Getting Started with hapi.js

This book will introduce hapi.js and walk you through the creation of your first working application using the out-of-the-box features hapi.js provides. Packed with real-world problems and examples, this book introduces some of the basic concepts of hapi.js and Node.js and takes you through the typical journey you'll face when developing an application. Starting with easier concepts such as routing requests, building APIs serving JSON, using templates to build websites and applications, and connecting databases, we then move on to more complex problems such as authentication, model validation, caching, and techniques for structuring your codebase to scale gracefully. You will also develop skills to ensure your application's reliability through testing, code coverage, and logging.

By the end of this book, you'll be equipped with all the skills you need to build your first fully featured application. This book will be invaluable if you are investigating Node.js frameworks or planning on using hapi.js in your next project.

Who this book is written for

If you are a JavaScript developer with or without Node.js experience and would like to learn to build applications, APIs, and web servers with the best-in-class framework hapi.js, this book is perfect for you.

What you will learn from this book

- Increase your productivity by taking advantage of the out-of-the-box features hapi.js provides
- Build secure API servers
- Create websites and applications using your favorite templating language
- Leverage hapi.js plugins to better structure your codebase
- Simplify your security workflows with the built-in authentication and authorization functionality of hapi.js
- Ensure application reliability with testing and code coverage
- Reduce code complexity using reusable validation logic with joi
- Gather insight into your application performance via logging
- Start the journey to building robust production-ready applications

Build well-structured, testable applications and APIs using hapi.js

Foreword by Jan Lehnardt, CEO, Neighbourhoodie Software

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Introducing hapi.js'
- A synopsis of the book’s content
- More information on **Getting Started with hapi.js**
About the Author

John Brett is a software engineer passionate about new technologies, open source, and building useful software while travelling the world. Starting out in Ireland with a degree in Computer Science and Software Engineer, he went on to join IBM through their premier internship Extreme Blue, later working on collaboration software as part of IBM's Software Group. There, John mainly worked with PHP and JavaScript, only working with Node.js in his spare time.

Later in his career, John moved to a growing SAAS company called D4H Technologies. As a lead engineer there, he developed tools to aid emergency response teams globally, all from a lighthouse in Dublin, Ireland. Here, John got more exposure to Node.js, leading D4H's first venture into Node using hapi.js, later joining the hapi.js core team.
Node.js has become one of the most exciting development platforms in recent times. Combined with the largest package ecosystem in the world, npm, it has become a game changer in enabling higher developer productivity strides in improved application performance and a more enjoyable developer experience overall, which has lead to widespread industry adoption.

This book is focused on hapi.js, which is a rich Node.js framework for building applications and services with the goal of enriching the development experience further. It aims to increase your productivity by providing the tools you need when you want them and getting out of the way when you don't.

Developed by the talented team at Walmart Labs to handle the pressures of the Black Friday traffic, the busiest day of online retail on the calendar, it is proven at scale and has since seen wider adoption, participation, and collaboration from the wider community. Its growing adoption can be put down to how easy it is to use and get started with along with its proven stability.

The Node.js programming paradigm is quite different to other server-side technologies, and as such, it can be common for developers to get stuck on design and programming problems, such as dealing with the asynchronous nature of JavaScript and structuring an application as it grows. This is where hapi.js excels; it enables developers to focus on writing reusable business logic instead of losing time focusing on building application infrastructure without obscuring important implementation details.

While it was originally built as a framework focused toward the rapid development of RESTful APIs, thanks to its fantastic ecosystem, it has grown into a fully-fledged framework suited to building static file servers, websites, and web applications with excellent support for building real-time applications added more recently.
Due to the modular plugin approach it provides, it allows an easier collaboration between teams in larger enterprise environments as well as an easy point of integration for third-party libraries.

If you plan to use hapi.js in your next project, whether it be a side project, work project, or open source project, this book seeks to give you an excellent point at which to get started and solidify any knowledge you might have if you have used Node or hapi.js before.

What this book covers

Chapter 1, Introducing hapi.js, serves as an introduction to hapi.js and some prerequisite learning of Node.js. It explains what hapi.js is, who created it, why it needs to exist, and how to create a simple hapi.js server.

Chapter 2, Adding Functionality by Routing Requests, is the beginning of your journey of building applications with hapi.js. It will cover some basics, such as routing and building APIs, websites, and applications with hapi.js, and will also cover some differences in design compared to other Node.js-based frameworks.

Chapter 3, Structuring Your Codebase with Plugins, introduces the plugin API in hapi.js, which is one of its core features. It will walk you through how to simplify an application structure gracefully using plugins and also cover server orchestration using modules from the hapi.js ecosystem.

Chapter 4, Adding Tests and the Importance of 100% Code Coverage, explores the importance of adding tests to an application. It will introduce you to some of the testing tools available in the hapi.js ecosystem and also guide you toward getting 100% code coverage with your tests.

Chapter 5, Security Applications with Authentication and Authorization, takes a deep dive into how authentication works in hapi.js with practical examples of different authentication protocols, including social logins. It also demonstrates scopes and how to use them to add permission levels to your applications.

Chapter 6, The joi of Reusable Validation, deals with the topic of validation in applications using the joi validation library from the hapi.js ecosystem. It covers how to validate objects and route configurations and how to reuse this to generate documentation for applications.

Chapter 7, Making Your Application Production Ready, covers multiple topics to leave you ready to build your first application, such as integrating popular databases and adding caching and logging to applications to help scale an application as well as give it visibility when running in a production environment. It also covers a variety of other sources to continue learning more about hapi.js.
Introducing hapi.js

hapi.js (commonly referred to as hapi) stands for HTTP API. It is a rich framework for building applications and services. It was originally designed for the rapid development of RESTful API services using JavaScript, but has since grown to be a full web application framework with out-of-the-box features for templating, input validation, authentication, caching, and more recently, support for real-time applications with web socket support.

The original philosophy that hapi was built around was increasing developer hapi-ness; the aim was to increase productivity by providing additional tools to help with development, but without getting in the way. It was also built with a security-first approach, meaning that the tools provided were developed with smart secure defaults, with the mindset of not giving the developers the ability to shoot themselves in the foot for not knowing some hidden configuration setting or implied design pattern.

hapi was created by the Mobile team at Walmart Labs, led by Eran Hammer (who created OAuth), to handle their traffic for events like Black Friday, one of the busiest days for online shopping in the US calendar.

hapi was born out of necessity; the Walmart team never intended to build a framework. They originally started with express, currently Node's most widely used framework. After hitting some limitations with express, and finding similar limitations in other frameworks, they finally discovered that it would be easier to create their own framework rather than hack an existing framework to meet their needs. Eran wrote a great post about this journey on his blog, http://hueniverse.com/2012/12/20/hapi-a-prologue; I encourage you to read it. Fortunately for us, hapi was born out of all this.
This chapter will be your introduction to hapi.js, and will cover the following topics:

- Introducing Node.js—a prerequisite to learning hapi.js
- A background on hapi.js
- Creating our first hapi.js server

## Node.js – a prerequisite to learning hapi.js

As this book is aimed at JavaScript developers, who may or may not have prior experience with Node.js (https://nodejs.org/), I would like to first introduce Node.js. Node.js (commonly referred to as Node) is a platform built on Chrome’s JavaScript runtime called V8 for easily building fast and scalable web applications in JavaScript. To put it succinctly: server-side JavaScript.

The thought of server-side JavaScript might seem strange to those unfamiliar with the concept, but it has some big advantages over other server-side technologies such as PHP, Java, Ruby, and many others. For one, V8, on which Node is based, is fast, and has a full-time team in Google constantly improving its performance with each release of Chrome. One of Node’s biggest differentiators as compared to other server-side technologies is its single-thread nature, encouraging the asynchronous coding style that you may be familiar with from working with JavaScript on the client side. Among its asynchronous nature, Node.js has many advantages, some of which are as follows:

- Reduces the need for multi-threading, always considered a tough problem in application development
- Allows sharing of code between browser and server, and avoiding a context switch between working with the browser and client
- It comes bundled with npm, currently the biggest package manager and one of the best solutions out there for managing your application dependencies on both client and server seriously, you will struggle to move back from Node after having the luxury of such an excellent resource

As a result of these and many other benefits of Node, many large organizations such as Netflix, Yahoo, Mozilla, PayPal, and of course the organization behind hapi.js, Walmart, have adopted Node. If you aren’t familiar with Node or npm, I suggest you take some time to read up on them at https://nodejs.org/about, https://docs.npmjs.com. The Node.js website covers the installation of Node, and I suggest you do this so you can participate in this next part.
Let's take a look at what a simple 'hello world' API server looks like in Node.js by using the example from the Node.js website (please take note that you need Node version 4 or greater to run this code, otherwise you will get syntax errors!):

```javascript
const http = require('http');                           // [1]
const hostname = '127.0.0.1';
const port = 1337;
http.createServer((req, res) => {                       // [2]
  res.writeHead(200, { 'Content-Type': 'text/plain' }); // [3]
  res.end('Hello World
');                             // [4]
}).listen(port, hostname, () => {                       // [5]
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

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:

- Log in or register to our website using your e-mail address and password.
- Hover the mouse pointer on the SUPPORT tab at the top.
- Click on Code Downloads & Errata.
- Enter the name of the book in the Search box.
- Select the book for which you're looking to download the code files.
- Choose from the drop-down menu where you purchased this book from.
- 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
- Zippeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux
Introducing hapi.js

I won't cover what happens here in detail, but will skim over the need to know parts to help understand how a typical Node server operates. With reference to the numbers in the code comments, let's go through each line:

- [1]: This is an example of using modules in Node.js. While many modules need to be downloaded using `npm install`, some modules like `http` are bundled with the Node binaries. In this example, we require the `http` module and assign it to the variable `http`. This is where we first see the ES6 syntax being introduced through the `const` keyword. Here, `const` basically specifies that we cannot redefine the `http` variable. If you want to read more about how Node modules work, you can find a good explanation in the Node API documentation at https://nodejs.org/api/modules.html.

- [2]: Here, we use the `createServer` method of the `http` module. We pass a callback function as an argument, which will then be called every time the server receives a request. This is the second example of the ES6 syntax being introduced, which is the arrow function. I suggest you read up on its differences with a normal function call at https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Functions/Arrow_functions. The callback function then takes two parameters: `req`, which contains information about the request that the server has received, and `res`, which is an object that contains the methods for responding to a request.

- [3]: Here we see the low level to which the Node `http` module goes, where you must set response headers and status codes; this is done by calling `res.writeHead()`.

- [4]: The line `res.end()` signals to the server that all of the response headers and body have been sent, and the server should consider this request complete.

- [5]: Now that we have a server defined, we must tell it to accept connections for a particular port and hostname, and this is done by the `server.listen()` method.

That was a lot of information at once, so don't worry if it didn't make sense immediately; the concepts introduced here will become clearer as we talk more about them throughout this book.

To run this example, you should have Node installed. As I mentioned previously, instructions can be found on the Node.js website. Next, you need to create a folder and change to its directory:

```
$ mkdir hello-node && cd hello-node
```
Then create a file called `hello-node.js`, paste it in the preceding code and run it using the following command:

```
$ node hello-node.js
```

If all goes well, you will see the following screen:

Looking back at the preceding example, think now, how would you add functionality to this server while keeping your code base manageable? It certainly is not trivial. Especially when taking into account that typical servers will want to be able to serve static content as well as plaintext, JSON, XML and many other types. Often, it would need to deal with different request URLs and respond accordingly. Generally, it would need to interact with some type of persistent data store such as MySQL, and we still haven't even thought about authentication, authorization, validation of requests, and a number of other features that a typical application would need. This is what we refer to as application infrastructure, and this is where hapi.js comes in.

hapi is, in one sense, an abstraction layer on top of this low-level server, for providing more intuitive APIs for solving the previously mentioned problems, so you don't need to. It is a rich framework that allows you, the developer, to focus on writing reusable application logic instead of spending time building infrastructure.

The hapi philosophy versus other frameworks
Throughout Walmart's journey of rebuilding their mobile services tier and working with other frameworks, they realized that certain values were vital to building applications in Node for keeping the code intuitive, maintainable, and scalable. As a result, hapi centered around the following concepts:

- Time and effort should be spent on delivering value, not on rebuilding application infrastructure
- Configuration is better than code
- Business logic must be isolated from the transport layer
- Open Source and community-centric from day one
Introducing hapi.js

Let me elaborate more on hapi and these points, the thought process behind them, and how they compare to other frameworks in Node.

**Building value, not infrastructure**

As I mentioned at the start of this chapter, hapi aimed to be a solution for the rapid development of RESTful APIs, but has since grown to be a framework for handling other aspects of a fullstack web application like templating, caching, and more. To describe it best, hapi is a unified framework that aims to solve the common problems with building web applications with out-of-the-box functionality so that you don't have to.

There are competing strategies' frameworks use to provide a boost in productivity. Some take the "all-in" and highly opinionated approach by providing large amounts of frameworks features, APIs, and code-generation tooling. Loopback ([http://loopback.io/](http://loopback.io/)) by Strongloop is an example of this in the Node world, or Rails ([http://rubyonrails.org/](http://rubyonrails.org/)) if you are familiar with Ruby. While this offers an initial productivity gain, I'm not a fan of this approach, as these frameworks usually have quite a large API surface area, which means that you spend a lot of time learning the framework API instead of learning JavaScript or whatever the underlying technology is. It also means that if the framework doesn't account for your particular needs, you have to hack against it, and when something goes wrong, you're left debugging generated or framework code which you're unfamiliar with. Experts in these types of frameworks can be very productive, but I generally prefer approaches that involve more exposure to application code and structure, as I find it's one of the best foundations in which to grow as a developer.

In contrast to the all-in framework approach, is the minimalist framework approach, which usually aims to act more as a user-friendly API. express on the server side or Zepto.js ([http://zeptojs.com/](http://zeptojs.com/), a lightweight jQuery alternative) are examples of these. There are lots of advantages of these kind of frameworks, as they remove a lot of boilerplate such as accounting for different runtime environments or browser differences, and usually have less performance overhead as the library size is small.

The downside to these frameworks is that, for larger apps, it is generally up to the developer to create their own framework on top of these to break up the business logic of the application into smaller, more loosely coupled, and manageable parts. This is one of the more complicated parts of building non-trivial applications. It also leads to spending time rebuilding common features of an application which generally don't add much business value, for example authentication. It was this desire to spend time on business value instead of building application infrastructure and common application features that made me research the available Node frameworks, and thankfully, I found hapi.
hapi treads the line quite nicely between being minimalist and providing too much tooling. Some of the out-of-the-box functionality I mentioned earlier covers areas like authentication, caching, validation, and of course, plugins. The hapi plugin system also provides the perfect abstraction for divvying up your business logic into manageable pieces. It also makes it trivial to avail of plugins from the hapi core team and the wider ecosystem to provide the building blocks you need. It encourages a great level at which to encapsulate the code in a structured and obvious manner. This makes hapi code bases usually much easier to maintain and comprehend due to their plugin-centric approach. I will cover plugins and the out-of-the-box functionality that hapi offers in detail in later chapters.

**Configuration is better than code**

One of hapi's code design principles is centered around the idea that configuration is better than code. This is one of the concepts I disagreed with before using hapi.js. I remember the nightmare of configuring development environments in Java and PHP, and I thought this is where Node.js excels—avoiding config problems! However, I've found application/framework configuration done well as an excellent approach for a number of reasons.

For one, when done with smart, secure defaults, it helps developers avoid making unnecessary mistakes. You shouldn't be punished for not knowing some configuration setting, or for some non-obvious convention. The thought process behind this is that it is easier to detect missing/wrong configuration than missing code. This gives hapi.js a huge advantage in that it provides much more detailed error messages for anything that is misconfigured.

You'll find so many more errors with detailed error messages on startup instead of at runtime. This speeds up the feedback loop on the effect your added/changed code has, which makes developing applications much more enjoyable and satisfying. A great example of this is in-route configuration. Any mistake here will be listed on server start, with line numbers and a detailed explanation of what is wrong, instead of trying to debug why handlers are not being called, or being called in the wrong order. These types of runtime errors often throw no error at all, and are quite difficult to debug.

Another example of this is limiting the file upload size by default. This means when you release your application to production, file uploads won't buffer an entire file into memory crashing your server. With hapi, by default, the max upload size is limited, and an error message will be returned to the user, making it clear that the file size is too large. You, the developer, then have the choice to increase this and learn about the pitfalls of buffering files into memory, while your application is up returning error messages, and not while your application is in production, being repeatedly brought down by large file uploads.
Introducing hapi.js

Separation of business logic from the transport layer

The hapi team encountered a number of issues in their early attempts to build their mobile tier at Walmart with other frameworks, a number of which stemmed from a mix of business logic with a transport layer. These issues ranged from trivial to complex.

In the preceding Node.js example, we've seen that we have to set headers, status codes, and specify the end of the response. While this is easy to follow in the preceding example, this quickly leads to unnecessarily complex code when returning more complicated structures. For example, if you swap out 'hello world' with a normal JavaScript object, the server now fails to return anything, and even worse, you won't discover this until you test how the server is responding to requests when running.

A more complicated example of this is routing collisions. In express and most other Node.js frameworks, routing is non-deterministic, which means that you can have two routes that will both be called for a particular URL path. This, like the preceding issue, is a more prevalent issue in larger applications where there may be different teams working on different parts of a bigger application. This can be quite hard to test for it is not an error, and can usually only be tested for at request time, much like the aforementioned issue. Furthermore, the order in which they are called is decided by the order in which they are registered.

This means that developers have to be careful when registering routes, or they may accidentally register a handler that is called twice, or expose a certain route before an authentication handler, for example, causing security issues. Routing conflicts may seem easy to spot, but when there are a large number of routes, and they can be defined using regular expressions as the routing patterns, this can happen more easily than you think.

To combat this, the hapi team designed a deterministic routing algorithm ensuring that there can be no conflicting routes. To aid productivity and developer happiness here, conflicting routes will fail on server start with the initial run, providing a detailed error message and thus making it simple to debug and fix.

They also created a reply interface that accepts objects, node streams, strings, promises, and buffers. This interface detects the data type, and acts accordingly so that you don't have to do extra work for the common, yet different response types.

Eliminating the possibility of conflicting routes, and reducing areas where you can have no error with no response via the reply interface are great examples of where hapi does its best to increase developer productivity by making it easier to catch common errors that can normally be difficult to spot.
Open source and community-centric

With hapi, you don’t just get a framework, you also join a welcoming and inclusive community that wants to help you learn and succeed. You will find that most questions are asked through GitHub instead of the usual Stack Overflow, and are answered quite frequently by other members of the community as well as the core team. The entire time I have been working with hapi, I have been continually impressed by the community—from how it deals with and communicates breaking changes, encourages newcomers, and in how people respond to issues on GitHub. If you are working with hapi, don’t neglect this as a resource.

Like most frameworks these days, hapi has been open source from day one, with a full-time team supporting it. This makes it a great learning resource, and I encourage you to read through the source code at times. Its style guide and talented contributors have created a code base with great examples of clean and well-structured code.

As a result of such great community support and involvement, the open source focus and hapi’s excellent plugin system, a large ecosystem of quality plugins has been created for hapi too. A website has also been created to navigate these at https://hapi-plugins.com/.

The encouraging and inclusive nature of the community meant that my first npm module was a hapi module, my first conference talks were about hapi, I eventually become a lead maintainer of one of the core library modules, and am now writing a book (this!) to share the positive experiences that I’ve encountered in developing with hapi and being a part of its community.

Ecosystem

The hapi core ecosystem was the first of its kind (that I’m aware of) to only use modules that had 100% code coverage, strictly adhered to SemVer (http://semver.org/), a specification for communicating breaking changes, and only depended on modules that adhered to the same criteria.

If you’re unfamiliar with SemVer, and it’s something I also encourage you to read about, it stands for semantic versioning. It’s a method of versioning modules that is easier for humans to read and understand (as opposed to machines), which communicates clearly breaking changes. When done properly, this means you can update the modules that your application depends on, availing of new features, performance improvements, bug fixes, and so on, without worrying about breaking your application.
Introducing hapi.js

In contrast, most modules/plugins for other frameworks are created by third-party users, and hugely vary in quality. Many have poor (or no) test coverage, and may cover only one particular use case. This leads to module discovery being a problem where the productivity increase of availing the existing modules is offset by having to take time to search, read the source, write your own tests, and debug issues of third-party modules.

Thankfully, the core hapi modules not only cover a wide range of functionality, but also have set a high standard for third-party modules; this has lead to an ecosystem of high-quality plugins for hapi. You can view the list of core modules within the hapi GitHub organization at https://github.com/hapijs.

Small modules

I often hear the argument of using only small modules to build applications in the Node community. With modules that do only one thing and do that one thing well, it makes it quite easy to use many of these to create full-featured applications easily. While this is appealing over learning the full API of a framework, it leads to the problem where the onus is now on the developer to decide on an application structure, write a lot of unnecessary code and infrastructure, and make a lot of unnecessary decisions.

The structure hapi provides, along with the use of configuration over code, leads to features and clever tools not possible with more minimal frameworks, or with the module-only approach. Examples of these are documentation generation tools like lout from the core team and hapi-swagger from Glenn Jones, which we will take a look at in later chapters.

Summarizing hapi

To summarize this section, hapi is a unified framework that aims to solve most problems that you encounter when building web applications as opposed to more minimalist frameworks that focus on providing more user-friendly APIs and utilities. It is building a lot of momentum and adoption in recent times, and you can check out the many companies using hapi on the hapijs.com community webpage at http://hapijs.com/community. But enough of the background—let’s take a look at what a sample hapi.js server looks like in the next section.
Creating our first hapi.js server

I mentioned earlier in the chapter that I like to approach code and examples with a problem solving, code-first approach. So let's try that here. Our aim is to create an initial server that can respond to requests and will be our first experience with hapi. Let's see what it takes to install hapi and get our first hapi server running. I assume that by this stage, you have Node.js installed; if not, the Node.js website explains how to do this quite well. I also assume you have some experience with the command-line interface of your OS of choice. So to start our hapi.js project, first we'll create a folder within which we will create our server, Open your command line and type the following:

```bash
$ mkdir hapi-hello && cd hapi-hello
```

This creates a directory called hapi-hello, and makes it the current directory. Next we'll use npm to create our project metadata using `npm init`. If you installed Node successfully, the `npm` command will also have been added to your command line/terminal. One of the reasons to run this any time you start a node project is that any Node modules you install will be installed into this directory instead of its parent directory, as is the npm algorithm for installing modules. This also creates a `package.json` file in your current directory with all the metadata from the `npm init` command. You should take a look at this now.

### Installing hapi

Next we will install the hapi module for use with this project:

```bash
$ npm install hapi --save
```

The `--save` flag tells npm to store this dependency in our `package.json` file. This is quite convenient for sharing this code with others, so we only need to share the actual application code and this `package.json` file. For another person to then run this project, they would just navigate to this directory and run `npm install`, and it would install all the dependencies listed in the `package.json` file. No messing around with the configuration of any kind.

### Our first hapi server

With that, let's look at the code involved in our first hapi.js server:

```javascript
const Hapi = require('hapi');                          // [1]
const server = new Hapi.Server();                       // [2]
server.connection({ port: 1337, host: '127.0.0.1' });   // [3]
server.route({                                            // [4]
```

[11]
Introducing hapi.js

```javascript
path: '/', // [4]
handler: function (request, reply) {
  return reply('Hello World\n'); // [4]
}; // [4]
server.start((err) => {
  if(err) { // [5]
    throw err; // [5]
  } // [5]
  console.log(`Server running at ${server.info.uri}`); // [5]
}); // [5]
```

Again, I won't cover every detail here, but will give you the general gist of what's happening. Hopefully, you've noticed some similarities and differences between this example and the vanilla Node example given previously. With reference to the numbers in the code example, let's examine each section:

- [1]: We require the hapi module and store it in a variable Hapi, similar to importing the http module.
- [2]: We create a server with the hapi module, which is, in fact, the only function of the hapi module.
- [3]: Next we add a connection, which is a listener. We add the port and host configuration. This is quite similar to the listen function of the http server object, but here we configure it first, before we call listen or start our server. If no port is specified, a random port will be assigned.
- [4]: A sample route is added. This is slightly different to the preceding example, as it will only respond to 127.0.0.1 with Hello World and a 404 error to all other requests, whereas the preceding node example would respond to any URL. We can, of course, easily update this example to do the same, but I will cover that in the next chapter. This is where we see the configuration versus code approach more clearly—here we configure the route options as opposed to using code with a configuration object.
- [5]: Finally, we start the server that we have created. We also make sure to handle the error provided from the server.start() callback. This then ensures that all configuration is correct; if not, we want to throw the error so the server will not start and will send the error to the console, making it easy for us to debug.

What I hope is evident from the preceding example is that even though slightly more verbose, the hapi example is quite simple to follow and understand. All configuration is done when creating the server. Configuration is done when creating a route, which means well-defined handlers that can focus on just handling the request and not dealing with other parts of the transport layer.
To run this example, create a file in your current directory called index.js, and either type or copy-and-paste the preceding code. Then simply run the following command:

$ node index.js

If all goes as planned, you should see the following screen:

If you navigate to this URL, you should have Hello World returned to you. If all goes well, you will see the following screen:

Hopefully, you've been following the examples and creating this server yourself, so congratulations on building your first hapi server! Let's take a look at the configuration versus code here and see how it makes code easier to read, despite being a little more verbose. If you remember the listen method from our vanilla node example:

```
.listen(1337, '127.0.0.1')
```

You would have noticed that it uses parameters to pass in options such as port and host. In hapi, the equivalent is as follows:

```
.connection({ port: 1337, host: '127.0.0.1' })
```

This method takes a configuration object. This may not seem significant at the moment as these examples are quite trivial, but imagine when you have much more parameters which are Booleans, Integers, Strings, and so on—identifying what they are becomes a bit tougher. hapi is based around this concept of configuration objects making for much more readable code.
**hapi plugins**

Through its plugin API, hapi encourages a plugin-centric approach to development both, in the use of third party modules and also for your own features within your application. The plugin API makes it very easy to break an application into smaller plugins so that code is more manageable and readable. This is one of my favorite features of the framework, as it provides a convention for structuring your code in a scalable manner. We'll explore the plugin API in detail in Chapter 3, *Structuring Your Codebase with Plugins*, but I just want to introduce the concept, and show how to register a third party plugin here. Taking the preceding server example, it would be handy to have the server routing table display all routes on startup, and fortunately, there is a plugin for this called blipp.

If you explore hapi's plugins on the hapi plugin page, www.hapijs.com/plugins, you'll notice that a lot of the plugins have silly or non-descriptive names such as poop, joi, inert, h2o2, and so on. This is an attempt at creating competition for plugin implementation, for example joi; a model schema validation library could be named hapi-validator, but that would then be the standard validation library which doesn't encourage competition to build a competing schema validation library. The reason for silly names is an attempt to reduce the seriousness of enterprise development and make it more enjoyable for developers.

So let's look at an example of using a third-party plugin. Adding blipp to our example, we get the following code:

```javascript
const Hapi = require('hapi');
const Blipp = require('blipp');                     // [1]
const server = new Hapi.Server();
server.connection({ port: 1337, host: '127.0.0.1' });
server.route({
  method: 'GET',
  path: '/',
  handler: function (request, reply) {
    return reply('Hello World
');
  }
});
server.register(Blipp, (err) => {                   // [2]
  if(err) {                                         // [3]
    throw err;                                      // [3]
  }                                                 // [3]
  server.start((err) => {
    if(err) {
      throw err;
    }
    console.log(`Server running at ${server.info.uri}`);
  });
});
```

---

[1] Introducing hapi.js
[2] hapi plugins
[3] Chapter 3, Structuring Your Codebase with Plugins

---
With reference to the numbers in the comments in the preceding code example, let's examine each section now:

- [1]: We require the `blipp` module. Don't forget to install blipp through npm (`npm install blipp`) if you're trying this yourself.
- [2]: We register the plugin using `server.register()`. This is an asynchronous function. This is immensely useful for doing operations like initializing database connections on startup. An array of plugins can also be passed in here, which is also very useful for registering multiple plugins at once.
- [3]: We handle the error callback here. If there's any error, we will just throw it, and since it is uncaught, it will end the script execution and send the output to the console, again making it easier to debug what might have caused the error.

If you run this example, you will get the following output, which is the server routing table generated by the `blipp` module:

![Output Image]

Don't worry if this seems complicated now—we are just introducing the concept here. We will explore the `server.register()` API in depth along with creating your own plugins in Chapter 3, Structuring Your Codebase with Plugins; for now it is enough to know that it exists.

It's interesting to note that all of hapi was in 'core' at the beginning, in one repo. But over time, the hapi team broke out what functionality it could into plugins, making the hapi core project quite small, and each separate module much more manageable. All these plugins can be viewed in the hapi.js GitHub organization. They cover everything from payload and query string parsing to route matching, serving static content, and smaller utility modules like blipp in the preceding example. This approach is also very good when building an application—start with things in the core, and push functionality to plugins as the application begins to grow.
hapi configuration

hapi also has a concept of cascading configuration, which it's good to be aware of. That is, configuration of higher-level objects like the server and connection can be overwritten at lower layers such as plugins and routes, where the configuration is only applied at that level, much like how styles in CSS are applied.

A good example of how this is useful is authentication. Where we add and configure authentication, we would want to apply it to all routes within a server, and would do so at the server level. Then for one route to have no authentication required, for example a login page, we would just configure authentication to not be required on a particular route's configuration object. This may sound complex and tough to grasp initially, but as you see more examples of this, it will become clearer. It is enough just to be aware of it for now.

The hapi style guide

I’d like to draw your attention to the style of the code used in the preceding example. It adheres to the hapi style guidelines, and so will all examples in this book, as much as possible. I encourage you now to read the full list of rules on the hapi style guide available at https://github.com/hapijs/contrib/blob/master/Style.md. You may find that you have some disagreements with this and that's fine, but the importance of a style guide is not the individual rules, but that all the code is uniform. This makes spotting bugs and messy code much easier, and leads to clear code throughout the codebase. If you look at the hapi source code or source of hapi modules that adhere to the source, you will see a perfect example of this.

One aspect of the style I’d like to draw attention to is the functions versus fat arrow (=>) usage. If you look back to the previous example, you will see both are used, which might look inconsistent. The pattern used here is that inline callbacks must use arrow functions, while other functions may use either the arrow syntax or the function keyword. I currently use the function keyword anywhere I am not using an inline callback so as to support the use of the this keyword inside the function body.

So now that we know what a hapi server looks like, and what the code should look like in keeping with the style guide, let’s take a look at how to add functionality to this application in the next chapter.
Summary

All code samples seen here as well as some extra material can be found online in the repository available at https://github.com/johnbrett/Getting-Started-with-hapi.js. If you have any questions about the code samples—in trying to understand code snippets or problems running them, feel free to open an issue.

In this chapter, we’ve introduced Node.js, its benefits and differences as compared to other server side technologies, and looked at what a simple Node.js server looks like. We looked at the reason for using a framework on top of this, mentioned some different frameworks, and introduced hapi.js as a choice. We talked at length as to why hapi.js is a good choice, who created it and why, and how its design principles differ from other Node.js frameworks.

Hopefully, at this stage, we’ve convinced you that hapi.js is a great choice for building web applications, and is something you want to learn about. Finally, we saw what a simple server looks like in hapi.js, using its style guide, and gave a quick overview of its excellent ecosystem and community. Next, let’s look at adding functionality to our examples, and at creating a trivial applications and websites using the hapi.js framework and ecosystem.
Where to buy this book
You can buy Getting Started with hapi.js 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.