Beginning KeystoneJS

A practical introduction to KeystoneJS using a real-world project

Manikanta Panati

Apress®

Beginning KeystoneJS

A practical introduction to KeystoneJS using a real-world project

Manikanta Panati

Apress®

Beginning KeystoneJS: A practical introduction to KeystoneJS using a real-world project

Manikanta Panati
Apex, North Carolina, USA

ISBN-13 (pbk): 978-1-4842-2546-2 ISBN-13 (electronic): 978-1-4842-2547-9
DOI 10.1007/978-1-4842-2547-9

Library of Congress Control Number: 2016962199

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Pramila Balan
Development Editor: Matthew Moodie
Technical Reviewer: Jibin George
Coordinating Editor: Prachi Mehta
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Image: Designed by Freepik

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com/rights-permissions.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484225462. For more detailed information, please visit http://www.apress.com/source-code.

Printed on acid-free paper

This book is dedicated to my
Dad, Mom, Gundi, Gundma & My Love!

Contents at a Glance

About the Author xiii

Introduction xv

■ Chapter 1: Meet KeystoneJS 1

■ Chapter 2: Building the IncTicket Application 13

■ Chapter 3: Introducing KeystoneJS Models 43

■ Chapter 4: Model Relations 55

■ Chapter 5: Integrating Web Forms 63

■ Chapter 6: Filtering Requests with Middleware 71

■ Chapter 7: Authenticating and Managing Your Users 75

■ Chapter 8: Searching Site Data 99

■ Chapter 9: Restful API for Mobile and SPA applications 107

■ Chapter 10: Using Template Engines 123

■ Chapter 11: Deploying to Ubuntu Cloud Server 133

Index 143

Contents

About the Author xiii

Introduction xv

Chapter 1: Meet KeystoneJS 1

1.1 Why Use KeystoneJS? 1

1.2 What Is KeystoneJS Best for? 2

1.3 KeystoneJS Versions 2

1.4 Installing MongoDB and Node.js 2

1.5 How to Install MongoDB 3

1.6 How to Install Node.js 4

1.7 Testing Whether Node.js Is Installed Properly 6

1.8 Useful Development and Debugging Tools 7

1.9 Summary 12

Chapter 2: Building the IncTicket Application 13

2.1 Installing KeystoneJS 13

2.1.1 Prerequisites 13

2.2 Creating a New KeystoneJS Application 14

2.3 Configuring Your KeystoneJS Application 16

2.4 Project Structure 17

2.5 Creating Your First Model/List 19

2.6 Creating an Administration Site for Your Models 21

2.6.1 Creating an Admin User 22

2.7 The KeystoneJS Administration Site 23
2.8 Modifying the Admin Menu 24
2.9 Adding Models to the Administration Site 24
2.10 Customizing the Way Models Are Displayed 26
2.11 Dynamically Adding Columns to Admin UI 27
2.12 Finding Data Using the Admin UI 28
2.13 Creating Your First Route 29
2.14 Tickets Route 30
2.15 URLs for Models 31
2.16 Creating Your First View 32
2.17 Creating Ticket List and Detail Views 32
2.18 Creating Templates for Your Views 34
2.19 Adding Pagination 39
2.20 Summary 42

Chapter 3: Introducing KeystoneJS Models 43

3.1 Introducing the Mongoose ODM 43
3.2 Mongoose Schemas and Keystone Lists 44
3.3 Adding Fields to Your Model 44
3.4 Defining Virtual Properties 46
3.5 Finding Data 46
3.6 Using the QueryBuilder 47
3.7 Query with a Single Operation 47
3.8 Retrieving All Tickets 48
3.9 Retrieving a Ticket by Slug 48
3.10 Selecting Specific Fields 48
3.11 Counting Documents 49

3.12 Ordering Documents 49
3.13 Conditional Filtering 50
3.14 Limiting Returned Documents 50
3.15 Check for Existence of a Field 51
3.16 Inserting a New Record 51
3.17 Updating Existing Records 53
3.18 Summary 53

Chapter 4: Model Relations 55

4.1 Defining Relations 55
4.2 Modeling One-to-Many Relations 56
4.3 Retrieving a One-to-Many Relation 57
4.4 Modeling Many-to-Many Relations 58
4.5 Retrieving a Many-to-Many Relation 60
4.6 Filtering Relations 60
4.7 Summary 61

Chapter 5: Integrating Web Forms 63

5.1 Creating a New Ticket Form 63
5.2 Summary 69

Chapter 6: Filtering Requests with Middleware 71

6.1 Introducing Middleware 71
6.2 Introducing KeystoneJS's Default Middleware 72
6.3 Defining Middleware in KeystoneJS 72
6.4 Assigning Middleware to Routes 73
6.5 Summary 74

Chapter 7: Authenticating and Managing Your Users 75

7.1 Configuring Authentication Options 75

7.2 The User Model 77

7.3 Registering Users 79

7.4 User Login 84

7.5 Logging Out a User 88

7.6 Password Recovery 88

7.7 Retrieving the Authenticated User 95

7.8 Restricting Access to Authenticated Users 95

7.9 Securing Your Application 96

7.9.1 Cross-Site Request Forgery 96

7.9.2 Cross-Site Scripting 97

7.9.3 Cookies 98

7.10 Summary 98

Chapter 8: Searching Site Data 99

8.1 MongoDB Full-Text Search 99

8.2 Indexing a Single Field 99

8.3 Indexing Multiple Fields/Wild Card Indexing 101

8.4 Adding Search to KeystoneJS 103

8.5 Summary 106

Chapter 9: Restful API for Mobile and SPA applications 107

9.1 Introducing Restful APIs 107

9.2 What Are We Building? 107

9.3 Tools for Working with Restful APIs 111

9.4 JSON Formatter Chrome Extension 112

9.5 POSTMAN REST Client 113

9.6 Serve Data with GET Requests 114
9.7 Update Data with POST and PUT 115
9.8 Removing Data with DELETE 119

Chapter 10: Using Template Engines 123
10.1 Introducing the Swig Template Engine 123
10.2 Basic Syntax for Swig 124
10.3 Output Tags 125
10.4 Logic Tags 125
10.4.1 Template Partials 129
10.5 Swig Filters 131
10.6 Summary 132

Chapter 11: Deploying to Ubuntu Cloud Server 133
11.1 Introduction 133
11.2 What Is DigitalOcean? 133
11.3 Provision a Server 133
11.4 Installing Node 135
11.5 Installing MongoDB 137
11.6 Install Redis for Caching 138
11.7 Install IncTicket Application 139
11.8 Manage Application with PM2 140
11.9 Install NGINX as Reverse Proxy Server 141
11.10 Summary 142

Index 143

About the Author

Manikanta Panati has spent the last 10 years perfecting enterprise-level application development using Microsoft and Open Source technologies. Recent projects include a very popular coupon site in Asia and a Node.js-powered application that aggregates and maintains business data for over 2.2 million businesses. With a Masters in Information Systems and a Masters in Project Management, and Microsoft Certified Technical specialist (MCTS), Microsoft Certified Professional Developer (MCPD) certifications, he still learns something new every day! Born and brought up in beautiful Bangalore, India, and presently based out of equally beautiful North Carolina, he works for a multinational financial institution managing migration and development of projects.

Introduction

KeystoneJS is an open source Node.js-based CMS and web application framework created by Jed Watson in 2013. The framework makes it very easy to build database-driven websites, applications, and APIs and is built upon Express, the de facto web server for Node.js and uses Mongo DB as its storage back end. Mongo DB is a very popular and powerful document store that is capable of storing data without it being structured in a schema.

KeystoneJS philosophy

Keystone is designed to make complicated things simple, without limiting the power or flexibility of node.js or the frameworks it is built on.

What Are Web Frameworks?

A web framework aims to assist a developer in delivering web applications quickly and easily. The term "framework" is relatively loosely defined and can include anything from a collection of components to a complete abstraction of workflow in an application. A framework typically provides a certain style and/or a certain structure that assists the developer, and this structure is generally based on specific design patterns. Some of the well-known web frameworks include Ruby on Rails, Laravel, Django, and Symfony. Ruby on Rails follows the popular MVC (Model-View-Controller) design pattern whereas Django and KeystoneJS follow the MVT (Model-View-Template) design pattern. Both the MVC and MVT design patterns allow for the logical separation of code and are very similar conceptually. Web frameworks encourage loose coupling and strict separation between pieces of application.

The Model-View-Template Design Pattern

KeystoneJS is based on a design pattern called Model-View-Template. A good understanding of this concept is the basis for working with KeystoneJS. Web application architecture generally comprises three pieces that work together – data access logic, business logic, and presentation logic. A good framework will aim for the logical separation of these pieces in an application into distinct subsystems so as to allow for a high degree of reusability of components. Here is roughly how the M, V, and T break down in KeystoneJS:

M stands for "Model," which represents the data access layer. Models typically contain a definition of the data and methods to interact with data such as how to access it, how to validate it, which behaviors it has, and the relationships between the data.

V stands for "View," which represents the business logic layer. Views contain the logic that accesses the model, performs any calculations, and defers the results to the appropriate template(s). View is like a bridge between models and templates.

T stands for "Template," which represents the presentation layer. Templates handle presentation-related decisions: how something should be displayed on a Web page or other type of document.

If you are familiar with other MVC Web-development frameworks, such as Laravel, you may consider KeystoneJS views to be the "controllers" and KeystoneJS templates to be the "views." In KeystoneJS, the "view" describes the data that gets presented to the user; it is not necessarily just how the data looks, but which data is presented. In contrast, Laravel and similar frameworks suggest that the controller's job includes deciding which data gets presented to the user, whereas the view is strictly how the data looks, not which data is presented.

Both MVC and MVT are very similar and interpretation of these concepts varies slightly from framework to framework and neither one is more "correct" than the other. It is good to get a proper understanding of the underlying concepts.

Introducing the IncTicket Project

The best way to learn about a new technology is to be able to visualize the various capabilities of the technology in terms of using them in the implementation of a real-world project. Throughout this book, I will introduce KeystoneJS features and syntax in conjunction with developing IncTicket, a web-based application that allows for the creation and management of incident tickets.

IncTicket will enable users to create tickets, assign statuses, set priorities, categories, and assign tickets to users. Other users can then interact with the tickets, updating their status and more.

Errata and Suggestions

"Have no fear of perfection – you'll never reach it," said Salvador Dali. When it comes to writing about the latest technology, I could not agree more! I might have made mistakes in both code and grammar, and probably completely misconstrued a few pieces of this text. If you would like to report an error, ask a question, or offer a suggestion, please reach me on twitter `@jangreejelabi`.

CHAPTER 1

Meet KeystoneJS

This chapter will introduce KeystoneJS along with its merits. We will cover how to install MongoDB and Node.js, which are needed to create the IncTicket application, using KeystoneJS and then running it.

This chapter will cover the following points:

- Introduction to KeystoneJS
- Installing MongoDB
- Installing Node.js
- Useful development and debugging tools

1.1 Why Use KeystoneJS?

Before we begin installing and using KeystoneJS, we will first look at why we use KeystoneJS framework over other frameworks available online. Simply put, KeystoneJS provides a standardized set of components that allow for fast and easy development of web applications that can be quickly developed, maintained, and extended.

KeystoneJS has a number of key features that makes it worth using, including:

- **Modularity** – Keystone will configure express – the de facto web server for node.js – for you to connect to your MongoDB database using Mongoose, the leading object data mapping (ODM) package.
- **Auto-generated Admin UI** – Whether you use it while you're building out your application, or in production as a database content management system, Keystone's Admin UI will save you time and make managing your data easy.
- **Session Management** – Keystone comes ready out of the box with session management and authentication features, including automatic encryption for password fields.

Electronic supplementary material The online version of this chapter (doi:10.1007/978-1-4842-2547-9_1) contains supplementary material, which is available to authorized users.

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_1

- **Email Sending** – Keystone makes it easy to set up, preview, and send template-based emails for your application. It also integrates with Mandrill.

Mandrill is an email API offered by MailChimp, the email marketing company. We will use it to send emails programmatically.

- **Form Processing** – Want to validate a form, upload an image, and update your database with a single line? Keystone can do that, based on the data models you've already defined.
- **Database Fields** – IDs, Strings, Booleans, Dates, and Numbers are the building blocks of your database. Keystone builds on these with useful, real-world field types like name, email, password, address, image, and relationship fields.

1.2 What Is KeystoneJS Best for?

KeystoneJS is a generic content management framework, meaning that it can be used for developing a variety of web applications using JavaScript. Because of its modular architecture and clean separation of various functionality, it is especially suitable for developing large-scale applications such as portals, forums, content management systems (CMS), e-commerce projects, RESTful Web services, and so on.

1.3 KeystoneJS Versions

KeystoneJS currently has two major versions available: 0.3.x and 0.4. At the time of writing this book, Version 0.3.x is the current generation of the framework and is in active development mode. Version 0.4 is a work in progress, adopting the latest technologies and protocols, including Mongoose 4, elemental UI, and core changes.

1.4 Installing MongoDB and Node.js

Let's start by looking at the process of installing MongoDB on a Windows workstation. MongoDB is an open source, document-oriented database that is designed to be both scalable and easy to work with. MongoDB stores data in JSON-like documents with dynamic schema instead of storing data in tables and rows like a relational database, for example, MySQL.

Let's install MongoDB database in a stand-alone mode. This is the quickest way to start a MongoDB server for the purpose of development.

1.5 How to Install MongoDB

- Navigate to the downloads page on the MongoDB official website, http://www.mongodb.org/downloads.
- Click on the download link for the latest stable release Zip Archive under Windows 32-bit or 64-bit depending on your machine architecture.

  Find the architecture of your machine by typing in the following command into the command prompt:

```
1   wmic os get osarchitecture
```

The output will be similar to:

```
1   OSArchiecture
2   64-bit
```

- Once the download completes, move the Zip archive to the C:\ drive and extract it.
- Rename the extracted folder (mongodb-win32-xxx-a.b.c where a.b.c is the version number) to mongodb.
- Create the default database path (c:\data\db). This is the location where the database files used by mongodb will reside.

```
1   c:\>mkdir data\db
```

- To start the mongodb database, Open a CMD prompt window, and enter the following commands (see Figure 1-1):

```
1   c:\> cd mongodb\bin
2   c:\mongodb\bin>mongod
```

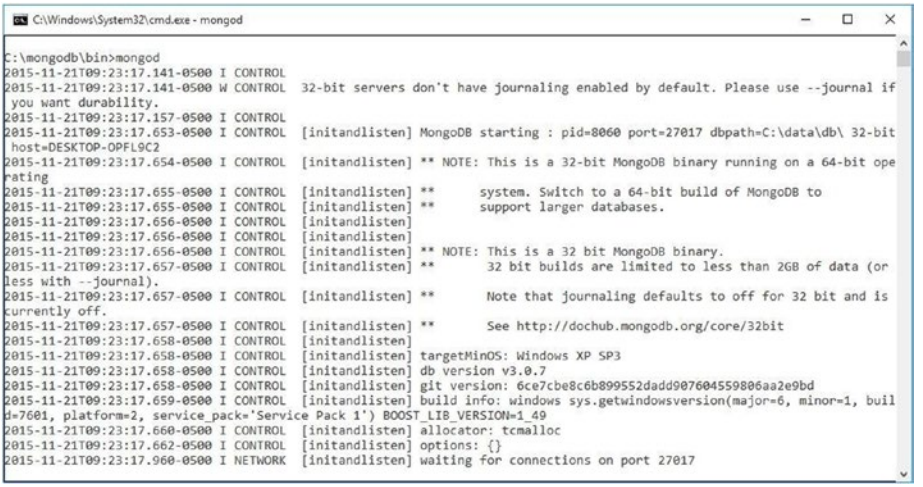


Figure 1-1. *Start MongoDB*

If you find the console log indicating **[initandlisten] waiting for connections on port 27017**, then the MongoDB server has started up correctly and is ready to accept connections from client.

1.6 How to Install Node.js

Next, we will look at the process of installing Node.js on a Windows workstation. Node.js is an open source, cross-platform runtime environment for developing web applications. Node.js applications are written in JavaScript and can be run within the Node.js runtime on OS X, Microsoft Windows, Linux, and a variety of other operating systems.

- Navigate to the downloads page on the Node.js official website, https://nodejs.org/en/download/.
- Click on the download link for the latest stable release .MSI under Windows 32-bit or 64-bit depending on your machine architecture.

Find the architecture of your machine by typing the following command into the command prompt:

```
1   wmic os get osarchitecture
```

The output will be similar to:

```
1   OSArchiecture
2   64-bit
```

- Once the download is complete, double-click on the .msi file, which will launch the Node installer.
- Proceed through each step of the installation wizard. See Figure 1-2.

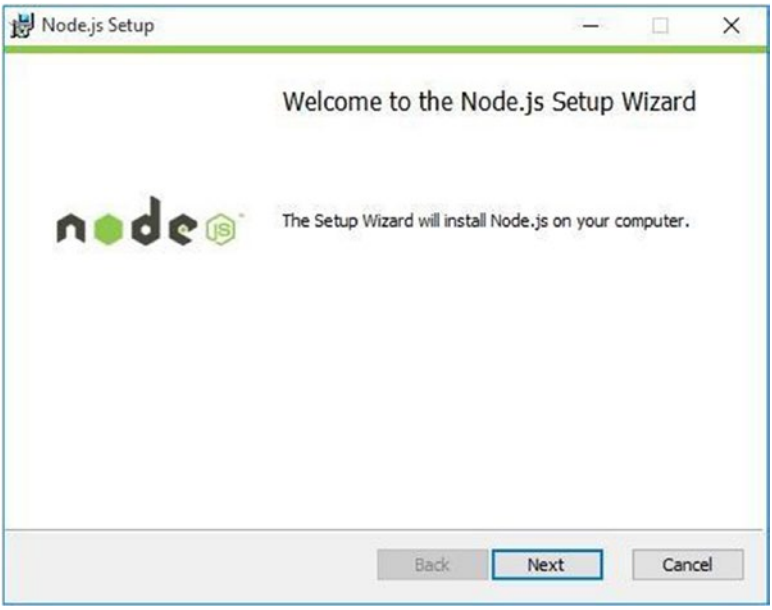


Figure 1-2. *Node.js Installer*

- At the custom setup screen during the installation, make sure that the wizard installs NPM (Node Package Manager) and configures the PATH environment variable along with installing the Node.js runtime. This should be enabled by default for all installs. See Figure 1-3.

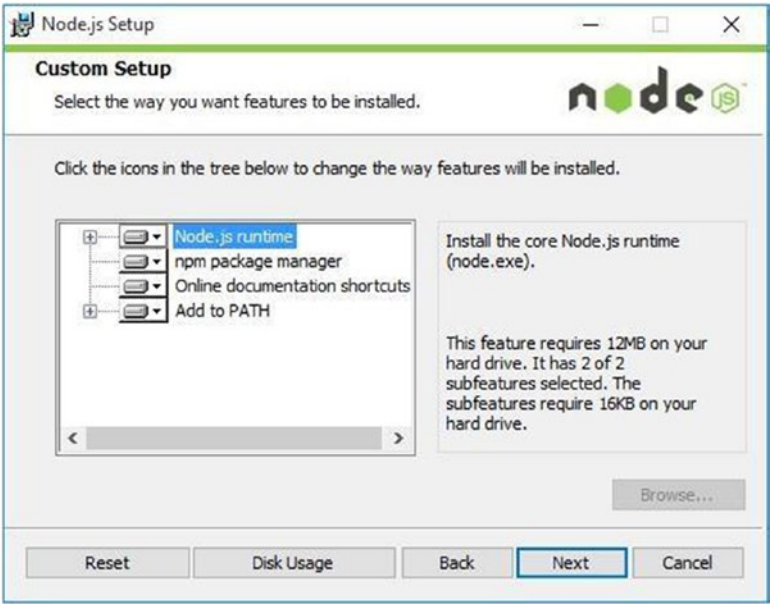


Figure 1-3. *Node.js Installer*

Once these steps have been completed, both Node and npm should be installed on your system.

1.7 Testing Whether Node.js Is Installed Properly

After going through the Node.js installation wizard, let's run a quick test to ensure everything is working properly.

Run the following commands on a new command prompt window. You might need to open a new instance of command prompt for the PATH variable changes to take effect.

```
1   c:\> node --version
2       v4.2.2
3
4   c:\> npm --version
5       2.14.7
```

If the Node installation was successful, you will see the version number that was installed as an output on the screen as a response to running the above commands.

1.8 Useful Development and Debugging Tools

I would like to introduce a couple of useful tools that make it really easy for us to develop Node.js- and MongoDB-based web applications. The first is Visual Studio Code, a code editor that offers excellent Node.js development and debugging support. It is free and available on multiple platforms – Linux, Mac OSX, and Windows. Visual Studio Code can be used for building and debugging modern web and cloud applications and includes great built-in support for C#, and Node.js development with TypeScript and JavaScript. It includes tooling for web technologies such as HTML, CSS, Less, Sass, and JSON. Code also integrates with package managers and repositories, and builds and other common tasks to make everyday workflows faster. Download Visual Studio Code from `https://code.visualstudio.com/`[1]

We can open up any folder on our filesystem using Visual Studio Code and get to editing files directly. Let us explore the GUI to get a better understanding of the various features. See Figure 1-4.

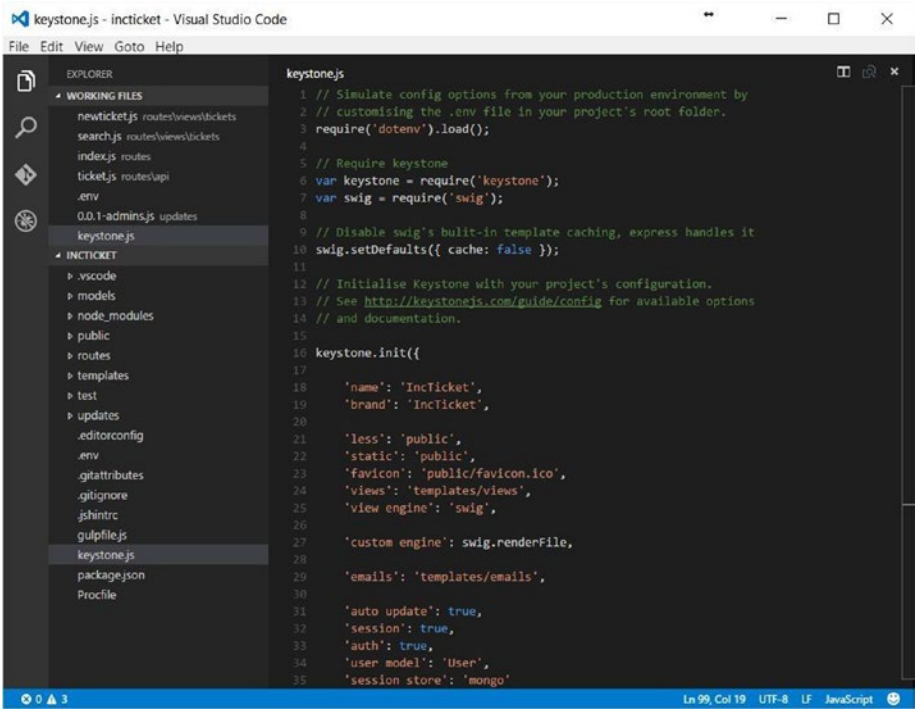


Figure 1-4. *Visual Studio Code*

[1] `https://code.visualstudio.com/`

The left pane has two sections; the first one has icons for the file explorer, file search, git integration and for debugging. The screenshot shows the explorer pane open. Visual Studio Code lists the files that are currently being worked upon in the working files section. Below the working files section is the list of files in the current directory that can be opened up for editing. The main area of the editor shows the file being edited and allows for multiple files to be opened at the same time. The editor also has a split view allowing us to look at two files side by side.

Debugging in Visual Studio Code is very easy. To start off, we need to define a launch configuration that can tell the editor about the starting point to our app and other configuration data. Below is the configuration setting we can use to debug KeystoneJS:

```
{
    "version": "0.2.0",
    "configurations": [
            {
                    "name": "Launch",
                    "type": "node",
                    "request": "launch",
                    "program": "${workspaceRoot}/keystoneJS",
                    "stopOnEntry": false,
                    "args": [],
                    "cwd": "${workspaceRoot}",
                    "runtimeExecutable": null,
                    "runtimeArgs": [
                            "--nolazy"
                    ],
                    "env": {
                            "NODE_ENV": "development"
                    },
                    "externalConsole": false,
                    "sourceMaps": false,
                    "outDir": null
            },
            {
                    "name": "Attach",
                    "type": "node",
                    "request": "attach",
                    "port": 5858,
                    "sourceMaps": false,
                    "outDir": null,
                    "localRoot": "${workspaceRoot}",
                    "remoteRoot": null
            }
    ]
}
```

Set the configuration information by clicking on the debug icon on the left pane, and then click the gear icon on the top bar. Once the configuration information is set, hit F5 to launch the application in debug mode. See Figure 1-5.

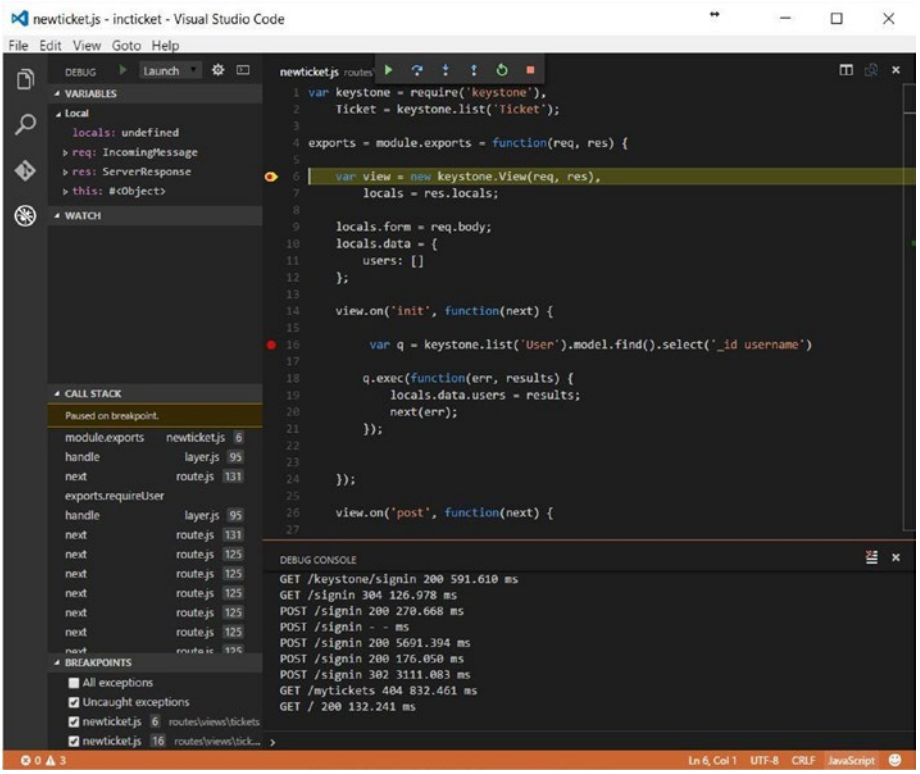


Figure 1-5. *Debugging with Visual Studio Code*

As shown above, we can set breakpoints in code, add variables to watch, and inspect the call stack. At the bottom of the editor is the node console where we can see any console interactions like console.log in our application. The node console also allows us to inspect variables inline.

Get more information on in-depth features of Visual Studio Code at `https://code.visualstudio.com/docs`[2]

The next tool is Robomongo. Robomongo is a desktop application that allows us to manage MongoDB databases. Robomongo runs on Mac OS X, Windows, and Linux and is free! It allows you to create new databases and view collections and to run queries. It

[2]`https://code.visualstudio.com/docs`

has all the features that the native MongoDB shell provides such as multiple connections, multiple results, and autocompletion. Download and install Robomongo from http://robomongo.org[3]

If you have MongoDB running locally, we can create a new connection as shown below in Figure 1-6.

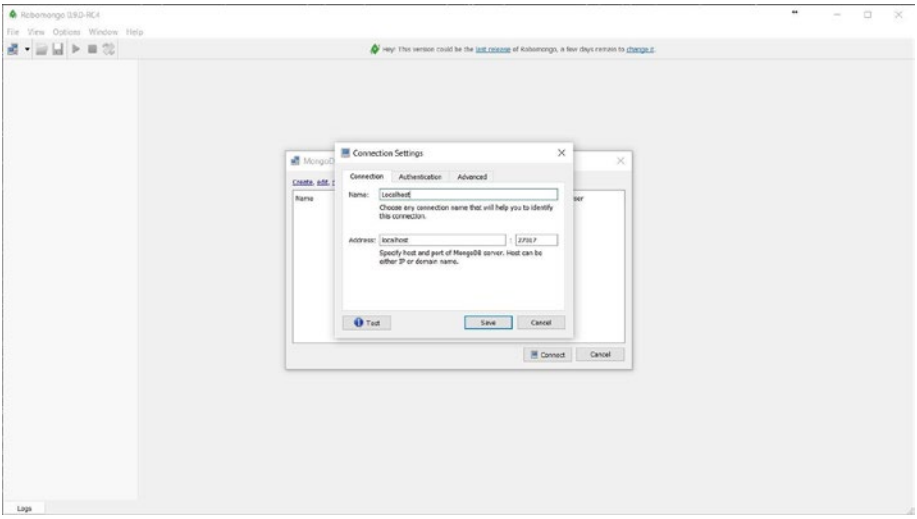


Figure 1-6. *Connect to MongoDB using Robomongo*

After connecting to the instance, we can browse all the collections using the sidebar. See Figure 1-7.

[3]http://robomongo.org

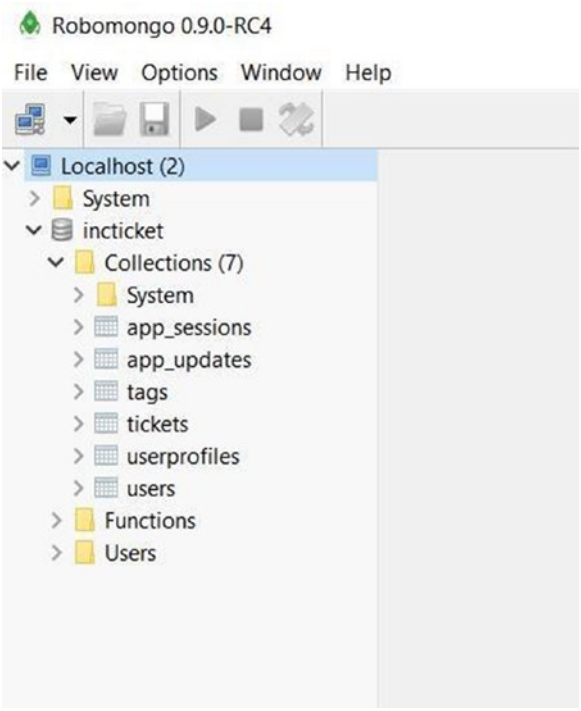


Figure 1-7. *Browse MongoDB collections*

We can issue queries against our MongoDB and collections using the query bar and visually inspect the returned documents, as shown in Figure 1-8.

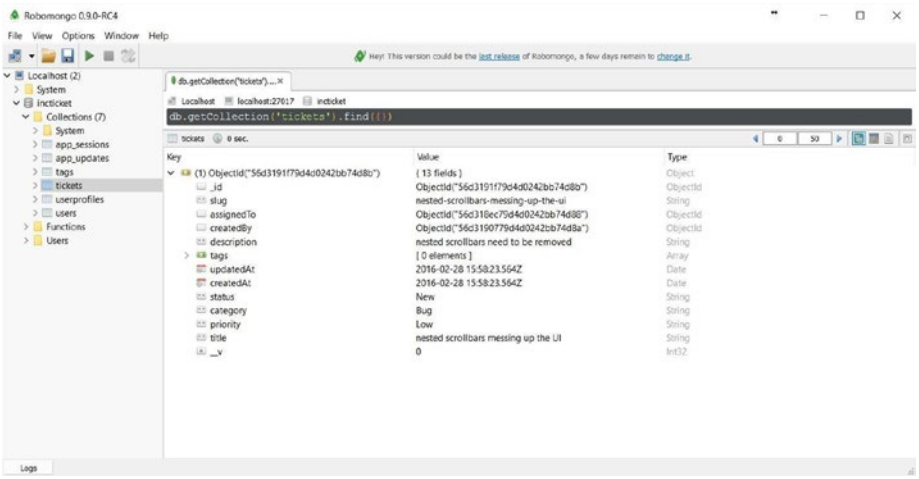


Figure 1-8. *Query MongoDB collections*

1.9 Summary

We have reached the end of the second chapter and we've covered the necessary requirements to begin building the IncTicket application. Onward!

CHAPTER 2

Building the IncTicket Application

This chapter will cover how to create the IncTicket application using KeystoneJS. At the end of this chapter you should have a general idea of how the framework works, understand how the different pieces interact with each other, and give you an understanding on how to easily create KeystoneJS projects with basic functionality. This chapter will get you up and running with a project without too many details.

This chapter will cover the following points:

- Installing KeystoneJS and creating your first project
- Designing models and understanding Mongo collections
- KeystoneJS Administration site for your models
- Working with Mongoose.js
- Building views, routes and templates
- Adding pagination to lists

2.1 Installing KeystoneJS

This section will walk you through installing KeystoneJS on a windows machine. The installation process does not differ much from OS to OS as most of the dependent components are cross-platform compatible. Since KeystoneJS is written in JavaScript, the installation is pretty simple.

2.1.1 Prerequisites

To set up KeystoneJS, you are going to need a few prerequisites. These are:

- Node.js
- Yeoman

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_2

- NPM
- Mongo DB

We already saw how to install Node.js, NPM, and MongoDB in the previous chapter.

KeystoneJS uses Node.js as the platform and uses Mongo DB as the storage back end. **NPM** is the Node Package Manager that enables easy management of Node.js packages from online repositories. It simplifies dependency management so that developers no longer have to manually download and manage scripts. NPM generally comes preinstalled with a Node.js installation.

The easiest way of installing KeystoneJS by using **Yeoman**, a helpful installer that will guide step by step through the process of installing KeystoneJS. Yeoman is a set of tools for automating development workflow. It scaffolds out a new application along with writing build configuration, pulling in build tasks, and NPM dependencies needed for the build. KeystoneJS provides a very handy generator to generate a new project.

First, let's start by installing yo.

```
1   c:\> npm install -g yo
```

Next, to install the yo keystone app generator, use the following command

```
1   c:\> npm install -g generator-keystone
```

This installs the generator as a global package and can be used to generate new projects without needing to reinstall the KeystoneJS generator.

2.2 Creating a New KeystoneJS Application

With yo installed, it's time to get down to business! We will start off creating the application locally, since it will be used to understand much of the instructional material in this book. Setting up a new KeystoneJS application is pretty trivial, thanks to the handy yo keystone application generator. Let's start off by creating a directory to save our project.

```
1   c:\> mkdir incticket
```

and change into that directory.

```
1   c:\> cd incticket
```

Now we can use the 'yo' command from Yeoman to generate the project. The generator will guide you through setting up the project with a few questions and then build the project by installing dependencies from npm. Most of the defaults will suffice for the creation of a project. All the settings can be later changed within the new application. See Figure 2-1.

```
1   c:\incticket> yo keystone
```

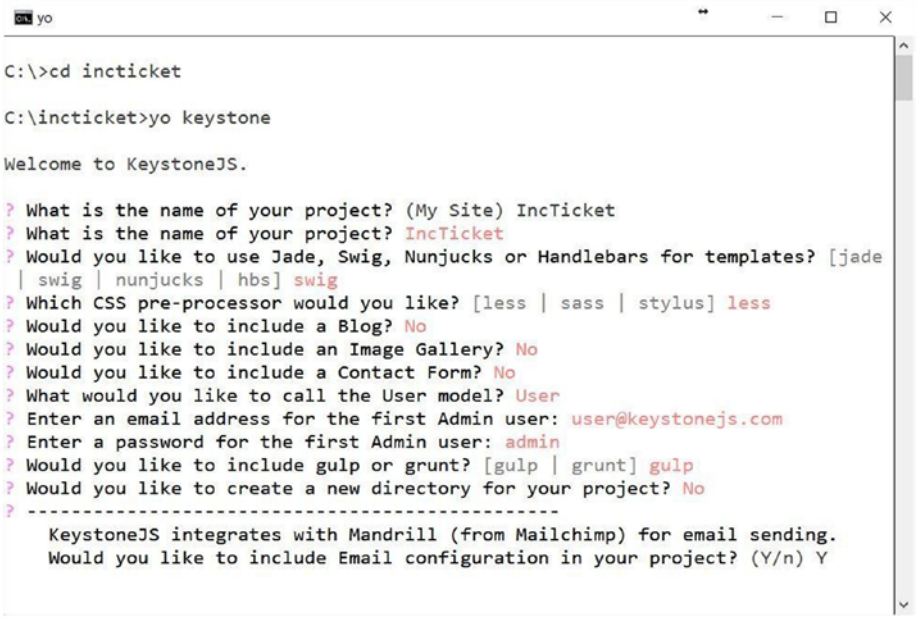


Figure 2-1. *Create project using yo*

The new project will connect to Mongo DB on local host by default. So if you have Mongo DB up and running, we can serve up KeystoneJS using the following command

```
1   c:\incticket> node keystone
```

The above command will serve up your project on port 3000. If you navigate to `http://localhost:3000`, you should see the KeystoneJS landing page (Figure 2-2).

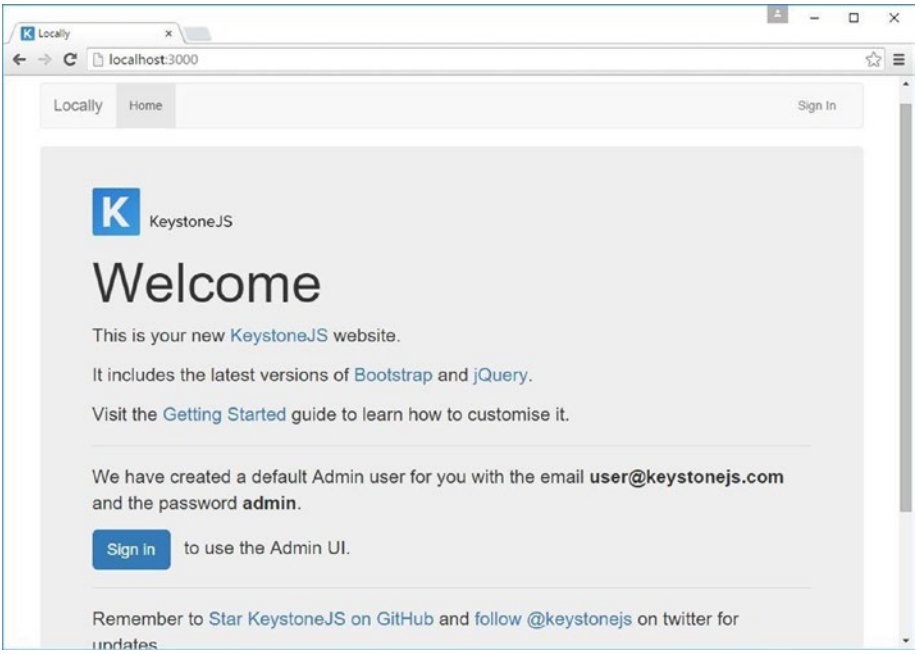


Figure 2-2. *KeystoneJS Default Landing Page*

2.3 Configuring Your KeystoneJS Application

A typical web application goes through a number of deployments in its lifetime: a production push, a staging site, and multiple instances in dev environments maintained by each developer. Although each of those deployments might run the same code, each of these deploys will have environment-specific configurations. The most common example would be database connection settings, such as MongoDB connection URL. Developers may share one instance of a development database, while the staging site and production sites each have their own MongoDB instances. Another example would be to use a different cache driver locally than you do on your production server.

A good solution to maintain separate application-specific configuration is to use environment variables, and keep the config data out of the code. We get a couple of advantages by using environment variables for saving configuration data:

- The configuration data can be easily changed between environments and even isolated deploys. This leads to less complex deploys, which saves time and money.
- There is a decreased chance of the production data leaking out into the wrong hands, which reduces the chances that your database might accidentally be wiped.

KeystoneJS uses an excellent node.js library, namely **dotenv**, to load the configuration data at runtime. In a fresh KeystoneJS installation, the root directory of your application will contain a .env file. This file can be used to hold all our configuration data. All of the variables listed in this file will be loaded into the `process.env` global object when your application starts. It is recommended that you do not commit this file to version control.

Each of the variables in the .env is declared as a key/value pair separated by an equals sign. Keys are generally written in uppercase. A fresh KeystoneJS install .env file would be:

```
1   COOKIE_SECRET=oQQ*sOpz5(bF4gpmoNwM|BDB~db+qwQ`K>Ik~*R2D;;F(8u["l5<.=&Q9
    w+Ul$E=
2   MANDRILL_API_KEY=NY8RRKyv1Bure9bdP8-TOQ
```

To access the configuration variables in our application, we can use them as:

```
1   var madrillApiKey = process.env.MANDRILL_API_KEY;
```

APP IN PRODUCTION

To put a KeystoneJS app into production mode, set the NODE_ENV=production key in the .env file. Setting this enables certain features, including template caching, simpler error reporting, and html minification.

By default, KeystoneJS tries to connect to a local instance of MongoDB and uses no authentication. However, if you want to specify a MongoDB connection string, it is pretty easy to do so using the .env file.

```
1   MONGO_URI=mongodb://user:password@localhost:27017/databasename
```

2.4 Project Structure

Let us take a look at the new directory structure to better understand the different parts that make up a KeystoneJS project. Below is the default directory structure of a KeystoneJS project (Figure 2-3).

Figure 2-3. *KeystoneJS Project Structure*

- The **lib** directory holds any additional libraries that could be needed by our project.
- The **models** folder holds the data models that our project would need. Example model is a User model that deals with user login and user preferences.
- The **public** directory holds static content, related to the web application, such as images, CSS, fonts, JavaScript.
- The **routes** directory contains index.js, middleware.js, and API and views directories.

  The **index.js** file initializes the application routes and associated views. Developers define various routes that respond to HTTP GET, POST, & other HTTP verbs. Each route consists of an HTTP protocol, a URL pattern, and a view that can be invoked as a response or an inline function.

  Middleware.js contains custom code that can be invoked before and after a route has been invoked. This gives the user a powerful option to do custom operations and checks related to authorization, authentication, logging, etc.

The **API**s directory holds controllers that allow for REST interfaces to be exposed that allow clients written in different languages to uniformly interact with our web application. To understand the APIs better, take a look at the Restful API for Mobile and SPA applications chapter.

- The **views** directory contains our application views that respond to various routes. Each view may interact with multiple models to fetch data related to a request and render a template with the data.
- The **templates** directory includes html templates that will be rendered per request to a route. The templates are generally composed during runtime by the inclusion of a master layout and various blocks of the page included via separate partial files. The data that the view queries with the help of models is combined with templates by a templating engine and converted to plain HTML for the browser to render. KeystoneJS supports various templating engines, each having its own syntax. We will look at **Swig**, a node.js templating engine, in a future chapter.
- The **layouts** directory will generally contain the master page that defines various blocks of the page where data can be injected before rendering. Each block has an identifying name. When partial templates are included with the master page, the identifying name is used by the templating engine to render the HTML appropriately. Pages can extend the master page to inherit the layout for a consistent appearance throughout the application.

2.5 Creating Your First Model/List

Let us begin by creating our first KeystoneJS model, also known as List â€“ the Ticket model, which will be used in the application to manage a list of incident tickets for a product. The term model and list will be used interchangeably within the book. A model is a JavaScript object that is an instance of the Keystone.List object, in which each attribute represents a field. KeystoneJS will create a MongoDB collection for each model defined in the models folder. When you create a model, Keystone offers you a practical API to query the database easily using Mongoose.js.

To begin, create a file named Ticket.js in the models directory with following code:

```
1  var keystone = require('keystone');
2  var Types = keystone.Field.Types;;
3
4  var Ticket = new keystone.List('Ticket',{
5          autokey: { from: 'title', path: 'slug', unique: true },
6  });
7
8  Ticket.add({
9          title: { type: String, initial: true, default: '', required:
           true },
```

```
10          description: { type: Types.Textarea },
11          priority: { type: Types.Select, options: 'Low, Medium, High',
            default: '\
12          Low' },
13          category: { type: Types.Select, options: 'Bug, Feature,
            Enhancement', de\
14          fault: 'Bug' },
15          status: { type: Types.Select, options: 'New, In Progress, Open,
            On Hold,\
16          Declined, Closed', default: 'New' },
17          createdBy: { type: Types.Relationship, ref: 'User', index:
            true, many: 17 f\
18          alse },
19          assignedTo: { type: Types.Relationship, ref: 'User', index:
            true, many: \
20  false },
21          createdAt: { type: Datetime, default: Date.now },
22          updatedAt: { type: Datetime, default: Date.now }
23      });
24
25  Ticket.defaultSort = '-createdAt';
26
27  Ticket.register();
```

We begin by requiring the Keystone library so we can use it. A KeystoneJS list allows us to define the attributes for the model that we intend to work with. The first parameter is a key that is used to identify collections uniquely in your MongoDB database. All documents related to the Ticket list will be saved in a collection within MongoDB named as tickets.

The call to register on our KeystoneJS list finalizes the model with any attributes and options we set. Let's take a look at some of the fields we defined for our model:

- **title**: This is the field for the ticket title. The field can hold a string describing the purpose for the ticket. The default option can be used to specify any default value for the field if the user does not input a value. The required option is useful to validate that the field has a value before it is saved. A database index is also used to enforce this.
- **slug**: The slug is used for SEO friendly URLs. The field is defined as part of the List options using the autokey plug-in. Autokey automatically generates a key for each model when it is saved, based on the value of another field. The value of the key is accessible via the 'slug' field on the object. In this case, we create a slug for each ticket from the title. The unique option indicates that we expect the key to be unique throughout the collection. If we create a ticket with the title set to 'My First Ticket' then the automatically generated slug would be similar to 'my-first-ticket'.
- **description**: This is the field to will be used to store the description of the ticket. The Textarea field type will display a text area within the admin UI.

- **priority**: This is a field to the priority of the ticket. We use a select field type, so the value for this field can be set to one of the given choices. The category and status fields are set up in a similar manner.
- **createdBy**: This field will hold a reference to the user that created a ticket. The field is like a foreign key that defines many-to-one relationships in a relational database. This field is displayed as an auto-suggest text box in the admin UI that allows us to pick a single user. Setting the many option to false indicates that only a single user can be selected. Setting the index option to true will tell KeystoneJS that we are interested in a database index to be created for this field. The assignedTo field is a similar relationship field that is used to store a reference to the user that the ticket is currently assigned to. This is the user that will be in charge of resolving the issue mentioned in the ticket.
- **createdAt**: This datetime field indicates when the ticket was created by the user. Since we are using the default value of Date.now, the date will be saved automatically when creating a new ticket object.

As you can see, KeystoneJS comes with different types of fields that you can use to define your models. You can find all field types at http://keystonejs.com/docs/database/#fieldtypes[1]

By setting the **defaultSort** property on the model, we are telling KeystoneJS to sort results by the createdAt field in descending order by default when we query the database. We specify descending order by using the negative prefix.

After saving our model, let us restart our application. Use the command below to restart the application via command line (first stopping the app if it is not already):

```
1   node keystoneJS
```

Restarting will cause keystoneJS to create collections for our models. In our case, we should see the tickets collection in MongoDB.

2.6 Creating an Administration Site for Your Models

Now that we have defined the Ticket model, let us see how to create an administration site to manage tickets. KeystoneJS comes with a built-in administration interface that is very useful for editing content. The KeystoneJS admin site is built dynamically by reading the model metadata and providing a production-ready interface for editing content. You can use it out of the box, configuring how you want your models to be displayed in it.

[1] http://keystonejs.com/docs/database/#fieldtypes

2.6.1 Creating an Admin User

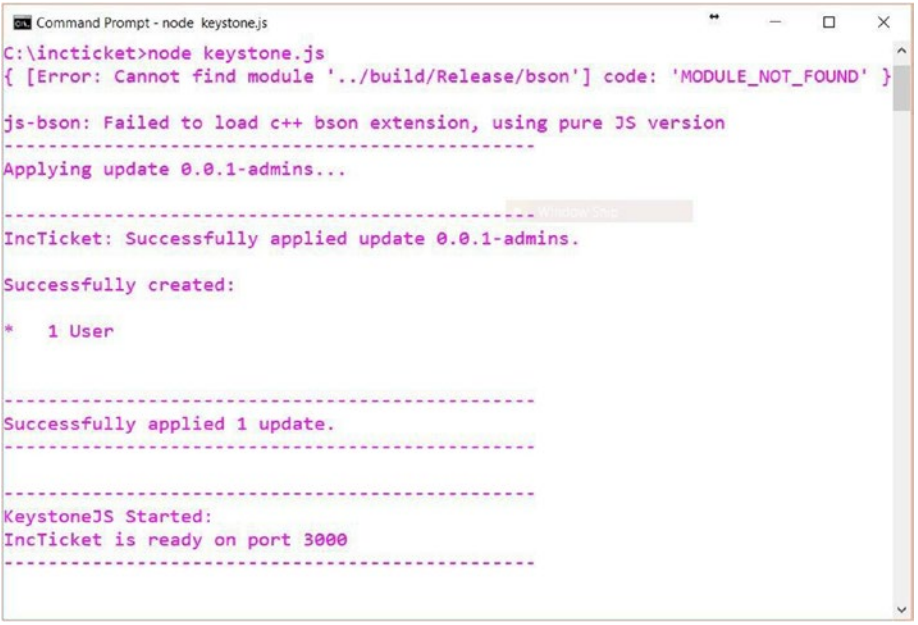
To begin with, we would need a user to manage the admin site. KeystoneJS includes code for the creation of a admin user by default when the app is started for the first time. The user is created using the **updates framework** provided by KeystoneJS. Updates provide an easy way to seed your database, transition data when your models change, or run transformation scripts against your database.

The default admin is created with the code below that is stored at `/updates/0.0.1-admins.js` with the following credentials:

- email - user@keystonejs.com
- password - admin

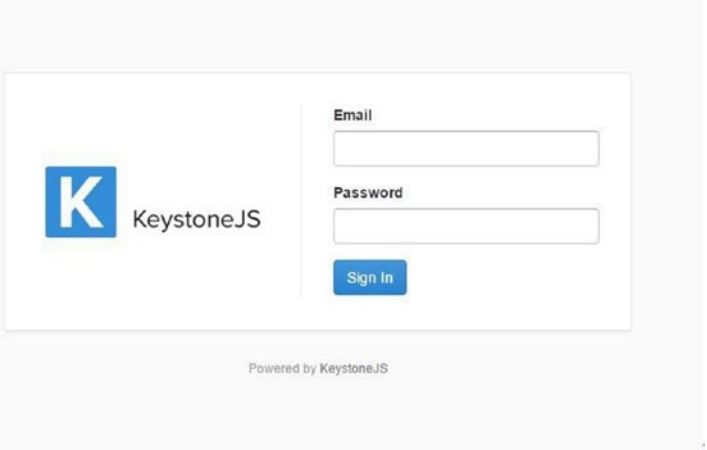
```
1   exports.create = {
2           User: [
3                   { 'name.first': 'Admin', 'name.last': 'User', email:
                    'user@keystonejs.com',
4                   password: 'admin', isAdmin: true }
5           ]
6   };
```

This script automatically creates a default Admin user when an empty database is used for the first time. Updates are run when the app is restarted using `node keystoneJS` command. An update is not applied twice; hence editing the file after starting keystoneJS at least once will not result in changes to admin credentials. We will, however, be able to change the credentials through the admin site.



2.7 The KeystoneJS Administration Site

Start up our app using the node keystoneJS command and open `http://127.0.0.1:3000/keystone/signin` in your browser. You should see the administration login page shown below.



Log in using the credentials of the user created in the previous step. You will see the admin site index page, as shown in the following screenshot:

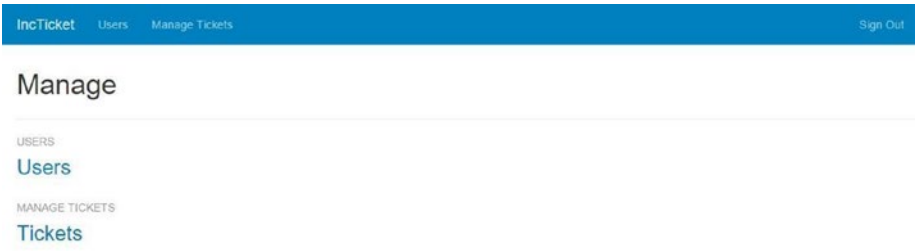
The user model seen on the page is automatically created for us by KeystoneJS. If you click on **Users** you will see the admin user created for us. You can edit the admin user's email address and password to suit your needs and use the new credentials to log in to the application next time.

2.8 Modifying the Admin Menu

The menu items in the administration site can be easily configured in the `/keystoneJS` file. The menu items are stored an object in the configuration with 'nav' set to be the key. As evident in the screenshot above, KeystoneJS classifies any new collections under the 'OTHER' header by default. Let's add the tickets menu item to the menu.

```
1   // Configure the navigation bar in Keystone's Admin UI
2
3   keystone.set('nav', {
4           'users': 'users',
5       'manageTickets': 'tickets'
6   });
```

The first parameter to the **nav** configuration item is the label of the menu item. The second is the collection. After making the changes above, restart the application and the new menu should reflect as below:



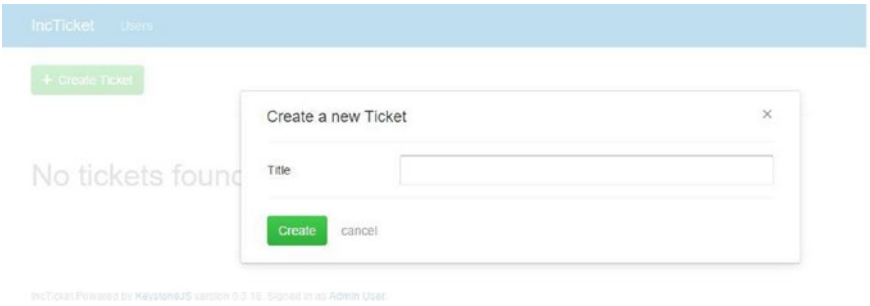
2.9 Adding Models to the Administration Site

When all the fields and options have been set on our model, a call to `Ticket.register()` will register the list with Keystone and finalize its configuration.

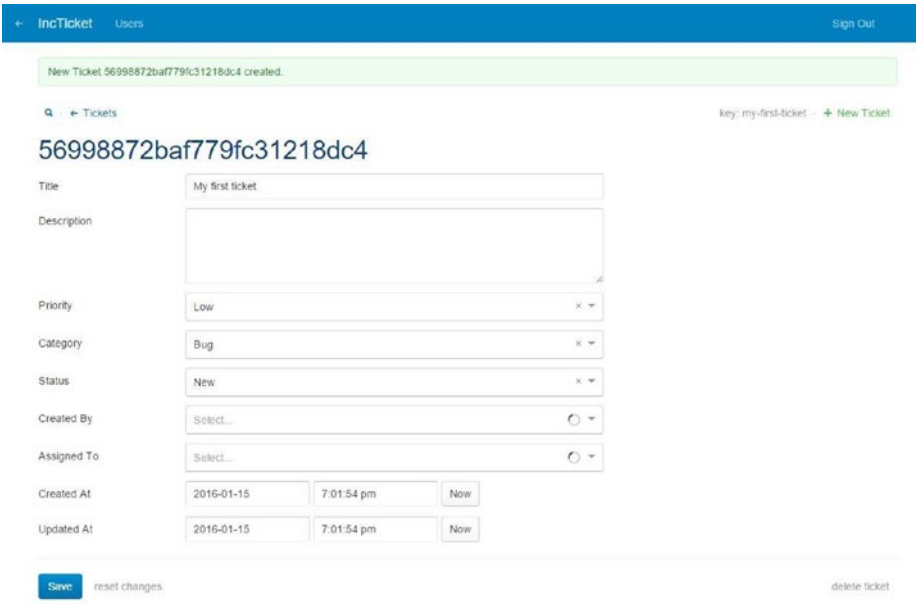
```
1   var keystone = require('keystone');
2
3   var Ticket = new keystone.List(...);
4   ...
5   Ticket.register();
```

When you register a model in KeystoneJS, you get a user-friendly interface generated by inspecting the models that allows you to list, edit, create, and delete objects in an intuitive way.

Click on the Tickets link and then click on the 'Create Ticket' link to add a new ticket. You will see the create item form pop up that KeystoneJS has generated dynamically for the model, as shown in the following screenshot:

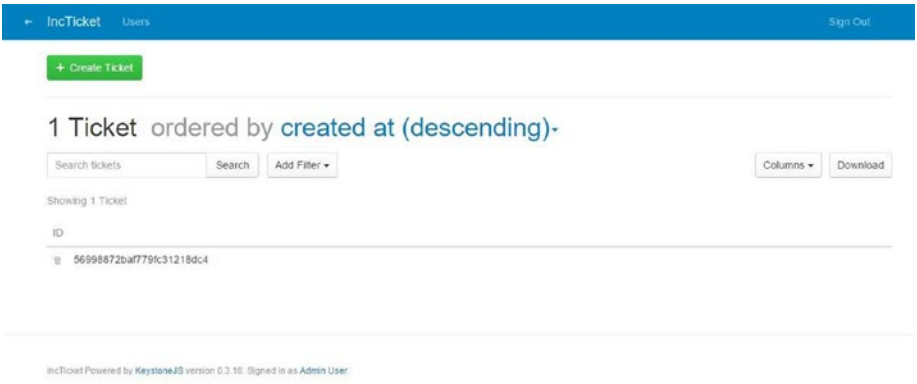


The title field is shown with a text input field in this form as per our definition in the model. The title field was marked with **initial: true** in the Ticket.js file. This causes the field to be shown in the create item form, in the Admin UI. Let us create a ticket with title 'My first ticket'. After creation, KeystoneJS creates a document in the MongoDB tickets collection and returns the object id. The Object id is a 24-character unique identifier that can be used to identify a document across all collections.



KeystoneJS uses different form widgets for each type of field. Even complex fields such as DateTime are displayed with an intuitive interface like a date and time picker form control.

Fill in the form and click on the Save button. You should be shown a successful message indicating your changes were saved. Click on the tickets link to be redirected to the tickets list page as shown in the following screenshot:



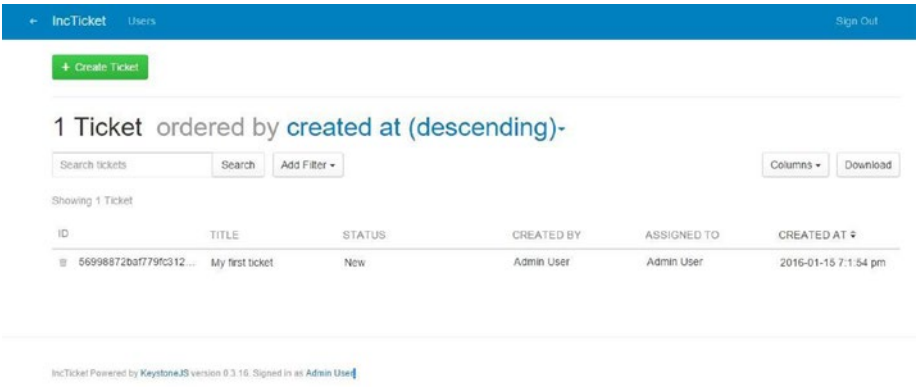
2.10 Customizing the Way Models Are Displayed

On the tickets list page, we see tickets are listed with the object id. This is not very helpful for managing tickets! Let us now see how we can customize the admin site. Edit the Ticket.js file in the models folder and include the following lines:

```
1
2   Ticket.defaultColumns = 'title|20%, status|15%, createdBy, assignedTo,
    createdAt\
3   ';
4   ...
5   Ticket.register();
```

We can set a few different options on the model to provide more information about how to display the model in the admin site and how to interact with it. The **defaultColumns** option allows you to set the fields of your model that you want to display in the admin list page. By default only the object id is displayed. In the above piece of code we are specifying that the title, status, createdBy, assignedTo and createdAt as the default columns to display in the Admin UI, with title and status being given column widths.

Restart the app, go back to your browser, and reload the ticket list page. Now it will look like this:



KeystoneJS is clever enough to recognize that the createdBy and assignedTo are relationship fields to the User model and pulls the user's name for display purposes.

As you can also notice, the page heading indicates that the tickets are ordered by createdAt in a descending order. This is due to another option we already set up on our model – the **defaultSort** option

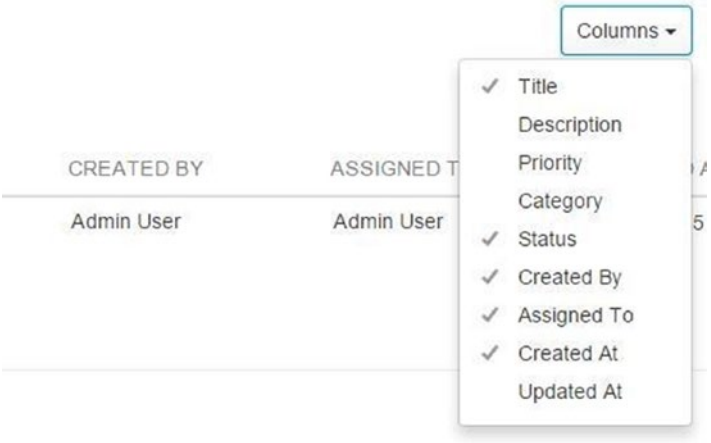
```
1   Ticket.defaultSort = '-createdAt';
```

The admin UI automatically lets us sort data in all the custom columns we added to our default-Columns option. Tickets can be sorted in ascending or descending order using the arrows next to the column headers. Data can be sorted across columns irrespective of whether they hold string, number, Boolean, or date values. How cool is that!

KeystoneJS provides very useful and intuitive ways of managing data through the powerful admin interface. This, I believe, is the key differentiating factor compared other existing CMS frameworks; even across CMSes based on other programming languages.

2.11 Dynamically Adding Columns to Admin UI

One of the most useful features for looking at data in the admin interface is the ability to dynamically add columns that we are interested in without having to modify the definition of the model in code. The columns drop-down on the top right lists all the fields defined on our model. Columns that have been listed in the defaultColumns option will appear with tick marks next to them. We can then pick any additional columns that we are interested in working with.



If a custom field was chosen, it will be added to the end of the displayed column headers. We can reset the list of displayed columns to the original state by using the 'Reset to default' option that will appear in the drop-down once one or more columns have been selected.

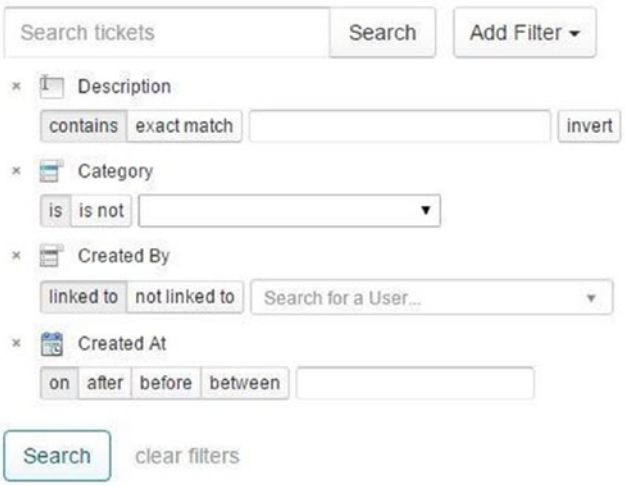
2.12 Finding Data Using the Admin UI

Perhaps the one feature that demonstrates the power of KeystoneJS admin interface is the search functionality. The search box below the page header can be used for searching for data. By default the search will look for data in a field that has been specified in the **autokey.from** path. In the case of searching for tickets, that would be the title field. We can also specify a comma-delimited list of fields to use for searching in Admin UI.

```
1   var Ticket = new keystone.List('Ticket',{
2           autokey: { from: 'title', path: 'slug', unique: true },
3           searchFields: 'description',
4   });
```

Above, we have specified that we are interested in the description field also being included in the search. When we search for a particular keyword, the search is performed using regular expressions to match any part of the saved data. If the search returns only a single search result object, then KeystoneJS will automatically take us to the edit page for that result.

The add filter button allows us to select multiple conditions that we can use to filter the search results. As seen in the screenshot below, the search filters are intelligent enough to provide logical options for filtering based on the type of the field that has been specified in the model.



A string field such as description is presented with a text box with options for matching exactly or on a contains condition. The invert option will try to find results that negate the selected condition, that is, if exact match option was selected then it will try to find results that don't exactly match the keywords; and if contains was selected then it will try to find results that don't contain the keyword.

A select field such as category is presented with options to match the select choices with a 'is' and 'is not' option. The drop-down for the select field is autopopulated with the predefined choices defined in the model.

A relationship field such as user is presented with options to find results that are either linked to or not linked to a related user. The UI also provides a very useful ajax based autocomplete text box to find the related user.

A datetime field such as createdAt is presented with multiple options to narrow down the search results. KeystoneJS can find results that were created on, created after or before a specific datetime, and between two datetime ranges. These are critical pieces of functionality that enhance the usability of an admin UI, which come out of the box with a KeystoneJS application.

2.13 Creating Your First Route

The first thing that comes to mind, after we visited the URL `http://localhost:3000` to see if KeystoneJS works, is how to create a page that is served as a response. To create our own page, we need to define an entry point to our application in the form of a URL and tell KeystoneJS to call a particular JavaScript function when a visitor accesses this URL. This mapping of a URL to a JavaScript function is called a route and forms the core of routing in KeystoneJS.

Routes are typically stored within the `/routes/index.js` file.

2.14 Tickets Route

Let us define a couple of routes in KeystoneJS that can be used to display a list of tickets and a ticket in detail. The most basic KeystoneJS route is simply a combination a URI, a HTTP verb (get, post, etc.), and a JavaScript function that accepts the request, response, and an optional callback handler:

```
1   // Setup Route Bindings
2   exports = module.exports = function(app) {
3   ...
4       app.get('/tickets', function(req, res){
5               res.send('We will show a list of tickets here');
6       });
7   ...
8   }
```

If we add the above code to our routes index file and navigate to http://localhost:3000/tickets, we should see the text – 'We will show a list of tickets here' displayed on the browser. The route we defined will respond to a HTTP get request. KeystoneJS routes can accept the following HTTP verbs:

- get
- post
- put
- head
- delete
- options
- trace
- copy
- lock
- mkcol
- move
- purge
- propfind
- proppatch
- unlock
- report
- mkactivity
- checkout

- merge
- m-search
- notify
- subscribe
- unsubscribe
- patch
- search
- connect

The piece of code below demonstrates how we pass a parameter to a route. Add the code below to the routes index file and we navigate to http://localhost:3000/tickets/test-ticket.

```
1   // Setup Route Bindings
2   exports = module.exports = function(app) {
3   ...
4       app.get('/tickets/:ticketslug', function(req, res){
5                       res.send('We will show a ticket that has a slug : '
                        + req.params.ticketslu\
6   g);
7       });
8   ...
9   }
```

We should see the text – 'We will show a ticket that has a slug: test-ticket' displayed on the browser. For this example, we have assumed that test-ticket is the slug for an existing ticket. The req.params collection can be used to get a reference to the value that is bound to :ticketslug URL parameter.

2.15 URLs for Models

In the above route, we saw that :ticketslug was used as a query parameter to refer to the slug for a Ticket object. The complete URL for a ticket model would be the `/tickets/:ticketslug`. This URL is not part of the model yet and in every place we intend to use or link to a Ticket, we will need to manually build the URL by concatenating the slug. To address this issue, we can use virtual functions to build the canonical URL for Ticket objects. The convention we will follow is to add a URL() virtual method to the model that returns the canonical URL of the object. Edit your Ticket.js model file and add the following before a call to the register() method:

```
1   Ticket.schema.virtual('url').get(function() {
2                   return '/tickets/'+this.slug;
3
4   });
```

The `Ticket.schema` is a reference to the underlying Mongoose schema that is used by Key-stone.List to interact with MongoDB. A virtual function exists on the model but is not persisted to the database. Next, We will use the URL() method in our templates rendered by our views.

2.16 Creating Your First View

In the previous section we were able to execute a piece of code in response to a HTTP request. Let us now see how we generate useful responses rather than just plain text. A view in KeystoneJS terminology is a regular JavaScript function attached to the keystone. View object that responds to a page request by generating a response. This response can be the rendered HTML content of a web page, or a redirect, or a 404 error, or an XML document, or an image or anything.

Views are typically stored in `/routes/views` directory.

Create the following directories and files inside your application directory:

```
routes/
    views/
        index.js
        tickets/
            ticketlist.js
            singleticket.js
```

Since we have already defined the necessary routes, let us create a view that can respond to those routes. Then, finally, we will create HTML templates to render the data generated by the views. Each view will render a template passing variables to it and will return an HTTP response with the rendered output.

2.17 Creating Ticket List and Detail Views

Let's start by creating a view to display the list of Tickets. Create a file named **ticketlist.js** in `/routes/views/tickets`. Edit the new file and make it look like this:

```
var keystone = require('keystone');

exports = module.exports = function(req, res) {

        var view = new keystone.View(req, res);
        var locals = res.locals;

        // locals.section is used to set the currently selected
        // item in the header navigation.
        locals.section = 'tickets';

```

```
12      locals.data = {
13                  tickets: [],
14          };
15
16       // Load all tickets
17          view.on('init', function(next) {
18
19                  var q = keystone.list('Ticket').model.find();
20
21                  q.exec(function(err, results) {
22                          locals.data.tickets = results;
23                          next(err);
24                  });
25          });
26          // Render the view
27          view.render('tickets/ticketlist');
28
29  };
```

You just created your first KeystoneJS view. The tickets view takes the request and response object as the parameters. Remember that these parameters are required by all views. In this view, we are retrieving all the tickets when the view is initialized. The `view.on('init'...)` method is called when the HTTP request comes through the route each time. We query our MongoDB inside this method and set the results to our locals.data.tickets array. These can then be used within the template when it is being rendered.

At the end of the code, we use the render() method provided by KeystoneJS to render the list of tickets with the given template. This function takes the template path. If we specify only the name of the template, KeystoneJS will look for a template with that name under `/templates/views` folder. A view returns a HttpResponse object with the rendered text (normally HTML code). The render() function takes the response context into account, so any variable set within the **response.localss** variable is accessible by the given template. Templates are rendered by Template Rendering Engines. KeystoneJS supports multiple rendering libraries such as Swig, Jade, Ebs. We will look at Swig in depth in a future chapter.

Let's create a second view to display a single ticket. Create a file named **singleticket.js** in `/routes/views/tickets`. Edit the new file and make it look like this:

```
1   var keystone = require('keystone');
2
3   exports = module.exports = function(req, res) {
4
5           var view = new keystone.View(req, res);
6           var locals = res.locals;
7
8           // locals.section is used to set the currently selected
9           // item in the header navigation.
10          locals.section = 'tickets';
11
```

```
12      locals.data = {
13                  ticket: {},
14          };
15
16      // Load all tickets
17          view.on('init', function(next) {
18
19      var q = keystone.list('Ticket').model.findOne({slug: req.params.
        ticketslug});
20
21                  q.exec(function(err, result) {
22                          if(result != null)
23                          {
24                                  locals.data.ticket = result;
25                          }
26                          else
27                          {
28                                  return res.status(404).send(keystone.
                                    wrapHTMLError('Sorry, no tic
29                                  (404)'));
30                          }
31
32                          next(err);
33                  });
34           });
35
36
37           // Render the view
38           view.render('tickets/singleticket');
39
40   };
```

This is the ticket detail view. This view takes a ticket slug to retrieve a published ticket with the given slug. Notice that when we created the Ticket model, we added the unique constraint parameter to the slug field. This way we ensure that there will be only one ticket with a slug for a given title, and thus, we can retrieve single tickets by slug. In the detail view, we are using the res.status() to return a HTTP 404 (Not found) exception if no object is found. Finally, we use the render() method to render the retrieved ticket using a template.

2.18 Creating Templates for Your Views

We have created routes and views for our application. Now it's time to add templates to display tickets in a user-friendly way.

Templates are typically stored in `/templates/views` directory.

Create the following directories and files inside your application directory:

```
templates/
    layouts/
        default.swig
    views/
        index.swig
        tickets/
            ticketlist.swig
            singleticket.swig
```

The `default.swig` file will include the main HTML structure of the website and divide the content into a main content area, header, and a footer section. During installation, KeystoneJS generates some bootstrap boiler plate code within default.swig for us. The `ticketlist.swig` and `singleticket.swig` files will inherit from the `default.swig` file and be rendered by the ticketlist and singleticket views respectively.

KeystoneJS supports many templating languages and Swig is one we will look at. Swig is a powerful template language that allows you to specify how data is displayed on the browser. It is based on template tags, which look like `{% tag %}`; template variables, which look like `{{ variable }}`; and template filters, which can be applied to variables and look like `{{ variable|filter }}`. You can see all Swig template tags and filters at http://paularmstrong.github.io/swig/docs/[2]

Let's look at the default.swig file. All the static assets related to our project such as js, css files are stored in the /public directory. Let's add a incticket.css stylesheet to the /public/styles folder. This style sheet will hold application-specific styling.

Since KeystoneJS uses **express.static** built-in middleware function in Express to serve static assets, we reference assets as if they resided in the root of the root of the application as shown below:

```
<link href="/styles/incticket.css" rel="stylesheet">
```

In default.swig, you can see that there are a few `{% block %}` tags. These tell KeystoneJS that we want to define a content block in that area. Templates that inherit from this template can fill the blocks with content. There is a predefined block called content that we can take advantage of.

Let's edit the `tickets/ticketlist.html` file and make it look like the following:

```
{% extends "../../layouts/default.swig" %}

{% block content %}

        <div class="container">
                <div class="panel panel-primary">
        <!-- Default panel contents -->
        <div class="panel-heading">Tickets</div>
        <div class="panel-body">
```

[2] http://paularmstrong.github.io/swig/docs/

```
10              <p>These are list of tickets in the system.</p>
11          </div>
12
13          <!-- Table -->
14          <table class="table  table-striped">
15              {% for ticket in data.tickets %}
16                 <tr>
17                 <td>
18                  <div class=' col-md-1'>
19                      <span class="label label-info pull-right">{{ticket.
                        status}}<\
20                      /span>
21                  </div>
22
23                  <a href='{{ticket.url}}'><b>{{ticket.title
                    |capitalize}}</b><a>
24
25                  <ul class="ticket-meta">
26                        <li> </li>
27                    <li>
28                        <small>Status</small><a href=""
                          rel="tag">{{ticket.statu\
29                        s}}</a>
30                    </li>
31                    <li>
32                        <small>Priority</small><a href=""
                          rel="tag">{{ticket.pri\
33                        ority}}</a>
34                    </li>
35                    <li>
36                        <small>Category</small><a href=""
                          rel="tag">{{ticket.cat\
37                        egory}}</a>
38                        </li>
39                        <li>
40                                <small>Last Updated</small>
41                                <abbr class="last-
                                  updated">{{ticket._.createdAt.format
                                  ('Do MMM\
42                                M YYYY')}}</abbr>
43                         </li>
44                             </ul>
45                 </td>
46                 </tr>
47              {% endfor %}
48          </table>
49          <div class="panel-footer"></div>
50          </div>
```

```
51
52            </div>
53    {% endblock %}
```

With the {% extends %} template tag, we are telling KeystoneJS to inherit from the layouts/de-fault.swig template. Then we are filling the content blocks of the base template with content. We iterate through the tickets and display their title, date, status, priority, and category, including a link in the title to the canonical URL of the ticket. In the title of the ticket, we are applying a template filter: capitalize – to uppercase the first letter of the input and lowercase the rest. You can concatenate as many template filters as you wish; each one will be applied to the output generated by the previous one.

Let us also update our route to point to the ticketlist view that we created.

```
1   // Setup Route Bindings
2   exports = module.exports = function(app) {
3   ...
4      app.get('/tickets', routes.views.tickets.ticketlist);
5   ...
6   }
```

Open the command prompt and execute the command node KeystoneJS to restart the application server. Open http://127.0.0.1:3000/tickets/ in your browser and you will see everything running. Note that you need to have some tickets in order to see them here. You should see something like this (Figure 2-4).

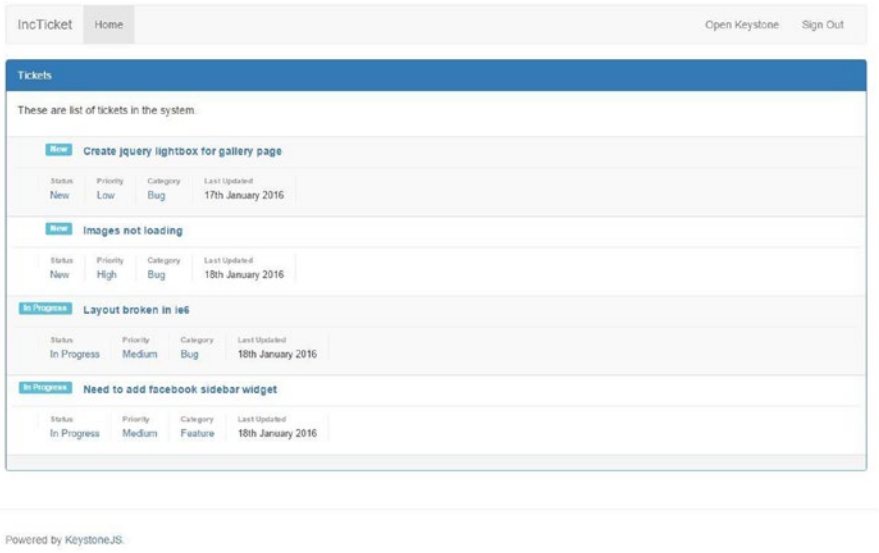


Figure 2-4. *List of Tickets*

Then, let's edit the tickets/singleticket.swig file and make it look like the following:

```
{% extends "../../layouts/default.swig" %}

{% block content %}

        <div class="container">
                <div class="panel panel-primary">
        <!-- Default panel contents -->
        <div class="panel-heading">Ticket details - {{data.ticket.
        title}}
        </div>
        <div class="panel-body">
            <span class="label label-info pull-right">{{data.ticket.
            status}}</sp\
an>
            <p>{{data.ticket.description}}</p>
        </div>
        <ul class="ticket-meta">
        <li>
            <small>Status</small><a href="" rel="tag">{{data.ticket.
            status}}</a>
        </li>
        <li>
            <small>Priority</small><a href="" rel="tag">{{data.ticket.
            priority}}\
</a>
        </li>
        <li>
            <small>Category</small><a href="" rel="tag">{{data.ticket.
            category}}\
</a>
        </li>
        <li>
            <small>Last Updated</small>
            <abbr class="last-updated">{{data.ticket._.createdAt.
            format('Do MMMM\
YYYY')}}</abbr>
        </li>
        </ul>
        <div class="panel-footer"></div>
        </div>

        </div>
{% endblock %}
```

Now, you can go back to your browser and click on one of the ticket titles to see the detailed view of a ticket. You should see the template rendered like this (Figure 2-5).

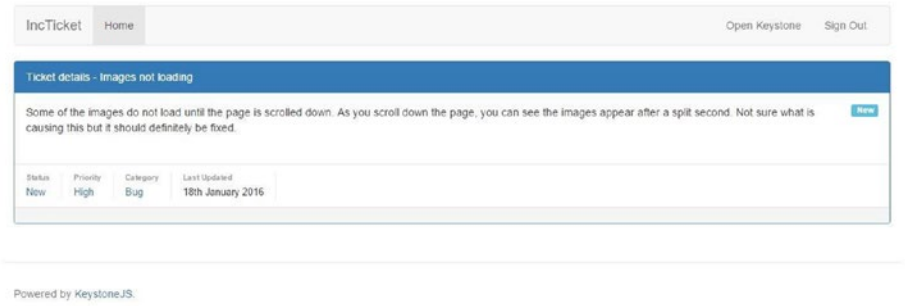


Figure 2-5. *Ticket detail*

Since we have created an SEO friendly URL for our tickets, the URL should look like http://127.0.0.1:3000/tickets/imagenot-loading

2.19 Adding Pagination

As your application grows in terms of content, you will soon realize that you need to split the list of tickets across several pages. KeystoneJS has a built-in pagination functionality that allows you to easily manage paginated data.

Edit the routes/views/ticketlist.js file as follows:

```
1   var keystone = require('keystone');
2
3   exports = module.exports = function(req, res) {
4
5           var view = new keystone.View(req, res);
6           var locals = res.locals;
7
8           // locals.section is used to set the currently selected
9           // item in the header navigation.
10          locals.section = 'tickets';
11
12      locals.data = {
13                  tickets: [],
14          };
15
16      // Load all tickets
17          view.on('init', function(next) {
18
19                  var q = keystone.list('Ticket').paginate({
20                      page: req.query.page || 1,
21                      perPage: 5,
```

```
22                      maxPages: 5
23                  });
24
25                  q.exec(function(err, results) {
26                          locals.data.tickets = results;
27                          next(err);
28                  });
29
30          });
31
32
33          // Render the view
34          view.render('tickets/ticketlist');
35
36   };
```

This is how pagination works:

- We call the List.paginate() that returns a query object. It accepts the following options
  - **page** - page to start at
  - **perPage** - number of results to return per page
  - **maxPages** - optional, causes the page calculation to omit pages from the beginning/mid-dle/end
- We get the page GET parameter that indicates the current page number.
- If the page parameter is not an integer, we retrieve the first page of results.
- We pass the retrieved objects to the template.

When you call exec on a paginated query, it will return a lot of metadata along with the results:

- **total**: all matching results (not just on this page)
- **results**: array of results for this page
- **currentPage**: the index of the current page
- **totalPages**: the total number of pages
- **pages**: array of pages to display
- **previous**: index of the previous page, false if at the first page
- **next**: index of the next page, false if at the last page
- **first**: the index of the first result included
- **last**: index of the last result included

Let us update our ticketlist.swig template to include the pagination links displayed below. We would also need to iterate over `data.tickets.results` instead of just `data.tickets` as we did earlier.

```
{% for ticket in data.tickets.results %}
....
{% endfor %}
```

```
<ul class="pagination">
        {% if data.tickets.totalPages > 1 %}

        {% if data.tickets.previous %}
        <li><a class="page-num" href="?page={{ data.tickets.previous
        }}">Previous</a></\
        li>
        {% else %}
        <li><a class="page-num" href="?page=1">Previous</a></li>
        {% endif %}

        {% for i,p in data.tickets.pages %}
        <li><a class="page-num {% if data.tickets.currentPage == p %}
        active {% endif %\
}" href="?page={% if p == '...' %} {% if i+1 == data.tickets.totalPages
%} 1 {% \
else %} {{ p }} {% endif %}{% else %}{{ p }}{% endif %}">{{ p }}
</a></li>
        {% endfor %}

        {% if data.tickets.next %}
        <li><a class="page-num" href="?page={{ data.tickets.next
        }}">Next</a></li>
        {% else %}
        <li><a class="page-num" href="?page={{data.tickets.
        totalPages}}">Next</a></li>
        {% endif %}

        {% endif %}
</ul>
```

Now, open http://127.0.0.1:3000/tickets/ in your browser. You should see the pagination at the bottom of the ticket list and you should be able to navigate through pages (Figure 2-6).

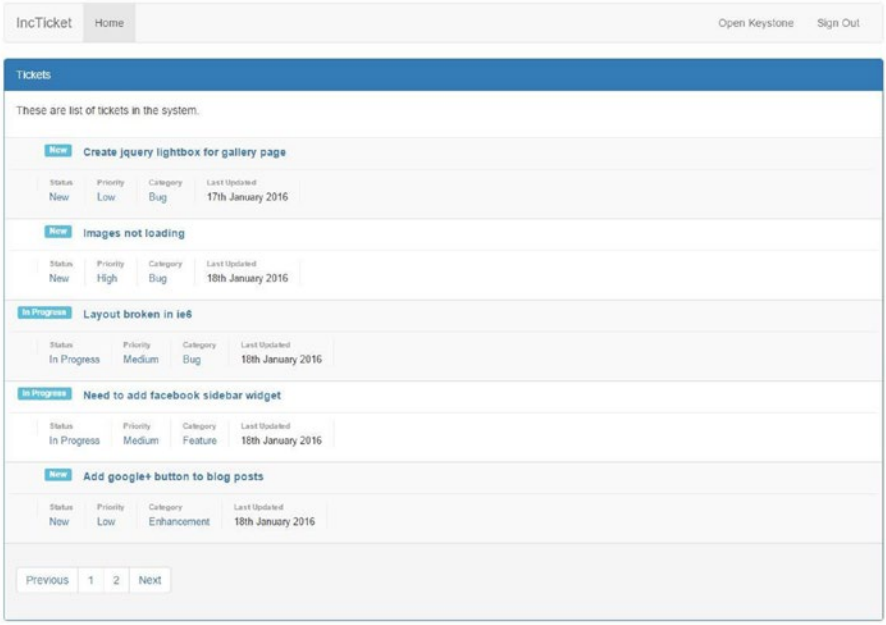


Figure 2-6. *List tickets with pagination*

2.20 Summary

In this chapter, you have learned the basics of the KeystoneJS web framework by creating functionality for a basic ticket listing. You have designed the data models, views, templates, and URLs for your application, including object pagination.

CHAPTER 3

Introducing KeystoneJS Models

3.1 Introducing the Mongoose ODM

MongoDB is very good at storing data in JSON format. And Mongoose, a Node.js Object Document Mapper, makes it easy to work with the data stored in your Mongo DB database. An ODM, or object document mapper, simply put, translates between your objects in code and the document representation of the data. It creates an abstraction layer between the application code and the database and allows for easier programming. MongoDB is schema less, meaning each object stored in a collection does not need to have properties similar to other objects. This provides great flexibility for the developers to store heterogeneous pieces of data.

In a relational database, a table schema provides a rigid structure to the data and the developer has to make sure to insert data that follows the schema, or else saving data will fail. If data requirements change or new properties are introduced, we would need to alter the underlying table to allow the change and/or provide a default value to the new property. When using MongoDB, if the data requirements change or a new property is added, we can directly save a document with the necessary information and simply retrieve it back. This means we technically don't need to adhere to any schema while working with Mongo DB.

While being schema less has advantages, it can quickly become difficult to keep track of the contents of our Mongo collections. We can use Mongoose to provide consistent readable schemas for our collection. It also gives us a clean interface for communication with our MongoDB database. Mongoose maps data in our Mongo collection into full-blown JavaScript objects that have data and methods for validation and other business logic.

Mongoose ODM is fully supported by the MongoDB team and offers a vast array of convenient features useful for querying, inserting, updating, and deleting documents; and managing relationships between documents.

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_3

3.2 Mongoose Schemas and Keystone Lists

Mongoose schemas are blueprints of documents that we intend to store in our MongoDB. The schema defines any fields, their data types, and any constraints we want to add on them before storing into a collection. The data types that Mongoose supports are the following:

- String
- Number
- Date
- Buffer
- Boolean
- Mixed
- ObjectId
- Array

The Mixed field can be used to store data of any arbitrary JSON representation. The ObjectId field can be used to store the ObjectId, a unique identifier for a document. This is useful while defining relationships. The array data type can be used to hold a collection of nested schemas.

While a schema defines the attributes of the document, a Model is a concrete instance of the schema that can be used to invoke schema methods and access data. Mongoose models provide features such as saving, finding, creating, and deleting documents.

KeystoneJS builds on top of Mongoose schemas to create "Lists." Lists basically wrap and extend a Mongoose schema. KeystoneJS automatically creates and registers a Mongoose schema and Mongoose model when a list is created.

KeystoneJS includes Mongoose as a package dependency. So we do not need to install Mongoose separately using the node package manager (npm). To access Mongoose from within our KeystoneJS application, we can use the following:

```
1   var keystone = require('keystone');
2   var mongoose = keystone.mongoose
3   var Schema   = mongoose.Schema;
```

3.3 Adding Fields to Your Model

In the previous chapter, we learned how to define the Ticket Model/List. Let us add a new Model that defines a Product to which a particular ticket is related to and add fields to our Product model so we can save data related to those fields and use them in our application. We will need to first visualize the type of data we want to save and the associated data types that will work. A Product model at the least contains a name, a team name that the product belongs to, a published status that will indicate if the particular product is in published or draft stage, and a create date to indicate when the product document was created within the application.

To add fields to our Product model, we should use the `Product.add(fields)` method, where fields is an object of keys and corresponding values. The keys identify the field name or is also referred to as field path. The values indicate the type of field and its associated options. A field is a defined as a JavaScript object that has a type attribute. The type attributes must be a basic data type that Mongoose natively supports such as string, number, or it should be a KeystoneJS Field Type. Each field can have additional options that alter its behavior such as filtering and indexing.

KeystoneJS fields allows you to easily add rich, functional fields to your application's models. Adding a KeystoneJS field to our model gives us a lot of added functionality such as the following:

- Rich controls in Keystone's Admin UI
- Complex data types; for example, the location field stores address and lng/lat point
- Formatting and validation methods
- Metadata about how fields relate to each other; for example, which fields depend on certain values in other fields

Here is a list of basic data types that are mapped to their corresponding KeystoneJS field types:

Data type	Field type
String	Text
Number	Number
Date	DateTime
Boolean	Boolean

Let us add our fields to the Product model.

```
1     var keystone = require('keystone');
2     var Types = keystone.Field.Types;
3
4     var Product = new keystone.List('Product');
5     Product.add({
6         name: { type: String, required: true },
7         team: { type: String },
8         publishedStatus: { type: Types.Select, options: 'draft,
          published', defa\
9   ult: 'draft' },
10        createdAt: { type: Date, default: Date.now } 11
11
12      });
```

```
13        Product.register();
```

As seen above, we have defined four properties/fields within the Product model with their types and additional options. The name, team, and createdAt fields are of the basic JavaScript data types, namely, String and Date respectively. The publishedStatus is a select field that uses KeystoneJS's built-in Field Types. By setting the required option, KeystoneJS will ensure that a new product document is not created without those fields being entered by the user. The default option enables us to specify fallback values for a field if a user does not enter one. The select field will render a HTML select drop-down with two options – draft and published. The default option on the publishedStatus is set to be draft.

3.4 Defining Virtual Properties

Virtual properties are accessors and mutators that allow us to format our fields when retrieving them from a model or setting their value. To define a virtual property, we add it to the underlying Mongoose schema. Let us add a virtual property that returns the full title of a product that is a combination of product name and product team name.

```
1   //our virtual property to display full title
2   Product.schema.virtual('fulltitle').get(function() {
3       return this.title + ' ' + this.team;
4   });
5
6   Product.register();
```

To access the Mongoose schema from our KeystoneJS List, we need to reach into the schema property of the List. This property is a passthrough for the options that we want the Mongoose schema to possess. The virtual method on the schema takes a parameter as the name of the property. We can use this property directly on our model within our application as if we had added it to our list. The advantage of virtual properties is that they are not persisted to the document saved within MongoDB.

When we fetch a document from our Mongo DB, we can use the virtual property as

```
1   console.log(productObj.fulltitle);
```

3.5 Finding Data

KeystoneJS provides you with numerous ways to fetch data from your database, each with their own appropriate use case. You can simply fetch all records in one go; a single record based on a field; data based on conditions; or a paginated list of either all or filtered data. There are two major syntaxes for querying data:

- Using the QueryBuilder interface
- Query with a Single Operation

3.6 Using the QueryBuilder

We can use the powerful QueryBuilder interface provided by Mongoose to build up the query in a fluent manner before execution. The QueryBuilder allows us to disperse code over multiple lines to compose our query before executing it at a certain point. Look at the example in the following snippet:

```
1   ticketQuery = keystone.list('Ticket').model.find();
2   ticketQuery.where('priority').in(['Low', 'Medium']);
3   ticketQuery.sort('-createdAt');
4   ticketQuery.select('title priority category');
5   ticketQuery.exec(function (err, tickets){
6     if (!err){
7       console.log(tickets); // output array of tickets
8     }
9   });
```

The QueryBuilder builder syntax is pretty simple to understand. In the above query, we are trying to retrieve all tickets where the priority is either low or medium. We want the tickets to be ordered by created date in a descending order (that is, most recent first) and only return the title, priority, and category fields. The query is actually executed when the exec function is called.

We can also chain the commands using dot syntax. The earlier code could be rewritten using the dot syntax method as in the following:

```
1   keystone.list('Ticket').model.find();
2       .where('priority').in(['Low', 'Medium'])
3       .sort('-createdAt');
4       .select('title priority category');
5       .exec(function (err, tickets){
6             if (!err){
7               console.log(tickets);
8             }
9       });
```

3.7 Query with a Single Operation

The QueryBuilder makes it easy to compose complex queries and helps improve the code readability and maintainability. Sometimes however, we might just need a simple query to satisfy our needs and it might be equally well readable in well-formatted code.

To use a single-query operation, we pass the necessary query conditions and constraints to the find() method. Below is a rewrite of the query we previously saw, but in a single operation. `Keystone.list('Ticket').model.find({ priority: { $in: ['Low', 'Medium'] }, }, ['title', 'pri-ority', 'category'], function(err, tickets){ if (!err){ console.log(tickets); } }).sort('-createdAt');`

We can also use a couple of helper methods that simplify querying: * **Model.find(query)** to return an array of documents matching the query * **Model.findOne(query)** to return the first document found that matches the query * **Model.findById(ObjectID)** to return a single document that matches the given ObjectID

3.8 Retrieving All Tickets

To fetch all tickets, we can use the find method:

```
1   var q = keystone.list('Ticket').model.find();
2
3   q.exec(function(err, results) {
4           var tickets = results;
5           next(err);
6   });
```

3.9 Retrieving a Ticket by Slug

To fetch a ticket that matches the slug, we can use the find method shown below. The slug can be read from req.params collection.

```
1   var q = keystone.list('Ticket').model.findOne({'slug':req.params.slug});
2
3   q.exec(function(err, result) {
4           var ticket = result;
5           next(err);
6   });
```

3.10 Selecting Specific Fields

For optimal performance, it is always advised to construct queries that retrieve only the necessary data required to render a view. For example, if you're displaying a list of tickets that only displays the ticket title, priority, and category, there is no reason to retrieve the id, createdAt, status, assignedTo, and other columns. You can restrict which columns are selected using the select method, as demonstrated below:

```
1   var q = keystone.list('Ticket').model
2       .findOne({'slug':req.params.slug})
3       .select('title priority category');
4
5   q.exec(function(err, result) {
6           var ticket = result;
7           next(err);
8   });
```

3.11 Counting Documents

To count the number of documents associated with a given query, use the `count` method.

```
1   var q = keystone.list('Ticket').model
2       .count({'category':'Bug'});
3
4   q.exec(function(err, count) {
5           console.log('There are %d tickets categorized as Bug', count);
6           next(err);
7   });
```

3.12 Ordering Documents

You can order records using the `sort` method in conjunction with the find method. The following example will retrieve all tickets records, ordered by title.

```
1   var q = keystone.list('Ticket').model.find()
2                   .sort('title');
3
4   q.exec(function(err, results) {
5           var tickets = results;
6           next(err);
7   });
```

By default, the results are sorted in ascending order. You can change this default behavior by prefixing a minus sign to the field that is being used to sort. We can also use the asc, desc, ascending, and descending keywords if we pass the condition as an object.

```
1   var q = keystone.list('Ticket').model.find()
2                   .sort('-title');
3
4   q.exec(function(err, results) {
5           var tickets = results;
6           next(err);
7   });
```

A short-form notation is to use 1 and -1 to represent ascending and descending respectively. We can also sort on multiple fields as shown below. We will sort by category ascending and priority descending.

```
1   var q = keystone.list('Ticket').model.find()
2                   .sort([['category', 1], ['priority', -1]]));
3
```

```
4   q.exec(function(err, results) {
5           var tickets = results;
6           next(err);
7   });
```

3.13 Conditional Filtering

We can use the where method to conditionally find documents with attributes that we are interested in. As an example, let us try to retrieve documents that are categorized as Bug and also with Low priority.

```
1   var q = keystone.list('Ticket').model.find()
2                   .where('category').equals('Bug")
3                   .where('priority').equals('Low');
4
5   q.exec(function(err, results) {
6           var tickets = results;
7           next(err);
8   });
```

As you see in the above code, multiple where clauses can be chained together, which makes it easy to read and maintain the code. The field that we are interested in comparing is passed to the where method, and the comparison operator is chained with the comparison value. The where condition can be filtered using comparison helpers such as equals, gt (greater than), lt (less than), in; this makes querying very easy. We saw an example of querying using the 'in' condition in our dot syntax sample above.

3.14 Limiting Returned Documents

Most often we will want to just retrieve a small subset of documents, for instance, the ten most recently added tickets. You can do so using the `limit` method:

```
1   var q = keystone.list('Ticket').model.find()
2                   .sort('-createdAt')
3                   .limit('10');
4
5   q.exec(function(err, results) {
6           var tickets = results;
7           next(err);
8   });
```

If you wanted to retrieve a subset of documents beginning at a certain offset, you can combine limit method with the skip method. The following example will retrieve ten recent tickets beginning with the sixth record:

```
var q = keystone.list('Ticket').model.find()
                .sort('-createdAt')
                .skip(5)
                .limit('10');

q.exec(function(err, results) {
        var tickets = results;
        next(err);
});
```

3.15 Check for Existence of a Field

We can use the `exists` method to determine whether a particular document contains a field, without actually loading it. For example, to determine a list of tickets that are unassigned, that is, the assignedTo field does not exist on the Ticket MongoDB document, use the following statements:

```
var q = keystone.list('Ticket').model.find()
                .where('assignedTo')
                .exists(false);


q.exec(function(err, results) {
        var tickets = results; //list of unassigned tickets
        next(err);
});
```

3.16 Inserting a New Record

To create and save a new document, use the `save` method. You'll first create a new instance of the desired model, update its attributes, and then execute the save method:

```
 var keystone = require('keystone')
         Ticket = keystone.list('Ticket');


     var newTicket = new Ticket.model();

         newTicket.title = 'Update sidebar widget to include Facebook';
         newTicket.category = 'Feature';
         newTicket.status = 'In Progress';
         newTicket.priority = 'Medium';
         newTicket.description = 'Should be able to add Facebook sidebar
         widget to inter\
         ior pages';

```

```
14          newTicket.save(function(err) {
15                  if (err) {
16                          console.error("Error adding new ticket to the
                            database:");
17                          console.error(err);
18                  } else {
19                          console.log("Added ticket " + newTicket.title +
                            " to the database.");
20                  }
21                  done(err);
22          });
```

■ **Note** Because we set the autokey option on our Ticket list, it will have generated a unique slug based on the title before it was saved to the database.

In a real-world scenario, you will get the necessary data for the attributes from a HTML form post in your view. If you have a JavaScript object with all the attributes set, then you can pass that to the model and invoke the save method to save it to the database.

```
1   var keystone = require('keystone')
2           Ticket = keystone.list('Ticket');
3
4   var ticketObj = {
5       'title': 'Update sidebar widget to include Facebook',
6       'category': 'Feature',
7       'status': 'In Progress',
8       'priority': 'Medium',
9       'description': 'Should be able to add Facebook sidebar widget to
        interior pa\
10  ges'
11  };
12
13  var newTicket = new Ticket.model(ticketObj);
14
15      newTicket.save(function(err) {
16              if (err) {
17                      console.error("Error adding new ticket to the
                        database:");
18                      console.error(err);
19              } else {
20                      console.log("Added ticket " + newTicket.title + "
                        to the database.");
21              }
22              done(err);
23      });
```

3.17 Updating Existing Records

Users will commonly want to update existing tickets, perhaps changing the assigned user or the status of the ticket. To do so, you'll use KeystoneJS's UpdateHandler functionality. This typically involves retrieving the desired document using its identifier, and setting the changed fields on the document and requesting KeystoneJS to process the updates.

Let us assume we receive the id of the ticket via a form post along with the changes. The code below first retrieves a ticket that matches the provided id. If the ticket is found, any matching fields and their values (from the form post) are set to the data object. The getUpdateHandler method on the matching ticket can process the updates to the document via a call to the process method. The data object is provided as an input to this method. “‘ Ticket.model.findById(req.body.id).exec(function(err, item) {

```
        if (err) return res.apiError('database error', err);
        if (!item) return res.apiError('not found');

        var data = req.body;

        item.getUpdateHandler(req).process(data, function(err) {

                if (err) return console.error('create error', err);

                console.log("Successfully updated the ticket");

        });

 });
```

```
## Deleting Documents

To delete a document, first locate the document, then use the
```remove``` metho\
d. Below, we will find a Ticket that has the slug need-to-add-facebook-
sidebar-w\
idget and delete it.
```

var keystone = require(‘keystone’), Ticket = keystone.list(‘Ticket’);
var ticketSlug = ‘need-to-add-facebook-sidebar-widget’;
Ticket.model.find({‘slug’: ticketSlug}) .remove(function(err)
{ // ticket has been deleted }); “‘

3.18 Summary

In this chapter, we learned how to create and work with models to save, retrieve, and manipulate data. These are the most basic operations in all web applications and KeystoneJS makes it a breeze to implement!
```

CHAPTER 4

Model Relations

In a fairly simplistic view, we have seen how to create individual models, add properties to them, interact with models, and learned about Mongo collections in the previous chapters. This was a great beginning point. However, in real-world applications there will always be a need for models to be interconnected with each other.

Interconnections, also known as relationships between models, make it possible for us to determine which incident tickets are associated with a product, associate a specific user with a list of tickets, and identify all tickets that belong to a particular queue.

KeystoneJS works in conjunction with Mongoose and MongoDB to provide very useful features for building and traversing relations between models with ease. KeystoneJS utilizes MongoDB's ability to store the ObjectIDs of related documents in a field (or many related ObjectIDs in an Array) to provide powerful relationship capabilities.

4.1 Defining Relations

To understand relations, let's start by looking at a simple use case. For instance, an incident ticket can be assigned to a single user, to be resolved, but a single user can handle multiple ticket resolutions. So how is this relationship formally defined in a KeystoneJS application? Furthermore, how do we traverse a relation, for instance, to show all tickets that have been assigned to a single user? KeystoneJS's wrapper aroung Mongoose.js makes such relationships a breeze to work with.

KeystoneJS supports thr following type of relationships:

- The **One-to-Many Relation**: One-to-Many relations are used when one model document can be associated with multiple documents of another single model. For instance, a user can have many tickets and one ticket can belong only to one user.
- The **Many-to-Many Relation**: Many-to-Many relations are used when one document of a model can be related to multiple documents from another model and vice versa. For instance, an incident ticket can be tagged with many tags and each tag could be associated with many incident tickets.

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_4

- The **One-to-One Relation**: Although modeling One-to-One relations in NoSQL is thought of as being counterintuitive, we can still achieve it within KeystoneJS. A one-to-one relation is used when one model document can belong to only one other model document. For example, a user can have only one user profile and one user profile can belong to only a single user. A user profile generally contains related information about a user such as a list of his social network accounts and his bio. The user profile will be a separate model that differs from the user's authentication model. Because one user can be associated with one user profile and one user profile can be related to only a single user, this is a good case for a one-to-one relationship. We can implement a One-to-One relation in KeystoneJS with an inherent One-to-Many relation with a constraint that limits the document from being saved if the One-to-One characteristic is violated.

4.2 Modeling One-to-Many Relations

For the purpose of illustrating how One-to-Many relationships can be easy implemented using KeystoneJS, let us look at the relationship between a user and ticket. One user can have many tickets; however one ticket will belong to only a single user.

If you recall from the earlier chapter, we have defined the relationship between a ticket and user on the Ticket Model. The field is of type Types.Relationship and the ref option is set to the User model, which indicates the Model it is related to. Setting many to false indicates that we can only select one user for this field.

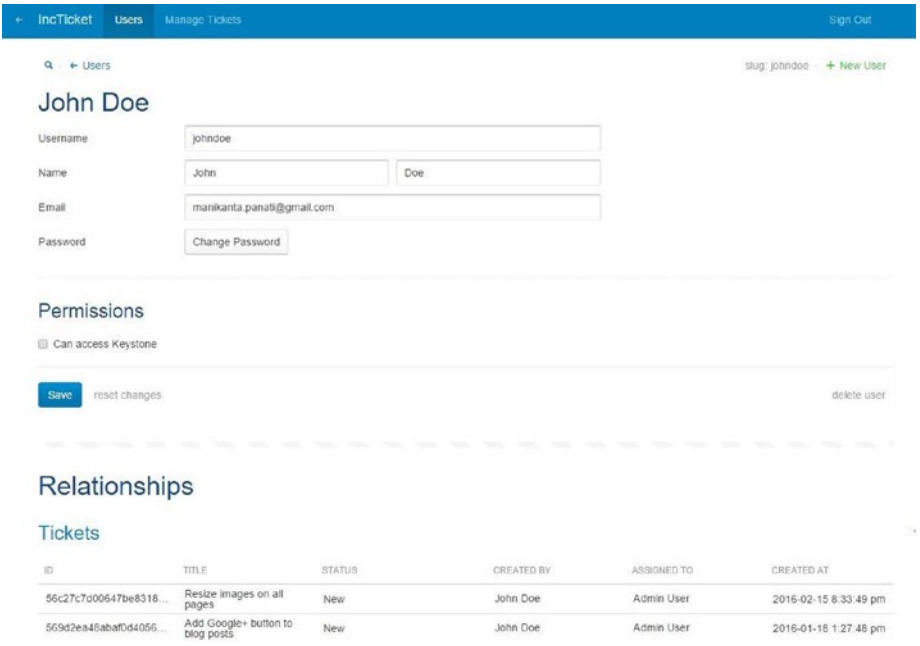
```
1   createdBy: { type: Types.Relationship, ref: 'User', index: true, many:
    false }
```

To represent the relationship from both sides, we can define the relationship on the user model as well. We can do this by calling the relationship method on the user model. Add the line below to the user model.

```
1   User.relationship({ path: 'tickets', ref: 'Ticket', refPath: 'createdBy'
    });
```

- **path**: This option defines the path of the relationship reference on the Model
- **ref** : This option is the key of the referred Model (the one that has the relationship field)
- **refPath**: This option specifies the field of the relationship being referred to in the referred Model

Restart the application and navigate to any user on the admin site and you should notice a relationship section on the page that links to all the tickets that were created by that user.



4.3 Retrieving a One-to-Many Relation

We can easily find one-to-many related items. Simply specify the ID of the item you wish to filter on like any other value. To find tickets that are created by a specific user, use the following code:

```
1   keystone.list('Ticket').model.find().where('createdBy', user.id).
    exec(function(e\
2   rr, tickets) {
3   // ...
4   });
```

KeystoneJS makes it very simple to be able to retrieve a related document automatically. To be able to get the user related to a ticket, we can use the populate method. KeystoneJS takes advantage of the underlying Mongoose populate functionality to provide this. The relevant code to load user information along with a ticket would be the following:

```
1   keystone.list('Ticket').model.findOne().populate('createdBy
    assignedTo').exec(fu\
2   nction(err, ticket) {
3
```

```
4       console.log('The ticket is created by ' + ticket.createdBy.
        username);
5       console.log('The ticket is assigned to ' + ticket.assignedTo.
        username);
6
7   });
```

To automatically populate all the tickets created by a user, we can use the populateRelated method. However this method can be inefficient if there are a lot of tickets per user.

```
1   keystone.list('User').model.findOne().exec(function(err, user) {
2     user.populateRelated('tickets', function(err) {
3       // tickets are populated. access using user.tickets
4     });
5   });
```

The user's tickets will be added as a field on the user object.

4.4 Modeling Many-to-Many Relations

For the purpose of illustrating Many-to-Many relations, let us introduce tagging to our tickets. Each ticket can be be tagged with multiple tags and a single tag can be associated with many tickets. Hence there is a many-to-many relationship between tickets and tags.

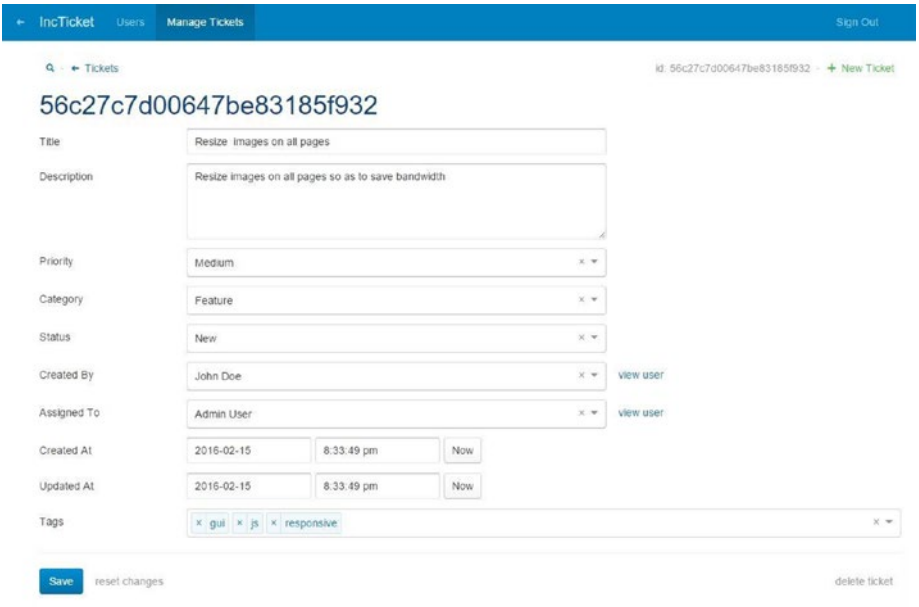
Add a field named tags to the ticket model as below: `tags: { type: Types.Relationship, ref: 'Tag', many: true }`, We have specified that the field type is Types.relationship and that the field is a reference pointing to the Tag model. By setting the many options to true, we specify that multiple tags can be selected.

Next, create a file named `Tag.js` in /models with the following code in it:

```
 1   var keystone = require('keystone');
 2   var Types = keystone.Field.Types;
 3
 4
 5   var Tag = new keystone.List('Tag', {
 6           autokey: { from: 'name', path: 'slug', unique: true }
 7   });
 8
 9   Tag.add({
10           name: { type: String, required: true }
11   });
12
13   Tag.relationship({ ref: 'Ticket', refPath: 'tickettags',
     path: 'tags' });
14   Tag.register();
```

We have defined a tag to contain just the name and a unique slug. This will prevent duplicate tags from being created. We have called the relationship method on the Tag list and specified that the model that it is related to is Ticket via the tags field.

Create some tags in the admin site and you should be able to pick multiple ones on the ticket edit page as shown below:



If you inspect the document stored in MongoDB, you will find that the tags are stored as an array of Mongo object ids.

```
  "tags": [
             {
                "$oid": "56d228f9b5726a582219b5c7"
             },
             {
                "$oid": "56d22903b5726a582219b5c8"
             },
             {
                "$oid": "56d22914b5726a582219b5c9"
             }
     ]
```

4.5 Retrieving a Many-to-Many Relation

To fetch all the tags associated with a ticket, we can use the populate method.

```
keystone.list('Ticket').model.findOne().populate('tags').
exec(function(err, tick\
et) {
  //tags populated as ticket.tags
});
```

To fetch all the tickets that as associated with a particular tag, we can use the in operator since the tags field is an array.

```
keystone.list('Ticket').model.find().where('tags').in([tag.id]).
exec(function(er\
r, tickets) {
    // all the tickets that were tagged with tag identified by tag.id
});
```

4.6 Filtering Relations

KeystoneJS also allows us to filter a relationship field using the filters option. The filters option is an object of key/value pairs, in which the keys correspond to the fields of the related model to be filtered, and the values will either be literals or field names in the current model, the value of which will be used to filter the relationship.

In the example below, the createdBy field will only allow selection of a User whose isAdmin field is set to true.

```
Ticket.add({
   ...
    createdBy: { type: Types.Relationship, ref: 'User', index: true,
    filters: { \
isAdmin: true } },
   ...
});
```

We can also filter by the value of another field on the model. Do this by setting the value of the filter to the name of the field, prefixed by a colon (:). Let's assume you wanted to assign a sample ticket based on the category that was selected. The code below will filter tickets and provide only the ones where the category is the same as the one that is selected for the ticket being created.

```
Ticket.add({
   ...
    category: { type: Types.Select, options: 'Bug, Feature,
    Enhancement', defaul\
t: 'Bug' },
```

```
5       similarTicket: { type: Types.Relationship, ref: 'Ticket', filters: {
        categor\
6   y: ':category' } },
7       ...
8   });
```

You can also filter by the current model's _id field similar to the above example.

■ **Note** You can only set filters on one-to-many relationships (i.e., when the many option is NOT set to true).

4.7 Summary

In this chapter, we read about how KeystoneJS provides an easy way to define relationships and query-related data. Building complex database drived applications should now be easy! Onward!.

CHAPTER 5

Integrating Web Forms

Most modern-day web application developers spend a considerable amount of time implementing forms that the user can use to interact with the application models. For instance, in our IncTicket application, users will use forms to create tickets and update them. Developing the HTML for a form might be simple enough, but integrating it with the application back end might be a little tricky. In this chapter, we will look at how we can integrate forms within a KeystoneJS application.

5.1 Creating a New Ticket Form

Let us create a form where an authenticated user in our application can create a new ticket. Create a new template named newticket.swig to hold our web form, and place it in the `/templates/views/tick-ets` folder.

```
{% extends "../../layouts/default.swig" %}

{% block content %}

<div class="container">
          <div class="panel panel-primary">
      <!-- Default panel contents -->
      <div class="panel-heading">Create a New Ticket</div>
      <div class="panel-body">
           <form class="form-horizontal custom-form" action="/
           createticket" method\
="post">
                  <div class="form-group">
                      <div class="col-sm-2 label-column">
                          <label for="name-input-field" class="control-
                          label">Titl\
e </label>
                      </div>
                      <div class="col-sm-6 input-column {% if
                      validationErrors.tit\
le %}has-error{% endif %}">
```


M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_5

```
19                          <input type="text" name="title"
                            placeholder="Title of th\
20  e ticket" class="form-control" value="{{form.title}}" />
21                      </div>
22                  </div>
23                  <div class="form-group">
24                      <div class="col-sm-2 label-column">
25                          <label for="email-input-field" class="control-
                            label">Des\
26  cription </label>
27                      </div>
28                      <div class="col-sm-6 input-column" {% if
                        validationErrors.de\
29  scription %}has-error{% endif %}>
30                          <textarea name="description"
                            placeholder="Describe the i\
31  ssue" class="form-control">{{form.description}}</textarea>
32                      </div>
33                  </div>
34                  <div class="form-group">
35                      <div class="col-sm-2 label-column">
36                          <label for="pawssword-input-field"
                            class="control-label"\
37  >Priority </label>
38                      </div>
39                      <div class="col-sm-6 input-column">
40                          <select class="form-control" name="priority">
41                                  <option value="Low" {% if form.
                                    priority == 'Low'\
42  %} selected{% endif %}>Low</option>
43                                  <option value="Medium" {% if form.
                                    priority == 'M\
44  edium' %} selected{% endif %}>Medium</option>
45                                  <option value="High" {% if form.
                                    priority == 'Hig\
46  h' %} selected{% endif %}>High</option>
47                          </select>
48                      </div>
49                  </div>
50                  <div class="form-group">
51                      <div class="col-sm-2 label-column">
52                          <label for="repeat-pawssword-input-field"
                            class="control\
53  -label">Category </label>
54                      </div>
55                      <div class="col-sm-6 input-column">
56                          <select class="form-control" name="category">
```

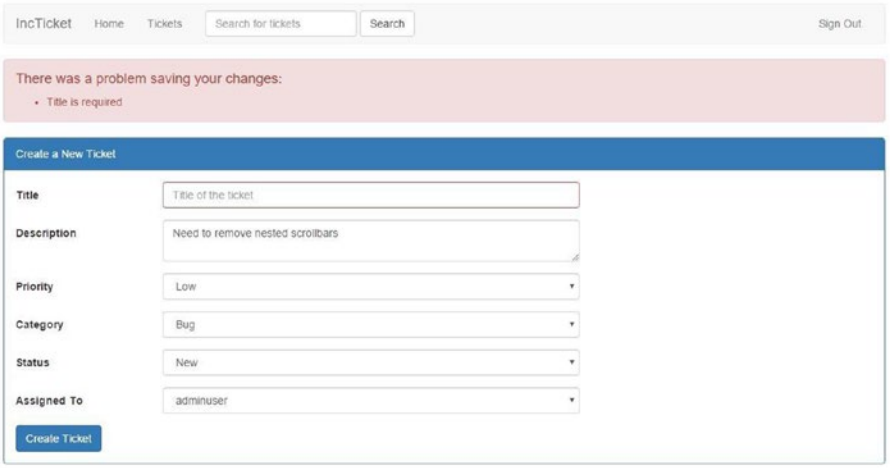
```
57                                  <option value="Bug"  {% if form.
                                    category == 'Bug\
58  ' %} selected{% endif %}>Bug</option>
59                                  <option value="Feature" {% if form.
                                    category == '\
60  Feature' %} selected{% endif %}>Feature</option>
61                                   <option value="Enhancement" {% if
                                     form.category \
62  == 'Enhancement' %} selected{% endif %}>Enhancement</option>
63                          </select>
64                      </div>
65                  </div>
66                  <div class="form-group">
67                      <div class="col-sm-2 label-column">
68                          <label for="repeat-pawssword-input-field"
                            class="control\
69  -label">Status </label>
70                      </div>
71                      <div class="col-sm-6 input-column">
72                          <select class="form-control" name="status">
73                                  <option value="New" selected>New</
                                    option>
74                                  <option value="In Progress">In
                                    Progress</option>
75                                  <option value="Open">Open</option>
76                                  <option value="On Hold">On Hold
                                    </option>
77                                  <option value="Declined">Declined
                                    </option>
78                                  <option value="Closed">Closed</option>
79                          </select>
80                      </div>
81                  </div>
82                  <div class="form-group">
83                      <div class="col-sm-2 label-column">
84                          <label for="repeat-pawssword-input-field"
                            class="control\
85  -label">Assigned To </label>
86                      </div>
87                      <div class="col-sm-6 input-column">
88                          <select class="form-control"
                            name="assignedTo">
89                          {% for assignUser in data.users %}
90                              <option value="{{assignUser.
                                id}}">{{assignUser.usern\
91  ame}}</option>
92                          {% endfor %}
```

```
                        </select>
                    </div>
                </div>
                <input type="submit" class="btn btn-primary submit-
                button" value\
  ="Create Ticket"/>
        </form>
      </div>
      </div>
  </div>
  {% endblock %}
```

The form is pretty straightforward. We have HTML form fields for each of the model fields on the Ticket model. Our view will perform all the input validations and will populate the validationErrors in the response, if any. We can use the snippet below to highlight any fields that failed validation by applying the bootstrap 'has-errors' css class.

```
{% if validationErrors.title %}has-error{% endif %}"
```

The view will also return flash messages if there are validation errors. The screenshot below is an example of input validation.



Add a route to the routes index file:

```
app.all('/createticket', middleware.requireUser, routes.views.tickets.
newticket);
```

As you see, we are taking advantage of the requireUser middleware to make sure that our user is first logged into the application before being able to create a ticket.

Next, create a file for our view named newticket.js, under /routes/views/tickets

```
var keystone = require('keystone'),
    Ticket = keystone.list('Ticket');

exports = module.exports = function(req, res) {

        var view = new keystone.View(req, res),
                locals = res.locals;

   locals.form = req.body;
       locals.data = {
             users: []
       };

       view.on('init', function(next) {

                var q = keystone.list('User').model.find().select('_id
                username')

               q.exec(function(err, results) {
                       locals.data.users = results;
                       next(err);
               });


         });

    view.on('post', function(next) {

        var newTicket = new Ticket.model(),
                data = req.body;

        data.createdBy = res.locals.user.id;

        newTicket.getUpdateHandler(req).process(data, {
                        flashErrors: true,
               }, function(err) {
            if (err) {
                    locals.validationErrors = err.errors;
                } else {
                    req.flash('success', 'Your ticket has been
                    created!');
                    return res.redirect('/tickets/' + newTicket.slug);
                }
                next();

```

```
            });

        });

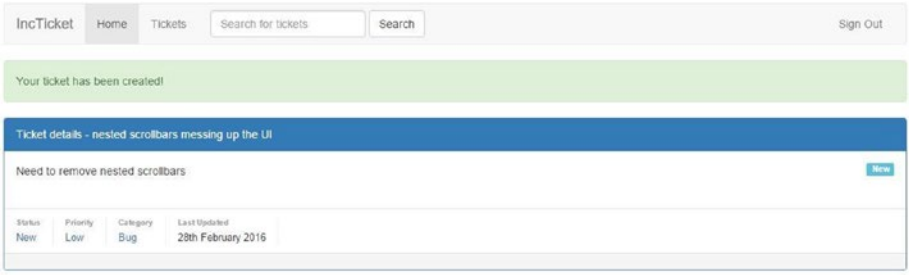
        // Render the view
        view.render('tickets/newticket');

};
```

On the view initialization, we attempt to retrieve a list of users. This is returned to the template during render to be used to show a drop-down list to select the user to whom the ticket can be assigned. As shown in the snippet below, we use a for loop to render the select control with the id of the user being the key and the username being the value.

```
<div class="form-group">
    <div class="col-sm-2 label-column">
        <label for="repeat-pawssword-input-field" class="control-
        label">Assigned\
To </label>
    </div>
    <div class="col-sm-6 input-column">
        <select class="form-control" name="assignedTo">
        {% for assignUser in data.users %}
            <option value="{{assignUser.id}}">{{assignUser.username}}
            </option>
        {% endfor %}
        </select>
    </div>
</div>
```

On form post, we create an instance of the Ticket model and gather the form post into the data variable. We set the createdBy property to the currently logged-in user. The form data is processed by the Ticket's update handler. If there are errors, we set the errors to the validationErrors variable; otherwise we redirect the user to the ticket details page as shown below.



We can use a similar form and view to be able to edit an existing ticket and update it using the update handler.

5.2 Summary

In this chapter, we were able to create some useful functionality for creating tickets. KeystoneJS's form validation makes it very convenient to implement complex forms in any application.

CHAPTER 6

Filtering Requests with Middleware

6.1 Introducing Middleware

Middleware is a piece of code that hooks into KeystoneJS's request/response processing cycle. Middleware is available globally throughout the application for altering input or output and is generally considered code that is not specific to any domain. Examples of middleware include authentication and authorization, caching, performance monitoring, and content compression. While all of these features are critical, none are specific to any one kind of project, and therefore it is a good idea to separate these out from the project's code in order to take advantage of them. In this chapter I'll introduce you to the middleware included in KeystoneJS and even show you how to write your own custom middleware solution.

Middleware consists of JavaScript methods that are responsible for performing a specific function. For example, KeystoneJS includes a flashMessages middleware function that retrieves flash messages from session before the view template is rendered. Middleware provides a convenient mechanism for filtering HTTP requests entering your application. We could define a middleware that makes sure that the user of your application is authenticated. If the user is not authenticated, the middleware will redirect the user to the login screen. However, if the user is authenticated, the middleware will allow the request to proceed further into the application. A logging middleware could be used to log all incoming requests to your application.

KeystoneJS's middleware resides in `/routes/middleware.js`

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_6

6.2 Introducing KeystoneJS's Default Middleware

Open your project's `/routes/middleware.js` file and you'll find three ready-made middleware solutions, including the following:

- **initLocals**: This middleware initializes standard view locals to include navlinks and the current user. View local is an object that contains response local variables scoped to the request and is therefore available to the view(s) rendered during that request / response cycle. This property is useful for exposing request-level information such as the request path name, authenticated user, user settings, and data.
- **flashMessages**: This middleware fetches and clears the flash messages before a view is rendered. Flash messages are an essential way of returning alerts/errors to users as they navigate from one page to another within your application.
- **requireUser**: This middleware is used to confirm a user is signed into the application. If not, the user is redirected to the login page. Further in the chapter, we will see how to write our own middleware that also checks if the user is an admin.

If you look further into the core libraries for KeystoneJS, there are additional middleware implemented ones such as the following:

- **cors**: Cross-Origin Resource Sharing is a mechanism for allowing clients to interact with APIs that are hosted on a different domain. CORS works by requiring the server to include a specific set of headers that allow a browser to determine if and when cross-domain requests should be allowed. The cors middleware makes it easy to add those headers.
- **initAPI**: This middleware adds apiResponse and apiError properties to the response. These are useful when building APIs using KeystoneJS. We will read more about this in the Restful API for Mobile and SPA applications chapter.

6.3 Defining Middleware in KeystoneJS

Creating a new middleware function is very easy in KeystoneJS. You would need to export a function within the middleware.js file and define the implementation of that function. The middleware file is already included into KeystoneJS's routing mechanism there by simplifying the creation and use of a middleware function. Let us define a middleware that verifies if the user is an administrator. This middleware can be used to protect parts of the application like the admin back end, which need a user to be logged in and be designated as an administrator.

```
exports.isAdmin = function(req, res, next) {

        if (!req.user) {
               req.flash('error', 'Please sign in to access this
               page.');
               res.redirect('/signin');
        } else if(!req.user.isAdmin)
        {
            req.flash('error', 'User does not belong to admin group');
            res.redirect('/');
        }
        else
        {
                next();
        }

}
```

KeystoneJS expects middleware functions to accept the following arguments:

- req - an express request object
- res - an express response object
- next - the method to call when the middleware has finished running (including any internal callbacks) to pass control to the next stage within the application

The isAdmin middleware function returns a HTTP redirect to the client if the user is not authenticated or if the user is not an administrator. If the user satisfies both these criteria, the request will be passed further into the application. To pass the request to the subsequent step in the application flow, simply call the next() callback.

6.4 Assigning Middleware to Routes

To assign middleware to specific routes, you should update the /routes/index.js file to include the HTTP verbs that the middleware should deal with, along with the route signature.

```
exports = module.exports = function(app) {

        app.all('/admin*', middleware.isAdmin);
        app.get('/admin/viewcontent', routes.views.viewcontent);
        app.post('/admin/editcontent', routes.views.editcontent);

}
```

To protect the admin page using your new middleware, update the application routes to use the isAdmin method. When the app receives any kind of HTTP request (GET, POST etc) with any route signature that matches /admin followed by optional URL segments, the middleware kicks in and passes the request through the isAdmin method. Once the isAdmin middleware returns a valid call, the application will continue to serve the corresponding view and template.

It is very easy to protect all routes that fall under a certain parent route segment. This is achieved using the asterisk following the parent route segment. The asterisk is part of the regular expression-based routing engine that indicates that the isAdmin method will be called prior to the request flowing to viewcontent or editcontent or even just the /admin method. The ordering of routes is important to have the necessary effect of middleware.

Use an array to assign multiple middleware to a route. Suppose you had another middleware method named logRequests that logs incoming requests, then we can use multiple middleware for a single route as shown below:

```
1   app.get('/admin/*', [middleware.requireUser, middleware.logRequests]);
```

6.5 Summary

Think of middleware as a series of “layers” that requires your application to pass through before they are completed. Each layer can examine the request and even reject it entirely.

CHAPTER 7

Authenticating and Managing Your Users

User authentication is critical to any modern web application. Various forms of user authentication mechanisms are available today. Traditionally, users log in to an application using a username and password. With the rise of social networking, single sign-on using an OAuth provider such as Facebook or Twitter has become a popular authentication method. In this chapter you will learn to implement username and password-based authentication for your application.

Out of the box, KeystoneJS provides the capability for users to be created at the admin back end only. An admin can log in to the administration panel and add a new user and set the user's password. There is no prebuilt functionality for a user to create an account via the front end. Let us learn how KeystoneJS makes it very simple to allow users to create an account, log in, and view tickets that were created by them. To start, let us look at the configuration options and routes we would need and also changes to the front end.

7.1 Configuring Authentication Options

To use KeystoneJS authentication system, we need to make sure it's set up correctly. The KeystoneJS JavaScript file contains all the authentication settings. In most cases the default settings are good to be used; however let us review the settings so you have a clear understanding of what is available.

- **session**: Set this option to true if you want your application to support session management. Loading sessions incurs a small overhead, so if your application doesn't need sessions you can turn this off.
- **auth**: This option indicates whether to enable built-in auth for Keystone's Admin UI, or a custom function to use to authenticate users. If this is set to false (or not defined), Keystone's Admin UI will be open to the public; hence it is good to set it to **true**.

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_7

- **user model**: This setting tells KeystoneJS what model will be used to maintain the user information (email address, password). By default it's set to User referring to the models/user.js file.
- **cookie secret**: Use this option to specify the encryption key to use for your cookies. This is usually set to a long, random string.
- **session store**: By default, KeystoneJS will use the in-memory session store provided by Express, which should only be used in development because it does not scale past a single process, and leaks memory over time. Use this option to specify an alternate session store such as MongoDB or Redis for persistent sessions. To use our MongoDB, set this option to **mongo**. KeystoneJS will not bundle any session store packages during installation. It is up to us to bring in any dependencies we need. To use MongoDB, install connect-mongo using NPM.

```
1   npm install connect-mongo --save
```

After enabling the session based authentication options, we should see an **app_sessions** collection in our MongoDB with documents like below to store session data.

```
1   {
2       "_id": "892GDPMz3DdiK8Jb7vWJoOw1JDSrVILt",
3       "session": "{\"cookie\":{\"originalMaxAge\":null,\"expires\":null,\
        "httpOnly\
4   \":true,\"path\":\"/\"},\"flash\":{}}",
5       "expires": {
6           "$date": "2016-03-07T22:35:10.698Z"
7       }
8   }
```

Authentication routes

We need to define routes and views related to user authentication so the user can click either a register link to create an account or a login link to be able to access their tickets. Start by making some amends to the master layout at `/templates/layouts/default.swig` to display the login link to guests and the logout link to users who are logged in. To check whether a visitor is logged in, we use the user object from our middleware. We will remove the existing sign-in link that points to the admin back end.

If you recall from Chapter 6, KeystoneJS's default middleware exposes a method named initLocals. This middleware function sets the current user object in the response that is returned to the template.

```
1   locals.user = req.user;
```

Update the navbar to include the below code:

```
{% if user %}
        <li><a href="/signout">Sign Out</a></li>
{% else %}
        <li><a href="/join">Create an Account</a></li>
        <li><a href="/signin">Sign In</a></li>
{% endif %}
```

In the above piece of code, we first check to see if the user object exists in the data that is returned to the template from a view. This object will be undefined for a guest or a user that has not logged into the site. If the user has logged in, we render a sign-out link. However, if the user has not logged in then we render both the links for creating an account and a sign-in link.

After including the markup for login and sign-in links, the navbar should look like below:

Figure 7-1. *Sign-in Links*

The routes to display the login form and registration form could not be easier. Create a folder named auth within the /routes folder to organize all your authentication-related views. We will also look at implementing password reset in case users forget their password.

```
app.all('/join', routes.views.auth.join);
app.all('/signin', routes.views.auth.signin);
app.get('/signout', routes.views.auth.signout);
app.all('/forgotpassword', routes.views.auth.forgotpassword);
app.all('/resetpassword/:key', routes.views.auth.resetpassword);
```

As you see, we have used the **app.all** method to specify the routes for join and sign-in. This makes it very convenient to display the registration form when the visitor does a GET on that route and then POST all the form parameters to the same route and handle the form post within the same view. This helps keep related code in a single view rather than having to define separate views for GET and POST, making it very easy for us to maintain our code base.

7.2 The User Model

To begin, you need to define the model that is going to be used to represent the users of your application. KeystoneJS already provides you with sensible defaults in a user model located at `/models/User.js`, where you can change the model or collection that is used to store your user accounts.

Let us to add a username property to the user model so users can set a unique username. Add the autokey list option to the user model so that a slug is generated from the username. Also, add the resetPasswordKey property and set the **hidden** option to true. This property will be used to allow the user to reset the password of their profile. The hidden option hides this field from being displayed on the admin UI.

```
var keystone = require('keystone'),
        Types = keystone.Field.Types;

var User = new keystone.List('User', {
        autokey: { path: 'slug', from: 'username', unique: true }
});

User.add({
    username: { type: String, required: true, unique: true, index:
    true, default\
:'' },
        name: { type: Types.Name, required: true, index: true },
        email: { type: Types.Email, initial: true, required: true,
        index: true },
        password: { type: Types.Password, initial: true, required: true
},
        resetPasswordKey: { type: String, hidden: true }
}, 'Permissions', {
        isAdmin: { type: Boolean, label: 'Can access Keystone', index:
        true }
});

// Provide access to Keystone
User.schema.virtual('canAccessKeystone').get(function() {
        return this.isAdmin;
});

 User.schema.virtual('url').get(function() {
    return '/users'+this.slug;
});

User.defaultColumns = 'name, email, isAdmin';
User.register();
```

Also, we should add a virtual property named URL to the user model, which can be used to show the user profile. The URL will be formed using the slug from the user's username prefixed with /**users** URL segment.

■ **Note** The user model must have a canAccessKeystone property that says whether a user can access Keystone's Admin UI or not.

7.3 Registering Users

Let us take a look at the template that can be used to allow a user to register at your site. The template will contain a form that will accept a user's username, first and last name, email address, and password. Create a template named join.swig and place it within /templates/views/auth directory.

```
{% extends "../../layouts/default.swig" %}

{% block content %}
<div class="container">
    <div class="panel panel-primary">
    <div class="panel-heading">Create An Account At IncTicket</div>
    <div class="panel-body">
     <div class="col-md-6">
    <form action="/join" method="post" class="form-horizontal">
      <fieldset>
      <div class="form-group required">
          <label class="col-md-4 control-label">User Name*</label>
          <div class="col-md-8">
            <input class="form-control" id="username" placeholder="Pick
            a user n\
ame" name="username" type="text" value="{{form.username}}">
          </div>
        </div>
                <div class="form-group required">
          <label class="col-md-4 control-label">First Name*</label>
          <div class="col-md-8">
            <input class="form-control" id="firstname"
            placeholder="First name" \
name="firstname" type="text" value="{{form.firstname}}">
          </div>
        </div>
                 <div class="form-group required">
           <label class="col-md-4 control-label">Last Name*</label>
           <div class="col-md-8">
             <input class="form-control" id="lastname"
             placeholder="Last name" na\
me="lastname" type="text" value="{{form.lastname}}">
          </div>
        </div>
                 <div class="form-group required">
           <label class="col-md-4 control-label">Email Address*</label>
           <div class="col-md-8">
             <input class="form-control" id="email" placeholder="Email
             address" n\
ame="email" type="email" value="{{form.email}}">
```

```
37            </div>
38          </div>
39                   <div class="form-group required">
40            <label class="col-md-4 control-label">Password*</label>
41            <div class="col-md-8">
42              <input class="form-control" id="password" name="password"
              placeholde\
43  r="password" type="password">
44            </div>
45          </div>
46                  <div class="form-group">
47            <label class="col-md-4 control-label"></label>
48            <div class="col-md-8">
49
50              <div style="clear:both"></div>
51                  <button class="btn btn-primary" type="submit">Join</
                    button>
52                      </div>
53         </div>
54       </fieldset>
55     </form>
56     </div>
57  </div>
58   </div>
59    </div>
60  {% endblock %}
```

The rendered markup will look like that below:

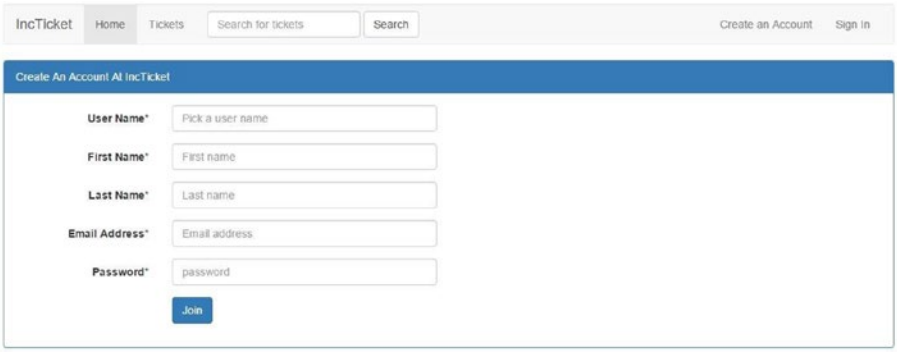


Figure 7-2. *Sign-Up Screen*

In the form, we have defined input fields for capturing the necessary data from the user. The form will POST to the `/join url`. If there are errors during user authentication such as an invalid email or password, we display those errors using the FlashMessages. renderMessages method that is offered by KeystoneJS.

■ **Note** The FlashMessages.renderMessages code is in our Default.swig layout file. This will help show messages on every page that extend the layout file.

Create the view named join.js under /routes/views/auth directory with the following code:

```
var keystone = require('keystone'),
        async = require('async');

exports = module.exports = function(req, res) {

        if (req.user) {
                return res.redirect('/tickets');
        }

        var view = new keystone.View(req, res),
                locals = res.locals;

        locals.section = 'createaccount';
        locals.form = req.body;

        view.on('post', function(next) {

                async.series([

                        function(cb) {

                                if (!req.body.username || !req.body.
                                firstname || !req.body.lastna
body.email || !req.body.password) {
                                req.flash('error', 'Please enter a
                                username, your name, e
');
                                return cb(true);
                                }

                                return cb();

                        },

```

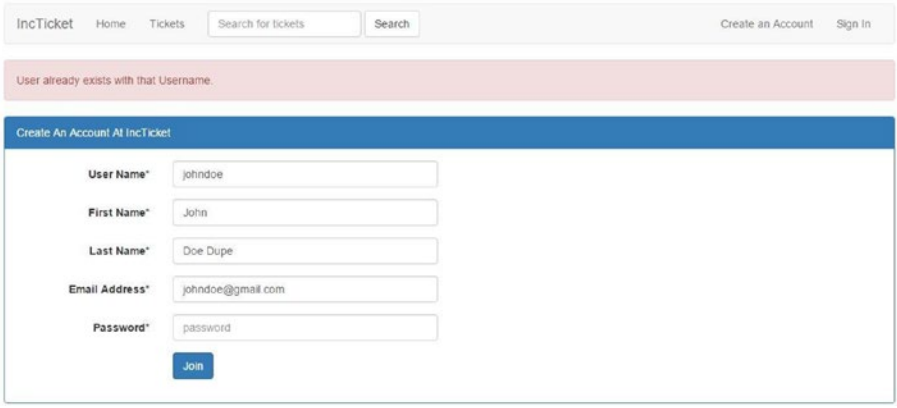
```
33              function(cb) {
34
35                                      keystone.list('User').model.findOne({
                                        username: req.body.username
36  n(err, user) {
37
38                                              if (err || user) {
39                                                      req.flash('error',
                                                        'User already exists
                                                        with that
40                                                      return cb(true);
41                                              }
42
43                                              return cb();
44
45                                      });
46
47                              },
48
49                              function(cb) {
50
51                                      keystone.list('User').model.findOne
                                        ({ email: req.body.email }, fu
52  user) {
53
54                                              if (err || user) {
55                                                      req.flash('error',
                                                        'User already
                                                        exists with that
56                                                      return cb(true);
57                                              }
58
59                                              return cb();
60
61                                      });
62
63                              },
64
65                              function(cb) {
66
67                                      var userData = {
68                          username: req.body.username,
69                                      name: {
70                                               first: req.body.
                                                 firstname,
71                                               last: req.body.
                                                 lastname,
72                                      },
```

```
73                                          email: req.body.email,
74                                          password: req.body.password
75                                  };
76
77                                  var User = keystone.list('User').
                                    model,
78                                          newUser = new
                                            User(userData);
79
80                                  newUser.save(function(err) {
81                                          return cb(err);
82                                  });
83
84                          }
85
86                  ], function(err){
87
88                          if (err) return next();
89
90                          var onSuccess = function() {
91                                          res.redirect('/tickets');
92                          }
93
94                          var onFail = function(e) {
95                                  req.flash('error', 'There was a problem
                                    signing you up, please tr
96                                  return next();
97                          }
98
99                          keystone.session.signin({ email: req.body.
                            email, password: req.body.passw
100 }, req, res, onSuccess, onFail);
101
102                 });
103
104         });
105
106         view.render('auth/join');
107
108 }
```

In the code above, we begin by checking if the user has already signed in. If yes, we then redirect the user to the /tickets route. We assign the request body to a local variable named form (`req.body`). This is useful for us to repopulate the form if there is an error such as a duplicate username or a duplicate email.

To test this, submit the form with an existing username. In the below screenshot, we submit the form twice with the username johndoe. The application indicates that a duplicate username has been detected; however notice that the other form fields such as

first name, last name, and email address have been populated with the values that were submitted. This provides for a good user experience on the sign-up page.



To set the values back to the form, we populate the locals.form object with the request body in the view. During the rendering of the template, we set the form input value from the locals object. At the first render, the value will be empty. However when a post back to the server is performed, then the value will contain the user input.

```
1    <input class="form-control" id="firstname" placeholder="First name"
     name="first\
2    name" type="text" value="{{form.firstname}}">
```

If you notice, we use the `async.series()` method to perform a series of checks during sign-up. First, we check if all the necessary form variables have been filled. Next, check if the user has input a duplicate username followed by a check for a duplicate email address. If all these conditions pass, we create a new user object from the user model and save it, followed by a call to the `keystone.session.signin` method to sign in the user.

The async.series method takes an array of functions and executes each one in order, not starting on one until the previous has finished. The async object automatically passes each function a callback argument. Once we manually call the callback, async knows that it's safe to move on to the next function.

7.4 User Login

To allow users to login to our application we will first need a template with a form to enter our login credentials. Create a template named signin.swig and place it within /templates/views/auth directory.

```
1   {% extends "../../layouts/default.swig" %}
2
3   {% block content %}
4
```

```
<div class="container">
    <div class="panel panel-primary">
    <!-- Default panel contents -->
    <div class="panel-heading">Login to IncTicket</div>
    <div class="panel-body">
    <div class="col-md-4">
    <form role="form" action="/signin" method="post">
    <div class="form-group">
        <label for="sender-email" class="control-label">Email
        address:</label>
        <div class="input-icon">
        <input class="form-control email" id="signin-email"
        placeholder="you@mai\
l.com" name="email" type="email" value="">
        </div>
    </div>
    <div class="form-group">
        <label for="user-pass" class="control-label">Password:</label>
        <div class="input-icon">
        <input type="password" class="form-control"
        placeholder="Password" name=\
"password" id="password">
        </div>
    </div>
    <div class="form-group">
        <input type="submit" class="btn btn-primary " value="Login">
    </div>
    </form>
    </div>
    </div>
    </div>
</div>

{% endblock %}
```

The markup above for our login form is pretty straightforward. We have defined input fields for the user's email address and the password. The form will POST to the `/signin url`. If there are errors during user authentication such as invalid email or password, we display those errors using the FlashMessages.renderMessages method that is offered by KeystoneJS. Flash messages are meant to be used only once. They are stored in the session and exist only between one request and response cycle. They are purged from the session automatically as soon as you retrieve them. Flash messages allow you to display once-off status messages to users, for example, form validation errors, success messages, etc. We have included the following piece of code in our layout file - /templates/layouts/Default.swig to render the flash messages.

```
{{ FlashMessages.renderMessages(messages) }}
```

Create the view named signin.js under /routes/views/auth directory with the following code:

```
var keystone = require('keystone'),
        async = require('async');

exports = module.exports = function(req, res) {

        if (req.user) {
                return res.redirect('/mytickets');
        }

        var view = new keystone.View(req, res),
                locals = res.locals;

        locals.section = 'signin';
         view.on('post', function(next) {

                if (!req.body.email || !req.body.password) {
                        req.flash('error', 'Please enter your email and
                        password.');
                        return next();
                }

                var onSuccess = function() {
                        res.redirect('/mytickets');
                }

                var onFail = function() {
                        req.flash('error', 'Input credentials were
                        incorrect, please try again.')
                        return next();
                }

                keystone.session.signin({ email: req.body.email,
                password: req.body.password }\
, req, res, onSuccess, onFail);

        });

        view.render('auth/signin');

}
```

In the code for the view, we begin by checking if the user has already logged in. If they have logged in, we redirect the user to the /mytickets route to display a list of tickets created by the user. If the user has not logged in and has submitted the login form, we will process the login request using the **view.on** method. This handy method allows us to

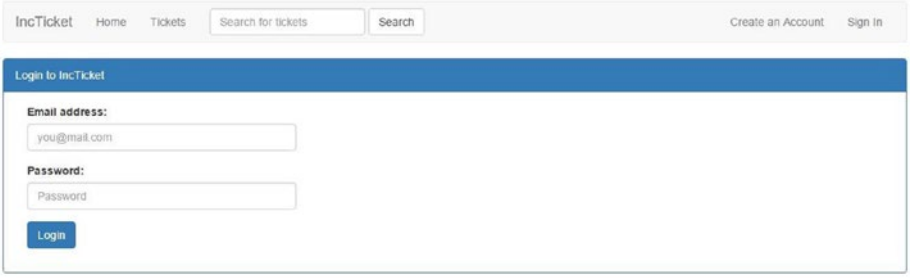
execute code depending on information in request object or on the type of HTTP VERB that was used. Since we POST the form to the view, we can process the login request using view.on(`'post, function..)`. To validate the form contents, we check if the user has provided an email address and a password. If either one is empty, we set a flash error indicating the missing data and return the callback.

When the form data has been provided, we specify a couple of functions that indicate what has to be done when the user has successfully authenticated and when the authentication fails. These functions (onSuccess, onFail) are passed to the **keystone.session.signin** method. This handy method will first check to see if the email that was provided conforms to an email address regex pattern, then it will query our MongoDB database to see if a user exists with the specified email address and compares the password. If the user has provided valid credentials, then the function will regenerate a new session and sets the below objects in the request. To complete the login, the onSuccess method is called next.

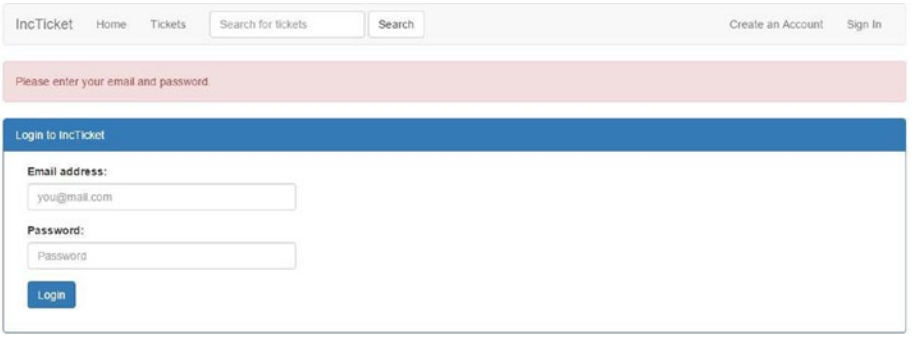
```
1   req.user = user;
2   req.session.userId = user.id;
```

■ **Note** If the 'cookie signin' config object has been set, then KeystoneJS will set a cookie on successful login.

Navigate to `http://localhost:3000/signin` and you should see the template rendered as below.



To test the validation and flash message rendering, click the login button without filling any credentials. We should see the validation error rendered as below:



Upon entering of valid email address and password, we should see the browser being redirected to /mytickets route.

7.5 Logging Out a User

To sign out a user from your application, you just need to call the keystone.session.signout method. The sign-out operation will clear the user's cookies, set the request user object to null, and regenerate a new session. Upon completion of sign-out, the user will be redirected to the root route of the application.

Create a view named signout.js under /routes/views/auth directory with the following code:

```
var keystone = require('keystone');

exports = module.exports = function(req, res) {

        keystone.session.signout(req, res, function() {
                res.redirect('/');
        });
};
```

7.6 Password Recovery

It is very easy to implement password recovery functionality. When a user wants to reset the password for their profile, we will generate a random key that will be set on the resetPasswordKey field on the user object and send an email to the user with a link to reset the password. After the user clicks on the link and returns to our application, we will validate that the key is actually valid for a user and then accept the new password.

Create a view named forgotpassword.js under /routes/views/auth directory with the following code:

```
var keystone = require('keystone'),
        User = keystone.list('User');

exports = module.exports = function(req, res) {

        var view = new keystone.View(req, res);

        view.on('post', function(next) {

                if (!req.body.email) {
                        req.flash('error', "Please enter an email
                        address.");
                        return next();
                }

                User.model.findOne().where('email', req.body.email).
                exec(function(err, user) {
                        if (err) return next(err);
                        if (!user) {
                                req.flash('error', "Sorry, That email
                                address is not registered i
cation.");
                                return next();
                        }

            user.resetPasswordKey = keystone.utils.
            randomString([16,24]);
            user.save(function(err) {
                if (err) return next(err);
                new keystone.Email({'templateName': 'forgotpassword',
                'templateE\
xt': 'swig'}).send({
                    user: user,
                    link: '/resetpassword/' + user.resetPasswordKey,
                    subject: 'Reset your Password',
                    to: user.email,
                    from: {
                        name: 'IncTicket',
                        email: 'info@incticket.com'
                    }
                }, function(err) {
                    if (err) {
                        console.error(err);
```

```
                        req.flash('error', 'Error sending reset
                        password email!'\
);
                        next();
                    } else {
                        req.flash('success', 'We have emailed you a
                        link to rese\
t your password');
                        res.redirect('/signin');
                    }
                });
            });
                });

        });

        view.render('auth/forgotpassword');
}
```

In the code above, we first check if the user provided an email address to reset the password for. If the email belongs to a valid account then we generate a random string between 16 to 24 characters and save it on the user object. Next, we attempt to send an email to the user with the link to reset the password. KeystoneJS makes it really easy to send emails. The **keystone.Email** method takes a templatePath parameter string that must be a folder in the emails path, and must contain either 'templateName/email.templateExt' or 'templateName.templateExt'. The **send** method takes an object that will be combined with the template and rendered before sending the email via Mandrill.

To configure the emails path, set the following option in KeystoneJS Javascript file:

```
'emails': 'templates/emails',
```

To configure the Mandrill API key, set the following key in your .env file:

```
MANDRILL_API_KEY= xxxx
```

To create the email template, create a template named email.swig in /templates/emails/forgotpass-word directory with the following markup:

```
Hello {{user.name.first}},
        <p>You recently requested a link to reset your IncTicket
        password.</p>
        <p>Please set a new password by following the link below:
        <a href='{{host}}{{link}}'>Click Here</a>
        </p>
```

Create a template named forgotpassword.swig under /templates/views/auth directory with the following markup:

```
{% extends "../../layouts/default.swig" %}

{% block content %}

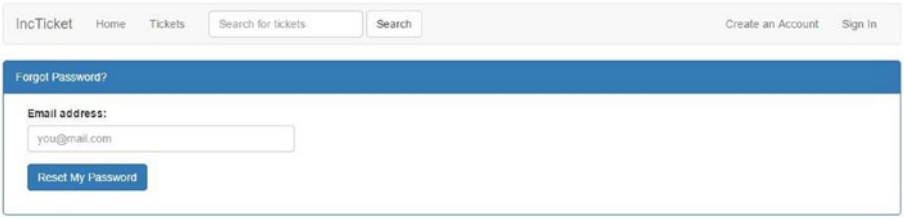
<div class="container">
    <div class="panel panel-primary">
    <!-- Default panel contents -->
    <div class="panel-heading">Forgot Password?</div>
    <div class="panel-body">

    <div class="col-md-4">
    <form role="form" action="/forgotpassword" method="post">
    <div class="form-group">
        <label for="email" class="control-label">Email address:</label>
        <div class="input-icon">
        <input class="form-control email" id="email" placeholder="you@
        mail.com" \
name="email" type="email" value="">
        </div>
    </div>

    <div class="form-group">
        <input type="submit" class="btn btn-primary " value="Reset My
        Password">\

    </div>
    </form>
    </div>
    </div>
    </div>
</div>
{% endblock %}
```

Let's update our sign-in page with a link to the forgotten password route. Navigate to http://localhost:3000/forgotpasswor and you should see the form rendered as below:



Let us fill out the form with a valid email id and inspect our MongoDB user's collection. We should see the resetPasswordKey generated and stored for that user document.

```
{
    "_id": {
        "$oid": "56cb93af9b6f78b82c19c8c6"
    },
    "slug": "mpanati",
    "email": "manikanta.panati@gmail.com",
    "password": "$2a$10$Araw8IGfRpKa9xaEF/
    I54OaCNvgE28MFEGOwqIbefMAtGs3px4I7K",
    "name": {
        "first": "Mani",
        "last": "Panati"
    },
    "username": "mpanati",
    "__v": 0,
    "resetPasswordKey": "3OkNJIydGImAb2vHZGrFd8n"
}
```

Create a template named resetpassword.swig under /templates/views/auth directory with the following markup:

```
{% extends "../../layouts/default.swig" %}

{% block content %}

<div class="container">
    <div class="panel panel-primary">
    <!-- Default panel contents -->
    <div class="panel-heading">Reset My Password</div>
    <div class="panel-body">

    <div class="col-md-4">
    <form role="form" action="/resetpassword" method="post">
    <input type='hidden' name='resetkey' value='{{key}}'/>
    <div class="form-group">
        <label for="password" class="control-label">Password:</label>
        <div class="input-icon">
        <input class="form-control email" id="password" name="password"
        type="pa\
ssword" value="">
        </div>
    </div>
    <div class="form-group">
        <label for="password_confirm" class="control-label">Confirm
        Password:</l\
```

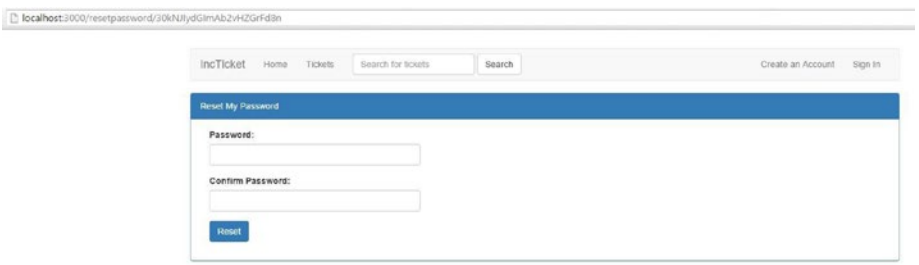
```
23  abel>
24          <div class="input-icon">
25          <input class="form-control" id="password_confirm"
            name="password_confirm\
26  " type="password" value="">
27          </div>
28      </div>
29
30      <div class="form-group">
31          <input type="submit" class="btn btn-primary " value="Reset">
32      </div>
33      </form>
34      </div>
35      </div>
36      </div>
37  </div>
38   {% endblock %}
39   {% endblock %}
```

Create a view named resetpassword.js under /routes/views/auth directory with the following code:

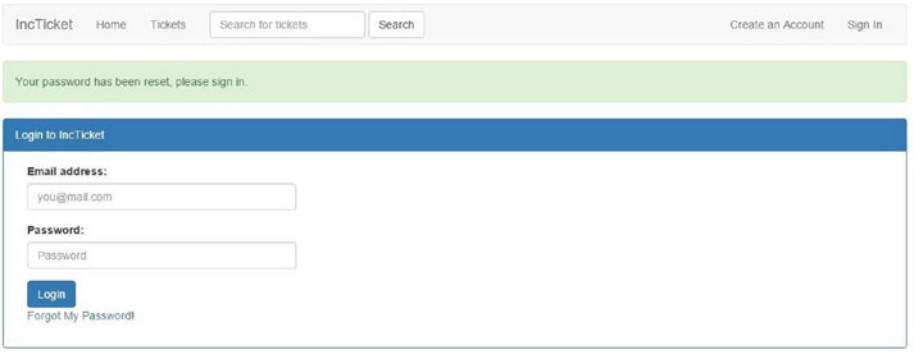
```
 1  var keystone = require('keystone'),
 2          User = keystone.list('User');
 3
 4  exports = module.exports = function(req, res) {
 5
 6          var view = new keystone.View(req, res),
 7                  locals = res.locals;
 8
 9          view.on('init', function(next) {
10
11                  User.model.findOne().where('resetPasswordKey', req.
                    params.key).exec(function(e\
12  rr, userFound) {
13                          if (err) return next(err);
14                          if (!userFound) {
15                                  req.flash('error', "Sorry, that reset
                                    password key isn't valid.")
16                                  return res.redirect('/forgotpassword');
17                          }
18                          locals.key = req.params.key;
19                          next();
20                  });
21
22        });
23
```

```
24        view.on('post', function(next) {
25
26                if (!req.body.password || !req.body.password_confirm) {
27                        req.flash('error', "Please enter, and confirm
                          your new password.");
28                        return res.redirect('/resetpassword/'+req.params.
                          key);
29                }
30
31                if (req.body.password != req.body.password_confirm) {
32                       req.flash('error', 'Please make sure both
                         passwords match.');
33                       return res.redirect('/resetpassword/'+req.params.
                         key);
34                }
35
36        User.model.findOne().where('resetPasswordKey', req.body.
          resetkey).exec(f\
37  unction(err, userFound) {
38                          if (err) return next(err);
39                          userFound.password = req.body.password;
40              userFound.resetPasswordKey = '';
41               userFound.save(function(err) {
42                   if (err) return next(err);
43                   req.flash('success', 'Your password has been reset,
                     please sign \
44  in.');
45                   res.redirect('/signin');
46               });
47                   });
48
49            });
50
51            view.render('auth/resetpassword');
52  };
```

Clicking on "Reset My Password" should bring you to the form to reset our password.



Populate the password field with a new password and a matching password in the confirm password field and click reset. The password for the user should be updated and the application should redirect to the sign-in page with a success flash message as shown below:



7.7 Retrieving the Authenticated User

To obtain a reference to the user's document associated with the authenticated user, use `{{user}}`. For instance, to retrieve the user's name you'll access user like this:

```
1   Hello {{ user.name.first }} {{ user.name.last }}!
```

We would, however, need to check if the user actually authenticated before accessing the user's details. The user object is populated only if the user has authenticated or else it would be undefined (for guests).

```
1   {% if user %}
2       Hello {{ user.name.first }} {{ user.name.last }}!
3   {% endif %}
```

7.8 Restricting Access to Authenticated Users

To restrict access to a route to be accessible only by authenticated users, we can rely on KeystoneJS's middleware. The middleware exposes a requireUser method that prevents people from accessing protected pages when they're not signed in. We can apply the middleware to the route as below:

```
1   app.all('/mytickets*', middleware.requireUser);
```

In the above example, we have specified that any subroutes under /mytickets such as /mytickets, /mytickets/new, /mytickets/edit/1 with any HTTP verb, would all need the user to be first logged in.

7.9 Securing Your Application

A good framework must possess the capability to provide security for the application being developed and KeystoneJS is no exception. KeystoneJS has capability built into it to prevent malicious attacks such as CSRF, IP range-based access control, and iframe protection headers. Let us look how we can take advantage of these features.

7.9.1 Cross-Site Request Forgery

Cross-site request forgery, commonly known as CSRF is a type of attack where a malicious website will send a request to another web application where the user is already authenticated. This enables the attacker to manipulate functionality in the target web application via the victim's authenticated browser. It is the responsibility of the affected web application to prevent such attacks, and neither the victim's browser nor the site hosting the CSRF can do anything to prevent it.

The most common method to prevent CSRF attacks is to append CSRF tokens to each request and associate them with the user's session (e.g., via cookies). Such tokens can be unique per session or per request. By including the token with each request, the developer can ensure that the request is valid and not coming from a source other than the user.

KeystoneJS's middleware makes the above-mentioned very simple to implement. In fact, it takes just three lines to implement CSRF protection automatically. Let us look at how to add CSRF protection to the reset password form we present to the user as an example.

The first step is to automatically generate a CSRF token and set it in the cookie. Use the KeystoneJS security middleware to do it on the GET route as below:

```
1   app.get('/resetpassword/:key', keystone.security.csrf.middleware.init,
    routes.vi\
2   ews.auth.resetpassword);
```

The next step is to set a hidden variable in the reset password form so that it can be posted back to the server along with the cookie behind the scenes. Add the following line to the template form:

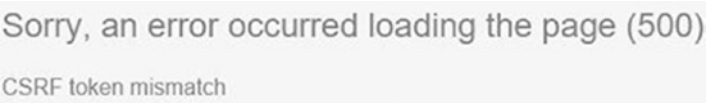
```
1   <input type='hidden' name='_csrf' value='{{csrf_token_value}}'/>
```

When keystone.security.csrf.middleware.init is called, it automatically adds a property named **csrf_token_value** to the response. KeystoneJS expects the CSRF token to be named **_csrf** when it is posted back to the server for validation.

The next step is to request KeystoneJS to validate the token before we start to process the form contents to reset the password. We can accomplish this with the validate security middleware method as shown below.

```
1   app.post('/resetpassword', keystone.security.csrf.middleware.validate,
    routes.vi\
2   ews.auth.resetpassword);
```

Adding the validate middleware method automatically checks for the validity of the incoming token. If the validation fails, KeystoneJS returns a response with a HTTP 500 status.



The CSRF token generation and validation can also be done manually, if we choose to. To generate the token, use the below methods and use in the template rendering:

```
var csrfTokenKey = keystone.security.csrf.TOKEN_KEY;
var csrfTokenValue = keystone.security.csrf.getToken(req, res);
```

To validate the token, use the validate methods:

```
if (keystone.security.csrf.validate(req)) {
    // CSRF is valid
    ...
  } else {
    // CSRF is not valid
    ...
}
```

7.9.2 Cross-Site Scripting

Cross-site scripting, commonly known as XSS, is a form of attack where the intention is to execute a piece of JavaScript on the user's browser when they are authenticated, in the hope of stealing cookies or session tokens. The most common way of performing such attacks is via unsanitized form inputs where the application accepts user input but does not sanitize it either before saving to the database or before rendering on the browser.

Avoiding such attacks is pretty easy. Swig, our rendering engine, will escape all output by default. There is no need to do any else! In case we want to render something that should not be escaped, then use the **autoescape** tag and set it to false.

```
{% autoescape false %}{{ somejs }}{% endautoescape %}
```

As a good practice, it is very important for us to sanitize the input from users. The Open Web Application Security Project (OWASP) has a list of rules that we can follow to mitigate XSS. More information is available at https://www.owasp.org/index.php/XSS(CSite_Scripting)_Prevention_Cheat_Sheet (https://www.owasp.org/index.php/XSS(Cross_Site_Scripting)_-Prevention_Cheat_Sheet)

7.9.3 Cookies

Sessions are persisted using an encrypted cookie storing the user's ID using the `cookie secret` option for encryption. All cookies are automatically signed and encrypted, which means that if they are tampered with, KeystoneJS will automatically discard them and make it very difficult to be read on the client side using JavaScript.

7.10 Summary

In this chapter, we have learned how to use KeystoneJS's built-in capability to add authentication features to our application quickly and also how to avoid common security problems.

CHAPTER 8

Searching Site Data

A good navigation menu makes a site very usable. Add a fast search to it and it makes the site much better. People use search engines like Google to find relevant information and expect to be able to find relevant information quickly and easily on our site as well. A good search experience can make users return to the website for future queries. Writing a good search system from scratch is a tough task but node, express, and keystoneJS make it relatively easy.

In this chapter we will look at a quick way to present search results from our keystoneJS application using Mongo DB's built-in full-text search capability.

8.1 MongoDB Full-Text Search

A good search system is indispensable to locating relevant online information on a website. To be able to provide precise search results in response to a query, the search system should not only cut down extra results but also use ranking and precision to organize the search results. A good search system will provide the most relevant results at the top. Beginning from version 2.4, MongoDB offered full-text search using Text indexes. MongoDB full-text search allows us to define a text index on any field in the document whose value is either a string or an array of string elements. When we create a text index on a field, MongoDB tokenizes, eliminates stop words, and stems the indexed field's content and adds it to the index. An important consideration is that a collection can contain only one text index. Hence it is very important for us to determine what we want fields to allow to be searched.

To create a text index, use the `db.collection.createIndex()` command against the MongoDB database.

8.2 Indexing a Single Field

To kick off things, let's create a text index on the title field of our ticket documents. Run the following query on the mongo shell:

```
1   db.tickets.createIndex({"title":"text"})
```


M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_8

To inspect the full-text index, let us retrieve a list of indexes on the tickets collection using db.tickets.getIndexes() method. We should see a response similar to the one below.

```
{
    "v" : 1,
    "key" : {
            "_fts" : "text",
            "_ftsx" : 1
    },
    "name" : "title_text",
    "ns" : "practicalkeystone.tickets",
    "weights" : {
            "title" : 1
    },
    "default_language" : "english",
    "language_override" : "language",
    "textIndexVersion" : 2
}
```

As you see, the name of the index is automatically set to title_text that indicates the indexed field and the text type of index. By default, the field is given a weight of 1. To test this newly created text index on the title field, we will search documents using the **$text** operator. Since we are interested in displaying the results according to decreasing level of relevance, let's get some statistics about how relevant the resulting documents are using the { $meta: "textScore" } expression. A higher textScore indicates a more relevant match. Use the sort command on textScore to order the documents. For the test, let us search for all the tickets that have the keyword 'image' in their subject field.

```
db.tickets.find({$text: {$search: "image"}}, {score: {$meta:
"textScore"}}).sort\
({score:{$meta:"textScore"}})
```

The above query returns the following documents:

```
{
    "_id": ObjectId("569cf4b858af0e88163ac180"),
    "slug": "images-not-loading",
    "title": "Images not loading",
    "updatedAt": ISODate("2016-01-18T14:20:40Z"),
    "createdAt": ISODate("2016-01-18T14:20:40Z"),
    "status": "New",
    "category": "Bug",
    "priority": "High",
    "__v": 0,
    "assignedTo": ObjectId("5692e414daa25ac42d792aed"),
    "createdBy": ObjectId("5692e414daa25ac42d792aed"),
    "description": "Some of the images do not load until the page is
    scrolled down\
```

```
14  . As you scroll down the page, you can see the images appear after a
    split secon\
15  d. Not sure what is causing this but it should definitely be fixed.
    \r\n",
16    "score": 0.75
17  }{
18    "_id": ObjectId("56c27c7d00647be83185f932"),
19    "slug": "resize-images-on-all-pages",
20    "assignedTo": ObjectId("5698e871a3497098256941b4"),
21    "createdBy": ObjectId("5692e414daa25ac42d792aed"),
22    "description": "Resize images on all pages so as to save bandwidth",
23    "updatedAt": ISODate("2016-02-16T01:33:49.867Z"),
24    "createdAt": ISODate("2016-02-16T01:33:49.867Z"),
25    "status": "New",
26    "category": "Feature",
27    "priority": "Medium",
28    "title": "Resize  images on all pages",
29    "__v": 0,
30    "score": 0.6666666666666666
31  }{
32    "_id": ObjectId("56c27c6100647be83185f931"),
33    "slug": "lazy-load-images-on-all-pages",
34    "assignedTo": ObjectId("5698e871a3497098256941b4"),
35    "createdBy": ObjectId("5692e414daa25ac42d792aed"),
36    "description": "Lazy load images on all pages so they load faster",
37    "updatedAt": ISODate("2016-02-16T01:33:21.110Z"),
38    "createdAt": ISODate("2016-02-16T01:33:21.110Z"),
39    "status": "New",
40    "category": "Feature",
41    "priority": "Medium",
42    "title": "Lazy load images on all pages",
43    "__v": 0,
44    "score": 0.625
45  }
```

As you can see, the first document has a score of 0.75 (since the keyword image appears first in its title) as opposed to the second document with a score of 0.66. The query has also sorted the returned documents in descending order of their score.

8.3 Indexing Multiple Fields/Wild Card Indexing

In the previous example, we created an index from the title field for our tickets. Most often, we might need to search on multiple fields on our documents. MongoDB allows indexing of multiple fields known as compound indexing or indexing of all fields in a document known as wild card indexing. We will look at an example of both.

Let us create a compound full-text index on the title and description fields of our tickets. Since MongoDB allows only for one full-text index per collection, we would need to drop the one that was created during the previous example.

Drop the index using the command below:

```
1   db.tickets.dropIndex("title_text");
```

Next, use the command below to create the compound index:

```
1   db.tickets.createIndex({"title":"text","description":"text"})
```

If we retrieve the list of indexes on the tickets collection, we should see the new compound index:

```
 1   {
 2       "v" : 1,
 3       "key" : {
 4               "_fts" : "text",
 5               "_ftsx" : 1
 6       },
 7       "name" : "title_text_description_text",
 8       "ns" : "practicalkeystone.tickets",
 9       "weights" : {
10               "description" : 1,
11               "title" : 1
12       },
13       "default_language" : "english",
14       "language_override" : "language",
15       "textIndexVersion" : 2
16   }
```

The way to query a compound index remains the same as a single field index except that MongoDB will search on two fields to compute its relevance score. As you see, the weights for both the title and description are set to 1 by default. We can specify relative weights for the indexed fields by using the **weights** option while creating the index. For each indexed field in the document, MongoDB multiplies the number of matches by the weight and sums the results. Using this sum, MongoDB then calculates the score for the document.

```
1   db.tickets.createIndex({"title":"text","description":"text"},{"weights":
    { title\
2   : 3, description:1 }});
```

To index an entire document using the wild card indexing, use the command below after dropping any existing text indexes:

```
1   db.tickets.createIndex({"$**":"text"})
```

The command above will automatically index all text fields in the documents. It is important to keep in mind that wild card indexes can be slow sometimes if the data is very large. For this reason, it is important to plan full-text indexes for proper search experience.

■ **Note:** MongoDB's full-text search is not a complete replacement for search applications such as Elastic search, SOLR, etc. However, it can be effectively used for the majority of applications that are built. If you need fine-grained control and efficient binary file indexing then I suggest investigating full-blown search solutions.

8.4 Adding Search to KeystoneJS

We will implement a search bar on our site, which will query our MongoDB full-text index for relevant results. Each result will contain a title of the ticket that is linked to the indexed ticket, followed by a summary. The results are ranked with the most relevant at the top.

Start by adding the markup below, for the search bar, to the navbar in `/templates/layouts/de-fault.swig`:

```
1   <form class="navbar-form navbar-left" role="search" action="/search"
    method="GE\
2   T">
3      <div class="form-group">
4          <input type="text" class="form-control" id="keywords"
           name="keywords" pl\
5   aceholder="Search for tickets">
6       </div>
7       <button type="submit" class="btn btn-default">Search</button>
8   </form>
```

The header should show a search bar as shown below in Figure 8-1:

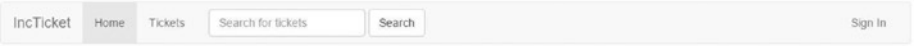


Figure 8-1. *Search Bar*

The search keywords will be sent to the /search route using a HTTP GET. Let us add this route to our `/routes/index.js` file. Using a GET is convenient in our case if we want to allow users to deep link to search results, that is,. share the URL for search results with other users.

```
1   app.get('/search', routes.views.tickets.search);
```

Next create the search view at /routes/views/tickets/search.js and add the below code:

```
var keystone = require('keystone');

exports = module.exports = function(req, res) {

        var view = new keystone.View(req, res),
                locals = res.locals;

        // Set locals
        locals.filters = {
                keywords: req.query.keywords
        };
        locals.data = {
                tickets: [],
                keywords: "",
        };

        // Load the current product
        view.on('init', function(next) {
                console.log('search keywords=' + locals.filters.
                keywords);
                locals.data.keywords = locals.filters.keywords;

                //search the full-text index
                keystone.list('Ticket').model.find(
                       { $text : { $search : locals.filters.keywords } },
                        { score : { $meta: "textScore" } }
                ).sort({ score : { $meta : 'textScore' } }).
                        limit(20).
                        exec(function(error, results) {
                                if(error) console.log(error);

                                 locals.data.tickets = results
                                next();
                });
        });

        // Render the view
        view.render('tickets/search');

};
```

In the code above, we retrieve the search keywords from the requested object and use the full-text index to search for tickets that match the keywords. To display the list of search results, create a template named **search.swig** in `/templates/views/tickets` and add the following code.

```
{% extends "../../layouts/default.swig" %}

{% block content %}

        <div class="container">
                <div class="panel panel-primary">
        <!-- Default panel contents -->
        <div class="panel-heading">Search Results</div>
        <div class="panel-body">
            <p>{{data.keywords | title}} search results (Limited to 20
            results)<\
/p>
        </div>

        <!-- Table -->
        <table class="table  table-striped">
            {% for ticket in data.tickets %}
               {% include 'ticket.swig' %}
            {% endfor %}
        </table>

        </div>

        </div>
{% endblock %}
```

Restart the application using node keystoneJS command and issue a search for the keyword 'image'. We should see search results similar to the ones below in Figure 8-2.

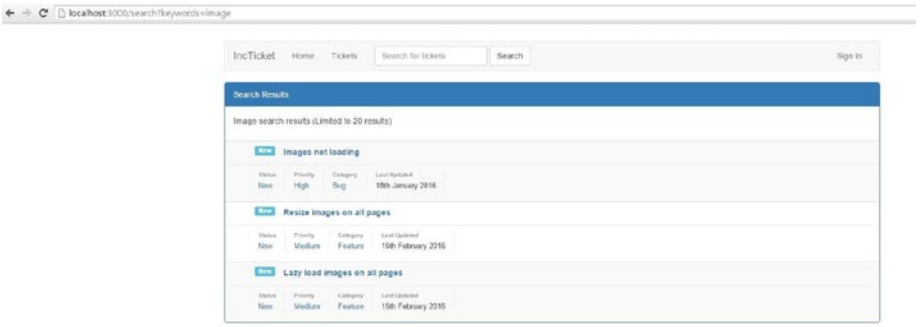


Figure 8-2. *List of Search Results*

8.5 Summary

This chapter has introduced the full-text search engine capability that MongoDB provides and how KeystoneJS easily enables a developer to add a search facility to a website. We have studied in detail at how to index data in our MongoDB tickets collection, provide weights to the fields that are important, and how to query and display the results. There are additional, powerful features such as text search for various languages that we can take advantage of.

CHAPTER 9

Restful API for Mobile and SPA applications

9.1 Introducing Restful APIs

REST stands for REpresentational State Transfer and is a pattern for developing APIs where we expose our business entities to be accessed and manipulated using HTTP requests. Each request can be one of these most often used protocols - GET, POST, PUT, PATCH, and DELETE. Each of these methods correspond to a particular, well-defined behavior that can be leveraged while building the API. Each entity is exposed under a logical grouping that allows for easier maintenance of the API, providing simplicity along with portability.

RESTful APIs are platform independent, and the fact that they are based on simple, well-established standards allows developers to achieve high performance, large scale, and secure communication when using REST APIs. The server is understood to be stateless in the RESTful paradigm, which means there is no notion of multiple requests acting within a single context on the server; this helps to scale out the services better. Content negotiation is also a part of the RESTful paradigm wherein the client can specify what format of data they are expecting back.

The RESTful paradigm should be treated as a useful set of rules to expose our application via endpoints. There may be many ways of implementing RESTful APIs and each developer tends to follow their own conventions as to what RESTful means to them. The simple bottom line should be that the APIs must be designed in such a manner that they serve the purpose of the application in a manner that you see fit and not get bogged down in following what a manual might specify.

9.2 What Are We Building?

Let's begin by outlining what our API is actually going to look like. To be able to work with the tickets that are stored in our MongoDB database, we are going to support a GET to an API called tickets that will return a list of tickets. We would like to include a parameter named page for paging the results. In addition, the API will allow users to get data about a specific ticket by ID. The API will also support getting a list of tickets for the current

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_9

logged-in user. The API should also allow users to Post a new ticket and allow them to delete an individual ticket. With these features in mind, let's see how we can expose a RESTful API for our Ticket Model. We will develop the following routes to allow for data retrieval and manipulation in a REST-based fashion:

- **GET /api/tickets**: This gets the list of tickets
- **GET /api/tickets/{id}**: This gets the ticket with ID {id}
- **POST /api/tickets**: This creates a new ticket
- **PUT /api/tickets/{id}**: This updates the ticket with ID {id}
- **DELETE /api/tickets/{id}**: This deletes the ticket with ID {id}

With an understanding of what APIs we intend to build, let us begin implementing them in our application. To start off, add the following routes that define the endpoints. The URL for a RESTful API is known as an endpoint.

```
1   // API
2   app.get('/api/tickets', keystone.middleware.api, routes.api.ticket.
    getTickets);
3   app.get('/api/tickets/:id', keystone.middleware.api, routes.api.ticket.
    getTicket\
4   ById);
5   app.post('/api/tickets', keystone.middleware.api, routes.api.ticket.
    createTicket\
6   );
7   app.put('/api/tickets/:id', keystone.middleware.api, routes.api.ticket.
    updateTic\
8   ketById);
9   app.delete('/api/tickets/:id', keystone.middleware.api, routes.api.
    ticket.delete\
10  TicketById);
```

The **keystone.middleware.api** parameter in the route adds the following shortcut methods for JSON API responses.

- res.apiResponse(data)
- res.apiError(key, err, msg, code)
- res.apiNotFound(err, msg)
- res.apiNotAllowed(err, msg)

The **apiReponse** method returns the response data in the JSON format. It can also automatically return data in JSONP if there is a callback specified in the request parameters.

The **apiError** method is a handy utility to return error messages to the client from our APIs. It returns an object with two keys – error and detail, which contain the exceptions that occurred. By default, it returns a HTTP status of 500 if the code parameter is not passed.

The **apiNotFound** method provides a quick way to raise a 404 (not found) exception from our APIs.

The **apiNotAllowed** method provides a quick way to raise a 403 (not allowed) HTTP response from our APIs.

Next, update the keystoneJS file with the code below to indicate to the application that we need to import views that are under the routes/api directory. By default, the code will only look for views under `/routes/views` directory.

```
// Import Route Controllers
var routes = {
        views: importRoutes('./views'),
    api: importRoutes('./api')
};
```

After defining the endpoints for the APIs, create the `ticket.js` view under **routes/api** directory to handle the responses.

ticket.js

```
var keystone = require('keystone'),
    Ticket = keystone.list('Ticket');

/**
 * List Tickets
 */
exports.getTickets = function(req, res) {
        Ticket.model.find(function(err, items) {

                if (err) return res.apiError('database error', err);

                res.apiResponse({
                        tickets: items
                });

        });
}

/**
 * Get Ticket by ID
 */
exports.getTicketById = function(req, res) {
        Ticket.model.findById(req.params.id).exec(function(err, item) {

                if (err) return res.apiError('database error', err);
                if (!item) return res.apiError('not found');

```

```
28                  res.apiResponse({
29                          ticket: item
30                  });
31
32          });
33  }
34
35
36  /**
37   * Create a Ticket
38   */
39  exports.createTicket = function(req, res) {
40
41          var item = new Ticket.model(),
42                  data = req.body;
43
44          item.getUpdateHandler(req).process(data, function(err) {
45
46                  if (err) return res.apiError('error', err);
47
48                  res.apiResponse({
49                          ticket: item
50                  });
51
52          });
53  }
54
55  /**
56   * Update Ticket by ID
57   */
58  exports.updateTicketById = function(req, res) {
59          Ticket.model.findById(req.params.id).exec(function(err, item) {
60
61                  if (err) return res.apiError('database error', err);
62                  if (!item) return res.apiError('not found');
63
64                  var data = req.body;
65
66                  item.getUpdateHandler(req).process(data, function(err) {
67
68                         if (err) return res.apiError('create error', err);
69
70                          res.apiResponse({
71                                  ticket: item
72                          });
73
```

```
74                  });
75
76          });
77  }
78
79  /**
80   * Delete Ticket by ID
81   */
82  exports.deleteTicketById = function(req, res) {
83          Ticket.model.findById(req.params.id).exec(function (err, item) {
84
85                  if (err) return res.apiError('database error', err);
86                  if (!item) return res.apiError('not found');
87
88                  item.remove(function (err) {
89                          if (err) return res.apiError('database error',
                            err);
90
91                          return res.apiResponse({
92                                  success: true
93                          });
94                  });
95
96          });
97  }
```

9.3 Tools for Working with Restful APIs

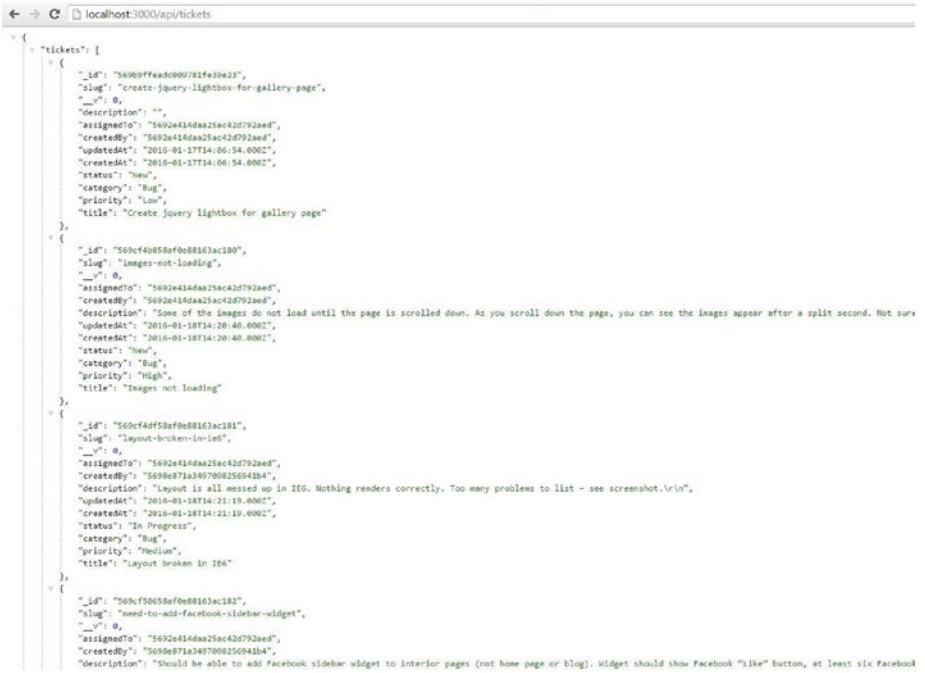
Now that we have implemented our APIs, how do we actually test whether they work? There are a couple of useful tools that make it really simple to issue RESTful requests (GET, POST, PUT, etc.) to our application API endpoints and look at the response. The first simple but often very useful tool is the JSON Formatter Chrome extension. By default, Chrome renders any JSON as plain text, making it hard to understand the structure of the response. The extension helps by converting any JSON responses in the browser into a well-formatted tree display with syntax highlighting and code folding. This can be useful for making a few test calls to APIs from within the browser for experimentation. The second tool is an application named POSTMAN REST client. POSTMAN makes it really easy to craft various kinds of requests to API endpoints. This application can be run as a Google Chrome extension within the browser or as a stand-alone application.

9.4 JSON Formatter Chrome Extension

To install the JSON Formatter Chrome extension, navigate to `https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?`[1] from Chrome and add the extension to the browser.

Here is the response from pointing to our tickets endpoint that returns JSON before JSON Formatter is installed:

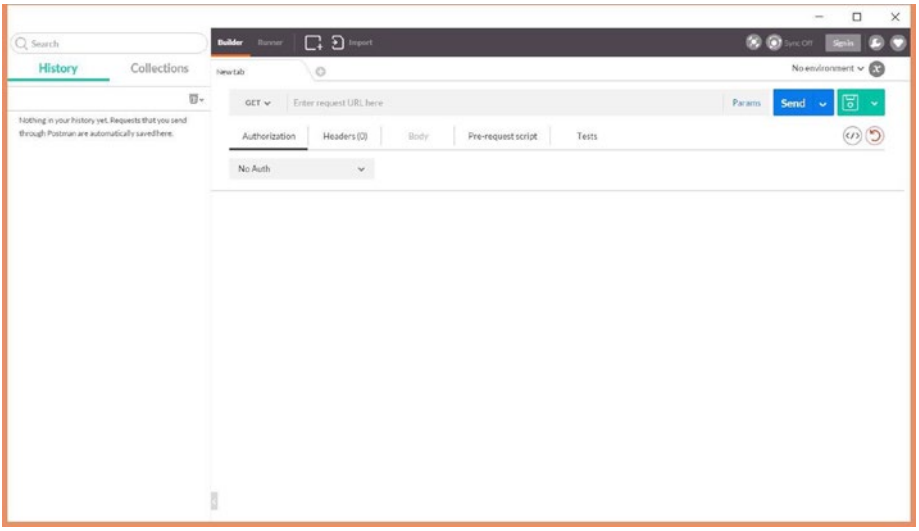
And here is that same URL after JSON Formatter has been installed:



[1] https://chrome.google.com/webstore/detail/json-formatter/bcjindcccaagfpapjjmafapmmgkkhgoa?

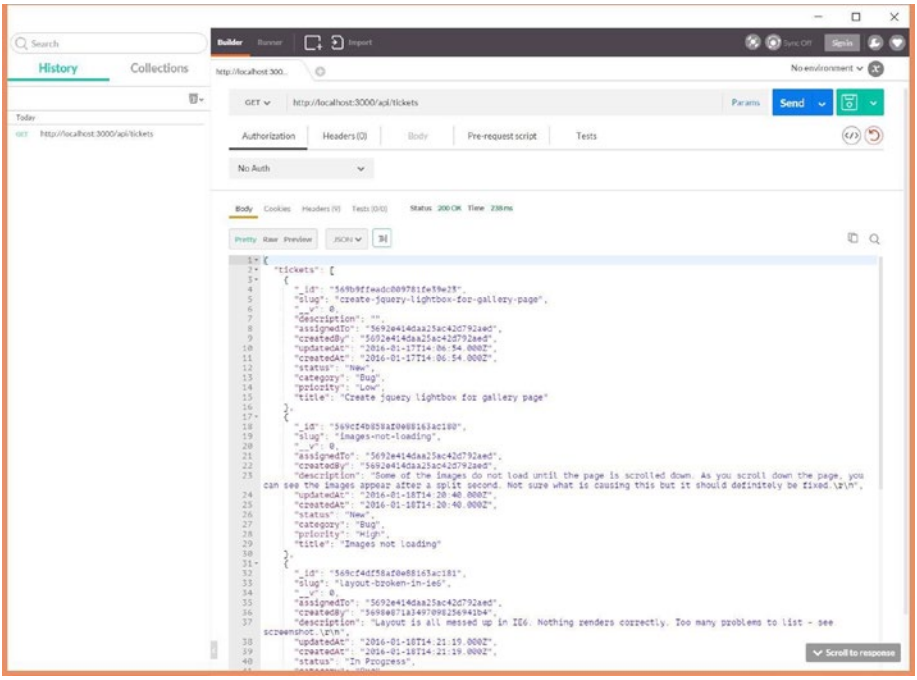
9.5 POSTMAN REST Client

Installing POSTMAN REST Client is simple. Visit `https://www.getpostman.com/apps`[2] and click on the 'Get the chrome app' button. This will take you to the Google Chrome Web Store, and provide an option to add the extension to Chrome. After installing the extension, launch it. You should be presented with a screen like the one below:



POSTMAN provides a very intuitive interface to make any kind of requests to any endpoint. To test our API call to get a list of tickets, enter the URL `http://localhost:3000/api/tickets` in the Enter request URL here field and leave the drop-down next to it as GET since we would like to make a HTTP GET request to our endpoint. Click on the Send button. This will make a HTTP GET similar to a browser and render the response at the bottom half of the application.

[2] `https://www.getpostman.com/apps`



A great feature of POSTMAN is that it formats the response from requests to make it readable and easy to work with. In our case, it has prettily printed the JSON list of tickets. It also provides nice code collapsing and search features for finding keywords in our response. The sidebar also maintains a history of requests that we have made that can be saved and replayed to various endpoints.

9.6 Serve Data with GET Requests

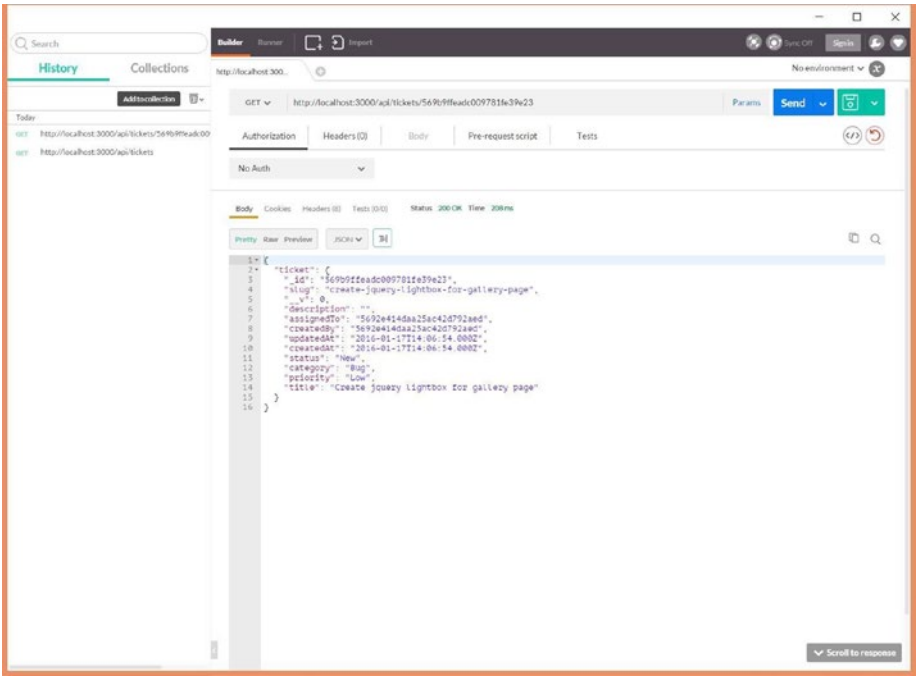
Exposing data from our application via a HTTP GET is fairly simple. For example, let us look at our implementation for getting a ticket by id. We use the Mongoose findById method to obtain a single document that matches the id passed via the request object. If there was an error querying the document, we use the api middleware method apiError to return a well-formatted error object with a 404 status. If the query returns a matching document, we send back JSON with the result document assigned to the key – 'ticket', using the apiResponse middleware method.

```
1   Ticket.model.findById(req.params.id).exec(function(err, item) {
2
3                   if (err) return res.apiError('database error', err);
4                   if (!item) return res.apiError('not found');
5
6                   res.apiResponse({
```

```
                            ticket: item
                    });

            });
```

The id for the document would be the _id field on the document that would be autogenerated by MongoDB for the tickets that we have created. Let us test the call to this endpoint using POSTMAN:



We can see that our API correctly returns a single instance of the ticket matching the id that was requested.

9.7 Update Data with POST and PUT

The most common HTTP verbs used to able to receive data on the server side are POST and PUT. POST is generally accepted as the verb to be used when we need to insert/save a new document. Let's take a look at endpoint code that accepts a POST request and inserts a ticket into our collection and returns the new document JSON.

```
var item = new Ticket.model(),
                data = req.body;

```

```
        item.getUpdateHandler(req).process(data, function(err) {

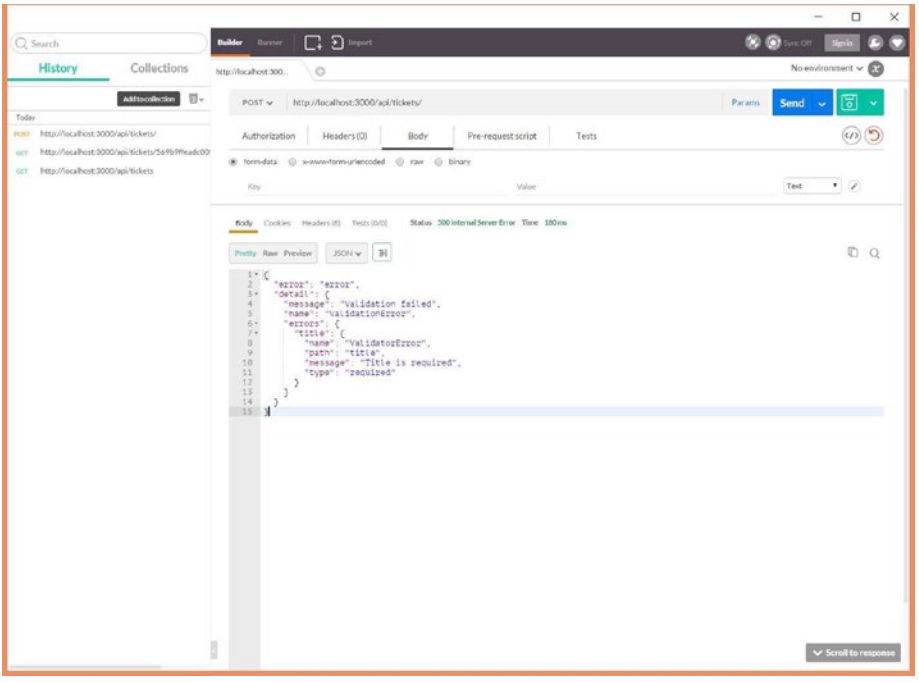
                if (err) return res.apiError('error', err);

                res.apiResponse({
                        ticket: item
                });

        });
```

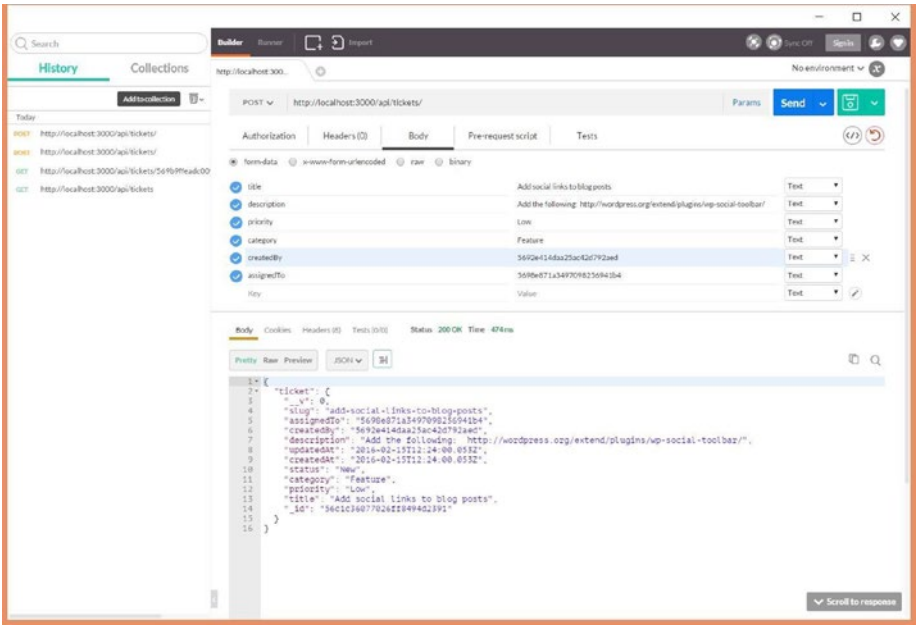
The getUpdateHandler method is added to the underlying Mongoose Model for our Ticket Key-stone.js List when we register the list during creation. The power of `KeystoneJS` lies in its ability to be able to do most of the heavy lifting for the developer and the update handler is one such example. The method single-handedly validates various criteria that have been specified during model definition. For example, we have specified that the title field is required. This will be properly validated automatically without the developer needing to check for such constraints programmatically at every instance. The method is capable of returning flash error messages (if using a view to display the response) or a collection of validation errors. If fields were marked as uneditable using the no-edit option during the creation of the model, then these fields will be skipped over automatically.

To check the validation in action, perform an empty POST to the `/api/tickets` endpoint.



The response is a well-formatted error message that clearly indicates the reason (Validation failed) and point of failure (Title is required). The HTTP status is also set to 500.

Next, provide the actual data for a new ticket using the form-data option in POSTMAN and specify each of the fields as a name/value pair with relevant data. The result indicates that our call was successful with a HTTP status 200 and returns the new ticket JSON data.



Reloading the main `/api/tickets` endpoint in a browser confirms that the new ticket has indeed been added to the tickets collection.

PUT requests to the endpoint work very similarly to how POST works except that the id of the resource we intend to modify is not passed through along with data we intend to change; instead it is passed via the URL. If you recall the route we defined, it included the id parameter. PUT requests are generally not used in scenarios where the id of the resource needs to be changed.

```
Ticket.model.findById(req.params.id).exec(function(err, item) {

                if (err) return res.apiError('database error', err);
                if (!item) return res.apiError('not found');

                var data = req.body;

                item.getUpdateHandler(req).process(data, function(err) {

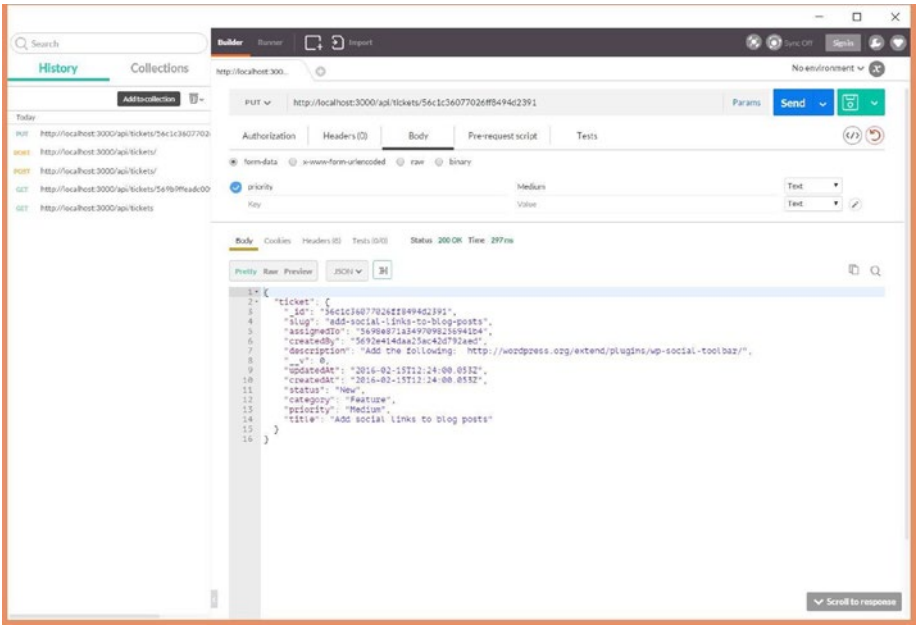
                        if (err) return res.apiError('create error',
                        err);

                        res.apiResponse({
                                ticket: item
                        });

                });

        });
```

Looking at the code, we see that we intend to first find the ticket that matches the id that was passed via the URL (req.params collection). If we do not find the document or encounter an error during the query, we return an error object to the client. If we do find the relevant document, we leverage the getUpdateHandler method to make updates to the document and save it. The update handler method will update only fields that we pass in and then leave the rest as is. The following screenshot shows a successful PUT request to update the ticket we created in the previous example. We have updated the ticket's priority field from Low to Medium. Note that only the priority field was updated and the rest of the ticket data remains the same as the original.



9.8 Removing Data with DELETE

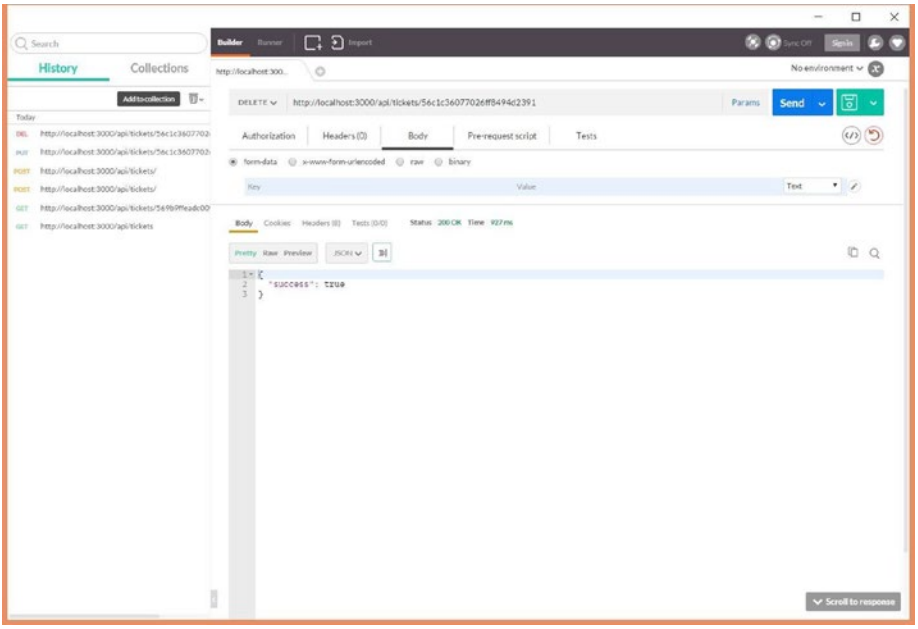
The last route we defined was to delete a ticket. The delete operation is pretty straightforward, and we should call our endpoint with a HTTP DELETE verb along with the id of the ticket we want to be removed.

```
1   Ticket.model.findById(req.params.id).exec(function (err, item) {
2
3                   if (err) return res.apiError('database error', err);
4                   if (!item) return res.apiError('not found');
5
6                   item.remove(function (err) {
7                           if (err) return res.apiError('database error',
                            err);
8
```

```
                        return res.apiResponse({
                                success: true
                        });
                });

        });
```

We use the Mongoose findById method to retrieve the document we intend to delete. If we do not find the document or encounter exceptions while removing the document, then we return an error to the client. If we do find the document, we remove it and return an object indicating the success status.



■ **Note** All of the endpoints defined in this chapter are extremely simple, that is, they do not enforce strict role validation and authentication rules. They may not be appropriate to be used in production environments as is. However, those rules can be easily added by leveraging the KeystoneJS middleware.

Another tool that can be used to interact with endpoints is cURL. cURL is a command-line tool for crafting requests and receiving responses from servers.

To install cURL on Mac/OSX, follow the steps below:

1. Download the cURL from http://curl.haxx.se/download.html
2. Open a Terminal and change directory to the folder where the file was downloaded
3. Extract the compressed file with below command:

```
$tar zxf curl-7.47.1.tar
```

4. Change directory to the extracted cURL directory

```
$cd curl-7.47.1
```

5. Run the make file , as follows and install cURL

```
$make && sudo make install
```

cURL on windows is distributed as a stand-alone executable from http://curl.haxx.se/download.html. Below are the cURL commands illustrating all the interactions we performed using POSTMAN: Get Ticket By Id:

```
curl -X GET -H "Cache-Control: no-cache"
"http://localhost:3000/api/tickets/569b9ffeadc009781fe39e23"
```

Create Ticket using POST:

```
curl -X POST -H "Cache-Control: no-cache"
-H "Content-Type: multipart/form-data; boundary=----
WebKitFormBoundary7MA4YWxkTr\
ZuOgW"
-F "title=Add social links to blog posts"
-F "description=Add the following:  http://wordpress.org/extend/
plugins/wp-socia\
l-toolbar/"
-F "priority=Low"
-F "category=Feature"
-F "createdBy=5692e414daa25ac42d792aed"
-F "assignedTo=5698e871a3497098256941b4"
"http://localhost:3000/api/tickets/"
```

Update Ticket using PUT:

```
curl -X PUT -H "Cache-Control: no-cache"
-H "Content-Type: multipart/form-data; boundary=----
WebKitFormBoundary7MA4YWxkTr\
ZuOgW"
-F "priority=Medium"
"http://localhost:3000/api/tickets/56c1c36077026ff8494d2391"
```

Delete Ticket By Id:

```
curl -X DELETE -H "Cache-Control: no-cache"
"http://localhost:3000/api/tickets/56c1c36077026ff8494d2391"
```

CHAPTER 10

Using Template Engines

By default, KeystoneJS advises the use of Jade as its template language, but you can configure KeystoneJS to support other rendering engines, such as Swig or Handlebars.

You can add a custom template engine by configuring the object passed to keystone.init method in /keystoneJS script file:

```
1   var swig = require('swig');
2
3   keystone.init({
4   /* .. */
5   'view engine': 'swig',
6   'custom engine': swig.renderFile
7   /* .. */
8   });
```

Make sure to use NPM to bring in the necessary template engine files before requiring it in your project.

10.1 Introducing the Swig Template Engine

The majority of pages in a website generally display data that is dynamic in nature. Dynamic content refers to HTML pages whose content is generated automatically and may be affected by conditional statements or the data source (such as a database table or document store). Dynamic data generally caters particularly to the context of the user who is logged in to the website. A web framework typically support one or more templating engines that makes it faster and easier to generate dynamic HTML content. KeystoneJS supports a variety of templating engines such as Jade, Handlebars, and EJS. Jade is the most common templating language used with KeystoneJS. However, we will look at how to configure and use swig, a simple, powerful, and extensible JavaScript templating engine.

Swig is robust, extendable, and customizable and supports object-oriented template inheritance. Swig's templating syntax and conventions are very intuitive and very similar to those followed by popular web frameworks such as Django, Jinja, and Twig.

To install swig, we can use the node package manager as below.

```
1   npm install swig --save
```


M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_10

Swig templates can either end with a .swig extension or a .html extension. You can control this in the keystone.init method in the keystone js file in the root of the application.

```
keystone.init({
    ...
    //use .html extension for template files
        'view engine': 'html',
    ...
});
```

10.2 Basic Syntax for Swig

The basic syntax for swig is really quite simple. Let's assume the following ticket JSON object is passed to a swig template via our view:

```
...
view.on('init', function(next) {

        var ticket = {
            "slug": "layout-broken-in-ie6",
            "title": "Layout broken in IE6",
            "status": "In Progress",
            "category": "Bug",
            "priority": "Medium",
            "description": "Layout is all messed up in IE6. Nothing
            renders corr\
            ectly. Too many problems to list. see screenshot.\r\n",
            tags : ['layouts, ie, render, image']
        }

        locals.data.ticket = ticket;

    });

        // Render the view
        view.render('displayticket');
};
```

There are two main types of markup in Swig:

- Output Tags
- Logic Tags

Find a list of all Swig tags at http://paularmstrong.github.io/swig/docs/tags/[1]

[1] http://paularmstrong.github.io/swig/docs/tags/

10.3 Output Tags

Output tags are surrounded by double curly brackets and will render content of a variable. If a variable is not defined an empty string will be output in its place. However, false values like null, false, 0 will be rendered as they are. To render variables in the template, we use the {{ }} in Swig. To render the title of the ticket, add the following markup to displayticket template:

```
<div class="panel panel-primary">
    <!-- Default panel contents -->
    <div class="panel-heading">Ticket details - {{data.ticket.title}}
    </div>
    ...
</div>
```

To output comments, use a curly brace followed by the hash sign. Comments are removed by the parser during rendering and will not be seen even if you do a view source on the rendered HTML page.

```
{#
    This is a comment.
#}
```

10.4 Logic Tags

Logic tags begin with a curly bracket and then percent sign. Logic tags control output through common conditional operators and helpers. To check if the description was entered and then render it, add the following markup to the template:

```
<div>
  {% if data.ticket.description %}
      Ticket description - {{data.ticket.description}}
  {% endif %}
</div>
```

Swig also has support for the if-else conditional check. For example, we can use the following to display alternate text if no description was provided.

```
<div>
 {% if data.ticket.description %}
      Ticket description - {{data.ticket.description}}
  {% else %}
      No ticket description available.
 {% endif %}
</div>
```

Adding Boolean conditionals are also pretty easy. Boolean conditionals like **and, or** can be used within logic tags. Below is an example illustrating the use of the 'or' conditionals:

```
<div>
 {% if data.ticket.category == 'Feature' or data.ticket.category ==
 'Enhanceme\
  nt' %}
      <label class='info'>{{data.ticket.category}}</label>
 {% else %}
      <label class='warning'>{{data.ticket.category}}</label>
 {% endif %}
</div>
```

Another way to express the 'or' conditional is to use the double pipe symbol - `|| <div> {% if data.ticket.category == 'Feature' || data.ticket.category == 'Enhancement' %} <label class='info' {% else %} <label class='warning'>{{data.ticket.category}}</label> {% endif %} </div>` Along the same lines, the 'and' condition can be specified using the '**and**' keyword or the double ampersand symbol - &&

We can also use built-in JavaScript functions within the conditional statements.

```
{% if data.ticket.description.indexOf("critical") > -1 %}
   This is a critical ticket!
{% endif %}
```

Swig also supports looping statements to loop over arrays and objects. To iterate over the tags array in the ticket object, add the following markup to the template:

```
<ul>
     {% for tag in data.ticket.tags %}
         <li> {{tag}} </li>
     {% endfor %}
 </ul>
```

The output on the page should be:

```
<ul>
      <li>layouts</li>
      <li>ie</li>
      <li>render</li>
      <li>image</li>
</ul>
```

Swig has a collection of very helpful loop control helpers. These provide additional information about the state of the loop in an iteration.

```
{% for tag in data.ticket.tags %}
    {% if loop.first %}<ul>{% endif %}
    <li>{{ loop.index }} - {{ tag }}</li>
    {% if loop.last %}</ul>{% endif %}
{% endfor %}
```

The output should be:

```
<ul>
     <li>1 - layouts</li>
     <li>2 - ie</li>
     <li>3 - render</li>
     <li>4 - image</li>
 </ul>
```

During every for loop iteration, the following helper variables are available:

- **loop.index**: The current iteration of the loop (1-indexed).
- **loop.index0**: The current iteration of the loop (0-indexed).
- **loop.revindex**: The number of iterations from the end of the loop (1-indexed).
- **loop.revindex0** The number of iterations from the end of the loop (0-indexed).
- **loop.key** If the iterator is an object, this will be the key of the current item; otherwise it will be the same as the loop.index.
- **loop.first** True, if the current item is the first in the object or array.
- **loop.last** True, if the current item is the last in the object or array.

To reverse a loop, we can use the reverse filter:

```
<ul>
{% for tag in data.ticket.tags | reverse %}
    <li>{{tag}}</li>
{% endfor %}
</ul>
```

Layouts

Layouts, in web applications, allow you to create a consistent design for the pages in your application. A layout defines the look and feel and standard behavior that you want for all of the pages in your application. Web applications consist of design elements such as a header and footer that are common to all pages. Creating a layout will enable to prevent the repeating of such elements in all the content pages. Layout can be inherited by individual content pages that contain the content you want to display. When users request the content pages, they merge with the layout to produce output that combines the design of the layout page with the content from the content page.

The default layout generated by KeystoneJS resides at `/templates/layouts.default.swig`

Layouts can define blocks of content areas that are placeholders for content from individual pages. Each page can reference a layout by using the **extends** tag. Below is a simple illustration of using a layout file.

Layout.swig*:*

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>{% block title %}IncTicket{% endblock %}</title> 6

  {% block head %}
  <link rel="stylesheet" href="/assets/css/incticket.css">
  {% endblock %}
</head>
<body>
  {% block content %}{% endblock %}
</body>
</html>
```

page.html:

```
{% extends "layout.html" %}

{% block title %}Ticket Details{% endblock %} 4

{% block head %}
  {% parent %}
  <link rel="stylesheet" href="/assets/css/singleticketcustom.css">
{% endblock %}

{% block content %}
<div class="panel panel-primary">
        <!-- Default panel contents -->
        <div class="panel-heading">Ticket details - {{data.ticket.
        title}}
        </div>
        ...
</div>
{% endblock %}
```

The rendered output would be:

```
<!doctype html>
<html>
<head>
  <meta charset="utf-8">
  <title>Ticket Details</title>

  <link rel="stylesheet" href="/assets/css/incticket.css">
  <link rel="stylesheet" href="/assets/css/singleticketcustom.css">
</head>
<body>
  <div class="panel panel-primary">
        <!-- Default panel contents -->
        <div class="panel-heading">Ticket details - Layout broken in IE6
        </div>
        ...
</div>
</body>
</html>
```

The **block** keyword defines an area in a template that can be overridden by a template extending this one and/or will override the current template's parent template block of the same name. The **extends** keyword makes the current template extend a parent template. This tag must be the first item in your template. The **parent** keyword injects the content from the parent template's block of the same name into the current block.

Advantages of Using Layouts: * Layouts centralize the common functionality of your pages so that developers can make updates in just one place. * They make it easy to create one set of controls and apply the results to a set of pages. For example, you can use controls on the layout to create a menu that applies to all pages. * Layouts allow fine-grained control over the layout of the final page by allowing you to control how the blocks are rendered. * They provide the capability to customize the master page from individual content pages.

10.4.1 Template Partials

In the previous section, we looked at layouts that act as the wrapper for design elements of the website on every page. Let us now take a look at creating template partials, which are small pieces of template that we can reuse and inject inside our layouts or other templates.

Template partials can be used to create reusable components in an application and reduce code duplication. Suppose we wanted to display a list of tickets on the main page, display a list of related tickets on the ticket details page, and display a list of tickets created by the user on a user's profile page. Rather than redundantly copying the code to display a list of tickets, we can manage this within a separate template file (known as template partial) and then include it in the templates as desired. This allows us to make changes to a single file and affect the behavior throughout our application.

We can include a template partial with the current local data available to it or by specifying a specific context. The **include** keyword is used to include template partials within other templates for rendering. The include function takes in the path to the template partial. We can use the **'with'** literal to specify an additional object that will become the data context for the partial when it is rendered. We can also specify the **'only'** keyword to restrict the context to be only the additional object that was passed in, that is, the included template will not be aware of any other local variables in the parent template.

Create a template partial named 'ticket.swig' in `/templates/views/tickets` and move the following code from ticketlist.swig to it.

ticket.swig partial:

```
<tr>
<td>
<div class=' col-md-1'>
    <span class="label label-info pull-right">{{ticket.status}}</span>
</div>

    <a href='{{ticket.url}}'><b>{{ticket.title |capitalize}}</b><a> 8

    <ul class="ticket-meta">
    <li> </li>
    <li>
        <small>Status</small><a href="" rel="tag">{{ticket.status}}</a>
    </li>
    <li>
        <small>Priority</small><a href="" rel="tag">{{ticket.
        priority}}</a>
    </li>
    <li>
        <small>Category</small><a href="" rel="tag">
        {{ticket.category}}</a>
    </li>
    <li>
        <small>Last Updated</small>
        <abbr class="last-updated">{{ticket._.createdAt.format
        ('Do MMMM YYYY')}}\
</abbr>
    </li>
    </ul>
</td>
</tr>
```

Update the ticketlist.swig to then use the ticket.swig partial.

```
<table class="table  table-striped">
        {% for ticket in data.tickets.results %}
           {% include 'ticket.swig' %}
        {% endfor %}
</table>
```

By default, Swig will complain if the template partial is not available at the path that was mentioned. We can use the **'ignore missing'** option to gracefully render an empty string in case the template partial was not available.

```
{% include 'ticket.swig' ignore missing %}
```

10.5 Swig Filters

Filters are methods through which output tags can process content. The rendering of variables in templates can be modified by filters. Filters are special functions that are applied after any object token in a variable block using the pipe character (|). Filters can also be chained together, one after another.

If, for example, we wanted to covert the title of a ticket to Title Case and strip any HTML tags that might have been input, we can use the title and striptags filters:

```
<div class="panel-heading">
    Ticket details - {{data.ticket.title | title | striptags}}
</div>
```

Here is a list of the available filters in Swig:

- **capitalize**: Capitalize words in the input
- **lower**: Convert an input string to lowercase
- **upper**: Convert an input string to uppercase
- **title**: Capitalizes every word given and lowercases all other letters in input
- **date**: Reformats a date
- **default**: A default return value can be specified if the input is undefined, null, or false
- **json**: Converts input to JavaScript object
- **striptags**: Strip html from string
- **safe**: Forces the input to not be auto-escaped. Swig escapes data by default.
- **replace**: Replace each occurrence of a string

- **escape**: Escape special characters in a string
- **addslashes**: Add backslashes to characters that need to be escaped
- **url_encode**: URL-encode a string
- **url_decode**: URL-decode a string
- **first**: Get the first element of the input array, object, or string
- **last**: Get the last element of the input array, object, or string
- **reverse**: Reverse sort the input values
- **sort**: Sort the input in an ascending order
- **join**: Join elements of the array with a delimiter
- **groupBy**: Group an array of objects by a key
- **uniq**: Remove all duplicate elements from an array

10.6 Summary

In this chapter, you learned about Swig, the powerful and flexible templating option for node.js. Consider using it the next time you need to support layouts, partials, and filters in your KeystoneJS projects. To learn more about swig, check out the Swig Documentation.[2]

[2] http://paularmstrong.github.io/swig

CHAPTER 11

Deploying to Ubuntu Cloud Server

11.1 Introduction

Let us look at how to provision a Ubuntu server from DigitalOcean and deploy our IncTicket application onto the server. Ubuntu is the most common Linux distribution used, now accounting for around 26 percent of all Linux web servers. I will be using Ubuntu 14.04 for this chapter.

11.2 What Is DigitalOcean?

DigitalOcean offers reliable, scalable, and inexpensive cloud computing services. They are a cloud infrastructure provider focused on simplifying web infrastructure for software developers. The service lets users deploy a complete Linux/unix server in under a minute with ease and provides critical functionality such as backups. Unlike a dedicated server, where the price is fixed up front on a monthly or yearly contractual basis, DigitalOcean, like Amazon Web Services, provides an hourly pricing based on usage without any contracts. This makes it a very attractive proposition for hosting web applications in a cost-effective manner.

Other similar service providers include Amazon Web Services and Rackspace. This chapter is not specific for DigitalOcean and will work similarly on other service providers' infrastructure.

11.3 Provision a Server

To be able to provision a server , first create an account with DigitalOcean at `http://digitalocean.com`.[1] Adding your SSH key to DigitalOcean at the time of provisioning a server makes it secure and easy to log in to the server. However, this is not mandatory since we can also use the root password to log in to the server that we will receive once we provision a server.

[1] `http://digitalocean.com`

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9_11

Now create a new droplet using the create droplet menu button on the top navigation. A droplet is DigitalOcean's terminology for a virtual private server. Select Ubuntu 14.04 as the distribution (Figure 11-1).

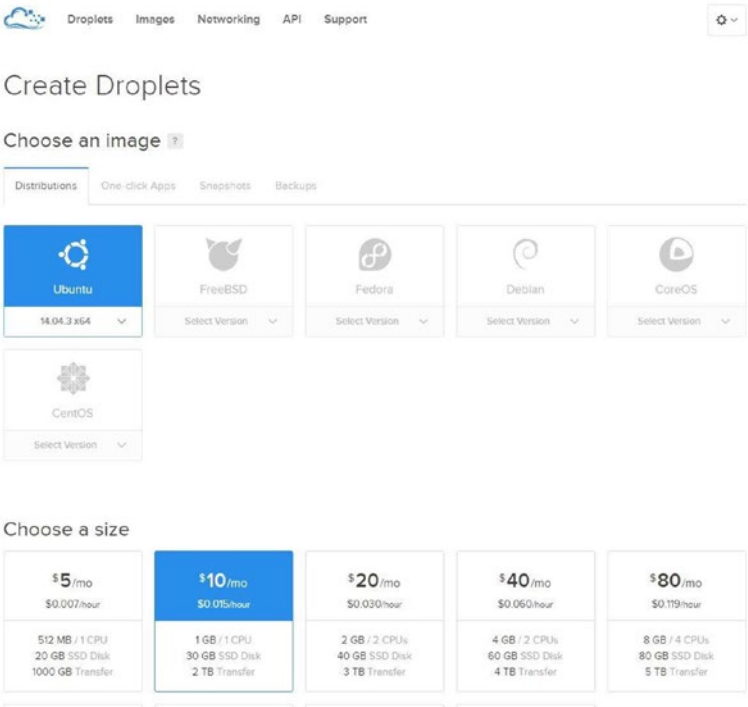


Figure 11-1. *Order a droplet*

Provide a sensible hostname for purposes of tracking this server and click create. The server should be provisioned quickly and be available in your dashboard. Write down the ip address since we will need it to SSH into our server (Figure 11-2).

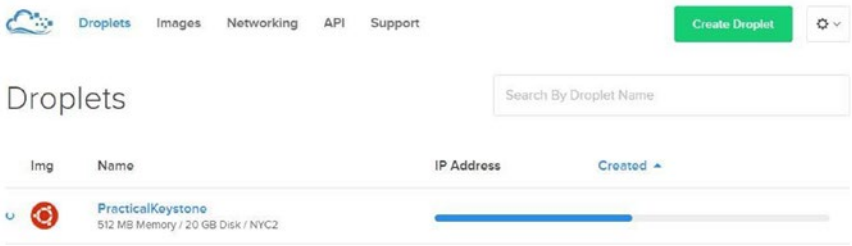


Figure 11-2. *Order a droplet*

We can use either the mac terminal or putty on windows to log in to our server. During the first login, we will be prompted to change our root password (Figure 11-3).

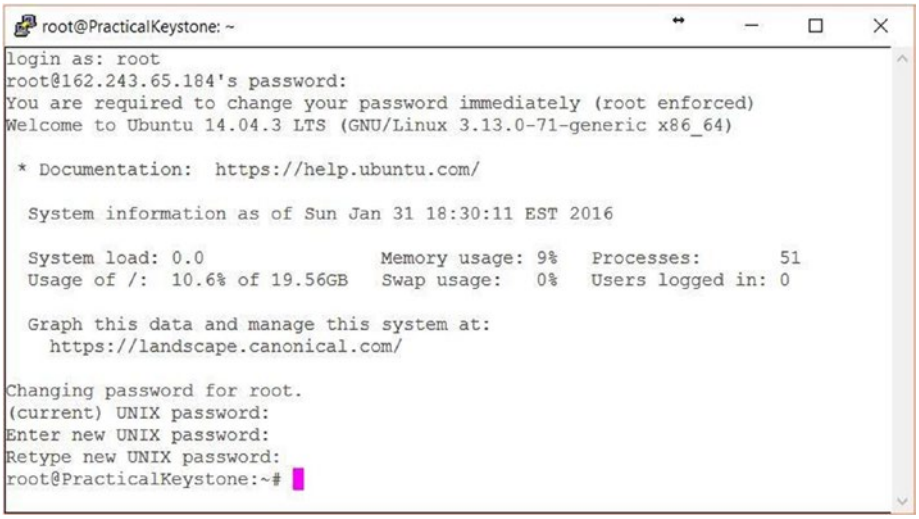


Figure 11-3. *Order a droplet*

11.4 Installing Node

Let us install Node on the server. On Ubuntu, you use the package manager **APT** to install packages and libraries. Each distribution of Linux uses its own package manager (such as yum for Fedora) so keep in mind the command if you use a different distribution.

Begin by upgrading apt-get and then updating apt-get. This is to ensure that the latest tools are installed and the system is up to date.

```
1   root@PracticalKeystone:~# apt-get upgrade
2   root@PracticalKeystone:~# apt-get update
```

To install Node runtime, use the command below. You might need to provide a confirmation that the installation was intentional on the command line:

```
1   root@PracticalKeystone:~# apt-get install nodejs
```

■ **Note** Due to a conflict with another package, the executable from the Ubuntu repositories is called nodejs instead of node.

We need to create a link to use just node as the name of the executable. Use the command below to achieve this.

```
1   sudo ln -s "$(which nodejs)" /usr/bin/node
```

Next install npm, which is the Node.js package manager. NPM will allow you to easily install modules and packages to use with Node.js. You can install this by typing:

```
1   root@PracticalKeystone:~# apt-get install npm
```

Check to see whether Node and NPM were installed successfully by checking the versions, as shown below:

```
1   root@PracticalKeystone:~# node --version
2   v0.10.25
3   root@PracticalKeystone:~# npm --version
4   1.3.10
```

As we see, the Node version that was installed is 0.10.25 is an outdated version. Let us update to the latest stable version of Node using **n**, which is a node helper package useful to install or upgrade a given Node version. Use the following commands to accomplish the update.

```
1   root@PracticalKeystone:~# npm cache clean -f
2   root@PracticalKeystone:~# npm install -g n
3   root@PracticalKeystone:~# n stable
4
5   root@PracticalKeystone:~# ln -sf /usr/local/n/versions/
    node/```<VERSION>```/bin/\
6   node /usr/bin/node
```

In the last command, replace the <VERSION> with the version of node in the `/usr/local/n/version-s/node/ folder`. At the time of writing this chapter, the stable version of node was 5.4.1.

11.5 Installing MongoDB

To install MongoDB package we will use the APT package manager. The default Ubuntu package repositories already contain MongoDB but they are mostly outdated versions. To ensure that we install the latest stable version, we need to add the MongoDB official repository. The first step is to add the GPG keys for the repository using the command below:

```
1   root@PracticalKeystone:~# apt-key adv --keyserver hkp://keyserver.
    ubuntu.com:80 \
2   --recv 7F0CEB10
```

Next, we have to add the MongoDB repository details so APT will know where to download the packages from, using the command below:

```
1   echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"
    /mongodb-org/3.\
2   0 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.0.list
```

After adding the repository details, we need to update the packages list using an apt-get update command.

```
1   root@PracticalKeystone:~# apt-get update
```

Now we are ready to install MongoDB using apt-get. Issue the command below to begin the installation.

```
1   root@PracticalKeystone:~# apt-get install -y mongodb-org
```

The command above will install several packages containing the latest stable version of MongoDB along with helpful management tools for the MongoDB server. After the package installation has completed, MongoDB will be automatically started. You can check the status of MongoDB by running the following command.

```
1   root@PracticalKeystone:~# service mongod status
```

If MongoDB is running, you'll see an output like below:

```
1   root@PracticalKeystone:/# service mongod status
2   mongod start/running, process 9713
```

You can also stop, start, and restart MongoDB using the service command.

- service mongod stop
- service mongod start
- service mongod restart

11.6 Install Redis for Caching

Redis is a flexible key-value data store that is a very good alternative to memcache for caching data in web applications. Redis is currently not available as a package to be installed using the APT package manager and needs to be compiled and installed on our server.

The installation is very simple and straightforward. Start by installing the prerequisites needed to compile and run Redis. We will need to download a compiler with build essential, which will help us install Redis from source and then install TCL.

```
1   root@PracticalKeystone:~# apt-get install build-essential
2   root@PracticalKeystone:~# apt-get install tcl8.5
```

Then begin to install Redis from the source.
Download the latest stable release of Redis from Redis.io.

```
1   root@PracticalKeystone:~# wget http://download.redis.io/releases/
    redis-stable.ta\
2   r.gz
```

Extract and change into that directory:

```
1   root@PracticalKeystone:~# tar xzf redis-stable.tar.gz
2   root@PracticalKeystone:~# cd redis-stable
```

Invoke the make command to compile the source code:

```
1   root@PracticalKeystone:~/redis-stable# make
```

Now run the recommended make test:

```
1   root@PracticalKeystone:~/redis-stable# make test
```

Next run the make install command to install the compiled binaries to /usr/local/bin.

```
1   root@PracticalKeystone:~/redis-stable# make install
```

At this point, Redis is installed on our machine. Let's run Redis as a daemon so it is running in the background. Redis comes with a shell script to install it as a background daemon. To run the script, change to the utils directory and run the `install_server.sh` script. The script will provide options to select the Redis port and configuration file. The default options are sufficient.

```
1   root@PracticalKeystone:~/redis-stable# cd ,
2   root@PracticalKeystone:~/redis-stable/utils# ./install_server.sh
```

Redis should be automatically started after installation as a daemon. To test the status, use the following command:

```
root@PracticalKeystone:~/redis-stable/utils# service redis_6379 status
Redis is running (19704)
```

You can also stop, start, and restart Redis using the service command.

- service redis_6379 stop
- service redis_6379 start
- service redis_6379 restart

To use Redis within a KeystoneJS application, I would recommend looking at cachegoose, a Mongoose caching module. Find more information at https://github.com/boblauer/cachegoose.[2]

11.7 Install IncTicket Application

To be able to retrieve our application from the github repository, we need to install git on the server first. Use the following command to install git:

```
root@PracticalKeystone:~# apt-get install git
```

Once git is installed, let us check out the repository to a directory named incticket and change into it.

```
root@PracticalKeystone:~# git clone https://github.com/manikantapanati/
incticket
root@PracticalKeystone:~# cd incticket
```

Now install the npm dependencies in package.json.

```
root@PracticalKeystone:/incticket# npm install --production
```

The production flag isn't required but will save some time and prevent installing some of your testing code that you don't need on the server.

Next, create an .env file to hold your configuration information specific to your deployment. An example would be:

```
COOKIE_SECRET=x/:dz9fvAALWMHrtAV9{P{8(T)UZnHZ}i@y)
vM#O:3y~+u*ZV=|8B"GGoXbOWl"P
MANDRILL_API_KEY=NY8RRKyv1Bure9bdP8-TOQ
MONGO_URI=mongodb://user:pass@server:port/practicalkeystone
```

[2] https://github.com/boblauer/cachegoose

■ **Note** It is not recommended to run an application as root for security reasons. However, to keep this chapter simple, we will run the application as root. In production, it is advised to create a new non-root user to run the application.

11.8 Manage Application with PM2

PM2 is an excellent process manager for Node.js applications. PM2 provides an easy way to manage and run applications as a service. Until now we have been running our application on the shell directly, which means the application will stop functioning if we close out of the terminal. To avoid this, we can use PM2 to keep our application running in the background. There are a few advantages to running the application using PM2 instead of just the terminal:

- PM2 will automatically restart your application if it crashes.
- PM2 will keep a log of your unhandled exceptions.
- PM2 can ensure that any applications it manages restart when the server reboots – as a service.

Use this command to install PM2 on our server:

```
1   root@PracticalKeystone:~/incticket# npm install pm2 -g
```

To confirm that PM2 is installed, try the `pm2 -h` command on the shell. You should see the help text.

Before we run our application under PM2, let us set up PM2 to launch on system startup (boot or reboot). This is pretty easy and we can accomplish it with the `startup` command. The startup option generates and configures a startup script to launch PM2 and its managed processes on server boot up. We must also specify the platform we are running on, which is Ubuntu, in our case:

```
1   root@PracticalKeystone:~/incticket# pm2 startup ubuntu
```

Output:

```
 1   [PM2] Generating system init script in /etc/init.d/pm2-init.sh
 2   [PM2] Making script booting at startup...
 3   [PM2] -linux- Using the command:
 4         su -c "chmod +x /etc/init.d/pm2-init.sh && update-rc.d pm2-init.
           sh default\
 5   s"
 6   Adding system startup for /etc/init.d/pm2-init.sh ...
 7     /etc/rc0.d/K20pm2-init.sh -> ../init.d/pm2-init.sh
 8     /etc/rc1.d/K20pm2-init.sh -> ../init.d/pm2-init.sh
 9     /etc/rc6.d/K20pm2-init.sh -> ../init.d/pm2-init.sh
10     /etc/rc2.d/S20pm2-init.sh -> ../init.d/pm2-init.sh
11     /etc/rc3.d/S20pm2-init.sh -> ../init.d/pm2-init.sh
```

```
12     /etc/rc4.d/S20pm2-init.sh -> ../init.d/pm2-init.sh
13     /etc/rc5.d/S20pm2-init.sh -> ../init.d/pm2-init.sh
14   [PM2] Done.
```

Now, run our application under PM2 using the command below:

```
1   root@practicalkeystone:~/incticket# pm2 start keystoneJS
```

You should a similar output indicating that the service has started:

```
 1   [PM2] restartProcessId process id 0
 2   [PM2] Process successfully started
 3   +-------------------------------------------------------------------
     ----------\
 4   ----+
 5   ¦ App name ¦ id ¦ mode ¦ pid    ¦ status ¦ restart ¦ uptime ¦ memory ¦
     watch\
 6   ing ¦
 7   +----------+----+------+-------+--------+---------+--------+----------
     ---+------\
 8   ----¦
 9   ¦ keystone ¦ 0  ¦ fork ¦ 24820 ¦ online ¦ 0    ¦ 0s     ¦ 10.488 MB ¦
     disab\
10   led ¦
11   +-------------------------------------------------------------------
     ---------- \
12   ----+
13
14   Use `pm2 show <id|name>` to get more details about an app
```

PM2 automatically assigns an App name based on the filename, in our case – keystoneJS and a PM2 id. PM2 also tracks information such as the PID of the process, its current status, and memory usage.

To stop our application, use this command:

```
1   root@practicalkeystone:~/incticket# pm2 stop keystone
```

To restart our application, use this command:

```
1   root@practicalkeystone:~/incticket# pm2 restart keystone
```

11.9 Install NGINX as Reverse Proxy Server

We have successfully set up our application and it will, by default, listen to port 3000. This is not very useful for web-facing applications since it prevents users from accessing it over HTTP that runs on port 80. We will set up an NGINX web server as a reverse proxy to address this. A reverse proxy server directs client requests to the appropriate back-end

server and provides an additional level of abstraction between clients and servers. This can be useful for things such as load balancing, caching, and security.

To install NGINX use APT as shown below:

```
root@practicalkeystone:~/incticket# sudo apt-get install nginx
```

Next, edit the NGINX configuration file:

```
root@practicalkeystone:~/incticket# vi /etc/nginx/sites-available/
default
```

and replace the contents with the configuration data below:

```
 server {
     listen 80;

     server_name practicalkeystone;

     location / {
         proxy_pass http://127.0.0.1:3000;
         proxy_http_version 1.1;
         proxy_set_header Upgrade $http_upgrade;
         proxy_set_header Connection 'upgrade';
         proxy_set_header Host $host;
         proxy_cache_bypass $http_upgrade;
     }
 }
```

The above setting configures the web server to respond to requests at its root on port 80. Any requests to our web server domain/ip address via a web browser would send the request to the application server on port 3000, which would be received and replied to by the KeystoneJS application.

After editing the configuration file, save and exit. Restart the NGINX service for the changes to take effect.

```
root@practicalkeystone:~/incticket# service nginx restart
```

11.10 Summary

In this chapter, you learned about setting up an Ubuntu server at DigitalOcean and installing all the necessary dependencies for running our KeystoneJS application. We installed NGINX as a reverse proxy in front of our Node process. Feel free to take this to the next level by configuring NGINX for load balancing, caching, and security. Learn more about NGINX at https://www.nginx.com/resources/wiki/start/inde.[3]

[3] https://www.nginx.com/resources/wiki/start/index.html#

Index

A

Authentication and user management
- configuration options, 75–77
- password recovery
  - email template, 90–91
  - forgotpassword.js, 88–90
  - keystone.Email method, 90
  - resetPasswordKey, 92–94
  - send method, 90
- registering users
  - async.series(), 84
  - duplicate username detection, 83
  - FlashMessages.renderMessages, 81
  - join.js, view, 81–83
  - sign-up window, 80
  - template, 79–80
- restrict access, 95
- retrieval, 95
- security
  - Cookies, 98
  - CSRF, 96–97
  - XSS, 97
- sign out, 88
- user login
  - keystone.session.signin method, 87
  - onSuccess method, 87
  - signin.js, 86
  - template creation, 84–85
  - validation error, 88
  - view.on method, 86
- user model, 77–78

Authentication configuration options
- app.all method, 77
- auth, 75
- cookie secret, 76
- initLocals, 76
- MongoDB, 76
- password reset, 77
- session, 75
- session store, 76
- Sign-in Links, 77
- user model, 76

B

Browse MongoDB collections, 11

C

Content management systems (CMS), 2
Cookies, 98
Cross-site request forgery (CSRF), 96–97
Cross-site scripting (XSS), 97

D, E

db.tickets.getIndexes() method, 100
Debugging
- in Visual Studio Code, 8–9

defaultColumns option, 26–27
defaultSort option, 27
DigitalOcean, 133

F, G, H

Filtering relations, 60–61
Full-text search, 99

I, J

IncTicket application
- additional columns, 27
- Administration Site, 24–25
- Admin Menu, 24
- an Admin User, 22

M. Panati, *Beginning KeystoneJS*, DOI 10.1007/978-1-4842-2547-9

IncTicket application (*cont.*)
autokey.from path, 28
custom field, 28
datetime field, 29
display, 26–27
environment-specific configurations, 16
first route, 29
HTTP response, 32
and KeystoneJS, 13–14
KeystoneJS Administration Site, 23
model/list, 19–21
new KeystoneJS, 14, 16
pagination, 39–42
project structure, 17–19
relationship field, 29
select field, 29
string field, 29
templates, display tickets, 34–39
ticket list and detail views, 32–34
tickets route, 30–31
in uppercase, 17
URLs, 31
index.js file, 18

K, L

KeystoneJS
0.3.x and 0.4 versions, 2
auto-generated Admin UI, 1
database fields, 2
development and debugging tools, 7–9, 11
email sending, 2
form processing, 2
installing and using, 1
modular architecture and clean separation, 2
modularity, 1
and MongoDB, 2–4
and Node.js, 4–6
session management, 1
KeystoneJS Project Structure, 18
KeystoneJS, search view, 103–105

M

Many-to-Many relations, 55, 58–60
Middleware
assigning to routes, 73
cors, 72
flashMessages, 72
initAPI, 72
initLocals, 72
isAdmin function, 73
KeystoneJS, 72
requireUser, 72
Middleware.js, 18
MongoDB database, 1–2
Mongoose schemas
advantages, 43
conditional filtering, 50
counting documents, 49
existing tickets, 53
exists method, 51
fetch all tickets, 48
finding data, 46
and Keystone Lists, 44
limiting returned documents, 50
MongoDB, 43
Mongoose ODM, 43
new document, 51, 53
optimal performance, 48
ordering documents, 49–50
properties/fields, 46
QueryBuilder, 47
single-query operation, 47
Ticket by Slug, 48
Ticket Model/List, 44
virtual properties, 46
Multiple Fields/Wild Card Indexing, 101, 103

N

Node.js Installer, 6
Node Package Manager (NPM), 5, 14

O

One-to-many relation, 55
retrieval, 57–58
Ticket Model, 56
user model, 56
One-to-One Relation, 56

P

Product model
add fields, 45
data types, 45
fields, 45–46
KeystoneJS fields, 45

Q

Query MongoDB collections, 11

R

render() method, 33
RESTful APIs
    apiError method, 108
    apiNotAllowed method, 109
    apiNotFound method, 109
    apiReponse method, 108
    DELETE, 119, 121
    endpoint, 108
    GET Requests, 114–115
    JSON Formatter Chrome extension, 112
    keystone.middleware.api
        parameter, 108
    POST and PUT Requests, 115–117, 119
    POSTMAN REST Client, 113
    protocols, 107
    ticket.js, 109–111
    tickets ID, 107
    tools, 111

S

Single field indexing, 99, 101
Swig filters, 131–132
Swig template engines
    advantages, layout, 129
    dynamic content, 123
    keystone.init method, 124
    layout, 127
    Layout.swig, 128
    logic tags, 125–127
    output tags, 125
    page.html, 128
    swig filters, 131–132
    syntax, 124
    template partials, 129–130

T

Ticket form creation
    createdBy property, 68
    'has-errors' css class, 66
    input validation, 66
    routes/views/tickets, 67
    template, 63
    users list retrival, 68
    web form, 64–66
Ticket.register(), 24

U

Ubuntu Cloud Server
    DigitalOcean, 133
    IncTicket installation, 139
    MongoDB installation, 137
    NGINX, 141
    Node installation, 135–136
    PM2, 140–141
    Redis installation, 138–139
    server provision, 133, 135
URL() virtual method, 31

V, W, X

Visual Studio Code, 7

Y, Z

Yeoman, 13–14