Sails.js Essentials

Sails.js makes it easy to build custom, enterprise-grade Node.js apps. It is designed to emulate the familiar MVC pattern of frameworks, but with support for the requirements of modern apps: data-driven APIs with a scalable, service-oriented architecture.

This book will provide practical examples to get you started with Sails.js and get you developing production-ready apps in no time. We will take you from a Node.js web server and a single threading system to a general MVC architecture. You will then learn to develop applications that you might be already familiar with using Sails.js. Finally, we will show you how to create a Chat app and a TODO application, and improve code quality using JSHint.

Who this book is written for

This book is for developers who want to learn about the development of production ready web applications using Sails.js.

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

What you will learn from this book

- Find out how to get started with Sails.js and develop production-ready apps
- Understand how to apply the MVC model of software development to Node.js using Sails.js
- Discover how to write quality code using Sails.js
- Integrate third-party APIs and databases such as Mongo and MySQL with Sails.js
- Get to know the advantages and importance of the MVC framework in software development
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Revisiting Node.js Concepts'
- A synopsis of the book’s content
- More information on Sails.js Essentials
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.
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.
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:

![Node.js Architecture Diagram]

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:

![Diagram of libuv working](image)

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
Revisiting Node.js Concepts

- 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:

![Event Loop Diagram](image)

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:

```javascript
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:

```javascript
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.
Where to buy this book

You can buy Sails.js Essentials from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.