



Community Experience Distilled

Sails.js Essentials

Get up to speed with Sails.js development with this fast-paced tutorial

Shaikh Shahid

[PACKT] open source*
PUBLISHING community experience distilled

Sails.js Essentials

Get up to speed with Sails.js development with this fast-paced tutorial

Shaikh Shahid



BIRMINGHAM - MUMBAI

Sails.js Essentials

Copyright © 2016 Packt Publishing

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

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2016

Production reference: 1190216

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

ISBN 978-1-78355-453-9

www.packtpub.com

Credits

Author

Shaikh Shahid

Project Coordinator

Nidhi Joshi

Reviewer

Diogo Resende

Proofreader

Safis Editing

Commissioning Editor

Kartikey Pandey

Indexer

Monica Ajmera Mehta

Acquisition Editor

Sonali Vernekar

Graphics

Jason Monteiro

Kirk D'Penha

Content Development Editor

Siddhesh Salvi

Production Coordinator

Conidon Miranda

Technical Editor

Manthan Raja

Cover Work

Conidon Miranda

Copy Editor

Vibha Shukla

About the Author

Shaikh Shahid has been a product developer for over two years. He has the experience of working on Node.js for more than two years. He loves to spread the word about Node.js and its various packages via his programming blog. Shahid is also very interested in software architecture and design and loves to develop software system from the core.

When he is not playing with Node.js or helping people, he watches movies, reads books, or travels to Goa.

I'd like to thank my father, who was not aware of my experiments with Node.js and similar things during my college days, and supported me at every step. I'd also like to thank Ashutosh sir and Jane for giving me an opportunity to work professionally on this awesome technology.

About the Reviewer

Diogo Resende is a passionate developer, obsessed with perfection in everything he works on. He loves everything about the Internet of Things: the ability to connect everything together and always being connected to the world. He is also the author of *Node.js High Performance*, Packt Publishing.

Diogo Resende studied computer science and graduated in engineering, which deepened his knowledge about computer networking and security, software development, and cloud computing. In the past 10 years, he has embraced different challenges to develop applications and services to connect people with embedded devices around the world, creating a bridge between the ancient and uncommon protocols and the Internet of today.

www.PacktPub.com

eBooks, discount offers, and more

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

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



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

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

Why subscribe?

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

Table of Contents

Preface	v
Chapter 1: Revisiting Node.js Concepts	1
Node.js architecture	2
V8	2
Event driven I/O – libuv	2
Single-threaded system and its working	3
Working of libuv – core of Node.js	3
Multi-threading versus single-threading	4
Event loop and non-blocking I/O model	5
Importance of event loop	5
Working of event loop	6
Summary	7
Chapter 2: Developing Node.js Web Server	9
Working of web servers	9
HTTP operations and their use	10
Create	10
Read	10
Update	10
Delete	10
Developing web server using HTTP module	11
HTTP headers and content-type	12
Developing web server using Express	12
Using Express to develop web server	12
Routers and middleware	16
Summary	17

Chapter 3: Introduction to Sails.js and MVC Concepts	19
Getting started with MVC concepts	19
Model	20
View	20
Controller	20
Installing Sails.js	21
Understanding directory structure of Sails.js project	22
The assets directory	23
The views directory	23
The node_modules directory	23
The api directory	23
The config directory	24
Adding database support	25
Configuring MySQL database with Sails.js	25
config/connections.js	26
config/models.js	26
Configuring MongoDB database with Sails.js	26
config/connections.js	27
config/models.js	27
Configuring the Grunt task runner file with JSHint	27
Summary	29
Chapter 4: Developing REST API Using Sails.js	31
Why it is called REST?	31
The REST CRUD operation	32
Database design for REST API	32
Building REST API in Sails.js	33
config/connections.js	34
config/models.js	34
Discussing migrate key	34
Running our code	35
Create new message	35
Read the message	37
Update the message	38
Delete the message	39
Defining custom controller	40
api/controllers/MessageController.js	40
Summary	40
Chapter 5: Build a Chat System Using Sails.js	41
Application architecture and flow	42
Creating a Sails.js app	42

Sails.js API for chat	44
Model definition and MySQL integration in the app	44
Sails.js controller to handle the chat operation	46
AngularJS app for client-side interaction	47
Running the application	52
Summary	54
Chapter 6: Building a Real-Time News Feed App Using Sails.js	55
Briefing Socket.IO	56
Using Socket in Sails.js	56
Discussing the database design of the app	57
Implementing the application	58
Summary	66
Chapter 7: Creating a TODO Single-Page Application	67
MongoDB support in Sails.js	68
Defining model for API	69
TODO app view design	72
/assets/js/app.js	74
/assets/js/services/ToDoService.js	75
Summary	77
Chapter 8: Sails.js Production Checklist	79
Sails.js migrate in detail	79
Sails.js security checklist	80
CSRF	80
CORS	81
DDOS	81
XSS	81
Sails.js deployment checklist	82
Configure production environment setting	82
Run app on port 80 if there is no proxy	82
Configure database settings	83
Estimate the traffic from all the endpoints	83
Sails.js hosting	83
Summary	83
Index	85

Preface

Sails.js Essentials will take you through the basics of Node.js and developing production-ready application in the Sails.js framework. This book covers interesting application and their development that will guide you through the practical aspects of software and development.

What this book covers

Chapter 1, Revisiting Node.js Concepts, takes you through some core concepts of Node.js and its working before we dive into the Sails.js and MVC concepts.

Chapter 2, Developing Node.js Web Server, explain how servers are built in Node.js. Throughout this book, we will deal with various web servers. Sails.js has an internal web server that is already written for production.

Chapter 3, Introduction to Sails.js and MVC Concepts, covers MVC concepts in brief and begins with the Sails.js installation and configuration.

Chapter 4, Developing REST API Using Sails.js, comes up with tools to build REST API faster and easier. REST APIs are essential building blocks of any web application.

Chapter 5, Build a Chat System Using Sails.js, covers how to develop a chat system using Sails.js. A chat system is very generic application across web applications.

Chapter 6, Building a Real-Time News Feed App Using Sails.js, teaches how to develop basic news feed app using Sails.js. Facebook and Twitter have very nice news feeds, which are updated as soon as a new status is needed.

Chapter 7, Creating a TODO Single-Page Application, covers how to develop a TODO application using Sails.js. TODO application needs no introduction as it's a famous application.

Chapter 8, Sails.js Production Checklist, covers how to choose Sails.js hosting and some tweaks and settings before hitting the deploy button.

What you need for this book

A computer with a generic operating system (Mac, Windows, or Linux) having capability to run Node.js and npm (node package manager).

Who this book is for

This book targets web developers, who want to build web apps with Sails.js.

Proficiency with JavaScript and Node.js is assumed along with familiarity of web development concepts. Familiarity with the MEAN (MongoDB, Express, AngularJS, and Node.js) stack is an added advantage.

Conventions

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

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You can validate the same concept using the `setTimeout()` function."

A block of code is set as follows:

```
console.log("i am first");
setTimeout(function timeout() {
  console.log("i am second");
}, 5000);
console.log("i am third");
```

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

```
npm install sails -g
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "Now, hit the **Send** button and see the response."

 Warnings or important notes appear in a box like this. 

 Tips and tricks appear like this. 

Reader feedback

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

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

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

Customer support

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

Downloading the example code

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

You can download the code files by following these steps:

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

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

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

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/SailsjsEssentials_ColorImages.pdf.

Errata

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

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

Piracy

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

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

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

Questions

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

1

Revisiting Node.js Concepts

Node.js – the game changer of server-side programming – is becoming popular day by day. Many popular frameworks such as **Express.js**, **Sails.js**, and **Mean.io** are developed on top of Node.js and software giants such as Microsoft, PayPal, and Facebook are shipping the production-ready applications that are stable like a rock!

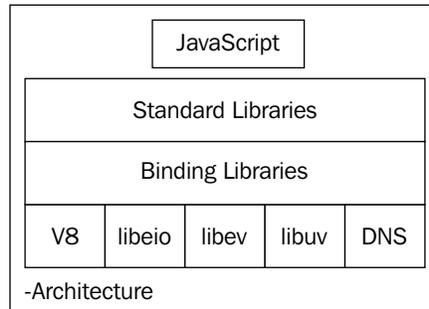
You might be aware about the approach that Node.js used, such as the event-driven programming, single-thread approach, and asynchronous I/O. How does Node.js really work? What's its architecture? How does it run asynchronous code?

In this chapter, we will see the answers to the preceding questions and cover the following aspects of Node.js:

- Node.js architecture
- Single-threaded system and its working
- Event loop and non-blocking I/O model

Node.js architecture

We all know that Node.js runs on top of V8 – Chrome runtime engine – that compiles the JavaScript code in the native machine code (one of the reasons why Google Chrome runs fast and consumes a lot of memory), followed by the custom C++ code – the original version has 8,000 **lines of code (LOC)** – and then, the standard libraries for programmers. The following is the figure of Node.js architecture:



V8

The V8 JavaScript engine is an open source JavaScript engine developed for the Chrome project. The innovation behind V8 is that it compiles the JavaScript code in native machine code and executes it. The developers used the **just-in-time (JIT)** compiler methodology to improve the code compilation time. It is open source and is used in the Node.js and MongoDB project.

Event driven I/O – libuv

The **libuv** library is a cross platform library that provides an asynchronous I/O facility by enabling an event-driven I/O operation. The libuv library creates a thread for the I/O operation (file, DNS, HTTP, and so on) and returns callback. Upon completion of the particular I/O operation, it returns the events so that the callee program does not have to wait for the completion of I/O operation. We will see more about libuv in the upcoming sections.

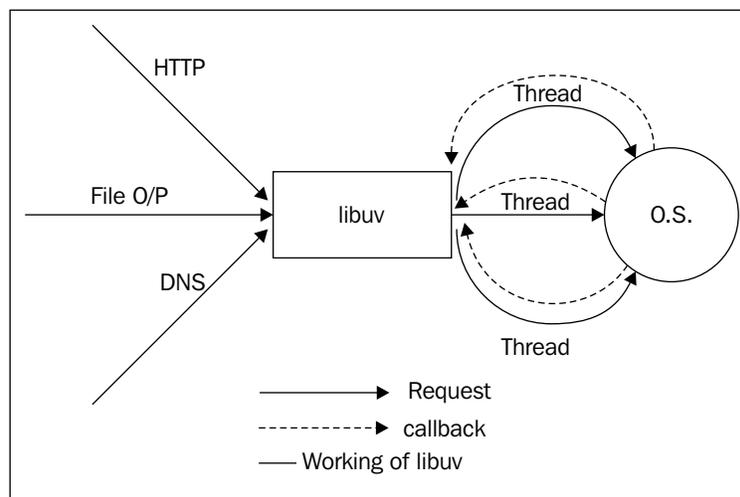
Single-threaded system and its working

Unlike Java, PHP, and other server-side technologies, Node.js uses single-threading over multi-threading. You might wonder how can a thread can be shared across a lot of users concurrently? Consider that I have developed a web server on Node.js and it is receiving 10,000 requests per second. Is Node.js going to treat each connection individually? If it does so, the performance would be low. Then, how does it handle concurrency with a single-thread system?

Here, libuv comes to the rescue.

Working of libuv – core of Node.js

As we mentioned in the previous section, libuv assigns threads for the I/O operation and returns the callback to the callee program. Therefore, Node.js internally creates threads for I/O operation; however, it gives the programmer access to a single runtime thread. In this way, things are simple and sweet:



When you make an HTTP request to web server running over Node.js. It creates the libuv thread and is ready to accept another request. As soon as the events are triggered by libuv, it returns the response to user.

The libuv library provides the following important core features:

- Fully featured event loop
- Asynchronous filesystem operations
- Thread pool

- Thread and synchronization primitives
- Asynchronous TCP and UDP sockets
- Child process
- Signal handling

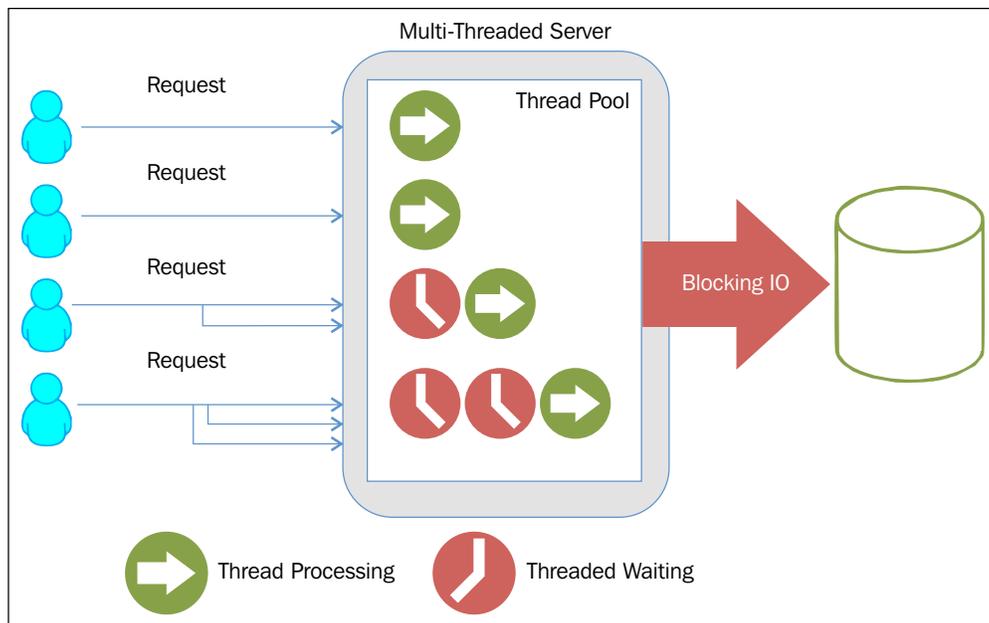
The libuv library internally uses another famous library called **libeio**, which is designed for threading and asynchronous I/O events and **libev**, which is a high-performance event loop. Therefore, you can treat libuv as a package wrapper for both of them.

Multi-threading versus single-threading

Multi-threading approach provides parallelism using threads so that multiple programs can simultaneously run. With advantages come the problems too; it is really difficult to handle concurrency and deadlock in a multi-threading system.

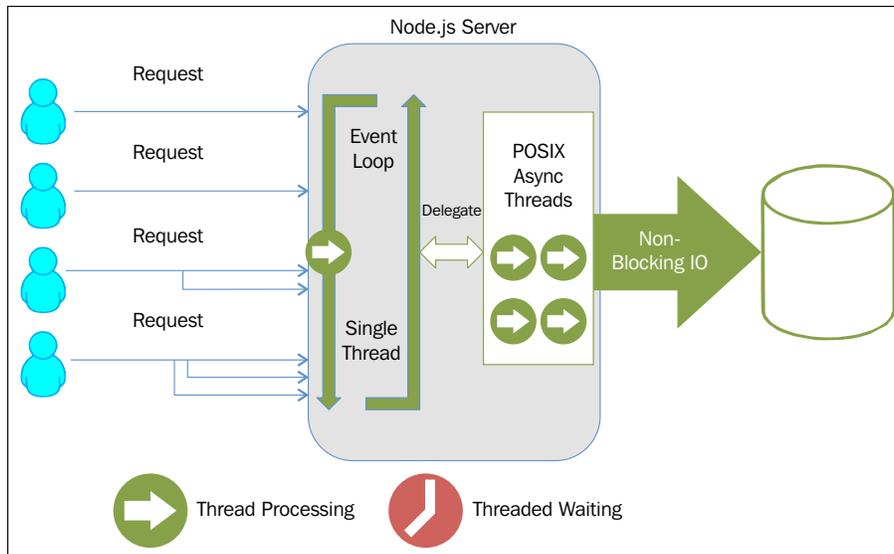
On the other hand, with single-threading, there is no chance of deadlock in the process and managing the code is also easy. You can still hack and busy the event loop for no reason; however, that's not the point.

Consider the following working diagram that is developed by StrongLoop – one of the core maintainers of Node.js:



Node.js uses single-threading for runtime environment; however, internally, it does create multiple threads for various I/O operations. It doesn't imply that it creates threads for each connection, libuv contains the **Portable Operating System Interface (POSIX)** system calls for some I/O operations.

Multi-threading blocks the I/O until the particular thread completes its operation and results in an overall slower performance. Consider the following image:



If the single-threading programs work correctly, they will never block the I/O and will be always ready to accept new connections and process them.

Event loop and non-blocking I/O model

As shown in the previous diagram, I/O does not get blocked by any thread in Node.js. Then, how does it notify to particular processes that the task has been done or an error has occurred? We will look at this in detail in this section.

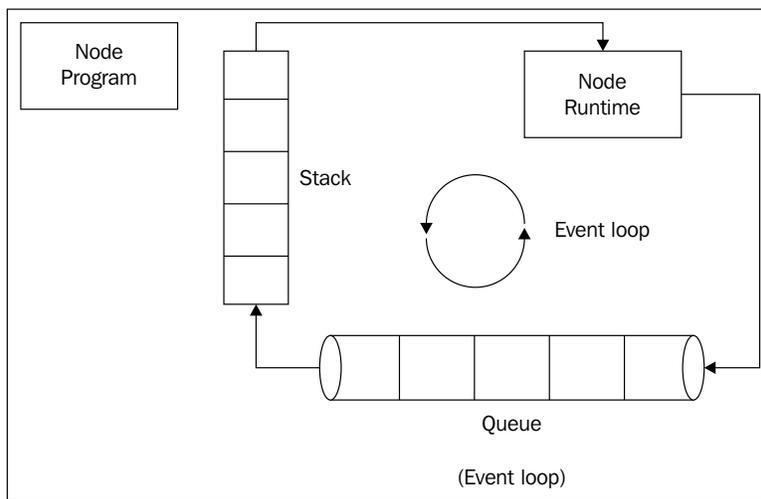
Importance of event loop

Node.js is asynchronous in nature and you need to program it in an asynchronous way, which you cannot do unless you have a clear understanding of event loop. If you know how the event loop works, you will no longer get confused and hopefully, never block the event loop.

Working of event loop

The Node.js runtime system has execution stack, where it pushes every task that it wishes to execute. Operating system *pops* the task from the execution stack and conducts the necessary action required to run the task.

To run the asynchronous code, this approach won't work. The libuv library introduces queue that stores the callback for each asynchronous operation. Event loop runs on specific interval, which is called *tick* in the Node.js terminology, and check the stack. If the stack is empty, it takes the callback from queue and pushes it in the stack for execution, as shown in the following figure:



The libuv library creates the thread and returns the callback to us. As it's an asynchronous operation, it goes to queue instead of the stack and the event loop fetches it when the stack is empty and does the execution.

You can validate the same concept using the `setTimeout()` function.

Consider the following code:

```
console.log("i am first");

setTimeout(function timeout() {
  console.log("i am second");
}, 5000);

console.log("i am third");
```

If you run the previous code, you will get an output similar to the following:

```
i am first
i am third
i am second
```

The reason is obvious, `setTimeout()` waits for five seconds and prints its output; however, that does not block the event loop.

Let's set the timer to 0 seconds and see what happens:

```
console.log("i am first");

setTimeout(function timeout() {
  console.log("i am second");
}, 0);

console.log("i am third");
```

The output is still the same:

```
i am first
i am third
i am second
```

Why so? Even if you set the timer to 0, it goes in the queue; however, it is immediately processed as its time is 0 second. The event loop recognizes that the stack is still not empty, that is, third console was in process; therefore, it pushes the callback after the next tick of event loop.

Summary

In this chapter, we discussed the architecture of Node.js, followed by its internal components: V8 and libuv. We also covered the working of event loop and how Node.js manages the performance improvement using single-thread processing.

In the next chapter, we will take a look at the development of the Node.js server using core modules as well as the **Express** web framework.

2

Developing Node.js Web Server

Node.js comes up with a web server module, which is in the binding section of its architecture. It provides you the necessary tools to build your own efficient web server that runs with single thread mechanism. With nearly any web application; you need web server to serve your application to public/private requests.

In this chapter, we will discuss the following points.

- Working of web servers
- HTTP operations and their uses
- Developing web server using HTTP module
- Developing web server using Express

Working of web servers

Web servers are computers with powerful configuration, sitting somewhere in secure private zone, serving web pages to you. When you enter any **Uniform Resource Locator (URL)**, there are some processes that take place in the background and take time depending on your bandwidth and other factors. What are these steps? How does the browser fetch these pages as soon as you enter a URL? Here is what happens internally:

1. When you enter a URL, your browser will first send a request to name server, which does the task of translating that domain name to the server IP address (it could also be a proxy).
2. Then, the browser sends an HTTP/GET request to port 80 of that IP address.

3. Web server receives the GET request and delivers the file (usually HTML) to the browser and the browser renders it to your display. This happens every time you send a request.

This whole operation is performed by the HTTP and TCP/IP protocol. TCP/IP does the network-level task, while HTTP mainly deals with serving request/response. You might be aware of Apache and NGINX—two of the most famous servers of all time. Well, now it's time to develop our own server!

HTTP operations and their use

HTTP provides various request types such as **GET**, **POST**, **PUT**, **DELETE**, and **HEAD**. Each of the request types is designed to provide a certain kind of operation. You may have heard of the **create, read, update, and delete (CRUD)** operations for various sites and programming blogs. Therefore, let's map these CRUD operations to our HTTP request types.

Create

The create operation generates some new data at the server end. The POST HTTP operation is meant to use when we are generating data. Therefore, in case you want to create a new user or message, you should use POST.

Read

The read operation is directly mapped to the GET operation of HTTP. You should always use the GET request type to read anything from the server. Whether it is for every set of data or a particular one, use GET.

Update

To update any existing data in the server, you should use the PUT HTTP operation. PUT is quite similar to the POST operation; however, its internal working is quite different. You can use this to perform update of any particular data in server.

Delete

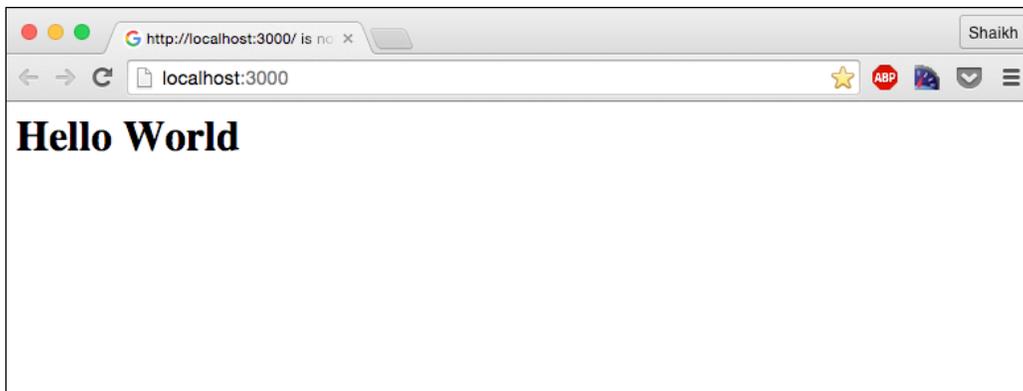
This is directly mapped to the DELETE HTTP operation. You can either delete everything or particular information based on your requirement. The DELETE operation works with the PARAMS, URL-encoded data as well as JSON-encoded data.

Developing web server using HTTP module

HTTP is the core module of Node.js. We can use this module to develop our custom web server, which can accept requests from different HTTP request types and send response to them with various headers. The following is the basic code snippet of a very simple web server in Node.js:

```
var http = require("http");
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-type' : 'text/html'});
  res.end("<h1>Hello World</h1>");
}).listen(3000);
console.log("Listening at Port 3000");
```

When we execute the preceding program and visit `localhost:3000` to view the app, we get the following output:



You can define more HTTP operations and paths for your server, which we refer to as **routes** in this book. For example, in the `http://www.example.com/about` URL, `/about` is the route.

We will not go much deeper in it as we will be using Sails.js that has a built-in efficient server. Let's consider some of the HTTP headers and content-type to understand the upcoming examples easily.

HTTP headers and content-type

HTTP is a flexible protocol and provides interface to deal with the various kinds of data, such as text, HTML, XML, images, and so on. To support all of them in various HTTP operations, it has predefined set of headers and content-type.

Headers are basically an indicator of what kind of data is send to the client. It can be send to the client before sending the actual data so that the browser is ready to accept this format of data.

For example, consider the following:

- To send an HTML response to the client, we need to set following header:
`res.writeHead(200,{'Content-type' : 'text/html'});`
- To send a JSON or XML response to the client, we need to set the following header:

```
res.writeHead(200,{'Content-type' : 'application/json'});  
res.writeHead(200,{'Content-type' : 'application/xml'});
```

Developing web server using Express

Express is a web development framework built on top of Node.js. Express is also the core module used in developing Sails.js. Before getting into the detail of Sails.js, let's briefly go through Express.

Using Express to develop web server

Let's develop our web server using Express. The following is a sample package.json:

```
{  
  "name": "expressServer",  
  "version": "0.0.1",  
  "dependencies": {  
    "express": "^4.13.3"  
  }  
}
```

Switch to the project directory and run the `npm install` command in order to install Express in the project. Let's develop our basic server file, as follows:

```
// load express module.
var express = require("express");
var app = express();
// Always use express inbuilt router.
var router = express.Router();

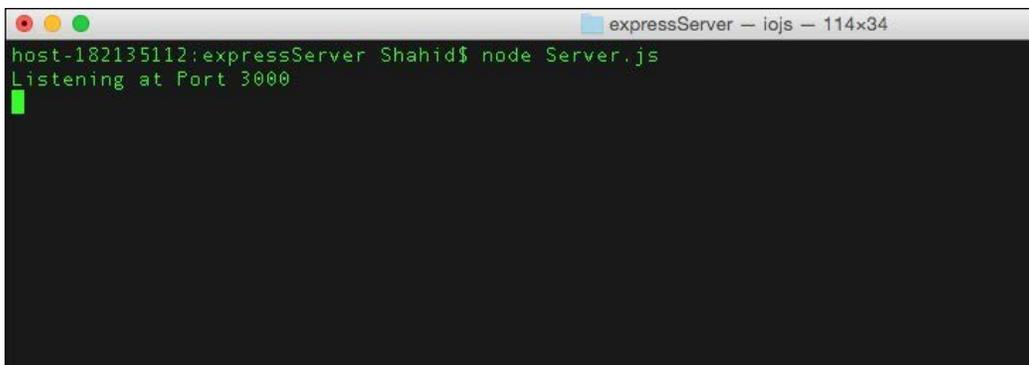
router.get('/',function(req,res){
  // Express determines the common header type.
  res.end("<h1>Hello World</h1>");
});

// This will navigate all router to proceed /home
app.use('/home',router);

app.listen(3000);

console.log("Listening at Port 3000");
```

Run the project using the `node Server.js` command and you will be able to see the following on the console:

A screenshot of a terminal window titled "expressServer -- iojs -- 114x34". The terminal shows the command "node Server.js" being executed, resulting in the output "Listening at Port 3000". The prompt "host-182135112:expressServer Shahid\$" is visible above the command. A green cursor is positioned at the end of the output line.

Open your browser and type `localhost:3000/home` in the browser. Notice how `/home` renders the first route:



You can also send HTML files as a response. All you need is to create the HTML file in the project directory and provide an absolute path in the `res.sendFile()` function.

Here is the code snippet of the HTML file:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Home page</title>
  </head>
  <body>
    <div id="home">
      <h1>Hello World</h1>
      <h3>I am sent as HTML response to you.</h3>
    </div>
  </body>
</html>
```

Downloading the example code



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

I am sure that you are familiar with the basic syntax of HTML. This is just a simple div with heading tags. Take a look at the following modified server file:

```
var express = require("express");
var app = express();
var router = express.Router();

router.get('/', function(req, res) {
  // __dirname will provide the location of project directory.
  res.sendFile(__dirname + '/index.html');
});

app.use('/home', router);

app.listen(3000);

console.log("Listening at Port 3000");
```

Restart the server and visit the same URL. To restart, terminate it from the terminal and run it again. In the upcoming tutorials, we will be using **nodemon**, which will automatically restart our program in case of any changes.

You will see the following output:



Routers and middleware

We have seen router and how to use it to control web applications. However, there maybe a scenario where we need to execute a piece of code for every router. Of course, you can either write this code everywhere or maintain it as a function call. However, for both the methods, extra code needs to be written for every route.

Here, middleware comes to the rescue. Middleware is also termed code injection in software architecture, where we write a piece of code in such a way that it is executed every time. For example, consider a situation where you want to check the route the user is calling and its type, such as GET, POST, and so on.

Therefore, instead of writing the code for every router, consider the following code to see what you can do:

```
app.use(function(req, res, next) {
  console.log("Route is "+ req.path + " and type is "+req.method);
  next();
});
```

Place this in the `Server.js` file and watch the console. Express will run this every time before hitting to any other router. Here is the `Server` code:

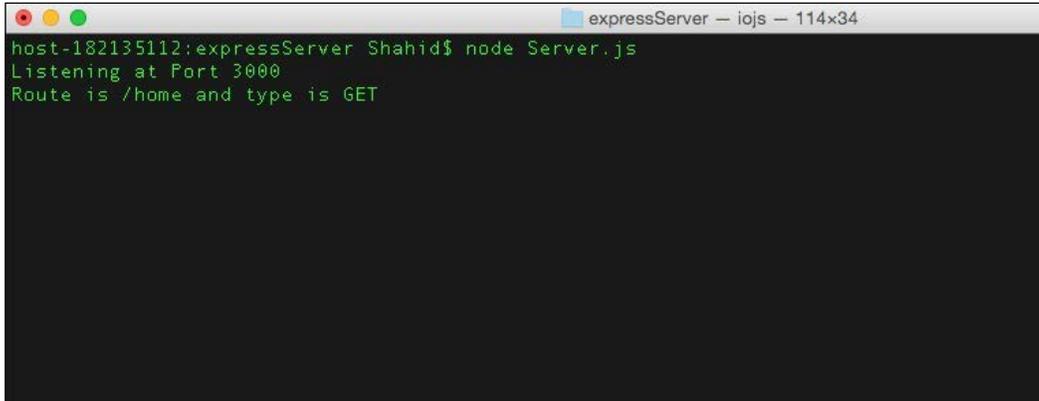
```
var express = require("express");
var app = express();
var router = express.Router();

router.get('/', function(req, res) {
  res.sendFile(__dirname + '/index.html');
});
app.use(function(req, res, next) {
  console.log("Route is "+ req.path + " and type is "+req.method);
  // next will pass the execution to next middleware or router.
  next();
});
app.use('/home', router);

app.listen(3000);

console.log("Listening at Port 3000");
```

Run the code again, enter the same URL in the browser, and check the console, as shown in the following screenshot:

A terminal window titled "expressServer - iojs - 114x34" with a dark background and green text. The text shows the command "node Server.js" being executed, followed by the output "Listening at Port 3000" and "Route is /home and type is GET".

```
host-182135112:expressServer Shahid$ node Server.js
Listening at Port 3000
Route is /home and type is GET
```

Summary

We covered the basics of a Node.js web server and its development using Express. You learned how to use middleware in Express to write less redundant code. In the next chapter, we are going to begin learning Sails.js, starting with the installation, understanding the directory structure, and configuring our development environment.

3

Introduction to Sails.js and MVC Concepts

Sails.js is a web framework designed to help developers produce scalable production-ready web applications at a fast pace. It follows the familiar **Model-View-Controller (MVC)** pattern that is adopted by **Active Server Pages (ASP)**, Ruby on Rails, and so on to develop applications. With built-in data flow support, API generation and many more, Sails.js is one of the first choice to develop a web application using Node.js, especially real-time applications such as chat system or multi-player game.

In this chapter we'll cover the following topics:

- Getting started with MVC concepts
- Installing Sails.js
- Understanding directory structure of Sails.js project
- Adding database support
- Configuring the Grunt task runner file with JSHint

Getting started with MVC concepts

We know that MVC is a software architecture pattern coined by Smalltalk engineers. MVC divides the application into three internal component and data gets passed via each components. Each component is responsible for their task and they pass their result to the next component. This separation provides a great opportunity of code reusability and loose coupling.

Model

The main component of MVC is model. Model represents knowledge, it could be single object or nested structure of objects. The model directly manages the data (stores the data as well), logic, and rules of the application.

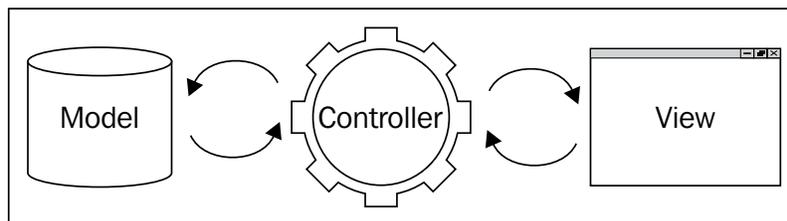
View

View is a visual representation of model. View takes the data from model and presents it (in a browser or console). View gets updated as soon as the model is changed. An ideal MVC application must have a system to notify other components about the changes. In web application, view is the HTML that we present in a web browser.

Controller

As the name implies, task of controller is to accept input and convert it to a proper command for model or view. Controller can send commands to model in order to make changes or update the state. It can also send commands to view to update the information.

For example, consider Google Docs to be an MVC application. View will be the screen, where you type. As you can see in the defined interval, it automatically saves the information in the Google server, which is controller that notifies model (Google backend server) to update the changes.



As you can see in the preceding diagram, let's once again consider Google Docs as an example, where Google database is the Model. As soon as you open Google Docs, the view gets loaded in the browser. As soon as you start typing, the controller wakes up and notifies the model that there is a change in view and the model updates its information with whatever you have typed.

When you load Google Docs again, the controller requests data from the model and passes it to the view to present it in the browser.

Installing Sails.js

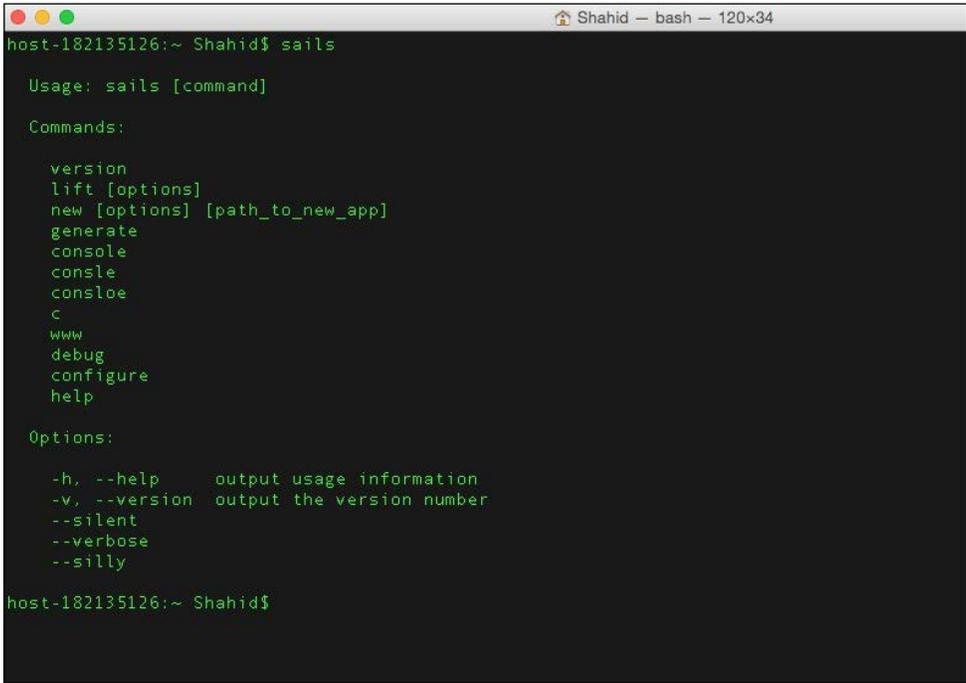
Make sure that you have **Node.js** and **npm** installed in your system. At the time of writing this book, the available version of stable Node.js is 4.2.1 and npm is 2.14.7.

Open the terminal and run the following command to install Sails.js:

```
npm install sails -g
```

You may need to provide **sudo** access if you are using Mac or Linux. It may take a while, depending on your Internet connection.

Once it is installed, you can check whether it went correctly by running the `sails` command in terminal, as shown in the following screenshot:

A terminal window titled "Shahid — bash — 120x34" showing the output of the 'sails' command. The output lists usage information, a list of commands, and a list of options.

```
host-182135126:~ Shahid$ sails
Usage: sails [command]

Commands:
  version
  lift [options]
  new [options] [path_to_new_app]
  generate
  console
  consle
  consloe
  c
  www
  debug
  configure
  help

Options:
  -h, --help      output usage information
  -v, --version   output the version number
  --silent
  --verbose
  --silly
```

Understanding directory structure of Sails.js project

Let's create a new sails project. Run the following command in the terminal:

```
sails new <projectName>
```

Sails will create a default project with all the folders such as `view`, `controllers`, and so on. Just switch to the project directory and list all the files. Here is how the directories look like using the `tree` command:

```
├─ api
  │ ── controllers
  │ ── models
  │ ── policies
  │ ── responses
  │ ── services
├─ assets
  │ ── images
  │ ── js
  │ │ ── dependencies
  │ ── styles
  │ ── templates
├─ config
  │ ── env
  │ ── locales
├─ node_modules
  │ ── ejs
  │ ── grunt
```

```
|   └─ grunt-contrib-clean
|   └─ rc
|   └─ sails
|   └─ sails-disk
└─ tasks
    └─ config
    └─ register
└─ views
```

The assets directory

This directory contains the static files such as image, HTML, JavaScript, and so on. In **Express.js**, we need to define the static path using the `express.static` code. In **Sails.js**, it will automatically do this for you. Just place the files that you want to serve as static in the `assets` directory and you are good to go!

The views directory

As the name suggests, it contains the files that we need to serve to the web browser. In **Sails.js**, by default, it is the **Embedded JavaScript (EJS)** templates file and you can also change this. **Sails.js** already has the ability to parse and render the EJS file; therefore, you do not need to add any view engine like we do in **Express**.

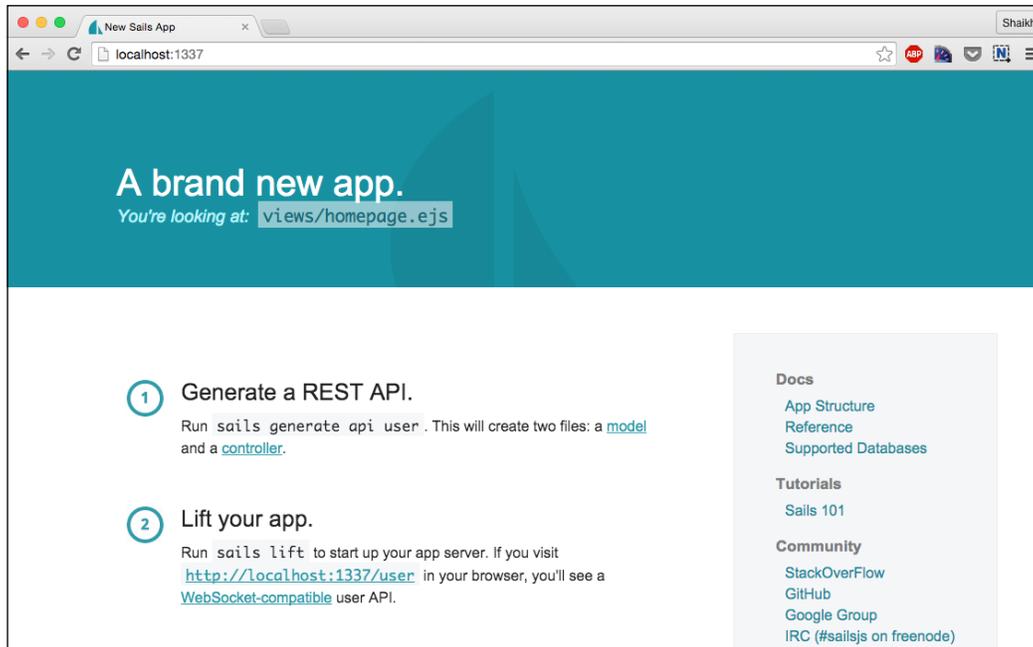
The node_modules directory

Sails.js uses various node modules to perform its task. It also includes the **Grunt** and **sails-disk** (to use disk as database storage) module by default. You can also include various modules if you need and they will also be stored in this directory.

The api directory

This is the most important directory of the **Sails.js** project. This directory contains the code of controller and model, and things associated with them such as policies, services, and so on.

The default application looks similar to the following screenshot in the browser:



Adding database support

Sails.js supports every major database and provides official adapters for them. By default, when you create new project, it uses disk as a database engine that we can change by editing some code in `config/connections.js` and `config/models.js`.

Configuring MySQL database with Sails.js

First of all, we need to install the official MySQL adapter for Sails.js called `sails-mysql`. You can install it via npm in your project by running the following command:

```
npm install --save sails-mysql
```

Once the installation is done. You need to provide the MySQL credentials in `config/connections.js` in order to let Sails.js make connection to database and then provide that connection in `config/models.js` to tell Sails.js which connection and database provider to use for the application.

config/connections.js

Consider the following code of the `config/connections.js` file:

```
module.exports.connections = {
  mysqlAdapter: {
    adapter: 'sails-mysql',
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'sampleDB'
  }
};
```

config/models.js

The following is the content of the `config/models.js` file:

```
module.exports.models = {
  connection: 'mysqlAdapter'
};
```

This is it. We have added the MySQL support in our Sails.js project. Now, you can run various queries and fetch results from the MySQL database in a standard way.

Let's look over the MongoDB connection.

Configuring MongoDB database with Sails.js

MongoDB is one of the fastest growing NoSQL database and has been widely used in many web applications. Sails.js provides support for MongoDB as well. You need to install the official **sails-mongo** module and use it in the same way we did for MySQL using the following command:

```
npm install --save sails-mongo
```

Once the installation is done. You need to provide the MySQL credentials in `config/connections.js` and then tell Sails.js to use them in `config/models.js`.

config/connections.js

The following is the content of the `config/connections.js` file:

```
module.exports.connections = {
  mongoAdapter: {
    adapter: 'sails-mongo',
    host: 'localhost',
    port: 27017
  }
};
```

This is the minimal setting you will need; however, you can provide username, password, and database name if you have them at hand.

config/models.js

The following is the content of the `config/models.js` file:

```
module.exports.models = {
  connection: 'mongoAdapter'
};
```

This is it for MongoDB as well. You can also add **Postgres** database support in similar manner. Sails.js is adaptable to any new database. You can use the connection name across the project to get/set new connections, fire query, and so on.

Configuring the Grunt task runner file with JSHint

One of the weakest points of JavaScript is its lack of compile-time debugging. Luckily, there are ways to achieve this. JSHint is a popular framework that does this task, that is, finds the compile-time errors in JavaScript code.

Sails.js uses Grunt as its default task runner and also comes up with default tasks such as building, concatenate JS and CSS files, and so on. You can run various tasks in Grunt. When you run the Sails.js project using `sails lift`, it runs the default task of Grunt by default.

Therefore, we need to add the JSHint task in a default task of Grunt so that whenever we lift our Sails.js app, it checks our code. First of all, we need to install the JSHint module via npm, as follows:

```
npm install grunt-contrib-jshint --save-dev
```

Once it is installed, we need to create a new `jshint.js` filename in the `tasks/config` folder. This folder contains all the tasks that Grunt has to run. The code for this is as follows:

```
module.exports = function(grunt) {
  grunt.config.set('jshint', {
    jshint : {
      myFiles : ['../../api/controllers/**/*.js']
    }
  });
  grunt.loadNpmTasks('grunt-contrib-jshint');
};
```

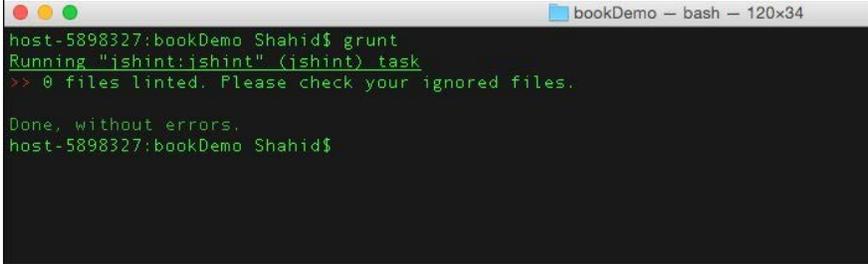
You can add more files in the code. Once the task file is created, we need to register the task (that is, `jshint`) in the `default.js` file present in the `config/register` folder. By default, the file will look similar to the following:

```
module.exports = function (grunt) {
  grunt.registerTask('default', ['compileAssets', 'linkAssets',
    'watch']);
};
```

After adding the `jshint` task, the file will look similar to the following:

```
module.exports = function (grunt) {
  grunt.registerTask('default', ['jshint', 'compileAssets',
    'linkAssets', 'watch']);
};
```

Now, if you run the `grunt` command in the terminal, JSHint will run first, as shown in the following screenshot:



```
host-5898327:bookDemo Shahid$ grunt
Running "jshint:jshint" (jshint) task
>> 0 files linted. Please check your ignored files.

Done, without errors.
host-5898327:bookDemo Shahid$
```

For the sake of clarity and a clean terminal, I have commented other tasks in `default.js`.

Summary

We covered the important points of MVC and installation and configuration of Sails.js in our system. You learned how to add database support in Sails.js and configure the Grunt task runner. In the next chapter, we will explore the development of **representational state transfer (REST)** API using Sails.js.

4

Developing REST API Using Sails.js

Representational state transfer (REST) is a software architectural style for the World Wide Web. REST APIs that use HTTP verbs as operational methods are called RESTful APIs. REST API is one of the important concept and it is very useful for web and mobile application due to its flexibility, adaptability, and uniformity.

In this chapter, we will cover the following topics:

- Why it is called REST?
- REST CRUD operation
- Database design for REST API
- Building REST API in Sails.js

Why it is called REST?

The name REST actually technically explains that the client initiates the transfer of representation of server state. What this really means is that unlike a web service, where we request the server to do something and the operation is not dependent on the type of HTTP verb, REST explains the operation by its HTTP verb and endpoint explains what actually happens.

The REST CRUD operation

CRUD is the **create, read, update, and delete** operation that is a common proof of a concept for any REST API. We will achieve the same using our Sails.js REST API. Before that, we need to choose where we should perform this CRUD operation. We will first use the MySQL database as the point to perform our CRUD operation.

In REST, we need to make sure that HTTP verbs (GET, POST, PUT, DELETE, and so on) give the clear explanation of their operation. For example, the GET /message should represent that we are trying to extract the messages. It shouldn't be GET /getAllmessage, which leads to ambiguity in understanding the REST operation.

Database design for REST API

Let's design a simple database to handle raw text messages. Therefore, message is the object on which we are going to perform the CRUD operation using the REST approach. To simplify this, let's use the following two fields:

- e-mail or username
- message

Create a database in MySQL via **phpMyAdmin** or command prompt and then create the preceding table with two fields. Here is the SQL query for reference:

```
CREATE SCHEMA IF NOT EXISTS `sailsApi` DEFAULT CHARACTER SET latin1 ;
CREATE TABLE IF NOT EXISTS `sailsApi`.`message` (
  `email` VARCHAR(255) NULL DEFAULT NULL COMMENT '',
  `message` VARCHAR(255) NULL DEFAULT NULL COMMENT '',
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '',
  `createdAt` DATETIME NULL DEFAULT NULL COMMENT '',
  `updatedAt` DATETIME NULL DEFAULT NULL COMMENT '',
  PRIMARY KEY (`id`) COMMENT '')
```

Building REST API in Sails.js

Let's create a new Sails.js project and develop our first Sails.js API. To create a new project, use the following command:

```
sails create <projectName>
```

To create a new API in Sails.js, you have the following two ways:

- Create model and controller manually
- Allow Sails.js to create them automatically

I will go for the second method as it's fairly simple and system-generated files are less prone to errors; therefore, you don't have to waste your time debugging these files. However, at the end of this chapter, we will take a look at the first method as well.

To create a new API, just execute the following command from the terminal in the project directory:

```
sails create api Message
```

This is it. We created a skeleton of our API by executing this command. Now, let's take a look at the changes made by this command to our project.

If you traverse to the `models` and `controllers` directory of the project, you will see new files created: `Message.js` in the `models` directory and `MessageController.js` in the `controllers` directory. For API, no view will be created.

Before taking a look over these files. Let's first configure our database that we are going to use for API. We will use MySQL as the database storage and, like we discussed in the last chapter, we need to edit `config/connections.js` and `config/models.js` to connect Sails.js to the database engine.

Before editing these files, create one database in MySQL with a name of your choice. If you have phpMyAdmin installed, you may need to visit `localhost/phpmyadmin` from the browser and create a new database. You can also do this via command line.

Assuming that you have created the database, let's connect to it using our Sails.js. First, let's define the connection parameter. We also assume that you have `sails-mysql` installed in your project, if not, do it using the following command in the project directory:

```
npm install --save sails-mysql
```

config/connections.js

Consider the following code of the `config/connections.js` file:

```
module.exports.connections = {
  mysqlAdapter: {
    adapter: 'sails-mysql',
    host: 'localhost',
    user: 'root',
    password: '',
    database: 'sailsApi'
  }
};
```

config/models.js

Add the connection variable to your `config/models.js` file, as follows:

```
module.exports.models = {
  connection: 'mysqlAdapter',
  migrate: 'safe'
};
```

We have defined the connection in the `models.js` file with the `migrate` parameter. Let's take a look at what this actually is.

Discussing migrate key

The `migrate` key controls and informs Sails.js whether to create/rebuild schema, collection, tables, and so on. In Node.js production environment, it is recommended and Sails.js uses `migrate` as `safe`, which basically means that the developer has to manually create tables, schema, collection, and so on.

However, for development and learning purpose, you can also use other options shown in the following:

- `migrate: 'safe'`: Developer should manually create database, table, and collection

- migrate: 'alter': Automatic table, schema, and collection creation; however, keep the existing data
- migrate: 'drop': Drop the schema each time and rebuild it when you lift the Sails.js app

For sake of practice and developing *code for production*, we will use migrate as safe only throughout this book.

Running our code

To run our code, type the following command in the terminal:

```
sails lift
```

Your app will be running on localhost:1337. Let's test it. We need to run the following test cases:

- Create new message
- Read the message
- Update the message
- Delete the message

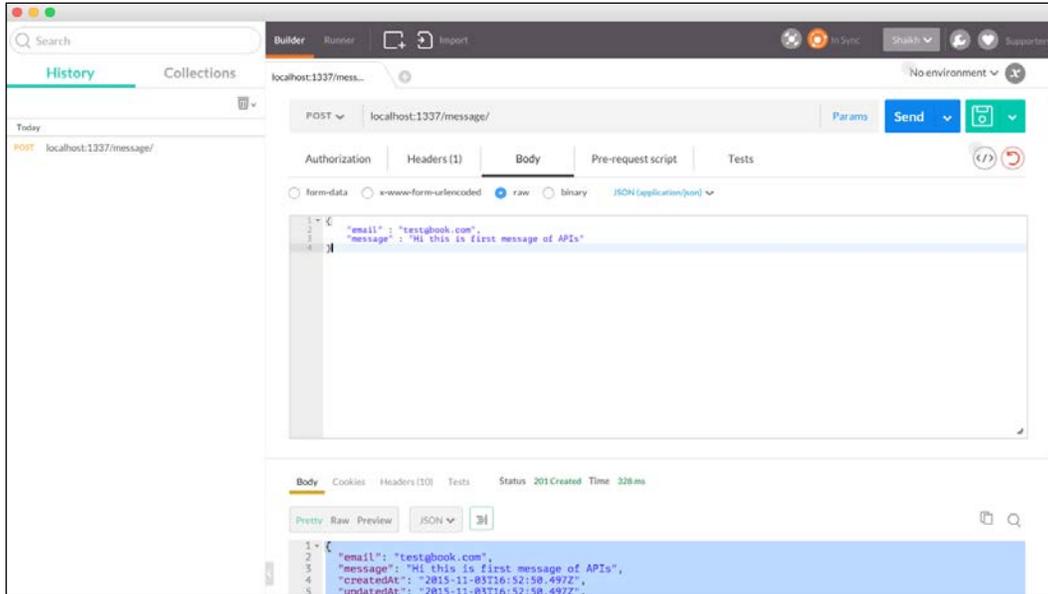
Create new message

You can use any REST simulator program to do the API testing. We recommend the **Postman** chrome extension. You can get it for free from the **Chrome Web Store**. It's really handy and useful.

To create a new message, select the method of request as **POST** and type `http://localhost:1337/message` in endpoint. Then, go to the **Body** section and select the **raw** radio button and provide the following JSON data:

```
{
  "email" : "test@book.com",
  "message" : "Hi this is first message of APIs"
}
```

Here is the screenshot of creating new message by API using Postman (API simulator) chrome app:

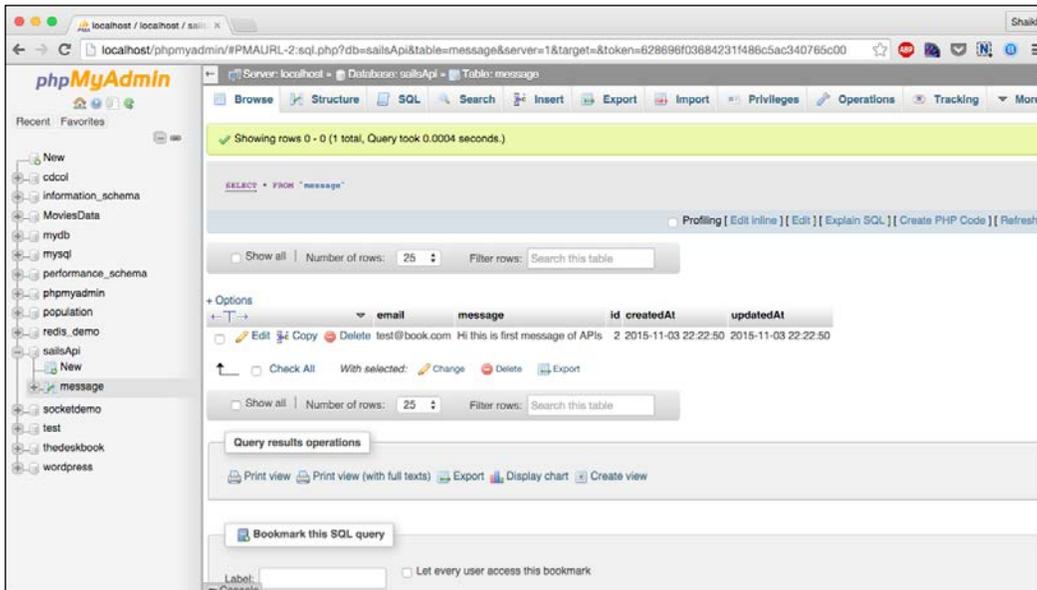


Now, hit the **Send** button and see the response. It should be something similar to the following:

```
{
  "email": "test@book.com",
  "message": "Hi this is first message of APIs",
  "createdAt": "2015-11-03T16:52:50.497Z",
  "updatedAt": "2015-11-03T16:52:50.497Z",
  "id": 2
}
```

You will also notice the 201 HTTP header, which is as per the HTTP standard for creation. Now, let's take a look at the database records.

Visit phpMyAdmin from the browser and choose the database that you have created. You will be able to see the preceding record present there, as shown in the following screenshot:



Read the message

We have two options to read our database records, as follows:

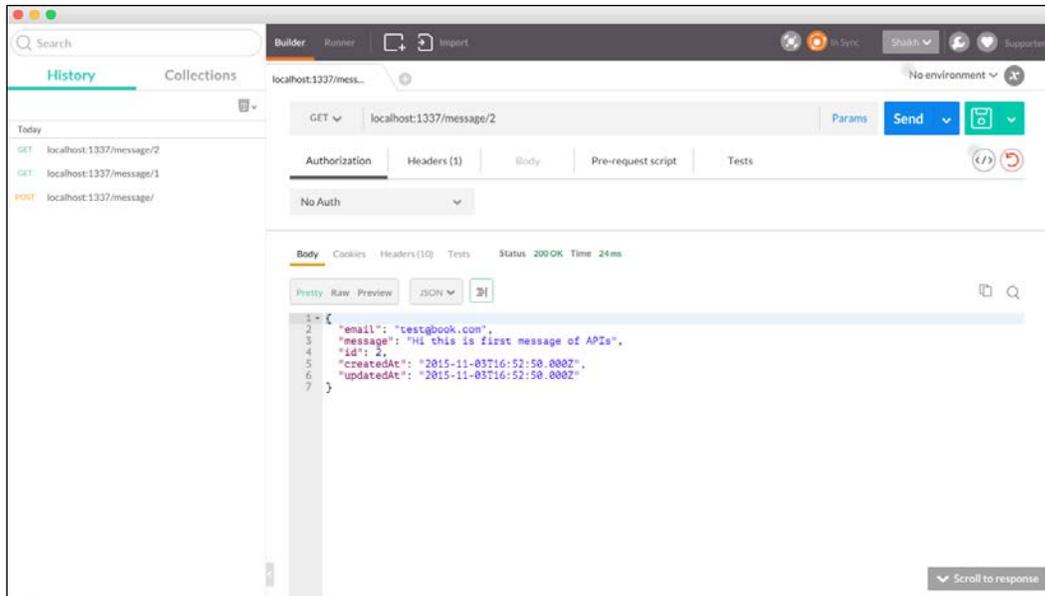
- Read all at once
- Read specific message

To read all the messages from the database, just change the HTTP method to **GET** and hit the **Send** button. You should be able to see all the messages from the database.

To read specific messages, you need to provide `id` of the record. The URL to enter will look similar to the following:

```
http://localhost:1337/message/:id {1, 2, and so on}
```

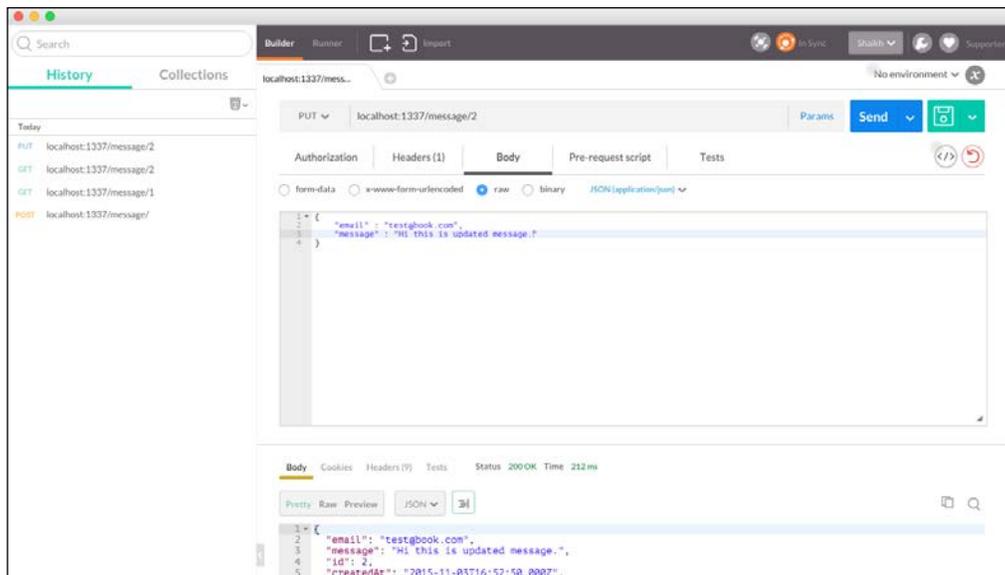
Let's fetch the record for user ID 1. Here is the request screenshot of Postman:



Update the message

We will use the `PUT` method to update the existing records in our database. Again, we need to pass the specific ID of the message that we wish to update. We cannot update all the records at once.

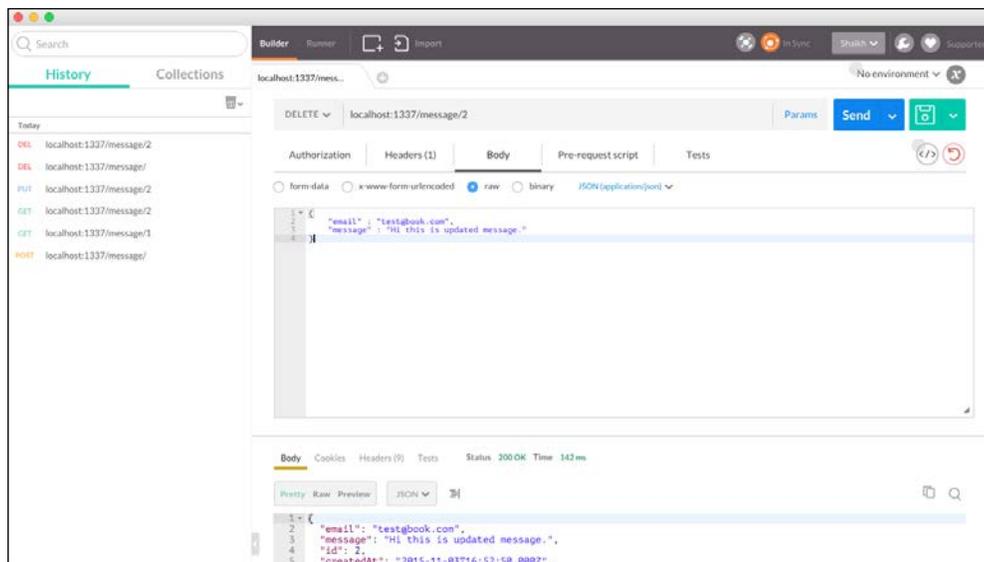
Therefore, we need to hit the URL with the ID of the record and pass the JSON data like we did in the *Create new message* section. The following is the request screenshot of Postman:



You can validate the data by fetching the same record from DB using the GET request.

Delete the message

To delete the message, you need to use the `/DELETE` HTTP method and pass the ID of data that you wish to delete. Again, here you cannot delete all the data at once. To perform the delete operation, refer to my request Postman screenshot:



Defining custom controller

You can also provide custom controller that can be called via the URL. To define custom controller, you need to edit the `api/controllers/MessageController.js` file and add the custom function. You can call the controller by providing the custom function name in the URL, right after the endpoint.

`api/controllers/MessageController.js`

The following is the content of the `api/controllers/MessageController.js` file:

```
module.exports =
{
  hi: function (req, res) {
    return res.send("Hi there!");
  },
  bye: function (req, res) {
    return res.redirect("http://www.google.com");
  }
};
```

Here `hi` and `bye` are specific controllers that can be called by hitting the `http://localhost:1337/message/hi` URL.

Summary

In this chapter, you learned about creating RESTful APIs in Sails.js very easily and in an effective way. We covered how to connect our Sails.js to external database and perform CRUD operation on them.

In the next chapter, we will learn how to develop real-world applications using Sails.js. We will develop a simple and similar Twitter feed, where every connected user can see tweets in real time.

5

Build a Chat System Using Sails.js

A chat system is one of the popular applications that we're already familiar with. Facebook, Google, or any other social media application have implemented one-to-one chat system, where you can chat with the person to whom you are already connected.

In this chapter, we will be building a chatting system that is purely *connection-oriented* rather than *user-oriented*. What we mean by this is that we don't require the user to sign up and provide their information and then find other user to chat with; all they need to do is come to the web app and start chatting on providing their name or stay anonymous with their message.

In this chapter, we will cover the following topics:

- Application architecture and flow
- Creating a Sails.js app
- Sails.js API for chat
- Model definition and MySQL integration in the app
- Sails.js controller to handle the chat operation
- AngularJS app for client-side interaction
- Running the application

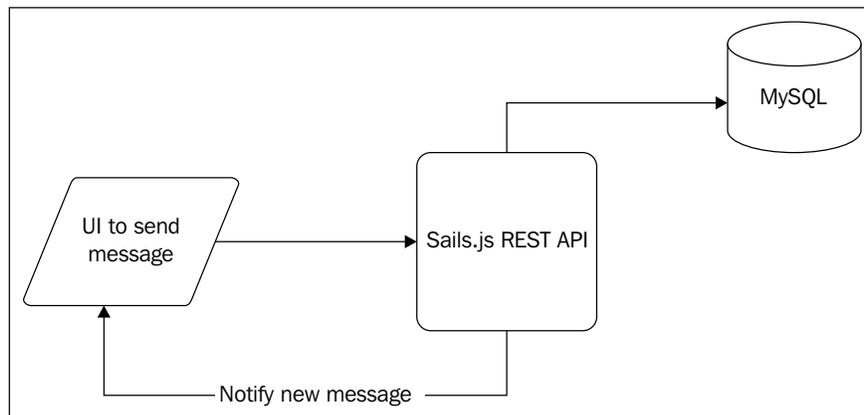
Application architecture and flow

Our chat application will be connection-oriented, requiring no sign-up or log-in process. All the user needs to do is open the web application and start chatting in the group, as we do in **Internet Relay Chat (IRC)**.

The flow of the application will be as follows:

1. The user opens the web app and subscribes to the socket connection.
2. We fetch the old messages (if any) from the database and show them to the user.
3. The user types in their name or leaves it as *Anonymous* and types a message.
4. Upon submitting the message, a new entry will be made in the database.
5. Every user connected to the socket will be notified in order to update their view.

The following diagram shows the sample architecture:



Creating a Sails.js app

In order to create a new Sails app, just open your terminal and run the following command:

```
sails create <app name>
```

This will create a fresh Sails.js application with default settings. In the next post, we will use **Embedded JavaScript (EJS)** templating for view engine; however, for this post, we will stick to the HTML pages in order to explain you how to use both of them.

To configure HTML as a default view engine in a Sails.js application, we need to make the following tweak. First, open `config/routes.js`, where you will see the following code:

```
module.exports.routes = {
  /*****
  *
  * Make the view located at `views/homepage.ejs` (or
  * `views/homepage.jade`, etc. depending on your default view
  * engine) your home page.
  *
  * (Alternatively, remove this and add an `index.html` file in
  * your `assets` directory)
  *
  *****/

  '/': {
    view: 'homepage'
  }

  /*****
  *
  * Custom routes here...
  *
  * If a request to a URL doesn't match any of the custom
  * routes above, it is matched against Sails route blueprints.
  * See `config/blueprints.js` for configuration options and
  * examples.
  *
  *****/
};
```

In order to make HTML the default routing, simply delete the following code or comment it out:

```
//Delete it
'/': {
  view: 'homepage'
}
```

This is it. Now, we can place our HTML files in the `/assets` folder and Sails.js will pick it from there.

Sails.js API for chat

We need API to communicate with the database (model) and view section. Let's generate one using the following command:

```
sails generate api <api name>
```

In our case, the API name is **Chat**. This will create the model and controller files in their respective folders. Before moving ahead, let's decide our model and database design.

Model definition and MySQL integration in the app

Create a database in MySQL with an appropriate name (for example, chatDemo) and then create a chat table using the following **Data Definition Language (DDL)** query:

```
CREATE TABLE IF NOT EXISTS `chatDemo`.`chat` (  
  `user` VARCHAR(255) NULL DEFAULT NULL COMMENT '',  
  `message` VARCHAR(255) NULL DEFAULT NULL COMMENT '',  
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '',  
  `createdAt` DATETIME NULL DEFAULT NULL COMMENT '',  
  `updatedAt` DATETIME NULL DEFAULT NULL COMMENT '',  
  PRIMARY KEY (`id`) COMMENT '')
```

Let's configure our model files. Before doing that, as we are going to use MySQL, let's add the MySQL adapter of Sails.js. Run the following command in the terminal:

```
sudo npm install --save sails-mysql
```

Open config/connections.js and edit the MySQL connection string with the database details. This should look similar to the following:

```
module.exports.connections = {  
  localDiskDb: {  
    adapter: 'sails-disk'  
  },  
}
```

```
mysqlAdapter: {
  adapter: 'sails-mysql',
  host: 'localhost',
  user: 'root',
  password: '',
  database: 'chatDemo'
}
};
```

Now, open `config/models.js` and add the MySQL adapter. Your code should look similar to the following:

```
module.exports.models = {

  connection : 'mysqlAdapter',
  migrate: 'safe'

};
```

The `migrate: 'safe'` command means that Sails.js will not create/update/delete any database-specific operation, that's the programmer's job. However, if you want Sails.js to do this, set `migrate` to either `alter` or `drop`.

We have configured the database. Now, let's define our API model. Open `/api/Models/Chat.js` and add the following code:

```
module.exports = {
  attributes: {
    user: {
      type: 'string',
      required: true
    },
    message: {
      type: 'string',
      required: true
    }
  }
};
```

Sails.js controller to handle the chat operation

We have connected MySQL, created our model, and are ready to write our controller file. Our controller file is responsible for the following actions:

- Accepting HTTP requests
- Responding to the requests
- Calling other services

The following is our controller file in `/api/controllers/ChatController.js`:

```
module.exports = {
  index: function (req, res) {
    var data = req.params.all();
    if(req.isSocket && req.method === 'POST') {
      Chat.query('INSERT into `chat` (`user`,`message`) VALUES
        ("'+data.user+'", "'+data.message+'")', function(err, rows) {
        if(err) {
          sails.log(err);
          sails.log("Error occurred in database operation");
        } else {
          Chat.publishCreate({id: rows.insertId, message :
            data.message , user:data.user});
        }
      });
    } else if(req.isSocket){
      Chat.watch(req.socket);
      sails.log( 'User subscribed to ' + req.socket.id );
    }
    if(req.method === 'GET') {
      Chat.query('SELECT * FROM `chat`', function(err, rows) {
        if(err) {
          sails.log(err);
          sails.log("Error occurred in database operation");
        } else {
          res.send(rows);
        }
      });
    }
  }
};
```

As you can see, we are using the controller endpoint, that is `/chat`, to add/retrieve new messages from the database.

When view makes a POST request, we will assume that it is a request to add a new message to the database. Hence, we will prepare a query and add it to our table. Once this is successfully done, we will notify every socket using the `publishCreate` method.

If the request method is `/GET` with the socket request, we will add this socket request to our model using the `watch` function. Sails.js will subscribe this socket internally to our model and we can use various methods to notify the user.

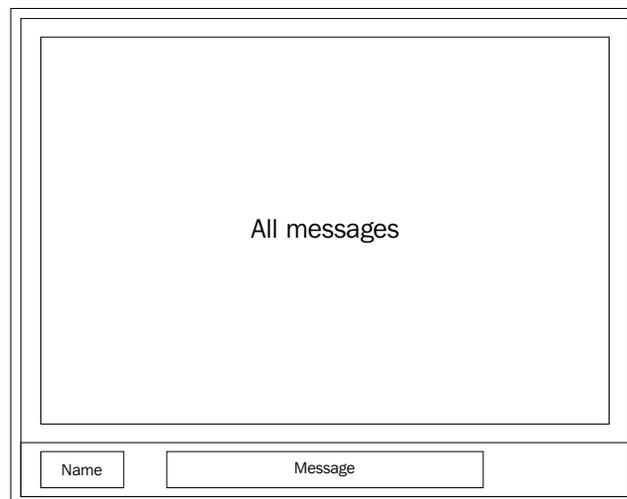
If the request method is `/GET` without the socket request, we will simply get all the messages from the database and send them to the UI. We will see what to do first—subscribe to the socket or get the messages—in the next section, where we are going to write our AngularJS application.

AngularJS app for client-side interaction

Our backend is almost done. We have the API exposed with the database integration. We need to design our UI now and call these APIs to make it functional.

We will use bootstrap for layout and AngularJS to handle the client-side interaction. Make sure that you download the bootstrap file from the official site and add it to the `/assets/styles` folder.

Here is a rough diagram of UI:



The following is our `Index.html` file in the `/assets` folder:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Sails Socket Demo - Maangalabs</title>
    <link rel="stylesheet" type="text/css"
      href="/styles/bootstrap.min.css">
    <link rel="stylesheet" type="text/css"
      href="/styles/style.css">
    <link href='https://fonts.googleapis.com/css?family=Open+Sans:
      300,600' rel='stylesheet' type='text/css'>
  </head>
  <body ng-app="socketApp" ng-controller="ChatController">
    <div class="navbar navbar-default navbar-fixed-top">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle=
          "collapse" data-target=".navbar-responsive-collapse">
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">Chat System V1.0</a>
      </div>
      <div class="navbar-collapse collapse navbar-responsive-
        collapse">
      </div>
    </div>
    <div class="col-md-12" style="padding:100px">
      <table class="table">
        <tr class="chat_message" ng-repeat="chat in chatList |
          orderBy:predicate:reverse | limitTo: 15">
          <td class="col-md-12 td_class"><strong>{{ chat.user }} :
            </strong> {{ chat.message }}</td>
        </tr>
      </table>
    </div>
    <div class="navbar navbar-inverse navbar-fixed-bottom" >
      <div class="col-lg-12">
        <form class="form_chat">
          <div class="col-lg-4 col-md-3">
            <input type="text" ng-model = "chatUser" class="form-
              control" placeholder="TypeYourNameHere">
          </div>
        </form>
      </div>
    </div>
  </body>
</html>
```

```

        <div class="col-lg-6 col-md-5">
            <input type="text" ng-model = "chatMessage" class=
                "form-control" placeholder="TypeYourMessageHere">
        </div>
        <button class="btn btn-default col-lg-2 col-md-2"
            ng-click="sendMsg()">Send</button>
        </form>
    </div>
</div>
<script type="text/javascript" src="/js/dependencies/
    sails.io.js"></script>
<script type="text/javascript"
    src="https://ajax.googleapis.com/ajax/libs/angularjs/
    1.2.27/angular.min.js"></script>
<script type="text/javascript" src="/js/app.js"></script>
</body>
</html>

```

Here is our `style.css` file to add some style to our UI. This file is located in `/assets/style/style.css`:

```

body{
    background: #ededed;
    font-family: 'Open Sans', sans-serif;
}

```

We are setting the background and font for the `body` section of the HTML page. For chat box, we have defined the form controls and are setting their style here, as shown in the following:

```

.navbar{
    border-radius: 0px;
}
.form_chat{
    padding:10px;
}
.form-control{
    width: inherit;
}
.chat_message{
    padding: 10px;
    color: #000;
    font-size: 15px;
background: #fff;
    font-family: 'Open Sans', sans-serif;
}

```

Bootstrap provides us the basic styling; however, we can overwrite them with our custom style. Here we are setting the custom styling for the HTML table and bootstrap layout:

```
.td_class{
  word-break:break-all;
  padding: 34px;
  padding-bottom: 0px;
  padding-top: 20px;
  border:0;
}
.navbar-brand{
  font-size: 14px;
  font-weight: 600;
  text-decoration: none;
}
.user_name{
  padding-bottom: 0;
  color: #fff;
  font-size: 15px;
}
.col-lg-4, .col-lg-6{
  padding-right: 3px;
  padding-left: 3px;
}
```

The following is our AngularJS file residing in `/assets/js/app.js`:

```
// Defining angular application.
var socketApp = angular.module('socketApp', []);
// Defining Angular controller.
socketApp.controller('ChatController', ['$http', '$log', '$scope',
  function($http, $log, $scope){
    $scope.predicate = '-id';
    $scope.reverse = false;
    $scope.baseUrl = 'http://localhost:1337'; // API endpoint
    $scope.chatList = [];
    // This function will call the socket endpoint and fetch all the
    messages.
    // Remember Sails provide Socket messages as an endpoint too.
    $scope.getAllMessages = function(){
      io.socket.get('/chat/');
      $http.get($scope.baseUrl+'/chat/')
        .success(function(success_data){
```

```

        $scope.chatList = success_data;
        $log.info(success_data);
    });
};
// Call above function on load of the page to load previous
chats.
$scope.getAllMessages();
$scope.chatUser = "Anonymous"; // Setting default name
$scope.chatMessage="";

// This is the event we generate from our backend system.
// When user send a message, we broadcast that message in order
to display it to user.
io.socket.on('chat',function(obj){
    if(obj.verb === 'created'){
        $log.info(obj)
        $scope.chatList.push(obj.data);
        $scope.$digest();
    }
});
// Function that gets called upon click of button.
$scope.sendMessage = function(){
    $log.info($scope.chatMessage);
    // Calling the socket API with POST request to add new message
in database.
    io.socket.post('/chat/', {user:$scope.chatUser,message:
    $scope.chatMessage});
    $scope.chatMessage = "";
};
}]);

```

If you notice, we are subscribing the user to the socket first and then fetching all the messages from database.

The `getAllMessages()` function makes the HTTP /GET call to the Chat API that provides us all the messages (limited to 300 rows) and we apply it to the UI using the `$digest` function.

The `sendMessage()` function makes the POST call to the socket that, in turn, becomes the HTTP call. If you recall, it will add the new messages to the database and emit the socket message.

In the socket notification chat, we will update the view using the same `$digest` function. By default, the name of the socket messages will be the name of API, that is, Chat.

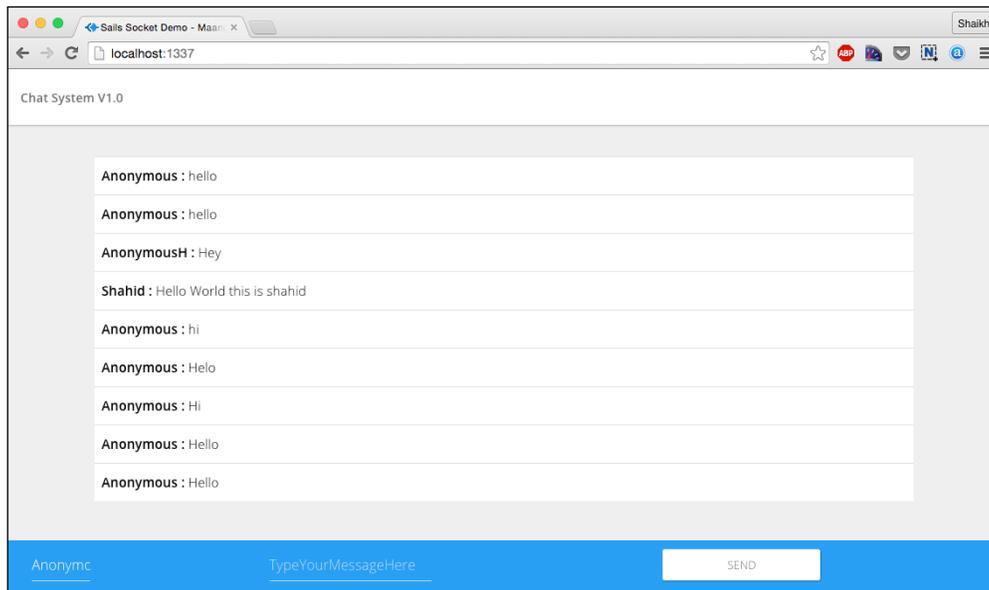
Running the application

To run the application, open the terminal and run the following command:

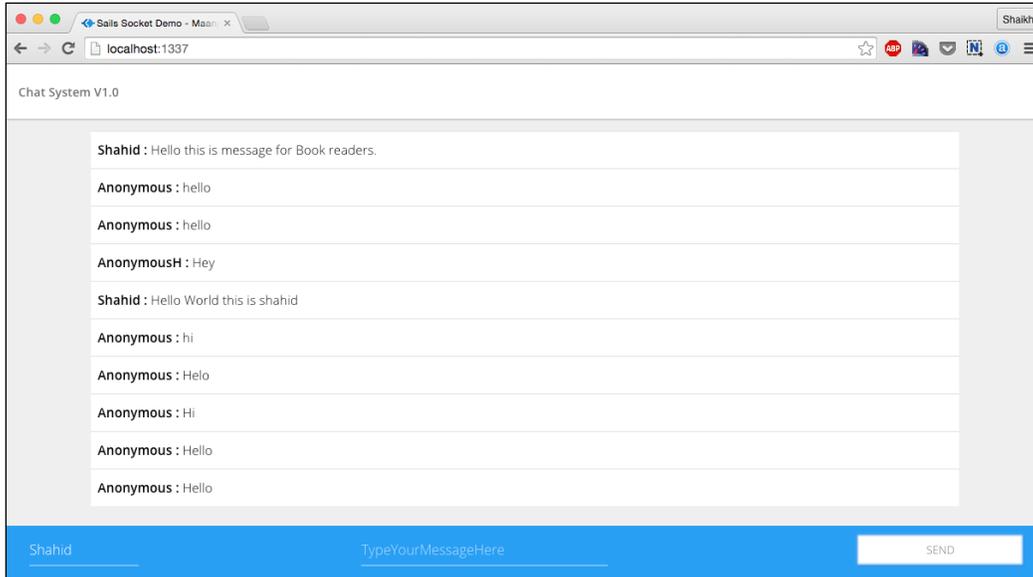
```
sails lift
```

```
UnixRoot-Machine:chatExample Shahid$ sails lift
info: Starting app...
info:
info:   Sails
info: v0.10.5
info:
info:   <|
info:  / \
info: /   \
info:/     \
info:=====
info:  \     /
info:  \   /
info:  \ /
info:  ---
info:
info: Server lifted in `~/Users/Shahid/Desktop/The Book/Sailsjs essentials/chapter 05/chatExample`
info: To see your app, visit http://localhost:1337
info: To shut down Sails, press <CTRL> + C at any time.
debug: -----
debug: :: Thu Nov 19 2015 15:13:42 GMT+0530 (IST)
debug: Environment : development
debug: Port       : 1337
debug: -----
```

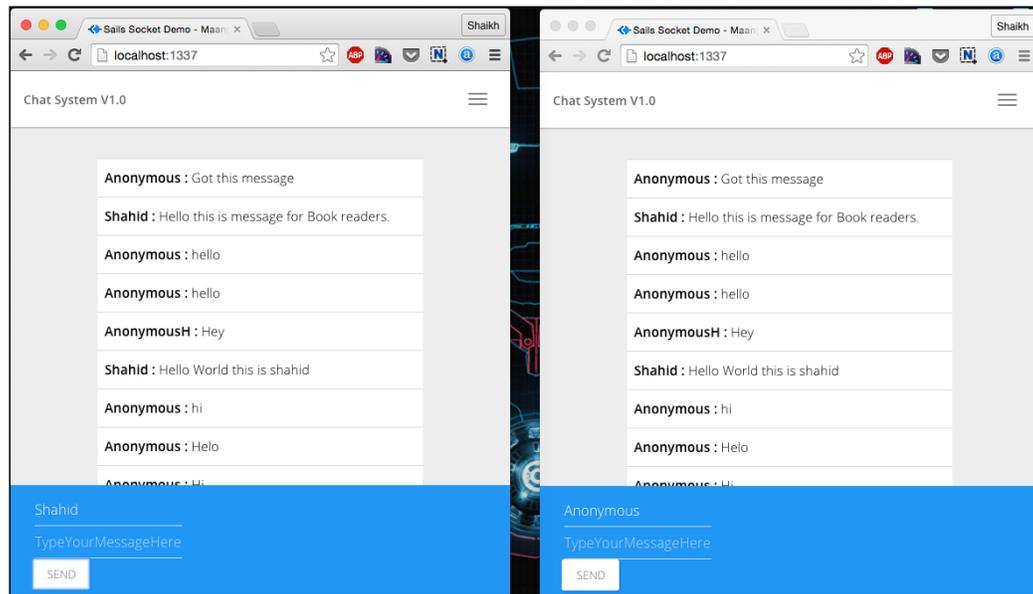
Now, we can visit our application at <http://localhost:1337/>, as shown in the following image:



To add a new message, you need to add the name or leave it as it is and type your message as shown in the following screenshot:



Now, let's test the socket broadcasting. Open two windows and send messages one after another, as shown in the following image:



Summary

We covered database integration and AngularJS integration in our Sails.js app. We also developed a full-fledged connection-oriented chat application in Sails.js. In the next chapter, we will develop one more interesting application that updates status across various users in real-time like Twitter.

6

Building a Real-Time News Feed App Using Sails.js

One of the promising and rare feature of Sails.js is its **WebSocket** integration. When you create a new Sails.js application, you have completely tested and featured socket integration. In case you are not aware of the web socket or Socket.IO platform, it's the revolutionary implementation of socket (old school BSD sockets) into the web, which allows us to send and receive messages without Ajax.

In this chapter, you will learn how to develop a real-time status updated application such as Twitter and Facebook, where any update posted by a person will get updated on their friends' walls. We won't be developing a full-featured application such as Twitter; however, we will develop its core (WebSocket and Database integration).

You will learn the following topics in this chapter:

- Briefing Socket.IO
- Using Socket in Sails.js
- Discussing the database design of the app
- Implementing the app

Briefing Socket.IO

Socket.IO is a platform that is currently maintained by Automattic, Inc. (owner of WordPress.com and Akismet) and is open source to all. While writing this book, Socket.IO is available only for the Node.js application but it will soon be available for major server-side languages.

You can install Socket.IO in your Node.js project and access it at every phase of the application such as routes, view, and so on. You can install Socket.IO using the following command:

```
npm install socket.io
```

Use `--save` to rewrite `package.json`. Then, you can include it in your server file and pass the HTTP instance to it. Luckily, we don't need to do anything as Sails.js does this all for us.

Using Socket in Sails.js

Sails.js integrates Socket by default and provides various wrapper functions to access or broadcast the Socket messages. If you like to have a look at the Socket.IO configuration in Sails.js, you can view the `sockets.js` file in the `config` folder. Here is a snippet of the code:

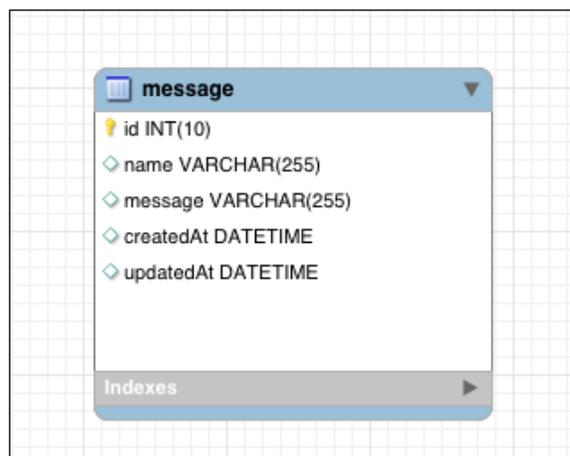
```
module.exports.sockets = {  
  
  onConnect: function(session, socket) {  
  },  
  onDisconnect: function(session, socket) {  
  },  
  transports: [  
    'websocket',  
    'htmlfile',  
    'xhr-polling',  
    'jsonp-polling'  
  ],  
  adapter: 'memory',  
  authorization: false,  
  'backwardsCompatibilityFor0.9SocketClients': false,  
  grant3rdPartyCookie: true,  
  origins: '*:*',  
  heartbeats: true,  
  'close timeout': 60,  
  'heartbeat timeout': 60,  
  'heartbeat interval': 25,  
  'polling duration': 20,  
  'flash policy port': 10843,  
}
```

```
'destroy buffer size': '10E7',
'destroy upgrade': true,
'browser client': true,
'browser client cache': true,
'browser client minification': false,
'browser client etag': false,
'browser client expires': 315360000,
'browser client gzip': false,
'browser client handler': false,
'match origin protocol': false,
store: undefined,
logger: undefined,
'log level': undefined,
'log colors': undefined,
'static': undefined,
resource: '/socket.io'
};
```

You don't need to understand all the settings right now; however, it is quite an important file. You can see that there are various switches provided to enable or disable a particular feature of Socket in Sails.js.

Discussing the database design of the app

Database design is quite simple. We need to store the status and the name of the user that posts it as well. Here is simple structure of the database:



The following is the SQL script for the database:

```
CREATE TABLE IF NOT EXISTS `sailsApi`.`message` (  
  `id` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT COMMENT '',  
  `name` VARCHAR(255) NULL DEFAULT NULL COMMENT '',  
  `message` VARCHAR(255) NULL DEFAULT NULL COMMENT '',  
  `createdAt` DATETIME NULL DEFAULT NULL COMMENT '',  
  `updatedAt` DATETIME NULL DEFAULT NULL COMMENT '',  
  PRIMARY KEY (`id`) COMMENT '')  
ENGINE = InnoDB  
AUTO_INCREMENT = 9  
DEFAULT CHARACTER SET = latin1;
```

Let's move to next step.

Implementing the application

Let's create our new Sails.js app, as shown in the following:

```
sails create <app name>
```

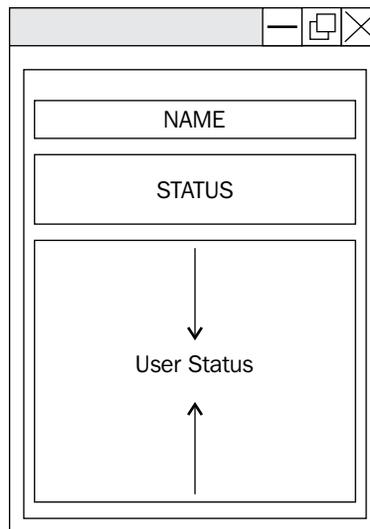
Once the app is created, we need an API to deal with our status updates. Let's create that as well:

```
sails generate api Message
```

Sails.js will create controller and model in the respective folders. Let's define the model first. Here is `models.js` in the `/api/models/` folder:

```
module.exports = {  
  attributes: {  
    name : { type: 'string' },  
    message : { type: 'string' }  
  }  
};
```

Before going to the controller and Socket modules, let's look at the view first. The user interface will be quite simple. We will have a textbox to accept the name and a text area to accept the status messages. It will look similar to the following diagram:



The following is the code to generate a similar view. We need to make changes in the `views/homepage.ejs` file, as follows:

```

<script type="text/javascript">
  setTimeout(function sunrise () {
    document.getElementsByClassName('header')[0].style.
      backgroundColor = '#118798';
  }, 0);
</script>

<div class="default-page" ng-app="sails-chat-example">
  <div class="header">
    <h3 id="main-title" class="container"><%= __('Real time status
      update example using sails.js.') %></h3>
  </div>
  <div class="main container clearfix">
    <ul class="getting-started">
      <div class="container" ng-controller="MainCtrl">
        <form class="form-horizontal" ng-submit="send()">
          <div class="form-group">
            <div class="col-sm-9">
              <input type="text" class="form-control"
                placeholder="Username" ng-model="data.name">
            </div>
          </div>
        </form>
      </div>
    </ul>
  </div>
</div>

```

```
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-9">
      <textarea rows=5 cols=5 class="form-control"
        placeholder="Message" ng-model="data.message">
      </textarea>
    </div>
  </div>
  <div class="form-group">
    <div class="col-sm-offset-3 col-sm-9">
      <button type="submit" class="btn btn-default">
        Post</button>
    </div>
  </div>
</form>
<div class="panel panel-default" ng-repeat="n in messages">
  <div class="panel-heading">
    <h3 class="panel-title">{{n.name}}</h3>
  </div>
  <div class="panel-body">
    {{n.message}}
  </div>
</div>
</div>
</ul>
</div>
</div>
```

To handle the UX features, we are going to use AngularJS. We will provide the working screenshot in the next chapter.

We need to add the Bootstrap and Angular dependencies. To add them, add the dependency in `views/layout.ejs`, as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <title>New Sails App</title>
    <meta name="viewport" content="width=device-width,
      initial-scale=1, maximum-scale=1">
```

```

<!--STYLES-->
<link rel="stylesheet"
  href="/styles/bootstrap/css/bootstrap-theme.min.css">
<link rel="stylesheet"
  href="/styles/bootstrap/css/bootstrap.min.css">
<link rel="stylesheet" href="/styles/importer.css">
<!--STYLES END-->
</head>

<body>
<%- body %>
<!--SCRIPTS-->
<script src="/js/dependencies/angular/angular.min.js"></script>
<script src="/js/dependencies/jquery/jquery.min.js"></script>
<script src="/js/dependencies/sails.io.js"></script>
<script src="/js/app.js"></script>
<script src="/js/bootstrap/js/bootstrap.min.js"></script>
<!--SCRIPTS END-->
</body>
</html>

```

We are done with the view section. Let's make changes in the controller file. We need to edit `/api/controllers/MessageController.js`, as follows:

```

module.exports = {
  status: function(req, res){
    Message.query("INSERT into
      message(`name`, `message`) VALUES ('"+req.param('name')+"',
      '"+req.param('message')+"')", function(err, rows) {
      if(!err) {
        Message.publishCreate({id:rows.insertId,
          name:req.param('name'), message:req.param('message')});
      } else {
        sails.log(rows);
      }
    });
  },
  subscribe: function(req, res){
    Message.watch(req);
  },
  index : function(req,res) {

```

```
Message.query("SELECT * FROM `message`", function(err, rows) {
  if(err) {
    res.json({"error" : true, "message" : "database error"});
  } else {
    res.ok(rows);
  }
});
}
```

Let's take a look at each of them. First of all, we need to have a controller that fetches all the messages from our database for the new user. This is what controller `index` will do.

Next, there is controller `subscribe` that, as the name implies, subscribes the new users to the existing socket.

An important controller is `chat`, which does the task of adding new messages to the database and notifying every user that is connected about the new message.

The `publishCreate()` method is invoked to notify and send broadcast messages to every user that is connected to the socket. As you may have noticed, we will also pass the data, that is, name and message with it in order to show it to our view.

Let's look at our Angular code now. The following code will reside in `assets/js/app.js`:

```
'use strict';

angular.module('sails-chat-example', [])

.controller('MainCtrl', ['$scope', '$http',
  function ($scope, $http) {
    $scope.messages = [];
    $scope.data = {
      name : null,
      message : null
    };
  }

  $scope.send = function() {
    io.socket.post('/message/status', $scope.data,
      function(res) {});
  }
};
```

```
io.socket.get('/message/subscribe', function(res){});

io.socket.on('message', function onServerSentEvent (msg) {
  switch(msg.verb) {

    case 'created':
      $scope.messages.push(msg.data);
      $scope.$apply();
      break;

    default: return;
  }
});

io.socket.on('connect',function(){
  $http.get('http://localhost:1337/Message')
  .success(function(data){
    for(var i = 0 ; i < data.length; i++) {
      $scope.messages.push(data[i]);
      $scope.$apply();
    }
  });
});

});
```

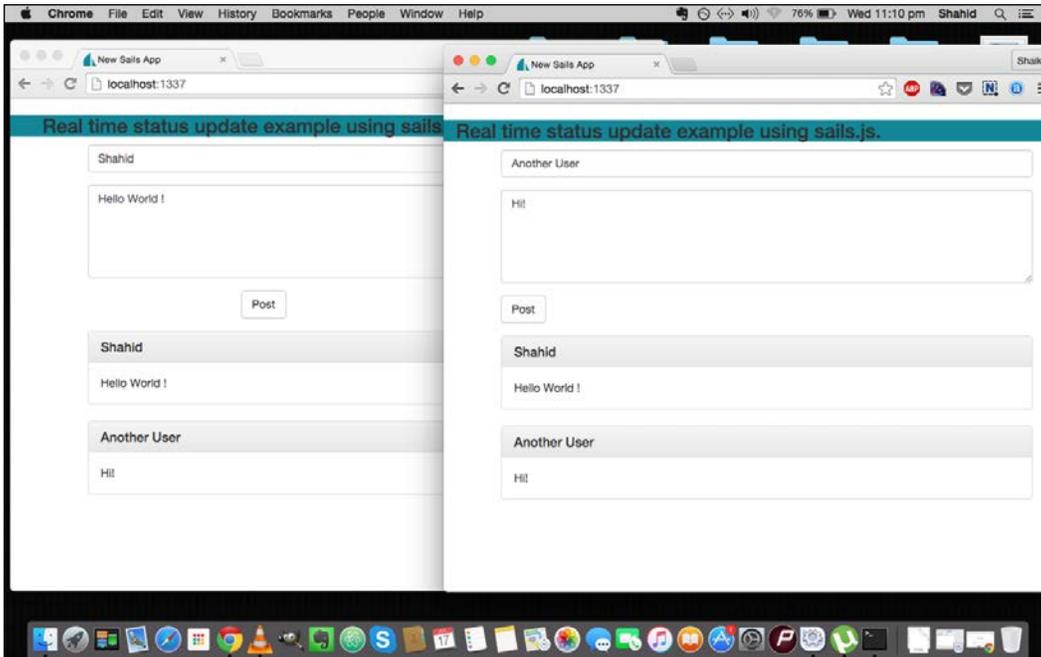
Before jumping to the explanation, I'd like to mention one more important feature of Sails.js and its Socket. When you subscribe the controller to the socket, you can perform the same CRUD operation that you did for the API for the Socket as well, that is, GET, POST, PUT and so on.

We have first created the Angular module and attached the controller to it. The controller contains our actual code. Therefore, when the user first loads the application, we need to subscribe the user to the Socket. We will do this using the following code:

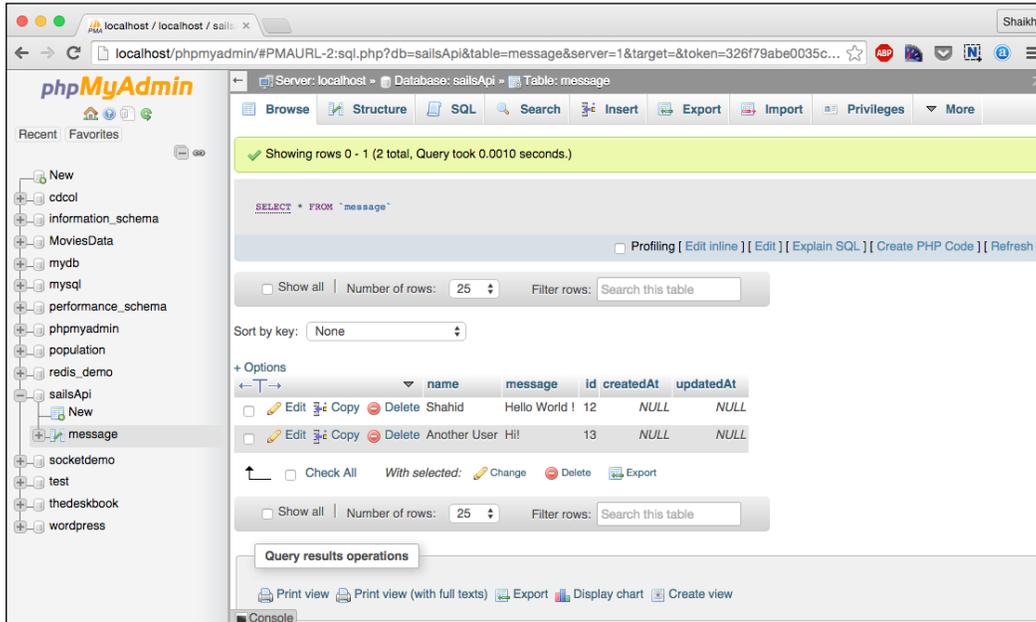
```
io.socket.get('/message/subscribe', function(res){});
```

Next, when the user is connected, we need to show the old status from our database to the user and we will be calling the API that we created — /Message.

Open the application at `http://localhost:1337` in your browser. In order to see the effect of Socket and real-time update, I would suggest you to open the application in two simultaneous browser windows, as shown in the following screenshot:



You may have observed that as soon as we add a status to a window, it is added to the database and other clients receive the socket notification. The following is the database view for both the messages:



Summary

In this chapter, we covered the Sockets and their uses in Sails.js. We also covered how to build real-time app using integral Socket of Sails.js. In the next chapter, we will be developing another commonly used application, **TODO app** purely using Sails.js.

7

Creating a TODO Single-Page Application

TODOs are one of the most common applications around the web and are also available for mobile users. In this chapter, we will try to build a simple TODO application, where a user can add, delete, and view their TODOs.

In the previous chapters, we have used MySQL as database and you have learned how to configure, connect, and perform queries over it. In this chapter, we will be using MongoDB for learning purposes. Make sure that you have installed MongoDB on your machine before moving ahead with this chapter.

In this chapter, we will learn the following topics:

- MongoDB support in Sails.js
- Defining model for API
- TODO app view design

Let's begin with creating a fresh Sails.js project. Run the following command to create one:

```
sails create todoApp
```

Sails.js will create a default folder structure for you. In this project as well, we will use HTML as a view engine rather than EJS, which we used in the previous chapter. To do so, edit `config/routes.js` and delete the default route, as follows:

```
'/': {  
  view: 'homepage'  
}
```

Let's create our API using the Sails.js command. Run the following command:

```
sails generate api todo
```

Sails.js will create controller and model for you.

MongoDB support in Sails.js

To enable MongoDB support in Sails.js, you need to install the connector and configure it in the model file. The `sails-mongo` package provides what we need. You need to install it using the following command in our project:

```
sudo npm install --save sails-mongo
```

You may not need `sudo` in a Windows-based system.

Once it is installed, we need to configure the connection and provide this connection to the default model file. Open the `config/connections.js` file and add the connection details of your MongoDB server.

Note that if you don't provide the MongoDB database a name, it will automatically create a new one if `migrate` is set to `alter` or `drop`, as follows:

```
module.exports.connections = {
  /*****
  *
  * MongoDB is the leading NoSQL database.
  * http://en.wikipedia.org/wiki/MongoDB
  *
  * Run: npm install sails-mongo
  *
  *****/
  mongoConnection: {
    adapter: 'sails-mongo',
    host: 'localhost',
    port: 27017,
    // user: 'username',
    // password: 'password',
    // database: 'your_mongo_db_name_here'
  }
};
```

Now open `config/models.js` and provide the connection information, as shown in the following:

```
module.exports.models = {
  connection: 'mongoConnection',
  migrate: 'alter'
};
```

 It's highly recommended to use `migrate as safe` in production environment.

This is it. Now, Sails.js will connect and perform the operation on your MongoDB collection. Let's move ahead and configure our API.

Defining model for API

We have generated the API and Sails.js created the default files for us. I explained in detail in *Chapter 4, Developing REST API Using Sails.js*, about the default functionality of REST APIs in Sails.js. Unless we need some extra functionality, we can use the default one given by Sails.js.

We need the following operation for our TODO app, let's validate whether the default functionality supports this:

Operation	End point
Create a new TODO task	/POST todo { JSON data }
Fetch all TODO tasks	/GET todo
Delete a TODO task	/Delete todo { JSON data }

Sails.js provides this over REST API, therefore, we don't need to write the code for the controller. Let's define a simple model. Open `/api/models/ToDo.js` and add the following code:

```
module.exports = {
  attributes: {
    value: {
      'type': 'text'
    }
  }
};
```

Creating a TODO Single-Page Application

This is pretty much it for the backend. Before jumping to the frontend, let's validate whether the APIs are working or not.

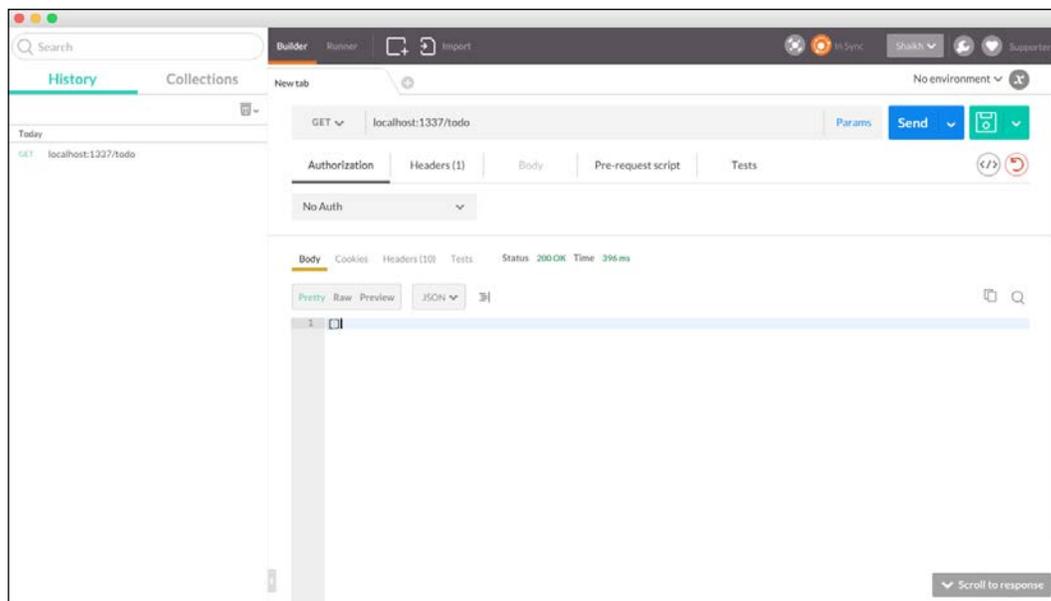
Lift the app using the following command:

```
sails lift
```

Open the HTTP simulator (we recommend the **Postman** Chrome extension) and enter the following URL:

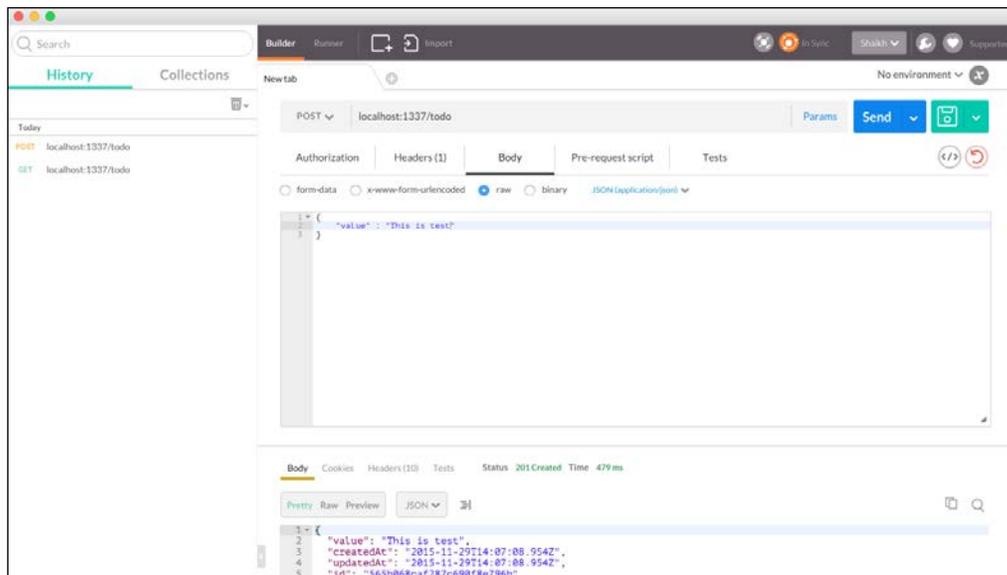
1. To get all TODOs, enter the following URL:
`/GET localhost:1337/todo`

You will see output as follows:



2. To add new TODO, enter the following URL:
`/POST localhost:1337/todo { data }`

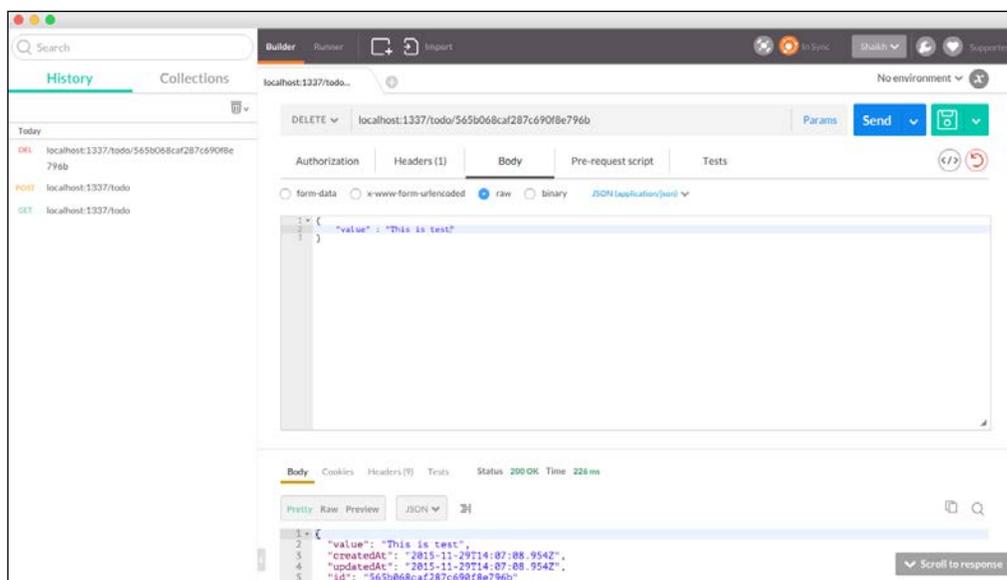
You will see output as follows:



3. To delete a TODO, enter the following URL:

`/DELETE localhost:1337/todo/:id`

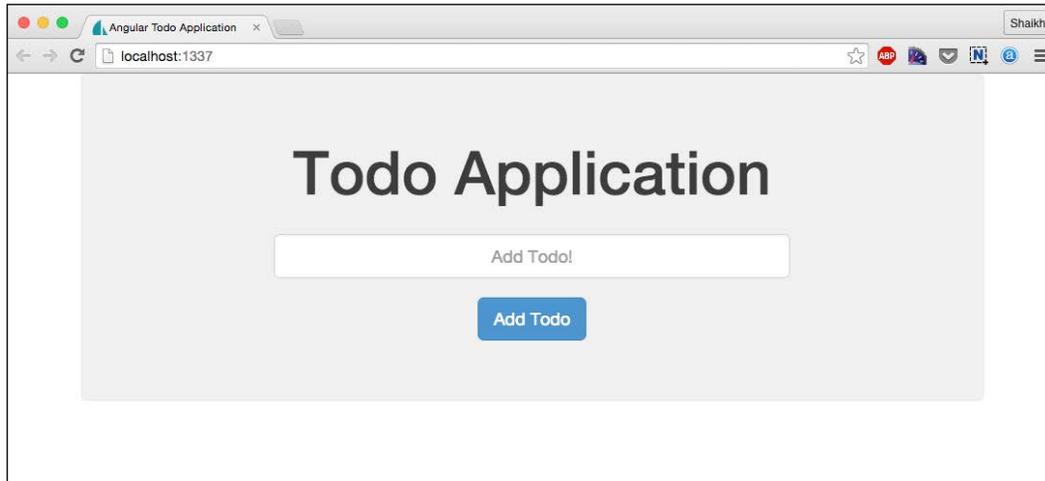
You will see output as follows:



Our API is working correctly. Let's design and code our frontend application.

TODO app view design

We will design a simple layout that supports responsiveness. Here is the end output that we will achieve after the code:



We are going to use AngularJS and Bootstrap as the JavaScript and CSS frameworks for designing this interface.

In the `/assets` folder, create a new `index.html` file that Sails.js will serve by default. Add the latest bootstrap file in the `/styles` folder and the `angular.js` file in the `/js` folder.

The following is our `index.html` file. First, we need to add Bootstrap and AngularJS files as a dependency:

```
<!DOCTYPE html>
<html ng-app="todoApp">
  <head>
    <title>Angular Todo Application</title>
    <meta name="viewport" content="width=device-width,
      initial-scale=1, maximum-scale=1">
    <!--STYLES-->
    <link rel="stylesheet" href="/styles/bootstrap.css">
    <link rel="stylesheet" href="/styles/importer.css">
    <!--STYLES END-->
    <!--SCRIPTS-->
    <script src="/js/angular.js"></script>
    <script src="/js/app.js"></script>
```

```
<script src="/services/ToDoService.js"></script>
<script src="/js/dependencies/sails.io.js"></script>
<!--SCRIPTS END-->
</head>
<body>
</body>
</html>
```

Now, we need to define the form controls. We need a textbox to add a new TODO task and a button to submit the task. Also, a list of tasks that the users have added before. The following code will be added to the `body` section of the preceding HTML code:

```
<div class="container" ng-controller="ToDoCtrl">
  <div class="jumbotron">
    <h1 align="center">ToDo Application</h1>
    <br>
    <div id="todo-form" class="row">
      <div class="col-sm-8 col-sm-offset-2 text-center">
        <form>
          <div class="form-group">
            <input type="text" class="form-control input-lg
              text-center" placeholder="Add Todo!"
              ng-model="formData.value">
            <br>
            <button type="submit" class="btn btn-primary btn-lg"
              ng-click="addTodo()">Add Todo</button>
          </div>
        </form>
      </div>
    </div>
    <div id="todo-list" class="row">
      <div class="col-sm-4 col-sm-offset-4">
        <div class="checkbox" ng-repeat="todo in todos">
          <label>
            <input type="checkbox" ng-click="removeTodo(todo)">
            {{ todo.value }}
          </label>
        </div>
      </div>
    </div>
  </div>
</div>
```

We created a separate `div` to allow the user to add a new TODO and to show what they have already added and can remove if they want to. The `todo-list` `div` will do a loop using `ng-repeat` over every `todo` object and show it as a checkbox. Upon checking this box, we will call a `removeTodo()` function to remove that TODO.

To add a new TODO, we have a `addTodo()` function. Also, upon loading, we will pull every TODO from MongoDB and show it here.

Let's take a look at our AngularJS application file.

/assets/js/app.js

The following is the `/assets/js/app.js` file:

```
'use strict';
var todoApp = angular.module('todoApp', []);
todoApp.controller('TodoCtrl', ['$scope', '$rootScope', 'TodoService',
function($scope, $rootScope, TodoService) {
  $scope.formData = {};
  $scope.todos = [];
  TodoService.getTodos().then(function(response) {
    console.log(response);
    $scope.todos = response;
  })
  $scope.addTodo = function() {
    console.log($scope.formData);
    TodoService.addTodo($scope.formData).then(function(response) {
      console.log(response);
      $scope.todos.push(response)
      $scope.formData = {};
    })
  }
  $scope.removeTodo = function(todo) {
    console.log(todo);
    TodoService.removeTodo(todo).then(function(response) {
      $scope.todos.splice($scope.todos.indexOf(todo), 1)
      console.log(response);
    })
  }
}
]
})
```

In this code, we have the following three functions:

- `getTodos()`: This will call AngularJS service, which in turn will make an HTTP call to the Sails.js web server to pull TODOs from the database
- `addTodo()`: This will call AngularJS service, which in turn will make an HTTP call to Sails.js web server to add a new TODO to the database
- `removeTodo()`: This will call AngularJS service, which in turn will make an HTTP call to Sails.js web server to delete a TODO from the database; it will also remove this element from browser DOM

Let's look at our services file. It is good practice in AngularJS to write services rather than making a direct HTTP call using the `$http` factory.

/assets/js/services/ToDoService.js

The following is the `/assets/js/services/ToDoService.js` file:

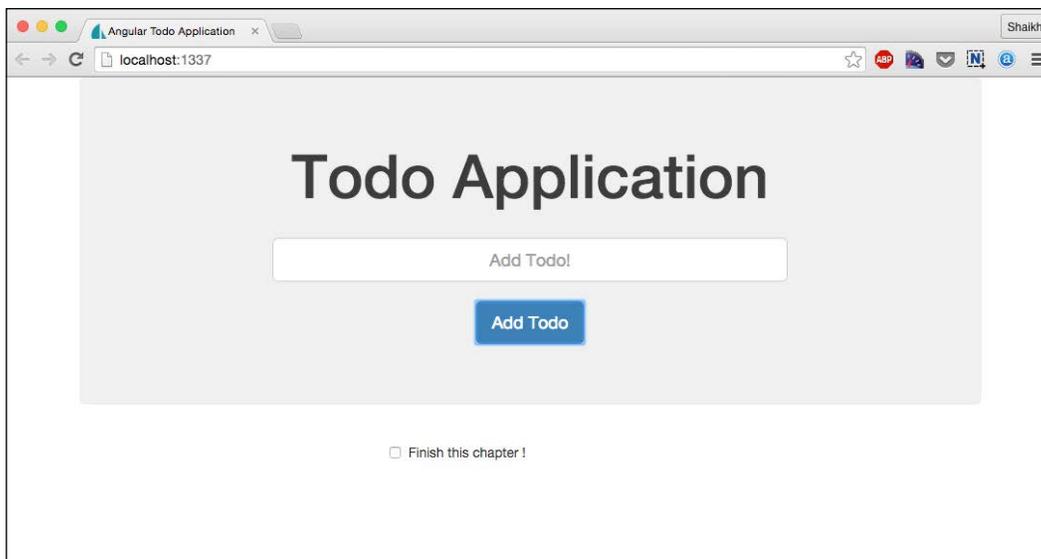
```
todoApp.service('ToDoService', function($http, $q) {
  return {
    'getTodos': function() {
      var defer = $q.defer();
      $http.get('/todo').success(function(resp) {
        defer.resolve(resp);
      }).error(function(err) {
        defer.reject(err);
      });
      return defer.promise;
    },
    'addTodo': function(todo) {
      console.log(todo);
      var defer = $q.defer();
      $http.post('/todo', todo).success(function(resp) {
        defer.resolve(resp);
      }).error(function(err) {
        defer.reject(err);
      });
      return defer.promise;
    },
    'removeTodo': function(todo) {
      console.log(todo);
      var defer = $q.defer();
      $http.delete('/todo/'+todo.id, todo).success(function(resp) {
        defer.resolve(resp);
      });
    }
  };
});
```

```
    }).error( function(err) {
      defer.reject(err);
    });
    return defer.promise;
  }
}
})
```

As you may have noticed, we are calling our API using the HTTP methods—GET, POST, and DELETE—by providing proper data. The `$q` is a service that makes us run the functions asynchronously and use their return value when its available.

You can refer to the official documentation ([https://docs.angularjs.org/api/ng/service/\\$q](https://docs.angularjs.org/api/ng/service/$q)) of `$q` for more details.

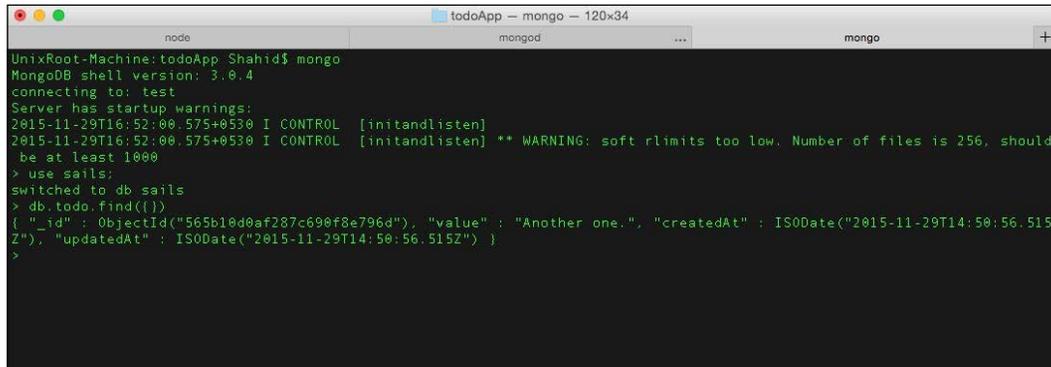
Now, we are done with the frontend design as well. Upon running our application, the following image is the end result:



We can mark the TODO as completed upon checking the box. Let's check our MongoDB collection for the same. Open MongoDB in the terminal and use the following commands to fetch the data:

```
use sails;
db.todo.find({})
```

Note that the collection name is the API name of Sails.js. The following is the output of the previous commands:

A terminal window titled 'todoApp - mongo - 120x34' showing the MongoDB shell interface. The user has connected to a 'test' database. The shell displays startup warnings and a warning about soft rlimits. The user enters 'use sails;' and the shell switches to the 'sails' database. Then, the user enters 'db.todo.find()' and the shell returns a JSON document representing a todo item.

```
UnixRoot-Machine:todoApp_Shahid$ mongo
MongoDB shell version: 3.0.4
connecting to: test
Server has startup warnings:
2015-11-29T16:52:00.575+0530 I CONTROL [initandlisten]
2015-11-29T16:52:00.575+0530 I CONTROL [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should
be at least 1000
> use sails;
switched to db sails
> db.todo.find()
{ "_id" : ObjectId("565b10d0af287c690f8e796d"), "value" : "Another one.", "createdAt" : ISODate("2015-11-29T14:50:56.515
Z"), "updatedAt" : ISODate("2015-11-29T14:50:56.515Z") }
>
```

Yes, it is working as expected. We have our TODO application ready to add/delete and list all the TODOs from super fast MongoDB.

Summary

We covered MongoDB integration and how to use it in our REST APIs. You learned how to use AngularJS services to call the Sails.js APIs and use them to develop modern web applications such as the TODO app. In the next chapter, we will cover some important topics about the production checklist, which we need to follow before going live.

8

Sails.js Production Checklist

Sails.js comes up with a lot of readymade stuff that we came across throughout this book. The project structure, database drivers, and code management approach of Sails.js is production-ready; however, there are some points that we have to look over before pushing it to the production.

In this chapter, we will cover the following topics:

- Sails.js migrate in detail
- Sails.js security checklist
- Sails.js deployment checklist
- Sails.js hosting

Sails.js migrate in detail

We briefly studied about **migrate** in *Chapter 5, Build a Chat System Using Sails.js* and *Chapter 6, Building a Real-Time News Feed App Using Sails.js*. Let's take a look at it again with some examples. The `migrate` is the keyword in the model file that tells Sails.js **object-relational mapper (ORM)** what to do when we initialize the application.

There are three values of `migrate`, as shown in the following:

- `safe`: Never do the database operation. Developer will do this before running application.
- `alter`: This migrates the model changes in database; however, it keeps the existing data if the model is already present.
- `drop`: This deletes the model and data and regenerates it every time the Sails application is lifted.

For production, the recommended or must-have value for the `migrate` key is `safe`. The single reason behind this is database integrity. You should not play with the production data as it has more value than your application.

On the other hand, for development purpose, you can use values other than `safe` as well.

Sails.js security checklist

This is one of the important parts in the Sails.js application. Like I previously mentioned, Sails.js comes up with every possible piece of code other than your business logic. Sails.js also supports various common security vulnerability patches, some of them are as follows:

- **Cross-Site Request Forgery (CSRF)**
- **Cross-Origin Resource Sharing (CORS)**
- **Distributed Denial of Service (DDoS)** attack
- **Cross-Site Scripting (XSS)** attack

CSRF

CSRF is an attack where web browsers running at the user's end perform some action without the permission of the user due to some malicious code loaded via the website, e-mail, or anything that they are currently using on their system.

For example, your application is running on Sails.js and a hacker uses the cookies of client browser to perform an extra HTTP call to another server in order to steal the current session data. Your application should detect such kind of attack and prevent it as much as possible.

To enable CSRF protection, open the `config/csrf.js` file and uncomment the following code and change `false` to `true`:

```
module.exports.csrf : true
```

This will make sure that all standard security provision related to CSRF is running into your application.

CORS

CORS is a mechanism where we need to allow our application to be called from multiple servers that reside on different domain. You might need to allow this for your application; however, we will recommend allowing only those domains that are trusted.

To enable this open `config/cors.js` and change the following code:

```
allRoutes: true
```

Also, change the origin to the domain name instead of `*`. You can also add more settings if your app requires it.

This is one of the most common and effective attack that can happen to your web application. You might have heard about the **Anonymous** group that took down various websites of the government officials as a protest. Most of the time, they use a DDOS attack to achieve their objectives.

DDOS

DDOS prevention is one of the biggest research domain in the security world. To do the most from our end, we should use the application in microservice architecture and perform its clustering so that if a domino goes, other will take its place. If one child node of the cluster goes down, another can take its place and the traffic is diverted among the remaining child nodes and the service is unaffected.

On the other hand, you should use memory store for session management, such as **Redis**, than using default store that uses your memory to handle the session. If a hacker, by any chance, puts your memory into overflow condition, your system may crash and that would be denial of the service for the user.

XSS

XSS is a type of attack where a malicious agent manages to inject client-side JavaScript into a website so that it runs in the trusted environment of your user's browsers.

In order to prevent this attack, you need to make sure that you are using strict validation of data before passing it to the data layer. Also, try not to commit common JavaScript mistakes, such as using the `eval()` function, which lead to an XSS attack.

Here is one of the popular references of XSS prevention steps:

```
https://www.owasp.org/index.php/XSS\_\(Cross\_Site\_Scripting\)\_Prevention\_Cheat\_Sheet
```

An XSS attack may or may not lead to a DoS attack as well. You need to be careful here and follow the mentioned guidelines.

Sails.js provides data validation functions for various types such as integer, Boolean, string, and so on. Check out the complete list of functions by visiting the given URL, which you can use to perform data validation in Sails.js:

<http://sailsjs.org/documentation/concepts/models-and-orm/validations>

Sails.js deployment checklist

Before you deploy your application, make sure that you have performed the following:

- Configure production environment setting
- Run app on port 80 if there is no proxy
- Configure the database settings
- Enable CSRF protection for your POST, PUT, and DELETE requests
- Enable CORS
- Estimate the traffic from all the endpoints (such as web, mobile, and so on)

Configure production environment setting

Open `config/env/production.js` and change the settings, such as the following:

- Port
- Database adapter

Run app on port 80 if there is no proxy

Port 80 is the default port, where the browser listens when we hit the request. If you are serving your application via Sails.js, then make sure that you have changed the port to 80 from the production file.

If you are using Nginx and Socket in your application, then make sure that you make the changes to relay WebSockets to your app in Nginx.

Configure database settings

If you are using relational database such as MySQL or Oracle, then make sure that you have set the migration setting in Sails.js configuration file properly. In the production environment, no auto-migration will be done by Sails.js, you need to make sure that database is configured properly.

Estimate the traffic from all the endpoints

If your application is used by multiple nodes such as mobile, web, and other systems, then you need to estimate your traffic for possible server configuration. Better switch to cloud if you cannot estimate or are uncertain about the traffic.

In order to deploy it to the server, we recommend you to use the **PM2** (check it in npm) process manager. You can also deploy using the **forever** node module or running the `sails lift` command in the daemon mode.

Sails.js hosting

You can host the Sails.js application in any **virtual private server (VPS)** or dedicated host. Sails.js runs over Node.js, which is quite compatible with any major operating system (we recommend the Linux version), so there is hardly any chance of compatibility issues.

There are some managed VPS services such as **Heroku**, **Modulus**, and so on, where you can easily deploy your application. You can also use **Amazon**, **DigitalOcean**, or any other service providers.

Summary

Sails.js is already a well-designed code that is quite ready for production. In this chapter, we looked over a few things related to the configuration, security, and deployment that will help us keep our application running smoothly in the production environment.

Index

Symbols

\$q service
URL 76

A

Active Server Pages (ASP) 19
AngularJS app
for client-side interaction 47-51
API
model, defining for 69-71
api directory 23
app
database design 57, 58
implementing 58-66
assets directory 23

C

chat application
architecture 42
flow 42
operation handling, Sails.js controller
used 46, 47
running 52, 53
Sails.js API for 44
chat system 41
Chrome Web Store 35
client-side interaction
AngularJS app 47-51
config directory 24
create operation 10
Cross-Origin Resource Sharing (CORS) 81
Cross-Site Request Forgery (CSRF) 80

Cross-Site Scripting (XSS) attack
about 81
URL 81
CRUD (create, read, update, and delete)
operation 10, 32

D

database
support, adding 25
Data Definition Language (DDL) query 44
data validation
URL 82
delete operation 10
deployment checklist
about 82
app, running on port 80 82
database settings, configuring 83
production environment setting,
configuring 82
traffic from all endpoints, estimating 83

E

Embedded JavaScript (EJS) templates file
about 23
flow 42
event loop
about 5
and non-blocking I/O model 5
working 6, 7
Express
used, for developing web server 12-15
Express.js 23

F

forever node module 83

G

Grunt

about 23
task runner file configuring, JSHint
used 27-29

H

HTTP headers 12

HTTP module

used, for developing web server 11

HTTP operations 10

I

Internet Relay Chat (IRC) 42

J

JSHint

used, for configuring Grunt task 27-29

just-in-time (JIT) 2

L

libuv library

working 3

lines of code (LOC) 2

M

middleware 16

migrate

about 79
alter value 79
drop value 79
safe value 79

model

and MySQL integration, in app 44, 45
defining, for API 69-71

Model-View-Controller (MVC) pattern 19

MongoDB database

config/connections.js file 27

config/models.js file 27

configuring, Sails.js used 26

MongoDB support

in Sails.js 68, 69

MVC concepts

about 19
controller 20
model 20
view 20

MySQL database

config/connections.js file 26

config/models.js file 26

configuring, Sails.js used 25

N

Node.js

about 1, 21
architecture 2

Node.js, architecture

libuv library 2
V8 JavaScript engine 2

node_modules directory 23

nodemon 15

npm 21

O

object-relational mapper (ORM) 79

P

phpMyAdmin 32

PM2 83

Portable Operating System Interface (POSIX) 5

Postman chrome extension 35

R

read operation 10

Redis 81

representational state transfer (REST)
API 29, 31

REST API

- api/controllers/MessageController.js file 40
- building 33
- code, running 35
- config/connections.js file 34
- config/models.js file 34
- custom controller, defining 40
- database design 32
- migrate 34

REST API, code running

- message, deleting 39
- message, reading 37, 38
- message, updating 38, 39
- new message, creating 35-37

routers 16

routes 11

S

sails-disk 23

Sails.js

- API, for chat 44
- app, creating 42, 43
- controller, to handle chat operation 46, 47
- deployment checklist 82
- directory structure 22
- hosting 83
- installing 21
- MongoDB support 68, 69
- security checklist 80
- Socket, using 56, 57
- used, for configuring MongoDB database 26
- used, for configuring MySQL database 25, 26

Sails.js, directory structure

- about 22
- api directory 23
- assets directory 23
- config directory 24, 25
- node_modules directory 23
- views directory 23

sails-mongo module 26

security checklist

- about 80
- Cross-Origin Resource Sharing (CORS) 81
- Cross-Site Request Forgery (CSRF) 80
- Cross-Site Scripting (XSS) attack 81

single-threaded system

- about 3
- versus multi-threading 4, 5

Socket

- using, in Sails.js 56, 57

Socket.IO 56

sudo 21

T

TODO app view design

- /assets/js/app.js file 74, 75
- /assets/js/services/ToDoService.js file 75-77
- about 72-74

TODOs 67

U

Uniform Resource Locator (URL) 9

update operation 10

V

V8 JavaScript engine 2

views directory 23

virtual private server (VPS) 83

W

web server

- developing, Express used 12-15
- developing, HTTP module used 11
- working 9, 10

WebSocket 55



Thank you for buying **Sails.js Essentials**

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

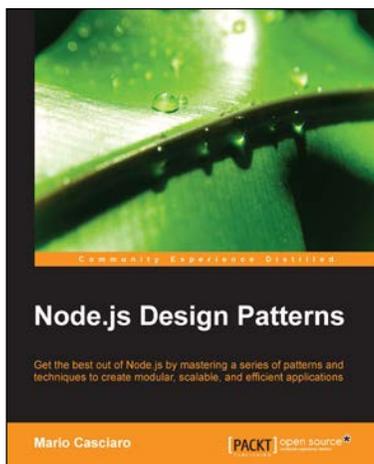
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



Node.js Design Patterns

ISBN: 978-1-78328-731-4 Paperback: 454 pages

Get the best out of Node.js by mastering a series of patterns and techniques to create modular, scalable, and efficient applications

1. Master the foundations of Node.js application design by diving into its core patterns and components.
2. Learn tricks, techniques and best practices to solve common design and coding challenges.
3. A hands-on guide presented out with a code-centric approach for designing and creating Node.js applications without friction.



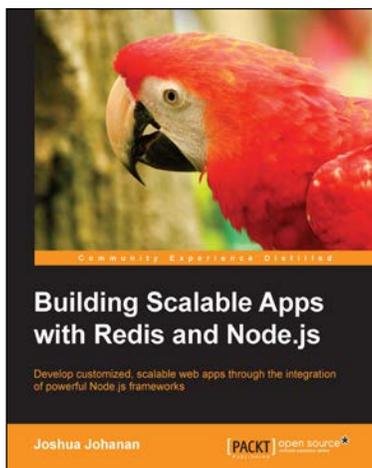
Node.js Blueprints

ISBN: 978-1-78328-733-8 Paperback: 268 pages

Develop stunning web and desktop applications with the definitive Node.js

1. Utilize libraries and frameworks to develop real-world applications using Node.js.
2. Explore Node.js compatibility with AngularJS, Socket.io, BackboneJS, EmberJS, and GruntJS.
3. Step-by-step tutorials that will help you to utilize the enormous capabilities of Node.js.

Please check www.PacktPub.com for information on our titles

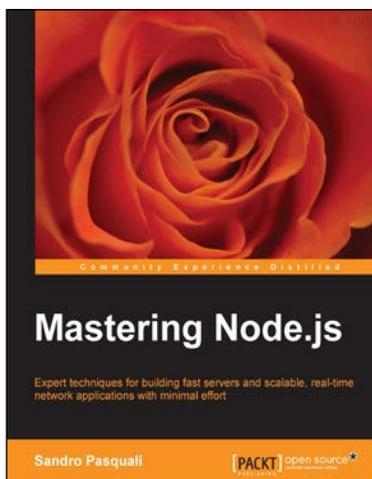


Building Scalable Apps with Redis and Node.js

ISBN: 978-1-78398-448-0 Paperback: 316 pages

Develop customized, scalable web apps through the integration of powerful Node.js frameworks

1. Design a simple application and turn it into the next Instagram.
2. Integrate utilities such as Redis, Socket.io, and Backbone to create Node.js web applications.
3. Learn to develop a complete web application right from the frontend to the backend in a streamlined manner.



Mastering Node.js

ISBN: 978-1-78216-632-0 Paperback: 346 pages

Expert techniques for building fast servers and scalable, real-time network applications with minimal effort

1. Master the latest techniques for building real-time, big data applications, integrating Facebook, Twitter, and other network services.
2. Tame asynchronous programming, the event loop, and parallel data processing.
3. Use the Express and Path frameworks to speed up development and deliver scalable, higher quality software more quickly.

Please check www.PacktPub.com for information on our titles