Keystone.js Succinctly®

by Manikanta Panati

Syncfusion | Technology Resource Portal

Keystone.js Succinctly

By

Manikanta Panati

Foreword by Daniel Jebaraj

Syncfusion

Deliver innovation with ease

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Hillary Bowling, online marketing manager, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to "enable AJAX support with one click," or "turn the moon to cheese!"

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and "Like" us on Facebook to help us spread the word about the *Succinctly* series!

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 Masters degrees in Information Systems and Project Management, and Microsoft Certified Technical Specialist (MCTS) and 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 the migration and development of projects.

Chapter 1 Introduction

What is Keystone.js?

Keystone.js is a Node.js web framework for developing database-driven websites, applications, and RESTful APIs. The framework is built on Express.js and MongoDB, and follows the Model-View-Template design pattern. Express.js is the de facto web application server framework for Node.js-based applications. MongoDB is a very popular NoSQL database. Keystone.js is free and open source. The framework does a lot of heavy lifting and allows developers to focus on clean and rapid development.

Keystone.js, as with Node.js applications, emphasizes reusability and modularity of components. The framework makes it very easy to manage the application templates, views, and routes. JavaScript is used throughout for configuration, files, and data model development. The framework helps with most common web tasks out of the box, such as authentication, content administration, session management, email-sending, and many more tasks.

The framework provides an automatic administration interface that can be used to create, read, update, and delete data in the application. The administration GUI is generated dynamically through inspection of models and user options. Developers can use 20+ built-in field types that provide the capability to manage data ranging from text, dates, geolocation, and HTML to images and files uploaded to Amazon S3 or Microsoft Azure.

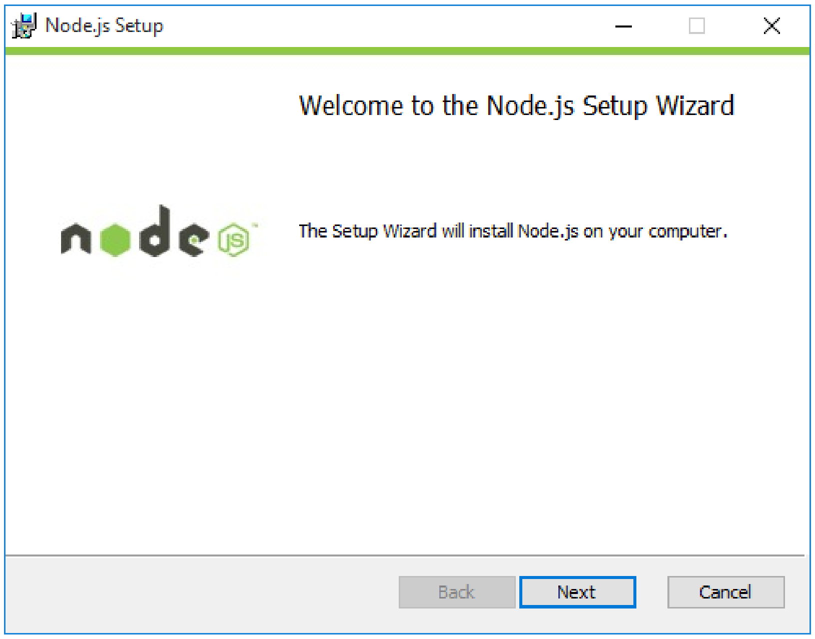
More information is available at the official website.

Installing Node.js

Node.js is an open-source, cross-platform runtime environment that is most commonly used for developing JavaScript-based, server-side web applications. Node.js is becoming a very popular tool of choice for building highly performant and scalable web applications due to its async model of handling requests on a single thread. Node.js can be installed on a wide variety of operating systems including Windows, Linux, and Mac OS. This book will primarily focus on running Keystone.js on Node.js in a Windows environment.

To install Node.js:

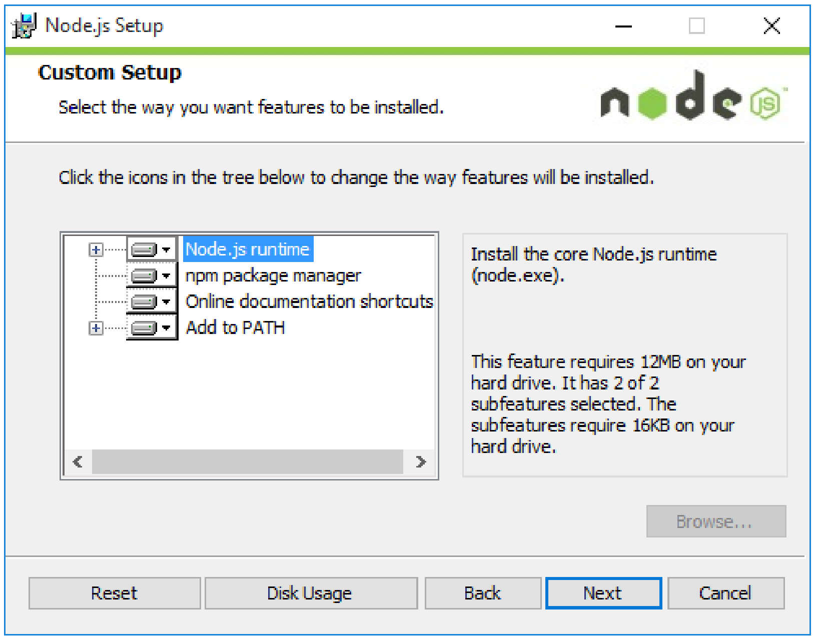
1. Visit the download page on the Node.js official website.
2. Click on the download link for the latest stable release .MSI under Windows 32-bit or 64-bit, depending on your machine architecture.
3. Once the download is complete, double-click on the installer file, which will launch the Node.js installer. Proceed through each step of the installation wizard.



The Setup Wizard will install Node.js on your computer.

Figure 1: Node 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 installations.



Select the way you want features to be installed.

Click the icons in the tree below to change the way features will be installed.

- Node.js runtime
- npm package manager
- Online documentation shortcuts
- Add to PATH

Install the core Node.js runtime (node.exe).

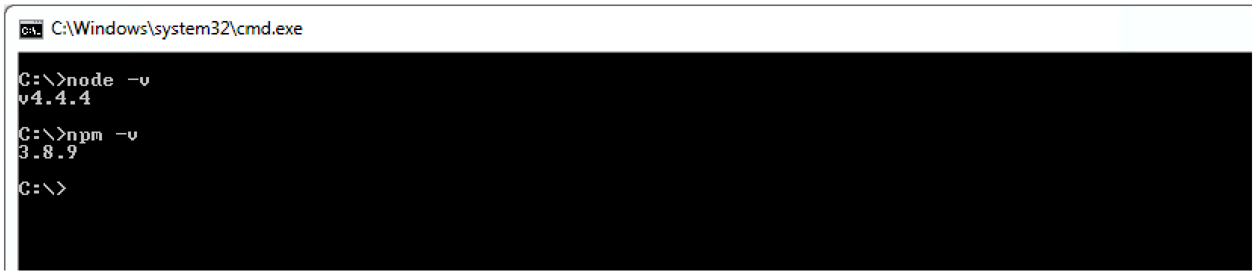
This feature requires 12MB on your hard drive. It has 2 of 2 subfeatures selected. The subfeatures require 16KB on your hard drive.

Figure 2: Custom Setup screen

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

Testing whether Node.js is installed 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. We should see the versions output on the screen.



```
C:\>node -v
v4.4.4

C:\>npm -v
3.8.9

C:\>
```

Figure 3: Node and NPM version check

Installing MongoDB

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 (such as MySQL).

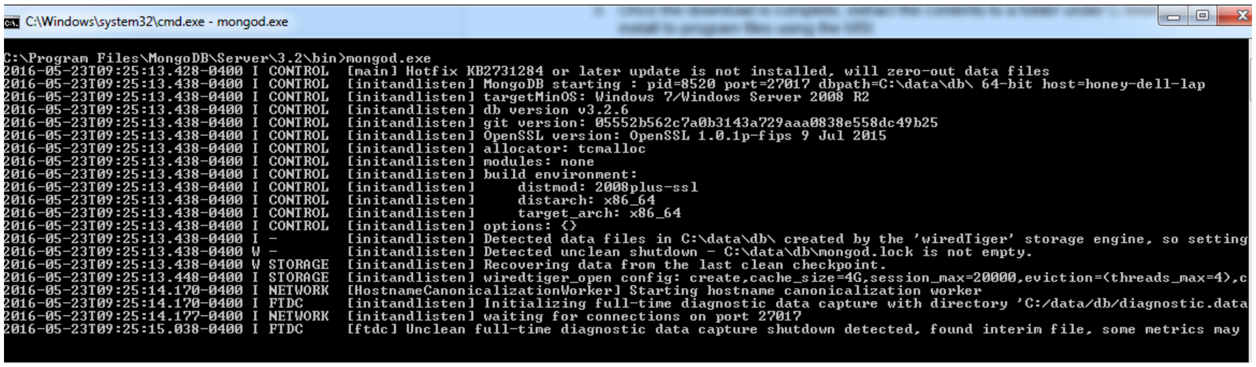
Tip: MLab offers a free managed sandbox for MongoDB that can be used as a test ground instead of installing MongoDB locally.

To install MongoDB locally:

1. Navigate to the MongoDB download page.
2. Click on the download link for the latest zip archive or MSI under Windows 32-bit or 64-bit, depending on your machine architecture.
3. Once the download is complete, extract the contents to a folder under **C:\mongodb** or install to program files using the MSI.
4. Create the default database path (**C:\data\db**). This is the location where the database files used by MongoDB will reside.
5. To start the MongoDB database, open a CMD prompt window, and enter the following code (you may have to allow access through a firewall).

Code Listing 1: Start MongoDB

```
C:\Program Files\MongoDB\Server\3.2\bin>mongod.exe
```



```
C:\Program Files\MongoDB\Server\3.2\bin>mongod.exe
2016-05-23T09:25:13.428-0400 I CONTROL  [main] Hotfix KB2731284 or later update is not installed, will zero-out data files
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] MongoDB starting : pid=8520 port=27017 dbpath=C:\data\db\ 64-bit host=honey-dell-lap
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] targetMinOS: Windows 7/Windows Server 2008 R2
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] db version v3.2.6
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] git version: 05552b562c7a0b3143a729aaa0838e558dc49b25
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] OpenSSL version: OpenSSL 1.0.1p-fips 9 Jul 2015
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] allocator: tcmalloc
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] modules: none
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] build environment:
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten]     distmod: 2008plus-ssl
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten]     distarch: x86_64
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten]     target_arch: x86_64
2016-05-23T09:25:13.438-0400 I CONTROL  [initandlisten] options: {}
2016-05-23T09:25:13.438-0400 I -        [initandlisten] Detected data files in C:\data\db\ created by the 'wiredTiger' storage engine, so setting
2016-05-23T09:25:13.438-0400 W -        [initandlisten] Detected unclean shutdown - C:\data\db\mongod.lock is not empty.
2016-05-23T09:25:13.438-0400 W STORAGE  [initandlisten] Recovering data from the last clean checkpoint.
2016-05-23T09:25:13.448-0400 I STORAGE  [initandlisten] wiredtiger_open config: create,cache_size=4G,session_max=20000,eviction=(threads_max=4),c
2016-05-23T09:25:14.170-0400 I NETWORK  [HostnameCanonicalizationWorker] Starting hostname canonicalization worker
2016-05-23T09:25:14.170-0400 I FTDC     [initandlisten] Initializing full-time diagnostic data capture with directory 'C:/data/db/diagnostic.data
2016-05-23T09:25:14.177-0400 I NETWORK  [initandlisten] waiting for connections on port 27017
2016-05-23T09:25:15.038-0400 I FTDC     [ftdc] Unclean full-time diagnostic data capture shutdown detected, found interim file, some metrics may
```

Figure 4: Start up the MongoDB server

Installing Yeoman

Yeoman is a set of tools for automating development workflow. It scaffolds out a new application along with writing build configuration and pulling in build tasks and NPM dependencies needed for the build. Keystone.js provides a very handy Yeoman generator to generate a new project.

To install Yeoman, issue the following command on a command prompt.

Code Listing 2: Install Yeoman

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

Next, to install the Yeoman keystone app generator, use the following command.

Code Listing 3: Install Yeoman generator for keystone

```
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 Keystone.js generator.

Summary

We have reached the end of the first chapter. Setting up a development environment is a very important task, and we just did this. We've covered the necessary requirements to begin working with Keystone.js application framework. Onwards!

Chapter 2 Creating Your First Project

The Keystone.js framework has very few requirements. We need to make sure we have the following installed:

- Node Runtime
- Node Package Manager
- MongoDB Database
- Yeoman package

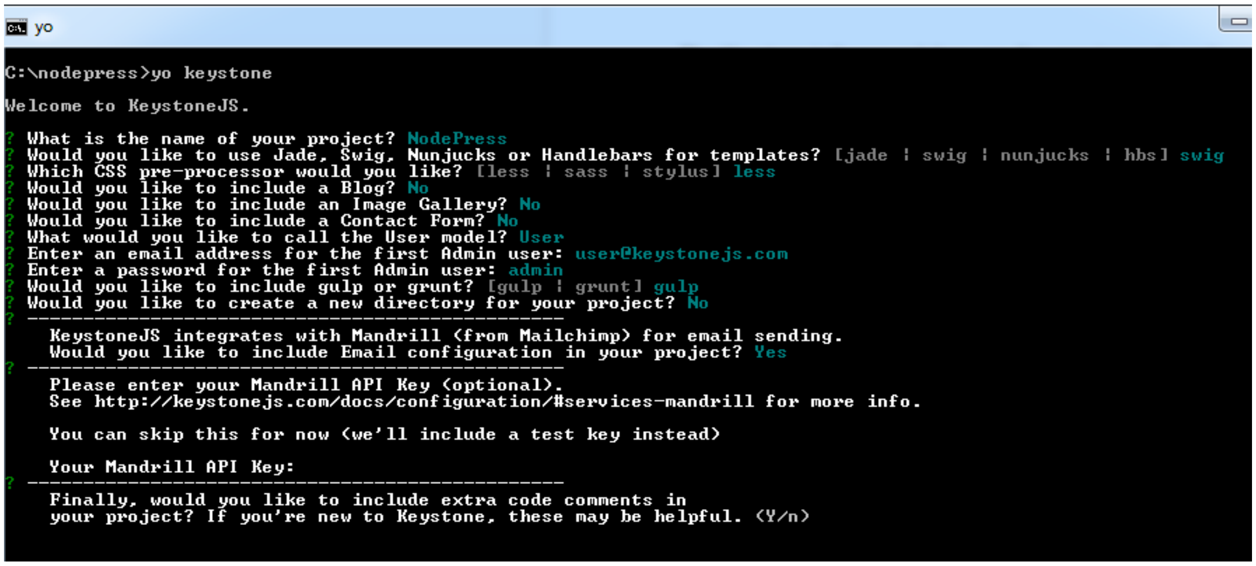
Scaffolding an empty project

Keystone.js utilizes NPM to manage its dependencies. So, before using Keystone.js, make sure you have NPM installed on your machine. The Yeoman keystone application generator can be used to quickly scaffold an empty 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 changed later within the new application.

Create a new project by issuing the following command at the command prompt within the folder you intend to keep the project in (in this case, **NodePress**).

Code Listing 5: Scaffold a Keystone.js project

```
c:\nodepress> yo keystone
```



```
C:\nodepress>yo keystone

Welcome to KeystoneJS.

? What is the name of your project? NodePress
? Would you like to use Jade, Swig, Nunjucks or Handlebars for templates? [jade | swig | nunjucks | hbs] swig
? Which CSS pre-processor would you like? [less | sass | stylus] less
? Would you like to include a Blog? No
? Would you like to include an Image Gallery? No
? Would you like to include a Contact Form? No
? What would you like to call the User model? User
? Enter an email address for the first Admin user: user@keystonejs.com
? Enter a password for the first Admin user: admin
? Would you like gulp or grunt? [gulp | grunt] gulp
? Would you like to create a new directory for your project? No
? ------------------------------------------------
    KeystoneJS integrates with Mandrill (from Mailchimp) for email sending.
    Would you like to include Email configuration in your project? Yes
? ------------------------------------------------
    Please enter your Mandrill API Key (optional).
    See http://keystonejs.com/docs/configuration/#services-mandrill for more info.

    You can skip this for now (we'll include a test key instead)

    Your Mandrill API Key:
? ------------------------------------------------
    Finally, would you like to include extra code comments in
    your project? If you're new to Keystone, these may be helpful. (Y/n)
```

Figure 5: Scaffold project using Yeoman

I specified the following information when creating the demo project:

```
Name: NodePress

Template: swig

Blog: no

Image Gallery: no

Contact Form: No

User: User

Admin email: user@keystonejs.com

Admin pwd: admin

Gulp or Grunt: gulp

New Directory: no

Mandrill email: Yes

Mandrill API Key: (none)

Include Comments: Yes
```

Application configuration

Keystone.js uses an excellent Node.js library, namely **dotenv**, to load the configuration data at runtime. In a fresh Keystone.js 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 receives a request. 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 equal sign. Keys are generally written in upper case.

Code Listing 6: Configuration settings

```
COOKIE_SECRET=oQQ*s0pz5(bF4gpmoNwM|BDB~db+qwQ`K>Ik~*R2D

MANDRILL_API_KEY=NY8RRKyv1Bure9bdP8-TOQ
```

To access the configuration variables in our application, we can use them as presented in Code Listing 7.

Code Listing 7: Access Configuration settings

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

Note: To put a Keystone.js 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, Keystone.js tries to connect to a local instance of MongoDB and uses no authentication. However, if you want to specify a MongoDB connection string, it's pretty easy to do so using the .env file.

Code Listing 8: MongoDB Connection setting

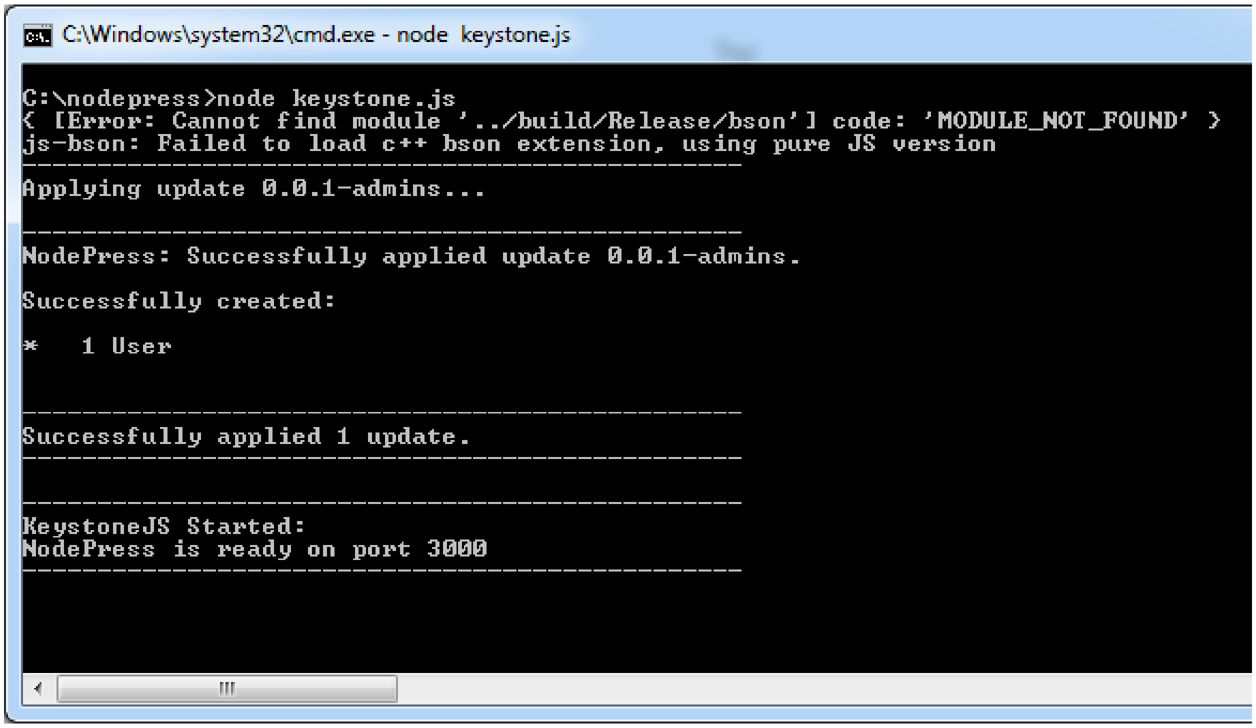
```
MONGO_URI=mongodb://user:password@localhost:27017/databasename
```

Launching the development server

After scaffolding the project, configuring the MongoDB connection setting, and starting the mongod.exe program, we can launch the development server using the following command.

Code Listing 9: Launch development server

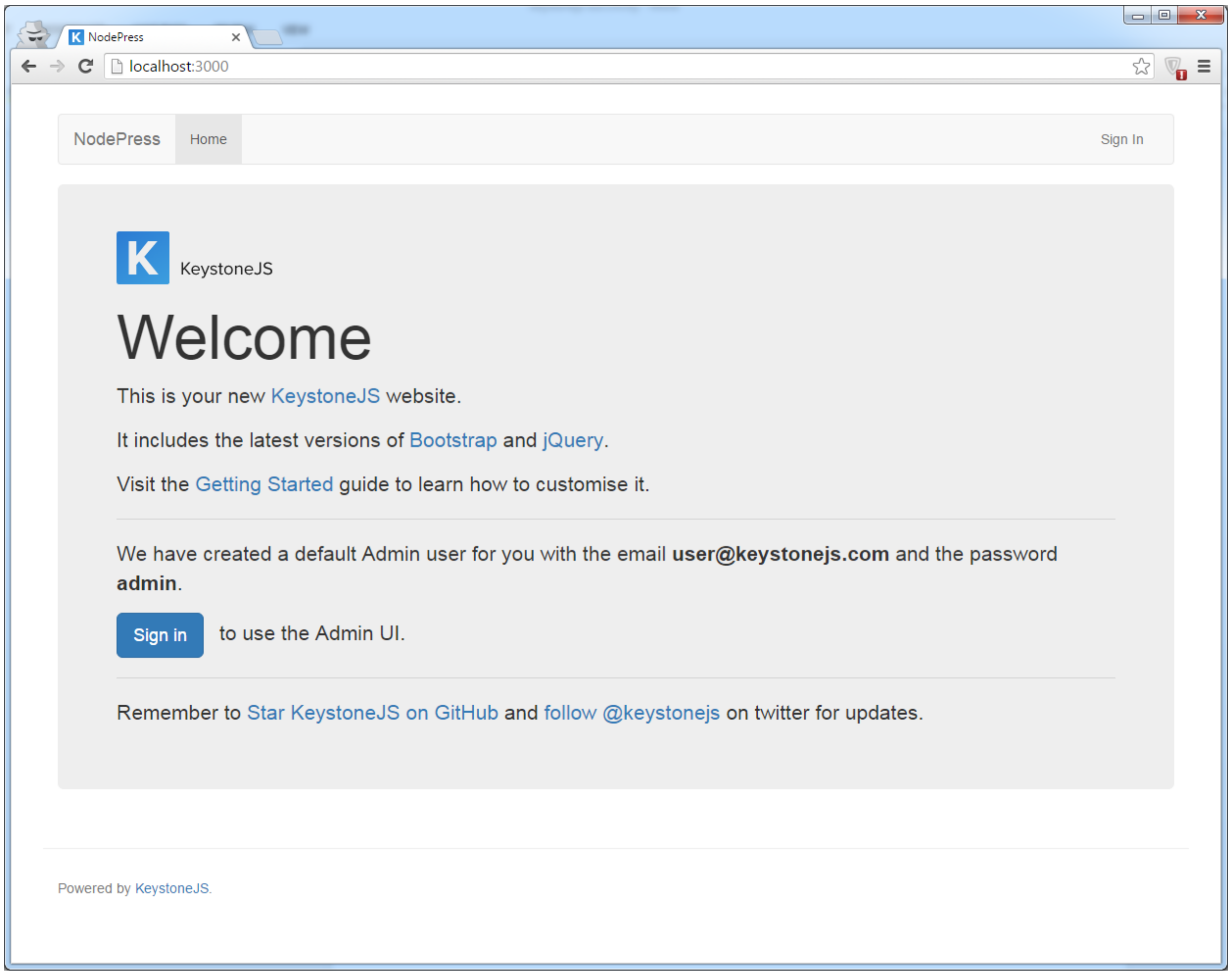
```
c:\nodepress> node keystone.js
```



```
C:\nodepress>node keystone.js
{ [Error: Cannot find module '../build/Release/bson'] code: 'MODULE_NOT_FOUND' }
js-bson: Failed to load c++ bson extension, using pure JS version
------------------------------------------------
Applying update 0.0.1-admins...

------------------------------------------------
NodePress: Successfully applied update 0.0.1-admins.

Successfully created:

*   1 User


------------------------------------------------
Successfully applied 1 update.
------------------------------------------------

------------------------------------------------
KeystoneJS Started:
NodePress is ready on port 3000
------------------------------------------------
```

Figure 6: First run of development server

This command will serve up your project on port 3000. At the first run, Keystone.js will attempt to apply an update—a framework function, which will try to create the admin user that was configured during the project scaffolding. Navigate to http://localhost:3000 and you should see the Keystone.js landing page.



NodePress
localhost:3000
NodePress Home Sign In

KeystoneJS

Welcome

This is your new KeystoneJS website.

It includes the latest versions of Bootstrap and jQuery.

Visit the Getting Started guide to learn how to customise it.

We have created a default Admin user for you with the email **user@keystonejs.com** and the password **admin**.

Sign in to use the Admin UI.

Remember to Star KeystoneJS on GitHub and follow @keystonejs on twitter for updates.

Powered by KeystoneJS.

Figure 7: Application landing page

Keystone.js administration site

Keystone.js comes with a built-in administration interface that is very useful for managing content. The Keystone.js 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.

Start up our app using the node keystone.js command and open http://127.0.0.1:3000/keystone/signin in your browser. You should see the administration login page shown in Figure 8.

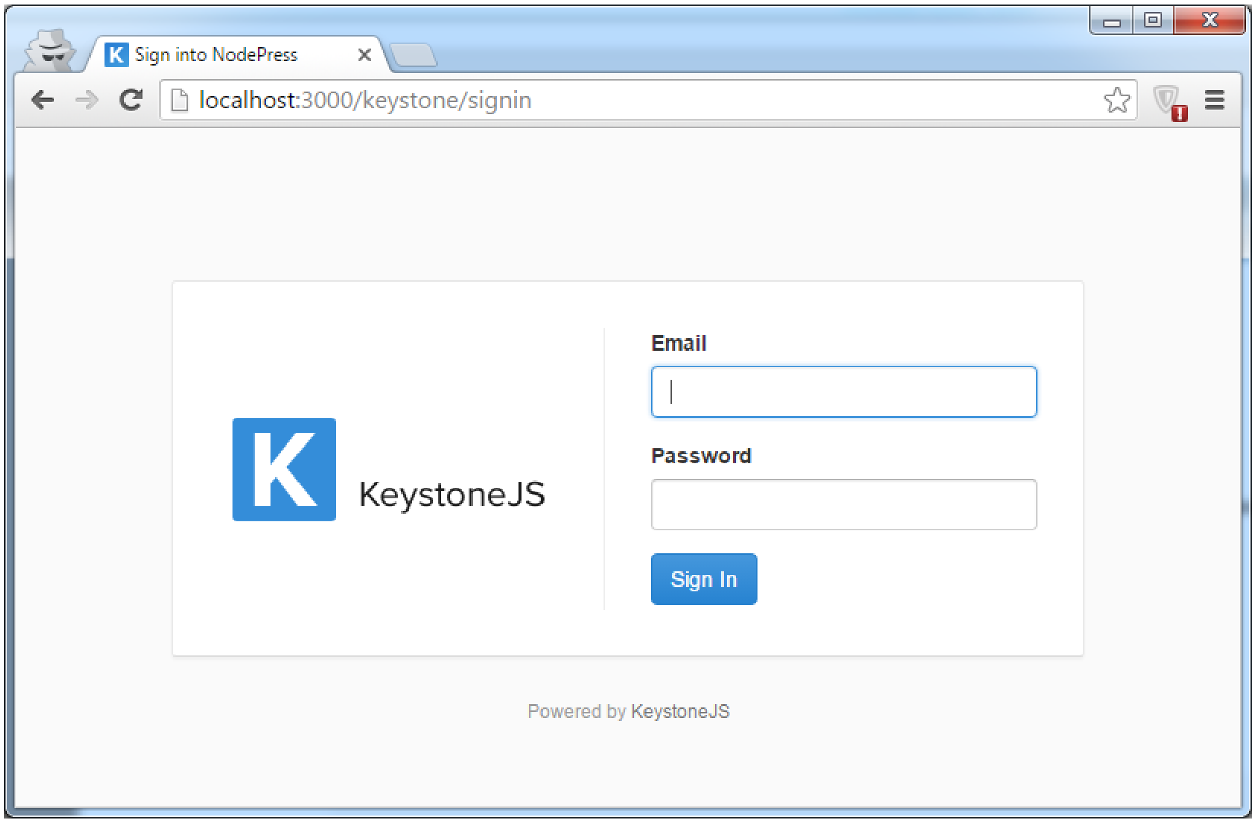
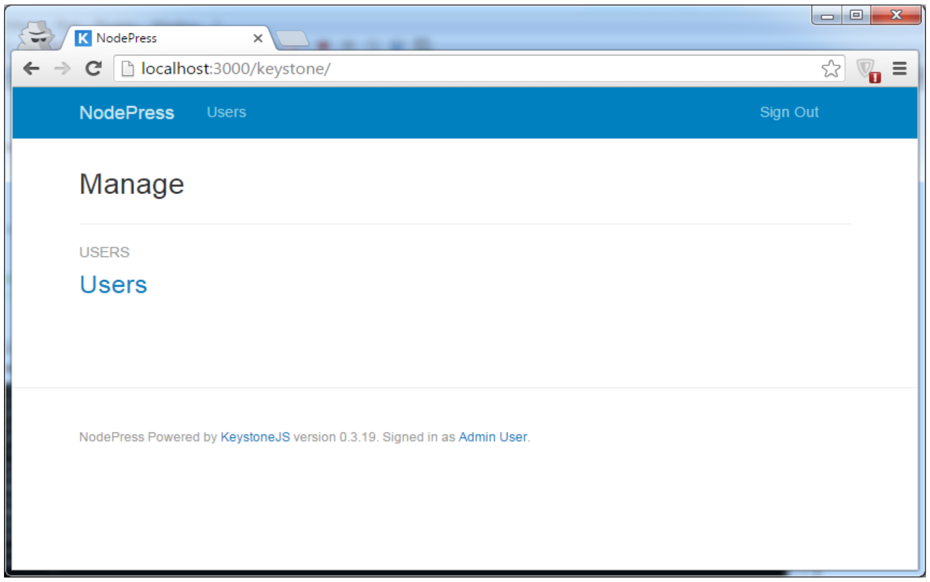


Figure 8: Administration login

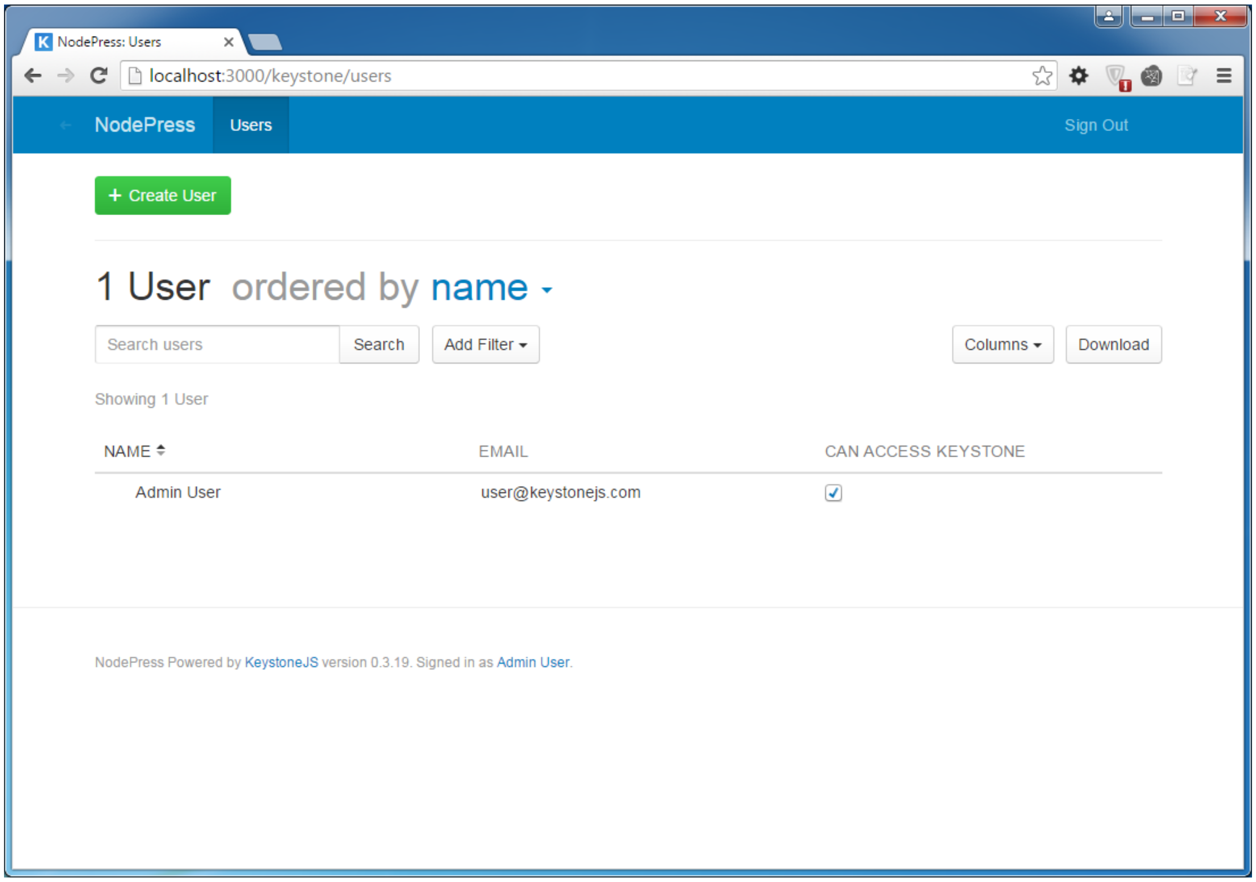
Log in using the user credentials set up during the scaffolding step (for example, user@keystonejs.com and "admin"). You will see the admin site index page, as shown in Figure 9.



NodePress Powered by KeystoneJS version 0.3.19. Signed in as Admin User.

Figure 9: Administration UI

The user model on the page is automatically created for us by Keystone.js. 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.



NAME	EMAIL	CAN ACCESS KEYSTONE
Admin User	user@keystonejs.com	✓

NodePress Powered by KeystoneJS version 0.3.19. Signed in as Admin User.

Figure 10: Manage users in administration UI

Summary

Now that we have set up our development environment and an empty project, we can move on to implementing advanced application features in Keystone.js. Before we can do anything visual, we need something to display. In the next chapter, you will be introduced to the Keystone.js models.

Chapter 3 Data Modeling in Keystone.js

The Mongoose object document mapper (ODM) included with Keystone.js provides a beautiful, simple API implementation for working with your MongoDB database. Each database collection has a corresponding "model" that is used to interact with that collection. Models allow you to query for data in your Mongo collections, as well as insert new documents into the collection.

Mongoose provides an abstract and common interface to the data in a document database. The ODM makes it very easy to convert data between JavaScript objects and the underlying Mongo documents.

Defining models

With Keystone.js, creating a model is as easy as defining a JavaScript file and specifying a number of attributes assigned to that file. Let's start with a very basic model for our news entries. Create a new file named **News.js** in the project's **Models** directory and enter the following code.

Code Listing 10: News.js model

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

/**
 * News Model
 * ==========
 */

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

News.add({
    title: { type: String, required: false },
    state: { type: Types.Select, options: 'draft, published, archived',
default: 'draft', index: true },
    author: { type: Types.Relationship, ref: 'User', index: true },
    publishedDate: { type: Types.Date, index: true, dependsOn: { state:
'published' } },
    content: { type: Types.Html, wysiwyg: true, height: 400 }
});

News.defaultColumns = 'title, state|20%, author|20%, publishedDate|20%';
News.register();
```

There is a lot going on, so let's start with the **`require`** statement and work our way down. We begin by importing the standard Keystone library and obtaining a reference to the **`Keystone`** field types.

Next is the **`News`** model definition. Our **`News`** model is an object that is an instance of the **`keystone.List`**. By relying on the **`keystone.List`**, our **`News`** object will inherit a variety of helpers that we'll use to query the database.

Before adding fields to the **`News`** model, we define the name of the model as the first parameter to the list—in our case, **`News`**. The second parameter is an object that can be used to assign behaviors to the **`News`** model. The **`autokey`** option is used to generate slugs for the model, which we will use to give our news entries some nice URLs. The URL is generated from the title of the post and can be accessed via the **`slug`** property of a news post. If the unique option is set to **`true`**, Keystone.js validates that no other post exists with the same title as the one being entered. This is an easy way to prevent duplicate news.

Each post can have multiple fields within it, which can be used to enter relevant data. The attributes of the **`News`** model are a simple mapping of the names and data that we wish to store in the database. They are listed as follows:

- **`Title:`** This field can hold a string and is used for storing the news post title. 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.
- **`State:`** This is a field in which to save the status of the news post. We use a select field type, so the value for this field can be set to one of the given choices. The default option is set to a **`draft`** status.
- **`Content:`** This is the field to be used to store the description of the ticket. The **`Text area`** field type will display a text area within the admin UI.
- **`Author:`** This field will hold a reference to the user who created the news post. The field is like a foreign key that defines many-to-one relationships in a relational database. This field is displayed as an autosuggest 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 Keystone.js that we are interested in a database index to be created for this field. The **`categories`** field can be set up similarly to a **`relationship`** field.
- **`PublishedDate:`** This date-time field indicates when the news post 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 post object.

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 Code Listing 10, we are specifying the **`title`**, **`state`**, **`author`**, and **`publishedDate`** as the default columns to display in the admin UI, with **`state`**, **`author`**, and **`publishedDate`** being given column widths. The call to register on our **`keystone.js`** list finalizes the model with any attributes and options we set.

Restart the application and refresh the administration page. You should see the option to manage news items, as shown in Figure 11.

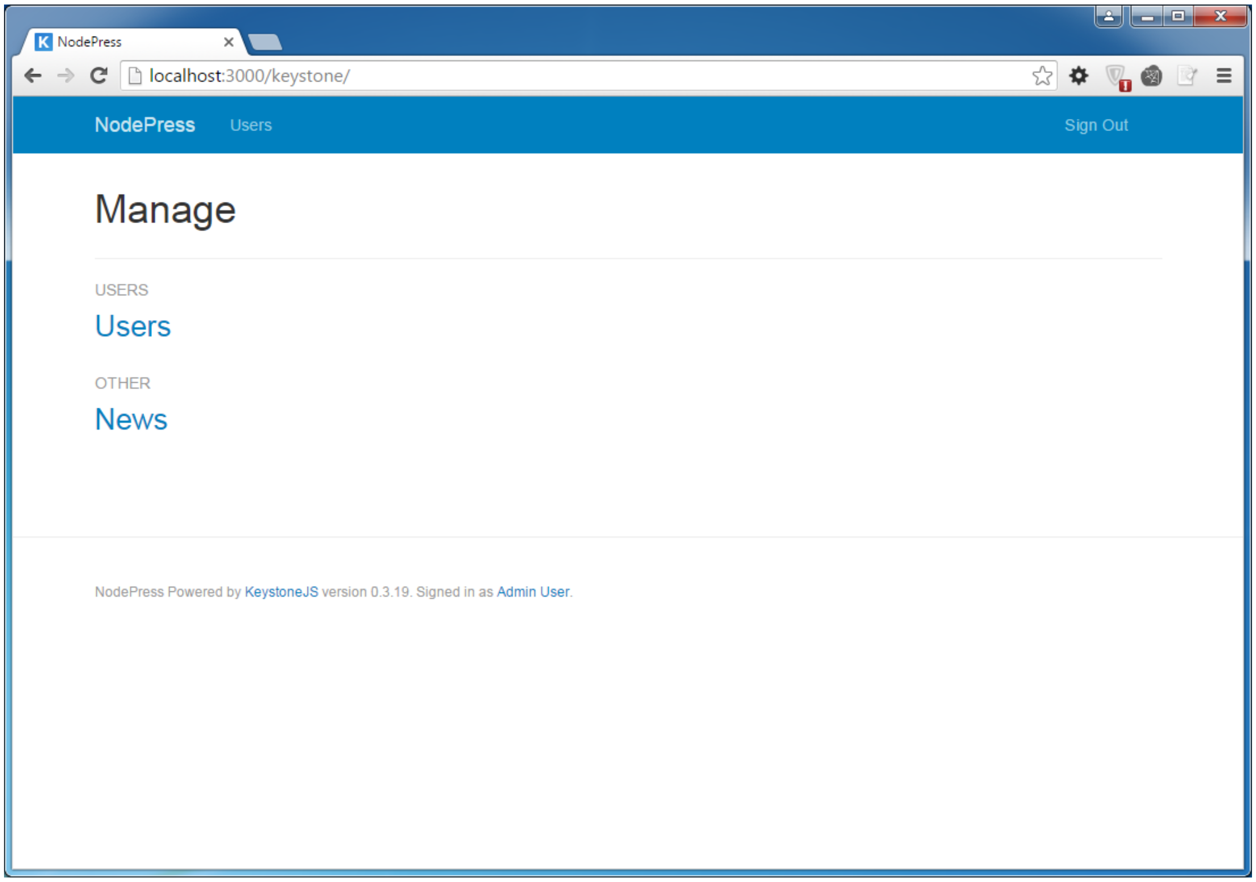
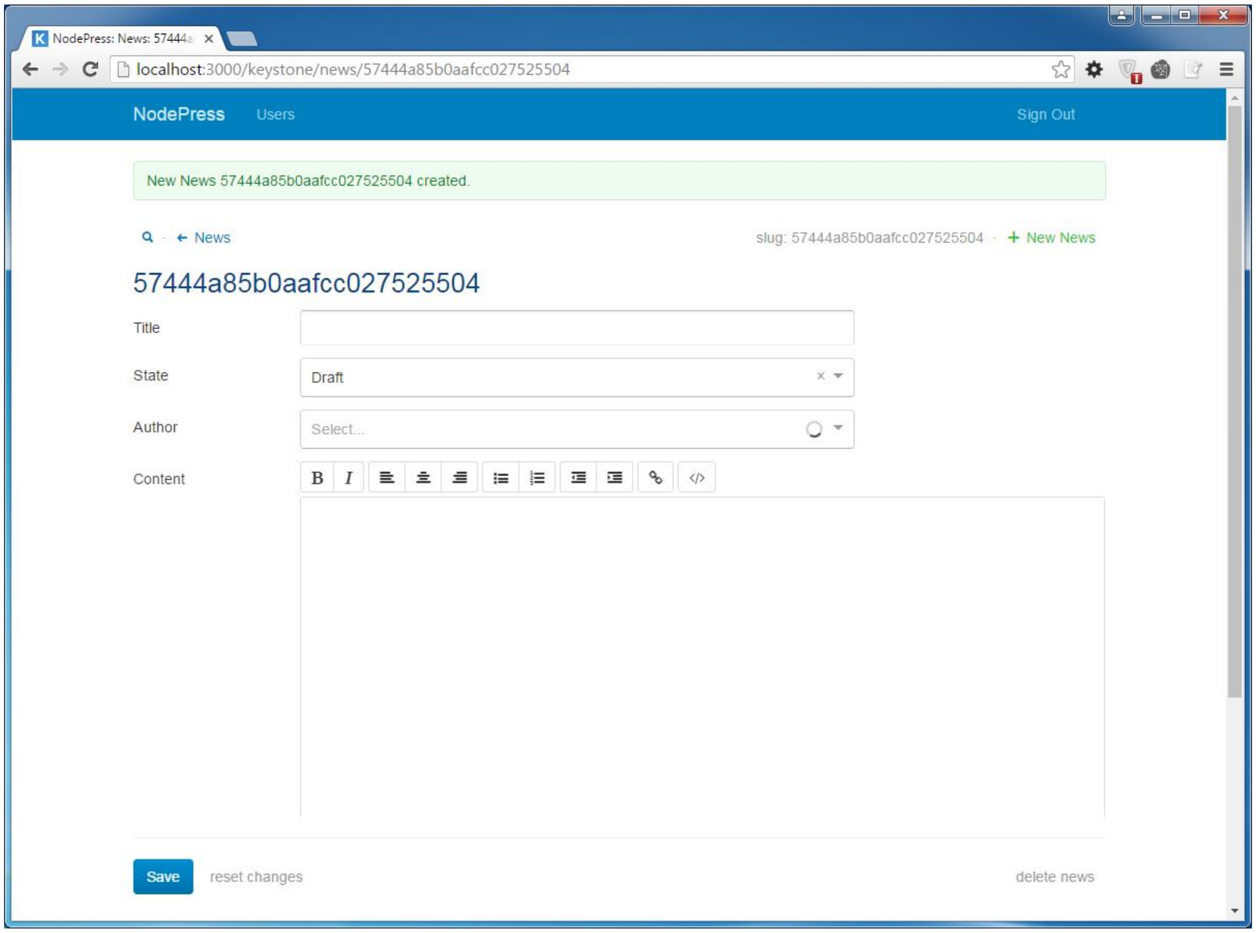


Figure 11: Manage news in administration UI

Click **News** and add a new news item with the green **Create News** button. You will be provided with an autogenerated UI with all the fields that were defined in the `News` model.



New News 57444a85b0aafcc027525504 created.

Figure 12: Manage news in administration UI

Timestamps

The **track** option in the list initialization options allows us to keep track of when and who created and last updated an item.

Code Listing 11: Specify track option on News.js model

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

/**
 * News Model
 * ==========
 */
var News = new keystone.List('News', {
    autokey: { path: 'slug', from: 'title', unique: true },
    /* Automatic change tracking */
    track: true
});
```

These fields are automatically added:

- **`CreatedAt`**: Enables tracking when the news post was created.
- **`CreatedBy`**: Enables tracking which user created the news.
- **`UpdatedAt`**: Enables tracking when the news post was last updated.
- **`UpdatedBy`**: Enables tracking which user last updated the news post.

An alternate way of registering the track functionality is to use the **`track`** property on the **`News`** list.

Code Listing 12: Specify track option on News.js model

```
News.track = true;
```

Collection names

Note that we did not tell Keystone.js which MongoDB collection to use for our **`News`** model. The plural name of the model will be used as the collection name unless another name is explicitly specified. So, in this case, Keystone.js will assume the **`News`** model stores documents in the **`News`** collection. You may specify a custom collection by defining a schema property on your model.

Code Listing 13: Specify custom collection for News.js model

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

/**
 * News Model
 * ==========
 */

var News = new keystone.List('News', {
    autokey: { path: 'slug', from: 'title', unique: true },
    track: true,
    /* custom collection name */
    schema: { collection: 'mynews' }
});

….
```

Primary keys

Keystone.js will assume that each document has a primary key column named **`_id`** that holds the MongoDB object ID. This field is generally used for querying as well as looking up related documents.

Adding relationships to model

Each model in an application can be related to another model in a couple of ways. They may be connected under a one-to-many relationship, or a many-to-many relationship.

One-to-many relationships are used when one model document can be associated with multiple documents of another single model. For instance, a user can author many news posts, and one news post can belong only to one user.

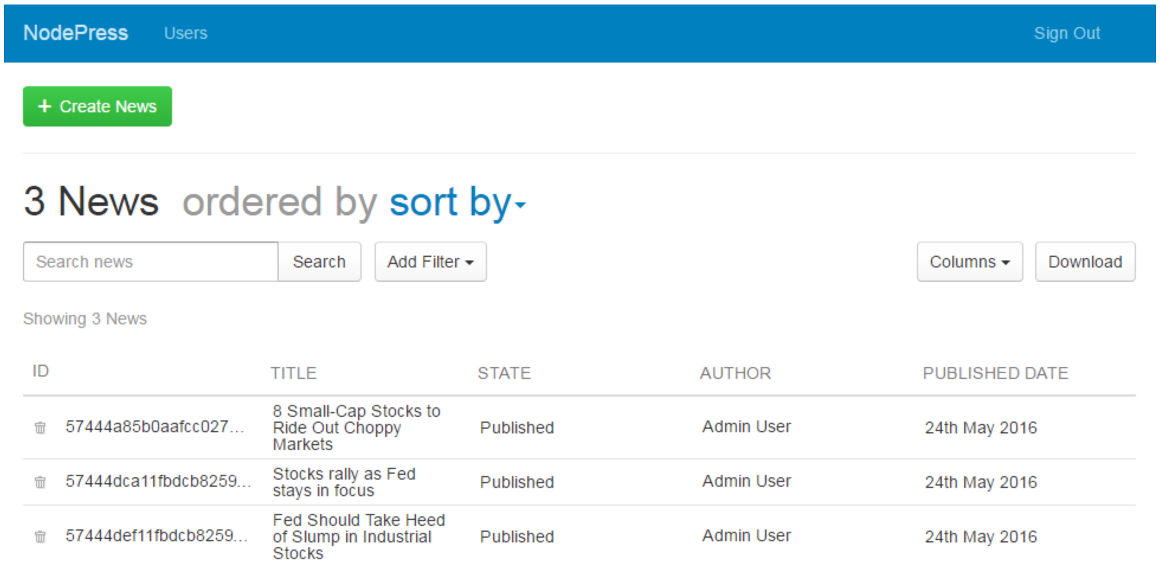
To define a one-to-many relationship, use the following code.

Code Listing 14: One-to-many relationship for the News.js model

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

/**
 * News Model
 * ==========
 */

News.add({
    author: { type: Types.Relationship, ref: 'User', index: true, many: 
false }
});
```

We have defined the relationship between a news post and a user on the **`News`** 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 the **`many`** option to **`false`** indicates that we can only select one user for this field.

Restart the application and add a few news items. The **`author`** column should be populated as per the relationship.



3 News ordered by sort by

Showing 3 News

ID	TITLE	STATE	AUTHOR	PUBLISHED DATE
57444a85b0aafcc027...	8 Small-Cap Stocks to Ride Out Choppy Markets	Published	Admin User	24th May 2016
57444dca11fbdcb8259...	Stocks rally as Fed stays in focus	Published	Admin User	24th May 2016
57444def11fbdcb8259...	Fed Should Take Heed of Slump in Industrial Stocks	Published	Admin User	24th May 2016

Figure 13: Authors related to News

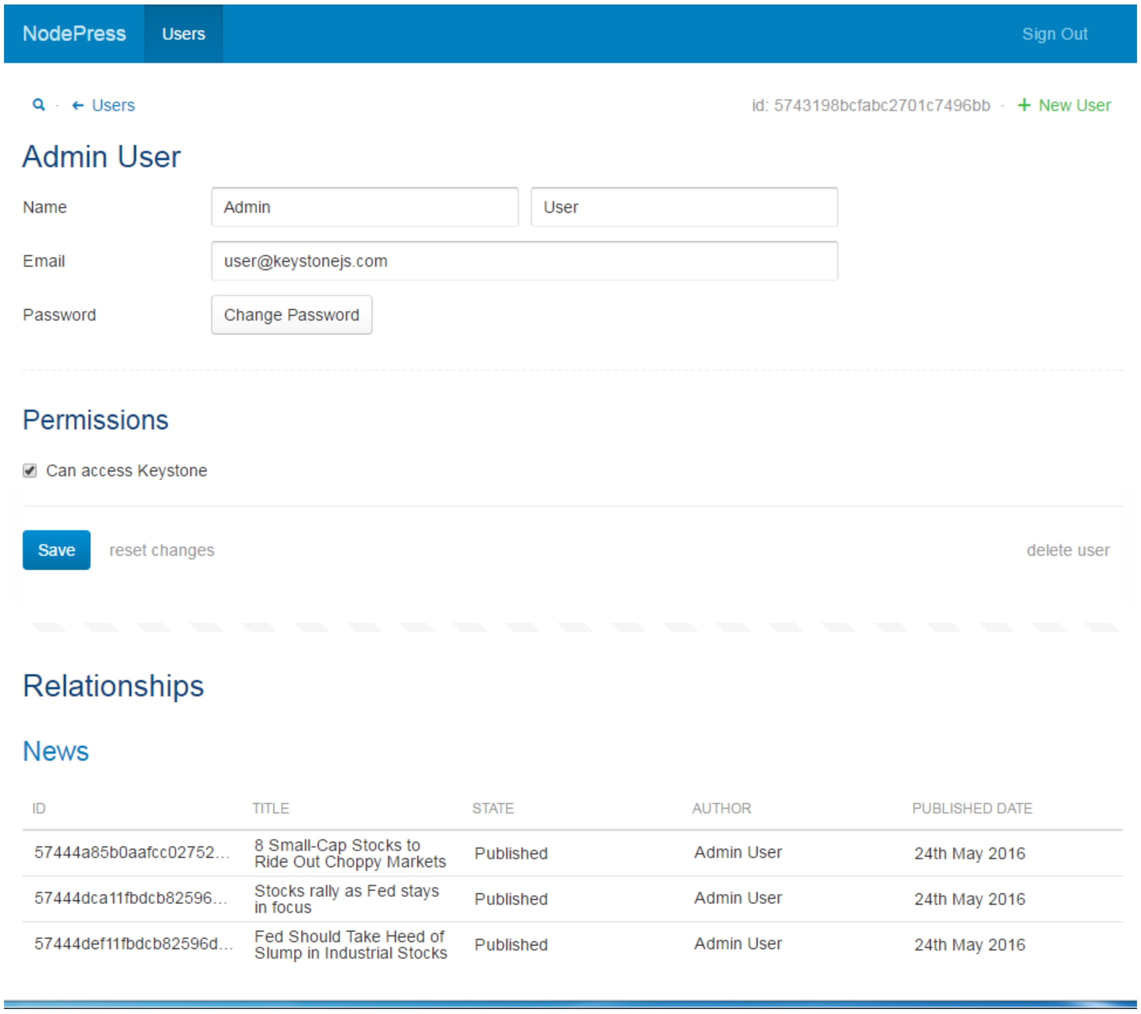
To represent the relationship from both sides, we can define the relationship on the user model as well. We can do this by calling the **`relationship`** method on the user model. Add the below line to the user model.

Code Listing 15: One-to-many relationship for the User.js model

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

/**
 * User Model
 * ==========
 */
User.relationship({ path: 'news', ref: 'News', refPath: 'author' });
```

- **`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.

Click on the admin user, and you should see the list of news articles authored by that user in the Relationships section.



id: 5743198bcfabc2701c7496bb

Admin User

Name	Admin	User
Email	user@keystonejs.com	
Password	Change Password	

Permissions

✓ Can access Keystone

Relationships

News

ID	TITLE	STATE	AUTHOR	PUBLISHED DATE
57444a85b0aafcc02752...	8 Small-Cap Stocks to Ride Out Choppy Markets	Published	Admin User	24th May 2016
57444dca11fbdcb82596...	Stocks rally as Fed stays in focus	Published	Admin User	24th May 2016
57444def11fbdcb82596d...	Fed Should Take Heed of Slump in Industrial Stocks	Published	Admin User	24th May 2016

Figure 14: News related to a single author

A many-to-many model relationship is defined as a one-to-many relationship with the exception of the **`many`** option set to **`true`**. Keystone.js provides an intuitive input tags user interface along with autosuggest to add many-to-many relationship data.

Keystone.js fields and data types

Each record stored within a MongoDB collection is referred to as a document. MongoDB supports 20 data types to store information within a document, including the following types:

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

These data types are sufficient to store raw data, but make the application very difficult to work with as it grows. Keystone.js addresses this problem by wrapping the basic data types with advanced functionality and calling them field types. There are quite a few field types available, and they are very simple to use. We have already used a few of these when we defined the **`News`** model earlier. The available field types are:

- **`Text`**
- **`Boolean`**
- **`Code`**
- **`Color`**
- **`Date`**
- **`Datetime`**
- **`Email`**
- **`Html`**
- **`Key`**
- **`Location`**
- **`Markdown`**
- **`Money`**
- **`Name`**
- **`Number`**
- **`Password`**
- **`Select`**
- **`Text`**
- **`Textarea`**
- **`Url`**
- **`AzureFile`**
- **`CloudinaryImage`**
- **`CloudinaryImages`**
- **`Embedly`**
- **`LocalFile`**
- **`S3 File`**

Some field types include helpful underscore methods, which are available on the item at the field's name preceded by an underscore. For example: use the format underscore method of the **`publishedDate DateTime`** field of the **`News`** model like in Code Listing 16.

Code Listing 16: Underscore method in Keystone.js field

```
console.log(news._.publishedDate.format('Do MMMM YYYY')); // 25th May
2016
```

Keystone.js model extensions

Virtual properties

Virtual properties allow us to format the data in fields when retrieving them from a model or setting their value. A virtual property is added to the underlying Mongoose schema. Let us add a virtual property that returns the year in which a news post was published.

Code Listing 17: Define virtual property on News model

```
News.schema.virtual('publishedYear').get(function () {
    return this._.publishedDate.format('YYYY')
});
```

The advantage of virtual properties is that they are not persisted to the document saved within MongoDB, yet are available on the document retrieved as a result of the query. The **`virtual`** property can be used similarly to a regularly defined property, as shown in Code Listing 18.

Code Listing 18: Display a virtual property

```
console.log(newsItem.publishedYear);
```

Virtual methods

Virtual methods are similar to virtual properties and are added to the schema of the list. These methods can be invoked from the templates if necessary. A good example is a method that can return a well-formed URI to a news item.

Code Listing 19: Define virtual method on News model

```
News.schema.methods.url = function () {
    return '/newsdetail/' + this.slug;
};
```

Pre and Post hooks

Keystone.js lists leverage the underlying Mongoose **`pre`** and **`post`** middleware. These are methods that are defined on the model and are automatically invoked before or after a certain operation, by the framework. A common example would be the **`pre`** and **`post save`** hooks, which are used to manipulate the data in the model before it is saved to the MongoDB collection.

For example, in our **`News`** model, we might want to automatically set the **`publishedDate`** value when the state is changed to published (but only if it hasn't already been set).

We might also want to add a method to check whether the post is published, rather than checking the **`state`** field value directly.

Before calling **`News.register()`**, we would add the following code.

Code Listing 20: Pre-save hook

```
News.schema.methods.isPublished = function () {
    return this.state == 'published';
}

News.schema.pre('save', function (next) {
    if (this.isModified('state') && this.isPublished() &&
!this.publishedDate) {
        this.publishedDate = new Date();
    }
    next();
});
```

Keystone.js model queries

To query data, we can use any of the Mongoose query methods on the Keystone.js model. Let us look at the queries that will be used in views (coming up in the next chapters).

Retrieving all news items

To fetch all news items, we can use the **`find`** method.

Code Listing 21: Find method

```
var q = keystone.list('News').model.find();

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

Retrieving a news item by slug:

To fetch a news item that matches the **`slug`**, we can use the **`findOne`** method shown in Code Listing 22. The **`slug`** can be read from the **`req.params`** collection.

Code Listing 22: FindOne method with filter

```
var q = keystone.list('News').model.findOne({'slug':req.params.slug})

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

Selecting specific fields

For optimal performance, it is always advised to construct queries that retrieve only the necessary data.

Code Listing 23: Select method

```
var q = keystone.list('News').model
```

```
    .findOne({'slug':req.params.slug})
    .select('title status author');

q.exec(function(err, result) {
    var newsitem = result;
    next(err);
});
```

Counting results

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

Code Listing 24: Count method

```
var q = keystone.list('News').model.count();

q.exec(function(err, count) {
   console.log('There are %d news items', count);
   next(err);
});
```

Ordering results

The **`sort`** method can be used in conjunction with the **`find`** method to order results of a query. The following example will retrieve all news, ordered by title.

Code Listing 25: Sort method

```
var q = keystone.list('News').model.find().sort('title');

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

By default, the results are sorted in ascending order. This default behavior can be reversed by prefixing a minus sign to the field that is being used to sort.

Code Listing 26: Reverse sort method

```
var q = keystone.list('News').model.sort('-title');

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

Filtering results

The **where** method can be used to conditionally find documents with attributes that we are interested in. Multiple **where** clauses can also be chained together. In the following example, let us try to retrieve news items that have status as **published**.

Code Listing 27: Where clause

```
var q = keystone.list('News').model
    .where('state').equals('published');

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

Limiting returned results

To retrieve a small subset of documents, for instance, the ten most recently added news items, we can do so using the **limit** method.

Code Listing 28: Limit clause

```
var q = keystone.list('News').model
    .limit(10);

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

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

Code Listing 29: Limit clause with skip

```
var q = keystone.list('News').model.skip(6)
    .limit(10);

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

Test 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 news items that are empty (that is, where the **`content`** field does not exist on the News MongoDB document), use the following statements.

Code Listing 30: Exists clause

```
var q = keystone.list('News').model
    .where('content')
    .exists(false);

q.exec(function(err, results) {
   var news = results; //list of news with missing content
   next(err);
});
```

Inserting a document programmatically

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.

Code Listing 31: Insert a new document

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


var newItem = new News.model();

newItem.title = 'Credit Suisse, Leader in Global Cleared Derivatives';
newItem.status = 'Published';
newItem.description = 'The FIS Derivatives Utility was designed to help
global capital markets firms better adapt to market challenges by
enabling market participants.';

newItem.save(function (err) {
    if (err) {
        console.error("Error adding News to the database:");
        console.error(err);
    } else {
        console.log("Added news “ + newItem.title + " to the database.");
    }
    done(err);
});
```

Upon saving, the new **`News`** items will have a unique **`slug`** generated based on the title because the autokeyoption was set on the model.

Updating an existing document programmatically

To update existing documents, we can leverage the Keystone.js **`UpdateHandler`** functionality. This process typically involves retrieving the desired document using its identifier, setting the changed fields on the document, and requesting Keystone.js to process the updates.

Code Listing 32: Update an existing document

```
var q = keystone.list('News').model.findOne({'slug':req.params.slug})

q.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 news item");

    });
});
```

Let us assume we receive the **`slug`** of a news item via a form post along with the changes. Code Listing 32 first retrieves a news item that matches the provided **`slug`**. If the item is found, any matching fields and their values (from the form post) are set to the data object. The **`getUpdateHandler`** method on the matching news item 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.

Deleting a document programmatically

To delete a document, first locate the document, and then use the **`remove`** method.

Code Listing 33: Delete a document

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


var q = keystone.list('News').model.findOne({ 'slug': req.params.slug })
            .remove(function (err) {
                if (err) return res.apiError('database error', err);
                console.log("Successfully deleted the news item");
            });
```

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 Keystone.js makes it a breeze to implement.

Chapter 4 Templating with Swig

Swig is a simple, powerful, and extendable JavaScript template engine. A template engine is typically used to display data that has been returned from a query to the database. The template engine combines data and markup to generate an output that can be rendered by the browser. Swig's syntax is very similar to many existing template engines such as Jinja2 and Django, used by other programming languages. If you are familiar with any of these technologies, you should find Swig really simple.

Some of the advantages of using Swig are that it:

- Has an object-oriented template inheritance.
- Applies filters and transformations to output in your templates.
- Automatically escapes all output for safe HTML rendering.
- Supports lots of iteration and conditionals.
- Is robust, extendable, and customizable.

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

Code Listing 34: Set up view engine

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

All templates reside within **/templates/views** folder within the application.

Basic template operations

In a template language, variables passed to the template are replaced at predefined locations in the template. In Swig, variable substitutions are defined by **`{{ }}`**. There are also control blocks defined by **`{% %}`**, which declare language functions, such as loops or **`if`** statements.

To render the title of a news item, use the following notation.

Code Listing 35: Render a variable

```
{{ news.title }}
```

Swig follows the same rules as JavaScript. If a key includes non-alphanumeric characters, it must be accessed using bracket-notation, not dot-notation.

We can also invoke functions and render the output as if we were rendering a variable. If you recall, we introduced a virtual method named **`url`** in the previous chapter (see Code Listing 19).

We can invoke the **`url`** method from the template and render a URL to the detail page for the news item.

Code Listing 36: Invoke a method from template

```
<a href='{{ news.url() }}'> {{news.title}} </a>
```

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.

Code Listing 37: Output code comments

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

Loops and control structures

Swig also provides convenient syntaxes for common control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with control structures, while also remaining similar to their JavaScript counterparts.

If statements

You may construct **`if`** statements using the **`if`**, **`elif`**, **`else`**, and **`endif`** directives.

Code Listing 38: If conditionals

```
{% if length(newsitems) > 10 %}
    I have more than 10 records
{% elif length(newsitems) < 5 %}
    I have less than 5 records!
{% else %}
    I don't have any records!
{% endif %}
```

Boolean operators like **`and`** and **`or`** can be used within logic tags. Code Listing 39 is an example illustrating the use of the **`and`** conditionals.

Code Listing 39: Boolean operators

```
{% if length(news.title) > 10 and length(news.content) > 10 %}
    {{news.title}} can be displayed.
{% endif %}
```

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

Code Listing 40: Use JS functions in conditionals

```
{% if news.title.indexOf("critical") > -1 %}
    This is a critical news item
{% endif %}
```

Loop statements:

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

Code Listing 41: Loop statements

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

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

Code Listing 42: Loop control helpers

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

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:

Code Listing 43: Reverse loop

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

Swig built-in filters

Filters are methods through which output can be manipulated before rendering. 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 convert the title of a news item to title case and strip any HTML tags that might have been input, we can use the **`title`** and **`striptags`** filters.

Code Listing 44: Use filters in templates

```
<div>
    News Title - {{newsItem.title | title | striptags}}
</div>
```

Here is a list of the 23 available filters in Swig:

- **`capitalize`:** Capitalizes words in the input.
- **`lower`:** Converts an input string to lowercase.
- **`upper`:** Converts an input string to uppercase.
- **`title`:** Capitalizes every word given and lower-cases 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`:** Strips HTML from string.
- **`safe`:** Forces the input to not be auto-escaped. Swig escapes data by default.
- **`replace`:** Replaces each occurrence of a string.
- **`escape`:** Escapes special characters in a string.
- **`addslashes`:** Adds backslashes to characters that need to be escaped.
- **`url_encode`:** URL-encodes a string.
- **`url_decode`:** URL-decodes a string.
- **`first`:** Gets the first element of the input array, object, or string.
- **`last`:** Gets the last element of the input array, object, or string.
- **`reverse`:** Reverse-sorts the input values.
- **`sort`:** Sorts the input in an ascending order.
- **`join`:** Joins elements of the array with a delimiter.
- **`groupBy`:** Groups an array of objects by a key.
- **`uniq`:** Removes all duplicate elements from an array.
- **`length`:** Gets the number of items in an array.
- **`raw`:** Similar to **`safe`**; prevents data from being auto-escaped.

Template inheritance

Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Swig template and use some kind of template inheritance/injection to be able to render specific partial templates. Swig makes this easy with extends and block directives.

Create a template named **layout.swig** and save it under the **/templates/layouts** folder with the following content.

Code Listing 45: Base layout template

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

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

Next, to use the base template in another template, create a template named **news.swig** and save it in the **templates/views** directory with the following content.

Code Listing 46: Extend base layout template

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

{% block title %}News Detail Page{% endblock %}

{% block content %}
<p>We will display the news details here</p>
{% endblock %}
```

Template partials

Templates can easily get bulky and difficult to maintain if we do not organize the contents in a good manner. An easy way to organize the different sections of a template is to use template partials. Template partials are pieces of templates that reside in separate files and are combined together to make a single template.

For example, the preceding news layout template can leverage multiple partials that are concerned with displaying the header and footer of the pages.

Code Listing 47: Header template

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

  {% block head %}
  <link rel="stylesheet" href="main.css">
  {% endblock %}
</head>
```

Code Listing 48: Footer template

```
<footer>
&copy; nodepress 2016. All rights reserved.
</footer>
</html>
```

Code Listing 49: Updated base template

```
{% include 'header.swig' %}

<body>
  {% block content %}{% endblock %}
</body>

{% include 'footer.swig' %}
```

Another recommended scenario in which to use template partials would be within loops. The code will be much easier to understand and maintain.

Code Listing 50: Template partial in loops

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

Macros

A **`macro`** is a function in Swig that returns a template or HTML string. This is used to avoid code that is repeated over and over again and reduce it to one function call. For example, the following is a **`macro`** to show a news item.

Code Listing 51: Macro in template

```
{% macro showNews(news) %}
<div class="post">
    <h2>
```

```
        <a href="{{ news.url() }}">{{ news.title }}</a>
    </h2>
    <p>Posted
{% if news.publishedDate %}
            <br>on {{ news._.publishedDate.format("MMMM Do, YYYY") }}
{% endif %}
</p>
</div>
{% endmacro %}
```

Now, to quickly display a news item in any template, call your **`macro`** using the following.

Code Listing 52: Invoke macro

```
{{ showNews(newsObj) }}
```

Summary

In this chapter, you learned about Swig, the powerful and flexible templating option for Node.js. There are other similar frameworks, and I would suggest you play around with a few to find the one that best fits your style of coding.

Chapter 5 Working with Views

Up until this point, we have seen how to retrieve data from the database (models) and how to use templates to display the data on the browser. The missing piece is the view, which is the link between the model and the templates. Views are responsible for handling data transmission to and from the template and the database.

Views are JavaScript modules and reside in a folder at **/routes/views** within the application.

Defining URL routes

Routes can be thought of as URL schemas that describe the mechanism for making requests to a web application. Routes are a combination of an HTTP request method (a.k.a. HTTP verb) and a path pattern used to define URLs in your application. Views are executed as a result of a route being serviced by the framework.

Routes are defined in a file residing at **/routes/index.js**.

Code Listing 53: Simple GET route

```
app.get('/', function (req, res) {
    res.send('welcome to nodepress');
});
```

Code Listing 53 illustrates a simple route. The route defines that the inline function should be executed as a result of receiving a request at the root/route in the application. The HTTP request should be an **`HTTP GET`**. The executed function receives both the request and response context objects as input. In the example, the string '**`welcome to nodepress`**' is output on the screen as part of the response.

Keystone.js supports the following routing methods that correspond to HTTP methods:

- **`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 most commonly used HTTP verbs are **`GET`** and **`POST`**. **`POST`** is used to send data to the server from web forms. The following example illustrates the syntax for a simple **`POST`** route.

Code Listing 54: Simple POST route

```
app.post('/addnews', function (req, res) {
    res.send('Received post to nodepress at addnews route');
});
```

There is a special routing method, **`app.all()`**, which is not derived from any HTTP method but can be used to respond to incoming requests on any HTTP verb.

Code Listing 55: Catch all route

```
app.all('/', function (req, res, next) {
    console.log('Will respond to all requests ...');
    next(); // pass control to the next handler
});
```

Building the list view

A list view is used to display a list of items with minimal information and links to a detailed view to see the data in depth. Let us see how to display a list of news items from our MongoDB database.

Add the following route to the application.

Code Listing 56: List view route

```
app.get('/news', routes.views.newslist);
```

Next, define the **`newslist`** view in **/routes/views/newslist.js** with the following code.

Code Listing 57: List view

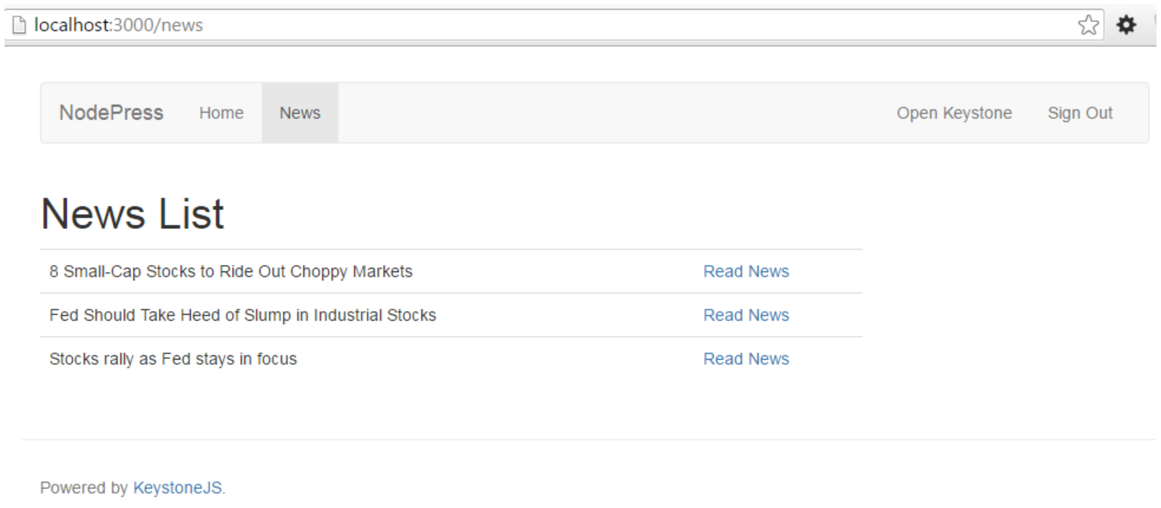
```
var keystone = require('keystone');
var async = require('async');

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

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

    // Init locals
    locals.section = 'news';

    locals.data = {
        news: []
    };

    // Load all news
    view.on('init', function (next) {

    keystone.list('News').model.find().sort('title').exec(function (err,
results)  {

            if (err || !results.length) {
                return next(err);
            }

            locals.data.news = results;
            next(err);
        });

    });

    // Render the template
    view.render('newslist');

};
```

The view is an instance of the **`keystone.View`** object. The view queries the database for all news posts in a sorted manner. The news items are stored in an array named **`data.news`**. At the end of the view, we invoke the **`render`** method and pass in the name of the template. This template will be combined with the data from the view and sent back to the browser to be rendered.

Code Listing 58: List view template

```
{% extends "../layouts/default.swig" %}
{% block intro %}
      <div class="container">
            <h1>News List</h1>
      </div>
{% endblock %}

{% block content %}
      <div class="container">
            <div class="row">
                  <div class="col-sm-8 col-md-9">

      {% if data.news.length %}

            <div class="news">
            <table class="table">
            {% for news in data.news %}
            <tr>
            <td>{{news.title}}</td><td><a href='{{news.url()}}'>Read
News</a></td>
             </tr>
            {% endfor %}
            </table>
            </div>

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

The template uses the Swig **`for`** loop to render the news posts. The rendered output will look like that shown in Figure 15.


localhost:3000/news

News List

8 Small-Cap Stocks to Ride Out Choppy Markets	Read News
Fed Should Take Heed of Slump in Industrial Stocks	Read News
Stocks rally as Fed stays in focus	Read News

Powered by KeystoneJS.

Figure 15: List of news

Building the detail view

Let us see how to use a detail view to display the details of a news post. Start off by creating a route for the news details.

Add the following route to the application.

Code Listing 59: List view route

```
app.get('/newsdetail/:slug', routes.views.newsdetail);
```

The **`slug`** is specified as a parameter in the URL.

Next, define the **`newsdetail`** view in **/routes/views/newsdetail.js** with the following code.

Code Listing 60: Details view

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

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

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

    // Set locals
    locals.section = 'news';
    locals.filters = {
        slug: req.params.slug
    };
    locals.data = {
        news: ''
    };

    // Load the current news
    view.on('init', function (next) {

        var q = keystone.list('News').model.findOne({
            state: 'published',
            slug: locals.filters.slug
        }).populate('author');

        q.exec(function (err, result) {
            locals.data.news = result;
            next(err);
        });

    });


    // Render the template
    view.render('newsdetail');

};
```

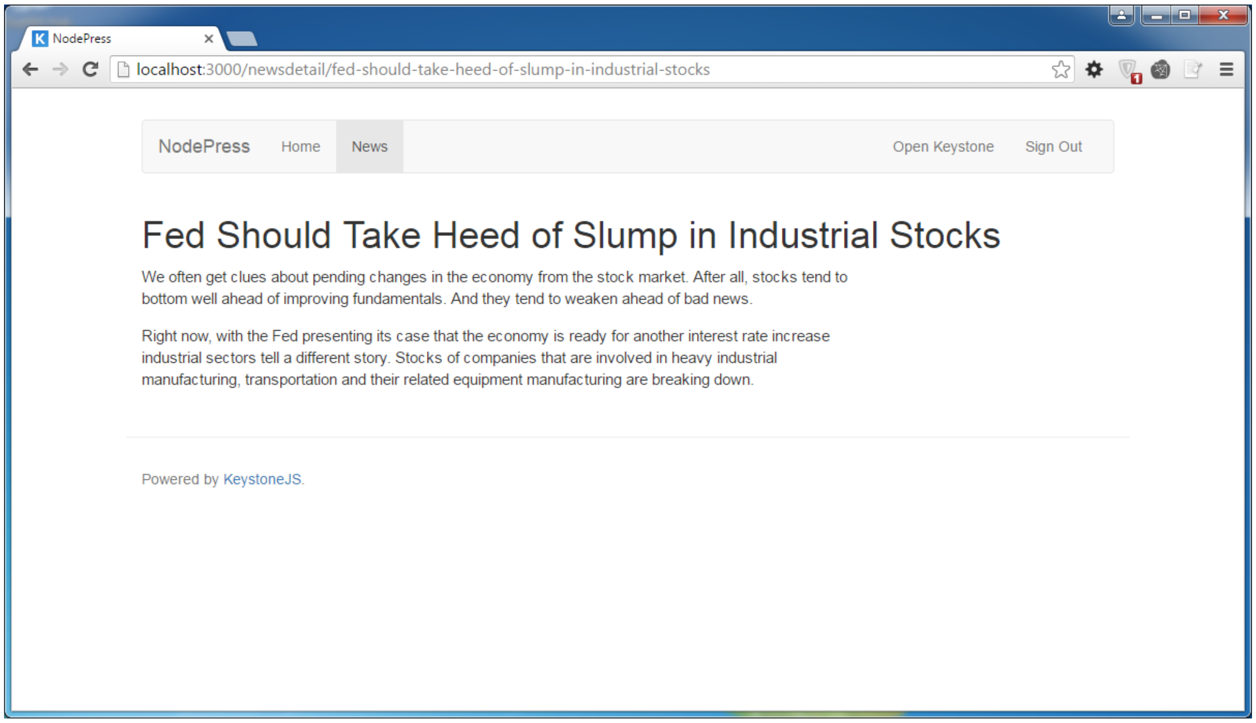
Create a template named **newsdetail.swig** under the **/templates/views** folder with the following content.

Code Listing 61: Details template

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

{% block intro %}
      <div class="container">
            <h1>{{data.news.title}}</h1>
      </div>
{% endblock %}

{% block content %}
      <div class="container">
            <div class="row">
                  <div class="col-sm-8 col-md-9">

                              {% if data.news.content %}

            <div class="news">
            {% autoescape false %} {{data.news.content}} {% endautoescape
%}
```

```
            </div>

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

When the **`autoescape`** property is set to **`false`**, HTML will be displayed as it is rather than being escaped as HTML entities. The output will look like Figure 16.



NodePress
localhost:3000/newsdetail/fed-should-take-heed-of-slump-in-industrial-stocks
NodePress
Home
News
Open Keystone
Sign Out

Fed Should Take Heed of Slump in Industrial Stocks

We often get clues about pending changes in the economy from the stock market. After all, stocks tend to bottom well ahead of improving fundamentals. And they tend to weaken ahead of bad news.

Right now, with the Fed presenting its case that the economy is ready for another interest rate increase industrial sectors tell a different story. Stocks of companies that are involved in heavy industrial manufacturing, transportation and their related equipment manufacturing are breaking down.

Powered by KeystoneJS.

Figure 16: News detail

Adding pagination links

Web applications generally need to display a large amount of data, and using pagination is a way of displaying chunks of data followed by links to load more data if needed.

Because pagination links are a feature that is used throughout the application, let's create the pagination included in our app's template directory.

Create a new template file in **/templates/views** named **page_links.swig**. Keystone.js pagination returns us an object from which we can determine, in the template, what page we are on, and how many pages there are in total.

Code Listing 62: Pagination links

```
{% if data.news.totalPages > 1 %}
<ul class="pagination">
{% if data.news.previous %}
      <li>
            <a href="?page={{ data.news.previous }}">
                  <span class="glyphicon glyphicon-chevron-left"></span>
            </a>
      </li>
{% else %}
      <li class="disabled">
            <a href="?page=1">
                  <span class="glyphicon glyphicon-chevron-left"></span>
            </a>
      </li>
{% endif %}
      {% for p in data.news.pages %}
            <li class="{% if data.news.currentPage == p %}active{% endif
%}">
                  <a href="?page={% if p == "..." %}{% if i %}{{
data.news.totalPages }}{% else %}1{% endif %}{% else %}{{ p }}{% endif
%}">{{ p }}</a>
            </li>
      {% endfor %}
{% if data.news.next %}
      <li>
            <a href="?page={{ data.news.next }}">
                  <span class="glyphicon glyphicon-chevron-right"></span>
            </a>
      </li>
{% else %}
      <li class="disabled">
            <a href="?page={{ data.news.totalPages }}">
                  <span class="glyphicon glyphicon-chevron-right"></span>
            </a>
      </li>
{% endif %}
</ul>
{% endif %}
```

Keystone.js provides a very easy API to fetch data from MongoDB in a paginated manner with just a few lines of code. Let's update the **`newslist.js`** view to fetch news in a paginated object.

Code Listing 63: Pagination view

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

exports = module.exports = function (req, res) {
    var view = new keystone.View(req, res);
    var locals = res.locals;

    // Set locals
    locals.section = 'news';
    locals.filters = {
        slug: req.params.slug
    };
    locals.data = {
        news: ''
    };

    // Load the current news
    view.on('init', function (next) {

        var q = keystone.list('News').model.findOne({
            state: 'published',
            slug: locals.filters.slug
        }).populate('author');

        q.exec(function (err, result) {
            locals.data.news = result;
            next(err);
        });

    });

    // Render the view
    view.render('newsdetail');

};
```

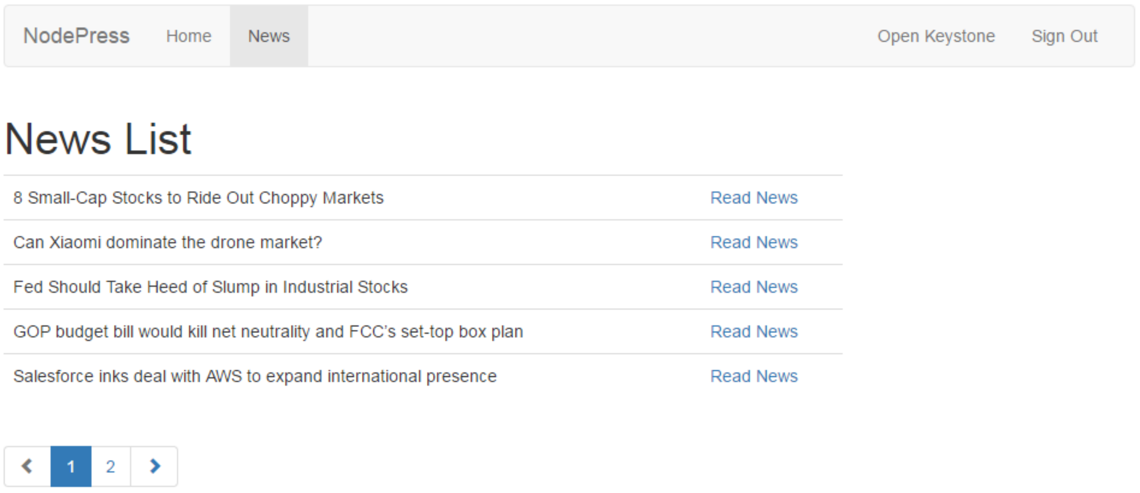
Next, update the **`newslist.swig`** to display the paginated links.

Code Listing 64: Pagination template

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

{% block intro %}
<div class="container">
<h1>News List</h1>
</div>
{% endblock %}

{% block content %}
<div class="container">
```

```
<div class="row">
<div class="col-sm-8 col-md-9">

{% if data.news.results.length %}

<div class="news">
      <table class="table">
      {% for news in data.news.results %}
      <tr>
            <td>{{news.title}}</td><td><a href='{{news.url()}}'>Read
News</a></td>
      </tr>
      {% endfor %}
      </table>
</div>
{% include 'page_links.swig' %}
{% endif %}
</div>
</div>
</div>
{% endblock %}
```

Restart the application, and the pagination links should appear as shown in the following figure.



8 Small-Cap Stocks to Ride Out Choppy Markets	Read News
Can Xiaomi dominate the drone market?	Read News
Fed Should Take Heed of Slump in Industrial Stocks	Read News
GOP budget bill would kill net neutrality and FCC's set-top box plan	Read News
Salesforce inks deal with AWS to expand international presence	Read News

Figure 17: Pagination links

The Keystone.js pagination object returns a lot of useful metadata along with the results:

- **`total`:** All matching results (not just on this page).
- **`results`:** Array of results for this page.
- **`currentPage`:** Index of the current page.
- **`totalPages`:** 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:`** Index of the first result included.
- **`last:`** Index of the last result included.

Serving static files

The **/public** directory holds static content related to the web application, such as images, CSS, fonts, and JavaScript. The **`LESS`** processor included within the framework will make sure that **`LESS`** files associated with the web application are compiled to CSS files during runtime.

Since Keystone.js uses the **`express.static`** built-in middleware function in Express to serve static assets, we reference assets as if they resided in the root of the application, as shown in Code Listing 65.

Code Listing 65: Link to CSS

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

The styles folder must reside within the **/public** folder.

Summary

In this chapter, we were able to understand the link among data models, views, templates, and URLs in your application, including object pagination.

Chapter 6 Forms and Validation

Let us see how to use forms to modify the contents of **`nodepress`** directly. We will create forms for working with the **`News`** model, learn how to receive and validate user data, and finally update the values in the database.

Handling form submissions

Let us create a form where an authenticated user can create a new **`News`** item. We will create a form that will be posted to the server. Create a new template named **createnews.swig** to hold our web form, place it in the **/templates/views/** folder, and rebuild the project.

Code Listing 66: Create News form

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

{% block content %}

<div class="container">
      <div class="panel panel-primary">
    <!-- Default panel contents -->
    <div class="panel-heading">Create News Item</div>
    <div class="panel-body">
         <form class="form-horizontal custom-form" action="/createnews"
method="post">
                <div class="form-group">
                    <div class="col-sm-2 label-column">
                        <label for="name-input-field" class="control-
label">Title </label>
                    </div>
                    <div class="col-sm-6 input-column {% if
validationErrors.title %}has-error{% endif %}">
                        <input type="text" name="title"
placeholder="Title of the News" class="form-control"
value="{{form.title}}" />
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-sm-2 label-column">
                        <label for="email-input-field" class="control-
label">Content </label>
                    </div>
                    <div class="col-sm-6 input-column" {% if
validationErrors.content %}has-error{% endif %}>
                        <textarea name="description"
```

```
placeholder="Describe the News" class="form-
control">{{form.content}}</textarea>
                    </div>
                </div>
                <div class="form-group">
                    <div class="col-sm-2 label-column">
                        <label for="" class="control-label">State
</label>
                    </div>
                    <div class="col-sm-6 input-column">
                        <select class="form-control" name="state">
                                <option value="draft" {% if form.state ==
'draft' %} selected{% endif %}>Draft</option>
                                <option value="published" {% if
form.state == 'published' %} selected{% endif %}>Published</option>
                                <option value="archived" {% if form.state
== 'archived' %} selected{% endif %}>Archived</option>
                        </select>
                    </div>
                </div>


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

The form will look like the following figure.

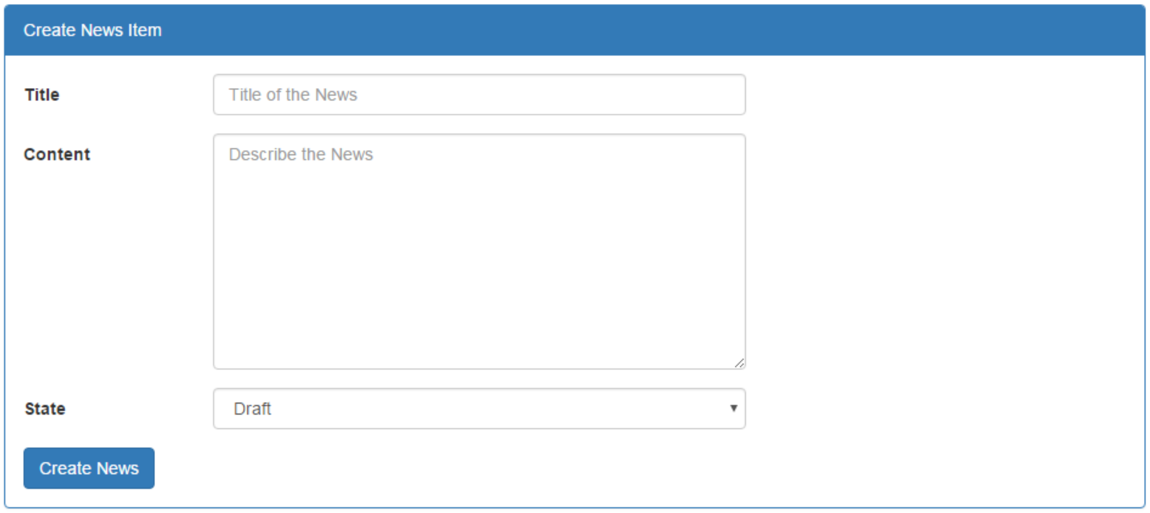


Figure 18: Create News form

The form is pretty straightforward. We have HTML form fields for each of the model fields on the **`News`** model.

Add a route to the routes index file.

Code Listing 67: Create news route

```
app.all('/createnews', middleware.requireUser, routes.views.createnews);
```

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 news item.

Validating input and error messages

Our view will perform all the input validations and will populate the **`validationErrors`** in the response, if any. Create a file for our view named **createnews.js** under the **/routes/views** folder.

Code Listing 68: Create News view

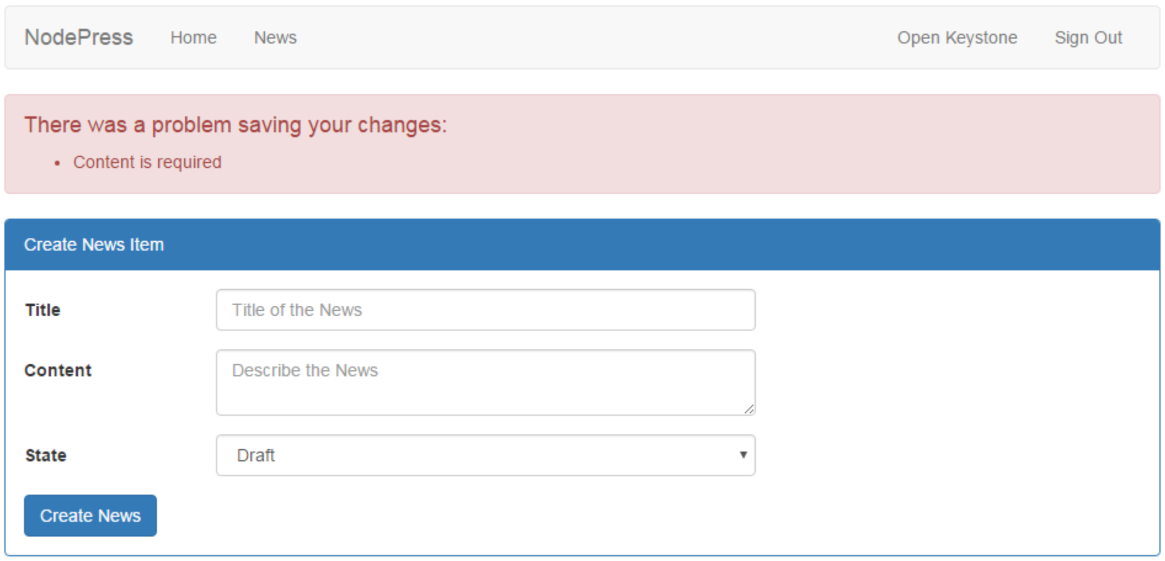
```
var keystone = require('keystone'),
    News = keystone.list('News');

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 newNews = new News.model(),
            data = req.body;

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

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

        });

    });

    // Render the view
    view.render('createnews');

};
```

The **`getUpdateHandler`** method will perform validation based on the model definition. We can use the following snippet to highlight any fields that failed validation by applying the bootstrap **`has-error`** CSS class in our template.

Code Listing 69: Bootstrap error CSS

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

Flash errors are used to show an aggregate of all the errors that occur. The following figure shows an example of flash errors.


NodePress Home News Open Keystone Sign Out

There was a problem saving your changes:

- Content is required

Figure 19: Flash errors

To receive the form data on the server side, we use the request body on the HTTP post. The **`getUpdateHandler`** method on an instance of the model can take in the post data and create a new entry in the database. It is important that the object keys in the input data read from the form post match up to the fields defined in the model.

Summary

In this chapter, we learned about the ease of integrating forms within a Keystone.js application. Keystone.js form validation makes it very convenient to implement complex forms in any application.

Chapter 7 Authenticating Users

Most dynamic web applications allow for some kind of user authentication and preference-saving functionality. Let us look at how to allow users to create an account and log in and out from our Keystone.js application.

Sessions & configuration

Sessions are used to track user activity on the server side and on the client side via cookies. They are generally used to save pieces of authentication data and user preferences. Sessions can either be stored in memory or can be persisted to storage such as MongoDB or Redis. Keystone.js supports storing sessions in MongoDB via the **`connect-mongo`** library. We can also save sessions to in-memory, Redis, Memcached, or a custom session store that we can implement.

There are a few configuration options that need to be set before using the session functionality. These options should be set in the **`keystone.init()`** function within the **keystone.js** file. The configuration options are:

- **`session`:** Set this option to **`true`** if you want your application to support session management.
- **`auth`:** This option indicates whether to enable built-in authentication for keystone's Admin UI, or a custom function to use to authenticate users.
- **`user model`:** This option indicates to Keystone.js which model will be used to maintain the user information.
- **`cookie secret`:** Use this option to specify the encryption key to use for your cookies.
- **`session store`:** This identifies which session storage option to use (in-memory, mongo, etc.).

To use MongoDB as the session store, we need to install **`connect-mongo`** as shown.

Code Listing 70: Install connect-mongo

```
npm install connect-mongo –save
```

After enabling the session-based authentication options, let us define routes that will be used for authentication. Add the following routes to the route index file.

Code Listing 71: Authentication routes

```
app.all('/join', routes.views.join);
app.all('/signin', routes.views.signin);
app.get('/signout', routes.views.signout);
```

Create an account

The first step in allowing users to create an account is to display the registration form. Create a file named **join.swig** in the **templates/views** folder with the following content.

Code Listing 72: Registration form markup

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

{% block content %}
<div class="container">
    <div class="panel panel-primary">
    <div class="panel-heading">Create An Account</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 name" 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" name="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" name="email" type="email" value="{{form.email}}">
          </div>
        </div>
            <div class="form-group required">
          <label class="col-md-4 control-label">Password*</label>
          <div class="col-md-8">
            <input class="form-control" id="password" name="password"
```

```
placeholder="password" type="password">
          </div>
        </div>
            <div class="form-group">
          <label class="col-md-4 control-label"></label>
          <div class="col-md-8">

            <div style="clear:both"></div>
                <button class="btn btn-primary"
type="submit">Join</button>
                </div>
        </div>
      </fieldset>
    </form>
    </div>
 </div>
  </div>
   </div>
 {% endblock %}
```

The rendered markup will look like the following figure.

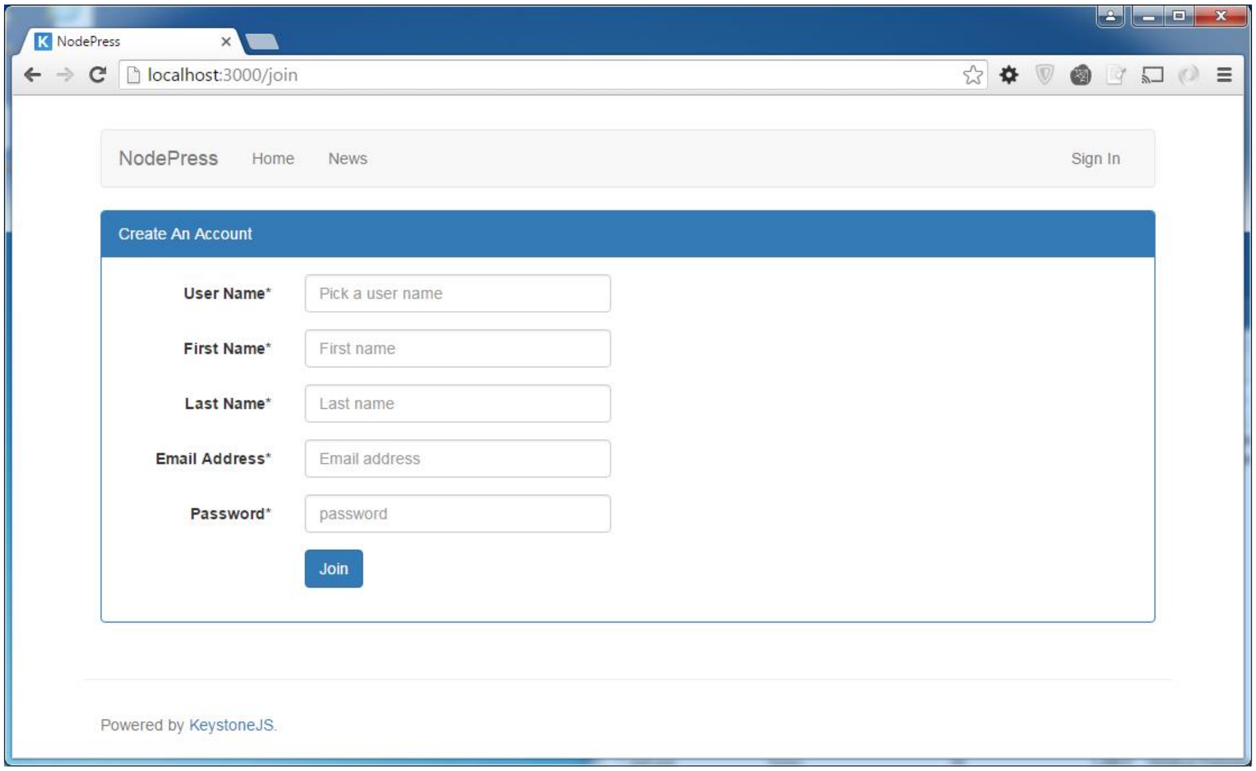


Figure 20: Create account form

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

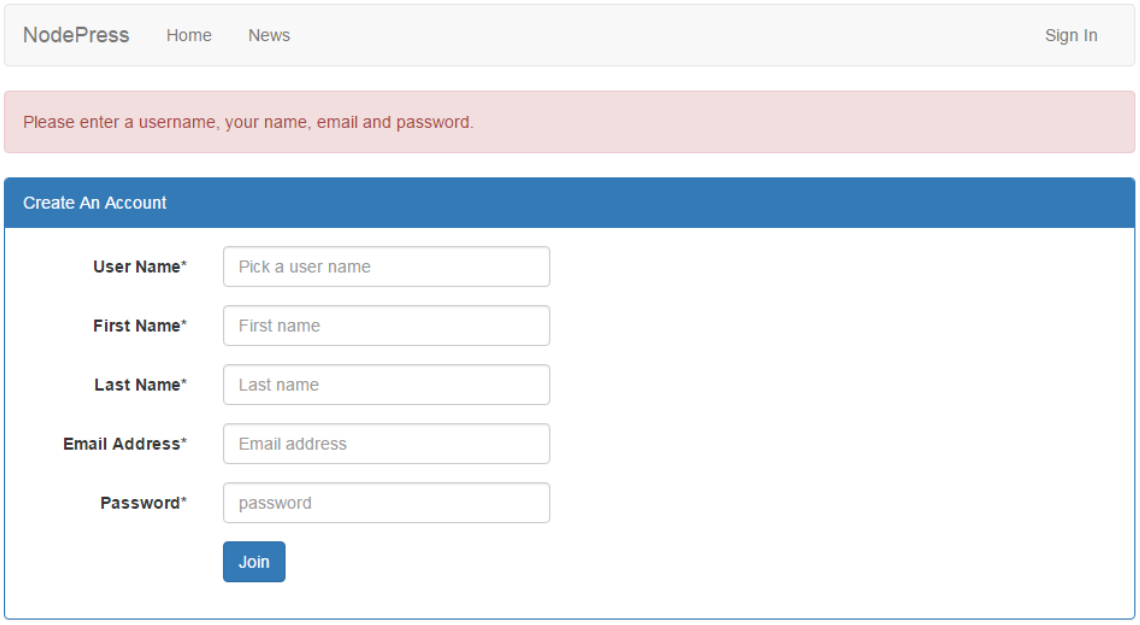
Code Listing 73: Registration view

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

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

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

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

                        return cb();

                    },

            function (cb) {

                keystone.list('User').model.findOne({ username:
req.body.username }, function (err, user) {

                    if (err || user) {
                        req.flash('error', 'User already exists with that
Username.');
                        return cb(true);
                    }

                    return cb();

                });
            },

                    function (cb) {

                        keystone.list('User').model.findOne({ email:
req.body.email }, function (err, user) {

                            if (err || user) {
                                req.flash('error', 'User already exists
with that email address.');
                                return cb(true);
                            }

                            return cb();

                        });

                    },

                    function (cb) {

                        var userData = {
                            username: req.body.username,
                            name: {
                                first: req.body.firstname,
                                last: req.body.lastname,
                            },
                            email: req.body.email,
                            password: req.body.password
                        };

                        var User = keystone.list('User').model,
                                newUser = new User(userData);

                        newUser.save(function (err) {
                            return cb(err);
                        });

                    }

        ], function (err) {

            if (err) return next();

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

            var onFail = function (e) {
                req.flash('error', 'There was a problem signing you up,
please try again.');
                return next();
            }

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

        });

    });

    view.render(join');

}
```

The view uses the excellent **`async`** library that performs multiple operations in series. The first (anonymous) function checks if the form inputs have been populated. The **`next`** method checks if the username entered on the form already exists. If it exists, we return an error to the user. The series operations terminate at this point. The **`next`** method checks if there is an existing user with the same email address.

After all these operations have successfully completed, the user object is constructed and saved to the database. On success, code to log in the user is called and the user is redirected to the homepage.

To test the flash messages that render the error messages from failed form validation, submit the form without filling any values. The error should appear as shown in the following figure.



Please enter a username, your name, email and password.

Create An Account

User Name*	Pick a user name
First Name*	First name
Last Name*	Last name
Email Address*	Email address
Password*	password

Figure 21: Form validation errors

Since we used the **`app.all`** method to define the route, both **`GET`** and **`POST`** are directed to a single action URL. During a **`GET`**, the form is rendered, and during a **`POST`**, the form is validated.

Sign in and sign out

Sign in

Now that the users are able to create an account, let us look at displaying a login form where the users can authenticate themselves. Add the following code within a new file named **signin.swig** and save it under the **templates/views** folder.

Code Listing 74: Sign-in markup

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

{% block content %}

<div class="container">
    <div class="panel panel-primary">
    <!-- Default panel contents -->
    <div class="panel-heading">Login to Nodepress</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@mail.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 for our login form is pretty straightforward. We have defined input fields for the user's email address and 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` static method that is offered by Keystone.js. We have included the following piece of code in our layout file **/templates/layouts/Default.swig** to render the flash messages.

Code Listing 75: Flash messages

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

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

Code Listing 76: Sign-in view

```
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('/');
        }

        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('signin');

}
```

In the view, we check whether the user has already logged in. If they have logged in, we redirect the user to the homepage. If the user has not logged in and has submitted the login form, we will process the login request. 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. If the user has provided valid credentials, then the function will regenerate a new session and complete the sign-in process.

The rendered login form will look like the following figure.

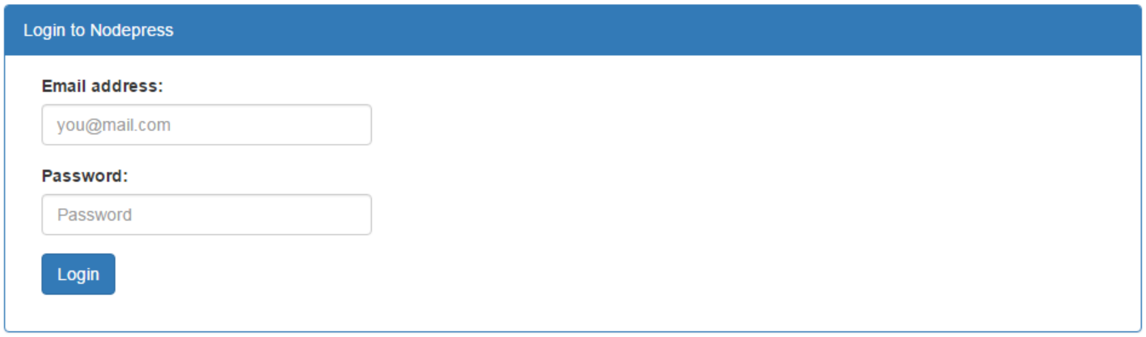


Figure 22: Login form

Sign out

To sign a user out, we should call the **`keystone.session.signout`** method. The **`signout`** operation will clear the user's cookies, set the request user object to null, and regenerate a new session. Upon completion, the user will be redirected to the homepage.

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

Code Listing 77: Sign-out view

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

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

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

Authentication middleware

Keystone.js has built-in middleware that can be leveraged as part of a request and response cycle. This is especially useful to check if requests need to be blocked or allowed based on whether the user has authenticated themselves.

To restrict access to a route to be accessible only to authenticated users, we can rely on Keystone.js 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 follows.

Code Listing 78: Protect route via middleware

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

The preceding piece of code applies the **`requireUser`** method before a request reaches any route that follows **`/profile`**.

The middleware code resides within the **routes/middleware.js** file. The **`requireUser`** method is implemented as shown in the following snippet.

Code Listing 79: RequireUser middleware

```
/**
      Prevents people from accessing protected pages when they're not
signed in.
 */

exports.requireUser = function (req, res, next) {

    if (!req.user) {
        req.flash('error', 'Please sign in to access this page.');
        res.redirect('/keystone/signin');
    } else {
        next();
    }

};
```

Summary

In this chapter, we looked at how we can easily set up an authentication system with Keystone.js. Features such as password recovery and reset can also be easily implemented. Readers should also look at securing applications using cookies and cross-site request forgery (CSRF) protection that Keystone.js facilitates.

Chapter 8 Administration Interface

The admin interface provided by Keystone.js is possibly the best reason for the popularity of Keystone.js over other similar Node.js web frameworks. The admin interface does a lot of heavy lifting for developers, with minimal configuration. It provides a fully featured and extremely tailored content management system (CMS) for Create, Update, Delete operations on models. In this chapter, let's learn how easy it is to configure and customize the admin interface to get the functionality we desire.

The admin panel

The admin interface can be accessed by navigating to http://localhost:3000/keystone. You will be provided with a login interface as shown in Figure 23.

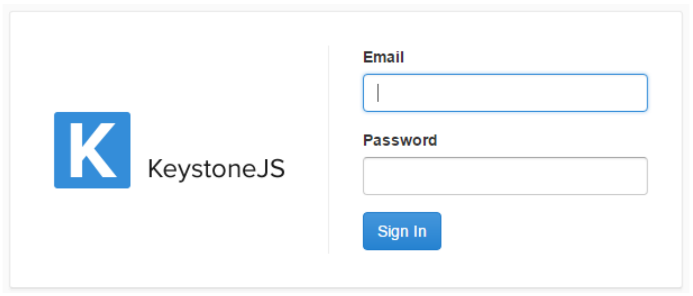


Figure 23: Admin login screen

To log in to the admin panel, use the credentials that were set during the creation of the project using Yeoman. The default credentials are **user@keystone.js** for email and **admin** for password. After logging in you should see something similar to this.

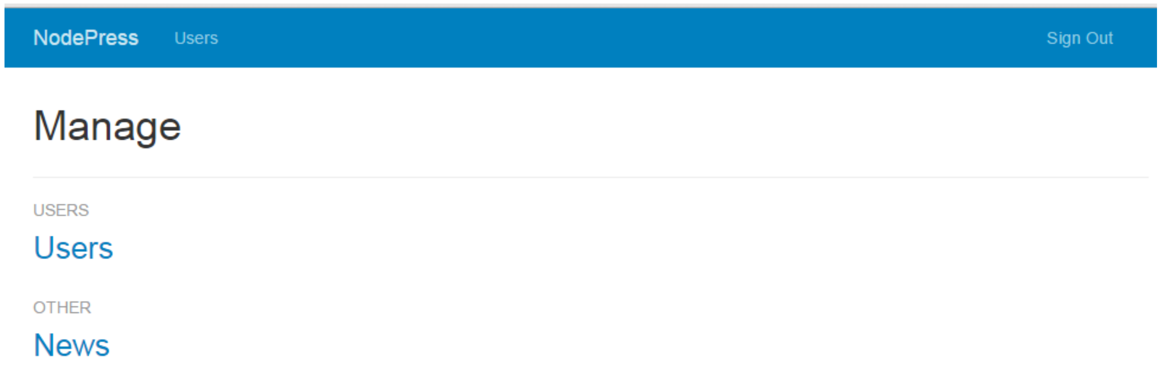
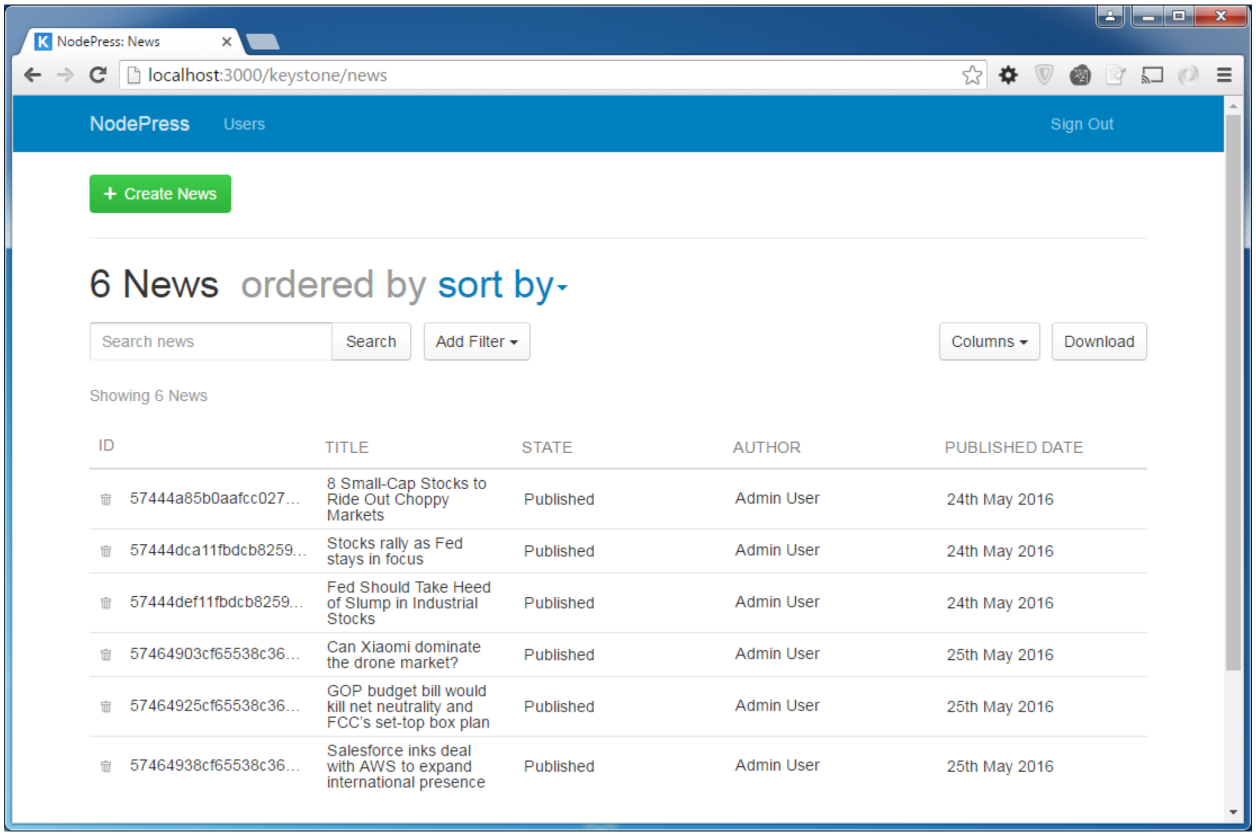


Figure 24: Admin interface

A few things to note in this screen. First, Keystone.js will by default add links to manage users to the navigation menu. Second, any models that we save to the **models** folder will show up in the admin area and are grouped under the **OTHER** label. Thus, the link to manage News shows up under the label **OTHER**.

Let's try editing one of the news items that we have in our MongoDB database. Click on the **News** link, and you should see a screen similar to the following.

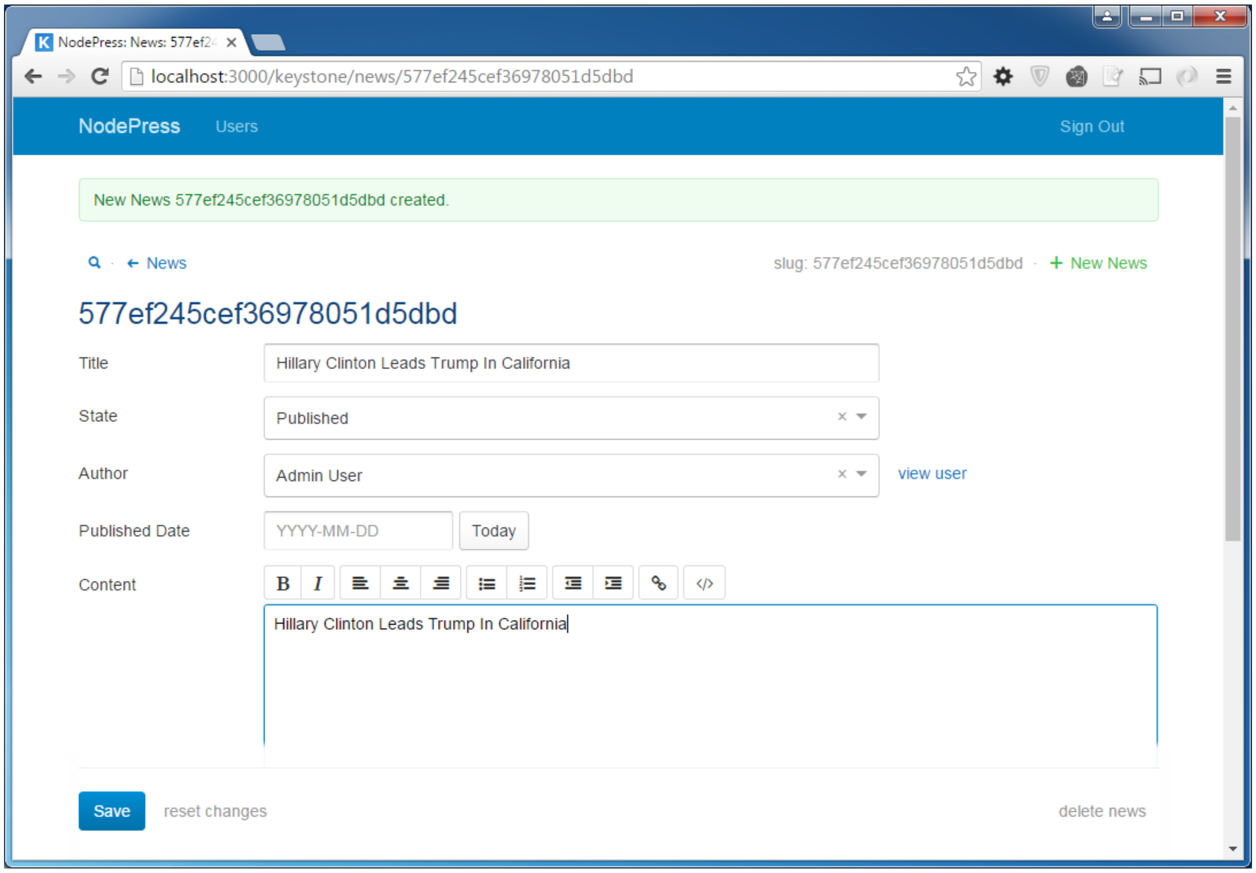


Showing 6 News

ID	TITLE	STATE	AUTHOR	PUBLISHED DATE
57444a85b0aafcc027...	8 Small-Cap Stocks to Ride Out Choppy Markets	Published	Admin User	24th May 2016
57444dca11fbdcb8259...	Stocks rally as Fed stays in focus	Published	Admin User	24th May 2016
57444def11fbdcb8259...	Fed Should Take Heed of Slump in Industrial Stocks	Published	Admin User	24th May 2016
57464903cf65538c36...	Can Xiaomi dominate the drone market?	Published	Admin User	25th May 2016
57464925cf65538c36...	GOP budget bill would kill net neutrality and FCC's set-top box plan	Published	Admin User	25th May 2016
57464938cf65538c36...	Salesforce inks deal with AWS to expand international presence	Published	Admin User	25th May 2016

Figure 25: News management interface

Click on the green **Create News** button at the top, and fill in the form to add a new news item. An autogenerated form is created, as shown in the following figure. The form is created by inspecting the models and model options dynamically.


NodePress: News: 577ef2

localhost:3000/keystone/news/577ef245cef36978051d5dbd

New News 577ef245cef36978051d5dbd created.

577ef245cef36978051d5dbd

Title: Hillary Clinton Leads Trump In California

State: Published

Author: Admin User

Published Date: YYYY-MM-DD

Content: Hillary Clinton Leads Trump In California

Figure 26: Add a news item

Just make sure to set the **State** field to **Published** for it to show up on the news page of the site. The **delete news** button can be used to remove unwanted entries. The Author input field automatically shows an autocomplete text box since it was marked as a relationship field type during model definition in code.

Customizing menus

The menu items in the administration site can be easily configured in the keystone.js file. The menu items are stored in an object in the configuration with **`nav`** as the key. Let's add the **`news`** menu item to the menu.

Code Listing 80: Updated nav menu

```
keystone.set('nav', {
    'users': 'users',
    'manageNews': 'news'
});
```

The updated navigation menu will look as shown in the following figure.

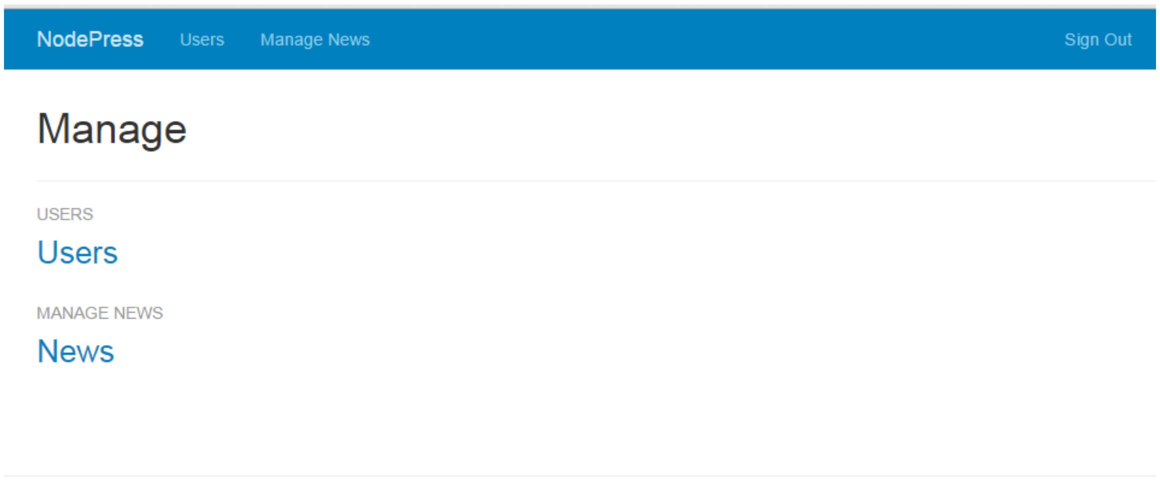


Figure 27: Modified nav menu

Customizing the list views

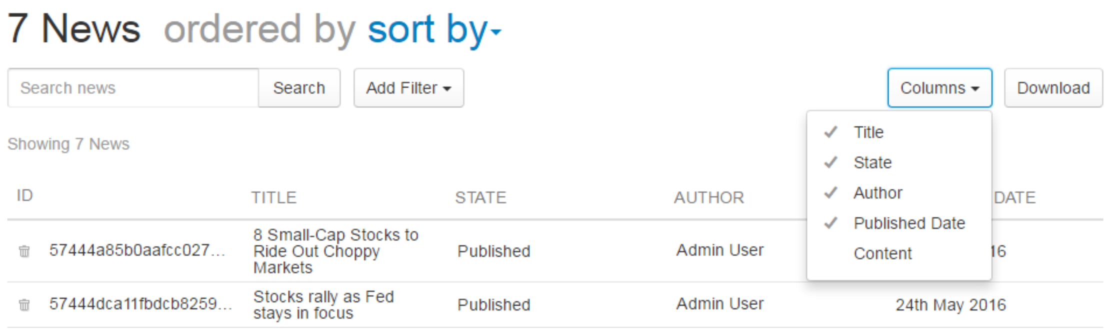
On the Manage News screen, we see a neatly organized list of news items. The title, state, author, and published date show up in the interface since we configured the corresponding option (**`defaultColumns`**) on the **`News`** model. Recall the following line of code from the **`News`** model.

Code Listing 81: Default columns on admin interface

```
News.defaultColumns = 'title, state|20%, author|20%, publishedDate|20%';
```

The default columns option is a comma-delimited list of default columns to display in the Admin UI list view. You can specify width in either pixels or percent after a | pipe character.

We can also select additional columns or remove columns from the list view using the Columns drop-down menu on the right-hand side, as shown in the following figure.



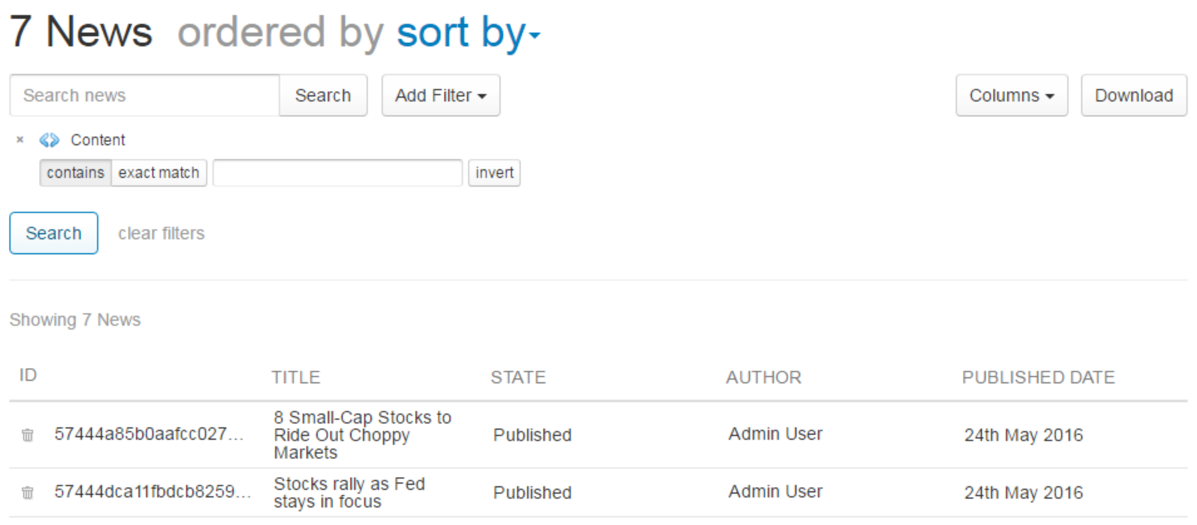
ID	TITLE	STATE	AUTHOR	DATE
57444a85b0aafcc027…	8 Small-Cap Stocks to Ride Out Choppy Markets	Published	Admin User	16
57444dca11fbdcb8259…	Stocks rally as Fed stays in focus	Published	Admin User	24th May 2016

Figure 28: Dynamic columns

Columns that are already defined are shown with a check next to them. The drop-down menu allows us to choose additional columns or remove existing columns on the fly.

Search and filter

Most admin panels allow for users to search and filter content. Keystone.js provides powerful search and filter options out of the box. The Search box provided in the admin panel performs a search on the title of the news by default. We can specify additional columns that need to be searched by setting the **`searchFields`** options on the model.



ID	TITLE	STATE	AUTHOR	PUBLISHED DATE
57444a85b0aafcc027…	8 Small-Cap Stocks to Ride Out Choppy Markets	Published	Admin User	24th May 2016
57444dca11fbdcb8259…	Stocks rally as Fed stays in focus	Published	Admin User	24th May 2016

Figure 29: Dynamic filters

Figure 29 shows an added filter that allows us to search on the content as well as title. We have options to search for an exact match or for content that contains our keywords. The invert option is used to negate the search query.

Keystone.js provides intuitive and powerful filter options depending on the field type. For example, in Figure 30, we have enabled the filter for the published date, and Keystone.js automatically shows options to filter by date on, after, before, and between two dates.

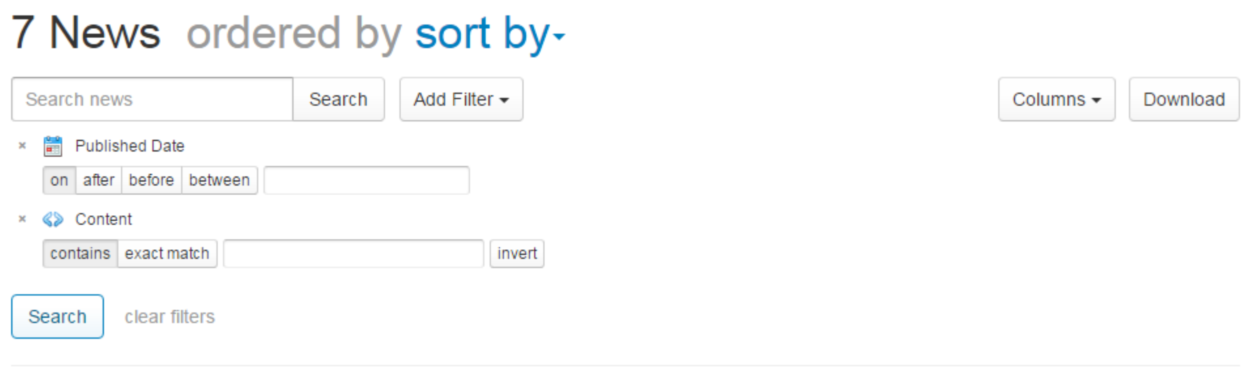


Figure 30: Intuitive date filter

Summary

We have covered the Keystone.js admin interface, which lets us handle routine Create, Read, Update, and Delete operations almost for free. We have a powerful and friendly way to create test data, and one that would serve us well for production purposes if we wanted.

Chapter 9 Building REST APIs

REST APIs expose interfaces that allow various clients to read and write data in a consistent manner. REST APIs are resource-centric, which means that the methods are only concerned with the underlying resource and will not respond to arbitrary service methods. REST API methods work by mapping HTTP verbs into API calls. The most common HTTP verbs used with REST APIs are **`GET`**, **`POST`**, **`PUT`**, and **`DELETE`**. 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. However, the idea should be to keep the API aligned with the application architecture and design.

Expose endpoint for retrieving news

Let’s begin by outlining what our API is actually going to look like. To be able to work with the news items that are stored in our MongoDB database, we are going to support an HTTP **`GET`** to an API called **`news`** that will return a list of news items. In addition, the API will allow users to get data about a specific news item by ID. Let us define the following endpoints to allow for data retrieval in a REST-based fashion. The URL for a RESTful API is known as an endpoint.

`GET /api/news:` This gets the list of tickets.

`GET /api/news/{id}:` This gets the ticket with ID {id}.

To start, add the following routes that define the endpoints.

Code Listing 82: GET API routes

```
app.get('/api/news', keystone.middleware.api, routes.api.news.getNews);
app.get('/api/news/:id', keystone.middleware.api,
routes.api.news.getNewsById);
```

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 exception that occurred. By default, it returns an 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 **keystone.js** file at the root of the application to include the **api** folder.

Code Listing 83: Include API routes

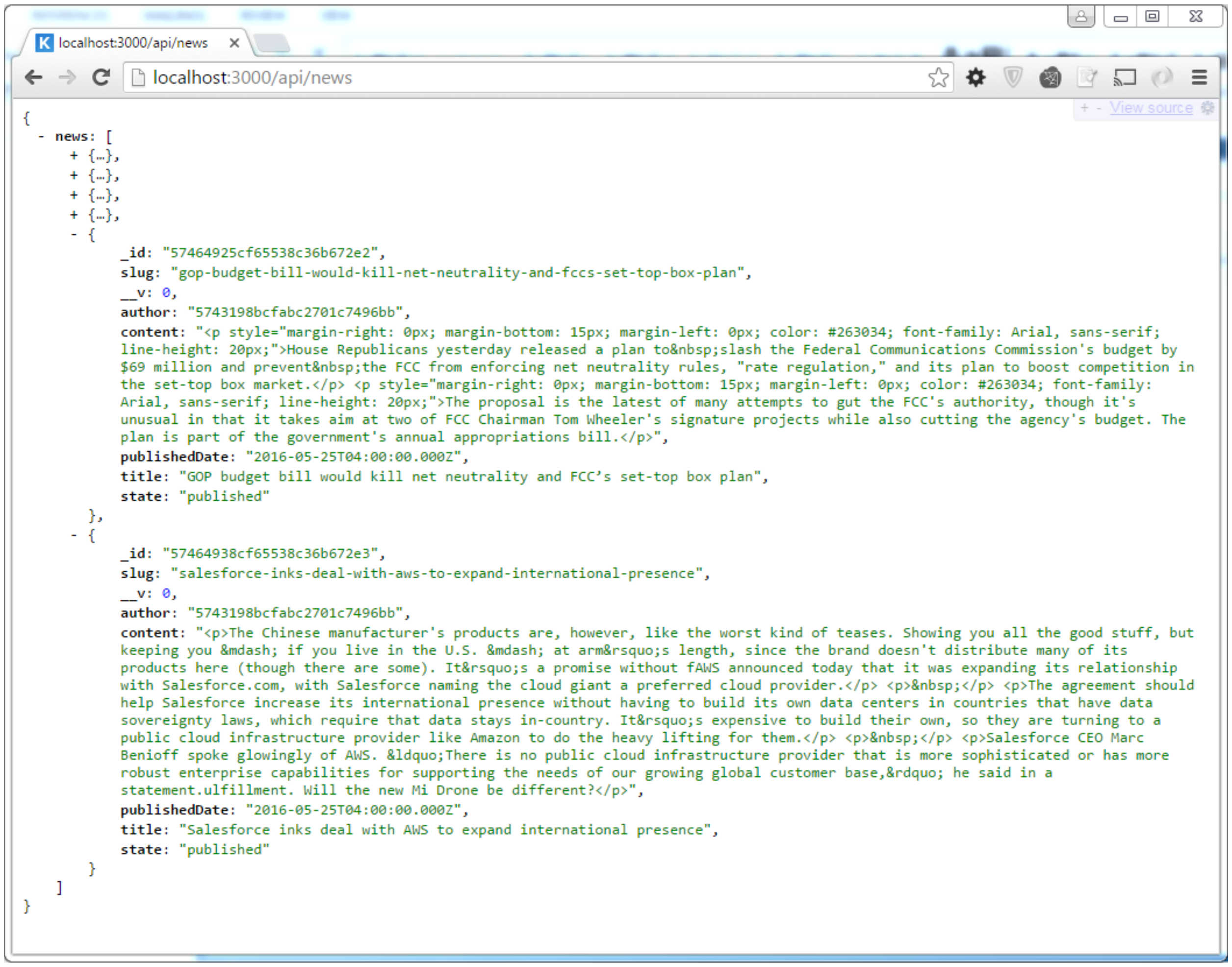
```
// Import Route Controllers
var routes = {
      views: importRoutes('./views'),
      api: importRoutes('./api')
};
```

Next, add a new JavaScript file to the **routes/api** folder and name it **news.js** with the following code.

Code Listing 84: API to get news

```
// API to get news
var keystone = require('keystone'),
News = keystone.list('News');

/**
* Get List of News
*/
exports.getNews = function(req, res) {
    News.model.find(function(err, items) {

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

        res.apiResponse({
            news: items
        });
    });
}

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

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

        res.apiResponse({
        news: item
        });
    });
}
```

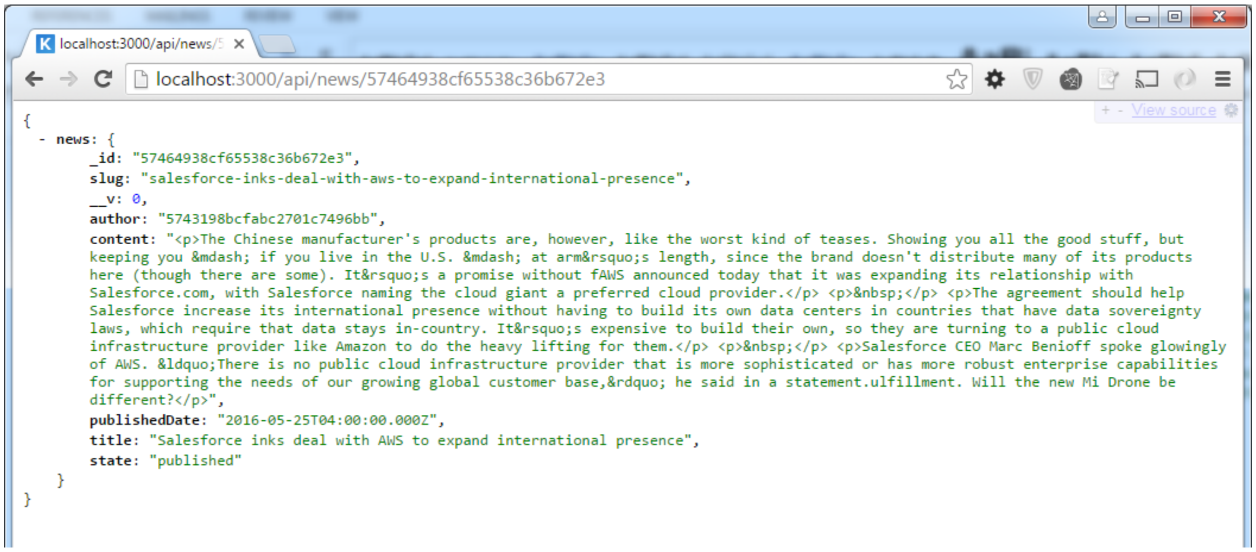
To test the API, restart the node application and navigate to http://localhost:3000/api/news using Chrome or the Postman add-on tool for the Chrome browser. We should see a list of news returned to in JSON format.



```
{
  - news: [
      + {…},
      + {…},
      + {…},
      + {…},
      - {
            _id: "57464925cf65538c36b672e2",
            slug: "gop-budget-bill-would-kill-net-neutrality-and-fccs-set-top-box-plan",
            __v: 0,
            author: "5743198bcfabc2701c7496bb",
            content: "<p style="margin-right: 0px; margin-bottom: 15px; margin-left: 0px; color: #263034; font-family: Arial, sans-serif;
            line-height: 20px;">House Republicans yesterday released a plan to&nbsp;slash the Federal Communications Commission's budget by
            $69 million and prevent&nbsp;the FCC from enforcing net neutrality rules, "rate regulation," and its plan to boost competition in
            the set-top box market.</p> <p style="margin-right: 0px; margin-bottom: 15px; margin-left: 0px; color: #263034; font-family:
            Arial, sans-serif; line-height: 20px;">The proposal is the latest of many attempts to gut the FCC's authority, though it's
            unusual in that it takes aim at two of FCC Chairman Tom Wheeler's signature projects while also cutting the agency's budget. The
            plan is part of the government's annual appropriations bill.</p>",
            publishedDate: "2016-05-25T04:00:00.000Z",
            title: "GOP budget bill would kill net neutrality and FCC’s set-top box plan",
            state: "published"
        },
      - {
            _id: "57464938cf65538c36b672e3",
            slug: "salesforce-inks-deal-with-aws-to-expand-international-presence",
            __v: 0,
            author: "5743198bcfabc2701c7496bb",
            content: "<p>The Chinese manufacturer's products are, however, like the worst kind of teases. Showing you all the good stuff, but
            keeping you &mdash; if you live in the U.S. &mdash; at arm&rsquo;s length, since the brand doesn't distribute many of its
            products here (though there are some). It&rsquo;s a promise without fAWS announced today that it was expanding its relationship
            with Salesforce.com, with Salesforce naming the cloud giant a preferred cloud provider.</p> <p>&nbsp;</p> <p>The agreement should
            help Salesforce increase its international presence without having to build its own data centers in countries that have data
            sovereignty laws, which require that data stays in-country. It&rsquo;s expensive to build their own, so they are turning to a
            public cloud infrastructure provider like Amazon to do the heavy lifting for them.</p> <p>&nbsp;</p> <p>Salesforce CEO Marc
            Benioff spoke glowingly of AWS. &ldquo;There is no public cloud infrastructure provider that is more sophisticated or has more
            robust enterprise capabilities for supporting the needs of our growing global customer base,&rdquo; he said in a
            statement.ulfillment. Will the new Mi Drone be different?</p>",
            publishedDate: "2016-05-25T04:00:00.000Z",
            title: "Salesforce inks deal with AWS to expand international presence",
            state: "published"
        }
    ]
}
```

Figure 31: API GetNews response JSON

To test the GetNewsById API call, navigate to http://localhost:3000/api/news/57464938cf65538c36b672e3 where you substitute one of your news item IDs in the URL. We should see a single JSON object returned as follows.


localhost:3000/api/news/57464938cf65538c36b672e3

```
{
  - news: {
        _id: "57464938cf65538c36b672e3",
        slug: "salesforce-inks-deal-with-aws-to-expand-international-presence",
        __v: 0,
        author: "5743198bcfabc2701c7496bb",
        content: "<p>The Chinese manufacturer's products are, however, like the worst kind of teases. Showing you all the good stuff, but
        keeping you &mdash; if you live in the U.S. &mdash; at arm&rsquo;s length, since the brand doesn't distribute many of its products
        here (though there are some). It&rsquo;s a promise without fAWS announced today that it was expanding its relationship with
        Salesforce.com, with Salesforce naming the cloud giant a preferred cloud provider.</p> <p>&nbsp;</p> <p>The agreement should help
        Salesforce increase its international presence without having to build its own data centers in countries that have data sovereignty
        laws, which require that data stays in-country. It&rsquo;s expensive to build their own, so they are turning to a public cloud
        infrastructure provider like Amazon to do the heavy lifting for them.</p> <p>&nbsp;</p> <p>Salesforce CEO Marc Benioff spoke glowingly
        of AWS. &ldquo;There is no public cloud infrastructure provider that is more sophisticated or has more robust enterprise capabilities
        for supporting the needs of our growing global customer base,&rdquo; he said in a statement.ulfillment. Will the new Mi Drone be
        different?</p>",
        publishedDate: "2016-05-25T04:00:00.000Z",
        title: "Salesforce inks deal with AWS to expand international presence",
        state: "published"
    }
}
```

Figure 32: API GetNewsById response JSON

Expose endpoint for creating news

The API should also allow users to create a new news post. The most common HTTP verbs used 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, inserts a news item into our collection, and returns the new document JSON. Update the routes file and include the following route.

Code Listing 85: API route for POST

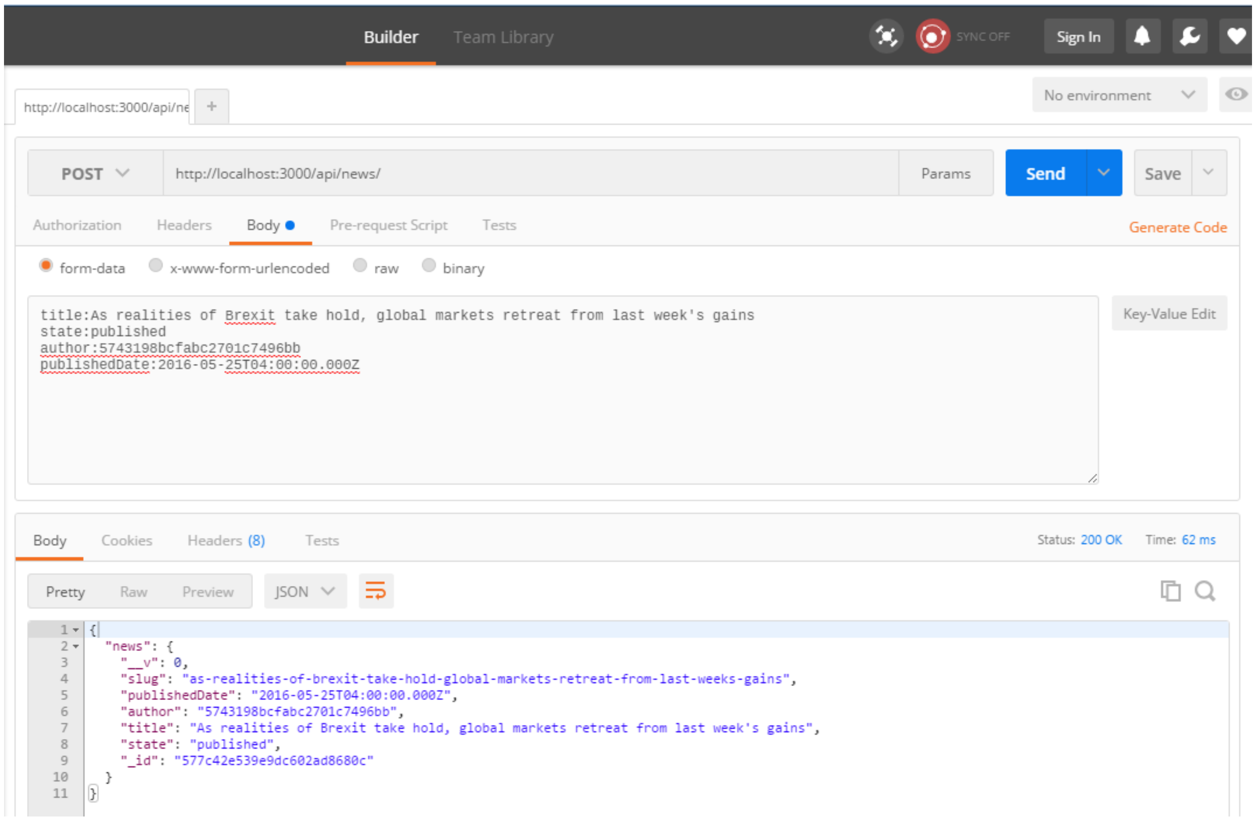
```
app.post('/api/news', keystone.middleware.api,
routes.api.news.createNews);
```

Next, add the following code to the **news.js** file under the **api** folder.

Code Listing 86: API route for POST

```
/**
* Create a News Item
*/
exports.createNews = function (req, res) {
    var item = new News.model(),
    data = req.body;

    item.getUpdateHandler(req).process(data, function (err) {
        if (err) return res.apiError('error', err);
        res.apiResponse({
            news: item
        });
    });
}
```

To test the create functionality, use the Postman tool for Chrome and craft a **`POST`** request to [http://localhost:3000/api/news.](http://localhost:3000/api/news)



POST http://localhost:3000/api/news/

[✓ form-data] x-www-form-urlencoded raw binary

```
title:As realities of Brexit take hold, global markets retreat from last week's gains
state:published
author:5743198bcfabc2701c7496bb
publishedDate:2016-05-25T04:00:00.000Z
```

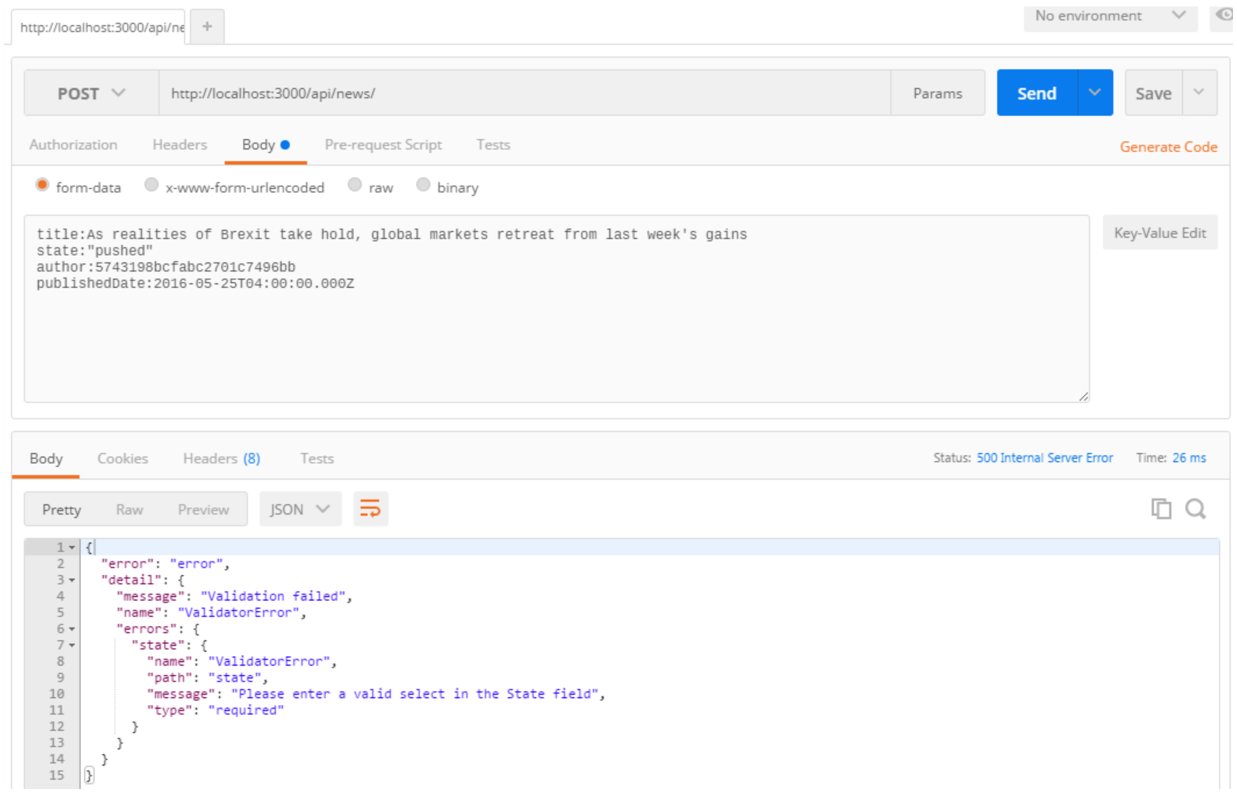
Status: 200 OK Time: 62 ms

```
{
  "news": {
    "__v": 0,
    "slug": "as-realities-of-brexit-take-hold-global-markets-retreat-from-last-weeks-gains",
    "publishedDate": "2016-05-25T04:00:00.000Z",
    "author": "5743198bcfabc2701c7496bb",
    "title": "As realities of Brexit take hold, global markets retreat from last week's gains",
    "state": "published",
    "_id": "577c42e539e9dc602ad8680c"
  }
}
```

Figure 33: API CreateNews response JSON

The **`getUpdateHandler`** method is the heart of this method. This method 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.

To check the validation in action, perform an invalid **`POST`** to the **/api/news** endpoint. In the following example, we have specified an incorrect state for the news (pushed instead of published). The returned response clearly mentions the reason for failure along with an appropriate HTTP error code (500).

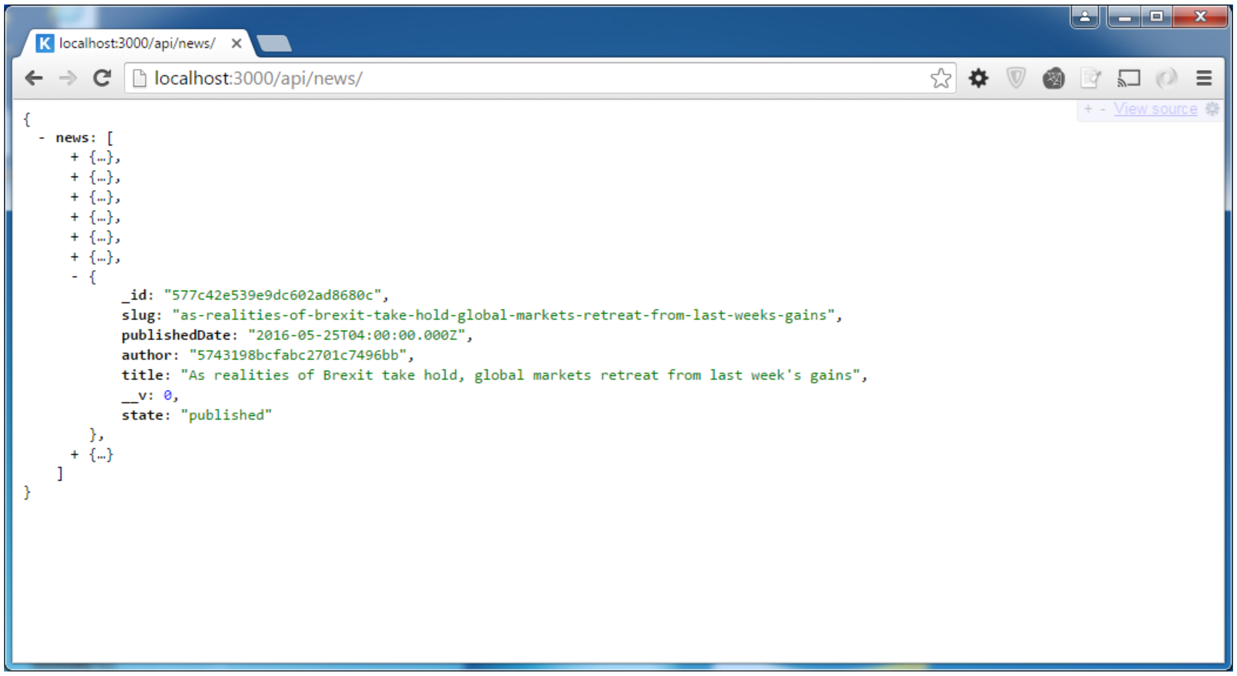


```
title:As realities of Brexit take hold, global markets retreat from last week's gains
state:"pushed"
author:5743198bcfabc2701c7496bb
publishedDate:2016-05-25T04:00:00.000Z
```

```
{
  "error": "error",
  "detail": {
    "message": "Validation failed",
    "name": "ValidatorError",
    "errors": {
      "state": {
        "name": "ValidatorError",
        "path": "state",
        "message": "Please enter a valid select in the State field",
        "type": "required"
      }
    }
  }
}
```

Figure 34: Testing the getUpdateHandler method

Now, issue a **`GET`** request to http://localhost:3000/api/news, and we should see the newly created news post returned.



```
{
  - news: [
      + {…},
      + {…},
      + {…},
      + {…},
      + {…},
      + {…},
      - {
            _id: "577c42e539e9dc602ad8680c",
            slug: "as-realities-of-brexit-take-hold-global-markets-retreat-from-last-weeks-gains",
            publishedDate: "2016-05-25T04:00:00.000Z",
            author: "5743198bcfabc2701c7496bb",
            title: "As realities of Brexit take hold, global markets retreat from last week's gains",
            __v: 0,
            state: "published"
        },
      + {…}
    ]
}
```

Figure 35: Fetch updated news

Expose endpoint for deleting news

The **`delete`** operation is pretty straightforward. We should expose an endpoint that responds to a call with an **`HTTP DELETE`** verb along with the ID of the news that needs to be removed. Add the following route to the application.

Code Listing 87: API route for DELETE

```
app.delete('/api/news/:id', keystone.middleware.api,
routes.api.news.deleteNewsById);
```

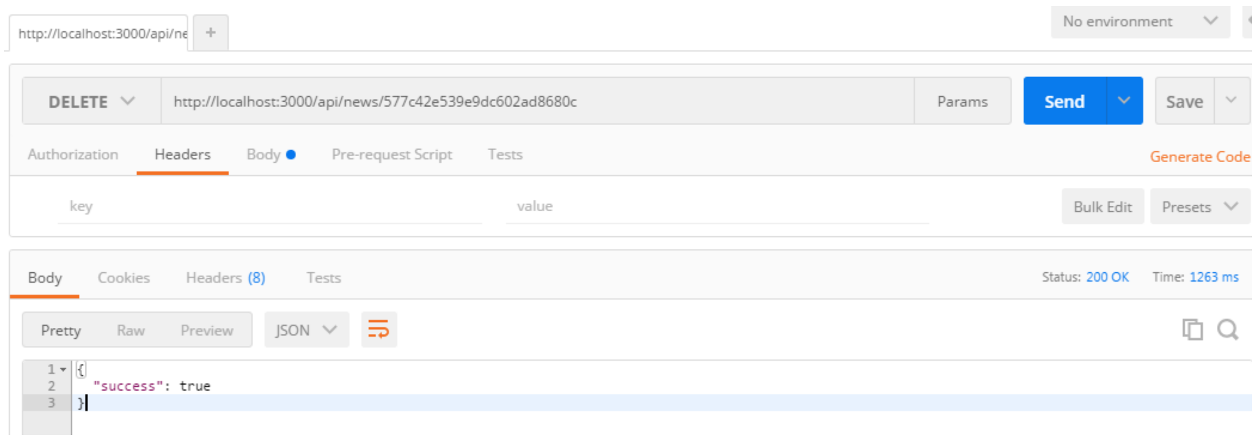
Add the following code to the news.js API file.

Code Listing 88: View for DELETE

```
/**
* Delete a News Item
*/
exports.deleteNewsById = function (req, res) {
    News.model.findById(req.params.id).exec(function (err, item) {

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

        item.remove(function (err) {
            if (err) return res.apiError('database error', err);

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

    });
}
```

The code uses the **`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.

Use Postman to issue a delete request to http://localhost:3000/api/news/{id}. Replace the **`id`** parameter with an existing ID from our news collection. If the document is deleted, we should see a success message returned.



http://localhost:3000/api/news/577c42e539e9dc602ad8680c

Status: 200 OK   Time: 1263 ms

```
{
  "success": true
}
```

Figure 36: Delete news item

Summary

In this chapter we saw how extremely simple it is to expose REST endpoints in a Keystone.js application. The endpoints do not enforce strict role validation and authentication rules and may not be appropriate for production environments as is. However, those rules can be easily added to the Keystone.js middleware.