed everything related to the CoreLayout improvements.

Tweaks - FalcorModel.js on the frontend

In the `src/falcorModel.js` file, identify the following code:

```
// already in your codebase, old code:
import falcor from 'falcor';
import FalcorDataSource from 'falcor-http-datasource';
```

```
class PublishingAppDataSource extends FalcorDataSource {
  onBeforeRequest ( config ) {
    const token = localStorage.token;
    const username = localStorage.username;
    const role = localStorage.role;

    if (token && username && role) {
      config.headers['token'] = token;
      config.headers['username'] = username;
      config.headers['role'] = role;
    }
  }
}

const model = new falcor.Model({
  source: new PublishingAppDataSource('/model.json')
});
export default model;
```

This code has to be improved by adding a new option to the `falcor.Model`:

```
import falcor from 'falcor';
import FalcorDataSource from 'falcor-http-datasource';
import {errorFunc} from '../layouts/CoreLayout';

class PublishingAppDataSource extends FalcorDataSource {
  onBeforeRequest ( config ) {
    const token = localStorage.token;
    const username = localStorage.username;
    const role = localStorage.role;

    if (token && username && role) {
      config.headers['token'] = token;
      config.headers['username'] = username;
      config.headers['role'] = role;
    }
  }
}

let falcorOptions = {
  source: new PublishingAppDataSource('/model.json'),
  errorSelector: function(path, error) {
    errorFunc(error.value, path);
    error.$expires = -1000 * 60 * 2;
    return error;
  }
};
```

```
const model = new falcor.Model(falcorOptions);
export default model;
```

The first thing we added is an import of `errorFunc` to the top of that file:

```
import {errorFunc} from './layouts/CoreLayout';
```

Besides `errorFunc`, we have introduced the `falcorOptions` variable. The source stays the same as in the previous version. We have added `errorSelector`, which is run every time the client side calls the backend and the `falcor-router` on the backend returns an `$error` sentinel.

More details on the error selector can be found at

<https://netflix.github.io/falcor/documentation/model.html#the-errorselector-value>.

Backend implementation of the `$error` sentinel

We will perform the backend implementation in two steps:

1. An error example, just to test our client-side code.
2. After we are sure that the error handling is working correctly, we will secure the endpoints properly.

Testing our `$error`-related code

Let's start with imports in the `server/routes.js` file:

```
import configMongoose from './configMongoose';
import sessionRoutes from './routesSession';
import jsonGraph from 'falcor-json-graph';
import jwt from 'jsonwebtoken';
import jwtSecret from './configSecret';

let $ref = jsonGraph.ref;
let $atom = jsonGraph.atom;
let $error = jsonGraph.error;
const Article = configMongoose.Article;
```

The only new thing is that you need to import the `$error` sentinel from `falcor-json-graph`.



The goal of our `$error` test is to replace a working route that is responsible for fetching articles (`articles[{{integers}}]`). After we break this route, we will be able to test whether our frontend and backend setup is working. After we test the errors (refer to the next screenshot), we will delete this breaking `$error` code from `articles[{{integers}}]`. Read on for details.

Test it with the article route:

```
{
  route: 'articles[{{integers}}]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];

    return {
      path: ['articles'],
      value: $error('auth error')
    }

    return Article.find({}, '_id', function(err, articlesDocs) {
      return articlesDocs;
    }).then ((articlesArrayFromDB) => {
      let results = [];
      articlesIndex.forEach((index) => {
        let currentMongoID =
          String(articlesArrayFromDB[index]['_id']);
        let articleRef = $ref(['articlesById', currentMongoID]);

        const falcorSingleArticleResult = {
          path: ['articles', index],
          value: articleRef
        };

        results.push(falcorSingleArticleResult);
      });
      return results;
    })
  }
},
```

As you can see, this is only a test. We will improve this code in a moment, but let's test whether the text in the `$error('auth error')` sentinel will be shown to the user.

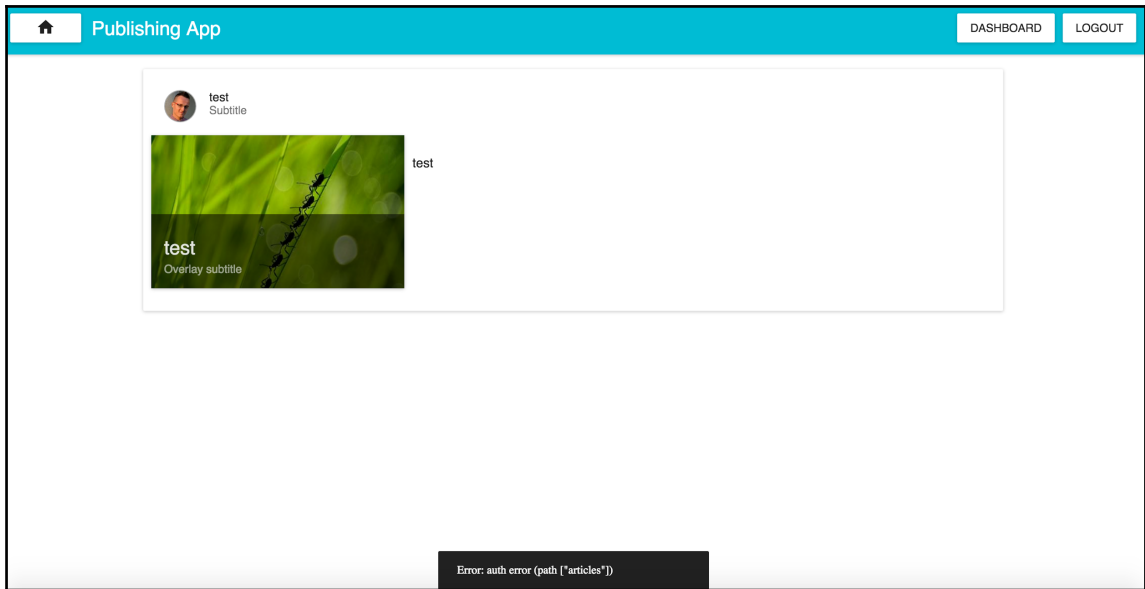
Run MongoDB:

```
$ mongod
```

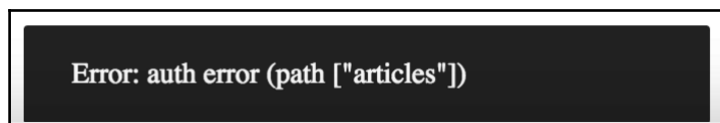
Then, run the server in another terminal:

```
$ npm start
```

After you run both these, point your browser to `http://localhost:3000`, and you will see for this error for 8 seconds:



As you can see, there is white text on a black background in the bottom of the window:



If you run the app, and on the main page you see the error message as on the screenshot, then it tells you that you are good!

Cleaning up \$error after a successful test

After you are sure that the error handling is working for you, you can replace the old code:

```
{
  route: 'articles[{{integers}}]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];

    return {
      path: ['articles'],
      value: $error('auth error')
    }
  }
  return Article.find({}, '_id', function(err, articlesDocs) {
```

Change it to the following, without error returning:

```
{
  route: 'articles[{{integers}}]',
  get: (pathSet) => {
    const articlesIndex = pathSet[1];
    return Article.find({}, '_id', function(err, articlesDocs) {
```

Now, the app will start working normally without throwing an error when you try to fetch an article from the backend.

Wrapping up the routes' security

We've already implemented some logic in `server/routes.js` that checks whether a user is authorized, with the following:

```
// this already is in your codebase:
export default ( req, res ) => {
  let { token, role, username } = req.headers;
  let userDetailsToHash = username+role;
  let authSignToken = jwt.sign(userDetailsToHash, jwtSecret.secret);
  let isAuthorized = authSignToken === token;
  let sessionObject = {isAuthorized, role, username};
  console.info(`&grave;The ${username} is authorized === &grave;,
isAuthorized);
```

In this code, you will find that we can create the following logic in the beginning of each role that requires authorization and the editor role:

```
// this is example of falcor-router $errors, don't write it:
if (isAuthorized === false) {
  return {
    path: ['HERE_GOES_THE_REAL_FALCOR_PATH'],
    value: $error('auth error')
  }
} elseif(role !== 'editor') {
  return {
    path: ['HERE_GOES_THE_REAL_FALCOR_PATH'],
    value: $error('you must be an editor in order
  to perform this action')
  }
}
```

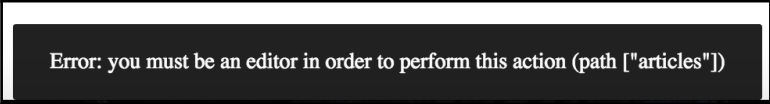
As you can see here, this is only an example (don't change it yet; we will implement it in a moment), with `path ['HERE_GOES_THE_REAL_FALCOR_PATH']`.

First, we check whether a user is authorized at all with `isAuthorized === false`; if not authorized, he will see an error (with the universal error mechanism that we just implemented):



Error: auth error (path ["articles"])

In future, we may have more roles in our publishing app, so in case someone isn't an editor, then he will see the following in the error:



Error: you must be an editor in order to perform this action (path ["articles"])

What routes to secure

For routes (`server/routes.js`) that require authorization in our application's articles, add the following:

```
route: 'articles.add',
```

Here's the old code:

```
// this is already in your codebase, old code:
{
  route: 'articles.add',
  call: (callPath, args) => {
    const newArticleObj = args[0];
    var article = new Article(newArticleObj);

    return article.save(function (err, data) {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
      else {
        return data;
      }
    }).then ((data) => {
// code has been striped out from here for the sake of brevity,
// nothing changes below
```

The new code with auth checks is as follows:

```
{
  route: 'articles.add',
  call: (callPath, args) => {
    if (sessionObject.isAuthenticated === false) {
      return {
        path: ['articles'],
        value: $error('auth error')
      }
    } else if (sessionObject.role !== 'editor' &&
      sessionObject.role !== 'admin') {
      return {
        path: ['articles'],
        value: $error('you must be an editor
          in order to perform this action')
      }
    }

    const newArticleObj = args[0];
    var article = new Article(newArticleObj);

    return article.save(function (err, data) {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
    })
  }
}
```

```
        else {
          return data;
        }
      }).then ((data) => {
// code has been striped out from here for
//the sake of brevity, nothing changes below
```

As you can find here, we have added two checks with `isAuthorized === false` and `role !== 'editor'`. The following routes content will be almost the same (just the path changes a little).

Here is the articles update:

```
route: 'articles.update',
```

This is the old code:

```
// this is already in your codebase, old code:
{
  route: 'articles.update',
  call: async (callPath, args) =>
  {
    const updatedArticle = args[0];
    let articleID = String(updatedArticle._id);
    let article = new Article(updatedArticle);
    article.isNew = false;

    return article.save(function (err, data) {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
    }).then ((res) => {
// code has been striped out from here for the
//sake of brevity, nothing changes below
```

The new code with the auth checks is as follows:

```
{
  route: 'articles.update',
  call: async (callPath, args) =>
  {
    if (sessionObject.isAuthorized === false) {
      return {
        path: ['articles'],
        value: $error('auth error')
      }
    } else if (sessionObject.role !== 'editor' &&
```

```
    sessionObject.role !== 'admin') {
      return {
        path: ['articles'],
        value: $error('you must be an editor
          in order to perform this action')
      }
    }

    const updatedArticle = args[0];
    let articleID = String(updatedArticle._id);
    let article = new Article(updatedArticle);
    article.isNew = false;

    return article.save(function (err, data) {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
    }).then((res) => {
// code has been striped out from here
//for the sake of brevity, nothing changes below

articles delete:
route: 'articles.delete',
```

Find this old code:

```
// this is already in your codebase, old code:

{
  route: 'articles.delete',
  call: (callPath, args) =>
  {
    let toDeleteArticleId = args[0];
    return Article.find({ _id: toDeleteArticleId }).remove((err) => {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
    }).then((res) => {
// code has been striped out from here
//for the sake of brevity, nothing changes below
```

Replace it with this new code with the auth checks:

```
{
  route: 'articles.delete',
  call: (callPath, args) =>
  {

    if (sessionObject.isAuthenticated === false) {
      return {
        path: ['articles'],
        value: $error('auth error')
      }
    } else if (sessionObject.role !== 'editor' &&
      sessionObject.role !== 'admin') {
      return {
        path: ['articles'],
        value: $error('you must be an
          editor in order to perform this action')
      }
    }

    let toDeleteArticleId = args[0];
    return Article.find({ _id: toDeleteArticleId }).remove((err) => {
      if (err) {
        console.info('ERROR', err);
        return err;
      }
    }).then((res) => {
      // code has been striped out from here
      //for the sake of brevity, nothing below changes
    })
  }
}
```

Summary

As you can see, the returns are almost the same--we can lower the code duplication. We can make a helper function for them so there will be less code, but you need to remember that you need to set a path similar to the one that you request when returning an error. For example, if you are on `articles.update`, then you need return an error in the article's path (or if you are on `XYZ.update`, then the error goes to the `XYZ` path).

In the next chapter, we will implement AWS S3 in order to have the ability to upload articles' cover photos. Besides that, we will generally improve our publishing application with new features.

6

AWS S3 for Image Upload and Wrapping Up Key Application Features

Currently we have an app that works but is missing some key features. Our focus for this chapter will include the following feature implementations/improvements:

- Opening a new AWS account
- Creating **Identity and Access Management (IAM)** for your AWS account
- Setting up an AWS S3 bucket
- Adding the ability to upload a photo for an article (add and edit article covers)
- Adding the ability to set up a title, subtitle, and "overlay subtitle" (on the add/edit article views)

Articles on the dashboard currently have HTML in the content; we need to improve that:



We need to finish this stuff. After we are done with these improvements, we will do some refactoring.

AWS S3 - an introduction

Amazon's AWS S3 is a simple storage service for static assets such as images on Amazon's servers. It helps you host safe, secure, and highly scalable objects (as images) in the cloud.

This approach of storing static assets online is quite convenient and easy--this is why we will use it throughout our book.

We will use it in our application, as it gives us many scalability features that wouldn't be so easy to access when hosting image assets on our own Node.js server.

In general, Node.js shouldn't be used for hosting assets larger than what we use it for now. Don't even think of implementing an image-upload mechanism (not recommended at all) on the Node.js server--we will employ Amazon's services for that.

Generating keys (access key ID and secret key)

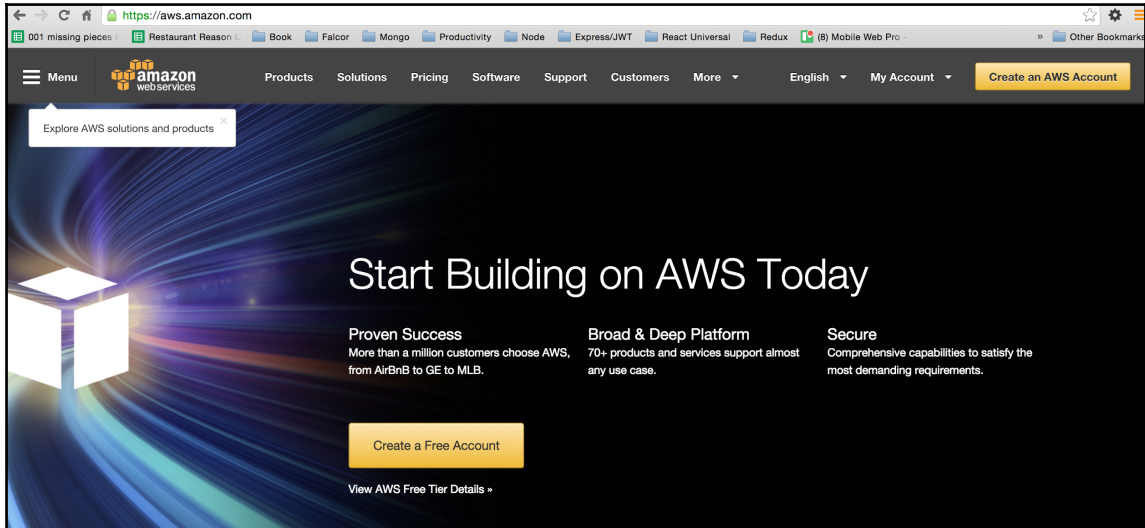
Before we start adding a new S3 bucket, we need to generate keys for our AWS account (`accessKeyId` and `secretAccessKey`).

An example set of details that we will need to keep in our Node.js app is as follows:

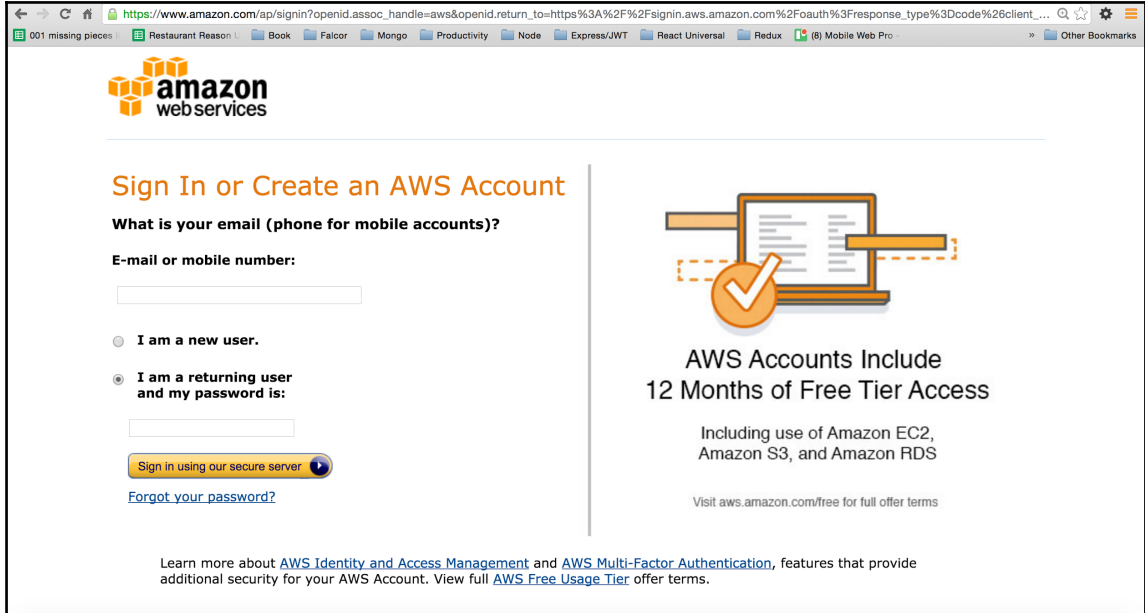
```
const awsConfig = {
  accessKeyId: 'EXAMPLE_LB7XH_KEY_BGTCA',
  secretAccessKey: 'ExAMpLe+KEY+FYliI9J1nvky5g2bInN26TCU+FiY',
  region: 'us-west-2',
  bucketKey: 'your-bucket-name-'
};
```

What is a bucket in Amazon S3? A **bucket** is a kind of namespace for files that you have in Amazon S3. You can have several buckets associated with different projects. As you can see, our next steps will be creating the `accessKeyId` and `secretAccessKey` associated with your account and `bucketKey` (kind of a namespace for the pictures for our articles). Define a region where you want to keep the files physically. If your project has a target specified for a location, it will speed up the loading of images and, in general, limit the latency because an image will be hosted closer to the client/user of our publishing application.

To create an AWS account, go to <https://aws.amazon.com/>:



Create an account or sign in to your account:



The next step is to create the IAM, described in detail in the next section.

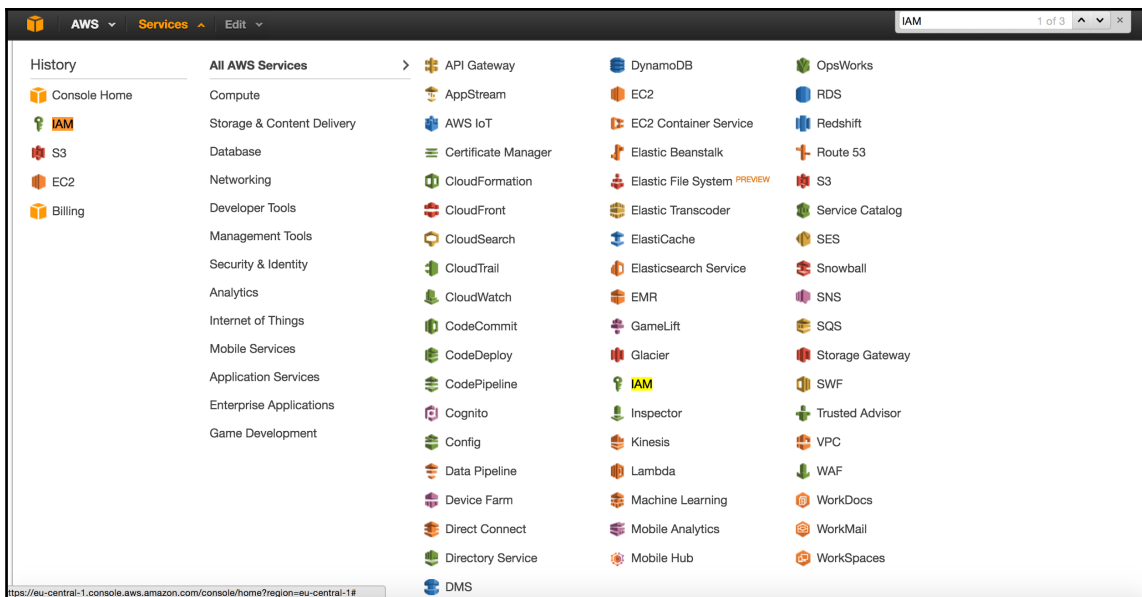


Regarding AWS creation

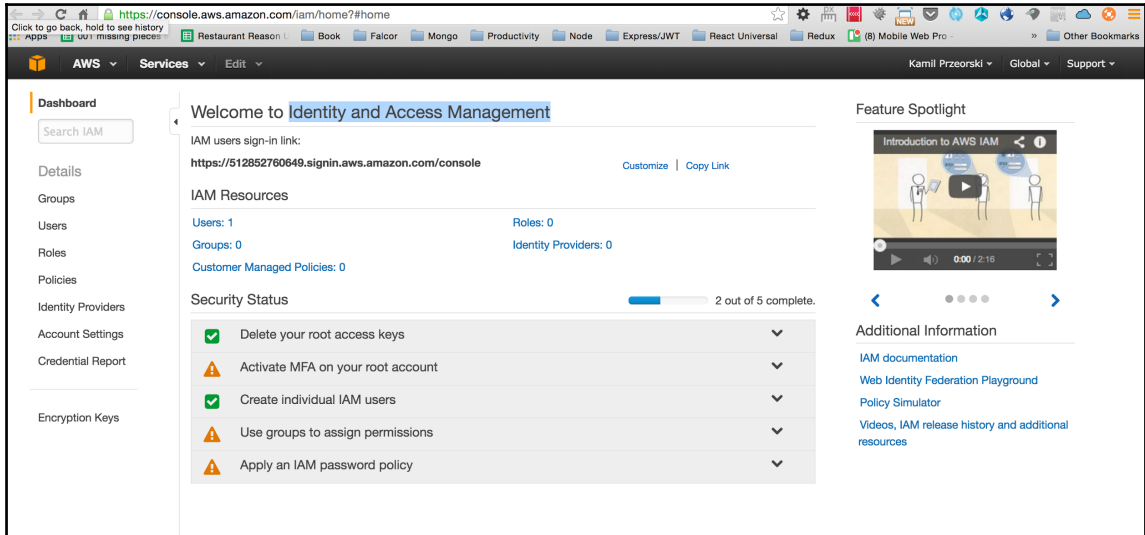
After you create an account for a certain region, if you want to create an S3 bucket, you need to choose the same region your account is assigned to; otherwise, you may have problems while setting up S3 in the following pages.

IAM

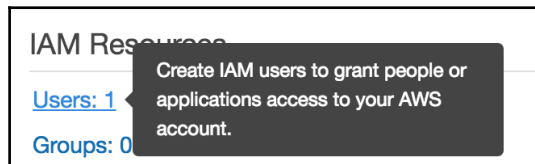
Let's prepare our new `accessKeyId` and `secretAccessKey`. You need to visit the IAM page in your Amazon console. You can find it in the **Services** list:



The IAM page looks like this (<https://console.aws.amazon.com/iam/home?#home>):



Go to **IAM Resources | Users**:



On the next page, you will see a button; click on it:



After clicking, you will see a form. Fill it in with at least one user, as in this screenshot (the screenshots are giving you the exact steps that you must accomplish, even if AWS's UX has been changed in the meantime):

Create User

Enter User Names:

1. publishingapp
- 2.
- 3.
- 4.
- 5.

Maximum 64 characters each

☒ Generate an access key for each user

Users need access keys to make secure REST or Query protocol requests to AWS service APIs.
For users who need access to the AWS Management Console, create a password in the Users panel after completing this wizard.

Cancel Create


After clicking on the **Create** button, copy the keys to a safe place (we will use them in a moment):

☒ Your 1 User(s) have been created successfully.

This is the last time these User security credentials will be available for download.

You can manage and recreate these credentials any time.

[Hide User Security Credentials](#)

 publishingapp	
Access Key ID:	AKIAI3Y54WVG5JM4VUHA
Secret Access Key:	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

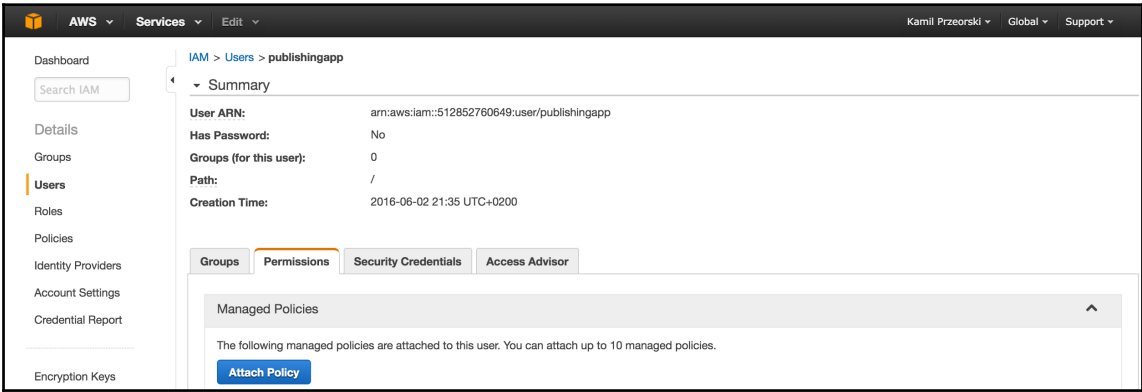


Do not forget to copy the keys (access key ID and secret access key). You will learn where to put them in the code in order to use S3 services later in the book. Of course, the ones in the screenshot aren't active. They are only examples; you need to have your own.

Setting up S3 permissions for the user

The last thing is to add AmazonS3FullAccess permissions with the following steps:

- 1. Go to the **Permissions** tab:

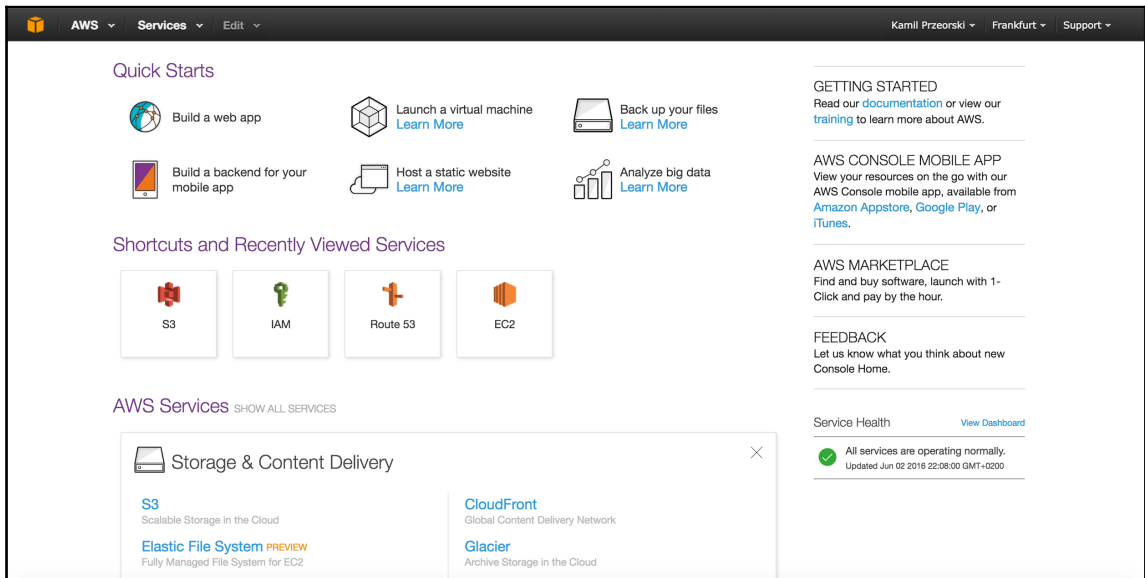


- 2. Click on **Attach Policy** and choose **AmazonS3FullAccess**. After attaching it, it will be listed as in the following example:

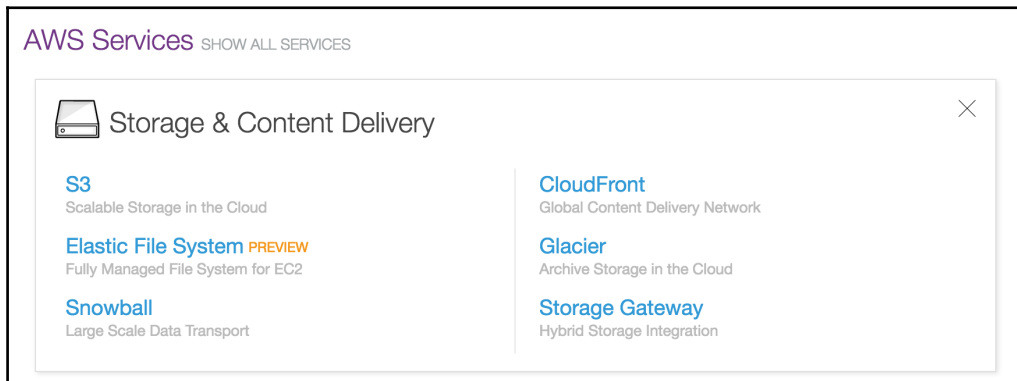


Now we'll move on to creating a new bucket for the image files.

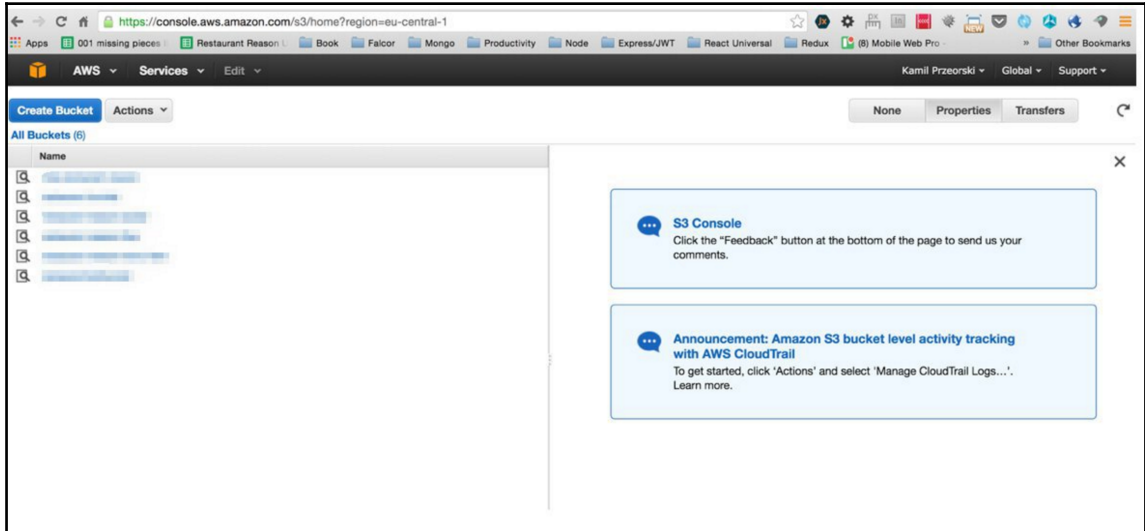
1. You are done with the keys and you have granted the S3 policy for the keys; now, we need to prepare the bucket that will keep the images. First of all, you need to go to the AWS console main page, which looks as follows (<https://console.aws.amazon.com/console/home>):



2. You will see something like **AWS Services SHOW ALL SERVICES** (alternatively, find it from the **Services** list the same way as **IAM**):

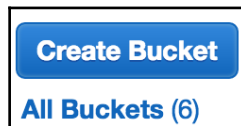


3. Click on **S3 - Scalable Storage in the Cloud** (as in the previous screenshot). After that, you will see a view similar to this (I have six buckets; you will have zero when you have a new account):

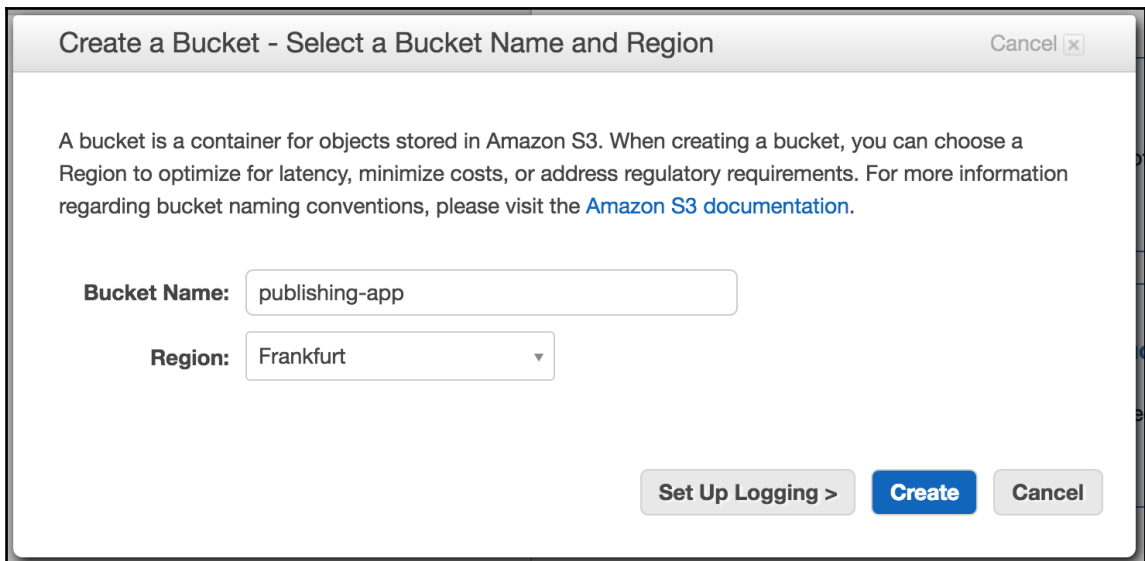


In that bucket, we will keep the static images of our articles (you will learn how exactly in the coming pages).

4. Create a bucket by clicking on the **Create Bucket** button:



5. Choose the **publishing-app** name (or another that works for you).



The screenshot shows a dialog box titled "Create a Bucket - Select a Bucket Name and Region" with a "Cancel" button in the top right corner. Below the title, there is explanatory text: "A bucket is a container for objects stored in Amazon S3. When creating a bucket, you can choose a Region to optimize for latency, minimize costs, or address regulatory requirements. For more information regarding bucket naming conventions, please visit the [Amazon S3 documentation](#)." Below this text, there are two input fields: "Bucket Name:" with the value "publishing-app" and "Region:" with a dropdown menu showing "Frankfurt". At the bottom right, there are three buttons: "Set Up Logging >", "Create" (in blue), and "Cancel".

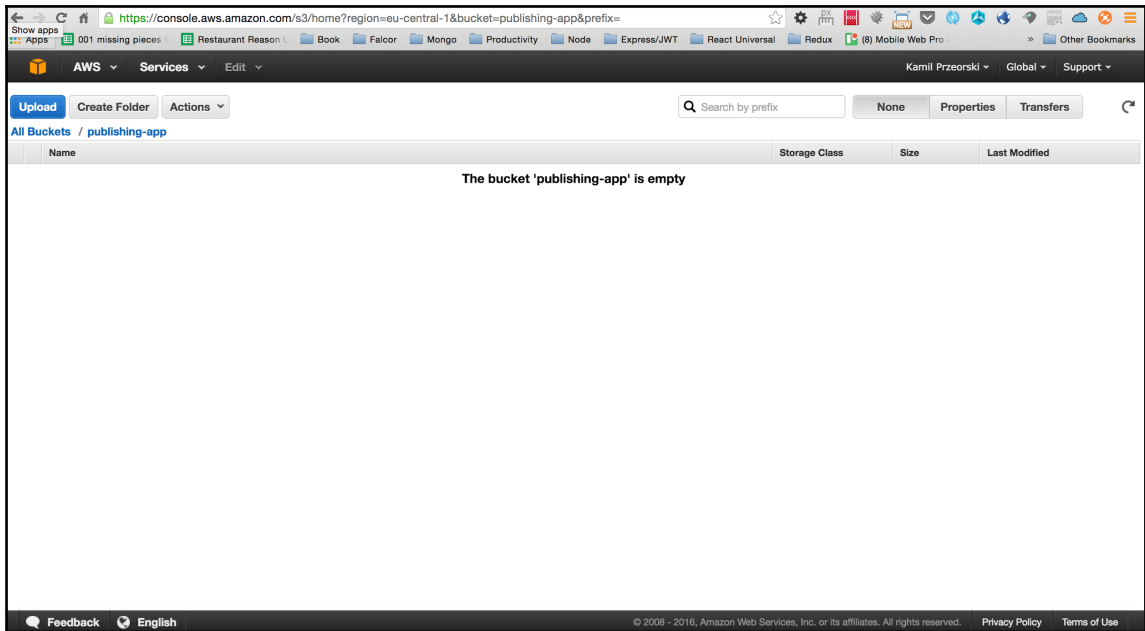


In the screenshot, we have chosen **Frankfurt**. But if, for example, when you create an account and your URL shows "`?region=us-west-2`", then choose **Oregon**. It's important to create the S3 bucket in the region that you have assigned your account to.

6. After the bucket has been created, click on it from the bucket list:



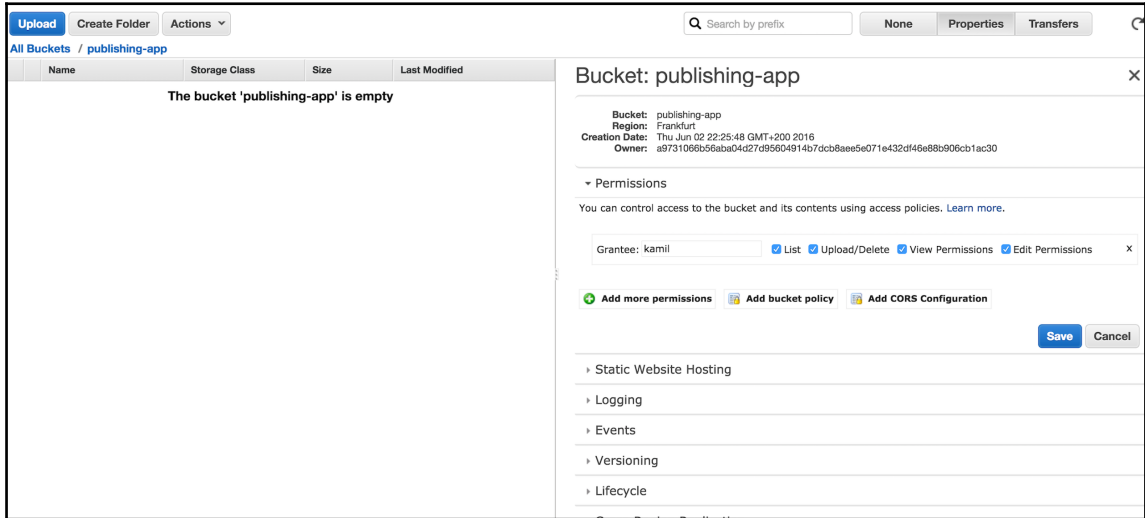
7. The empty bucket with the **publishing-app** name will look as follows:



8. When you are in this view, the URL in the browser tells you the exact region and bucket (so you can use it later when performing configuration on the backend):

```
// just an example link to the bucket
https://console.aws.amazon.com/s3/home?region=eu-central-
1&bucket=publishing-app&prefix=
```

9. The last thing is to make sure that the **CORS** configuration for the **publishing-app** bucket is correct. Click on the **Properties** tab in that view, and you will get a detailed view of it:



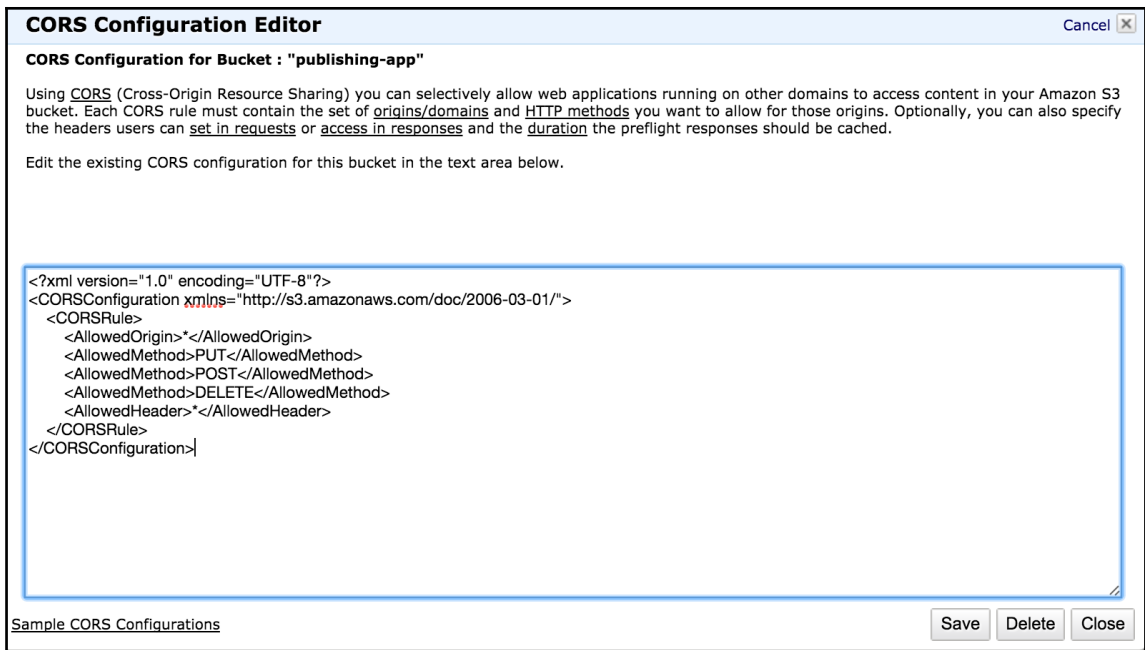
10. Then, click on the **Add CORS** button:



11. After that, paste the following into the text area (the following is the cross-origin resource sharing definition; it defines a way for the Pub app which is loaded in one domain to interact with resources in a different domain within the AWS services):

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com
/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>*</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <MaxAgeSeconds>3000</MaxAgeSeconds>
    <AllowedHeader>*</AllowedHeader>
  </CORSRule>
</CORSConfiguration>
```

12. It will now look like the following example:



The screenshot shows the 'CORS Configuration Editor' window for the bucket 'publishing-app'. It includes a 'Cancel' button in the top right. The main text explains that CORS (Cross-Origin Resource Sharing) allows web applications on other domains to access content in an Amazon S3 bucket, and that each rule must specify origins, HTTP methods, and headers. Below this, it says to edit the existing configuration in the text area. The text area contains the following XML code:

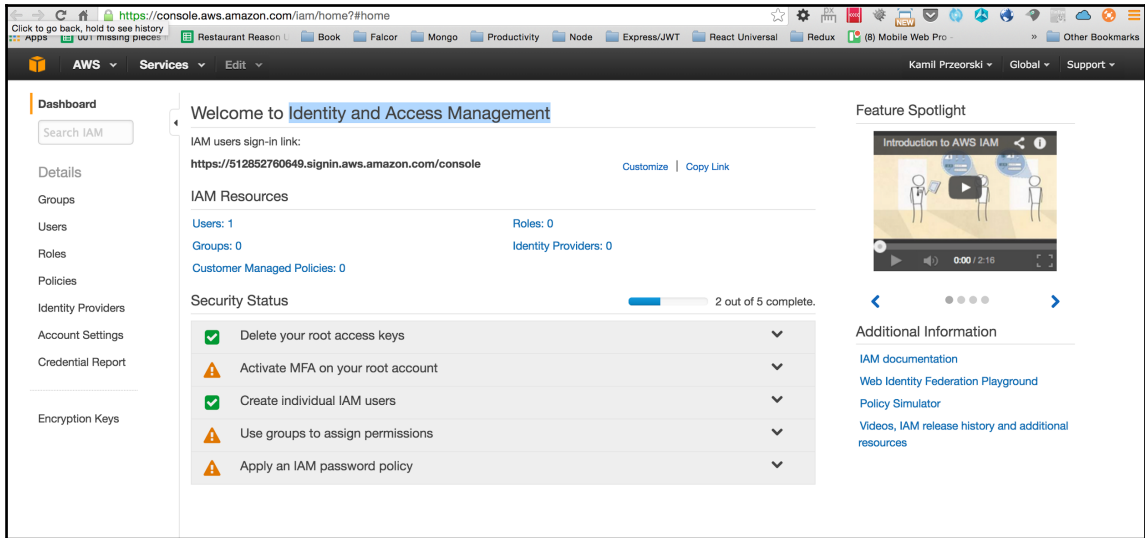
```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>*</AllowedOrigin>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>DELETE</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
  </CORSRule>
</CORSConfiguration>
```

At the bottom left, there is a link for 'Sample CORS Configurations'. At the bottom right, there are three buttons: 'Save', 'Delete', and 'Close'.

13. Click on the **Save** button. After all the steps are done, we can start with coding the image upload feature.

Coding the image upload feature in the AddArticleView

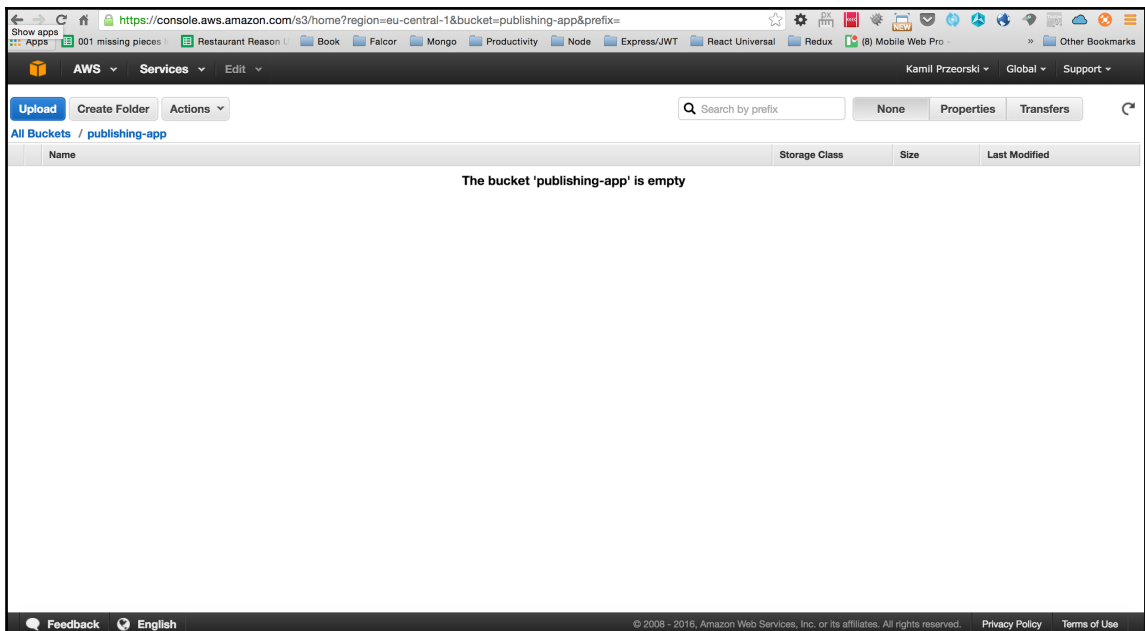
Before you are able to continue, you need to have your access details for your S3 bucket that you created in the previous pages. `AWS_ACCESS_KEY_ID` comes from a previous subsection, where we created a user while being in that view:



`AWS_SECRET_ACCESS_KEY` is the same as the AWS access key (as you can already guess by the name itself). `AWS_BUCKET_NAME` is the name of your bucket (in our book, we've called it **publishing-app**). For `AWS_REGION_NAME`, we will use `eu-central-1`.



The easiest way to find `AWS_BUCKET_NAME` and `AWS_REGION_NAME` is to look at the URL while you are in that view (described in the previous subsection).



Check the browser's URL in that view:

`https://console.aws.amazon.com/s3/home?region=eu-central-1#&bucket=publishing-app&prefix=`

The region and bucket names are clearly in that URL (I want to make this very clear as your region and bucket name can be different, depending on where you live).



Also, make sure that your CORS are set up correctly and your permissions/attach policy is done exactly as described above. Otherwise, you can have problems with everything described in the following subsections.

Environment variables in Node.js

We will pass all four parameters (`AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, `AWS_BUCKET_NAME`, and `AWS_REGION_NAME`) via the node's environment variables.

First, let's install a node library that will create environment variables from a file so that we will be able to use them within our localhost:

```
npm i -save node-env-file@0.1.8
```

What are these environment variables? In general, we will use them to pass the variables of some sensitive data to the app--we are talking here specifically about AWS secret keys and MongoDB's login/password information for the current environment setup (if it is development or production).

You can read those environment variables via accessing them, like in the following examples:

```
// this is how we will access the variables in
//the server.js for example:
env.process.AWS_ACCESS_KEY_ID
env.process.AWS_SECRET_ACCESS_KEY
env.process.AWS_BUCKET_NAME
env.process.AWS_REGION_NAME
```

In our local development environment, we will keep that information in the server's directory, so do this from your command prompt:

```
$ [[you are in the server/ directory of your project]]
$ touch .env
```

You have created a `server/.env` file; the next step is to put content in it (from this file, the `node-env-file` will read the environment variables when our app is running):

```
AWS_ACCESS_KEY_ID=*_*_*_*_ACCESS_KEY_HERE*_*_*_*_
AWS_SECRET_ACCESS_KEY=*_*_*_*_SECRET_KEY_HERE*_*_*_*_
AWS_BUCKET_NAME=publishing-app
AWS_REGION_NAME=eu-central-1
```

Here, you can see the structure of a node's environment file. Each new line comes with a key and value. There, you need to paste the keys you created while reading this chapter. Replace those values with your own: `*_*_*_*_*_ACCESS_KEY_HERE*_*` and `*_*_*_*_*_SECRET_KEY_HERE*_*`.

After you have created the `server/.env` file, install the required dependency that will abstract the whole gig with the image upload; use `npm` for this when in the project's directory:

```
npm i --save react-s3-uploader@3.0.3
```

The `react-s3-uploader` component works quite well for our use case, and it abstracts the `aws-sdk` features for us quite well. The main point here is that we need to have configured the `.env` file well (with the correct variables) and the `react-s3-uploader` will do the job at the backend and frontend for us (as you will see soon).

Improving our Mongoose article schema

We need to improve the schema, so we will have a place in our article collection for storing the URL of an image. Edit the old article schema:

```
// this is old codebase to improve:
var articleSchema = new Schema({
  articleTitle: String,
  articleContent: String,
  articleContentJSON: Object
},
{
  minimize: false
})
);
```

Change it to the new, improved version:

```
var articleSchema = new Schema({
  articleTitle: String,
  articleContent: String,
  articleContentJSON: Object,
  articlePicUrl: { type: String, default:
    '/static/placeholder.png' }
},
{
  minimize: false
})
);
```

As you can see, we have introduced the `articlePicUrl` with a default value of `/static/placeholder.png`. Now, we will be able to save an article with a picture's URL variable in the article's object.



If you forgot to update the Mongoose model, then it won't let you save that value into the database.

Adding routes for S3's upload

We need to import one new library into the `server/server.js` file:

```
import s3router from 'react-s3-uploader/s3router';
```

We'll end up with something like the following:

```
// don't write it, this is how your server/server.js
//file should look like:
import http from 'http';
import express from 'express';
import cors from 'cors';
import bodyParser from 'body-parser';
import falcor from 'falcor';
import falcorExpress from 'falcor-express';
import FalcorRouter from 'falcor-router';
import routes from './routes.js';

import React from 'react'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
import { renderToStaticMarkup } from 'react-dom/server'
import ReactRouter from 'react-router';
import { RoutingContext, match } from 'react-router';
import * as hist from 'history';
import rootReducer from '../src/reducers';
import reactRoutes from '../src/routes';
import fetchServerSide from './fetchServerSide';

import s3router from 'react-s3-uploader/s3router';

var app = express();
app.server = http.createServer(app);

// CORS - 3rd party middleware
app.use(cors());

// This is required by falcor-express middleware
// to work correctly with falcor-browser
app.use(bodyParser.json({extended: false}));
app.use(bodyParser.urlencoded({extended: false}));
```

I'm putting all this here so you can make sure that your `server/server.js` file matches this.

One more thing to do is to modify the `server/index.js` file. Find this:

```
require('babel-core/register');
require('babel-polyfill');
require('./server');
```

Change it to the following improved version:

```
var env = require('node-env-file');
// Load any undefined ENV variables from a specified file.
env(__dirname + './.env');

require('babel-core/register');
require('babel-polyfill');
require('./server');
```

Just for clarification, `env(__dirname + './.env');` is telling us the location of the `.env` file in our structure (you can find from `console.log` that the `__dirname` variable is a system location of a server's file--this must match the real `.env` file's location so it can be found by the system).

The next part is to add this to our `server/server.js` file:

```
app.use('/s3', s3router({
  bucket: process.env.AWS_BUCKET_NAME,
  region: process.env.AWS_REGION_NAME,
  signatureVersion: 'v4',
  headers: {'Access-Control-Allow-Origin': '*'},
  ACL: 'public-read'
}));
```

As you can see here, we have started using the environment variable that we defined in the `server/.env` file. For me, the `process.env.AWS_BUCKET_NAME` is equal to `publishing-app`, but if you have defined it differently, then it will retrieve another value from `server/.env` (thanks to the `env` express middleware that we just defined).

Based on that backend configuration (environment variables and setting up `s3router` with `import s3router from 'react-s3-uploader/s3router'`), we will be able to use the AWS S3 bucket. We need to prepare the frontend, which first will be implemented on the `add-an-article` view.

Creating the ImgUploader component on the frontend

We will create a dump component called `ImgUploader`. This component will use the `react-s3-uploader` library, which does the job of abstracting the upload to Amazon S3. On a callback, you receive `information:onProgress`, and you can find the progress in percent with that callback, so a user can see the status of an `uploadonError`. This callback is fired when an error occurs `onFinish`: this callback sends us back the location of a file that has been uploaded to S3.

You will learn more details further in the chapter; let's create a file first:

```
$ [[you are in the src/components/articles directory of your
project]]
$ touch ImgUploader.js
```

You have created the `src/components/articles/ImgUploader.js` file, and the next step is to prepare the imports. So to the top of the `ImgUploader` file, add the following:

```
import React from 'react';
import ReactS3Uploader from 'react-s3-uploader';
import {Paper} from 'material-ui';

class ImgUploader extends React.Component {
  constructor(props) {
    super(props);
    this.uploadFinished = this.uploadFinished.bind(this);

    this.state = {
      uploadDetails: null,
      uploadProgress: null,
      uploadError: null,
      articlePicUrl: props.articlePicUrl
    };
  }

  uploadFinished(uploadDetails) {
    // here will be more code in a moment
  }

  render () {
    return <div>S3 Image uploader placeholder</div>;
  }
}
```

```
ImgUploader.propTypes = {  
  updateImgUrl: React.PropTypes.func.isRequired  
};  
export default ImgUploader;
```

As you can see here, we have initiated the `ImgUploader` component with `div` that returns a temporary placeholder in the render function.

We have also prepared `propTypes` with a required property called `updateImgUrl`. This will be a callback function that will send a final, uploaded image's location (which has to be saved in the database--we will use this `updateImgUrl` props in a moment).

In the state of that `ImgUploader` component, we have the following:

```
// this is already in your codebase:  
this.state = {  
  uploadDetails: null,  
  uploadProgress: null,  
  uploadError: null,  
  articlePicUrl: props.articlePicUrl  
};
```

In these variables, we will store all the states of our components, depending on the current status and `props.articlePicUrl`, and we'll send the URL details up to the `AddArticleView` component (we will do it later in the chapter, after finishing the `ImgUploader` component).

Wrapping up the `ImgUploader` component

The next step is to improve the `uploadFinished` function in our `ImgUploader`, so find the old empty function:

```
uploadFinished(uploadDetails) {  
  // here will be more code in a moment  
}
```

Replace it with the following:

```
uploadFinished(uploadDetails) {  
  let articlePicUrl = '/s3/img/'+uploadDetails.filename;  
  this.setState({  
    uploadProgress: null,  
    uploadDetails: uploadDetails,  
    articlePicUrl: articlePicUrl  
  });  
};
```

```
    this.props.updateImgUrl(articlePicUrl);  
  }  
}
```

As you can see, the `uploadDetails.filename` variable comes from the `ReactS3Uploader` component, which we have imported on top of the `ImgUploader` file. After a successful upload, we set the `uploadProgress` back to null, set the details of our upload, and send back the details via the callback using `this.props.updateImgUrl(articlePicUrl)`.

The next step is to improve our render function in `ImgUploader`:

```
render () {  
  let imgUploadProgressJSX;  
  let uploadProgress = this.state.uploadProgress;  
  if(uploadProgress) {  
    imgUploadProgressJSX = (  
      <div>  
        {uploadProgress.uploadStatusText}  
        ({uploadProgress.progressInPercent}%)  
      </div>  
    );  
  } else if(this.state.articlePicUrl) {  
    let articlePicStyles = {  
      maxWidth: 200,  
      maxHeight: 200,  
      margin: 'auto'  
    };  
    imgUploadProgressJSX = <img src={this.state.articlePicUrl}  
      style={articlePicStyles} />;  
  }  
  
  return <div>S3 Image uploader placeholder</div>;  
}
```

This render is incomplete, but let's describe what we have added so far. The code is simply all about getting information about `uploadProgress` via `this.state` (the first `if` statement). The `else if(this.state.articlePicUrl)` is all about rendering the image after the upload is complete. Okay, but where we will get that information? Here is the rest:

```
let uploaderJSX = (  
  <ReactS3Uploader  
    signingUrl='/s3/sign'  
    accept='image/*'  
    onProgress={(progressInPercent, uploadStatusText) => {  
      this.setState({  
        uploadProgress: { progressInPercent,  
          uploadStatusText },  
      }  
    }  
  )  
);
```

```
        uploadError: null
      });
    }
    onError={ (errorDetails) => {
      this.setState({
        uploadProgress: null,
        uploadError: errorDetails
      });
    }}
    onFinish={ (uploadDetails) => {
      this.uploadFinished(uploadDetails);
    }} />
  );
```

The `uploaderJSX` variable is the exact same as our `react-s3-uploader` library. As you can see from the code, for progress, we set the state with `uploadProgress: { progressInPercent, uploadStatusText }` and we set up `uploadError: null` (in case the user receives an error message). On error, we set the state, so we can tell the user. On finish, we run the `uploadFinished` function, which was described in detail previously.

The complete render function of `ImgUploader` will look as follows:

```
render () {
  let imgUploadProgressJSX;
  let uploadProgress = this.state.uploadProgress;
  if(uploadProgress) {
    imgUploadProgressJSX = (
      <div>
        {uploadProgress.uploadStatusText}
        ({uploadProgress.progressInPercent}%)
      </div>
    );
  } else if(this.state.articlePicUrl) {
    let articlePicStyles = {
      maxWidth: 200,
      maxHeight: 200,
      margin: 'auto'
    };
    imgUploadProgressJSX = <img src={this.state.articlePicUrl}
      style={articlePicStyles} />;
  }

  let uploaderJSX = (
    <ReactS3Uploader
      signingUrl='/s3/sign'
      accept='image/*'
      onProgress={ (progressInPercent, uploadStatusText) => {
```

```
        this.setState({
          uploadProgress: { progressInPercent,
            uploadStatusText },
          uploadError: null
        });
      }
      onError={ (errorDetails) => {
        this.setState({
          uploadProgress: null,
          uploadError: errorDetails
        });
      }
    }
    onFinish={ (uploadDetails) => {
      this.uploadFinished(uploadDetails);
    } } />
  );
}

return (
  <Paper zDepth={1} style={{padding: 32, margin: 'auto',
    width: 300}}>
    {imgUploadProgressJSX}
    {uploaderJSX}
  </Paper>
);
}
```

As you can see, this is the whole render of `ImgUploader`. We use an inline-styled `Paper` component (from `material-ui`), so the whole thing will look better to an article's end user/editor.

AddArticleView improvements

We need to add the `ImgUploader` component to `AddArticleView`; first, we need to import it into the `src/views/articles/AddArticleView.js` file, like this:

```
import ImgUploader from '../../components/articles/ImgUploader';
```

Next, in the constructor of `AddArticleView`, find this old code:

```
// this is old, don't write it:
class AddArticleView extends React.Component {
  constructor(props) {
    super(props);
    this._onDraftJSChange = this._onDraftJSChange.bind(this);
    this._articleSubmit = this._articleSubmit.bind(this);
```



```
    this.state = {
      title: 'test',
      contentJSON: {},
      htmlContent: '',
      newArticleID: null
    };
  }
}
```

Change it to the following improved version:

```
class AddArticleView extends React.Component {
  constructor(props) {
    super(props);
    this._onDraftJSChange = this._onDraftJSChange.bind(this);
    this._articleSubmit = this._articleSubmit.bind(this);
    this.updateImgUrl = this.updateImgUrl.bind(this);

    this.state = {
      title: 'test',
      contentJSON: {},
      htmlContent: '',
      newArticleID: null,
      articlePicUrl: '/static/placeholder.png'
    };
  }
}
```

As you can see, we have bound `this` to the `updateImgUrl` function and added a new state variable called `articlePicUrl` (by default, we will point to `/static/placeholder.png` in case a user doesn't choose a cover).

Let's improve the functions of this component:

```
// this is old codebase, just for your reference:
async _articleSubmit() {
  let newArticle = {
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON
  }

  let newArticleID = await falcorModel
    .call(
      'articles.add',
      [newArticle]
    )
    .then((result) => {
      return falcorModel.getValue(
        ['articles', 'newArticleID']
      )
    })
}
```

```
        ).then((articleID) => {
            return articleID;
        });
    });

    newArticle['_id'] = newArticleID;
    this.props.articleActions.pushNewArticle(newArticle);
    this.setState({ newArticleID: newArticleID });
}
```

Change this code to the following:

```
async _articleSubmit() {
    let newArticle = {
        articleTitle: this.state.title,
        articleContent: this.state.htmlContent,
        articleContentJSON: this.state.contentJSON,
        articlePicUrl: this.state.articlePicUrl
    }

    let newArticleID = await falcorModel
        .call(
            'articles.add',
            [newArticle]
        )
        .then((result) => {
            return falcorModel.getValue(
                ['articles', 'newArticleID']
            ).then((articleID) => {
                return articleID;
            });
        });

    newArticle['_id'] = newArticleID;
    this.props.articleActions.pushNewArticle(newArticle);
    this.setState({ newArticleID: newArticleID });
}

updateImgUrl(articlePicUrl) {
    this.setState({
        articlePicUrl: articlePicUrl
    });
}
```

As you can see, we have added `articlePicUrl: this.state.articlePicUrl` to the `newArticle` object. We have also introduced a new function called `updateImgUrl`, which is simply a callback that sets a new state with the `articlePicUrl` variable (in `this.state.articlePicUrl`, we keep the image URL of the current article that is going to be saved to the database).

The only thing to improve in `src/views/articles/AddArticleView.js` is our current render. Here is the old one:

```
// your current old codebase to improve:
return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Add Article</h1>
    <WYSIWYGeditor
      name='addarticle'
      title='Create an article'
      onChangeTextJSON={this._onDraftJSChange} />
    <RaisedButton
      onClick={this._articleSubmit}
      secondary={true}
      type='submit'
      style={{margin: '10px auto', display: 'block',
        width: 150}}
      label={'Submit Article'} />
  </div>
);
```

We need to improve this code using `ImgUploader`:

```
return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Add Article</h1>
    <WYSIWYGeditor
      name='addarticle'
      title='Create an article'
      onChangeTextJSON={this._onDraftJSChange} />

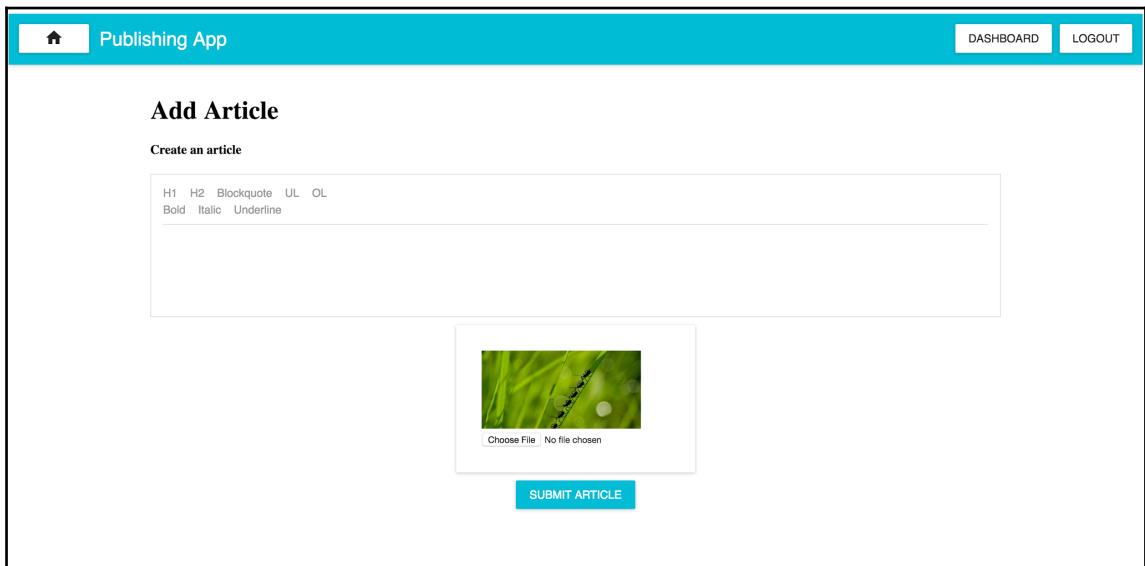
    <div style={{margin: '10px 10px 10px 10px'}}>
      <ImgUploader
        updateImgUrl={this.updateImgUrl}
        articlePicUrl={this.state.articlePicUrl} />
    </div>

    <RaisedButton
      onClick={this._articleSubmit}
      secondary={true}
```

```
        type='submit'
        style={{margin: '10px auto', display: 'block',
        width: 150}}
        label={'Submit Article'} />
    </div>
  );
}
```

You can see that we use the properties for sending down the current `articlePicUrl` (this will be handy later and also give us the default `placeholder.png` location) and the callback to update the `img` URL, called `updateImgUrl`.

If you visit <http://localhost:3000/add-article>, you will see a new image picker between the WYSIWYG box and the **SUBMIT ARTICLE** button (check the screenshot):



Of course, if you followed all the instructions correctly, after clicking on **Choose File**, you will be able to upload a new image to the S3 bucket, and the image in the thumbnail will be replaced, as in the following example:


Add Article

Create an article

H1 H2 Blockquote UL OL
Bold Italic Underline

An article about dogs

Text goes here



Choose File Screen Shot...3.47.11.png

SUBMIT ARTICLE

As you can see, we can upload an image. The next step is to unmock viewing them so we can see that our article has a dog on the cover (and the dog's image comes from our article collection in the DB).

Some remaining tweaks for PublishingApp, ArticleCard, and DashboardView

We can add an article. We need to unmock the image URLs in our views so we can see the real URL from the database (instead mocked in an `img src` property).

Let's start with `src/layouts/PublishingApp.js` and improve the old `_fetch` function:

```
// old codebase to improve:
async _fetch() {
  let articlesLength = await falcorModel.
    getValue('articles.length').
    then((length) => length);

  let articles = await falcorModel.
    get(['articles', {from: 0, to: articlesLength-1},
      ['_id', 'articleTitle', 'articleContent',
        'articleContentJSON']]).
    then((articlesResponse) => {
```

```
        return articlesResponse.json.articles;
    }).catch(e => {
        return 500;
    });

    if(articles === 500) {
        return;
    }

    this.props.articleActions.articlesList(articles);
}
```

Replace this code with the following:

```
async _fetch() {
    let articlesLength = await falcorModel.
        getValue('articles.length').
        then((length) => length);

    let articles = await falcorModel.
        get(['articles', {from: 0, to: articlesLength-1},
            ['_id','articleTitle', 'articleContent',
            'articleContentJSON', 'articlePicUrl']]).
        then((articlesResponse) => {
            return articlesResponse.json.articles;
        }).catch(e => {
            console.debug(e);
            return 500;
        });

    if(articles === 500) {
        return;
    }

    this.props.articleActions.articlesList(articles);
}
```

As you can see, we have started to fetch `articlePicUrl` via the `falcorModel.get` method.

The next step, also in the `PublishingApp` file, is to improve the `render` function, so you need to improve the following code:

```
// old code:
this.props.article.forEach((articleDetails, articleKey) => {
    let currentArticleJSX = (
        <div key={articleKey}>
            <ArticleCard
```

```
        title={articleDetails.articleTitle}
        content={articleDetails.articleContent} />
    </div>
  );
```

Add to it a new property, which will pass down the image URL:

```
this.props.article.forEach((articleDetails, articleKey) => {
  let currentArticleJSX = (
    <div key={articleKey}>
      <ArticleCard
        title={articleDetails.articleTitle}
        content={articleDetails.articleContent}
        articlePicUrl={articleDetails.articlePicUrl} />
    </div>
  );
```

As you can see, we are passing the fetched `articlePicUrl` to the `ArticleCard` component.

Improving the ArticleCard component

After we pass the `articlePicUrl` variable via properties, we need to improve the following (`src/components/ArticleCard.js`):

```
// old code to improve:
render() {
  let title = this.props.title || 'no title provided';
  let content = this.props.content || 'no content provided';

  let paperStyle = {
    padding: 10,
    width: '100%',
    height: 300
  };

  let leftDivStyle = {
    width: '30%',
    float: 'left'
  }

  let rightDivStyle = {
    width: '60%',
    float: 'left',
    padding: '10px 10px 10px 10px'
  }
}
```

```
return (  
  <Paper style={paperStyle}>  
    <CardHeader  
      title={this.props.title}  
      subtitle='Subtitle'  
      avatar='/static/avatar.png'  
    />  
  
    <div style={leftDivStyle}>  
      <Card >  
        <CardMedia  
          overlay={<CardTitle title={title}  
            subtitle='Overlay subtitle' />>  
          <img src='/static/placeholder.png' height='190' />  
        </CardMedia>  
      </Card>  
    </div>  
    <div style={rightDivStyle}>  
      <div dangerouslySetInnerHTML={{__html: content}} />  
    </div>  
  </Paper>);  
}
```

Change it to the following:

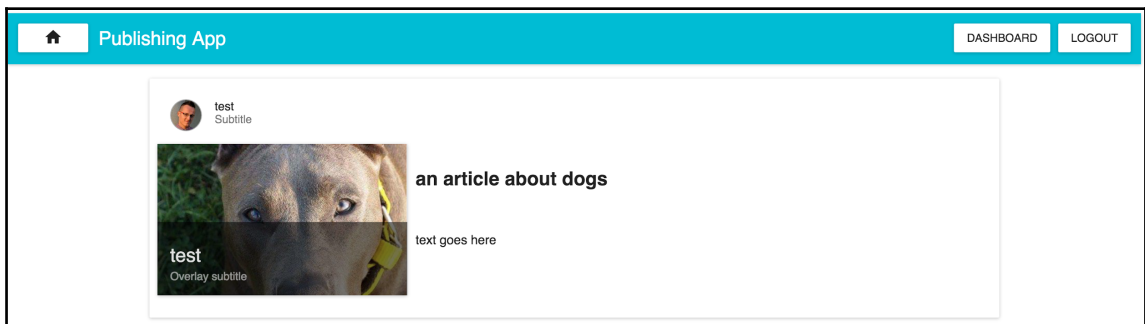
```
render() {  
  let title = this.props.title || 'no title provided';  
  let content = this.props.content || 'no content provided';  
  let articlePicUrl = this.props.articlePicUrl ||  
    '/static/placeholder.png';  
  
  let paperStyle = {  
    padding: 10,  
    width: '100%',  
    height: 300  
  };  
  
  let leftDivStyle = {  
    width: '30%',  
    float: 'left'  
  }  
  
  let rightDivStyle = {  
    width: '60%',  
    float: 'left',  
    padding: '10px 10px 10px 10px'  
  }  
}
```



```
return (  
  <Paper style={paperStyle}>  
    <CardHeader  
      title={this.props.title}  
      subtitle='Subtitle'  
      avatar='/static/avatar.png'  
    />  
  
    <div style={leftDivStyle}>  
      <Card >  
        <CardMedia  
          overlay={<CardTitle title={title}  
            subtitle='Overlay subtitle' />>  
          <img src={articlePicUrl} height='190' />  
        </CardMedia>  
      </Card>  
    </div>  
    <div style={rightDivStyle}>  
      <div dangerouslySetInnerHTML={{__html: content}} />  
    </div>  
  </Paper>);  
)
```

At the beginning of render, we use `let articlePicUrl = this.props.articlePicUrl || '/static/placeholder.png';`, and later, we use that in our image's JSX (`img src={articlePicUrl} height='190'`).

After these two changes, you can see the article with a real cover, like this:



Improving the DashboardView component

Let's improve the dashboard with the cover, so in `src/views/DashboardView.js`, find the following code:

```
// old code:
render () {
  let articlesJSX = [];
  this.props.article.forEach((articleDetails, articleKey) => {
    let currentArticleJSX = (
      <Link
        to={`&grave;/edit-article/${articleDetails['_id']}&grave;`}
        key={articleKey}>
      <ListItem

        leftAvatar={<img src='/static/placeholder.png'
                          width='50'
                          height='50' />}
        primaryText={articleDetails.articleTitle}
        secondaryText={articleDetails.articleContent}
      />
    </Link>
  );

  articlesJSX.push(currentArticleJSX);
});
// below is rest of the render's function
```

Replace it with the following:

```
render () {
  let articlesJSX = [];
  this.props.article.forEach((articleDetails, articleKey) => {
    let articlePicUrl = articleDetails.articlePicUrl ||
      '/static/placeholder.png';
    let currentArticleJSX = (
      <Link
        to={`&grave;/edit-article/${articleDetails['_id']}&grave;`}
        key={articleKey}>
      <ListItem

        leftAvatar={<img src={articlePicUrl} width='50'
                          height='50' />}
        primaryText={articleDetails.articleTitle}
        secondaryText={articleDetails.articleContent}
      />
    </Link>
  );
};
```

```
articlesJSX.push(currentArticleJSX);
});
// below is rest of the render's function
```

As you can see, we have replaced the mocked placeholder with a real cover photo, so on our articles dashboard (which is available after login) we will find real images in the thumbnails.

Editing an article's cover photo

Regarding the article's photo, we need to make some improvements in the `src/views/articles/EditArticleView.js` file, such as importing `ImgUploader`:

```
import ImgUploader from '../../components/articles/ImgUploader';
```

After you have imported `ImgUploader`, improve the constructor of `EditArticleView`. Find the following code:

```
// old code to improve:
class EditArticleView extends React.Component {
  constructor(props) {
    super(props);
    this._onDraftJSChange = this._onDraftJSChange.bind(this);
    this._articleEditSubmit = this._articleEditSubmit.bind(this);
    this._fetchArticleData = this._fetchArticleData.bind(this);
    this._handleDeleteTap = this._handleDeleteTap.bind(this);
    this._handleDeletion = this._handleDeletion.bind(this);
    this._handleClosePopover =
      this._handleClosePopover.bind(this);

    this.state = {
      articleFetchError: null,
      articleEditSuccess: null,
      editedArticleID: null,
      articleDetails: null,
      title: 'test',
      contentJSON: {},
      htmlContent: '',
      openDelete: false,
      deleteAnchorEl: null
    };
  }
}
```

Replace it with the new, improved constructor:

```
class EditArticleView extends React.Component {
  constructor(props) {
    super(props);
    this._onDraftJSChange = this._onDraftJSChange.bind(this);
    this._articleEditSubmit = this._articleEditSubmit.bind(this);
    this._fetchArticleData = this._fetchArticleData.bind(this);
    this._handleDeleteTap = this._handleDeleteTap.bind(this);
    this._handleDeletion = this._handleDeletion.bind(this);
    this._handleClosePopover =
      this._handleClosePopover.bind(this);
    this.updateImgUrl = this.updateImgUrl.bind(this);

    this.state = {
      articleFetchError: null,
      articleEditSuccess: null,
      editedArticleID: null,
      articleDetails: null,
      title: 'test',
      contentJSON: {},
      htmlContent: '',
      openDelete: false,
      deleteAnchorEl: null,
      articlePicUrl: '/static/placeholder.png'
    };
  }
}
```

As you can see, we have bound this to the new `updateImgUrl` function (which will be the `ImgUploader` callback), and we create a new default state for the `articlePicUrl`.

The next step is to improve the current `_fetchArticleData` function:

```
// this is old already in your codebase:
_fetchArticleData() {
  let articleID = this.props.params.articleID;
  if(typeof window !== 'undefined' && articleID) {
    let articleDetails = this.props.article.get(articleID);
    if(articleDetails) {
      this.setState({
        editedArticleID: articleID,
        articleDetails: articleDetails
      });
    } else {
      this.setState({
        articleFetchError: true
      })
    }
  }
}
```

```
    }  
  }  
}
```

Replace it with the following improved code:

```
_fetchArticleData() {  
  let articleID = this.props.params.articleID;  
  if(typeof window !== 'undefined' && articleID) {  
    let articleDetails = this.props.article.get(articleID);  
    if(articleDetails) {  
      this.setState({  
        editedArticleID: articleID,  
        articleDetails: articleDetails,  
        articlePicUrl: articleDetails.articlePicUrl,  
        contentJSON: articleDetails.articleContentJSON,  
        htmlContent: articleDetails.articleContent  
      });  
    } else {  
      this.setState({  
        articleFetchError: true  
      })  
    }  
  }  
}
```

Here, we have added to an initial fetch some new `this.setState` variables, such as `articlePicUrl`, `contentJSON`, and `htmlContent`. The article fetch is here because we need to set up a cover in the `ImgUploader` of a current image that might potentially be changed. The `contentJSON` and `htmlContent` is here in case the user doesn't edit anything in the WYSIWYG editor and we need to have a default value from the database (otherwise, the edit button would save empty values into the database and break the whole editing experience).

Let's improve the `_articleEditSubmit` function. This is the old code:

```
// old code to improve:  
async _articleEditSubmit() {  
  let currentArticleID = this.state.editedArticleID;  
  let editedArticle = {  
    _id: currentArticleID,  
    articleTitle: this.state.title,  
    articleContent: this.state.htmlContent,  
    articleContentJSON: this.state.contentJSON  
  }  
  // striped code for our convience
```

Change to the following improved version:

```
async _articleEditSubmit() {
  let currentArticleID = this.state.editedArticleID;
  let editedArticle = {
    _id: currentArticleID,
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON,
    articlePicUrl: this.state.articlePicUrl
  }
  // striped code for our convenience
```

The next step is to add a new function to the EditArticleView component:

```
updateImgUrl(articlePicUrl) {
  this.setState({
    articlePicUrl: articlePicUrl
  });
}
```

The last step in order to finish the article-editing cover is to improve the old render:

```
// old code to improve:
let initialWYSIWYGValue =
  this.state.articleDetails.articleContentJSON;

return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Edit an existing article</h1>
    <WYSIWYGEditor
      initialValue={initialWYSIWYGValue}
      name='editarticle'
      title='Edit an article'
      onChangeTextJSON={this._onDraftJSChange} />

    <RaisedButton
      onClick={this._articleEditSubmit}
      secondary={true}
      type='submit'
      style={{margin: '10px auto',
        display: 'block', width: 150}}
      label={'Submit Edition'} />
    <hr />
  </div>
)
```

Replace it with the following:

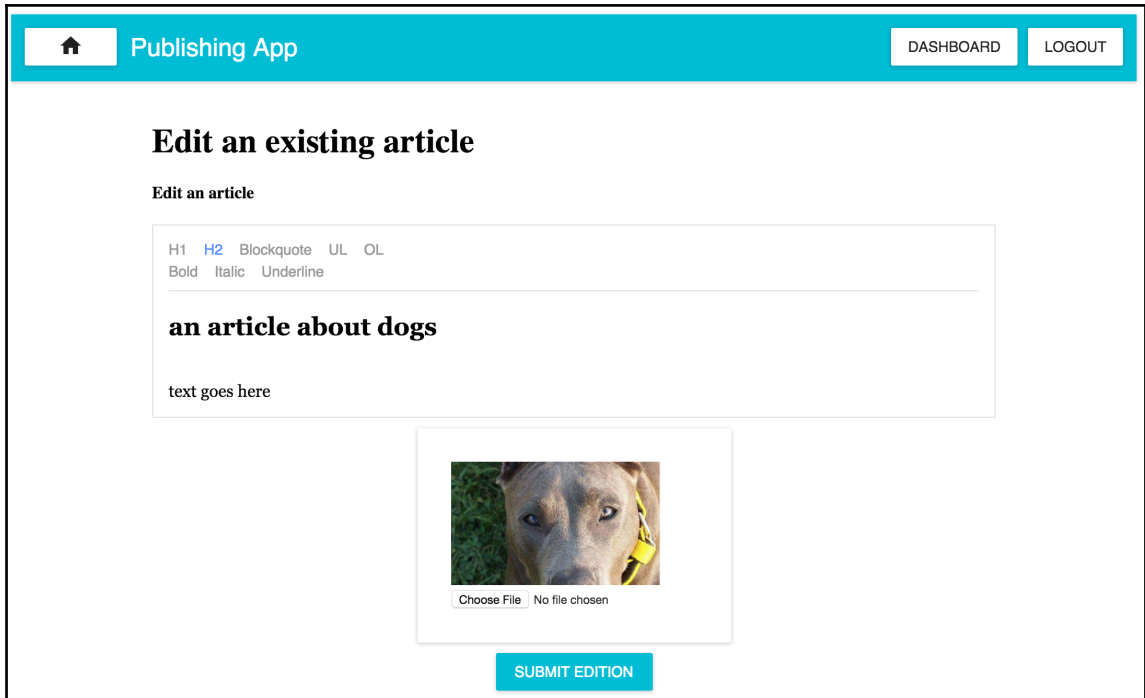
```
let initialWYSIWYGValue =
  this.state.articleDetails.articleContentJSON;

return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Edit an existing article</h1>
    <WYSIWYGeditor
      initialValue={initialWYSIWYGValue}
      name='editarticle'
      title='Edit an article'
      onChangeTextJSON={this._onDraftJSChange} />

    <div style={{margin: '10px 10px 10px 10px'}}>
      <ImgUploader updateImgUrl={this.updateImgUrl}
        articlePicUrl={this.state.articlePicUrl} />
    </div>

    <RaisedButton
      onClick={this._articleEditSubmit}
      secondary={true}
      type='submit'
      style={{margin: '10px auto',
        display: 'block', width: 150}}
      label={'Submit Edition'} />
  </div>
</hr>
```

As you can see, we have added `ImgUploader` and styled it exactly the same way as in `AddArticleView`. The rest of `ImgUploader` does the job for us in order to allow our users to edit article photos.



In this screenshot, you can see how the edit view should look after all the recent improvements.

Adding the ability to add/edit the title and subtitle of an article

In general, we shall improve the article's model in the `server/configMongoose.js` file. Start by finding the following code:

```
// old codebase:
var articleSchema = new Schema({
  articleTitle: String,
  articleContent: String,
  articleContentJSON: Object,
  articlePicUrl: { type: String, default:
    '/static/placeholder.png' }
},
```



```
    {
      minimize: false
    }
  );
```

Replace it with the improved code, as follows:

```
var defaultDraftJSObject = {
  'blocks' : [],
  'entityMap' : {}
}

var articleSchema = new Schema({
  articleTitle: { type: String, required: true, default:
    'default article title' },
  articleSubTitle: { type: String, required: true, default:
    'default subtitle' },
  articleContent: { type: String, required: true, default:
    'default content' },
  articleContentJSON: { type: Object, required: true, default:
    defaultDraftJSObject },
  articlePicUrl: { type: String, required: true, default:
    '/static/placeholder.png' }
},
{
  minimize: false
})
```

As you can see, we have added a lot of required properties in our model; this will affect the ability to save incomplete objects, so in general, our model will be more consistent throughout the life of our publishing app.

We have also added a new property to our model called `articleSubTitle`, which we will be using later in this chapter.

AddArticleView improvements

In general, we will add two `DefaultInput` components (title and subtitle), and the whole form will be using `formsy-react`, so in `src/views/articles/AddArticleView.js`, add new imports:

```
import DefaultInput from '../../components/DefaultInput';
import Formsy from 'formsy-react';
```

The next step is to improve `async _articleSubmit`, so change the old code:

```
// old code to improve:
async _articleSubmit() {
  let newArticle = {
    articleTitle: articleModel.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON,
    articlePicUrl: this.state.articlePicUrl
  }

  let newArticleID = await falcorModel
    .call(
      'articles.add',
      [newArticle]
    ).
    // rest code below is striped
}
```

Replace it with the following:

```
async _articleSubmit(articleModel) {
  let newArticle = {
    articleTitle: articleModel.title,
    articleSubTitle: articleModel.subTitle,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON,
    articlePicUrl: this.state.articlePicUrl
  }

  let newArticleID = await falcorModel
    .call(
      'articles.add',
      [newArticle]
    ).
}
```

As you can see, we have added `articleModel` in the `_articleSubmit` arguments; this will come from `formsy-react`, the same way we implemented it in the `LoginView` and `RegisterView`. We have also added the `articleSubTitle` property to the `newArticle` object.

The old render function return looks like this:

```
// old code below:
return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Add Article</h1>
    <WYSIWYGeditor
      name='addarticle'
    >
```

```
        title='Create an article'
        onChangeTextJSON={this._onDraftJSChange} />

<div style={{margin: '10px 10px 10px 10px'}}>
  <ImgUploader updateImgUrl={this.updateImgUrl}
    articlePicUrl={this.state.articlePicUrl} />
</div>

<RaisedButton
  onClick={this._articleSubmit}
  secondary={true}
  type='submit'
  style={{margin: '10px auto',
    display: 'block', width: 150}}
  label='Submit Article' />
</div>
);
```

Change it to the following:

```
return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Add Article</h1>

    <Formsy.Form onSubmit={this._articleSubmit}>
      <DefaultInput
        onChange={(event) => {}}
        name='title'
        title='Article Title (required)' required />

      <DefaultInput
        onChange={(event) => {}}
        name='subTitle'
        title='Article Subtitle' />

      <WYSIWYGeditor
        name='addarticle'
        title='Create an article'
        onChangeTextJSON={this._onDraftJSChange} />

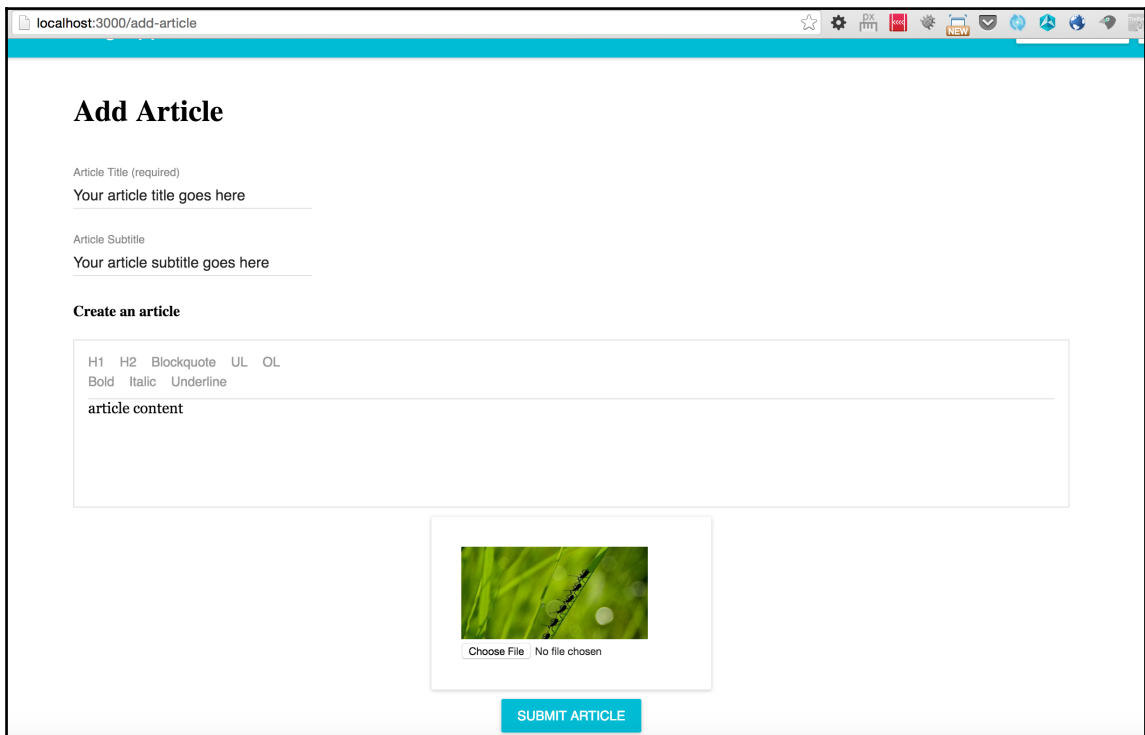
      <div style={{margin: '10px 10px 10px 10px'}}>
        <ImgUploader updateImgUrl={this.updateImgUrl}
          articlePicUrl={this.state.articlePicUrl} />
      </div>

      <RaisedButton
        secondary={true}
        type='submit'
```

```
        style={{margin: '10px auto',
          display: 'block', width: 150}}
        label={'Submit Article'} />
      </Formsy.Form>
    </div>
  );
```

In this code snippet, we have added `Formsy.Form` the same way as in the `LoginView`, so I won't describe it in detail. The most important thing to notice is that with `onSubmit`, we call the `this._articleSubmit` function. We have also added two `DefaultInput` components (title and subtitle): the data from those two inputs will be used in `async _articleSubmit(articleModel)` (as you already know based on previous implementations in this book).

Based on the changes in the `Mongoose` config and in the `AddArticleView` component, you are now able to add a title and subtitle to a new article, as in the following screenshot:

A screenshot of a web browser window showing a form titled "Add Article". The browser's address bar shows "localhost:3000/add-article". The form has a light blue header with the title "Add Article". Below the header, there are two input fields: "Article Title (required)" with the placeholder "Your article title goes here" and "Article Subtitle" with the placeholder "Your article subtitle goes here". Below these fields is a section titled "Create an article" which contains a rich text editor. The editor has a toolbar with options for H1, H2, Blockquote, UL, OL, Bold, Italic, and Underline. The text area of the editor contains the placeholder "article content". Below the editor is a file upload section with a preview of a green image and a "Choose File" button. At the bottom of the form is a blue "SUBMIT ARTICLE" button.

We're still missing the ability to edit the title and subtitle, so let's implement that now.

Ability to edit an article title and subtitle

Go to the `src/views/articles/EditArticleView.js` file and add new imports (in a similar way to the add view):

```
import DefaultInput from '../../components/DefaultInput';
import Formsy from 'formsy-react';
```

Improve the old `_articleEditSubmit` function from the current version:

```
// old code:
async _articleEditSubmit() {
  let currentArticleID = this.state.editedArticleID;
  let editedArticle = {
    _id: currentArticleID,
    articleTitle: this.state.title,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON,
    articlePicUrl: this.state.articlePicUrl
  }
  // rest of the function has been striped below
```

Change it to the following:

```
async _articleEditSubmit(articleModel) {
  let currentArticleID = this.state.editedArticleID;
  let editedArticle = {
    _id: currentArticleID,
    articleTitle: articleModel.title,
    articleSubTitle: articleModel.subTitle,
    articleContent: this.state.htmlContent,
    articleContentJSON: this.state.contentJSON,
    articlePicUrl: this.state.articlePicUrl
  }
  // rest of the function has been striped below
```

As you can see, we do the same thing as in the `AddArticleView`, so you should be familiar with it. The last thing to do is update `render` so that we are able to input the title and subtitle that will be sent as a callback to `_articleEditSubmit` with data in the `articleModel`. The old `return` in the `render` function is as follows:

```
// old code:
return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Edit an existing article</h1>
    <WYSIWYGeditor
      initialValue={initialWYSIWYGValue}>
```

```
        name='editarticle'
        title='Edit an article'
        onChangeTextJSON={this._onDraftJSChange} />
<div style={{margin: '10px 10px 10px 10px'}}>
  <ImgUploader updateImgUrl={this.updateImgUrl}
    articlePicUrl={this.state.articlePicUrl} />
</div>
<RaisedButton
  onClick={this._articleEditSubmit}
  secondary={true}
  type='submit'
  style={{margin: '10px auto',
    display: 'block', width: 150}}
  label={'Submit Edition'} />
<hr />
{/* striped below */}
```

The new improved return in the render function is as follows:

```
return (
  <div style={{height: '100%', width: '75%', margin: 'auto'}}>
    <h1>Edit an existing article</h1>
    <Formsy.Form onSubmit={this._articleEditSubmit}>
      <DefaultInput
        onChange={(event) => {}}
        name='title'
        value={this.state.articleDetails.articleTitle}
        title='Article Title (required)' required />

      <DefaultInput
        onChange={(event) => {}}
        name='subTitle'
        value={this.state.articleDetails.articleSubTitle}
        title='Article Subtitle' />

      <WYSIWYGeditor
        initialValue={initialWYSIWYGValue}
        name='editarticle'
        title='Edit an article'
        onChangeTextJSON={this._onDraftJSChange} />

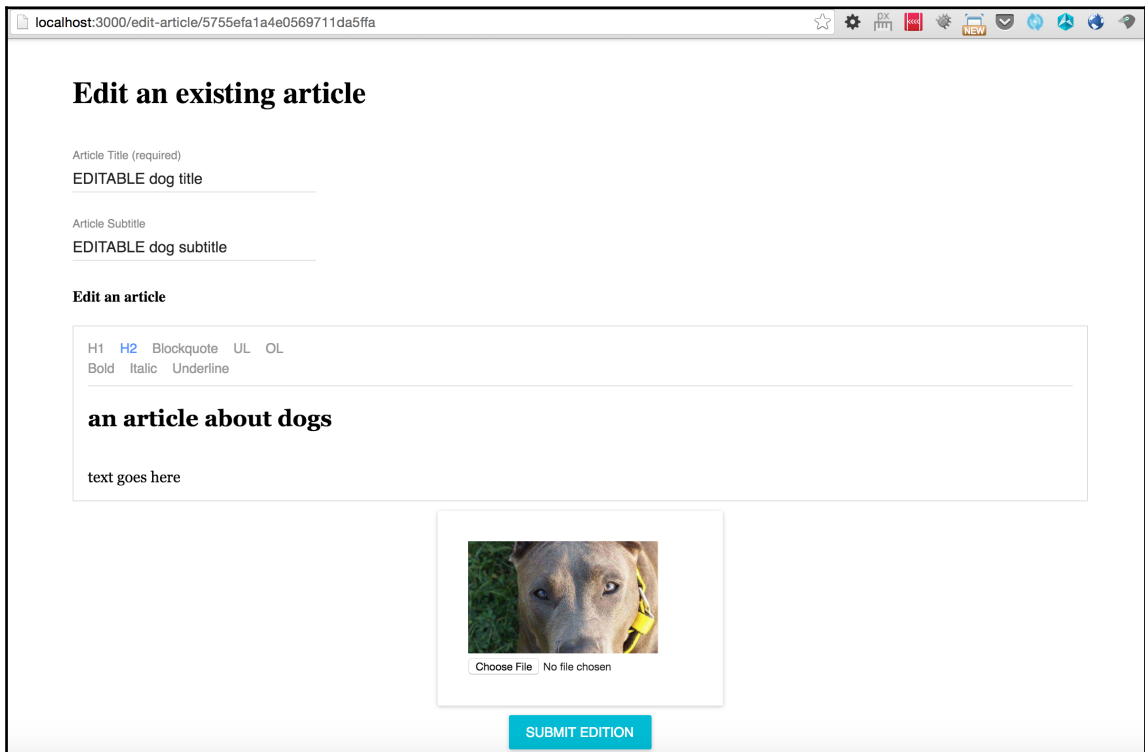
      <div style={{margin: '10px 10px 10px 10px'}}>
        <ImgUploader updateImgUrl={this.updateImgUrl}
          articlePicUrl={this.state.articlePicUrl} />
      </div>

      <RaisedButton
        onClick={this._articleEditSubmit}
```

```
secondary={true}
type='submit'
style={{margin: '10px auto',
display: 'block', width: 150}}
label={'Submit Edition'} />
</Formsy.Form>
{/* striped below */}
```

We are doing the same thing here as we did in the `AddArticleView`. We are introducing `Formsy.Form`, which is calling back the article's title and subtitle when a user hits the submit button (**SUBMIT EDITION**).

Here is an example of how it should look:



The screenshot shows a web browser window with the address bar displaying `localhost:3000/edit-article/5755efa1a4e0569711da5ffa`. The page title is "Edit an existing article". Below the title, there are two text input fields: "Article Title (required)" with the placeholder text "EDITABLE dog title", and "Article Subtitle" with the placeholder text "EDITABLE dog subtitle". Below these fields is a section titled "Edit an article" which contains a rich text editor. The editor has a toolbar with options for H1, H2 (selected), Blockquote, UL, OL, Bold, Italic, and Underline. The main text area of the editor contains the text "an article about dogs" followed by "text goes here". Below the text area is a file upload section with a placeholder image of a dog's face and a "Choose File" button. Below the file upload section is a blue button labeled "SUBMIT EDITION".

ArticleCard and PublishingApp improvements

Improve the render function in ArticleCard so it will also show the subtitle (currently, it's mocked). The `src/components/ArticleCard.js` file's old content is as follows:

```
// old code:
render() {
  let title = this.props.title || 'no title provided';
  let content = this.props.content || 'no content provided';
  let articlePicUrl = this.props.articlePicUrl ||
    '/static/placeholder.png';

  let paperStyle = {
    padding: 10,
    width: '100%',
    height: 300
  };

  let leftDivStyle = {
    width: '30%',
    float: 'left'
  }

  let rightDivStyle = {
    width: '60%',
    float: 'left',
    padding: '10px 10px 10px 10px'
  }

  return (
    <Paper style={paperStyle}>
      <CardHeader
        title={this.props.title}
        subtitle='Subtitle'
        avatar='/static/avatar.png'
      />

      <div style={leftDivStyle}>
        <Card >
          <CardMedia
            overlay={<CardTitle title={title}
              subtitle='Overlay subtitle' />}>
            <img src={articlePicUrl} height='190' />
          </CardMedia>
        </Card>
      </div>
      <div style={rightDivStyle}>
```



```
        <div dangerouslySetInnerHTML={{__html: content}} />
      </div>
    </Paper>;
  }
}
```

Let's change this to the following:

```
render() {
  let title = this.props.title || 'no title provided';
  let subTitle = this.props.subTitle || '';
  let content = this.props.content || 'no content provided';
  let articlePicUrl = this.props.articlePicUrl ||
    '/static/placeholder.png';

  let paperStyle = {
    padding: 10,
    width: '100%',
    height: 300
  };

  let leftDivStyle = {
    width: '30%',
    float: 'left'
  }

  let rightDivStyle = {
    width: '60%',
    float: 'left',
    padding: '10px 10px 10px 10px'
  }

  return (
    <Paper style={paperStyle}>
      <CardHeader
        title={this.props.title}
        subtitle={subTitle}
        avatar='/static/avatar.png'
      />

      <div style={leftDivStyle}>
        <Card >
          <CardMedia
            overlay={<CardTitle title={title}
              subtitle={subTitle} />>
            <img src={articlePicUrl} height='190' />
          </CardMedia>
        </Card>
      </div>
    </Paper>
  );
}
```

```
        <div style={rightDivStyle}>
          <div dangerouslySetInnerHTML={{__html: content}} />
        </div>
      </Paper>);
    }
  }
```

As you can see, we have defined a new `subTitle` variable and have then used it in the `CardHeader` and `CardMedia` components, so now it will show the subtitle as well.

Another thing to do is to make the `PublishingApp` also fetch the subtitle that has been introduced in the chapter, so we need to improve the following old code:

```
// old code:
async _fetch() {
  let articlesLength = await falcorModel.
    getValue('articles.length').
    then((length) => length);

  let articles = await falcorModel.
    get(['articles', {from: 0, to: articlesLength-1},
      ['_id', 'articleTitle', 'articleContent',
        'articleContentJSON', 'articlePicUrl']]).
    then((articlesResponse) => {
      return articlesResponse.json.articles;
    }).catch(e => {
      console.debug(e);
      return 500;
    });
  // no changes below, striped
```

Replace it with the following:

```
async _fetch() {
  let articlesLength = await falcorModel.
    getValue('articles.length').
    then((length) => length);

  let articles = await falcorModel.
    get(['articles', {from: 0, to: articlesLength-1}, ['_id',
      'articleTitle', 'articleSubTitle', 'articleContent',
        'articleContentJSON', 'articlePicUrl']]).
    then((articlesResponse) => {
      return articlesResponse.json.articles;
    }).catch(e => {
      console.debug(e);
      return 500;
    });
}
```

As you can see, we have started `falcorModel.get` with the `articleSubTitle` property.

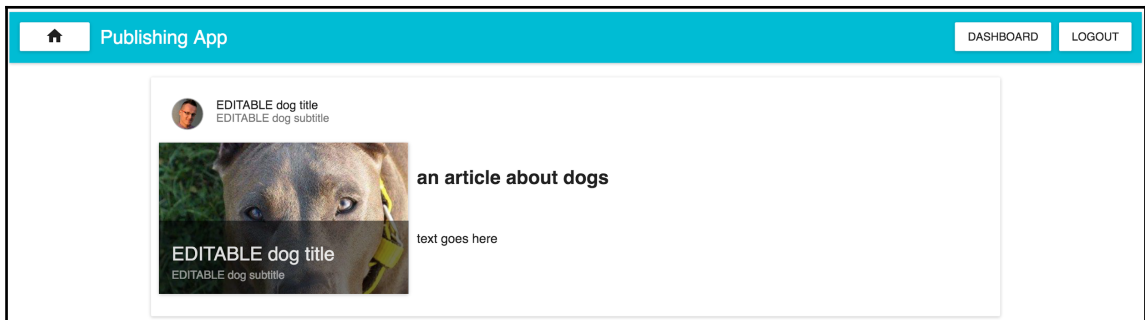
Of course, we need to pass this `subTitle` property to the `ArticleCard` component in the `PublishingApp` class's render function:

```
// old code:
this.props.article.forEach((articleDetails, articleKey) => {
  let currentArticleJSX = (
    <div key={articleKey}>
      <ArticleCard
        title={articleDetails.articleTitle}
        content={articleDetails.articleContent}
        articlePicUrl={articleDetails.articlePicUrl} />
    </div>
  );
```

In the end, we will get the following:

```
this.props.article.forEach((articleDetails, articleKey) => {
  let currentArticleJSX = (
    <div key={articleKey}>
      <ArticleCard
        title={articleDetails.articleTitle}
        content={articleDetails.articleContent}
        articlePicUrl={articleDetails.articlePicUrl}
        subTitle={articleDetails.articleSubTitle} />
    </div>
  );
```

After all these changes on the main page, you can find an edited article with the title, subtitle, cover photo, and content (created by our WYSIWYG editor):



Dashboard improvement (now we can strip the remaining HTML)

The last step in this chapter is to improve the dashboard. It will string the HTML from the props in order to make a better look and feel when a user will browse our application. Find the following code:

```
// old code:
this.props.article.forEach((articleDetails, articleKey) => {
  let articlePicUrl = articleDetails.articlePicUrl ||
    '/static/placeholder.png';
  let currentArticleJSX = (
    <Link to={`&grave;/edit-article/${articleDetails['_id']}&grave;`}
      key={articleKey}>
      <ListItem

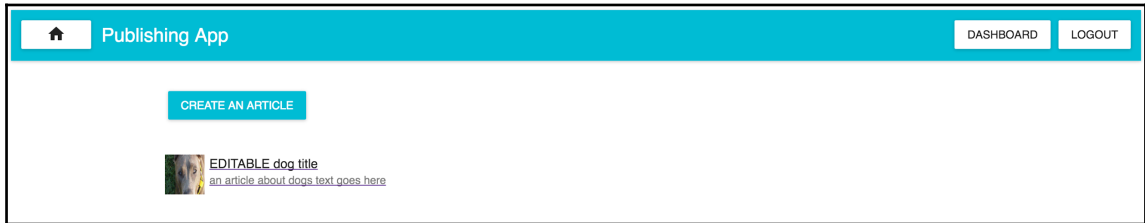
        leftAvatar=<img src={articlePicUrl} width='50'
          height='50' />
        primaryText={articleDetails.articleTitle}
        secondaryText={articleDetails.articleContent}
      />
    </Link>
  );
});
```

Replace it with the following:

```
this.props.article.forEach((articleDetails, articleKey) => {
  let articlePicUrl = articleDetails.articlePicUrl ||
    '/static/placeholder.png';
  let articleContentPlanText =
    articleDetails.articleContent.replace(/<\/?[>]+(>|$/g,
    '');
  let currentArticleJSX = (
    <Link to={`&grave;/edit-article/${articleDetails['_id']}&grave;`}
      key={articleKey}>
      <ListItem

        leftAvatar=<img src={articlePicUrl} width='50'
          height='50' />
        primaryText={articleDetails.articleTitle}
        secondaryText={articleContentPlanText}
      />
    </Link>
  );
});
```

As you can see, we simply strip the HTML tags from the HTML so that we will get better `secondaryText` without the HTML markup, as in this example:



Summary

We have implemented all of the features that are within the book's scope. The next step is to start working on the deployment of this application.

If you want to improve your coding skills, it's a good idea to implement some features completely on your own. These are some ideas for features that are still missing from our publishing app.

We could have a separate link to a certain article so you can share it with a friend. This could be useful if you want to create a human-readable unique slug associated with a certain article in the database. So, instead of linking to something such as `http://reactjs.space/570b6e26ae357d391c6ebc1d` (`reactjs.space` is a domain that we will use on our production server), a user can share a link such as `http://reactjs.space/an-article-about-a-dog`.

There could be a way to associate an article with the editor who posted it. Currently, it's mocked. You can unmock it.

A user can't change their user details while they are logged in--this could be a good way to practice more full-stack development.

A user can't set up their avatar's image--you can add this feature in a similar way to how we implemented the image cover.

Create a more robust DraftJS WYSIWYG editor with plugins. Robust plugins are easy to implement for mentions, stickers, emoji, hashtags, undo/redo, and more. Visit <https://www.draft-js-plugins.com/> for more details about them. Implement one or two that you like the most.

In the next chapter, we will start deploying our MongoDB instance online using www.mLab.com, which is a Database-as-a-Service provider and helps us build scalable MongoDB nodes with ease.

Let's start with the deployment fun!

7

The MongoDB Deployment on mLab

We have come to the point where we need to start planning the deployment of our application. We have chosen MongoDB as our database. There are different approaches of using it for scaling--you can do everything on your own with your own servers (more time consuming and demanding) or you can use services that do replications/scaling for you, such as Database-as-a-Service providers.

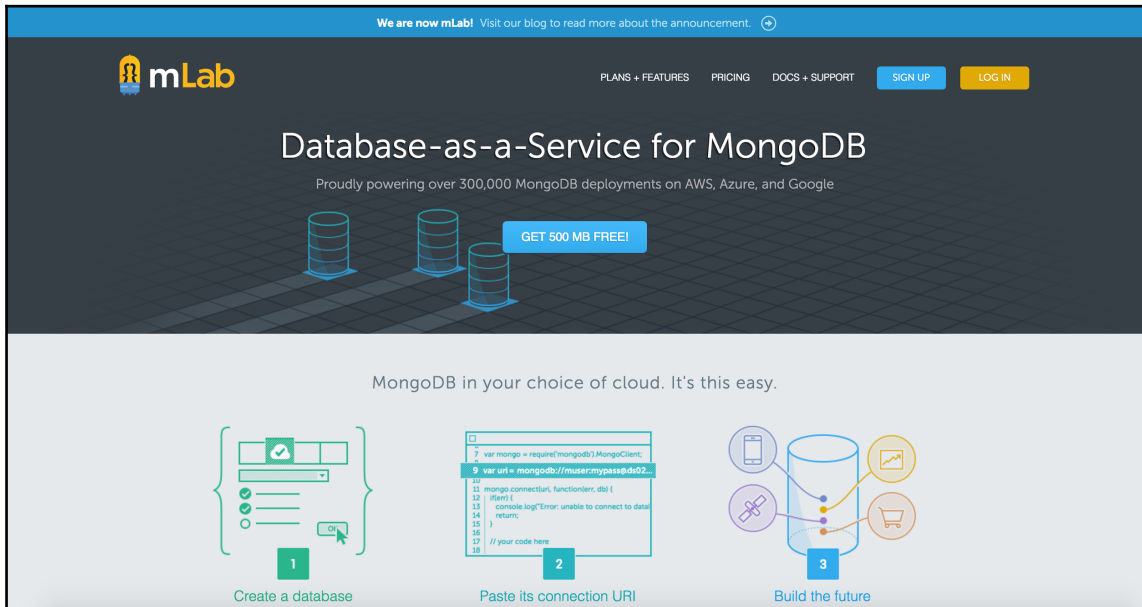
In this chapter, we will cover the following topics:

- Creating an mLab account and creating a new MongoDB deployment
- How a replica set works in MongoDB and how you can use it within mLab
- Testing the replica set on a live demo (flip-flop from mLab)
- Setting up the database user and password
- Learning about what you need to prepare for deployment on AWS EC2

mLab overview

We will use mLab in our case in order to spend less time configuring the low-level stuff on MongoDB and more time building a robust scalable application.

If we go to www.mLab.com, there is a free DB plan (that we will use in this chapter) and a paid DB plan:



In general, mLab provides several interesting features such as the following:

- **Tools for cloud automation:** These provide on-demand provisioning (preparing) on AWS, Azure, or Google; replica sets (described in detail later in this chapter); and **sharded clusters**. These also provide seamless, zero-downtime scaling, and high availability via automatic failover.
- **Tools for backup and recovery:** These provide automatic backups, which can help in later project stages in case of an emergency.
- **Tools for monitoring and alerts:** For example, there is a **slow queries** tool that helps you to find slow queries, which can be optimized by adding an index.
- **Tools for online data browsing:** You can browse the MongoDB's collection via the browser when you are logged into mLab's administration panel.

Replica set connections and high availability

In MongoDB, there is a feature that ensures high availability using automatic failover. In short, failover is a feature that ensures that if a primary server (that has the most important copy of your database) fails, then a secondary member's database becomes the primary one if the original primary server is unavailable.

A secondary member's database is a server that keeps the so-called **read-only backup** of your database.

The primary and the secondary databases very often replicate themselves in order to be in sync all the time. The secondary server is mostly for read operations.

This whole replica set feature is quite time-consuming to implement from scratch (without mLab), but mLab provides this feature in order to *abstract* this part so that our whole process will be more automated.

MongoDB failover

mLab also provides a great tool for testing the failover scenario in our app, which is available at <http://flip-flop.mlab.com>.

The screenshot shows the web interface of the flip-flop.mlab.com service. The header includes the mLab logo and the text "FLIP-FLOP". The main heading reads "flip-flop is a MongoDB Replica Set demonstration and experimentation service". Below this, a paragraph explains that the service uses a MongoDB replica set with three nodes: "flip" (primary), "flop" (secondary), and an arbiter. It states that every 60 seconds, the primary will step down and the cluster will "failover" to the other node. A section titled "Visualize the replica set election process" features a diagram of two server nodes, "flip" (blue) and "flop" (grey), connected by three small circles representing network links. To the right of the diagram, the status is shown as "Status: flip is PRIMARY" and "Next Failover 31 SECONDS". Below the diagram, there is a section for viewing logs, with a checkbox for "arbiter's logs" selected and checkboxes for "flip's logs" and "flop's logs" unselected. A message at the bottom of the log section states: "Logs will start streaming when there is replica set activity. Please wait..."

Here, we can test how automatic failover with the MongoDB replica set works. As we can see in the preceding screenshot, there are three nodes: the replica's **flip** and **flop**, and an arbiter. In the flip-flop's demo, you can connect to the arbiter server, and the primary server will step down and the cluster will failover to the other node. You can experiment with it--try on your own and have fun!

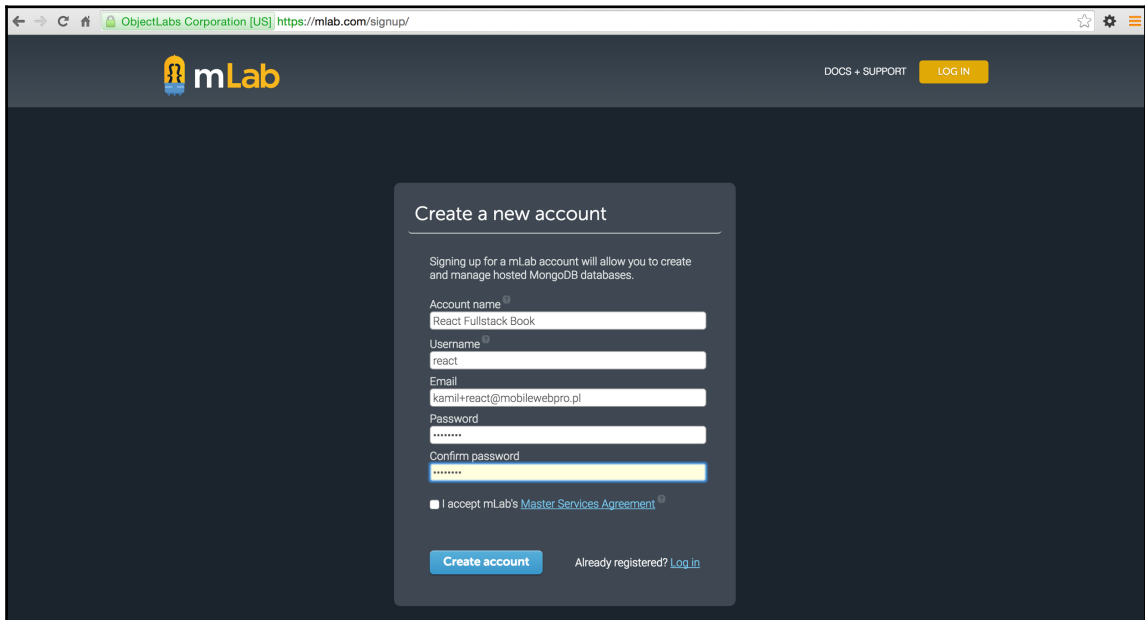
You can find more documentation on how to play with the flip-flop's demo at <http://docs.mlab.com/connecting/#replica-set-connections>.

Free versus paid plan in mLab

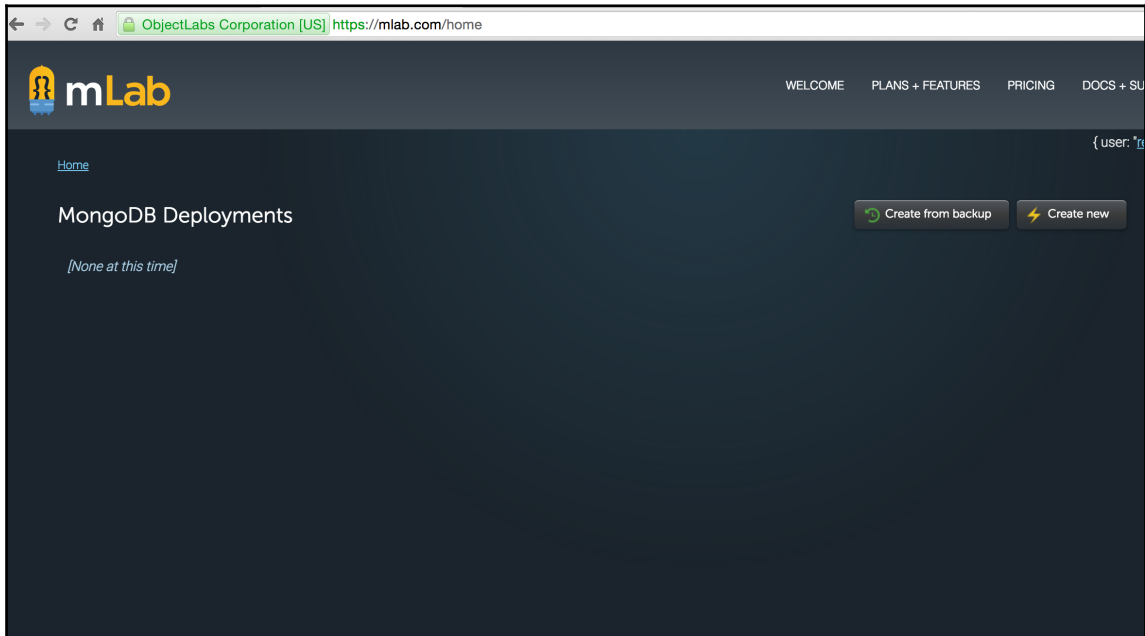
In this book, we will guide you through the process of using mLab with a free plan. In mLab, the replica set is available in the paid plans (starting at \$15/month), and, of course, you can use the flip-flop's demo for free in order to play with that very important feature of MongoDB.

The new mLab's account and node

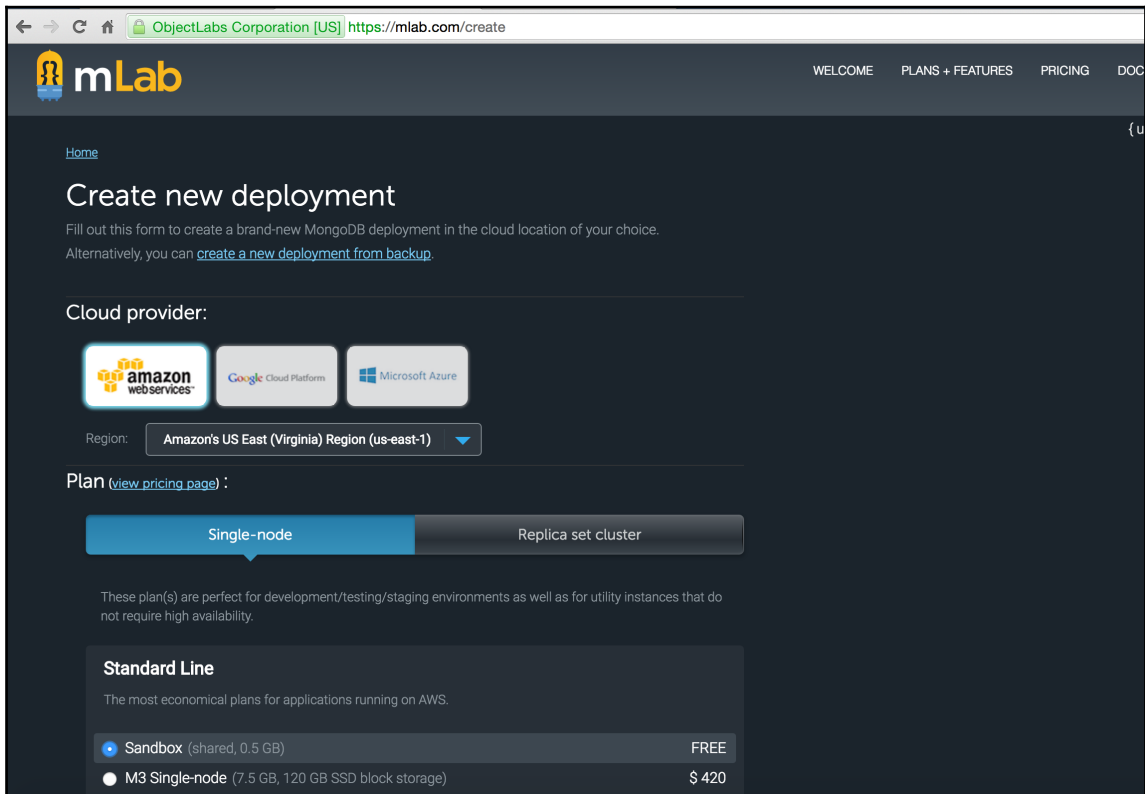
1. Go to <https://mlab.com/signup/>, as shown in the following screenshot:

A screenshot of the mLab website's signup page. The browser's address bar shows 'ObjectLabs Corporation [US] https://mlab.com/signup/'. The page has a dark blue header with the mLab logo on the left and 'DOCS + SUPPORT' and a yellow 'LOG IN' button on the right. The main content area is dark blue and features a white 'Create a new account' form. The form includes a sub-header 'Create a new account', a note about signing up, and several input fields: 'Account name' (with 'React Fullstack Book' entered), 'Username' (with 'react' entered), 'Email' (with 'kamil@react@mobilewebpro.pl' entered), 'Password' (with masked characters), and 'Confirm password' (with masked characters). Below the fields is a checkbox for 'I accept mLab's Master Services Agreement' and a blue 'Create account' button. A link for 'Already registered? Log in' is also present.

2. Verify your e-mail by clicking on the confirm link in your inbox.
3. Click on the **Create new** button, as shown in the following screenshot:

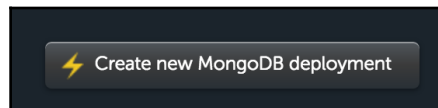


4. You are on the **Create new deployment** page. Choose **Single-node | Sandbox (FREE)**, as shown in the following screenshot:

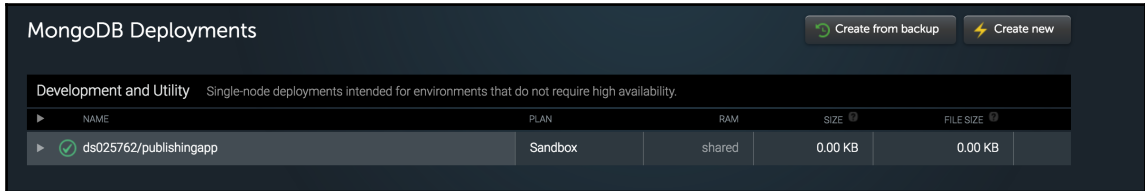


The screenshot shows the mLab 'Create new deployment' page. The browser address bar displays 'https://mlab.com/create'. The page header includes the mLab logo and navigation links: WELCOME, PLANS + FEATURES, PRICING, and DOC. The main heading is 'Create new deployment', followed by instructions to fill out the form to create a MongoDB deployment in the cloud. Below this, the 'Cloud provider' section offers three options: Amazon Web Services (selected), Google Cloud Platform, and Microsoft Azure. The 'Region' dropdown is set to 'Amazon's US East (Virginia) Region (us-east-1)'. The 'Plan' section shows two options: 'Single-node' (selected) and 'Replica set cluster'. A note states that these plans are for development/testing/staging environments. Under the 'Standard Line' section, two plans are listed: 'Sandbox (shared, 0.5 GB)' for FREE and 'M3 Single-node (7.5 GB, 120 GB SSD block storage)' for \$420.

5. While you are still at `https://mlab.com/create` (**Create new deployment**), set the database name as `publishingapp` and click on the **Create new MongoDB deployment** button, as shown in the next screenshot:



6. After following the preceding steps, you should be able to find the MongoDB deployment on the dashboard (<https://mlab.com/home>), as shown in the following screenshot:

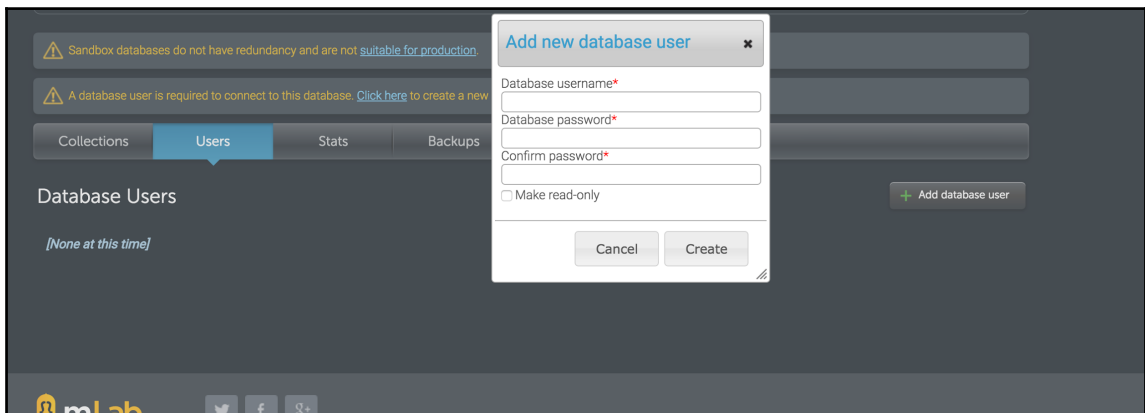


Creating the database's user/password and other configurations

Right now, the database is ready for our publishing application, but it's still empty.

There are steps that we need to take in order to use it:

1. Create a user/password combination. We need to click on the database that has been just created and find a tab called **Users**. After you click on it, click on the **Add new database** user button and then fill in the details on the form, as shown in the following screenshot:



2. Let's assume for this book that our details are as follows:

DB username: usermlab

DB password: pwdblalab

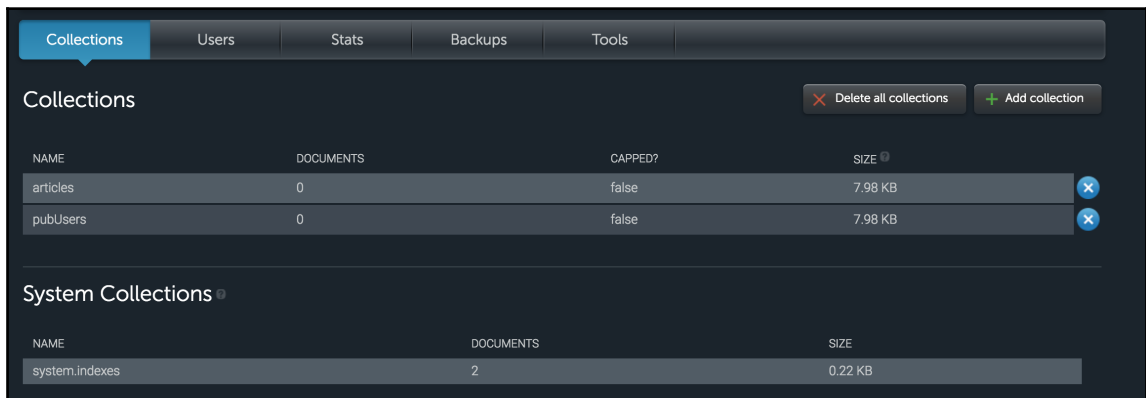
In the places where we will use the username and password, I will use these details.

3. After that, we need to create the collections that are identical to our local MongoDB:

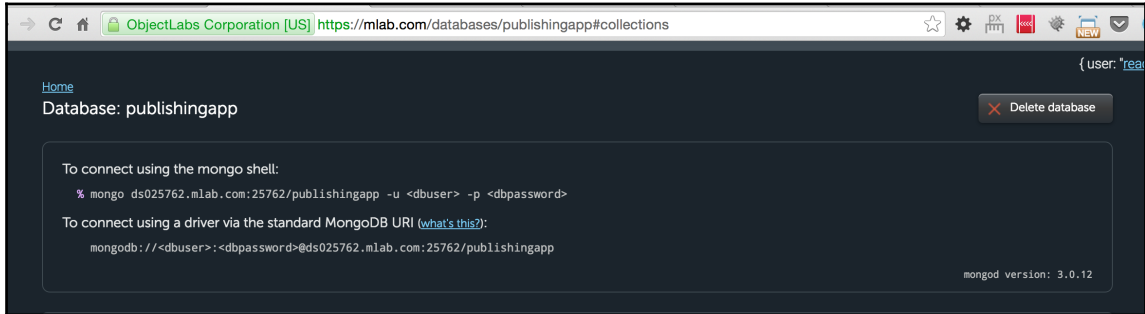
Collections | Add collection | articles

Collections | Add collection | pubUsers

4. After performing all the preceding steps, you should see something like the following screenshot:



5. At this stage, the last thing is to write down the Mongo details from the following screenshot:



Config wrap up

We need to keep and share all the information from mLab along with the AWS S3 details. The details will be useful in the next chapter when deploying our app on Amazon AWS EC2.

At this point in the book, there are details that we need to keep separately:

```
AWS_ACCESS_KEY_ID=<<access-key-obtained-in-previous-chapter>>
AWS_SECRET_ACCESS_KEY=<<secret-key-obtained-in-previous-chapter>>
AWS_BUCKET_NAME=publishing-app
AWS_REGION_NAME=eu-central-1
MONGO_USER=usermlab
MONGO_PASS=pwdmlab
MONGO_PORT=<<port-from-your-mlab-node>>
MONGO_ENV=publishingapp
MONGO_HOSTNAME=<<hostname-from-your-mlab-node>>
```

Make sure, that you have replaced the ports and the hostname to the correct one (as provided by mLab in the preceding screenshot).

All the Mongo env variables can be obtained from mLab, where you can find a link similar to the following (this is an example copied from the account created while writing this chapter):

```
mongo ds025762.mlab.com:25762/publishingapp -u <dbuser> -p <dbpassword>
```

Summary

In the next chapter, we will start using those environment variables on our production server on the AWS EC2's platform. Keep all these details noted in an easily accessible, safe place, as we will use them soon.

The last thing is to check if the app is running correctly and to use the remote mLab MongoDB (instead of the local MongoDB that was run with the `mongd` command). You can do this by running it with `npm start`, and then you shall see the empty home page of the publishing app. Because we moved away from the local database and the remote is empty, you need to register a new user and try to publish a new article with mLab under the hood to store the data.

8

Docker and the EC2 Container Service

We have done all the stuff related to database-as-a-backend with mLab. The publishing application should be working 100 percent remotely on the mLab MongoDB instance, so you don't need to run the `mongod` command anymore.

It's time to prepare our Docker container and deploy it on EC2 completely with the use of ECS (EC2 Container Service) and load balancers.

What is Docker? It's a very useful piece of software that is open source and helps you pack, ship, and run any app as a light (in comparison to a virtual machine, for example) container.

A container's goals are similar to virtual machines--the big difference is that Docker was created with software development in mind, as opposed to VMs. You need to also be aware that a fully virtualized system has its own resources allocated to it, which causes minimal resource sharing, which is different for Docker containers. Of course, in VMs, you get more isolation, but the cost is that the VMs are much heavier (requiring more disk space, RAM, and other resources). Docker's containers are lightweight and are able to share more things among different containers in comparison to VMs.

The good part is that Docker's containers are hardware and platform independent, so all worries about whether what you are working on will run everywhere are disappearing.

Generally, Docker's benefits are that it increases developers' productivity, helps them ship software faster, helps move the software from local development machines to production deployments on AWS, and so on. Docker also allows versioning (similar to Git) of your software, which can be helpful when you need a quick rollback on the production server.

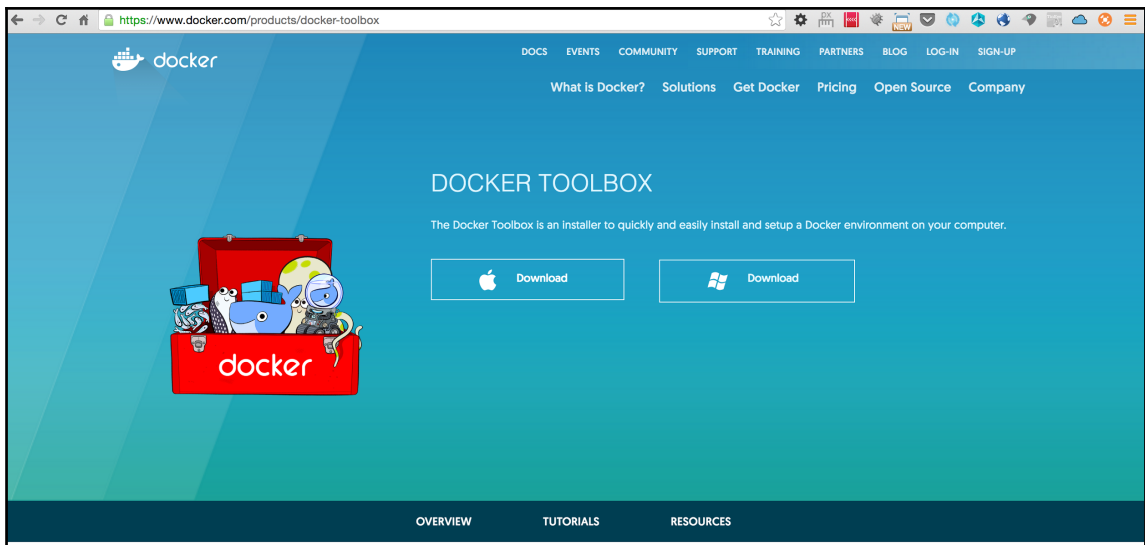
In this chapter, you will learn the following things:

- Installing the Docker app on your machine with Docker Toolbox on non-Linux machines
- Testing whether your Docker setup is correct
- Preparing the publishing app in order to use mLab Mongo for the database
- Creating a new Docker container for the publishing app
- Creating your first Dockerfile, which will deploy the publishing app on Linux CentOS
- EC2 Container Service
- AWS load balancers
- Using Amazon Route 53 for DNS services
- AWS identity and access management (IAM)

Docker installation with Docker Toolbox

Installing Docker is quite easy. Visit the official installation page at <https://docs.docker.com/engine/installation/> because it will guide you best depending on your operating system. There are easy-to-follow installers for iOS and Windows and a lot of instructions for different Linux distributions.

If you are using a non-Linux machine, then you also need to install Docker Toolbox for Windows or OS X. This is quite simple with its installers, which are available at <https://www.docker.com/products/docker-toolbox>, as shown in the following screenshot:



If you are using Linux, there are some extra steps to be performed as you need to turn on virtualization in BIOS:



- Install the Docker machine with instructions from the official docs at <https://docs.docker.com/machine/install-machine/>
- For Ubuntu, you need to install VirtualBox manually from <http://help.ubuntu.com/community/VirtualBox>
- For other Linux distributions visit https://www.virtualbox.org/wiki/Linux_Downloads

After you have installed Docker (together with Toolbox on OS X and Windows) on your local machine, run in the Terminal the following command:

```
$ docker info
```

After you run this command, you will be able to see something similar to the following screenshot:

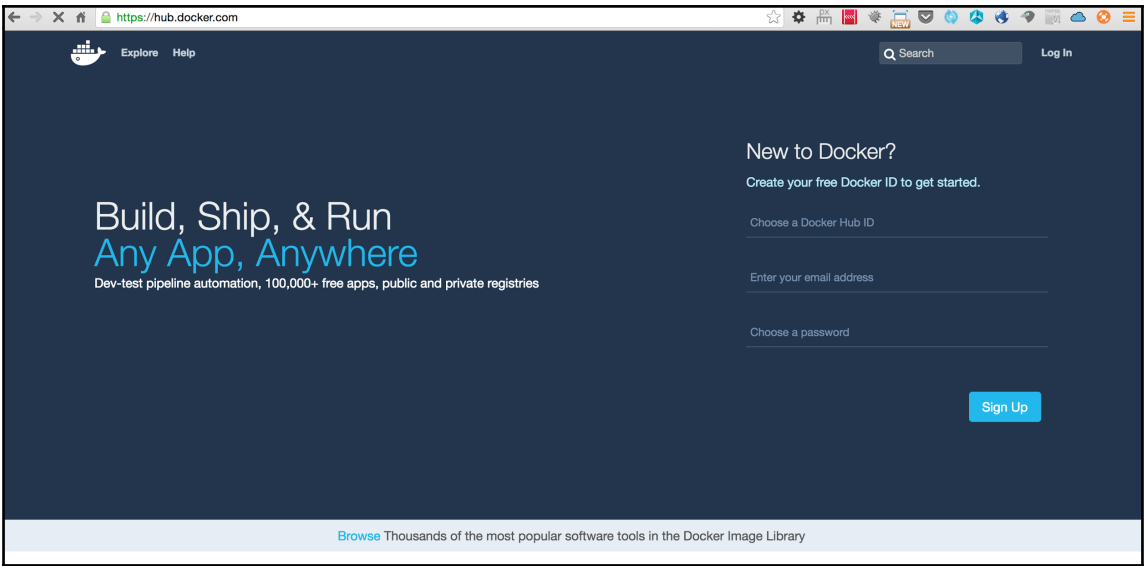
```
przeor@Kamils-MBP React-Convention-Book (8) $ docker info
Containers: 7
  Running: 2
  Paused: 0
  Stopped: 5
Images: 1
Server Version: 1.11.1
Storage Driver: aufs
  Root Dir: /mnt/sda1/var/lib/docker/aufs
  Backing Filesystem: extfs
  Dirs: 52
  Dirperm1 Supported: true
Logging Driver: json-file
Cgroup Driver: cgroupfs
Plugins:
  Volume: local
  Network: null host bridge
Kernel Version: 4.4.8-boot2docker
Operating System: Boot2Docker 1.11.1 (TCL 7.0); HEAD : 7954f54 - Wed Apr 27 16:36:45 UTC 2016
OSType: linux
Architecture: x86_64
CPUs: 1
Total Memory: 1.955 GiB
Name: default
ID: 3070:CLQS:G3EJ:J7BS:TKMH:XMUW:LCEQ:I2DP:CYLE:4D0D:HR5S:GMID
Docker Root Dir: /mnt/sda1/var/lib/docker
Debug mode (client): false
Debug mode (server): true
  File Descriptors: 18
  Goroutines: 42
  System Time: 2016-06-26T07:51:06.347220635Z
  EventsListeners: 0
Registry: https://index.docker.io/v1/
Labels:
  provider=virtualbox
przeor@Kamils-MBP React-Convention-Book (8) $
```

If you can see something like this, then your installation is successful. Let's continue with Docker.

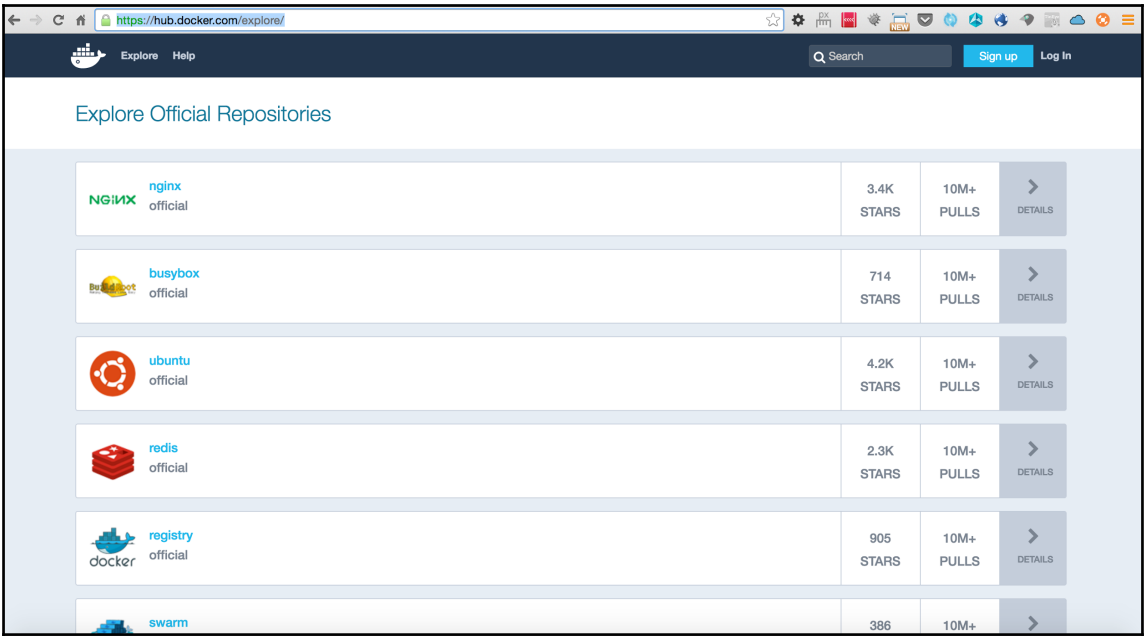
Docker Hub - an hello world example

Before we start creating the publishing app's Docker container, let's start playing with an official Docker *hello world* example that will make you aware of how the Docker Hub works.

Docker Hub is to Docker containers as GitHub is to Git's repositories. You can have public and private containers in Docker. The main page of Docker Hub look like this:



Just to give you a feel for it, if you visit `https://hub.docker.com/explore/`, you can see different containers that are ready for use, like this, for example:



Just for our demo exercise, we will use a container called `hello world`, which is publicly available at <https://hub.docker.com/r/library/hello-world/>.

In order to run this `hello-world` example, run the following in your Terminal:

```
$ docker run hello-world
```

After you run this, you will see something similar to the following:

```
przeor@Kamils-MacBook-Pro docker $ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world

a9d36faac0fe: Pull complete
Digest: sha256:e52be8ffeb1f374f440893189cd32f44cb166650e7ab185fa7735b7dc48d619
Status: Downloaded newer image for hello-world:latest

Hello from Docker.
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker Hub account:
https://hub.docker.com

For more examples and ideas, visit:
https://docs.docker.com/engine/userguide/
```

Let's understand what's just happened: we use the `docker run` command in order to start the container based on an image (in our example, we have used the `hello world` container image). In this case, we do the following:

1. Run the command that tells Docker to start the container named `hello-world` with no extra commands.
2. After you hit Enter, Docker will download Docker Hub.
3. Then it will start the container in the VM, using Docker Toolbox on non-Linux systems.

The `hello-world` image comes from the public registry called Docker Hub as mentioned before (which you can visit at <https://hub.docker.com/r/library/hello-world/>).

Dockerfile example

Every image is composed of a Dockerfile. An example Dockerfile for the `hello-world` example looks like the following:

```
// source:
https://github.com/docker-library/hello-world/blob/master/Dockerfile
FROM scratch
COPY hello /
CMD ["/hello"]
```

A Dockerfile is a set of instructions that tell Docker how to build a container image. We will create our own in a moment. An analogy for a Dockerfile can be the Bash language that you can use on any Linux/Unix machine. Of course, it's different, but the general idea of writing instructions to create the job is similar.

Modifications to our codebase in order to create it

Currently, we are sure that our Docker application's setup is working correctly.

First of all, we need to make some modifications to our current codebase as there are small tweaks to make it work properly.

Make sure that the following files have the proper content.

The `server/.env` file's content has to be as follows:

```
AWS_ACCESS_KEY_ID=<<__AWS_ACCESS_KEY_ID__>>
AWS_SECRET_ACCESS_KEY=<<__AWS_SECRET_ACCESS_KEY__>>
AWS_BUCKET_NAME=publishing-app
AWS_REGION_NAME=eu-central-1
MONGO_USER=<<__your_mlab_mongo_user__>>
MONGO_PASS=<<__your_mlab_mongo_pass__>>
MONGO_PORT=<<__your_mlab_mongo_port__>>
MONGO_ENV=publishingapp
MONGO_HOSTNAME=<<__your_mlab_mongo_hostname__>>
```



For now, we will load the environment variables from a file, but later we will load them from the AWS panel. It's not really production-secure to keep all that secret data on the server. We use it now for the sake of brevity; later, we'll delete it in favor of a more secure approach.

Regarding the Mongo environment variables, we learned them in the previous chapter about setting up mLab (get to back to the chapter if you missed any of the details required at this point).

The `server/index.js` file's content has to be as follows:

```
var env = require('node-env-file');
// Load any undefined ENV variables from a specified file.
env(__dirname + '/.env');

require("babel-core/register");
require("babel-polyfill");
require('./server');
```

Make sure you are loading `.env` from the file at the beginning of `server/index.js`. It will be required in order to load the mLab Mongo details from the environment variables (`server/.env`).

The `server/configMongoose.js` file's content has to be replaced. Find the following code:

```
// this is old code from our codebase:
import mongoose from 'mongoose';
var Schema = mongoose.Schema;

const conf = {
  hostname: process.env.MONGO_HOSTNAME || 'localhost',
  port: process.env.MONGO_PORT || 27017,
  env: process.env.MONGO_ENV || 'local',
};
mongoose.connect(`mongodb://${conf.hostname}:
  ${conf.port}/${conf.env}&grave;`);
```

The new version of the same improved code has to be as follows:

```
import mongoose from 'mongoose';
var Schema = mongoose.Schema;

const conf = {
  hostname: process.env.MONGO_HOSTNAME || 'localhost',
  port: process.env.MONGO_PORT || 27017,
  env: process.env.MONGO_ENV || 'local',
};

let dbUser
if (process.env.MONGO_USER && process.env.MONGO_PASS) {
  dbUser = {user: process.env.MONGO_USER, pass:
```



```
    process.env.MONGO_PASS}
  } else {
    dbUser = undefined; // on local dev not required
  }
  mongoose.connect(`mongodb://${conf.hostname}:${conf.port}/${conf.env}
    &grave;; dbUser);
```

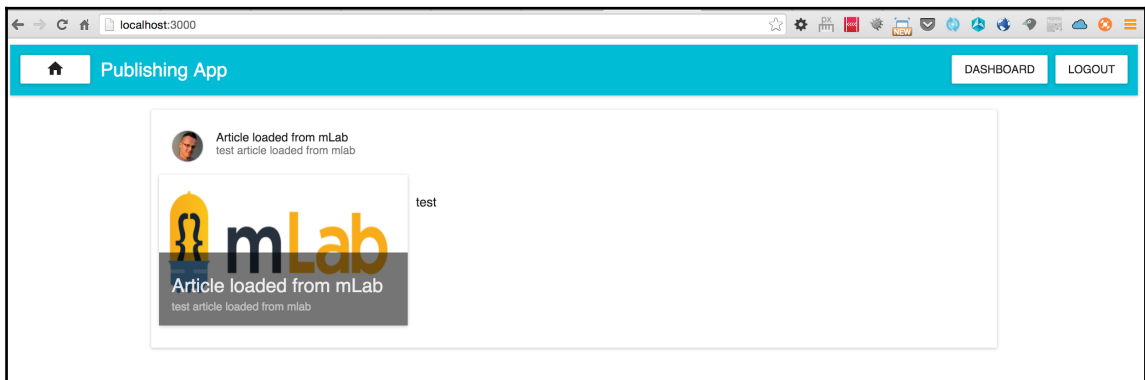
As you can see, we have added the ability to connect with a specific DB's user. We need it because the localhost on which we were working didn't require any user, but when we start using the mLab MongoDB, then specifying our database's user is a must. Otherwise, we won't be able to authenticate correctly.

From this point, you don't need to run the `mongod` process in the background of your system, because the app will connect with the mLab MongoDB node that you created in the previous chapter. The mLab MongoDB (free version) runs 24/7, but if you plan to use it for production-ready apps, then you need to update it and start using the replica set feature as well (which was mentioned in the previous chapter).

You can try to run the project with this command:

```
npm start
```

Then you should be able to load the app:



The important difference now is that all the CRUD operations (read/write via our publishing app) are done on our remote MongoDB (not on the local one).

After the publishing app uses the mLab MongoDB, we are ready to prepare our Docker image and then deploy it on several instances of AWS EC2 with AWS Load Balancer and EC2 Container Service.

Working on the publishing app Docker image

Before continuing, you should be able to run your project locally by using remote mLab MongoDB. It's required because we will start running our publishing app in the Docker container. Our app will then connect with Mongo remotely. We won't run any MongoDB process in any Docker container. This is why it's so important to use mLab in the following steps.

Let's create the Dockerfile by executing the following command in a terminal/command line:

```
[[your are in the project main directory]]
$ touch Dockerfile
```

Enter the following content in your new Dockerfile:

```
FROM centos:centos7

RUN yum update -y
RUN yum install -y tar wget
RUN wget -q https://nodejs.org/dist/v4.0.0/node-v4.0.0-linux-x64.tar.gz -O
- | tar xzf - -C /opt/

RUN mv /opt/node-v* /opt/node
RUN ln -s /opt/node/bin/node /usr/bin/node
RUN ln -s /opt/node/bin/npm /usr/bin/npm

COPY . /opt/publishing-app/

WORKDIR /opt/publishing-app

RUN npm install
RUN yum clean all

EXPOSE 80
CMD ["npm", "start"]
```

Let's look step by step at the Dockerfile we are going to use in our publishing app together with Docker:

- `FROM centos:centos7`: This says that we will use as a starting point the CentOS 7 Linux distribution from the https://hub.docker.com/r/_/centos/public Docker repository.

You can use any other package as a starting point, such as Ubuntu, but we are using CentOS 7 because it's more lightweight and generally very good for web app deployment. You can find further details at <https://www.centos.org/>.



Documentation of all commands is available at <https://docs.docker.com/engine/reference/builder/>.

- `RUN yum update -y`: This updates packages from the command line with `yum-standard` for any Linux setup.
- `RUN yum install -y tar wget`: This installs two packages as `tar` (for unpacking files) and `wget` (for downloading files).
- `RUN wget -q https://nodejs.org/dist/v4.0.0/node-v4.0.0-linux-x64.tar.gz -O - | tar xzf - -C /opt/*`: This command downloads `node4.0.0` to our CentOS container, unpacks it, and puts all the files into the `/opt/` directory.
- `RUN mv /opt/node-v /opt/node*`: This renames the folder we just downloaded and unpacked (with `node`) to simply `node` without version naming.
- `RUN ln -s /opt/node/bin/node /usr/bin/node`: We are linking the `/opt/node/bin/node` location with a `/usr/bin/node` link, so we are able to use a simple `$ node` command in the Terminal. This is standard stuff for Linux users.
- `RUN ln -s /opt/node/bin/npm /usr/bin/npm`: The same as with `node`, but with `npm`. We are linking it in order to make usage easier and linking it to `$ npm` on our CentOS 7.
- `COPY . /opt/publishing-app/`: This copies all the files in the context (The `.` (dot) sign is the location when you start the container build. We will do that in a moment.) It copies all the files into the `/opt/publishing-app/` location in our container.

In our case, we have created the `Dockerfile` in our publishing app's directory, so it will copy all the project files in the container to the given location at `/opt/publishing-app/`.

- `WORKDIR /opt/publishing-app`: After we have our publishing app's files in our Docker container, we need to choose the working directory. It's similar to `$ cd /opt/publishing-app` on any Unix/Linux machine.

- `RUN npm install`: When we are in our working directory, which is `/opt/publishing-app`, then we run the standard `npm install` command.
- `RUN yum clean all`: We clean the `yum` cache.
- `EXPOSE 80`: We define the port that is using our publishing application.
- `CMD ["npm", "start"]`: Then, we specify how to run the application in our Docker container.

We will also create in the main project directory, a `.dockerignore` file:

```
$ [[in the main directory]]  
$ touch .dockerignore
```

The file content will be as following:

```
.git  
node_modules  
.DS_Store
```

We don't want to copy over the mentioned files (`.DS_Store` is specific to OS X).

Building the publishing app container

Currently, you will be able to build the Docker's container.

In the main directory of the project, you need to run the following:

```
docker login
```

The `login` command will prompt you to insert your Docker username and password. After you are authenticated correctly, you can run the `build` command:

```
docker build -t przeor/pub-app-docker .
```



Of course, the username and the container name combination has to be yours. Replace it with your details.

That preceding command will build the container with the use of Dockerfile commands. This is what you will see (step 1, step 2, and so on):

```
przeor@Kamils-MBP React-Convention-Book (8) $ docker build -t przeor/pub-app-docker .
Sending build context to Docker daemon 409.3 MB
Step 1 : FROM centos:centos7
----> 05188b417f30
Step 2 : RUN yum update -y
----> Using cache
----> 5d7163075f9e
Step 3 : RUN yum install -y tar wget
----> Running in 5317e091d40e
Loaded plugins: fastestmirror, ovl
Loading mirror speeds from cached hostfile
* base: ftp.pbone.net
* extras: ftp.pbone.net
* updates: ftp.pbone.net
Package 2:tar-1.26-29.el7.x86_64 already installed and latest version
Resolving Dependencies
--> Running transaction check
----> Package wget.x86_64 0:1.14-10.el7_0.1 will be installed
--> Finished Dependency Resolution

Dependencies Resolved

=====
Package              Arch             Version           Repository        Size
=====
Installing:
wget                 x86_64           1.14-10.el7_0.1   base              545 k
=====

Transaction Summary
=====
Install 1 Package

Total download size: 545 k
Installed size: 2.0 M
Downloading packages:
Running transaction check
Running transaction test
Transaction test succeeded
```

After a successful build, you will see in your Terminal/command line something similar to this:

```
[[[striped from here for the sake of brevity]]]
Step 12 : EXPOSE 80
----> Running in 081e0359cbd5
----> ce0433b220a0
Removing intermediate container 081e0359cbd5
Step 13 : CMD npm start
----> Running in 581df04c8c81
----> 1970dde57fec
Removing intermediate container 581df04c8c81
Successfully built 1910dde57fec
```

As you can see here from the Docker Terminal, we have built a container in a successful manner. The next step is to test it locally and then learn a little bit more of Docker's basics and finally start working on our AWS deployment.

Running the publishing app container locally

In order to test whether the container has been built correctly, perform the following steps.

Run this command:

```
$ docker-machine env
```

The preceding command will give you output similar to this:

```
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/przeor/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
# eval $(docker-machine env)
```

We are looking for the `DOCKER_HOST` IP address; in this case, it's `192.168.99.100`.

This Docker host IP will be used to check whether our application is running correctly in the container. Note it down.

The next step is to run our local container with the following command:

```
$ docker run -d -p 80:80 przeor/pub-app-docker npm start
```

Regarding flags: the `d` flag stands for "detached," so the process will run in the background. You can list all running Docker processes with the following command:

```
docker ps
```

An example output would be as follows:

przeor@Kamils-MBP React-Convention-Book (8) \$ docker ps						
CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
2b38ecab82da	przeor/pub-app-docker	"npm start"	2 seconds ago	Up 1 seconds	0.0.0.0:80->3000/tcp	prickly_morse

The `-p` flag is telling us that the container's port 80 is bound to port 80 on the Docker IP host. So if we expose our Node app on port 80 in the container, then it will be able to run on a standard port 80 on the IP (in the examples, it will be `192.168.99.100:80`; obviously, port 80 is for all HTTP requests).

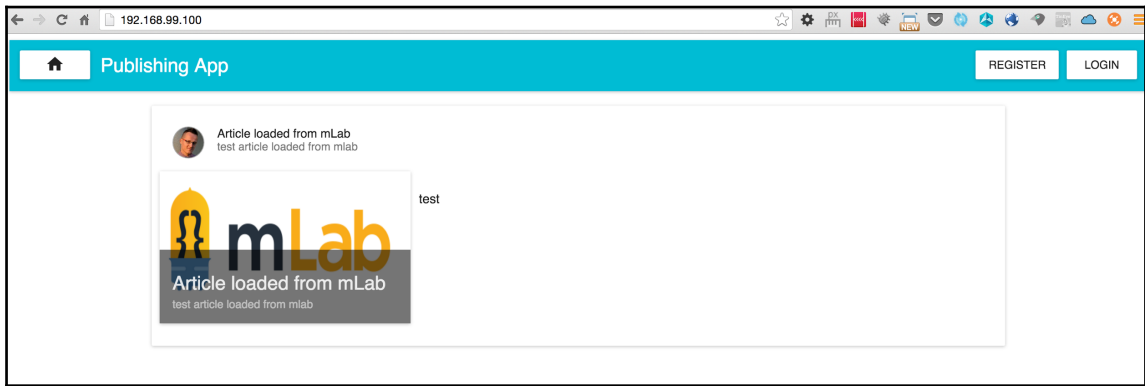
The `przeor/pub-app-docker` command will specify the name of the container that we want to run.

With `npm start`, we tell which command we want to run in the Docker container right after starting (otherwise, the container would run and stop immediately).



More references about `docker run` are available at <https://docs.docker.com/engine/reference/run/>.

The preceding command will run the app, as shown in the following screenshot:



As you can see, the IP address in the browser's URL bar is **`http://192.168.99.100`**. It's our Docker host IP.

Debugging a container

In case the container doesn't work for you, like in the following screenshot, use the following command to debug and find the reason:

```
docker run -i -t -p 80:80 przeor/pub-app-docker
```

This command with the `-i -t -p` flags will show you all the logs in the Terminal/command line, like in the following screenshot (this is just an example in order to show you the ability to debug a Docker container locally):

```
przeor@Kamils-MBP React-Convention-Book (8) $ docker run -i -t -p 80:3000 przeor/pub-app-docker
> React-Convention-Book@1.0.0 start /opt/publishing-app
> npm run webpack; node server

> React-Convention-Book@1.0.0 webpack /opt/publishing-app
> webpack --config ./webpack.config.js

Hash: d0f197d6c3f9b95330e5
Version: webpack 1.13.1
Time: 989ms
  Asset      Size  Chunks             Chunk Names
  app.js  257 kB          0  [emitted]  main
    [0] multi main 40 bytes {0} [built] [1 error]
    + 298 hidden modules

ERROR in multi main
Module not found: Error: Cannot resolve 'file' or 'directory' ./src/App.js in /opt/publishing-app
 @ multi main
    Started on port 3000
```

Pushing a Docker container to a remote repository

If a container works for you locally, then it's almost ready for AWS deployment.

Before pushing the container, let's add the `.env` file to `.dockerignore`, because you have in it all the sensitive data that you won't put into containers. So, into the `.dockerignore` file, add the following:

```
.git
node_modules
.DS_Store
.env
```

After you add `.env` to `.gitignore`, we need to change the `server/index.js` file and add an additional `if` statement:

```
if(!process.env.PORT) {
  var env = require('node-env-file');
  // Load any undefined ENV variables form a specified file.
  env(__dirname + './.env');
}
```


This `if` statement checks whether we're running the app locally (with an `.env` file) or remotely on an AWS instance (then we pass the `env` variables in a more secure manner).

After you have added the `.env` file into `.dockerignore` (and modified `server/index.js`), build the container that will be ready for the push:

```
docker build -t przeor/pub-app-docker
```

Regarding the environment variables, we will add them via AWS advanced options. You will learn about this later, but to get a general idea of how to add them when running them on the localhost, check out the following example (fake data provided in the command's flag):

```
$ docker run -i -t -e PORT=80 -e AWS_ACCESS_KEY_ID='AKIMOCKED5JM4VUHA' -e
AWS_SECRET_ACCESS_KEY='k3JxMOCKED0oRI6w3ZEmENE1I01' -e
AWS_BUCKET_NAME='publishing-app' -e AWS_REGION_NAME='eu-central-1' -e
MONGO_USER='usermlab' -e MONGO_PASS='MOCKEDpassword' -e MONGO_PORT=25732 -e
MONGO_ENV='publishingapp' -e MONGO_HOSTNAME='ds025761.mlab.com' -p 80:80
przeor/pub-app-docker
npm start
```



Make sure that you have provided your correct `AWS_REGION_NAME`. Mine is `eu-central-1`, but yours can be different.

As you can see, everything from the `server/.env` file has been moved to the Docker run command in the Bash terminal:

```
AWS_ACCESS_KEY_ID=<<__AWS_ACCESS_KEY_ID__>>
AWS_SECRET_ACCESS_KEY=<<__AWS_SECRET_ACCESS_KEY__>>
AWS_BUCKET_NAME=publishing-app
AWS_REGION_NAME=eu-central-1
MONGO_USER=<<__your_mlab_mongo_user__>>
MONGO_PASS=<<__your_mlab_mongo_pass__>>
MONGO_PORT=<<__your_mlab_mongo_port__>>
MONGO_ENV=publishingapp
MONGO_HOSTNAME=<<__your_mlab_mongo_hostname__>>
PORT=80
```

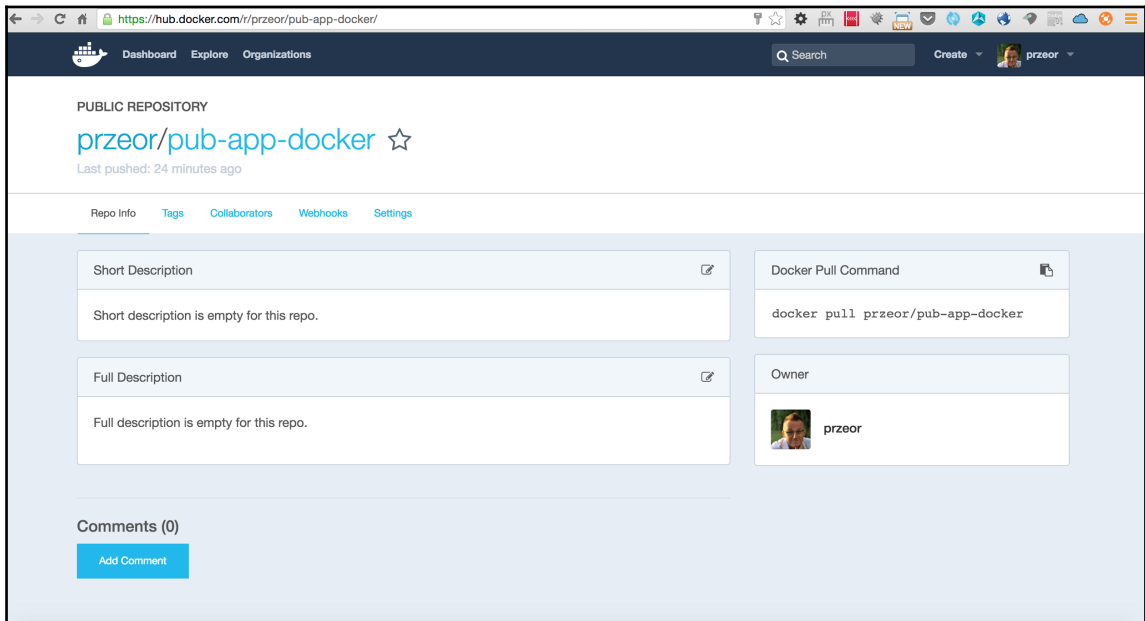
As you can find here, the `-e` flag is for an `env` variable. The last thing is to push the container to the remote repository hosted by Docker Hub:

```
docker push przeor/pub-app-docker
```

Then you will be able to find in your Bash/command line something similar to the following:

```
przeor@Kamils-MBP React-Convention-Book (8) $ docker push przeor/pub-app-docker
The push refers to a repository [docker.io/przeor/pub-app-docker]
33ba8800e2ea: Pushing [=====>] 1.693 MB
eff91c3da535: Pushed
2b72ae476cfe: Pushing [>] 1.604 MB/289.9 MB
a7594ed2e23f: Pushed
a42a2f35e8e3: Pushed
3d63725309e1: Pushing [=>] 1.089 MB/34.63 MB
7c84b3606af5: Waiting
eec7a88cba44: Waiting
acbd44cc3d10: Waiting
c4af1604d3f2: Waiting
```

The link to the pushed repo will be similar to this one:



The preceding screenshot has been made from the pushed Docker repository.

A summary of useful Docker commands

The following are a few useful Docker commands:

- This command will list all the images, and `docker rm` can delete the repo from your local machine in case you want to delete it:

```
docker images
docker rm CONTAINER-ID
```

- You can use just the first three characters from `CONTAINER-ID`. You don't need to write down whole container ID. This is a convenience.
- This one is used for stopping a running Docker container:

```
docker ps
docker stop CONTAINER-ID
```

- You can use version tag of your containers with the following approach:

```
docker tag przeor/pub-app-docker:latest przeor/pub-app-
docker:0.1
docker images
```

- After you have listed the Docker images, you may notice that you have two containers, one with the tag `latest` and the other with `0.1`. This is a way to track changes, because if you push the container, the tag will also be listed on Docker Hub.
- Check your container's local IP:

```
$ docker-machine env
```

- Build your container from a Dockerfile:

```
docker build -t przeor/pub-app-docker .
```

- Run your container in "detached" mode:

```
$ docker run -d -p 80:80 przeor/pub-app-docker npm start
```

- Run your container to debug it without detaching it so that you can find what is going on in the container's Bash terminal:

```
docker run -i -t -p 80:80 przeor/pub-app-docker
```

Introduction to Docker on AWS EC2

Two chapters ago, we implemented Amazon AWS S3 for static image uploading. You should already have an AWS account, so you are ready for the following steps to create our deployment on AWS.

In general, you can use the steps with free AWS tiers, but we will use the paid version in this tutorial. Read the AWS EC2 pricing before starting this section on how to deploy Docker containers on AWS.

AWS also has great Docker container support with their service called **EC2 Container Service (ECS)**.

If you bought this book, it probably means you haven't been using AWS so far. Because of this, we will first deploy Docker manually on EC2 in order to show you how the EC2 instances work so that you can get more knowledge from the book.

Our main goal is to make the deployment of our Docker containers automatic, but for now, we will start with a manual approach. If you have already used EC2, you can skip the next subsection and go straight to ECS.

Manual approach - Docker on EC2

We were running our Docker container locally with the following command (a few pages previously):

```
$ docker run -d -p 80:80 przeor/pub-app-docker npm start
```

We will do the same thing, not locally but on the EC2 instance, 100% manually for now; later, we will do it 100% automatically with AWS ECS.

Before we continue, let's understand what EC2 is. It's a scalable computing capacity located in the Amazon Web Services cloud. In EC2, you don't need to invest money upfront in buying any hardware. Everything you pay is for the time spent using an EC2 instance. This allows you to deploy applications faster. Very quickly, you can add new virtual servers (when there is a bigger web traffic demand). There are some mechanisms to scale the number of EC2 instances automatically with the use of **AWS CloudWatch**. Amazon EC2 gives you the ability to scale up or down to handle changed requirements (such as spikes in popularity)--this feature reduces your need to forecast traffic (and saves you time and money).

For now, we will use only one EC2 instance (later in the book, we will see more EC2 instances with load balancers and ECS).

Basics - launching an EC2 instance

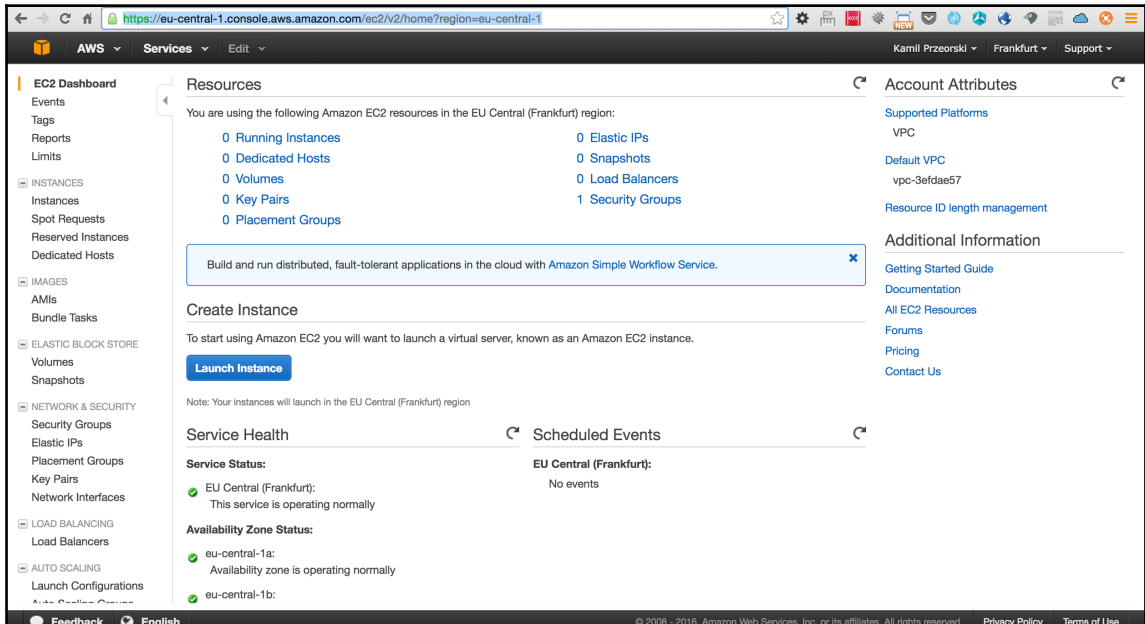
We will launch an EC2 instance, then log in to it via SSH (you can use **PuTTY** on Windows OS).

Log in to AWS Console by visiting this link:

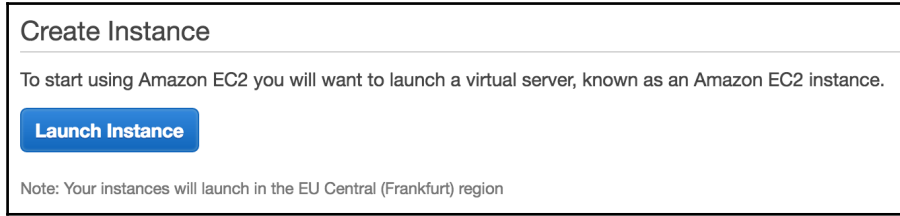
<https://eu-central-1.console.aws.amazon.com/console/home>.

Click on the EC2 link: <https://eu-central-1.console.aws.amazon.com/ec2/v2/home>

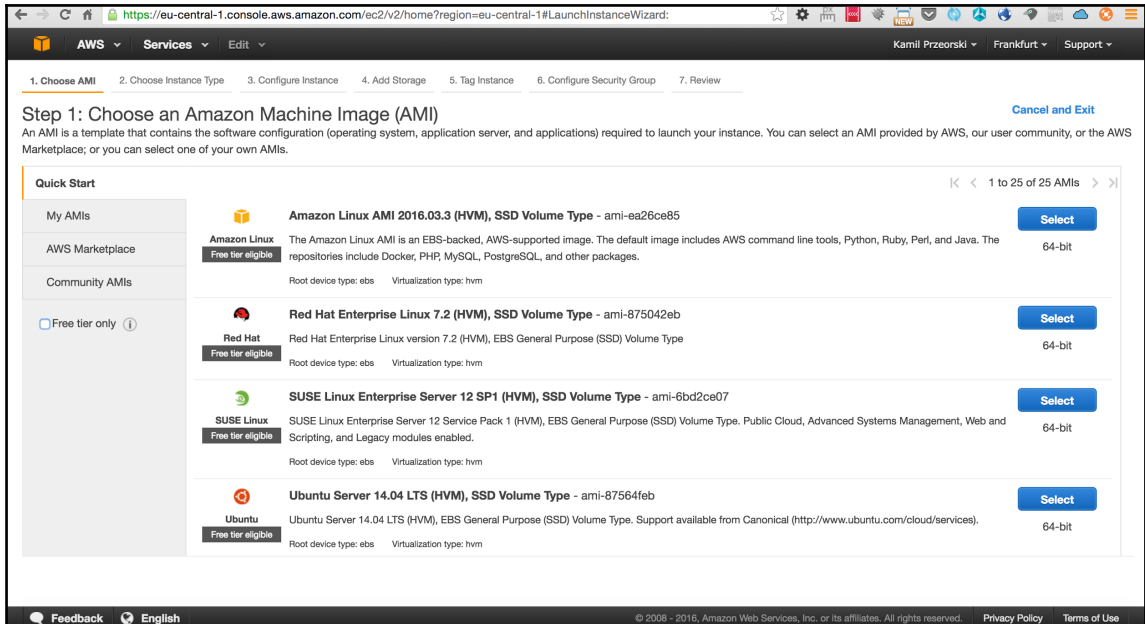
Then click on the blue **Launch Instance** button:



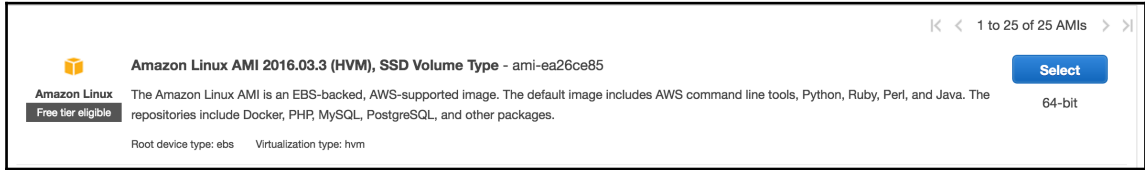
The button looks like this:



After you click on the button, you will be redirected to the **Amazon Machine Image (AMI)** page:

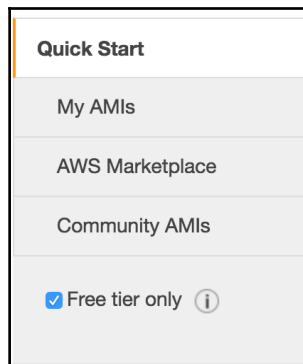


The AMI has a list of images that you can run an EC2 instance with. Each image has a list of preinstalled software. For example, the most standard image is the following:



It has preinstalled software; for example, the Amazon Linux AMI is an EBS-backed, AWS-supported image. The default image includes AWS command-line tools, Python, Ruby, Perl, and Java. The repositories include Docker, PHP, MySQL, PostgreSQL, and other packages.

On the same page, you can also find other AMIs to buy on the marketplace or created and shared by the community for free. You can also filter the images so that it will list only the free tier:

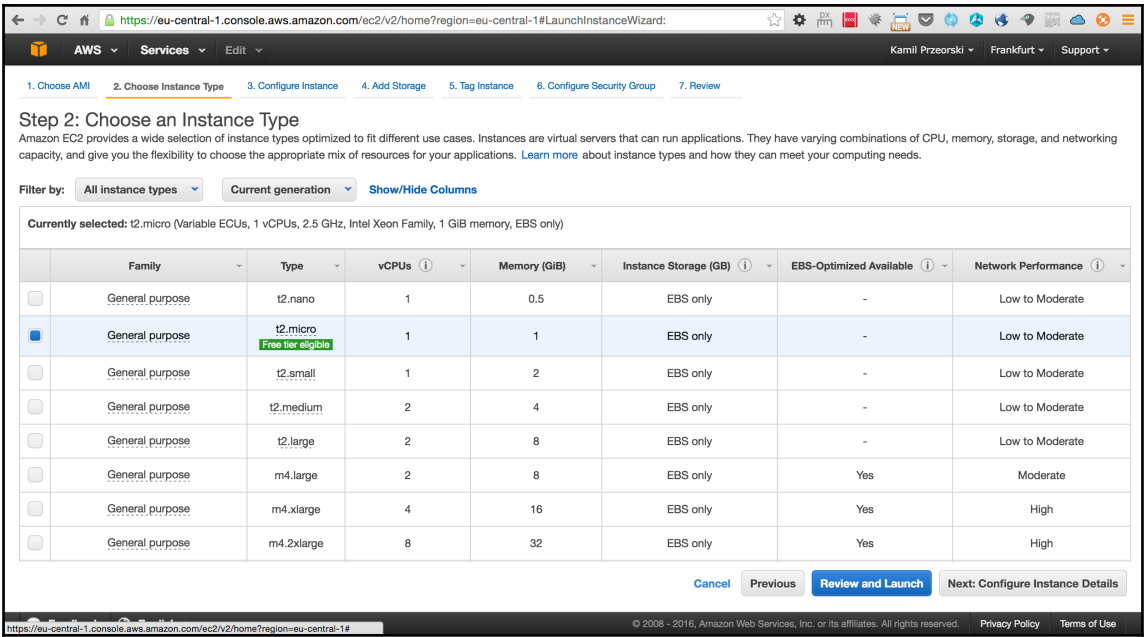


For the sake of making this step-by-step guide simple, let's choose the image that is in the preceding screenshot; its name will be similar to Amazon Linux AMI 2016.03.3 (HVM), SSD Volume Type.



The name of the image may slightly vary; don't worry about it.

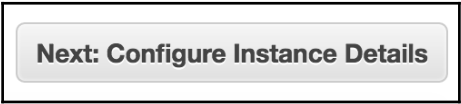
Click on the blue **Select** button. Then you will be transferred to the **Step 2: Choose an Instance Type** page, as shown in the following screenshot:



From this page, select the following:

<input checked="" type="checkbox"/>	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate
-------------------------------------	-----------------	--------------------------------	---	---	----------	---	-----------------

Then, click on this button:

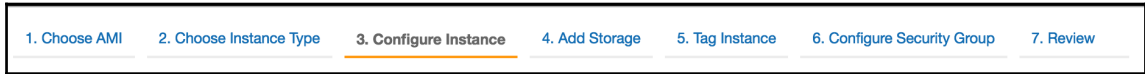


The simplest method is to choose the default options:

1. Review.
2. Configure security group (we will make some changes in this tab).
3. Tag instance (keep the options default).
4. Add storage (keep the options default).
5. Configure the instance (keep the options default).

6. Choose an instance type.
7. Choose an AMI.
8. Generally, keep clicking on the next button until we get to the Configure security.

An indicator of progress you can find at the top is this:



Our goal for now is to get to the security configuration page because we need to customize slightly the allowed ports. A security group consists of rules that control network traffic for an EC2 instance (a.k.a. firewall options). For security, set the name to `ssh-and-http-security-group`:

Step 6: Configure Security Group
A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. [Learn more](#) about Amazon EC2 security groups.

Assign a security group: ☒ Create a new security group
☐ Select an existing security group

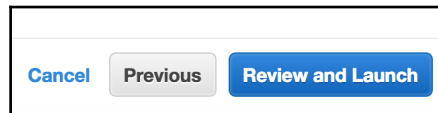
Security group name:
Description:

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ
SSH	TCP	22	Anywhere 0.0.0.0/0
HTTP	TCP	80	Anywhere 0.0.0.0/0

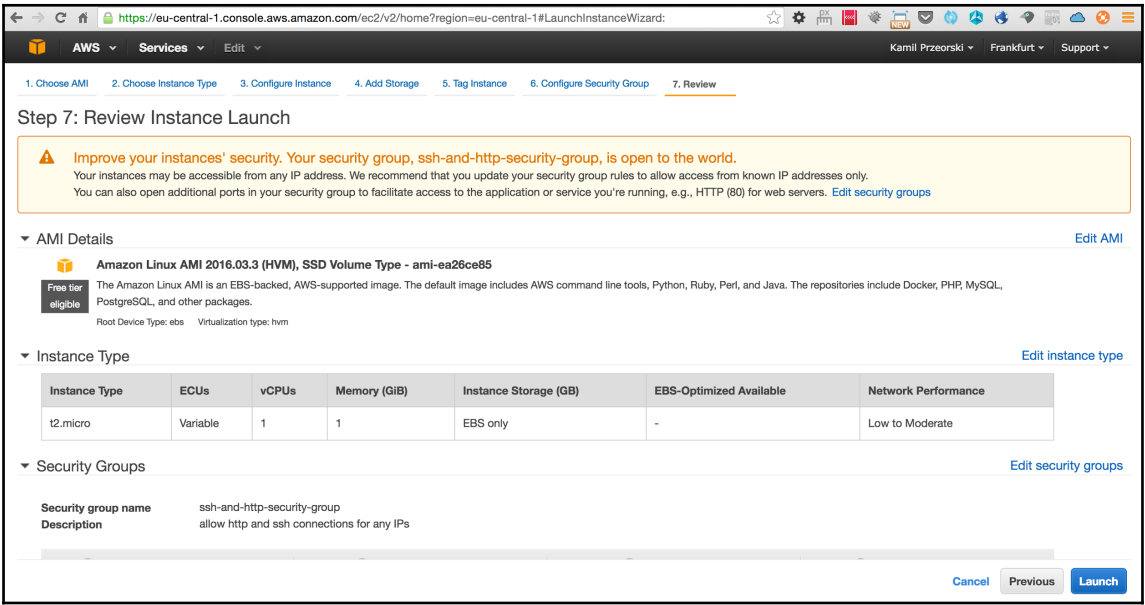
Warning
Rules with source of 0.0.0.0/0 allow all IP addresses to access your instance. We recommend setting security group rules to allow access from known IP addresses only.

As you can find here, you also need to click on the **Add Rule** button and add a new one called **HTTP**. This will allow our new EC2 instance to be available via port **80** for all the IPs.

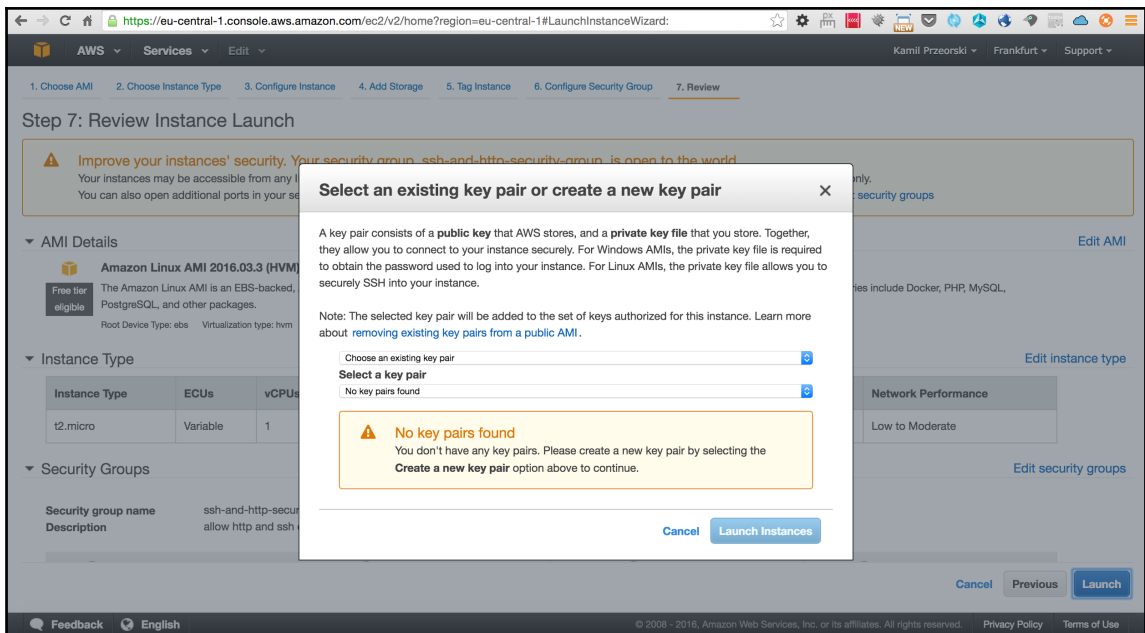
After you have added the name and HTTP port 80 as the new rule, you can click on the **Review and Launch** button:



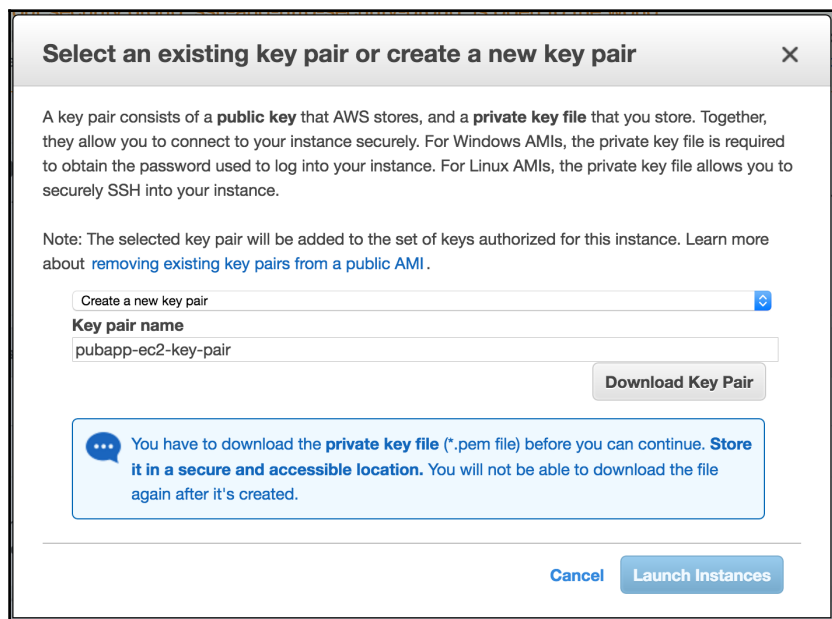
Then, after you are happy with reviewing the instance, click on the blue button called **Launch** in that view:



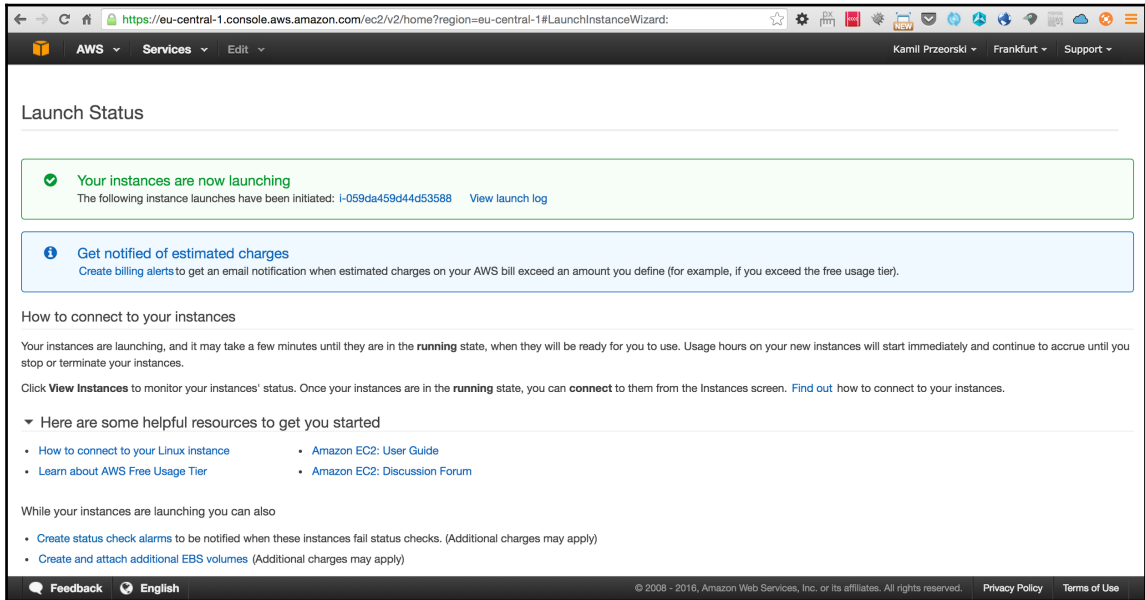
After you click on the **Launch** button, you will see a modal that says **Select an existing key pair or create a new key pair**:



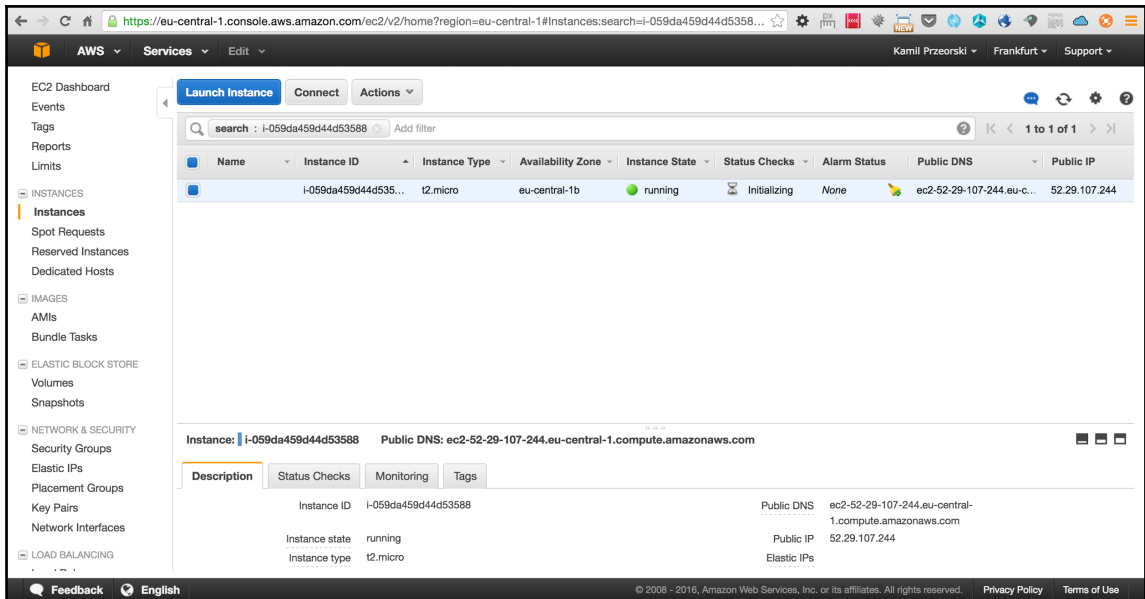
Usually, you'll need to create a new key pair. Give it the name `pubapp-ec2-key-pair` and then click on the **Download** button, as shown in the following screenshot:



After you have downloaded `pubapp-ec2-key-pai`, you will be able to click on the blue **Launch** button. Next, you will see the following:



From this screen, you can go directly to the EC2 launch logs (click on the **View launch log** link) so that you will be able to find your instance listed, as seen in the following screenshot:



Great. Your first EC2 has been launched successfully! We need to log in to it and set up the Docker container from there.

Save the public IP of your EC2 instance. In the preceding launch log, you can find that the machine we've just created has the public IP **52.29.107.244**.

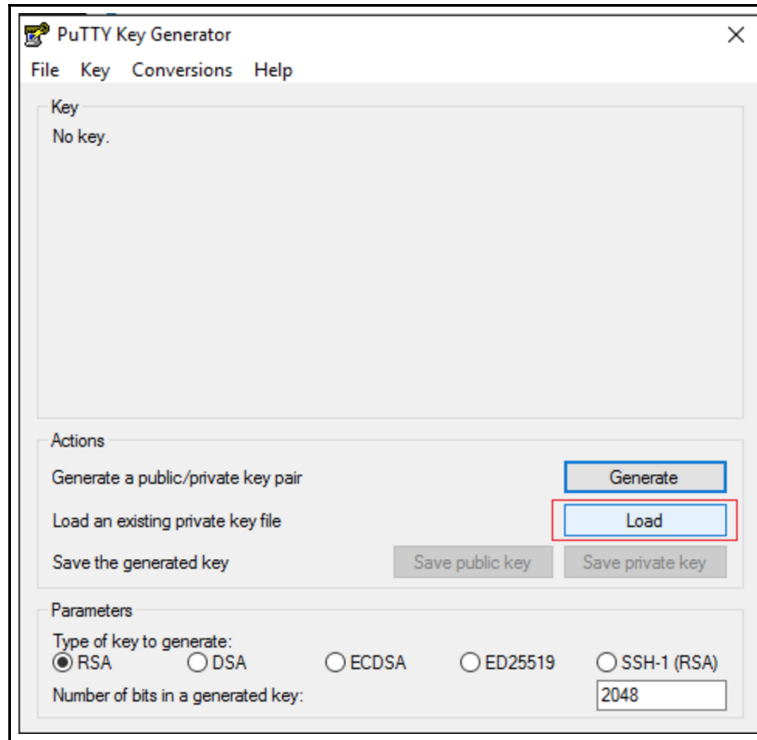
Your IP will be different (of course, this is just an example). Save it somewhere; we will use it in a moment as you'll need it to log in via SSH to the server and install the Docker app.

SSH access via PuTTY - Windows users only

If you don't work on Windows, you can skip this subsection.

We'll be using PuTTY, which is available for download at <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html> (putty.exe, pageant.exe, and puttygen.exe).

Download key pairs for the EC2 instance, and convert them to ppk using `puttygen.exe`:

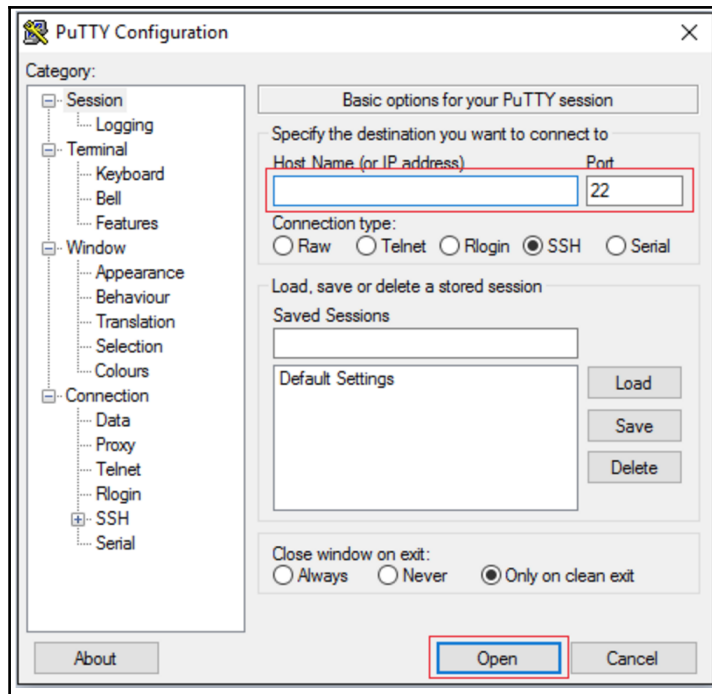


Click on the **Load** button and choose the `pubapp-ec2-key-pair.pem` file, and then convert it to ppk.

Then you need to click on the **Save private key** button. You are done; you can close `puttygen.exe` and open `pageant.exe`. From it, do the following:

- Choose **Add Key**
- Then check whether your key has been added correctly to the Pageant key list

If your private key is on the list, you are ready to use `putty.exe`.



If you have opened the PuTTY program, you need to log in via SSH by typing your EC2 instance IP and clicking on the **Open** button, as shown in the preceding screenshot. PuTTY allows using SSH connections on Windows.

Connecting to an EC2 instance via SSH

In a previous chapter, after we launched the EC2 instance, we found out our public IP (remember that your public IP will be different): `52.29.107.244`. We need to connect to the remote EC2 instance with this public IP.

I've saved `pubapp-ec2-key-pair.pem` in my Downloads directory, so go to the directory where you have downloaded your `.pem` file:

```
$ cd ~/Downloads/
$ chmod 400 pubapp-ec2-key-pair.pem
$ ssh -i pubapp-ec2-key-pair.pem ec2-user@52.29.107.244
```



In PuTTY on Windows, it will look similar after this step. You need to provide in the PuTTY box the IP and ports in order to correctly log in to the machine. When you get a prompt to type a username, use `ec2-user`, as in the SSH example.

After a successful login, you will be able to see this:

```
przeor@Kamils-MBP Downloads $ chmod 400 pubapp-ec2-key-pair.pem
przeor@Kamils-MBP Downloads $ ssh -i pubapp-ec2-key-pair.pem ec2-user@52.29.107.244

  _ | _ | _ |
  _ | ( _ | /   Amazon Linux AMI
  _ | \ _ | _ |

https://aws.amazon.com/amazon-linux-ami/2016.03-release-notes/
1 package(s) needed for security, out of 1 available
Run "sudo yum update" to apply all updates.
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file or directory
[ec2-user@ip-172-31-26-81 ~]$
```

The following instructions are for all OS users (OS X, Linux, and Windows) as we are logged in to the EC2 instance via SSH. The following commands are required next:

```
[ec2-user@ip-172-31-26-81 ~]$ sudo yum update -y
[ec2-user@ip-172-31-26-81 ~]$ sudo yum install -y docker
[ec2-user@ip-172-31-26-81 ~]$ sudo service docker start
```

These commands will update the `yum` package manager and install and start the Docker service in the background:

```
[ec2-user@ip-172-31-26-81 ~]$ sudo usermod -a -G docker ec2-user
[ec2-user@ip-172-31-26-81 ~]$ exit

> ssh -i pubapp-ec2-key-pair.pem ec2-user@52.29.107.244
[ec2-user@ip-172-31-26-81 ~]$ docker info
```


After you run the `docker info` command, it will show something similar to the following output:

```
[ec2-user@ip-172-31-26-81 ~]$ docker info
Containers: 0
  Running: 0
  Paused: 0
  Stopped: 0
Images: 0
Server Version: 1.11.1
Storage Driver: devicemapper
  Pool Name: docker-202:1-263714-pool
  Pool Blocksiz: 65.54 kB
  Base Device Size: 10.74 GB
  Backing Filesystem: xfs
  Data file: /dev/loop0
  Metadata file: /dev/loop1
  Data Space Used: 11.8 MB
  Data Space Total: 107.4 GB
  Data Space Available: 6.971 GB
  Metadata Space Used: 581.6 kB
  Metadata Space Total: 2.147 GB
  Metadata Space Available: 2.147 GB
  Udev Sync Supported: true
  Deferred Removal Enabled: false
  Deferred Deletion Enabled: false
  Deferred Deleted Device Count: 0
  Data loop file: /var/lib/docker/devicemapper/devicemapper/data
WARNING: Usage of loopback devices is strongly discouraged for production use. Either use `--storage-opt dm.thinpooldev` or use `
loop_devices=true` to suppress this warning.
  Metadata loop file: /var/lib/docker/devicemapper/devicemapper/metadata
  Library Version: 1.02.93-RHEL7 (2015-01-28)
```

If you look at the preceding screenshot, you'll see that everything is all right, and we can continue with running the publishing app's Docker container with the following command:

```
[ec2-user@ip-172-31-26-81 ~]$ docker run -d PORT=80 -e
AWS_ACCESS_KEY_ID='AKIMOCKED5JM4VUHA' -e
AWS_SECRET_ACCESS_KEY='k3JxMOCKED0oRI5w3ZEmENE1I01' -e
AWS_BUCKET_NAME='publishing-app' -e AWS_REGION_NAME='eu-central-1' -e
MONGO_USER='usermlab' -e MONGO_PASS='MOCKEDpassword' -e MONGO_PORT=25732 -e
MONGO_ENV='publishingapp' -e MONGO_HOSTNAME='ds025761.mlab.com' -p 80:80
przeor/pub-app-docker npm start
```



Make sure you have provided your correct `AWS_REGION_NAME`. Mine is `eu-central-1`, but yours could be different.

As you can see, everything from the `server/.env` file has been moved to the `docker run` command in the Bash terminal:

```
AWS_ACCESS_KEY_ID=<<__AWS_ACCESS_KEY_ID__>>
AWS_SECRET_ACCESS_KEY=<<__AWS_SECRET_ACCESS_KEY__>>
AWS_BUCKET_NAME=publishing-app
AWS_REGION_NAME=eu-central-1
MONGO_USER=<<__your_mlab_mongo_user__>>
MONGO_PASS=<<__your_mlab_mongo_pass__>>
MONGO_PORT=<<__your_mlab_mongo_port__>>
MONGO_ENV=publishingapp
MONGO_HOSTNAME=<<__your_mlab_mongo_hostname__>>
PORT=80
```

Also make sure to rename `AWS_BUCKET_NAME`, `AWS_REGION_NAME`, or `MONGO_ENV` if you have a different one (if you set it differently than what was suggested in the previous chapters).

Then, in order to check whether everything went well, you can also use the following:

```
[ec2-user@ip-172-31-26-81 ~]$ docker ps
```

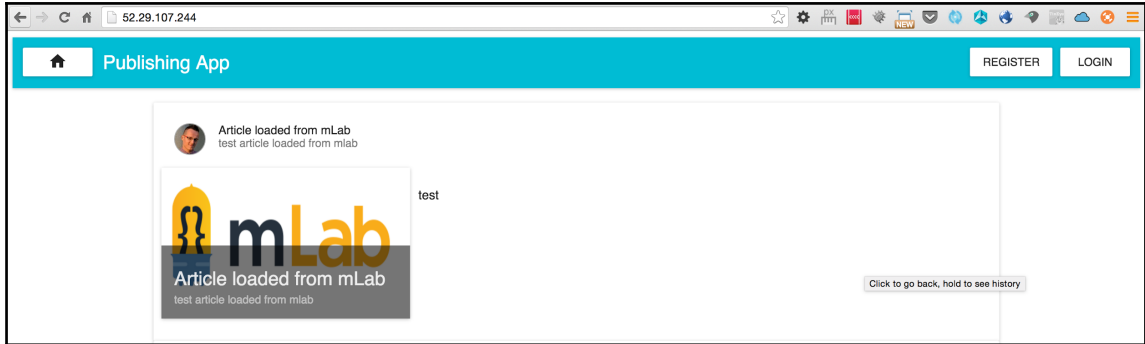
This command will show you whether the Docker process runs correctly in the background as a detached container. And after 10-30 seconds, when `npm start` will run the whole project, you can test with this:

```
[ec2-user@ip-172-31-26-81 ~]$ curl http://localhost
```

After the application has been bootstrapped correctly, you can see output similar to the following:

```
[ec2-user@ip-172-31-26-81 ~]$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED            STATUS              PORTS               NAMES
70170999ed76       przeor/pub-app-docker  "npm start"             20 seconds ago    Up 19 seconds      0.0.0.0:80->3000/tcp  sleepy_raman
[ec2-user@ip-172-31-26-81 ~]$ curl http://localhost
<!doctype html>
<html>
<head>
<title>Publishing App Server Side Rendering</title>
<link rel="stylesheet" type="text/css" href="/static/styles-draft-js.css" />
</head>
<body>
<div id="publishingAppRoot"><div style="color:rgba(0, 0, 0, 0.87);background-color:#00bcd4;transition:all 450ms cubic-bezier(0.23, 1, 0.32, 1) 0ms;box-sizing:border-box;font-family:Roboto, sans-serif;webkit-tap-highlight-color:rgba(0,0,0,0);box-shadow:0 1px 6px rgba(0, 0, 0, 0.12), 0 1px 4px rgba(0, 0, 0, 0.12);border-radius:0px;position:relative;z-index:100;width:100%;display:-webkit-box;display:-moz-box;display:-ms-flexbox;display:-webkit-flex;min-height:64px;padding-left:24px;padding-right:24px;ui-prepared:-webkit-transition:all 450ms cubic-bezier(0.23, 1, 0.32, 1) 0ms;-moz-transition:all 450ms cubic-bezier(0.23, 1, 0.32, 1) 0ms;-ms-transition:all 450ms cubic-bezier(0.23, 1, 0.32, 1) 0ms;webkit-box-sizing:border-box;-moz-box-sizing:border-box;-ms-box-sizing:border-box;"><div style="margin-top:8px;margin-right:8px;margin-left:-16px;ui-prepared:"><a class="" href="/"></div style="color:rgba(0, 0, 0, 0.87);background-color:#ffffff;transition:all 450ms cubic-bezier(0.23, 1, 0.32, 1) 0ms;box-sizing:border-box;font-family:Roboto, sans-serif;webkit-tap-highlight-color:rgba(0,0,0,0);box-shadow:0 1px 6px rgba(0, 0, 0, 0.12), 0 1px 4px rgba(0, 0, 0, 0.12);border-radius:2px;display:inline-block;min-width:88px;height:36px;margin:5px;padding-top:5px;ui-prepared:-webkit-tra
```

After you visit the EC2 instance's public IP (in our example, it is 52.29.107.244), you will be able to find our publishing app available online as we have set up the security group of our EC2 instance with the exposed port 80 to the world. The following is the screenshot:



If you see our publishing app under a public IP, then you have just deployed a Docker container on Amazon AWS EC2 successfully!

The process we just went through is very inefficient and manual, but shows exactly what is going on under the hood when we start using ECS.

We are missing the following in our current approach:

- Integration with other Amazon services, such as load balancing, monitoring, alerting, crash recovery, and route 53.
- Automation, as currently we are unable to efficiently deploy 10 Docker containers quickly. This is also important if you want to deploy different Docker containers for different services as, for example, you can have separate containers for serving the frontend, backend, and even the database (in our case, we use mLab, so we don't need one here).

You've just learned the basics of Amazon Web Services.

Basics of ECS - AWS EC2

The EC2 Container Service helps you create a cluster of Docker Container instances (many copies of the same container on several EC2 instances). Each container is deployed automatically--this means you don't need to log in to any of the EC2 instances via SSH as we did it in the previous chapter (manual approach). The whole job is done by the AWS and Docker software, which you will learn to use in the future (a more automated approach).

For example, you set that you want to have five different EC2 instances--the group of EC2 instances in the exposed port 80 so you are able to find the publishing application under the `http://[[EC2_PUBLIC_IP]]` address. Additionally, we are adding a load balancer between all the EC2 instances and the rest of the world so that in case there is any spike in traffic or any of the EC2 instances break, the load balancer will replace the broken EC2 instance with a new one or scale down/up the number of EC2 instances based on the traffic.

A great feature of the AWS load balancer is that it pings each EC2 instance with port 80, and if the pinged instance doesn't respond with the correct code (200), then it terminates the broken instance and turns on a fresh new instance with the Docker Container that has the image of our publishing app. This helps us maintain continuous availability of our application.

Additionally, we will use Amazon Route 53 in order to have a highly available and scalable cloud **domain name system** (DNS) web service so we will be able to set up a top level domain; in our case, I will use a domain I have bought specially for the book:
`http://reactjs.space`.

That will be our HTTP address, of course. If you build a different service, you need to buy your own domain in order to follow the instructions and learn how Amazon Route 53 works.

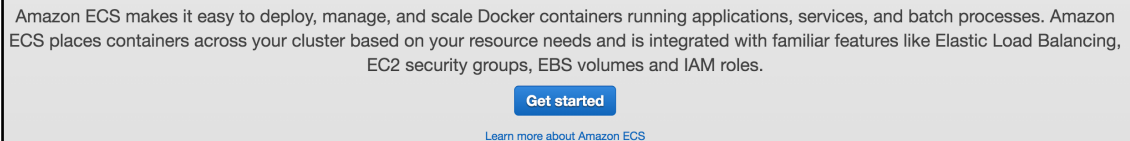
Working with ECS

Before we start working on ECS, let's understand some basic nomenclature:

- **Cluster:** This is the main part of our process that will pool underlying resources as EC2 instances and any attached storage. It clusters many EC2 instances into one containerized application that aims to be scalable.

- **Task definition:** This task determines what Docker Containers you are going to run on each EC2 instance (that is, the `docker run` command) and it also helps you define more advanced options, such as environment variables that you want to pass down into a container.
- **Service:** This is a kind of glue between the cluster and a task definition. The service handles the login of a running task on our cluster. This also contains the management of revisions of the task (combination of a container and its settings) you want to run. Every time you change any setting in your task, it creates a new revision of your task. In the service, you specify what the task is and its revision that you want to run on your EC2 instances in your ECS.

Visit the AWS Console and find the ECS. Click on the link to go to the EC2 Container Service Console. There, you will find a blue button named **Get started**:



After that, you will see an ECS wizard with the following steps:

1. Create a task definition.
2. Configure service.
3. Configure cluster.
4. Review.

Step 1 - creating a task definition

In ECS, a task definition is a recipe for a container. It's something that helps an ECS understand what Docker Container you want to run on the EC2 instances. It's a recipe or a blueprint of steps that the ECS has automatically done in order to successfully deploy our publishing app's container.

The details for this step are shown in the following screenshot:

Create a task definition

An Amazon ECS task definition is a blueprint or recipe for containers. You can modify parameters in the task definition to suit your particular application (for example, to provide more CPU resources or change the port mappings). [Learn more](#)

Task definition name*

pubapp-task

Container name*

pubapp-container

Image*

przeor/pub-app-docker

Custom image format: [registry-url]/[namespace]/[image]:[tag]

Maximum memory (MB)*

300

The amount of allocated memory for your container. ECS recommends 300-500 MB as a starting point for web applications.

Port mappings

Host port	Container port	Protocol	
80	80	tcp	

+ Add port mapping

In the preceding screenshot, you can find that our task definition name is `pubapp-task`. The container name is `pubapp-container`.

For **Image**, we use the same argument as when we were running a container locally with `docker run`. In the case of `przeor/pub-app-docker`, ECS will know that it has to download the container from `https://hub.docker.com/r/przeor/pub-app-docker/`.

For now, let's keep the maximum memory at the default value (300). Set both port mappings to 80.



At the time of writing this book, there are some problems if your container doesn't expose port 80. It's probably a bug with the ECS wizard; without the wizard, any port can be used on the container.

Click on **Advanced options** in the task definition view:

Advanced options

You will see a slide panel with additional options:

Edit container

Entry point ⓘ

Command ⓘ

Working directory ⓘ

Env Variables

Key	Value	
PORT	80	✕
AWS_SECRET_ACCESS_	[REDACTED]bjf7xR	✕
AWS_BUCKET_NAME	publishing-app	✕
AWS_ACCESS_KEY_ID	[REDACTED]VU+	✕
AWS_REGION_NAME	eu-central-1	✕
MONGO_USER	[REDACTED]	✕
MONGO_PASS	[REDACTED]	✕
MONGO_PORT	[REDACTED]	✕
MONGO_ENV	publishingapp	✕
MONGO_HOSTNAME	[REDACTED]	✕

We need to specify the following things:

- **Command:** This has to be separated with commas, so we use `npm, start`.
- **Working directory:** We use `/opt/publishing-app` (identical path is set in the Dockerfile).

- **Env variables:** Here, we specify all values from the `server/.env` file. This part is important to set up; the app will not work correctly without the correct details provided via the environment variables.
- **Rest of the values/inputs:** Keep them at the default without changes.



It's very important to add all the environment variables. We need to be very careful as it's easy to make a mistake here that will break the app inside an EC2 instance.

After all these changes, you can click on the **Next** button.

Step 2 - configuring the service

Generally, a service is a mechanism that keeps a certain amount of EC2 instances running while checking their health at the same time (using the **Elastic Load Balancing (ELB)**). ELB automatically distributes incoming application traffic across multiple Amazon EC2 instances. If a server doesn't respond on port 80 (the default but can be changed to more advanced health checks), then the service runs a new service while the unhealthy one is being shut down. This helps you maintain very high availability for your application.

Configure service

Create a name for your service and set the desired number of tasks to start with. A service auto-recovers any stopped tasks to maintain the desired number that you specify here. Later, you can update your service to deploy a new image or change the running number of tasks. [Learn more](#)

Service name*

pubapp-service

Desired number of tasks*

3

The service name is `pubapp-service`. In this book, we will set up three different EC2 instances (you can set up fewer or more; it's up to you), so this is the number for the *desired number of tasks* input.

In the same step, we also have to set up the **Elastic Load Balancer (ELB)**:

Elastic load balancing

Create an Elastic Load Balancing load balancer and configure your service to run behind it. [Learn more](#)

Container name: host port

pubapp-container:80

Configure the listener protocol and port for your load balancer. The ELB health check field is automatically populated to match the protocol and port of your load balancer.

ELB listener protocol*

HTTP

ELB listener port*

80

ELB health check

http:80/

Service IAM role

The Amazon ECS service scheduler makes calls to the Amazon EC2 and Elastic Load Balancing APIs on your behalf to register and deregister container instances with your load balancers. If you do not have the `ecsServiceRole` already, we can create one for you.

Select IAM role for service

You are giving permission to EC2 Container Service to create and use `ecsServiceRole`.

* Required

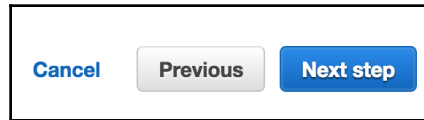
Cancel

Previous

Next step

- **Container name: host port:** Choose from the drop-down list `pubapp-container:80`
- **ELB listener protocol*:** HTTP
- **ELB listener port*:** 80
- **ELB health check:** Keep default; you can change it while you are out of the wizard (on the specific ELB's page)
- **Service IAM role:** The wizard will create this for us

After all this, you can click on the **Next step** button to continue:



Step 3 - configuring the cluster

Now, you'll set up the details of the ECS container agent, called a cluster. Here, you specify the name of your cluster, what kind of instances you'd like to use, the number of instances (it has to be bigger than the number required by the service), and the key pair.

Configure cluster

Your Amazon ECS tasks run on container instances (Amazon EC2 instances that are running the ECS container agent). Configure the instance type, instance quantity, and other details of the container instances to launch into your cluster

Cluster name*

pubapp-ecs-cluster

EC2 instance type*

t2.micro

Number of instances*

5

Key pair

pubapp-ec2-key-pair

You will not be able to SSH into your EC2 instances without a key pair. You can create a new key pair in the [EC2 console](#).

- **Cluster name:** Our cluster name is `pubapp-ecs-cluster`.
- **EC2 instance type:** `t2.micro` (in production, use a bigger one).
- **Number of instances:** Five, and that means the service will keep three instances alive and another two instances will be on the bench, waiting for any fatal situations. By bench, I mean that at a time (with our setup), we'll use only three instances, whereas another two are ready for use, but not actively used (traffic is not redirected to them).
- **Key pair:** I specified the key pair called `pubapp-ec2-key-pair` earlier in this chapter. Always keep them in a safe place for later use.

On the same page, you will also find the security group and container instance IAM roles setup, but we'll keep it at the default for now:

Security group

By default, your instances are accessible from any IP address. We recommend that you update the below security group ingress rule to allow access from known IP addresses only. ECS automatically opens up port 80 to facilitate access to the application or service you're running.

Allowed ingress source(s)*

Anywhere

0.0.0.0/0

Container instance IAM role

The Amazon ECS container agent makes calls to the Amazon ECS API actions on your behalf, so container instances that run the agent require the `ecsInstanceRole` IAM policy and role for the service to know that the agent belongs to you. If you do not have the `ecsInstanceRole` already, we can create one for you.

Container instance IAM role

You are giving permission to EC2 Container Service to create and use `ecsInstanceRole`.

* Required

Cancel

Previous

Review & launch

Step 4 - reviewing

The last thing is to review whether everything looks good:

The screenshot shows the AWS Management Console interface for the 'Getting Started with Amazon EC2 Container Service (ECS)' wizard. The 'Review' step is active, showing a summary of the configuration. On the left, a sidebar lists the steps: 'Step 1: Create a task definition', 'Step 2: Configure service', 'Step 3: Configure cluster', and 'Step 4: Review'. The main content area is titled 'Review' and includes a sub-header 'Task definition' with an 'Edit' button. Below this, the task definition details are listed: 'Task definition name' is 'pubapp-task', 'Container name' is 'pubapp-container', 'Image name' is 'przeor/pub-app-docker', 'Memory' is '300', and 'Port mappings' are 'host: 80; container: 80; protocol: tcp'. A section for 'Task definition JSON' is collapsed. Below that is the 'Service configuration' section with an 'Edit' button. The service configuration details are: 'Service name' is 'pubapp-service', 'Number of tasks' is '3', 'IAM role for ECS service' is 'ecsServiceRole', 'Container name: host port' is 'pubapp-container : 80', and 'ELB listener port' is '80'.

Task definition name	pubapp-task
Container name	pubapp-container
Image name	przeor/pub-app-docker
Memory	300
Port mappings	host: 80; container: 80; protocol: tcp

Task definition JSON

Service name	pubapp-service
Number of tasks	3
IAM role for ECS service	ecsServiceRole
Container name: host port	pubapp-container : 80
ELB listener port	80

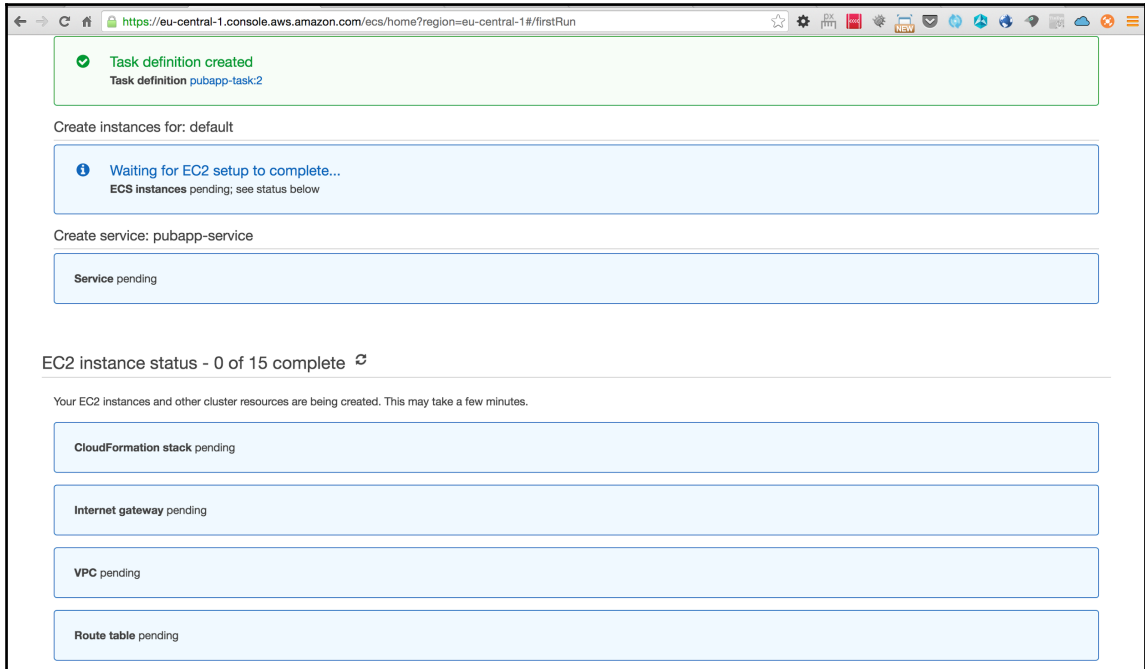
Then, choose **Launch instances & run service**:

The screenshot shows three buttons in a row: 'Cancel' (blue text), 'Previous' (grey button), and 'Launch instance & run service' (blue button with white text).

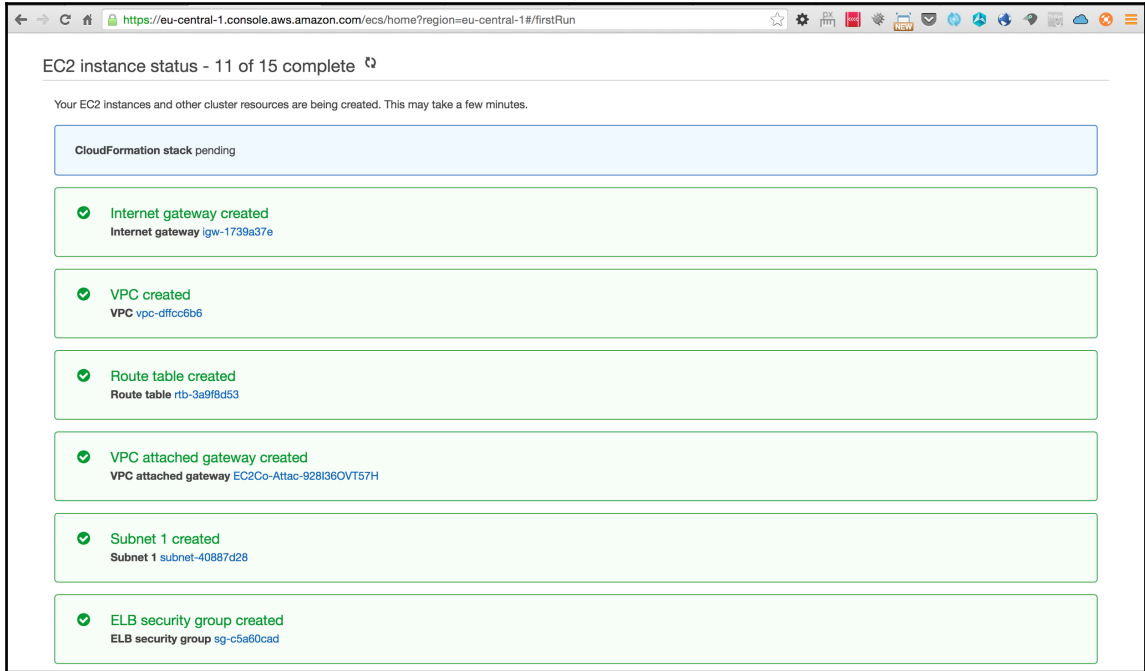
Cancel Previous Launch instance & run service

Launch status

After you have clicked on the **Launch** button, you will find a page with the status. Keep it open until you get all the boxes green with success indicators:



Here's what it looks like all up and running:



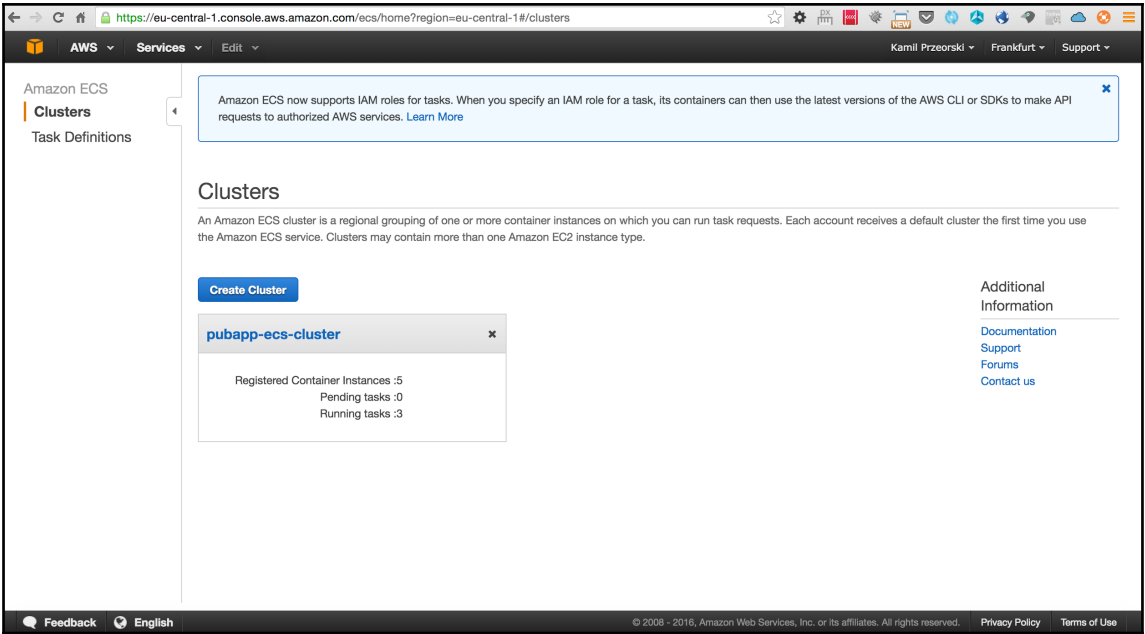
After all the boxes have a success indicator, then you will be able to click on the **View service** button that is at the top:



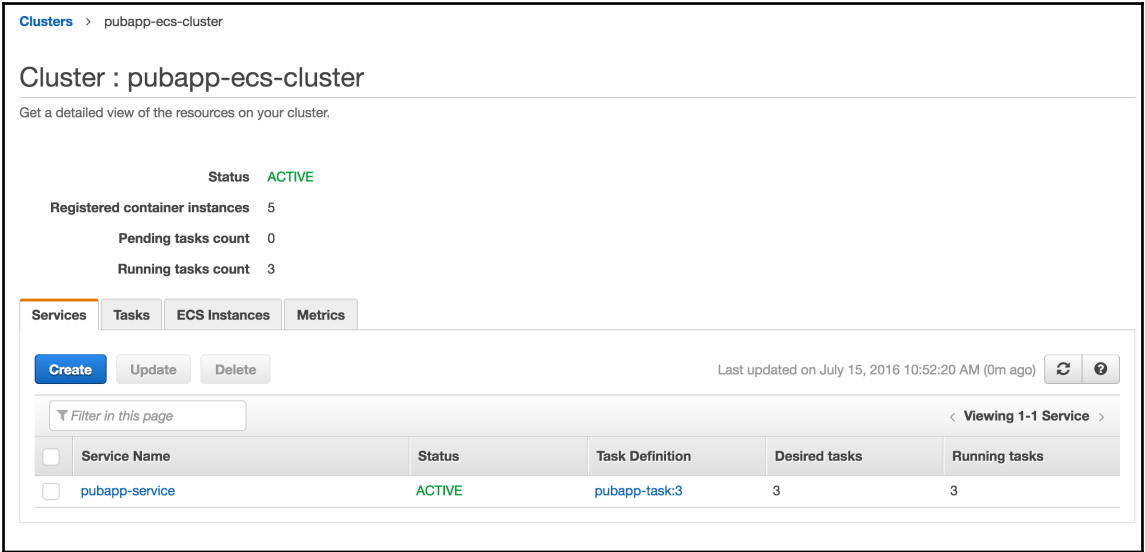
Click on that button (**View service**) after it becomes available.

Finding your load balancer address

After you click on the **View service** button, you will see the main dashboard, where all your clusters are listed (currently there will only be one):



Click on **pubapp-ecs-cluster** and you will see the following:



On the preceding screen, click on **pubapp-service** from the list:

<input type="checkbox"/>	Service Name
<input type="checkbox"/>	pubapp-service

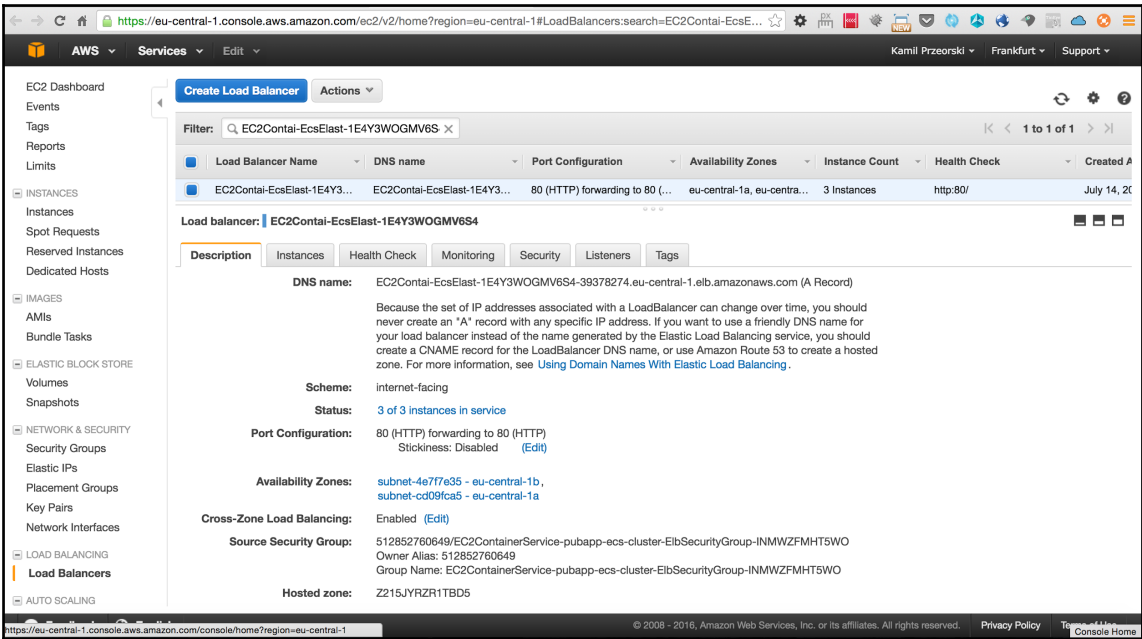
Then, you will see the following:

The screenshot shows the AWS Management Console interface for the 'pubapp-ecs-cluster' service. The left sidebar shows 'Amazon ECS' and 'Clusters'. The main content area displays the 'Service : pubapp-service' details. A notification banner at the top states: 'Amazon ECS now supports IAM roles for tasks. When you specify an IAM role for a task, its containers can then use the latest versions of the AWS CLI or SDKs to make API requests to authorized AWS services. Learn More'. Below this, the service details are shown: Cluster: pubapp-ecs-cluster, Status: ACTIVE, Task Definition: pubapp-task:3, Desired count: 3, Pending count: 0, Running count: 3, and Service Role: ecsServiceRole. To the right, the 'Load Balancing' section shows a table with one entry: Load Balancer Name: EC2Contai-EcsElast-1E4Y3WOGMV6S4, Container Name: pubapp-container, and Container Port: 80. Below this, the 'Deployment Options' section shows 'Minimum healthy percent' at 100 and 'Maximum percent' at 200. At the bottom, there are tabs for 'Tasks', 'Events', 'Deployments', and 'Metrics'. The 'Tasks' tab is active, showing a filter bar with 'Task status: Running Stopped' and a table with columns: Task, Task Definition, Last status, and Desired status. The table currently shows 1-3 tasks.

From this page, choose the Elastic Balancer:

Load Balancer Name
EC2Contai-EcsElast-1E4Y3WOGMV6S4

The final view of ELB is as follows:



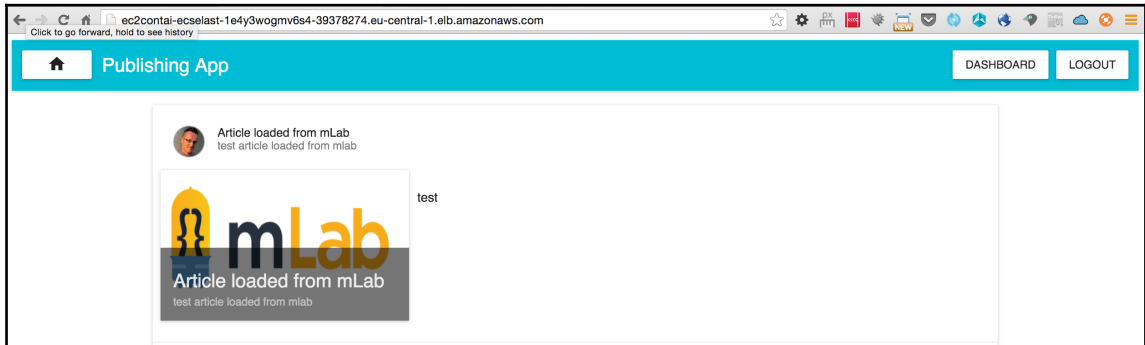
In the preceding view, you will find (under the **Description Name** tab) an elastic balancer address like this one:

DNS name:
EC2Contai-EcsElast-1E4Y3WOGMV6S4-39378274.eu-central-1.elb.amazonaws.com (A Record)



If you try to open the address and it doesn't work, then give it more time. The EC2 instances may be in progress in terms of running our Docker publishing app container. We must be patient during the initial run of our ECS cluster.

This is the address of your ELB, which you can put into the browser and see the publishing app:



AWS Route 53

The last step left in this chapter is to set up Route 53, which is a highly available and scalable cloud DNS web service.

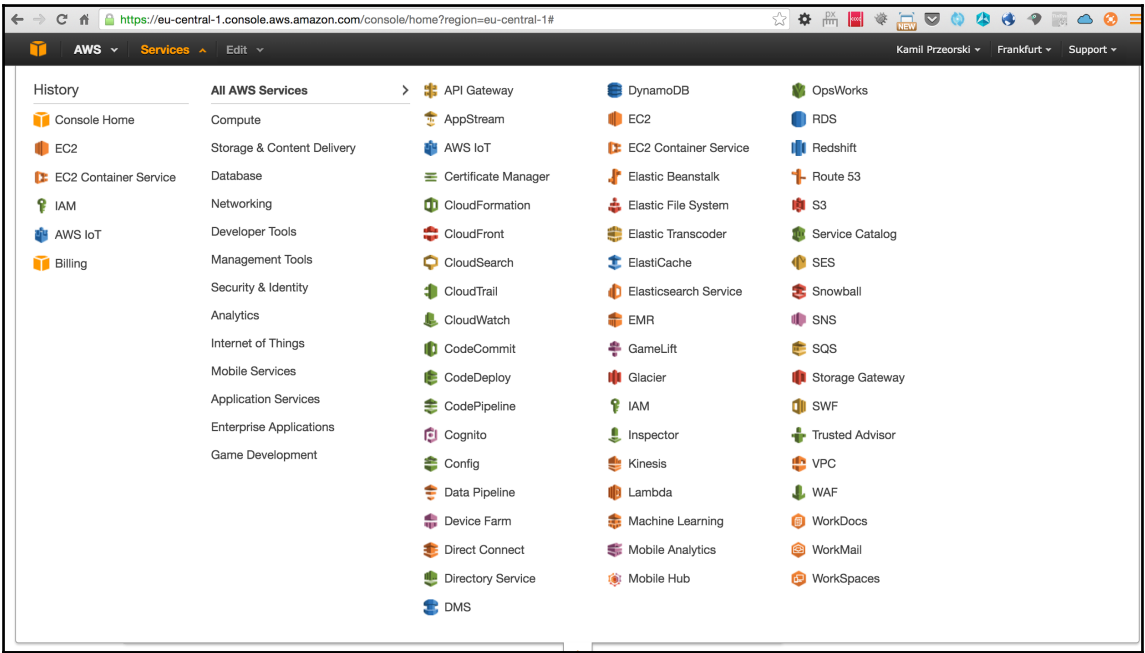
For this step, you have two options:

- Having your own domain already registered
- Registering a new domain via Route 53

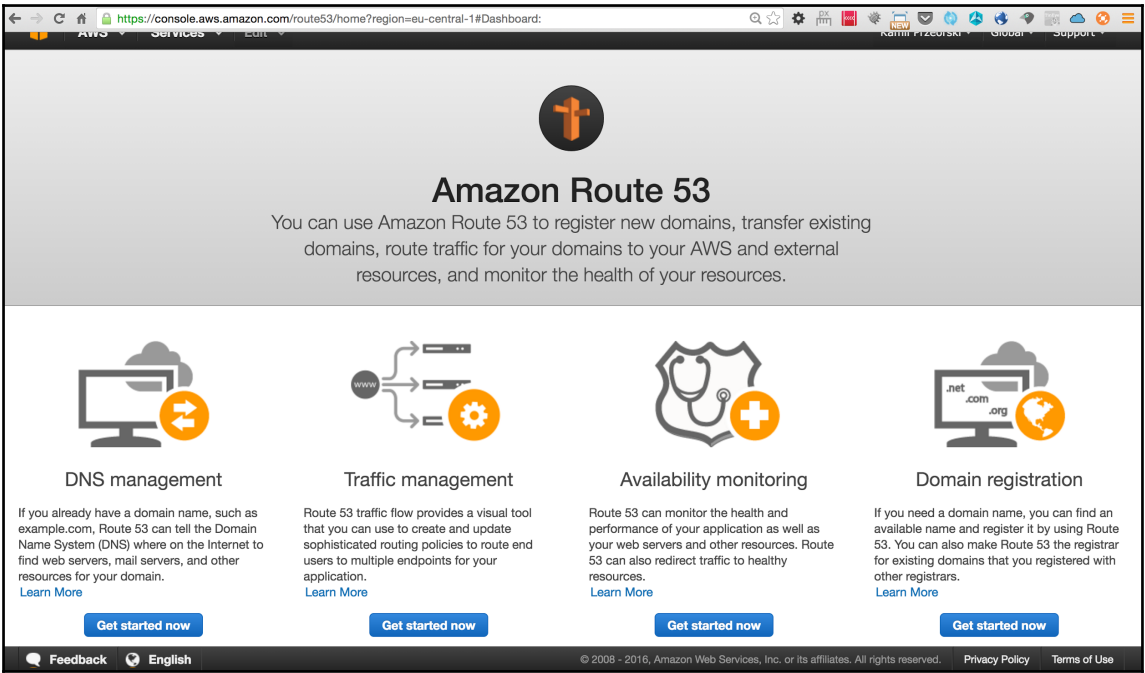
In the following procedure, we will use the first option, so we assume that we have already registered the `reactjs.space` domain (of course, you need to have your own domain in order to successfully follow these steps).

We will route end users to the publishing app by translating the name `http://reactjs.space` into the address of our ELB (`EC2Contai-EcsElast-1E4Y3WOGMV6S4-39378274.eu-central-1.elb.amazonaws.com`) so that users will be able to visit our application in a more user-friendly manner by typing `reactjs.space` into the browser's address bar.

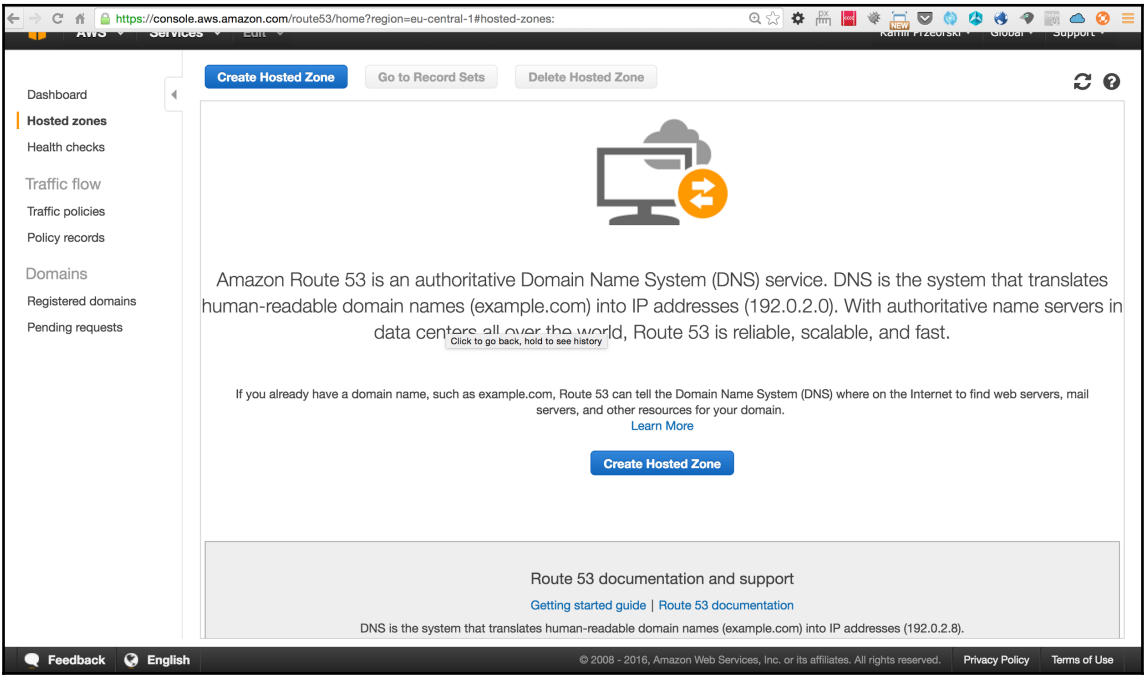
Choose Route 53 from the AWS services list:



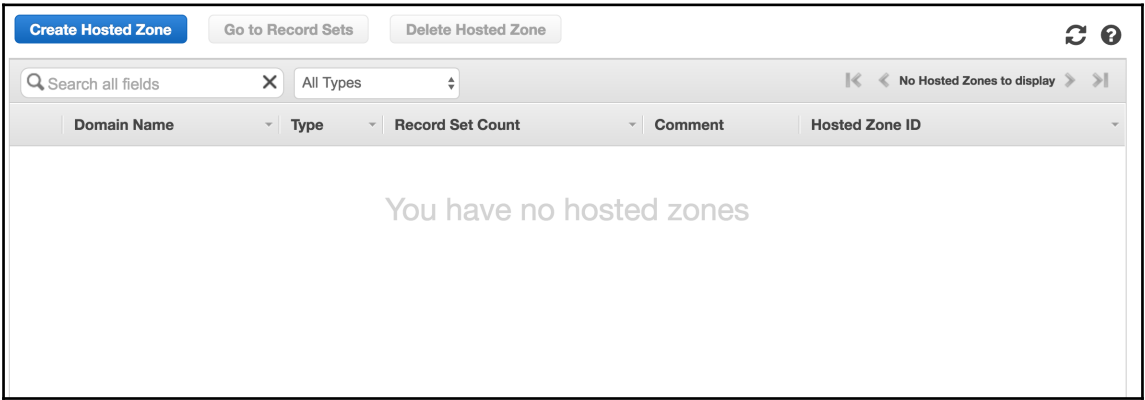
You will be able to see a main page like the following:



The next step is to create a hosted zone on Route 53, so click on the blue button called **Create Hosted Zone**:



After this, you won't see any hosted zones, so click again on the blue button:



The form will have a **Domain Name** field, where you put your domain name (in our case, it's `reactjs.space`):

Create Hosted Zone

A hosted zone is a container that holds information about how you want to route traffic for a domain, such as `example.com`, and its subdomains.

Domain Name:

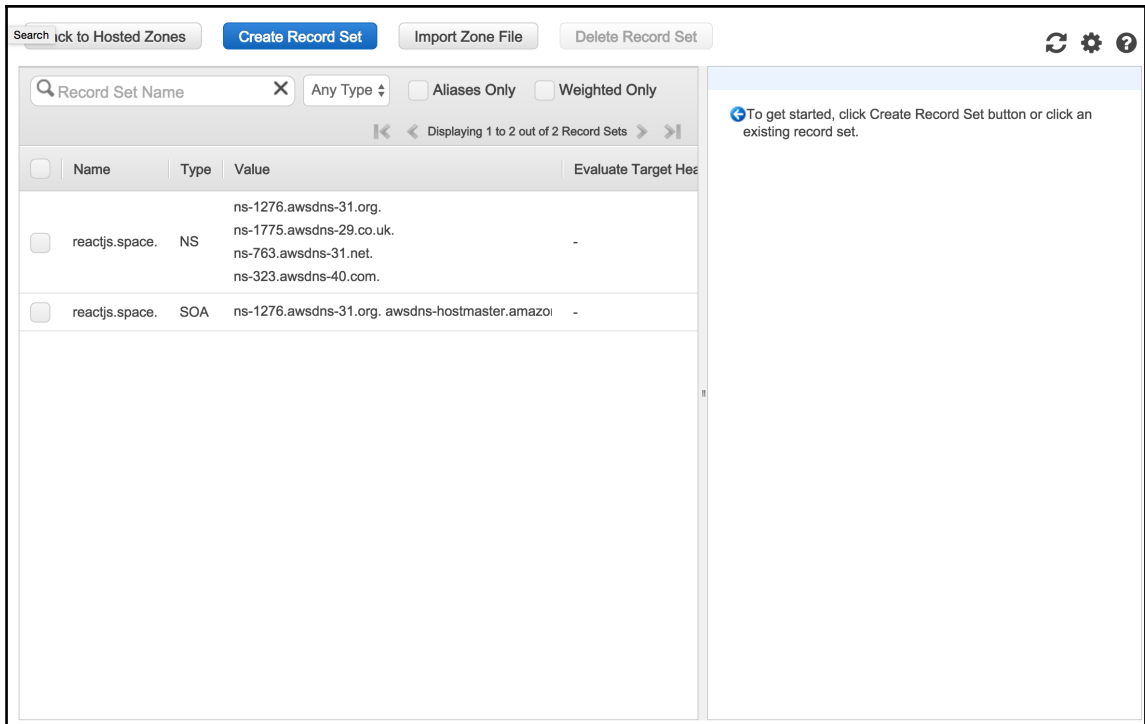
Comment:

Type:

A public hosted zone determines how traffic is routed on the Internet.

Create

Success! Now you will be able to see your DNS names:



The next step is to park the DNSes on your domain's provider. The last step is to change DNS settings at your domain registrar; in my case, they're as follows (yours will be different):

```
ns-1276.awsdns-31.org.  
ns-1775.awsdns-29.co.uk.  
ns-763.awsdns-31.net.  
ns-323.awsdns-40.com.
```

Notice the . (dots) at the end; you can get rid of them so the final DNSes that we have to change are as follows:

```
ns-1276.awsdns-31.org  
ns-1775.awsdns-29.co.uk  
ns-763.awsdns-31.net  
ns-323.awsdns-40.com
```

After all these steps, you can visit the `http://reactjs.space` website (the DNS change may take up to 48 hours).

The last thing is to create an alias of the `reactjs.space` domain that points to our Elastic Load Balancer. Click the following button:



Then, you'll have the following view:

Create Record Set

Name:

Type: A – IPv4 address

Alias: ☒ Yes ☐ No

Alias Target:

You can also type the domain name for the resource. Examples:

- CloudFront distribution domain name: `d111111abcdef8.cloudfront.net`
- Elastic Beanstalk environment CNAME: `example.elasticbeanstalk.com`
- ELB load balancer DNS name: `example-1.us-east-1.elb.amazonaws.com`
- S3 website endpoint: `example.s3-website-us-east-1.amazonaws.com`
- Resource record set in this hosted zone: `www.example.com`

[Learn More](#)

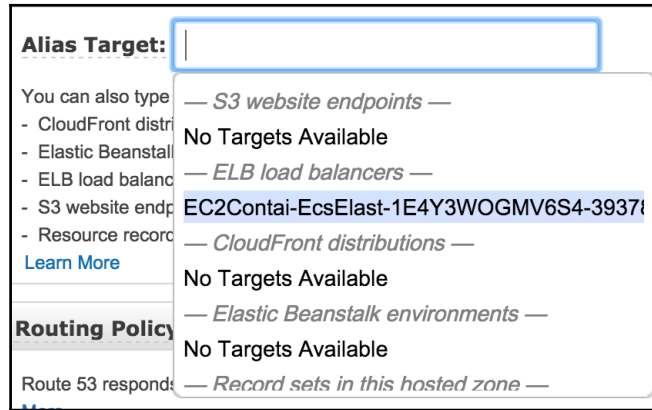
Routing Policy: Simple

Route 53 responds to queries based only on the values in this record. [Learn More](#)

Evaluate Target Health: ☐ Yes ☒ No

Create

Choose **Yes** from the alias's radio button and then select the ELB from the list, as shown in the following example:



Currently, everything will be working after the DNS changes are finished (which may take up to 48 hours). To improve the experience with our application, let's also make an alias from `www.reactjs.space` to `reactjs.space`, so if anyone types `www.` before the domain name, it will work as intended.

Click again on the button called **Create Record Set**, choose an alias, and type `www.`, after which you will be able to choose the `www.reactjs.space` domain. Do so and hit the **Create** button:

Create Record Set

Name:

Type:

Alias: ☒ Yes ☐ No

Alias Target:

You can also type

— Record sets in this hosted zone —

www.reactjs.space.

- CloudFront distr

- Elastic Beanstalk

- ELB load balancer DNS name: example-1.us-east-1.elb.amazonaws.com

- S3 website endpoint: example.s3-website-us-east-1.amazonaws.com

- Resource record set in this hosted zone: www.example.com

[Learn More](#)

Routing Policy:

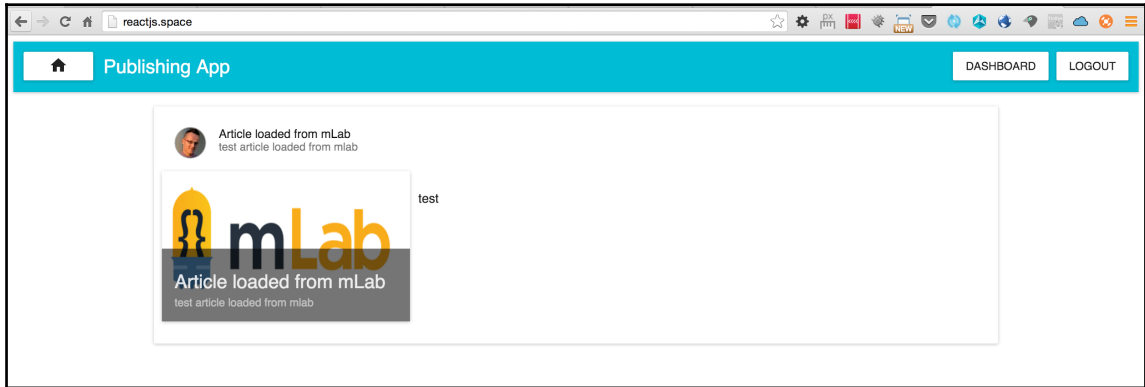
Route 53 responds to queries based only on the values in this record. [Learn More](#)

Evaluate Target Health: ☐ Yes ☒ No

Create

Summary

We are done with all the AWS/Docker setup. After a successful DNS change, you will be able to find our application under the `http://reactjs.space` address:



The next chapter will talk about the basics of continuous integration and also help you wrap up the remaining things in the app before it is 100% production ready (minification is missing so far).

Let's continue in the next chapter with a more detailed description of the remaining topics that are going to be covered in the book.

9

Continuous Integration with Unit and Behavioral Tests

We made it; congratulations! We have created a full-stack app that runs under a certain domain name (in this book its *reactjs.space*). The missing parts in the whole setup are the deployments processes. Deployments should be zero downtime. We need to have a redundant server of our application.

We are also missing some steps in our app to make it professionally work as minification, unit, and behavioral tests.

In this chapter, we will introduce you to some additional concepts that are required in order to master the full-stack development. The remaining missing pieces are left for you as homework.

When to write unit and behavioral tests

Generally, there are some advices about when to write unit and/or behavioral tests.

We in ReactPoland often have clients who run startups. As a general governance for them, we suggest the following:

- If your startup is looking for traction and you need your product in order to make it, then don't worry about tests
- After you have created your **Minimum Viable Product (MVP)**, you *must* have those tests when expanding your application
- If you are a well-established company, which is building an app for your clients and you know their needs very well, then you must have tests

The preceding first two points are related to startups and young companies. The third point is related mostly to well-established companies.

Depending on where you and your product are, then you need to decide on your own, if it is worth to write the tests.

React conventions

There is a project that shows fully how the full-stack development setup should be looking like at <http://ReactJS.co>.

Visit this website and learn how to integrate your app with unit and behavioral tests and learn up-to-date best conventions about how to make React Redux applications.

Karma for testing

We won't guide you in setting up the tests in this chapter because it's not in the scope of this book. The aim of this chapter is intended to present you with online resources that will help you to understand the bigger picture.

Karma is the one of most popular tools for unit and behavioral tests. The main goal is to bring a productive testing environment while working on any application.

There are many features that you are provided with by using this test runner. There is a nice video that explains the big picture about Karma at <https://karma-runner.github.io>.

Some of the main features are as follows:

- **Tests on real devices:** You can use real browsers and real devices such as phones, tablets, or PhantomJS to run the tests (PhantomJS is a headless WebKit scriptable with a JavaScript API; it has fast and native support for various web standards: DOM handling, CSS selector, JSON, Canvas, and SVG.) There are different environments, but one tool that runs on them all.
- **Remote control:** You can run the tests remotely, for example, on each save from your IDE, so that you don't have to do it manually.
- **Testing framework agnostic:** You can write your tests in Jasmine, Mocha, QUnit, and other frameworks. It's totally up to you.
- **Continuous integration:** Karma works great with CI tools such as Jenkins, Travis, or CircleCI.

How to write unit and behavioral tests

Let's provide an example of how to set up properly a project in order to have an ability to write tests.

Visit the GitHub repo of the very popular Redux starter kit at

<https://github.com/davezuko/react-redux-starter-kit>.

Then visit the `package.json` file of this repository. We can find out there what the possible commands/scripts are:

```
"scripts": {
  "clean": "rimraf dist",
  "compile": "better-npm-run compile",
  "lint": "eslint src tests server",
  "lint:fix": "npm run lint -- --fix",
  "start": "better-npm-run start",
  "dev": "better-npm-run dev",
  "dev:no-debug": "npm run dev -- --no_debug",
  "test": "better-npm-run test",
  "test:dev": "npm run test -- --watch",
  "deploy": "better-npm-run deploy",
  "deploy:dev": "better-npm-run deploy:dev",
  "deploy:prod": "better-npm-run deploy:prod",
  "codecov": "cat coverage/*/lcov.info | codecov"
},
```

As you can find, after the NPM test that it runs the following command:

```
"test": {
  "command": "babel-node ./node_modules/karma/bin/
    karma start build/karma.conf",
  "env": {
    "NODE_ENV": "test",
    "DEBUG": "app:*"
  }
}
```

You can find the configuration file of Karma located at `build/karma.conf` from

<https://github.com/davezuko/react-redux-starter-kit/blob/master/build/karma.conf.js>.

And the content (July 2016) is as follows:

```
import { argv } from 'yargs'
import config from '../config'
import webpackConfig from './webpack.config'
import _debug from 'debug'

const debug = _debug('app:karma')
debug('Create configuration.')

const karmaConfig = {
  basePath: '../', // project root in relation to bin/karma.js
  files: [
    {
      pattern: &grave;./${config.dir_test}/test-bundler.js&grave;;,
      watched: false,
      served: true,
      included: true
    }
  ],
  singleRun: !argv.watch,
  frameworks: ['mocha'],
  reporters: ['mocha'],
  preprocessors: {
    [&grave;${config.dir_test}/test-bundler.js&grave;]: ['webpack']
  },
  browsers: ['PhantomJS'],
  webpack: {
    devtool: 'cheap-module-source-map',
    resolve: {
      ...webpackConfig.resolve,
      alias: {
        ...webpackConfig.resolve.alias,
        sinon: 'sinon/pkg/sinon.js'
      }
    },
  },
  plugins: webpackConfig.plugins,
  module: {
    noParse: [
      //sinon.js/
    ],
    loaders: webpackConfig.module.loaders.concat([
      {
        test: /sinon(\/)pkg(\/)sinon.js/,
        loader: 'imports?define=>false,require=>false'
      }
    ])
  },
}
```

```
// Enzyme fix, see:
// https://github.com/airbnb/enzyme/issues/47
externals: {
  ...webpackConfig.externals,
  'react/addons': true,
  'react/lib/ExecutionEnvironment': true,
  'react/lib/ReactContext': 'window'
},
sassLoader: webpackConfig.sassLoader
},
webpackMiddleware: {
  noInfo: true
},
coverageReporter: {
  reporters: config.coverage_reporters
}
}

if (config.globals.__COVERAGE__) {
  karmaConfig.reporters.push('coverage')
  karmaConfig.webpack.module.preLoaders = [{
    test: /\. (js|jsx)$/,
    include: new RegExp(config.dir_client),
    loader: 'isparta',
    exclude: /node_modules/
  }]
}

// cannot use &grave;export default&grave;; because of Karma.
module.exports = (cfg) => cfg.set(karmaConfig)
```

As you can see in `karma.conf.js` they are using Mocha (check the line with `"frameworks: ['mocha']"`). The rest of the options used in the config files are described in the documentation that is available at

<http://karma-runner.github.io/1.0/config/configuration-file.html>. If you are interested in learning the Karma configuration, then `karma.conf.js` should be your starting file.

What is Mocha and why do you need it?

In the Karma config file, we have found that it uses Mocha as the JS testing framework (<https://mochajs.org/>). Let's analyze the codebase.

We can find `dir_test : 'tests'` in the `config/index.js` file, so based on that variable, Karma's config knows that the Mocha's tests are located in the `tests/test-bundler.js` file.

Let's see what is in the `tests` directory at <https://github.com/davezuko/react-redux-starter-kit/tree/master/tests>. As you can see in the `test-bundler.js` file, there are plenty of dependencies:

```
// -----  
// Test Environment Setup  
// -----  
import 'babel-polyfill'  
import sinon from 'sinon'  
import chai from 'chai'  
import sinonChai from 'sinon-chai'  
import chaiAsPromised from 'chai-as-promised'  
import chaiEnzyme from 'chai-enzyme'  
  
chai.use(sinonChai)  
chai.use(chaiAsPromised)  
chai.use(chaiEnzyme())  
  
global.chai = chai  
global.sinon = sinon  
global.expect = chai.expect  
global.should = chai.should()
```

Let's roughly describe what is used there:

- Babel-polyfill emulates a full ES6 environment
- Sinon is a standalone and test framework agnostic JavaScript test for spies, stubs, and mocks

Spies are useful if in a tested piece of code, you call for an other external's services. You can check if it was called, what parameters it had, if it returned something, or even how many times it was called!

The stubs concept is very similar to the spies concept. The biggest difference is that stubs replace the target function. They also replace the called code with custom behavior (replacing it) such as throwing exceptions or returning a value. They are also able to call a callback function that has been provided as a parameter. Stubs code returns a specified result.

Mocks are kind of *smarter stubs*. Mocks are used for asserting data and should never return data, when a stub is used simply for returning data and should never assert. Mocks can file your tests (when asserting), while stubs can't.

Chai is the BDD/TDD assertion framework for Node.js and the browser. In the previous example, it has been paired with the Mocha testing framework.

Testing CoreLayout step-by-step

Let's analyze the `CoreLayout.spec.js` tests. This component has a role similar to the CoreLayout in the publishing app, and so it's a good way to describe how you can start writing tests for your application.

The CoreLayout tests file location (July 2016) is available at

<https://github.com/davezuko/react-redux-starter-kit/blob/master/tests/layouts/CoreLayout.spec.js>.

The content is as follows:

```
import React from 'react'
import TestUtils from 'react-addons-test-utils'
import CoreLayout from 'layouts/CoreLayout/CoreLayout'

function shallowRender (component) {
  const renderer = TestUtils.createRenderer()

  renderer.render(component)
  return renderer.getRenderOutput()
}

function shallowRenderWithProps (props = {}) {
  return shallowRender(<CoreLayout {...props} />)
}

describe('(Layout) Core', function () {
  let _component
  let _props
  let _child
```

```
beforeEach(function () {
  _child = <h1 className='child'>Child</h1>
  _props = {
    children: _child
  }

  _component = shallowRenderWithProps(_props)
})

it('Should render as a <div>.', function () {
  expect(_component.type).to.equal('div')
})
})
```

The `react-addons-test-utils` library makes it easy to test React components with Mocha. The method that we used in the preceding example is **shallow rendering**, which is available at

<https://facebook.github.io/react/docs/test-utils.html#shallow-rendering>.

This feature helps us test the `render` function and is the result of rendering a one level deep in our components. Then we can assert facts about what its `render` method returns, as shown in the following:

```
function shallowRender (component) {
  const renderer = TestUtils.createRenderer()

  renderer.render(component)
  return renderer.getRenderOutput()
}

function shallowRenderWithProps (props = {}) {
  return shallowRender(<CoreLayout {...props} />)
}
```

First, we provide a component in the `shallowRender` method (in this example, it will be `CoreLayout`). Later, we use `method.render` and then we return the output with the use of `renderer.getRenderOutput`.

In our case, that function is called here (note that the semicolons are missing in the following example, because the starter that we are describing has different linting options than ours):

```
describe('(Layout) Core', function () {
  let _component
  let _props
  let _child
```

```
beforeEach(function () {
  _child = <h1 className='child'>Child</h1>
  _props = {
    children: _child
  }

  _component = shallowRenderWithProps(_props)
})

it('Should render as a <div>.', function () {
  expect(_component.type).toEqual('div')
})
})
```

You can find that the `_component` variable contains the result of the `renderer.getRenderOutput`. This value is asserted as follows:

```
expect(_component.type).toEqual('div')
```

In that test, we test our code if it returns `div`. But if you visit the documentation, then you can find the code example as follows:

```
<div>
  <span className="heading">Title</span>
  <Subcomponent foo="bar" />
</div>
```

You can also find the assertion example as follows:

```
var renderer = ReactTestUtils.createRenderer();
result = renderer.getRenderOutput();
expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

As you can see in the preceding two examples, you can expect a type as `div` or you can expect more specific information about the `CoreLayout` return (depending on your needs).

The first test asserts the type of a component (if it is `div`), and the second example test asserts if a `CoreLayout` returns correct components that are as follows:

```
[
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]
```

The first one is a unit test because this isn't testing exactly if users see a correct thing. The second one is a behavioral test.

Generally, Packt has many books on **Behavior-Driven Development (BDD)** and **Test-Driven Development (TDD)**.

Continuous integration with Travis

In the given example, you can find a `.yaml` file at

<https://github.com/davezuko/react-redux-starter-kit/blob/master/.travis.yml>.

This is a configuration file for Travis. What is this? It's a hosted CI service used to build and test software. Generally, it's a tool that is free for open source projects to use. If you want a hosted Travis CI for private projects, then their fees apply.

Configuration for Travis is made by adding the `.travis.yml` file, as mentioned earlier. The YAML form is a text file that is placed to the root directory of your project. This file's content describes all the steps that have to be done to test, install, and build a project.

The Travis CI goal is to make every commit to your GitHub account and to run the tests, and when the tests are passing, you can deploy the app to a staging server on an Amazon AWS. The continuous integration is not in the scope of this book, so if you are interested in adding this step to the whole publishing app project, there are books related to this as well.

Summary

Our publishing app is working. As with any digital project, there are still plenty of stuff that we can improve in order to have a better end product. For example, the following homework is for you:

- Add a minifaction on the frontend side so that it will be lighter when loading over the Internet.
- As mentioned earlier, you need to start using Karma and Mocha for unit and behavioral tests. An example setup was described in detail in this chapter.
- You need to choose a CI tool such as Travis, create your YML file, and prepare the environment on AWS.

That is all you can additionally do besides all that has been covered in the 350+ pages of this book, where you built a full-stack React + Redux + Falcor + Node + Express + Mongo application. I hope to keep in touch with you; follow me on Twitter/GitHub in order to keep in touch or send me an e-mail if you have any additional questions.

Good luck in getting your hands dirty with the next commercial full-stack applications and see you again.

Index

A

- action
 - creating 162
- AddArticleView component
 - about 131
 - tweaks 157, 158
- Amazon Machine Image (AMI) 309
- app frontend
 - about 118
 - article list, improving 121
 - handleServerSideRender, improving 120
 - routes, changing in Falcor 121
 - website header, improving 121
- application
 - improving 181
- article's feature implementation
 - deleting 169, 170, 174
- ArticleCard component
 - about 123, 125
 - dashboard 128, 129
 - Dashboard View, modifying 135, 137
 - formatting features, adding to WYSIWYG 145, 146, 150
 - new article, pushing into article reducer 150
 - views/articles/AddArticleView component, improving 143
 - WYSIWYG, using 137, 138
- articles
 - editing 160
 - importing, to MongoDB 13
- auth required routes 181
- AWS account
 - reference 226
- AWS CloudWatch 307
- AWS Route 53
 - setting up 337, 338, 339, 340, 341, 343, 344,

- 345
- AWS S3
 - ability, adding to add/edit title and subtitle of article 263
 - ability, to edit article title and subtitle 268
 - about 225
 - AddArticleView, improvements 247, 250, 264, 265
 - article's cover photo, editing 258, 261, 262, 263
 - ArticleCard component, improving 254
 - ArticleCard, improvements 271
 - Dashboard, improvements 275
 - DashboardView component, improving 257
 - environmental variables, in Node.js 238
 - IAM, creating 227, 228, 229
 - image upload feature, coding in AddArticleView 237
 - ImgUploader component, creating on frontend 243
 - ImgUploader component, wrapping up 244
 - keys, generating 225, 226
 - Mongoose article schema, improving 240
 - permissions, setting up for user 230, 231, 232, 233, 234, 235
 - PublishingApp, improvements 272
 - routes, adding for S3's upload 240, 242
 - tweaks, for ArticleCard 252
 - tweaks, for DashboardView 252
 - tweaks, for PublishingApp 252

B

- B2C 102
- backend
 - wrapping up 119
- basic concepts, Falcor
 - client-side Falcor 35
 - client-side Falcor + Redux 42

- Falcor's model, moving to backend 42
- Falcor's router, configuring 46
- full-stack's Falcor, running 49
- MongoDB/Mongoose calls, adding based on
 - Falcor's router 51
- no more coupling, on client and server side 35
- second route, for returning two articles from
 - backend 47
- tight coupling and latency, versus one model
 - everywhere 34
- basic concepts, Redux
 - immutability 22
 - impure functions 23
 - pure functions 23
 - reducer function 23
 - reducer, and webpack config 24
 - single immutable state tree 21
- behavioral tests
 - writing 347, 349
- bucket 225

C

- Chai 353
- client-side Falcor 35
- cluster 323
- collections
 - viewing, in Robomongo GUI 10
- container
 - debugging 302, 303
- continuous integration
 - with Travis 356
- CoreLayout
 - improvements 154, 155
 - testing 353
- Cross-Origin Resource Sharing (CORS) 17
- CRUD routes, securing
 - \$error sentinel basics 209, 210
 - \$error, cleaning up after successful test 218
 - \$error-related code, testing 215
 - about 209
 - backend implementation, of \$error sentinel 215
 - DRY management, on client side 210, 211
 - FalcorModel.js, on frontend 213

D

- dashboard link
 - adding, to article's edition 162
- DashboardView
 - components, creating 87
 - improving 156
 - login's mechanism, finishing 89
 - security 91
 - successful logins, handling in LoginView's
 - component 90
- database
 - user/password, creating 284, 285
- DB configs
 - app, double-checking before JWT
 - implementation 67
 - JWT, implementing in routesSession.js file,
 - implementing 68
 - Mongoose users' model, creating 68
 - route.js file, improving 66
 - separating 65
 - successful login, on falcor-route 71
- DefaultInput component
 - working on 82
- dependencies
 - installing 27
- Docker commands 306
- Docker container
 - pushing, to remote repository 303, 304, 305
- Docker Hub
 - hello world example 291
 - reference 292
- Docker machine
 - reference 290
- Docker on AWS EC2
 - about 307
 - EC2 instance connection, via SSH 318, 320, 321, 322
 - EC2 instance, launching 308, 309, 310, 311, 312, 313
 - manual approach 307
 - SSH access, via PuTTY 316, 317, 318
- docker run
 - reference 302
- Docker Toolbox

- Docker, installing with 289, 291
- reference 289
- Docker
 - installing, with Docker Toolbox 289, 291
 - reference, for official installation page 289
- Dockerfile example
 - about 294
 - modifications, to codebase 294, 295
 - publishing app container, building 299
 - publishing app container, running locally 301, 302
 - working on publishing app Docker image 297, 298, 299
- domain name system (DNS) 323
- draft-js
 - reference 141
- DRY (don't repeat yourself) 209

E

- ECS
 - about 307
 - cluster, configuring 329, 330
 - launch status 332, 333
 - load balancer address, finding 333, 334, 335, 336, 337
 - reviewing step 331
 - service, configuring 327, 328
 - task definition, creating 324, 325, 326
 - working with 323
- edit mode in
 - src/components/articles/WYSIWYGeditor.js 163
- EditArticleView
 - improvements 165, 167
 - render improvements 167
- editor's registration
 - working on 91
- Elastic Load Balancer (ELB) 328
- Elastic Load Balancing (ELB) 327
- environment
 - preparing 8
- error selector
 - reference 215
- example collection
 - importing, into database 11

- Express.js
 - about 7
 - server setup, performing with 13

F

- Falcor code, improving on frontend
 - about 182, 183, 184
 - routes.js, improving 184, 185
 - server.js, improving 184, 185
- Falcor's router
 - configuring 46
- Falcor, and Relay/GraphQL
 - similarities 179
 - technical differences 179, 180
- falcor-router's login (backend)
 - creating 63
 - working 64
- Falcor
 - basic concepts 33
 - frontend side 72
 - problem solving 177
 - sentinel implementation 188
 - versus Relay/GraphQL 178
- flip-flop's demo
 - reference 281
- frontend implementation
 - about 95
 - RegisterView file, creating 96
- frontend side, Falcor
 - about 72
 - changes, in src/app.js 79
 - CoreLayout component 73
 - LoginView component 74
 - remaining configuration, for configureStore and rootReducer 76
 - root's container, for app 75
 - screenshots, of running app 80
- full-stack app
 - working 55

G

- GraphQL
 - versus Falcor 178

H

high availability 280

I

IAM page

reference 228

impure functions 23

issuu

reference 6

J

JS object

versus Maps 155

JSON 8

JSON envelopes

in Falcor 182

reference 182

JSON Graph

in Falcor 181

reference 182

JSON Web Token (JWT)

about 57, 58

reference 58

JWT token

audience 59

expiration date 59

issue date 59

issuer 59

structure 59

subject 59

K

Karma

about 348

reference 348

L

Linux distributions

reference 290

login form 82

login's factor-route

working on 62

LoginForm component

working on 82

working, with LoginView 84

LoginView Component

src/views/LoginView.js, improving 85

M

Map

versus JS object 155

MapHelpers

for improving reducers 151, 152

Medium

reference 6

Minimum Viable Product (MVP) 347

mLab account

creating 281, 283

mLab node 281

mLab

features 279

free, versus paid plan 281

overview 278, 279

reference 279

Mocha

about 352

need for 352

reference 352

mocks 353

MongoDB failover 280

MongoDB's users collection

about 60, 61

initPubUsers.js file, importing into MongoDB 61

MongoDB

articles, importing to 13

installing 9

reference 9

running 10

Mongoose

adding, to project 17

N

Netflix-Falcor.js 7

Node Version Manager (NPM) 7

Node.js

about 7

reference 9, 16

server setup, performing with 13

Node

- installing 8
- npm dev script 27
- NVM
 - about 8
 - installing 8
 - reference 8

P

- package.json
 - double-check, performing with 53
- project
 - running 18
- PublishingApp
 - improving 156
- pure functions 23

R

- React + Redux application
 - wrapping up 29
- React conventions
 - about 348
 - Karma, for testing 348
- React
 - about 179
 - reference 348
- read-only backup 280
- reducer function 23
- reducers
 - article's reducer 76
 - editor's reducer 76
 - routing's reducer 76
 - session's reducer 76
- Redux starter kit
 - reference 349
- Redux
 - about 8
 - basic concepts 21
- register's falcor-route
 - adding 92
- Relay
 - versus Falcor 178
- remote repository
 - Docker container, pushing to 303, 304, 305
- replica set connections 280
- Robomongo GUI

- collections, viewing in 10
- for MongoDB 9
- Robomongo
 - about 9
 - reference 10
- routes' security
 - wrapping up 218
- routes
 - securing 219

S

- sentinel 181
- sentinel implementation, Falcor
 - \$ref sentinel 188
 - \$ref sentinel, example 189
 - about 188
 - article, deleting 205, 206
 - article, editing 205
 - articles' numberOfLikes, improving with \$ref 191
 - articles[{integers}] route, improving 197
 - frontend - edit and delete 207, 208
 - frontend changes, for adding articles 201, 203
 - JSON Graph atoms 196, 197
 - Mongoose config improvements 192
 - new route, in server/routes.js 199
 - practical use, of \$ref 192
 - route returns 204
 - server/route.js improvements 194, 196
- server setup
 - performing, with Express.js 13
 - performing, with Node.js 13
- server-side rendering
 - about 102
 - database response, mocking 103, 105
 - frontend tweaks 112, 114, 116
 - handleServerSideRender function 106, 108
 - server/server.js, double-checking 110
- server/route.js
 - double-check, performing with 53
- server
 - working on 16
- service 324
- shallow rendering
 - reference 354
- sharded clusters 279

- single immutable state tree 21
- slow queries tool 279
- src/app.js
 - working on 28
- src/layouts/PublishingApp.js
 - working on 28
- stubs 353
- stylesheet, for draft-js WYSIWYG
 - draft-js skeleton, coding 140, 141

T

- task definition 324

U

- unit tests
 - writing 347, 349

V

- Vagrant
 - reference 8
- virtual JSON
 - one model everywhere 178
- VirtualBox
 - reference 8, 290

W

- web pages
 - versus web applications 33
- what you see is what you get (WYSIWYG) 128
- WordPress
 - reference 6
- World Wide Web (WWW) 33