BUILD POWERFUL WEB APPLICATIONS QUICKLY!

Practical Keystone.js

Node.js, Express & MongoDB

Manikanta Panati

Practical Keystone.js

A Hands On Introduction To Keystone.js Using a Real-World Project

Manikanta Panati

This book is for sale at http://leanpub.com/keystonejs

This version was published on 2016-02-28

Tweet This Book!

Please help Manikanta Panati by spreading the word about this book on Twitter!

The suggested hashtag for this book is #keystonejs.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#keystonejs

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

Contents

Introduction **1**
What are web frameworks? 1
The Model-View-Template design pattern 1
Introducing the IncTicket Project 2
Errata and Suggestions 2

Chapter 1. Meet Keystone.js **3**
Why use Keystone.js 3
What is Keystone.js Best for? 4
Keystone.js Versions 4
Installing MongoDB and Node.js 4
How to install Node.js 6
Useful Development and Debugging Tools. 8
Summary 14

Chapter 2. Building the IncTicket Application **15**
Installing Keystone.js 15
Creating a new Keystone.js Application 16
Configuring Your Keystone.js Application 18
Project Structure 19
Creating Your First Model/List 21
Creating an administration site for your models 23
Creating Your First Route 31
Creating Your First View 34
Creating templates for your views 37
Adding pagination 42
Summary 45

Introduction

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

Keystone.js philosophy

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

What are web frameworks?

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

The Model-View-Template design pattern

Keystone.js is based on a design pattern called Model-View-Template. A good understanding of this concept is the basis for working with Keystone.js. Web application architecture generally comprises of three pieces that work together - data access logic, business logic, and presentation logic. A good framework will aim for the logical separation of these pieces in an application into distinct subsystems so as to allow for high degree of re-usability of components. Here's roughly how the M, V, and T break down in Keystone.js:

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

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

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

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

Both MVC and MVT are very similar and interpretation of these concepts varies slightly from framework to framework and no one is more "correct" than the other. It is good to get a proper understanding of the underlying concepts.

Introducing the IncTicket Project

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

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

Errata and Suggestions

"Have no fear of perfection - you'll never reach it", said Salvador Dali. When it comes to writing about the latest technology, I could'nt agree more!. I might have made mistakes in both code and grammar, and probably completely misconstrued a few pieces of this text. If you would like to report an error, ask a question or offer a suggestion, please reach me on twitter @jangreejelabi.

Chapter 1. Meet Keystone.js

This chapter will introduce Keystone.js along with its merits. We will cover how to install MongoDB and Node.js that are needed create the IncTicket application using Keystone.js and run it.

This chapter will cover the following points:

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

Why use Keystone.js

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

Keystone.js has a number of key features that makes it worth using including:

- **Modularity** - Keystone will configure express - the de facto web server for node.js - for you and connect to your MongoDB database using Mongoose, the leading ODM package.
- **Auto-generated Admin UI** - Whether you use it while you're building out your application, or in production as a database content management system, Keystone's Admin UI will save you time and make managing your data easy.
- **Session Management** - Keystone comes ready out of the box with session management and authentication features, including automatic encryption for password fields.
- **Email Sending** - Keystone makes it easy to set up, preview and send template-based emails for your application. It also integrates with Mandrill.
- **Form Processing** - Want to validate a form, upload an image, and update your database with a single line? Keystone can do that, based on the data models you've already defined.
- **Database Fields** - IDs, Strings, Booleans, Dates and Numbers are the building blocks of your database. Keystone builds on these with useful, real-world field types like name, email, password, address, image and relationship fields.

What is Keystone.js Best for?

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

Keystone.js Versions

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

Installing MongoDB and Node.js

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

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

How to install MongoDB

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

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

```
wmic os get osarchitecture
```

The output will be similar to:

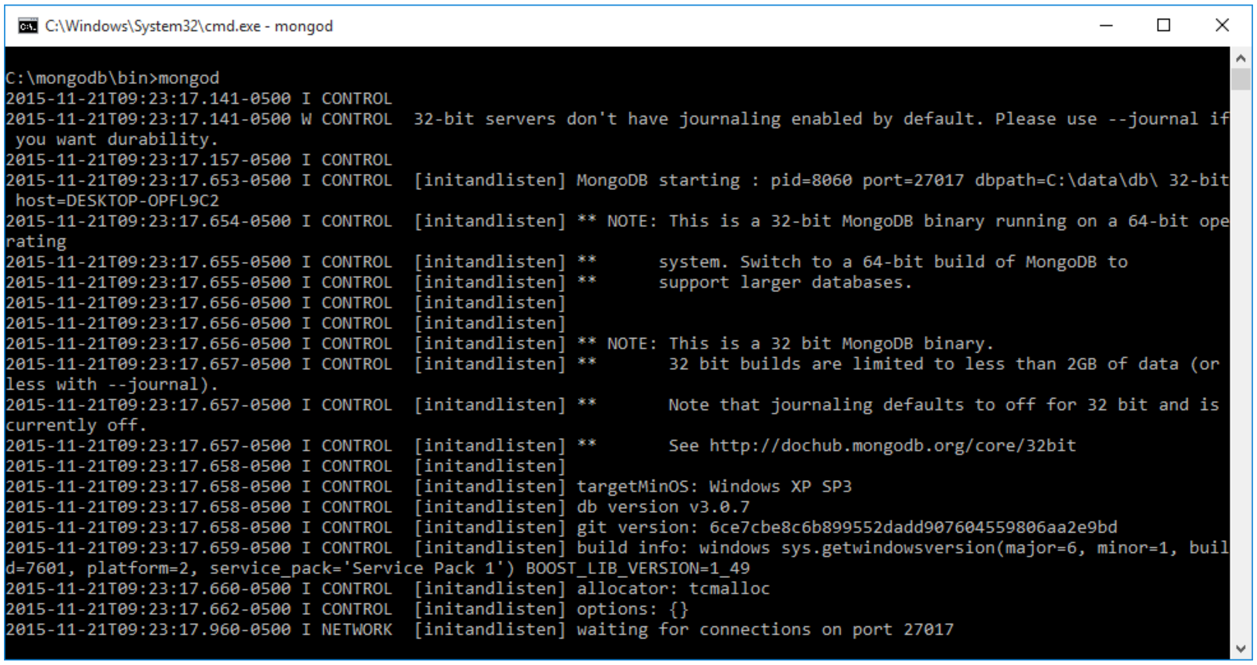
```
OSArchiecture
64-bit
```

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

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

- To start the mongodb database, Open a CMD prompt window, and enter the following commands:

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



```
C:\mongodb\bin>mongod
2015-11-21T09:23:17.141-0500 I CONTROL
2015-11-21T09:23:17.141-0500 W CONTROL  32-bit servers don't have journaling enabled by default. Please use --journal if
 you want durability.
2015-11-21T09:23:17.157-0500 I CONTROL
2015-11-21T09:23:17.653-0500 I CONTROL  [initandlisten] MongoDB starting : pid=8060 port=27017 dbpath=C:\data\db\ 32-bit
 host=DESKTOP-OPFL9C2
2015-11-21T09:23:17.654-0500 I CONTROL  [initandlisten] ** NOTE: This is a 32-bit MongoDB binary running on a 64-bit ope
rating
2015-11-21T09:23:17.655-0500 I CONTROL  [initandlisten] **       system. Switch to a 64-bit build of MongoDB to
2015-11-21T09:23:17.655-0500 I CONTROL  [initandlisten] **       support larger databases.
2015-11-21T09:23:17.656-0500 I CONTROL  [initandlisten]
2015-11-21T09:23:17.656-0500 I CONTROL  [initandlisten]
2015-11-21T09:23:17.656-0500 I CONTROL  [initandlisten] ** NOTE: This is a 32 bit MongoDB binary.
2015-11-21T09:23:17.657-0500 I CONTROL  [initandlisten] **        32 bit builds are limited to less than 2GB of data (or
less with --journal).
2015-11-21T09:23:17.657-0500 I CONTROL  [initandlisten] **        Note that journaling defaults to off for 32 bit and is
currently off.
2015-11-21T09:23:17.657-0500 I CONTROL  [initandlisten] **        See http://dochub.mongodb.org/core/32bit
2015-11-21T09:23:17.658-0500 I CONTROL  [initandlisten]
2015-11-21T09:23:17.658-0500 I CONTROL  [initandlisten] targetMinOS: Windows XP SP3
2015-11-21T09:23:17.658-0500 I CONTROL  [initandlisten] db version v3.0.7
2015-11-21T09:23:17.658-0500 I CONTROL  [initandlisten] git version: 6ce7cbe8c6b899552dadd907604559806aa2e9bd
2015-11-21T09:23:17.659-0500 I CONTROL  [initandlisten] build info: windows sys.getwindowsversion(major=6, minor=1, buil
d=7601, platform=2, service_pack='Service Pack 1') BOOST_LIB_VERSION=1_49
2015-11-21T09:23:17.660-0500 I CONTROL  [initandlisten] allocator: tcmalloc
2015-11-21T09:23:17.662-0500 I CONTROL  [initandlisten] options: {}
2015-11-21T09:23:17.960-0500 I NETWORK  [initandlisten] waiting for connections on port 27017
```

Start MongoDB

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

How to install Node.js

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

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

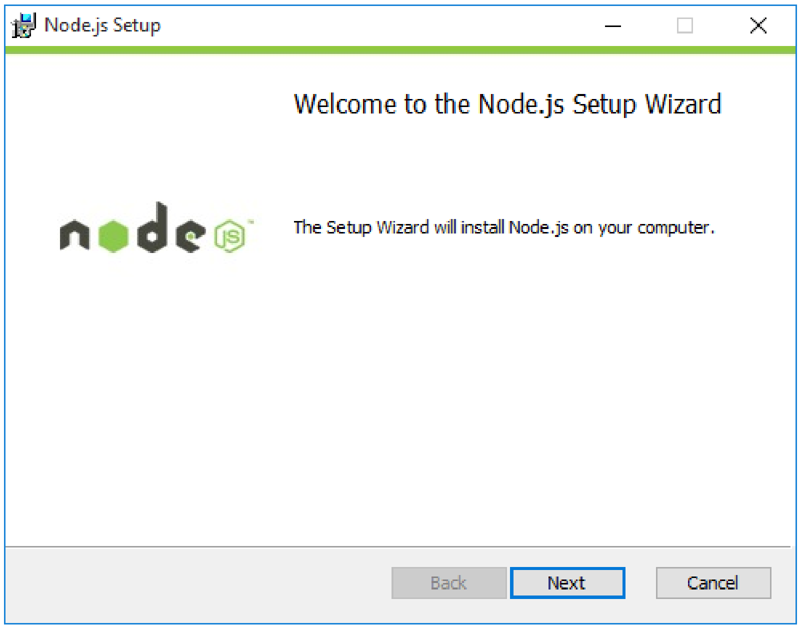
Find the architecture of your machine by typing in the following command into the command prompt:

```
wmic os get osarchitecture
```

The output will be similar to:

```
OSArchiecture
64-bit
```

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

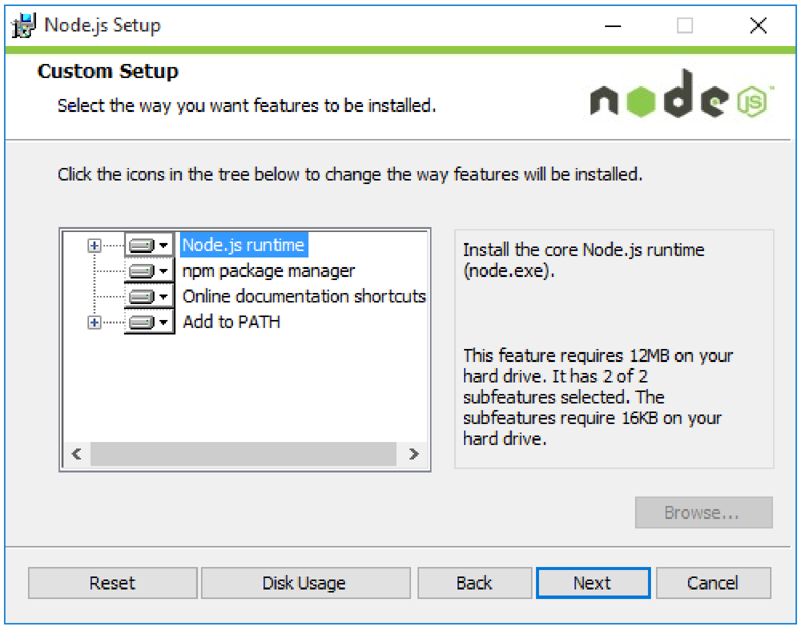


Welcome to the Node.js Setup Wizard

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

Node.js Installer

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


Node.js Setup

Custom Setup

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.

Node.js Installer

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

Testing whether Node.js is installed properly

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

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

```
c:\> node --version
    v4.2.2

c:\> npm --version
    2.14.7
```

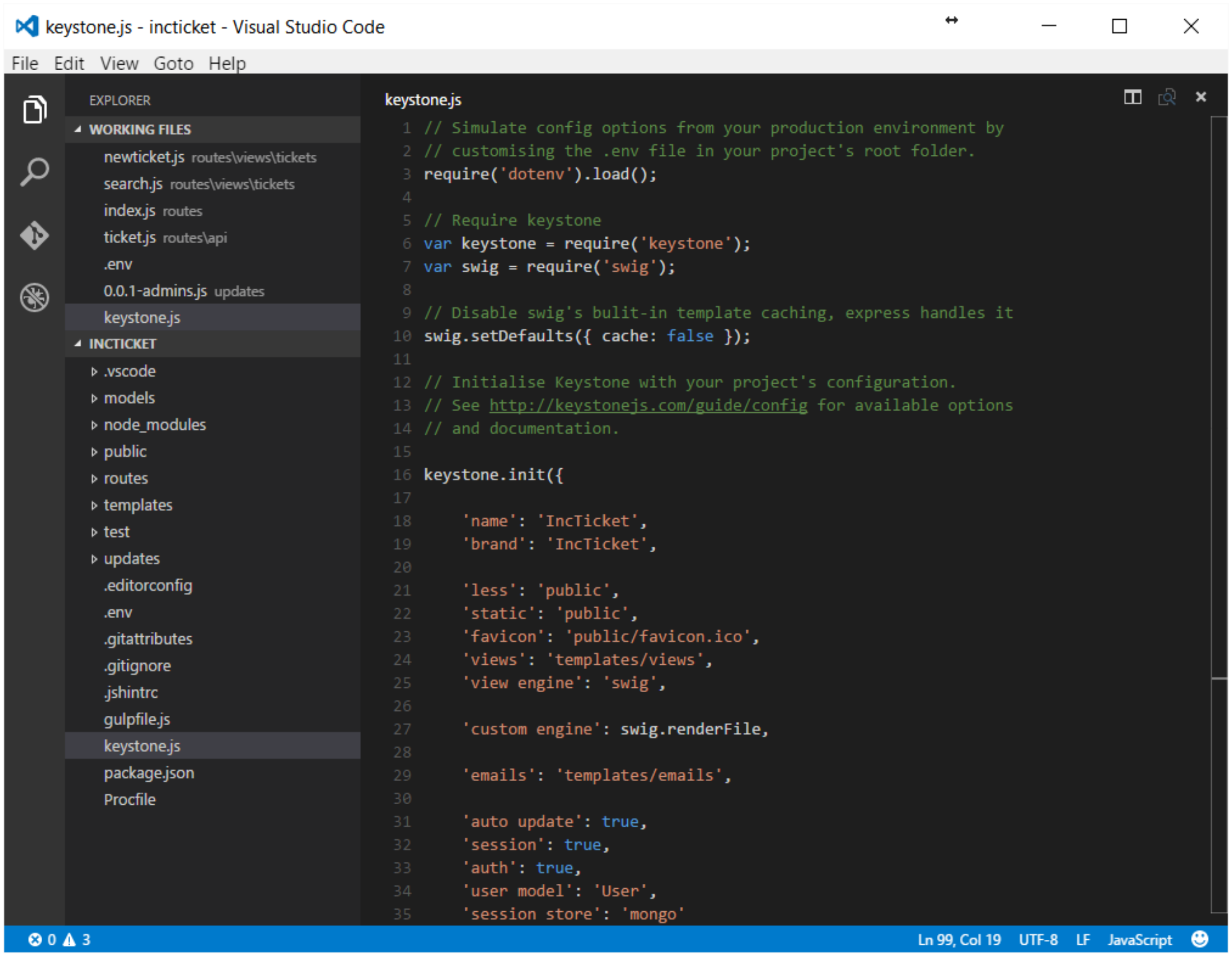
If the Node installation was successful, you will see the version number that was installed as an output on the screen as a response to running the above commands.

Useful Development and Debugging Tools.

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

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

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



```
// Simulate config options from your production environment by
// customising the .env file in your project's root folder.
require('dotenv').load();

// Require keystone
var keystone = require('keystone');
var swig = require('swig');

// Disable swig's bulit-in template caching, express handles it
swig.setDefaults({ cache: false });

// Initialise Keystone with your project's configuration.
// See http://keystonejs.com/guide/config for available options
// and documentation.

keystone.init({

    'name': 'IncTicket',
    'brand': 'IncTicket',

    'less': 'public',
    'static': 'public',
    'favicon': 'public/favicon.ico',
    'views': 'templates/views',
    'view engine': 'swig',

    'custom engine': swig.renderFile,

    'emails': 'templates/emails',

    'auto update': true,
    'session': true,
    'auth': true,
    'user model': 'User',
    'session store': 'mongo'
```

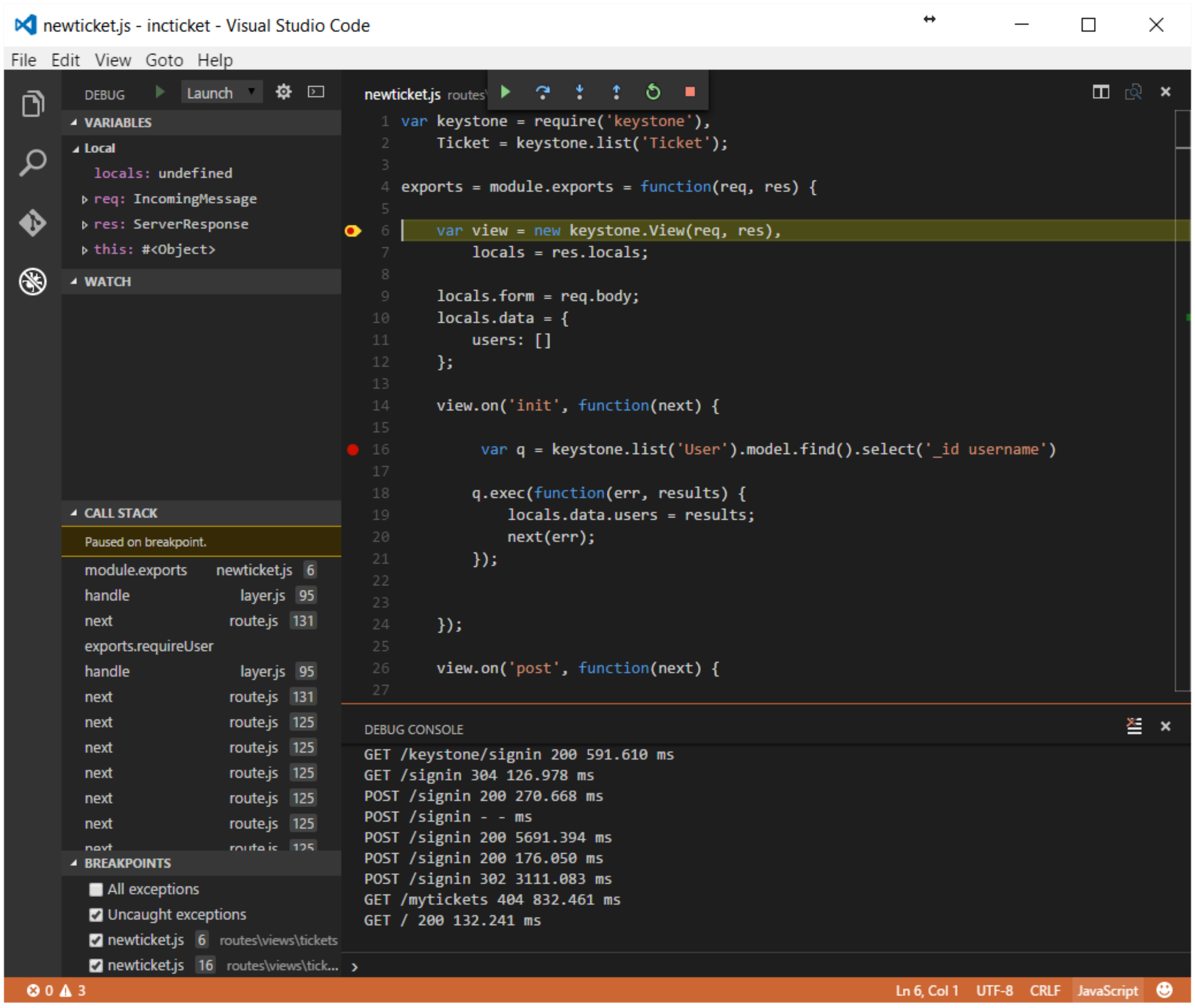
Visual Studio Code

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

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

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

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



```
Local
  locals: undefined
▷ req: IncomingMessage
▷ res: ServerResponse
▷ this: #<Object>
```

```
Paused on breakpoint.
module.exports      newticket.js  6
handle              layer.js      95
next                route.js      131
exports.requireUser
handle              layer.js      95
next                route.js      131
next                route.js      125
next                route.js      125
next                route.js      125
next                route.js      125
next                route.js      125
next                route.js      125
```

```
☐ All exceptions
☑ Uncaught exceptions
☑ newticket.js  6   routes\views\tickets
☑ newticket.js  16  routes\views\tick...
```

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

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

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

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

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

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

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


    });

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

```
GET /keystone/signin 200 591.610 ms
GET /signin 304 126.978 ms
POST /signin 200 270.668 ms
POST /signin - - ms
POST /signin 200 5691.394 ms
POST /signin 200 176.050 ms
POST /signin 302 3111.083 ms
GET /mytickets 404 832.461 ms
GET / 200 132.241 ms
```

Debugging with Visual Studio Code

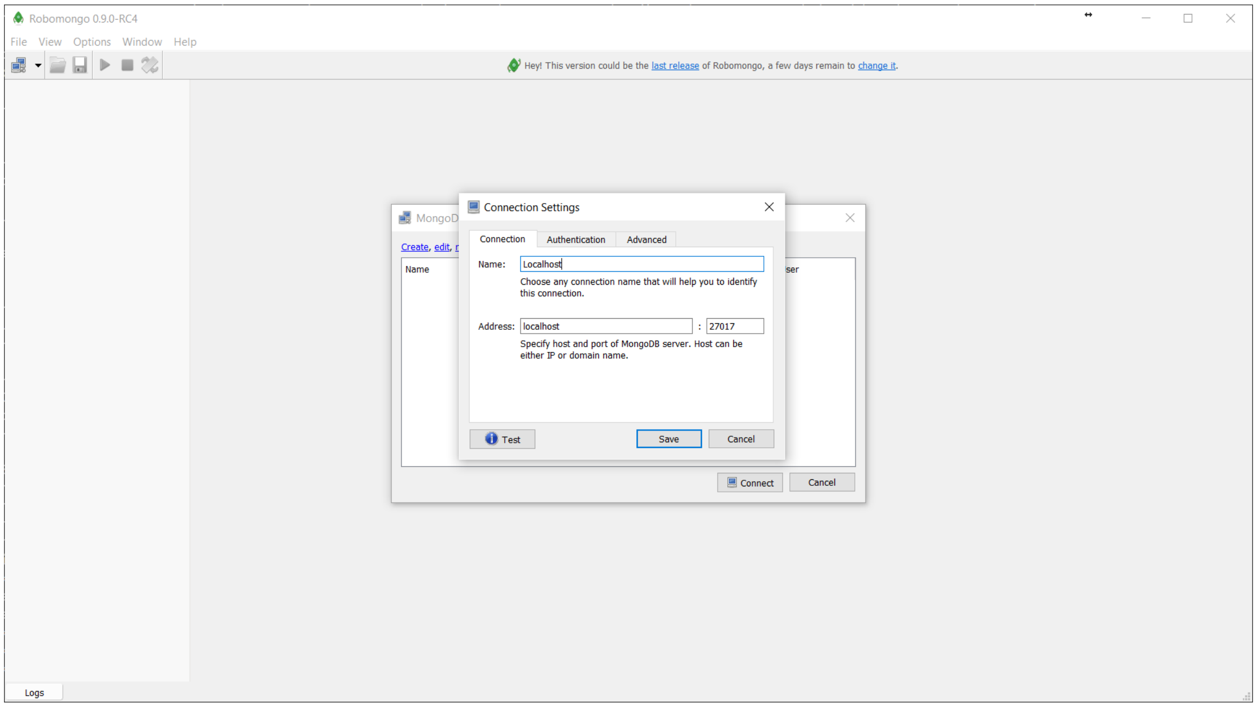
As shown above, we can set breakpoints in code, add variables to watch and inspect the call stack. At the bottom of the editor is the node console where we can see any console interactions like console.log in our application. The node console also allows us to inspect variables inline.

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

The next tool is Robomongo. Robomongo is a desktop application that allows us to manage MongoDB databases. Robomongo runs on Mac OS X, Windows, and Linux and is free!. It allows you to create new databases and view collections and to run queries. It has all the features that the native MongoDB shell provides such as multiple connections, multiple results, and autocompletion. Download and install Robomongo from http://robomongo.org[3].

If you have MongoDB running locally, we can create a new connection as shown below:

[2]https://code.visualstudio.com/docs
[3]http://robomongo.org


Robomongo 0.9.0-RC4
File View Options Window Help
Hey! This version could be the last release of Robomongo, a few days remain to change it.
Connection Settings
Connection
Authentication
Advanced
Test
Save
Cancel
Connect
Cancel
Logs

Name: Localhost

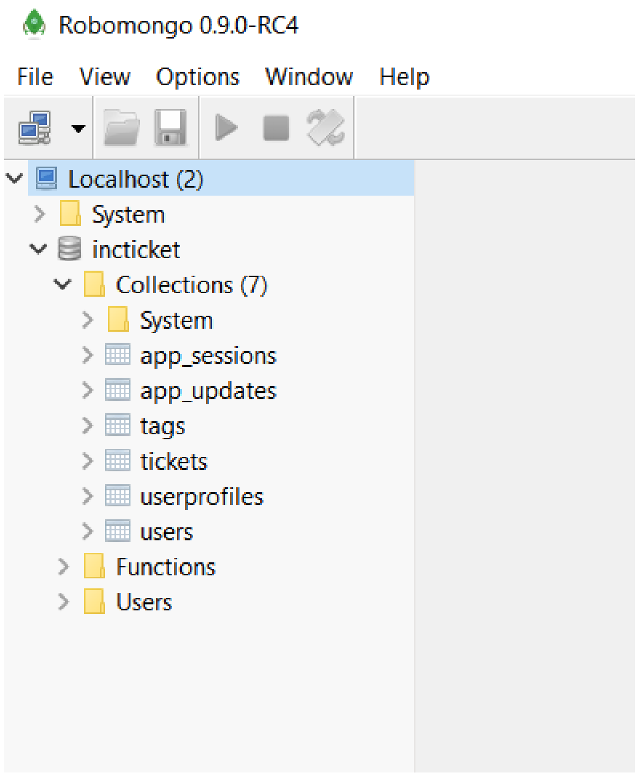
Choose any connection name that will help you to identify this connection.

Address: localhost : 27017

Specify host and port of MongoDB server. Host can be either IP or domain name.

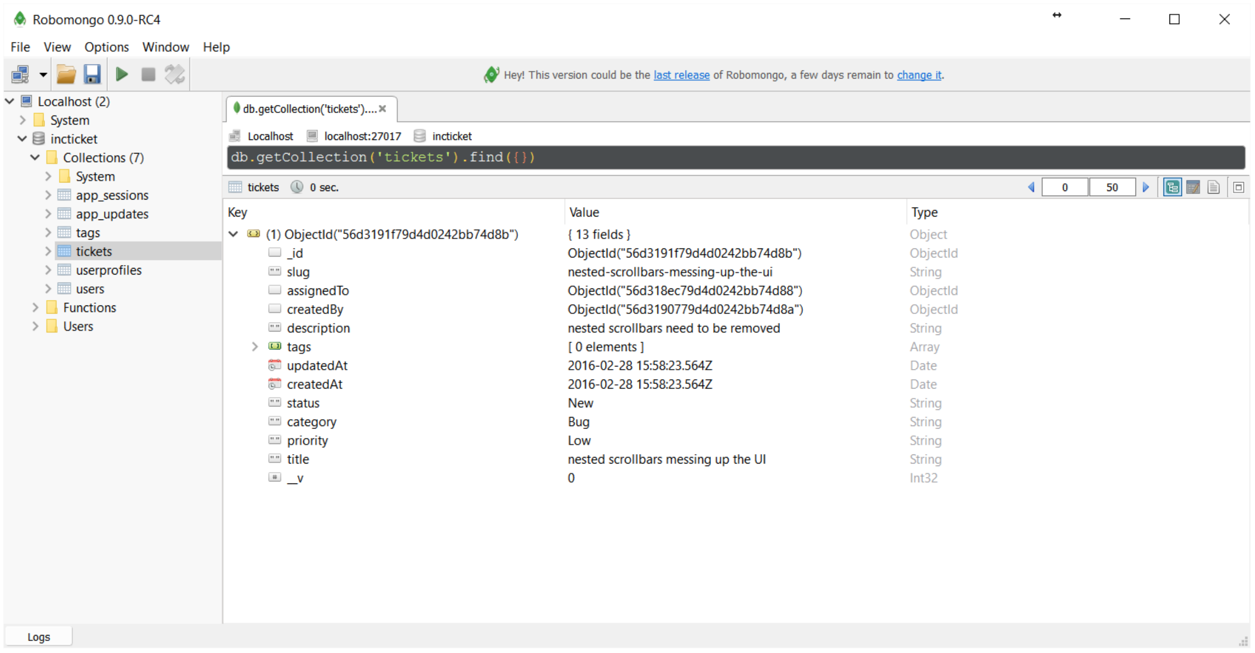
Connect to MongoDB using Robomongo

After connecting to the instance, we can browse all the collections using the sidebar.



Browse MongoDB collections

We can issue queries against our MongoDB and collections using the query bar and visually inspect the returned documents.



```
db.getCollection('tickets').find({})
```

```
Key                                          Value                                          Type
(1) ObjectId("56d3191f79d4d0242bb74d8b")     { 13 fields }                                  Object
    _id                                      ObjectId("56d3191f79d4d0242bb74d8b")           ObjectId
    slug                                     nested-scrollbars-messing-up-the-ui            String
    assignedTo                               ObjectId("56d318ec79d4d0242bb74d88")           ObjectId
    createdBy                                ObjectId("56d3190779d4d0242bb74d8a")           ObjectId
    description                              nested scrollbars need to be removed           String
    tags                                     [ 0 elements ]                                 Array
    updatedAt                                2016-02-28 15:58:23.564Z                       Date
    createdAt                                2016-02-28 15:58:23.564Z                       Date
    status                                   New                                            String
    category                                 Bug                                            String
    priority                                 Low                                            String
    title                                    nested scrollbars messing up the UI            String
    __v                                      0                                              Int32
```

Query MongoDB collections

Summary

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

Chapter 2. Building the IncTicket Application

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

This chapter will cover the following points:

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

Installing Keystone.js

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

Prerequisites

To setup Keystone.js, you are going to need a few prerequisites. These are:

- Node.js
- Yeoman
- NPM
- Mongo DB

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

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

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

First, let's start by installing yo.

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

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

```
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.

Creating a new Keystone.js Application

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

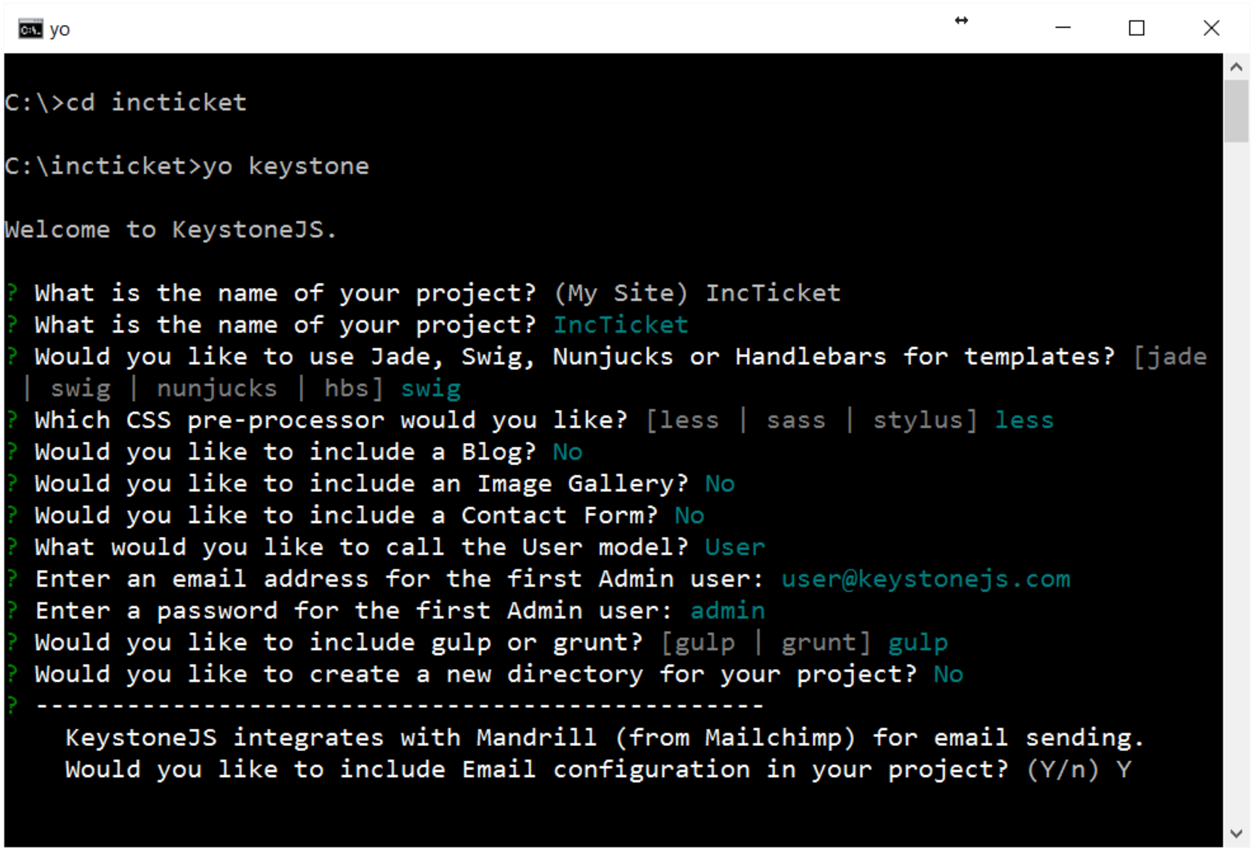
```
c:\> mkdir incticket
```

and change into that directory.

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

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

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


yo

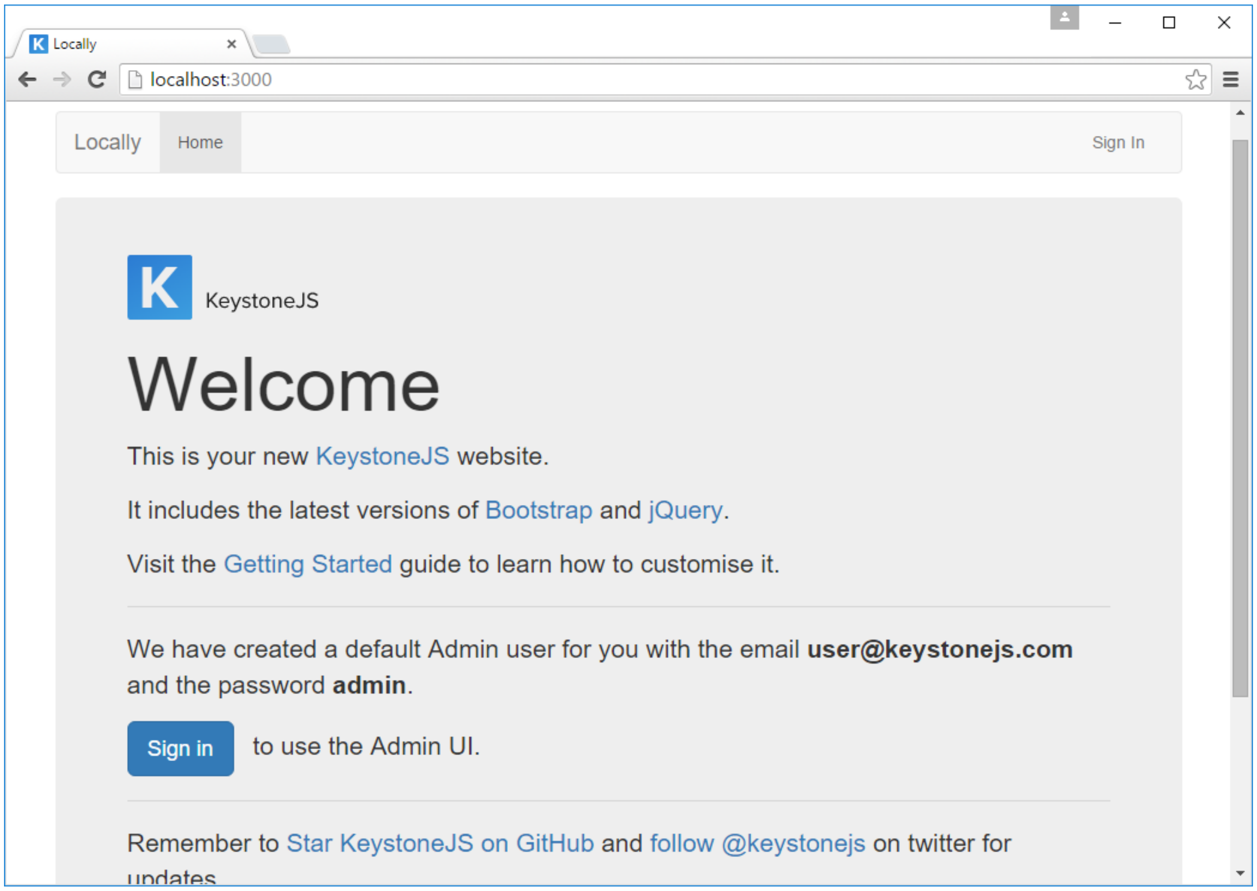
```
C:\>cd incticket

C:\incticket>yo keystone

Welcome to KeystoneJS.

? What is the name of your project? (My Site) IncTicket
? What is the name of your project? IncTicket
? 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 to include 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? (Y/n) Y
```

Create project using yo

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

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

The above command will serve up your project on port 3000. If you navigate to http://localhost:3000, you should see the Keystone.js landing page.



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

Keystone.js Default Landing Page

Configuring Your Keystone.js Application

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

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

- The configuration data can be easily changed between environments and even isolated deploys. This leads to less complex deploys which saves time and money.

- There is a decreased chance of the production data leaking out into the wrong hands which reduces the chances that your database might accidentally be wiped.

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 you do not commit this file to version control.

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

```
COOKIE_SECRET=oQQ*s0pz5(bF4gpmoNwM|BDB~db+qwQ`K>Ik~*R2D;;F(8u["l5<.=&Q9w+Ul$E=
MANDRILL_API_KEY=NY8RRKyv1Bure9bdP8-TOQ
```

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

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

App in Production

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 is pretty easy to do so using the .env file.

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

Project Structure

Let us take a look at the new directory structure to better understand the different parts that make up a Keystone.js project. Below is the default directory structure of a Keystone.js project:

```
|--lib
|
|--models
|
|--public
|
|--routes
|   |--api
|   |
|   |--views
|   |
|   |--index.js
|   |
|   |--middleware.js
|   |
|--templates
|   |--includes
|   |
|   |--layouts
|   |
|   |--mixins
|   |
|   |--views
|   |
|--updates
|
|--package.json
|
|--keystone.js
|
```

Keystone.js Project Structure

- The **lib** directory holds any additional libraries that could be needed by our project.
- The **models** folder holds the data models that our project would need. Example model is a User model that deals with user login and user preferences.
- The **public** directory holds static content, related to the web application, such as images, CSS, fonts, Javascript. The LESS processor included within the framework will make sure that LESS files associated with the web application is compiled to CSS files during runtime.
- The **routes** directory contains index.js, middleware.js, and API & views directories.

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

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

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

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

Creating Your First Model/List

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

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

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

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

Ticket.add({
        title: { type: String, initial: true, default: '', required: true },
        description: { type: Types.Textarea },
        priority: { type: Types.Select, options: 'Low, Medium, High', default: '\
Low' },
        category: { type: Types.Select, options: 'Bug, Feature, Enhancement', de\
fault: 'Bug' },
        status: { type: Types.Select, options: 'New, In Progress, Open, On Hold,\
 Declined, Closed', default: 'New' },
        createdBy: { type: Types.Relationship, ref: 'User', index: true, many: f\
alse },
        assignedTo: { type: Types.Relationship, ref: 'User', index: true, many: \
false },
        createdAt: { type: Datetime, default: Date.now },
        updatedAt: { type: Datetime, default: Date.now }
    });

Ticket.defaultSort = '-createdAt';

Ticket.register();
```

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

The call to register on our keystone.js list finalizes the model with any attributes and options we set.

Let's take a look at some of the fields we defined for our model:

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

displayed as an auto-suggest text box in the admin UI that allows us to pick a single user. Setting the many option to false indicates that only a single user can be selected. Setting the index option to true will tell keystone.js that we are interested in a database index to be created for this field. The assignedTo field is a similar relationship field that is used to store a reference to the user that the ticket is currently assigned to. This is the user that will be incharge of resolving the issue mentioned in the ticket.

- **createdAt**: This datetime field indicates when the ticket was created by the user. Since we are using the default value of Date.now, the date will be saved automatically when creating a new ticket object.

As you can see, Keystone.js comes with different types of fields that you can use to define your models. You can find all field types at http://keystonejs.com/docs/database/#fieldtypes[4]

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

After saving our model, let us restart our application. Use the below command to restart the application via command line

```
node keystone.js
```

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

Creating an administration site for your models

Now that we have defined the Ticket model, let us see how to create an administration site to manage tickets. Keystone.js comes with a built-in administration interface that is very useful for editing content. The Keystone.js admin site is built dynamically by reading them 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.

Creating an admin user

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

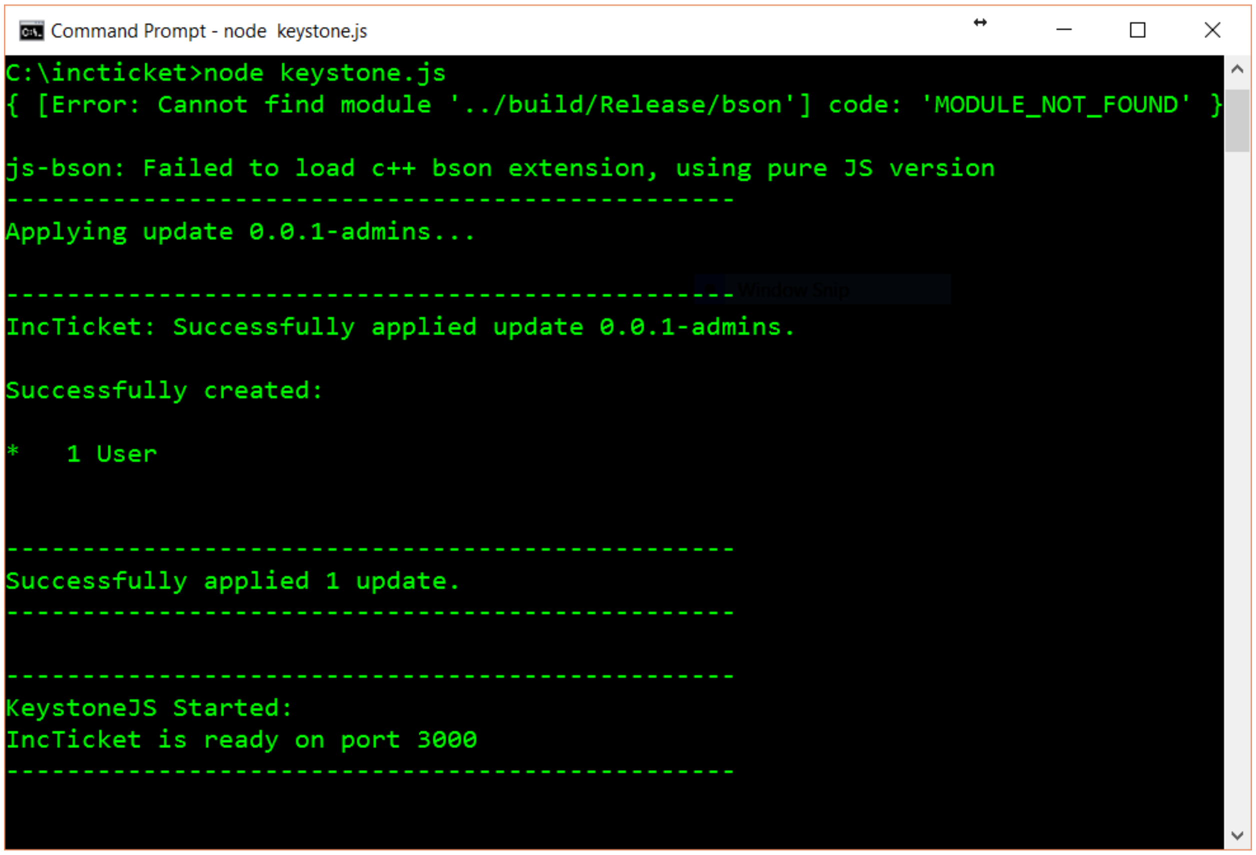
The default admin is created with the below code that is stored at `/updates/0.0.1-admins.js` with the following credentials:

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

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

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

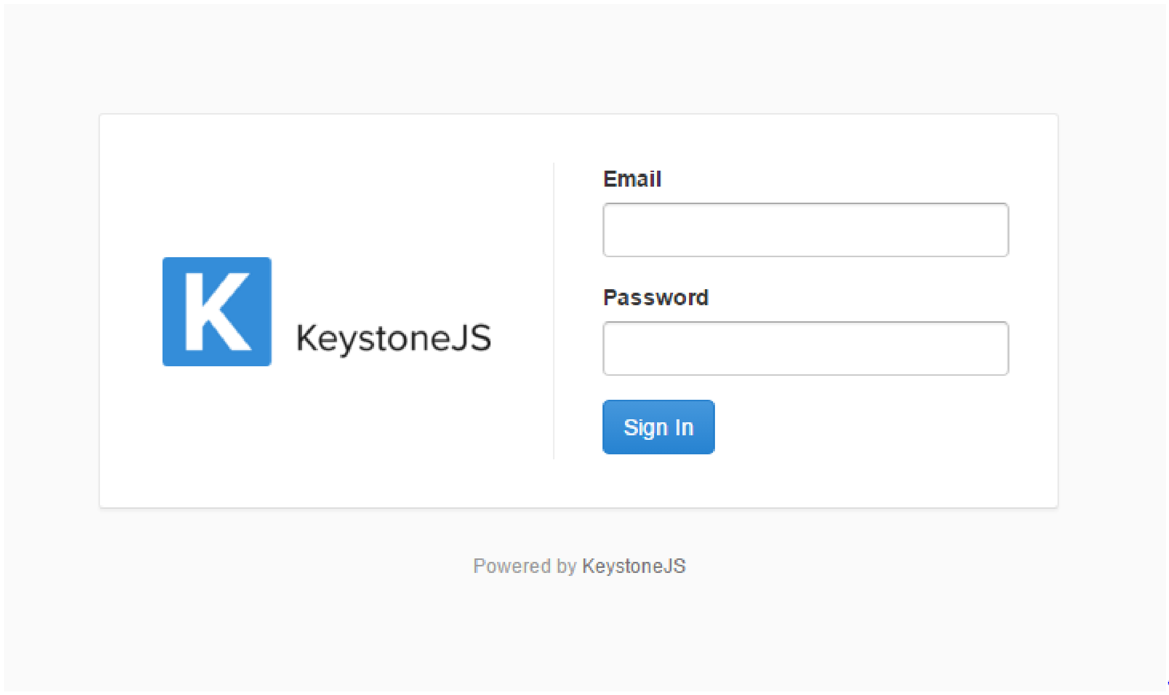
This script automatically creates a default Admin user when an empty database is used for the first time. Updates are run when the app is restarted using `node keystone.js` command. An update is not applied twice, hence editing the file after starting keystone.js atleast once will not result in changes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



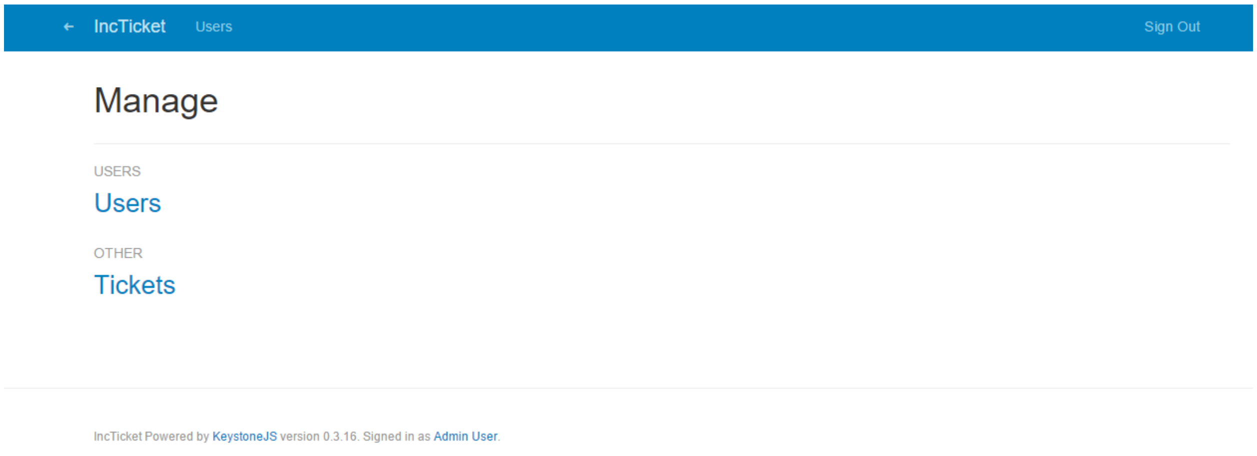
```
C:\incticket>node keystone.js
{ [Error: Cannot find module '../build/Release/bson'] code: 'MODULE_NOT_FOUND' }

js-bson: Failed to load c++ bson extension, using pure JS version
------------------------------------------------
Applying update 0.0.1-admins...

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

Successfully created:

*    1 User


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

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

The Keystone.js administration site

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 below.



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



IncTicket Powered by KeystoneJS version 0.3.16. Signed in as Admin User.

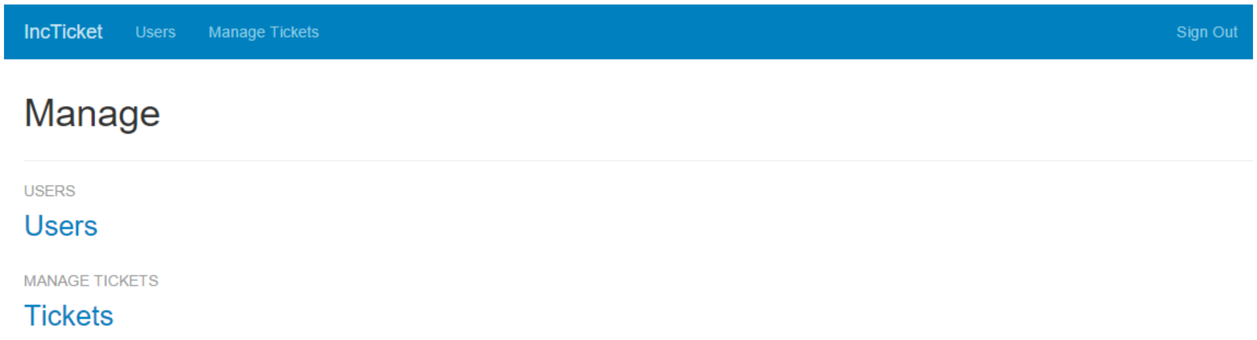
The user model seen on the page is automatically created for us by Keystone.js. If you click on **Users** you will see the admin user created for us. You can edit the admin user's email address and password to suit your needs and use the new credentials to login to the application next time.

Modifying the admin menu

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

```
// Configure the navigation bar in Keystone's Admin UI

keystone.set('nav', {
        'users': 'users',
    'manageTickets': 'tickets'
});
```

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



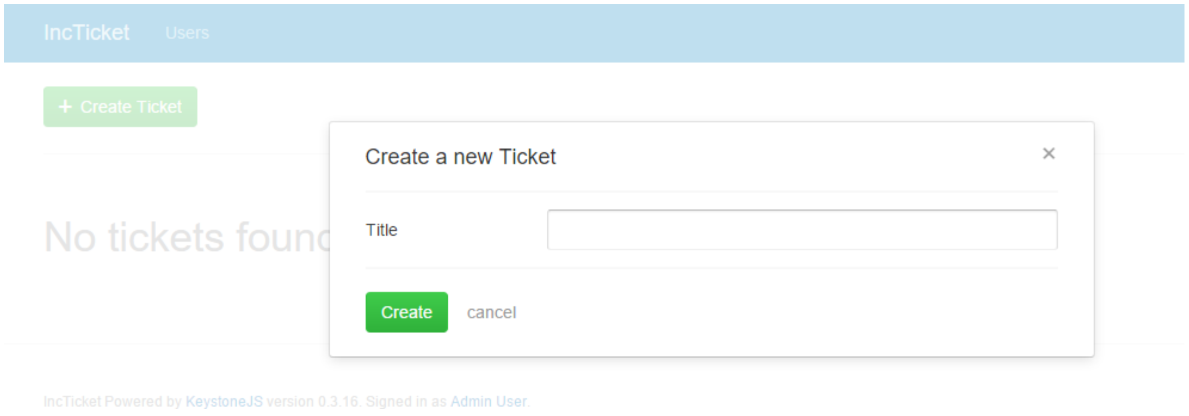
Adding models to the administration site

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

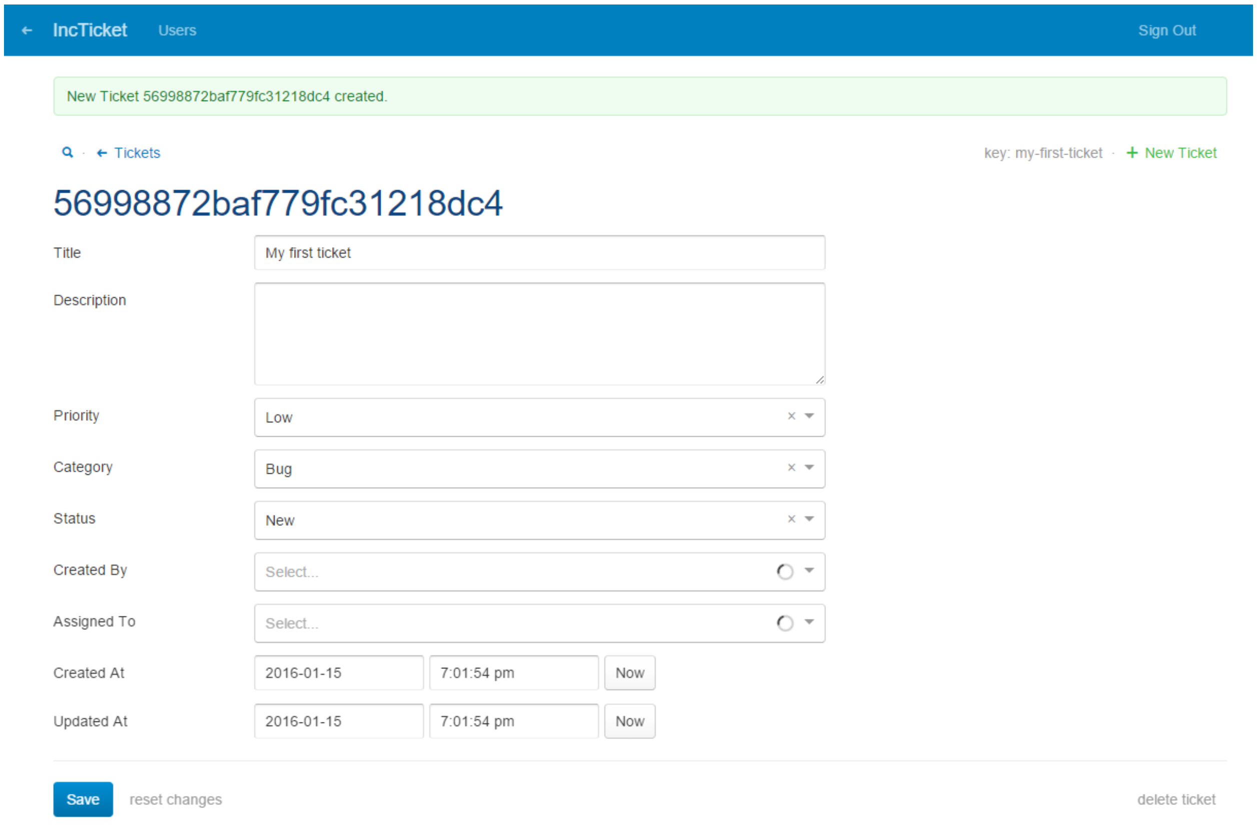
```
var keystone = require('keystone');

var Ticket = new keystone.List(...);
...
Ticket.register();
```

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

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



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



New Ticket 56998872baf779fc31218dc4 created.

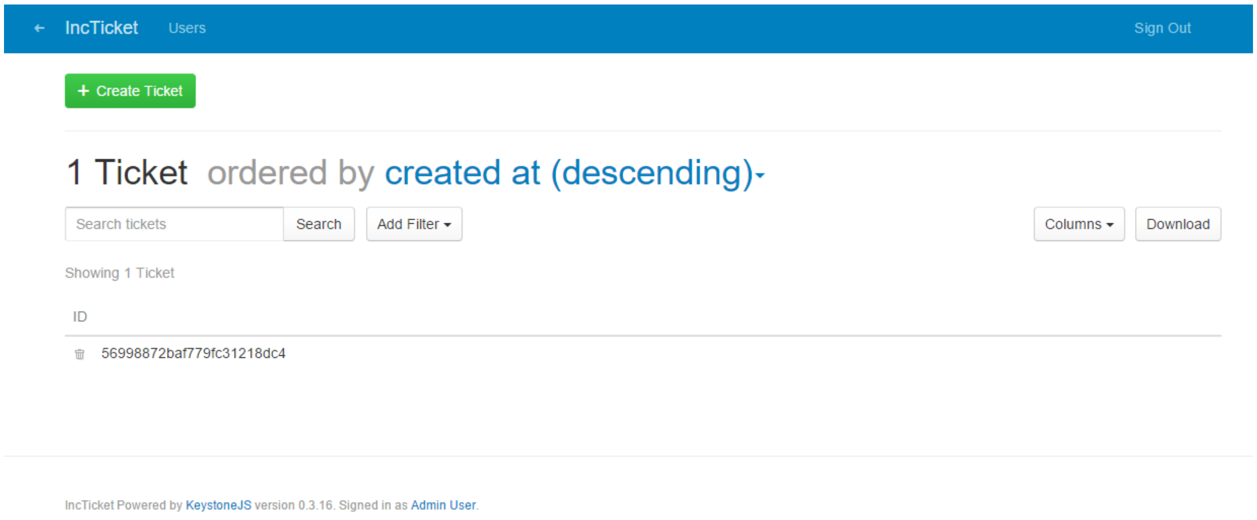
key: my-first-ticket

56998872baf779fc31218dc4

Title	My first ticket
Description	
Priority	Low
Category	Bug
Status	New
Created By	Select...
Assigned To	Select...
Created At	2016-01-15 7:01:54 pm
Updated At	2016-01-15 7:01:54 pm

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

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



1 Ticket ordered by created at (descending)

Showing 1 Ticket

ID

56998872baf779fc31218dc4

IncTicket Powered by KeystoneJS version 0.3.16. Signed in as Admin User.

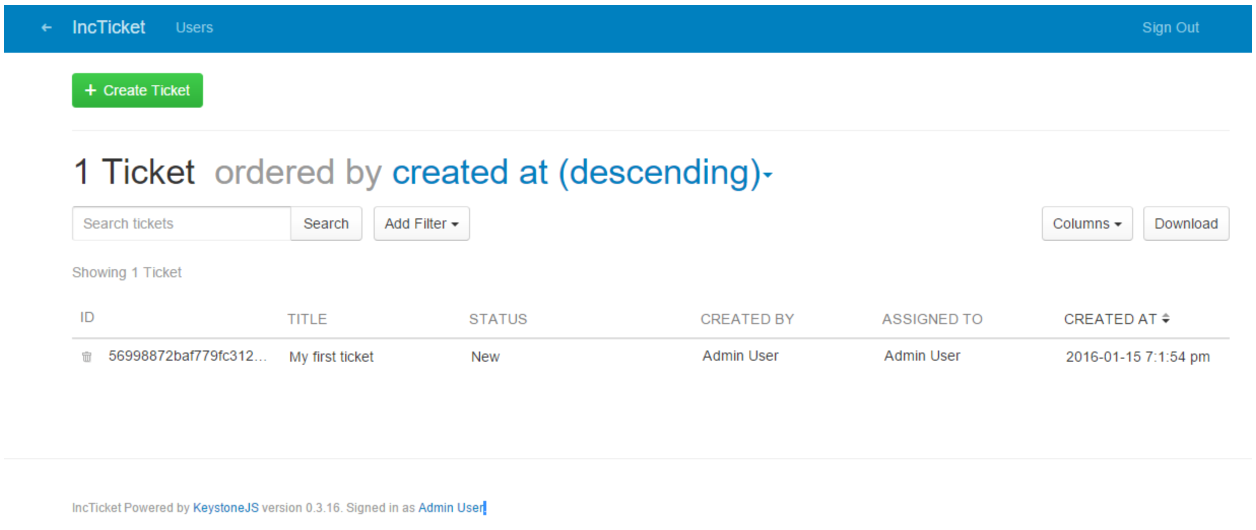
Customizing the way models are displayed

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

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

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

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



1 Ticket ordered by created at (descending)

Showing 1 Ticket

ID	TITLE	STATUS	CREATED BY	ASSIGNED TO	CREATED AT
56998872baf779fc312...	My first ticket	New	Admin User	Admin User	2016-01-15 7:1:54 pm

IncTicket Powered by KeystoneJS version 0.3.16. Signed in as Admin User

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

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

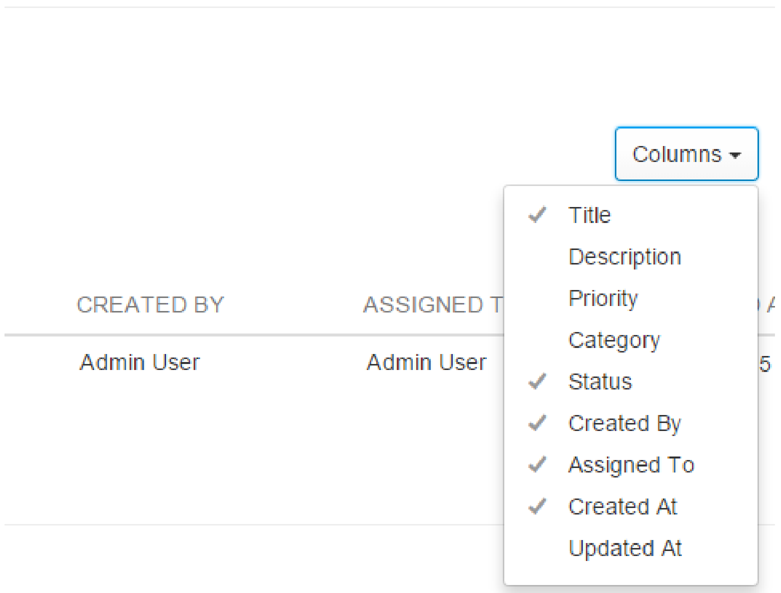
```
Ticket.defaultSort = '-createdAt';
```

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

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

Dynamically adding columns to Admin UI

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



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

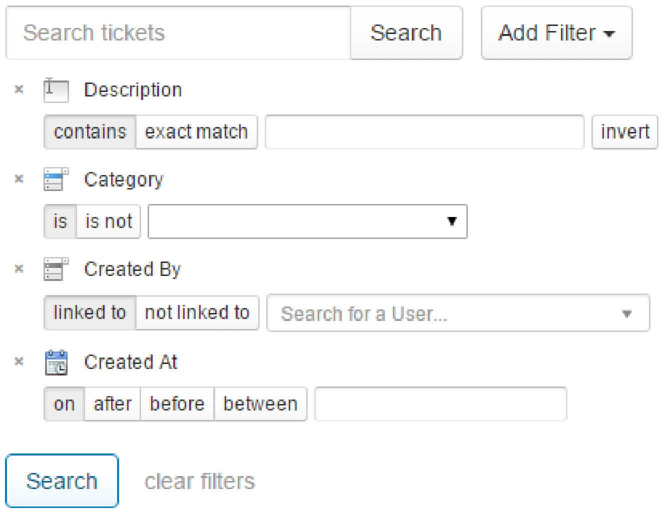
Finding data using the Admin UI

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

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

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

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



A string field such as description is presented with a text box with options for matching exactly or on a contains condition. The invert option will try to find results that negate the selected condition i.e. if exact match option was selected then it will try to find results that dont exactly match the keywords and if contains was selected then it will try to find results that dont contain the keyword.

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

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

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

Creating Your First Route

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

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

Tickets Route

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

```
// Setup Route Bindings
exports = module.exports = function(app) {
...
    app.get('/tickets', function(req, res){
            res.send('We will show a list of tickets here');
    });
...
}
```

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

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

- subscribe
- unsubscribe
- patch
- search
- connect

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

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

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

URLs for models

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

```
Ticket.schema.virtual('url').get(function() {
                return '/tickets/'+this.slug;

});
```

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

Creating Your First View

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

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

Create the following directories and files inside your application directory:

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

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

Creating Ticket list and detail views

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

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

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

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

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

    locals.data = {
                tickets: [],
        };

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

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

                q.exec(function(err, results) {
                        locals.data.tickets = results;
                        next(err);
                });
        });
        // Render the view
        view.render('tickets/ticketlist');

};
```

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

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

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

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

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

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

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

  locals.data = {
        ticket: {},
    };

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

  var q = keystone.list('Ticket').model.findOne({slug: req.params.ticketslug});

        q.exec(function(err, result) {
            if(result != null)
            {
                locals.data.ticket = result;
            }
            else
            {
                return res.status(404).send(keystone.wrapHTMLError('Sorry, no tick
(404)'));
            }

            next(err);
        });
    });


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

};
```

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

Creating templates for your views

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

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

Create the following directories and files inside your application directory:

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

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

Keystone.js supports many templating languages and Swig is one we will look at. Swig is a powerful template language that allows you to specify how data is displayed on the browser. It is based on template tags, which look like `{% tag %}` , template variables, which look like `{{ variable }}` and template filters, which can be applied to variables and look like `{{ variable|filter }}`. You can see all swig template tags and filters at http://paularmstrong.github.io/swig/docs/[5].

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

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

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

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

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

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

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

{% block content %}

        <div class="container">
                <div class="panel panel-primary">
        <!-- Default panel contents -->
        <div class="panel-heading">Tickets</div>
        <div class="panel-body">
            <p>These are list of tickets in the system.</p>
        </div>

        <!-- Table -->
        <table class="table  table-striped">
            {% for ticket in data.tickets %}
               <tr>
               <td>
                <div class='  col-md-1'>
                    <span class="label label-info pull-right">{{ticket.status}}<\
/span>
                </div>

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

                 <ul class="ticket-meta">
                       <li>&nbsp;</li>
                    <li>
                        <small>Status</small><a href="" rel="tag">{{ticket.statu\
s}}</a>
                    </li>
                    <li>
                        <small>Priority</small><a href="" rel="tag">{{ticket.pri\
ority}}</a>
                    </li>
                    <li>
```

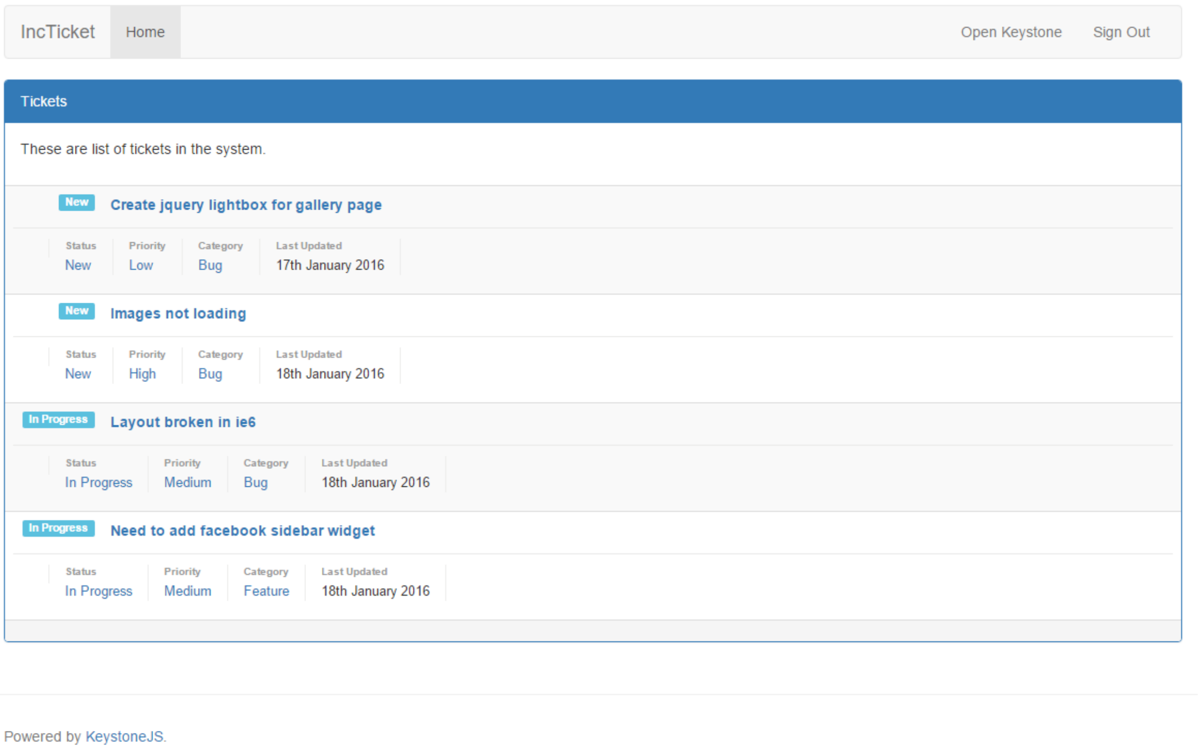
```
                        <small>Category</small><a href="" rel="tag">{{ticket.cat\
egory}}</a>
                    </li>
                    <li>
                                <small>Last Updated</small>
                                <abbr class="last-updated">{{ticket._.createdAt.format('Do MMM\
M YYYY')}}</abbr>
                        </li>
                              </ul>
               </td>
               </tr>
            {% endfor %}
        </table>
        <div class="panel-footer"></div>
        </div>

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

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

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

```
// Setup Route Bindings
exports = module.exports = function(app) {
...
   app.get('/tickets', routes.views.tickets.ticketlist);
...
}
```

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



Tickets

These are list of tickets in the system.

New **Create jquery lightbox for gallery page**

Status	Priority	Category	Last Updated
New	Low	Bug	17th January 2016

New **Images not loading**

Status	Priority	Category	Last Updated
New	High	Bug	18th January 2016

In Progress **Layout broken in ie6**

Status	Priority	Category	Last Updated
In Progress	Medium	Bug	18th January 2016

In Progress **Need to add facebook sidebar widget**

Status	Priority	Category	Last Updated
In Progress	Medium	Feature	18th January 2016

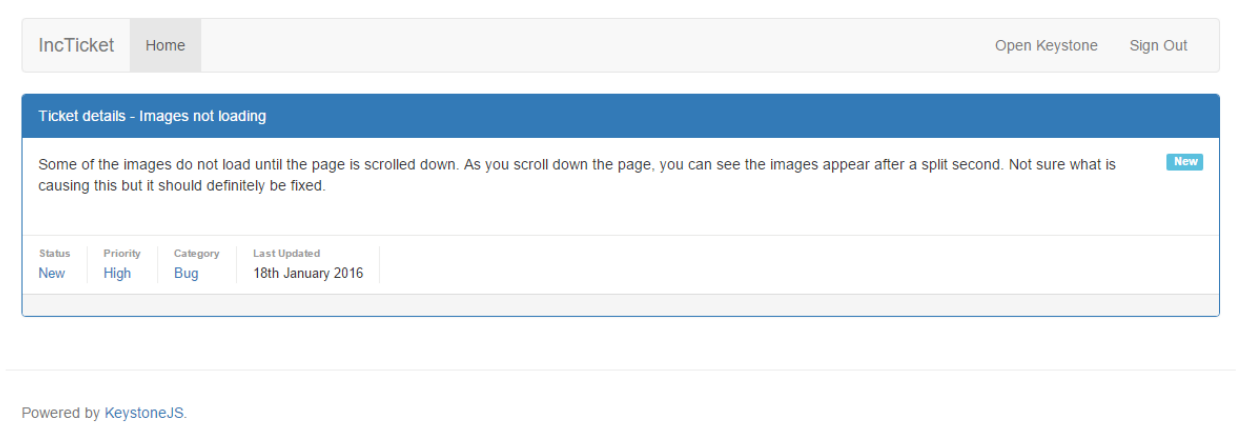
Powered by KeystoneJS.

List of Tickets

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

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

{% block content %}

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

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

Now, you can go back to your browser and click on one of the ticket titles to see the detail view of a ticket. You should see the template rendered like this:



Ticket details - Images not loading

Some of the images do not load until the page is scrolled down. As you scroll down the page, you can see the images appear after a split second. Not sure what is causing this but it should definitely be fixed.

New

Status	Priority	Category	Last Updated
New	High	Bug	18th January 2016

Powered by KeystoneJS.

Ticket detail

Since we have created a SEO friendly URL for our tickets, the URL should look like http://127.0.0.1:3000/tickets/image not-loading.

Adding pagination

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

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

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

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

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

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

    locals.data = {
                tickets: [],
        };

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

                var q = keystone.list('Ticket').paginate({
                    page: req.query.page || 1,
                    perPage: 5,
                    maxPages: 5
                });

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

        });


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

};
```

This is how pagination works:

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

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

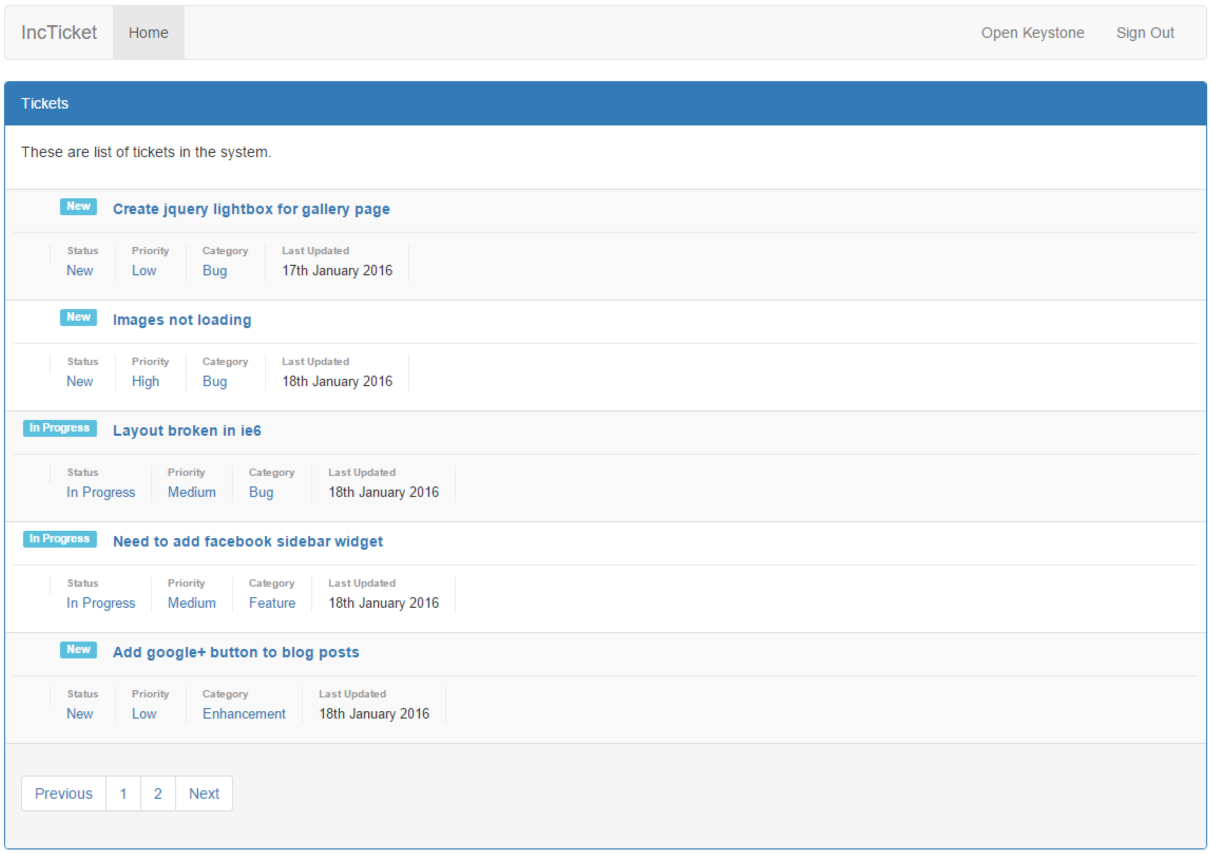
- **total**: all matching results (not just on this page)
- **results**: array of results for this page
- **currentPage**: the index of the current page
- **totalPages**: the total number of pages
- **pages**: array of pages to display
- **previous**: index of the previous page, false if at the first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

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

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

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

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

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

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

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

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



These are list of tickets in the system.

New **Create jquery lightbox for gallery page**

Status	Priority	Category	Last Updated
New	Low	Bug	17th January 2016

New **Images not loading**

Status	Priority	Category	Last Updated
New	High	Bug	18th January 2016

In Progress **Layout broken in ie6**

Status	Priority	Category	Last Updated
In Progress	Medium	Bug	18th January 2016

In Progress **Need to add facebook sidebar widget**

Status	Priority	Category	Last Updated
In Progress	Medium	Feature	18th January 2016

New **Add google+ button to blog posts**

Status	Priority	Category	Last Updated
New	Low	Enhancement	18th January 2016

List tickets with pagination

Summary

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