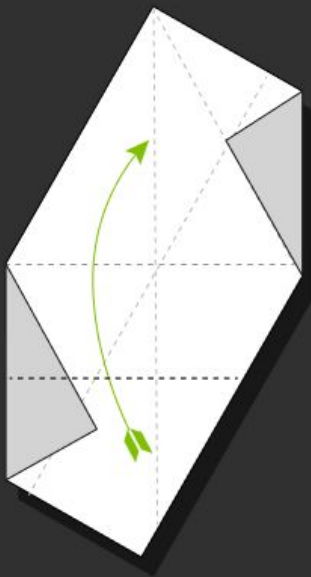
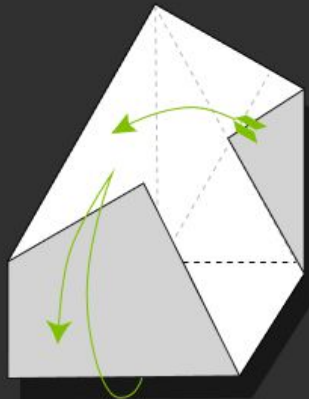


# NODEJS

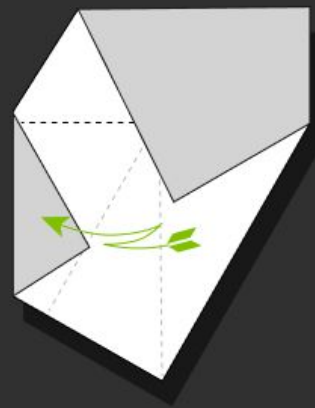
## RELATED TOOLS & SKILLS



4.



5.



6.

LEVEL UP YOUR NODE KNOWLEDGE

# Node.js: Related Tools & Skills

Copyright © 2018 SitePoint Pty. Ltd.

**Cover Design:** Alex Walker

## Notice of Rights

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 embodied in critical articles or reviews.

## Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

## Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood  
VIC Australia 3066  
Web: [www.sitepoint.com](http://www.sitepoint.com)  
Email: [books@sitepoint.com](mailto:books@sitepoint.com)

## About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, design, and more.

# Preface

While there have been quite a few attempts to get JavaScript working as a server-side language, Node.js (frequently just called Node) has been the first environment that's gained any traction. It's now used by companies such as Netflix, Uber and Paypal to power their web apps. Node allows for blazingly fast performance; thanks to its event loop model, common tasks like network connection and database I/O can be executed very quickly indeed.

From a beginner's point of view, one of Node's obvious advantages is that it uses JavaScript, a ubiquitous language that many developers are comfortable with. If you can write JavaScript for the client-side, writing server-side applications with Node should not be too much of a stretch for you.

In this book, we'll take a look at a selection of the related tools and skills that will make you a much more productive Node developer.

## Who Should Read This Book?

This book is for anyone who wants to start learning server-side development with Node.js. Familiarity with JavaScript is assumed, but we don't assume any previous back-end development experience.

## Conventions Used

### CODE SAMPLES

Code in this book is displayed using a fixed-width font, like so:

```
<h1>A Perfect Summer's Day</h1>
<p>It was a lovely day for a walk in the park.
The birds were singing and the kids were all back at school.</p>
```

Where existing code is required for context, rather than repeat all of it, `:` will be displayed:

```
function animate() {  
    :  
    new_variable = "Hello";  
}
```

Some lines of code should be entered on one line, but we've had to wrap them because of page constraints. An `↵` indicates a line break that exists for formatting purposes only, and should be ignored:

```
URL.open("http://www.sitepoint.com/responsive-web-  
↵design-real-user-testing/?responsive1");
```

You'll notice that we've used certain layout styles throughout this book to signify different types of information. Look out for the following items.

## TIPS, NOTES, AND WARNINGS

### Hey, You!

Tips provide helpful little pointers.

### Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.

### Make Sure You Always ...

... pay attention to these important points.

---

### **Watch Out!**

Warnings highlight any gotchas that are likely to trip you up along the way.

---

# Chapter 1: Unit Test Your JavaScript Using Mocha and Chai

**BY JANI HARTIKAINEN**

Have you ever made some changes to your code, and later found it caused something else to break?

I'm sure most of us have. This is almost inevitable, especially when you have a larger amount of code. One thing depends on another, and then changing it breaks something else as a result.

But what if that didn't happen? What if you had a way of knowing when something breaks as a result of some change? That would be pretty great. You could modify your code without having to worry about breaking anything, you'd have fewer bugs and you'd spend less time debugging.

That's where unit tests shine. They will *automatically* detect any problems in the code for you. Make a change, run your tests and if anything breaks, you'll immediately know what happened, where the problem is *and* what the correct behavior should be. This completely eliminates any guesswork!

In this article, I'll show you how to get started unit testing your JavaScript code. The examples and techniques shown in this article can be applied to both browser-based code and Node.js code.

The code for this tutorial is available from our [GitHub repo](#).

## What Is Unit Testing

When you test your codebase, you take a piece of code — typically a function — and verify it behaves correctly in a specific situation. Unit testing is a structured and automated way of doing this. As a result, the more tests you write, the bigger the benefit

you receive. You will also have a greater level of confidence in your codebase as you continue to develop it.

The core idea with unit testing is to test a function's behavior when giving it a certain set of inputs. You call a function with certain parameters, and check you got the correct result.

```
// Given 1 and 10 as inputs...
var result = Math.max(1, 10);

// ...we should receive 10 as the output
if(result !== 10) {
  throw new Error('Failed');
}
```

In practice, tests can sometimes be more complex. For example, if your function makes an Ajax request, the test needs some more set up, but the same principle of "given certain inputs, we expect a specific outcome" still applies.

## Setting up the Tools

For this article, we'll be using Mocha. It's easy to get started with, can be used for both browser-based testing and Node.js testing, and it plays nicely with other testing tools.

The easiest way to install Mocha is through npm (for which we also need to install Node.js). If you're unsure about how to install either npm or Node on your system, consult our tutorial: A Beginner's Guide to npm — the Node Package Manager

With Node installed, open up a terminal or command line in your project's directory.

- If you want to test code in the browser, run `npm install mocha chai --save-dev`
- If you want to test Node.js code, in addition to the above, run `npm install -g mocha`

This installs the packages `mocha` and `chai`. Mocha is the library that allows us to run tests, and Chai contains some helpful functions that we'll use to verify our test results.

## TESTING ON NODE.JS VS TESTING IN THE BROWSER

The examples that follow are designed to work if running the tests in a browser. If you



want to unit test your Node.js application, follow these steps.

- For Node, you don't need the test runner file.
- To include Chai, add `var chai = require('chai');` at the top of the test file.
- Run the tests using the `mocha` command, instead of opening a browser.

## Setting up a Directory Structure

You should put your tests in a separate directory from your main code files. This makes it easier to structure them, for example if you want to add other types of tests in the future (such as integration tests or functional tests).

The most popular practice with JavaScript code is to have a directory called `test/` in your project's root directory. Then, each test file is placed under `test/someModuleTest.js`. Optionally, you can also use directories inside `test/`, but I recommend keeping things simple — you can always change it later if necessary.

## Setting up a Test Runner

In order to run our tests in a browser, we need to set up a simple HTML page to be our *test runner* page. The page loads Mocha, the testing libraries and our actual test files. To run the tests, we'll simply open the runner in a browser.

If you're using Node.js, you can skip this step. Node.js unit tests can be run using the command `mocha`, assuming you've followed the recommended directory structure.

Below is the code we'll use for the test runner. I'll save this file as `testrunner.html`.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mocha Tests</title>
    <link rel="stylesheet" href="node_modules/mocha/mocha.css">
  </head>
  <body>
    <div id="mocha"></div>
    <script src="node_modules/mocha/mocha.js"></script>
    <script src="node_modules/chai/chai.js"></script>
    <script>mocha.setup('bdd')</script>

    <!-- load code you want to test here -->

    <!-- load your test files here -->
```

```
<script>
  mocha.run();
</script>
</body>
</html>
```

The important bits in the test runner are:

- We load Mocha's CSS styles to give our test results nice formatting.
- We create a div with the ID `mocha`. This is where the test results are inserted.
- We load Mocha and Chai. They are located in subfolders of the `node_modules` folder since we installed them via npm.
- By calling `mocha.setup`, we make Mocha's testing helpers available.
- Then, we load the code we want to test and the test files. We don't have anything here just yet.
- Last, we call `mocha.run` to run the tests. Make sure you call this *after* loading the source and test files.

## The Basic Test Building Blocks

Now that we can run tests, let's start writing some.

We'll begin by creating a new file `test/arrayTest.js`. An individual test file such as this one is known as a *test case*. I'm calling it `arrayTest.js` because for this example, we'll be testing some basic array functionality.

Every test case file follows the same basic pattern. First, you have a `describe` block:

```
describe('Array', function() {
  // Further code for tests goes here
});
```

`describe` is used to group individual tests. The first parameter should indicate what we're testing — in this case, since we're going to test array functions, I've passed in the string `'Array'`.

Secondly, inside the `describe`, we'll have `it` blocks:

```
describe('Array', function() {
  it('should start empty', function() {
    // Test implementation goes here
  });

  // We can have more its here
});
```

`it` is used to create the actual tests. The first parameter to `it` should provide a human-readable description of the test. For example, we can read the above as "it should start empty", which is a good description of how arrays should behave. The code to implement the test is then written inside the function passed to `it`.

All Mocha tests are built from these same building blocks, and they follow this same basic pattern.

- First, we use `describe` to say what we're testing - for example, "describe how array should work".
- Then, we use a number of `it` functions to create the individual tests - each `it` should explain one specific behavior, such as "it should start empty" for our array case above.

## Writing the Test Code

Now that we know how to structure the test case, let's jump into the fun part — implementing the test.

Since we are testing that an array should start empty, we need to create an array and then ensure it's empty. The implementation for this test is quite simple:

```
var assert = chai.assert;

describe('Array', function() {
  it('should start empty', function() {
    var arr = [];

    assert.equal(arr.length, 0);
  });
});
```

Note on the first line, we set up the `assert` variable. This is just so we don't need to keep typing `chai.assert` everywhere.

In the `it` function, we create an array and check its length. Although simple, this is a good example of how tests work.

First, you have something you're testing — this is called the *System Under Test* or *SUT*. Then, if necessary, you do something with the SUT. In this test, we're not doing anything, since we're checking the array starts as empty.

The last thing in a test should be the validation — an *assertion* which checks the result. Here, we are using `assert.equal` to do this. Most assertion functions take parameters in the same order: First the "actual" value, and then the "expected" value.

- The *actual* value is the result from your test code, so in this case `arr.length`
- The *expected* value is what the result *should* be. Since an array should begin empty, the expected value in this test is `0`

Chai also offers two different styles of writing assertions, but we're using `assert` to keep things simple for now. When you become more experienced with writing tests, you might want to use the `expect` assertions instead, as they provide some more flexibility.

## Running the Test

In order to run this test, we need to add it to the test runner file we created earlier.

If you're using Node.js, you can skip this step, and use the command `mocha` to run the test. You'll see the test results in the terminal.

Otherwise, to add this test to the runner, simply add:

```
<script src="test/arrayTest.js"></script>
```

Below:

```
<!-- load your test files here -->
```

Once you've added the script, you can then load the test runner page in your browser of choice.

## The Test Results

When you run your tests, the test results will look something like this:



Note that what we entered into the `describe` and `it` functions show up in the output — the tests are grouped under the description. Note that it's also possible to nest `describe` blocks to create further sub-groupings.

Let's take a look at what a failing test looks like.

On the line in the test that says:

```
assert.equal(arr.length, 0);
```

Replace the number 0 with 1. This makes the test fail, as the array's length no longer matches the expected value.

If you run the tests again, you'll see the failing test in red with a description of what went wrong.



One of the benefits of tests is that they help you find bugs quicker, however this error is not very helpful in that respect. We can fix it though.

Most of the assertion functions can also take an optional `message` parameter. This is the message that is displayed when the assertion fails. It's a good idea to use this parameter to make the error message easier to understand.

We can add a message to our assertion like so:

```
assert.equal(arr.length, 1, 'Array length was not 0');
```

If you re-run tests, the custom message will appear instead of the default.

Let's switch the assertion back to the way it was — replace `1` with `0`, and run the tests again to make sure they pass.

## Putting It Together

So far we've looked at fairly simple examples. Let's put what we've learned into practice and see how we would test a more realistic piece of code.

Here's a function which adds a CSS class to an element. This should go in a new file `js/className.js`.

```
function addClass(el, newClass) {  
  if(el.className.indexOf(newClass) === -1) {  
    el.className += newClass;  
  }  
}
```

To make it a bit more interesting, I made it add a new class only when that class doesn't exist in an element's `className` property — who wants to see `<div class="hello hello hello hello">` after all?

In the best case, we would write tests for this function *before* we write the code. But test-driven development is a complex topic, and for now we just want to focus on writing tests.

To get started, let's recall the basic idea behind unit tests: We give the function certain inputs and then verify the function behaves as expected. So what are the inputs and behaviors for this function?

Given an element and a class name:

- if the element's `className` property does not contain the class name, it should be added.
- if the element's `className` property does contain the class name, it should not be added.

Let's translate these cases into two tests. In the `test` directory, create a new file `classNameTest.js` and add the following:

```
describe('addClass', function() {  
  it('should add class to element');  
  it('should not add a class which already exists');  
});
```

We changed the wording slightly to the "it should do X" form used with tests. This means that it reads a bit nicer, but is essentially still the same human-readable form we listed above. It's usually not much more difficult than this to go from idea to test.

But wait, where are the test functions? Well, when we omit the second parameter to `it`, Mocha marks these tests as *pending* in the test results. This is a convenient way to set up a number of tests — kind of like a todo list of what you intend to write.

Let's continue by implementing the first test.

```
describe('addClass', function() {  
  it('should add class to element', function() {  
    var element = { className: '' };  
  
    addClass(element, 'test-class');  
  
    assert.equal(element.className, 'test-class');  
  });  
  
  it('should not add a class which already exists');  
});
```

In this test, we create an `element` variable and pass it as a parameter to the `addClass` function, along with a string `test-class` (the new class to add). Then, we check the class is included in the value using an assertion.

Again, we went from our initial idea — given an element and a class name, it should be added into the class list — and translated it into code in a fairly straightforward manner.

Although this function is designed to work with DOM elements, we're using a plain JS object here. Sometimes we can make use of JavaScript's dynamic nature in this fashion to simplify our tests. If we didn't do this, we would need to create an actual element and it would complicate our test code. As an additional benefit, since we don't use DOM, we can also run this test within Node.js if we so wish.

## RUNNING THE TESTS IN THE BROWSER

To run the test in the browser, you'll need to add `className.js` and `classNameTest.js` to the runner:

```
<!-- load code you want to test here -->
<script src="js/className.js"></script>

<!-- load your test files here -->
<script src="test/classNameTest.js"></script>
```

You should now see one test pass and another test show up as pending, as is demonstrated by the following CodePen. Note that the code differs slightly from the example in order to make the code work within the CodePen environment.

### Live Code!

See the Pen [Unit Testing with Mocha \(1\)](#),

Next, let's implement the second test...

```
it('should not add a class which already exists', function() {
  var element = { className: 'exists' };

  addClass(element, 'exists');

  var numClasses = element.className.split(' ').length;
  assert.equal(numClasses, 1);
});
```

It's a good habit to run your tests often, so let's check what happens if we run the tests now. As expected, they should pass.

Here's another CodePen with the second test implemented.



## Live Code!

See the Pen [Unit Testing with Mocha \(2\)](#),

But hang on! I actually tricked you a bit. There is a third behavior for this function which we haven't considered. There is also a bug in the function — a fairly serious one. It's only a three line function but did you notice it?

Let's write one more test for the third behavior which exposes the bug as a bonus.

```
it('should append new class after existing one', function() {
  var element = { className: 'exists' };

  addClass(element, 'new-class');

  var classes = element.className.split(' ');
  assert.equal(classes[1], 'new-class');
});
```

This time the test fails. You can see it in action in the following CodePen. The problem here is simple: CSS class names in elements should be separated by a space. However, our current implementation of `addClass` doesn't add a space!

## Live Code!

See the Pen [Unit Testing with Mocha \(3\)](#).

Let's fix the function and make the test pass.

```
function addClass(el, newClass) {
  if(el.className.indexOf(newClass) !== -1) {
    return;
  }

  if(el.className !== '') {
    //ensure class names are separated by a space
    newClass = ' ' + newClass;
  }
}
```

```
    el.className += newClass;
}
```

And here's a final CodePen with the fixed function and passing tests.

## Live Code!

See the Pen [Unit Testing with Mocha \(4\)](#).

## RUNNING THE TESTS ON NODE

In Node, things are only visible to other things in the same file. As `className.js` and `classNameTest.js` are in different files, we need to find a way to expose one to the other. The standard way to do this is through the use of `module.exports`. If you need a refresher, you can read all about that here: [Understanding module.exports and exports in Node.js](#)

The code essentially stays the same, but is structured slightly differently:

```
// className.js

module.exports = {
  addClass: function(el, newClass) {
    if(el.className.indexOf(newClass) !== -1) {
      return;
    }

    if(el.className !== '') {
      //ensure class names are separated by a space
      newClass = ' ' + newClass;
    }

    el.className += newClass;
  }
}
```

```
// classNameTest.js

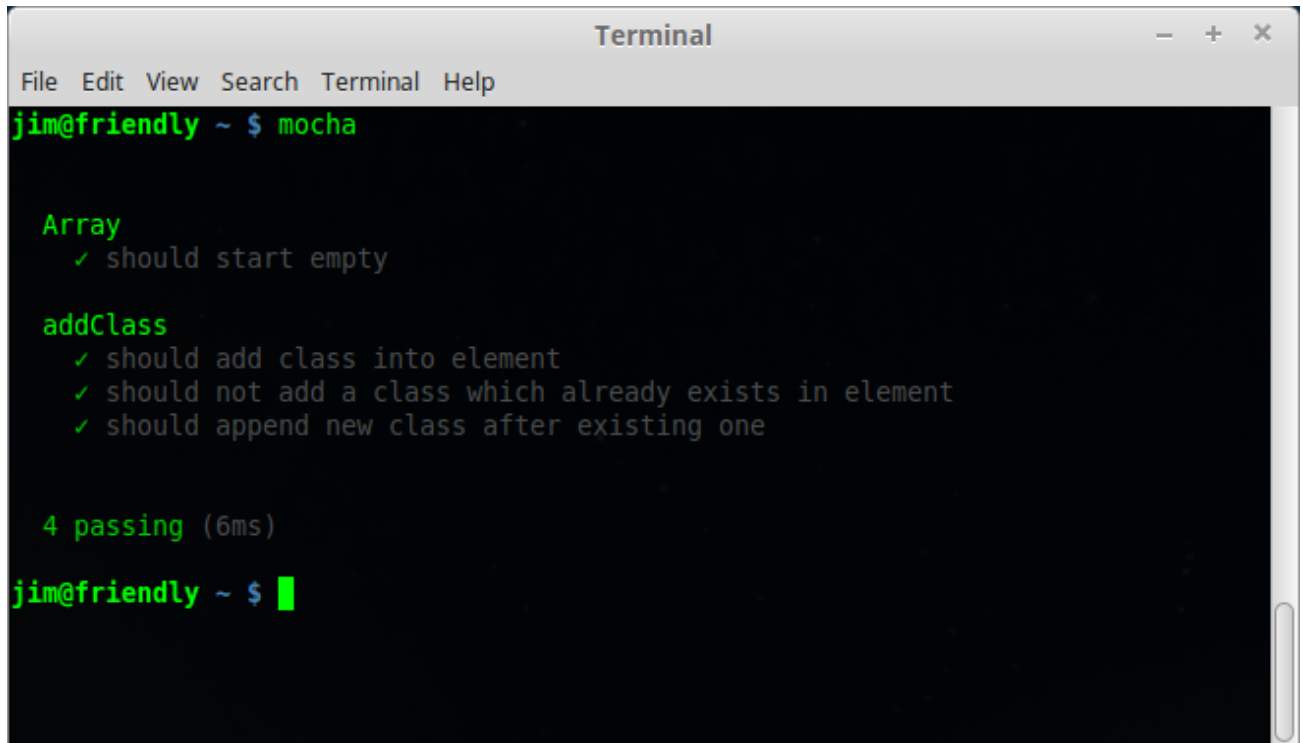
var chai = require('chai');
var assert = chai.assert;

var className = require('../js/className.js');
var addClass = className.addClass;

// The rest of the file remains the same
```

```
describe('addClass', function() {  
    ...  
});
```

And as you can see, the tests pass.

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The prompt is "jim@friendly ~ \$". The command "mocha" has been executed. The output shows two test suites: "Array" with one passing test "should start empty", and "addClass" with three passing tests: "should add class into element", "should not add a class which already exists in element", and "should append new class after existing one". The summary shows "4 passing (6ms)". The prompt "jim@friendly ~ \$" is followed by a green cursor.

```
Terminal  
File Edit View Search Terminal Help  
jim@friendly ~ $ mocha  
  
Array  
  ✓ should start empty  
  
addClass  
  ✓ should add class into element  
  ✓ should not add a class which already exists in element  
  ✓ should append new class after existing one  
  
4 passing (6ms)  
jim@friendly ~ $
```

## What's Next?

As you can see, testing does not have to be complicated or difficult. Just as with other aspects of writing JavaScript apps, you have some basic patterns which repeat. Once you get familiar with those, you can keep using them again and again.

But this is just scratching the surface. There's a lot more to learn about unit testing.

- Testing more complex systems
- How to deal with Ajax, databases, and other "external" things?
- Test-Driven Development

If you want to continue learning this and more, I've created a [free JavaScript unit testing quickstart series](#). If you found this article useful, you should definitely [check it out here](#).

Alternatively, if video is more your style, you might be interested in SitePoint Premium's course: [Test-Driven Development in Node.js](#).

**WOW! eBook**  
[www.wowebook.org](http://www.wowebook.org)

# Chapter 2: An Introduction to Functional JavaScript

**BY M. DAVID GREEN**

You've heard that JavaScript is a functional language, or at least that it's capable of supporting functional programming. But what is functional programming? And for that matter, if you're going to start comparing programming paradigms in general, how is a functional approach different from the JavaScript that you've always written?

Well, the good news is that JavaScript isn't picky when it comes to paradigms. You can mix your imperative, object-oriented, prototypal, and functional code as you see fit, and still get the job done. But the bad news is what that means for your code. JavaScript can support a wide range of programming styles simultaneously within the same codebase, so it's up to you to make the right choices for maintainability, readability, and performance.

Functional JavaScript doesn't have to take over an entire project in order to add value. Learning a little about the functional approach can help guide some of the decisions you make as you build your projects, regardless of the way you prefer to structure your code. Learning some functional patterns and techniques can put you well on your way to writing cleaner and more elegant JavaScript regardless of your preferred approach.

## Imperative JavaScript

JavaScript first gained popularity as an in-browser language, used primarily for adding simple hover and click effects to elements on a web page. For years, that's most of what people knew about it, and that contributed to the bad reputation JavaScript earned early on.

As developers struggled to match the flexibility of JavaScript against the intricacy of the browser document object model (DOM), actual JavaScript code often looked something like this in the real world:

```
var result;
function getText() {
    var someText = prompt("Give me something to capitalize");
    capWords(someText);
    alert(result.join(" "));
};
function capWords(input) {
    var counter;
    var inputArray = input.split(" ");
    var transformed = "";
    result = [];
    for (counter = 0; counter < inputArray.length; counter++) {
        transformed = [
            inputArray[counter].charAt(0).toUpperCase(),
            inputArray[counter].substring(1)
        ].join("");
        result.push(transformed);
    }
};
document.getElementById("main_button").onclick = getText;
```

So many things are going on in this little snippet of code. Variables are being defined on the global scope. Values are being passed around and modified by functions. DOM methods are being mixed with native JavaScript. The function names are not very descriptive, and that's due in part to the fact that the whole thing relies on a context that may or may not exist. But if you happened to run this in a browser inside an HTML document that defined a `<button id="main_button">`, you might get prompted for some text to work with, and then see the an `alert` with first letter of each of the words in that text capitalized.

Imperative code like this is written to be read and executed from top to bottom (give or take a little variable hoisting). But there are some improvements we could make to clean it up and make it more readable by taking advantage of JavaScript's object-oriented nature.

## Object-Oriented JavaScript

After a few years, developers started to notice the problems with imperative coding in a shared environment like the browser. Global variables from one snippet of JavaScript clobbered global variables set by another. The order in which the code was called affected the results in ways that could be unpredictable, especially given the delays introduced by network connections and rendering times.

Eventually, some better practices emerged to help encapsulate JavaScript code and make it play better with the DOM. An updated variation of the same code above, written to an object-oriented standard, might look something like this:

```
(function() {
  "use strict";
  var SomeText = function(text) {
    this.text = text;
  };
  SomeText.prototype.capitalize = function(str) {
    var firstLetter = str.charAt(0);
    var remainder = str.substring(1);
    return [firstLetter.toUpperCase(), remainder].join("");
  };
  SomeText.prototype.capitalizeWords = function() {
    var result = [];
    var textArray = this.text.split(" ");
    for (var counter = 0; counter < textArray.length; counter++) {
      result.push(this.capitalize(textArray[counter]));
    }
    return result.join(" ");
  };

  document.getElementById("main_button").addEventListener("click", function() {
    var something = prompt("Give me something to capitalize");
    var newText = new SomeText(something);
    alert(newText.capitalizeWords());
  });
})();
```

In this object-oriented version, the constructor function simulates a class to model the object we want. Methods live on the new object's prototype to keep memory use low. And all of the code is isolated in an anonymous immediately-invoked function expression so it doesn't litter the global scope. There's even a "use strict" directive to take advantage of the latest JavaScript engine, and the old-fashioned `onclick` method has been replaced with a shiny new `addEventListener`, because who uses IE8 or earlier anymore? A script like this would likely be inserted at the end of the `<body>` element on an HTML document, to make sure all the DOM had been loaded before it was processed so the `<button>` it relies on would be available.

But despite all this reconfiguration, there are still many artifacts of the same imperative style that led us here. The methods in the constructor function rely on variables that are scoped to the parent object. There's a looping construct for iterating across all the members of the array of strings. There's a `counter` variable that serves no purpose

other than to increment the progress through the `for` loop. And there are methods that produce the side effect of modifying variables that exist outside of their own definitions. All of this makes the code more brittle, less portable, and makes it harder to test the methods outside of this narrow context.

## Functional JavaScript

The object-oriented approach is much cleaner and more modular than the imperative approach we started with, but let's see if we can improve it by addressing some of the drawbacks we discussed. It would be great if we could find ways to take advantage of JavaScript's built-in ability to treat functions as first-class objects so that our code could be cleaner, more stable, and easier to repurpose.

```
(function() {  
    "use strict";  
    var capify = function(str) {  
        return [str.charAt(0).toUpperCase(), str.substring(1)].join("");  
    };  
    var processWords = function(fn, str) {  
        return str.split(" ").map(fn).join(" ");  
    };  
    document.getElementById("main_button").addEventListener("click", function() {  
        var something = prompt("Give me something to capitalize");  
        alert(processWords(capify, something));  
    });  
})();
```

Did you notice how much shorter this version is? We're only defining two functions: `capify` and `processWords`. Each of these functions is pure, meaning that they don't rely on the state of the code they're called from. The functions don't create side effects that alter variables outside of themselves. There is one and only one result a function returns for any given set of arguments. Because of these improvements, the new functions are very easy to test, and could be snipped right out of this code and used elsewhere without any modifications.

There might have been one keyword in there that you wouldn't recognize unless you've peeked at some functional code before. We took advantage of the new `map` method on `Array` to apply a function to each element of the temporary array we created when we split our string. `Map` is just one of a handful of convenience methods we were given when modern browsers and server-side JavaScript interpreters implemented the ECMAScript 5 standards. Just using `map` here, in place of a `for` loop, eliminated the

`counter` variable and helped make our code much cleaner and easier to read.

## Start Thinking Functionally

You don't have to abandon everything you know to take advantage of the functional paradigm. You can get started thinking about your JavaScript in a functional way by considering a few questions when you write your next program:

- Are my functions dependent on the context in which they are called, or are they pure and independent?
- Can I write these functions in such a way that I could depend on them always returning the same result for a given input?
- Am I sure that my functions don't modify anything outside of themselves?
- If I wanted to use these functions in another program, would I need to make changes to them?

This introduction barely scratches the surface of functional JavaScript, but I hope it whets your appetite to learn more.



# Chapter 3: An Introduction to Gulp.js

**BY CRAIG BUCKLER**

**Developers spend precious little time coding. Even if we ignore irritating meetings, much of the job involves basic tasks which can sap your working day:**

- generating HTML from templates and content files
- compressing new and modified images
- compiling Sass to CSS code
- removing `console` and `debugger` statements from scripts
- transpiling ES6 to cross-browser-compatible ES5 code
- code linting and validation
- concatenating and minifying CSS and JavaScript files
- deploying files to development, staging and production servers.

Tasks must be repeated every time you make a change. You may start with good intentions, but the most infallible developer will forget to compress an image or two. Over time, pre-production tasks become increasingly arduous and time-consuming; you'll dread the inevitable content and template changes. It's mind-numbing, repetitive work. Wouldn't it be better to spend your time on more profitable jobs?

If so, you need a *task runner* or *build process*.

## That Sounds Scarily Complicated!

Creating a build process will take time. It's more complex than performing each task manually, but over the long term, you'll save hours of effort, reduce human error and save your sanity. Adopt a pragmatic approach:

- Automate the most frustrating tasks first.
- Try not to over-complicate your build process. An hour or two is more than enough for the initial setup.
- Choose task runner software and stick with it for a while. Don't switch to another option on a whim.

Some of the tools and concepts may be new to you, but take a deep breath and concentrate on one thing at a time.

## Task Runners: the Options

Build tools such as GNU Make have been available for decades, but web-specific task runners are a relatively new phenomenon. The first to achieve critical mass was Grunt — a Node.js task runner which used plugins controlled (originally) by a JSON configuration file. Grunt was hugely successful, but there were a number of issues:

1. Grunt required plugins for basic functionality such as file watching.
2. Grunt plugins often performed multiple tasks, which made customisation more awkward.
3. JSON configuration could become unwieldy for all but the most basic tasks.
4. Tasks could run slowly because Grunt saved files between every processing step.

Many issues were addressed in later editions, but Gulp had already arrived and offered a number of improvements:

1. Features such as file watching were built in.
2. Gulp plugins were (*mostly*) designed to do a single job.
3. Gulp used JavaScript configuration code that was less verbose, easier to read, simpler to modify, and provided better flexibility.
4. Gulp was faster because it uses Node.js streams to pass data through a series of piped plugins. Files were only written at the end of the task.

Of course, Gulp itself isn't perfect, and new task runners such as Broccoli.js, Brunch and webpack have also been competing for developer attention. More recently, npm itself has been touted as a simpler option. All have their pros and cons, but Gulp remains the favorite and is currently used by more than 40% of web developers.

Gulp requires Node.js, but while some JavaScript knowledge is beneficial, developers

from all web programming faithfuls will find it useful.

## What About Gulp 4?

This tutorial describes how to use Gulp 3 — the most recent release version at the time of writing. Gulp 4 has been in development for some time but remains a beta product. It's possible to use or switch to Gulp 4, but I recommend sticking with version 3 until the final release.

## Step 1: Install Node.js

Node.js can be downloaded for Windows, macOS and Linux from [nodejs.org/download/](https://nodejs.org/download/). There are various options for installing from binaries, package managers and docker images, and full instructions are available.

### For Windows Users

Node.js and Gulp run on Windows, but some plugins may not install or run if they depend on native Linux binaries such as image compression libraries. One option for Windows 10 users is the new bash command-line, which solves many issues.

Once installed, open a command prompt and enter:

```
node -v
```

This reveals the version number. You're about to make heavy use of `npm` — the Node.js package manager which is used to install modules. Examine its version number:

```
npm -v
```

### For Linux Users

Node.js modules can be installed globally so they're available throughout your system. However, most users will not have permission to write to the

global directories unless `npm` commands are prefixed with `sudo`. There are a number of options to fix `npm` permissions and tools such as `nvm` can help, but I often change the default directory. For example, on Ubuntu/Debian-based platforms:

```
cd ~  
  
mkdir .node_modules_global  
npm config set prefix=$HOME/.node_modules_global  
npm install npm -g
```

Then add the following line to the end of `~/.bashrc`:

```
export PATH="$HOME/.node_modules_global/bin:$PATH"
```

Finally, update with this:

```
source ~/.bashrc
```

## Step 2: Install Gulp Globally

Install Gulp command-line interface globally so the `gulp` command can be run from any project folder:

```
npm install gulp-cli -g
```

Verify Gulp has installed with this:

```
gulp -v
```

## Step 3: Configure Your Project

**For Node.js Projects**

WOW! eBook  
[www.wowebook.org](http://www.wowebook.org)

You can skip this step if you already have a `package.json` configuration file.

*Note for Node.js projects: you can skip this step if you already have a `package.json` configuration file.*

Presume you have a new or pre-existing project in the folder `project1`. Navigate to this folder and initialize it with `npm`:

```
cd project1
npm init
```

You'll be asked a series of questions. Enter a value or hit **Return** to accept defaults. A `package.json` file will be created on completion which stores your `npm` configuration settings.

### Git Users

Node.js installs modules to a `node_modules` folder. You should add this to your `.gitignore` file to ensure they're not committed to your repository. When deploying the project to another PC, you can run `npm install` to restore them.

For the remainder of this article, we'll presume your project folder contains the following sub-folders:

`src` folder: preprocessed source files

This contains further sub-folders:

- `html` - HTML source files and templates
- `images` — the original uncompressed images
- `js` — multiple preprocessed script files
- `scss` — multiple preprocessed Sass `.scss` files

## build folder: compiled/processed files

Gulp will create files and create sub-folders as necessary:

- `html` — compiled static HTML files
- `images` — compressed images
- `js` — a single concatenated and minified JavaScript file
- `css` — a single compiled and minified CSS file

Your project will almost certainly be different but this structure is used for the examples below.

### Following Along on Unix

Tip: If you're on a Unix-based system and you just want to follow along with the tutorial, you can recreate the folder structure with the following command:

```
mkdir -p src/{html,images,js,scss} build/{html,images,js,c
```

*Tip: If you're on a Unix-based system and you just want to follow along with the tutorial, you can recreate the folder structure with the following command:*

```
mkdir -p src/{html,images,js,scss} build/{html,images,js,css}
```

## Step 4: Install Gulp Locally

You can now install Gulp in your project folder using the command:

```
npm install gulp --save-dev
```

This installs Gulp as a development dependency and the "devDependencies" section of `package.json` is updated accordingly. We'll presume Gulp and all plugins are

development dependencies for the remainder of this tutorial.

## ALTERNATIVE DEPLOYMENT OPTIONS

Development dependencies are not installed when the `NODE_ENV` environment variable is set to `production` on your operating system. You would normally do this on your live server with the Mac/Linux command:

```
export NODE_ENV=production
```

Or on Windows:

```
set NODE_ENV=production
```

This tutorial presumes your assets will be compiled to the `build` folder and committed to your Git repository or uploaded directly to the server. However, it may be preferable to build assets on the live server if you want to change the way they are created. For example, HTML, CSS and JavaScript files are minified on production but not development environments. In that case, use the `--save` option for Gulp and all plugins, i.e.

```
npm install gulp --save
```

This sets Gulp as an application dependency in the "dependencies" section of `package.json`. It will be installed when you enter `npm install` and can be run wherever the project is deployed. You can remove the `build` folder from your repository since the files can be created on any platform when required.

## Step 4: Create a Gulp Configuration File

Create a new `gulpfile.js` configuration file in the root of your project folder. Add some basic code to get started:

```
// Gulp.js configuration
var
  // modules
  gulp = require('gulp'),

  // development mode?
  devBuild = (process.env.NODE_ENV !== 'production'),
```

```
// folders
folder = {
  src: 'src/',
  build: 'build/'
}
;
```

This references the Gulp module, sets a `devBuild` variable to `true` when running in development (or non-production mode) and defines the source and build folder locations.

## ES6

ES5-compatible JavaScript code is provided in this tutorial. This will work for all versions of Gulp and Node.js with or without the `--harmony` flag. Most ES6 features are supported in Node 6 and above so feel free to use arrow functions, `let`, `const`, etc. if you're using a recent version.

`gulpfile.js` won't do anything yet because you need to ...

## Step 5: Create Gulp Tasks

On its own, Gulp does nothing. You must:

1. install Gulp plugins, and
2. write tasks which utilize those plugins to do something useful.

It's possible to write your own plugins but, since almost 3,000 are available, it's unlikely you'll ever need to. You can search using Gulp's own directory at [gulpjs.com/plugins/](http://gulpjs.com/plugins/), on [npmjs.com](http://npmjs.com), or search "*gulp something*" to harness the mighty power of Google.

Gulp provides three primary task methods:

- `gulp.task` — defines a new task with a name, optional array of dependencies and a function.
- `gulp.src` — sets the folder where source files are located.



- `gulp.dest` — sets the destination folder where build files will be placed.

Any number of plugin calls are set with `pipe` between the `.src` and `.dest`.

## IMAGE TASK

This is best demonstrated with an example, so let's create a basic task which compresses images and copies them to the appropriate `build` folder. Since this process could take time, we'll only compress new and modified files. Two plugins can help us: `gulp-newer` and `gulp-imagemin`. Install them from the command-line:

```
npm install gulp-newer gulp-imagemin --save-dev
```

We can now reference both modules the top of `gulpfile.js`:

```
// Gulp.js configuration

var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
```

We can now define the image processing task itself as a function at the end of `gulpfile.js`:

```
// image processing
gulp.task('images', function() {
  var out = folder.build + 'images/';
  return gulp.src(folder.src + 'images/**/*')
    .pipe(newer(out))
    .pipe(imagemin({ optimizationLevel: 5 }))
    .pipe(gulp.dest(out));
});
```

All tasks are syntactically similar. This code:

1. Creates a new task named `images`.
2. Defines a function with a return value which ...
3. Defines an `out` folder where build files will be located.
4. Sets the Gulp `src` source folder. The `/**/*` ensures that images in sub-folders are also processed.

5. Pipes all files to the `gulp-newer` module. Source files that are newer than corresponding destination files are passed through. Everything else is removed.
6. The remaining new or changed files are piped through `gulp-imagemin` which sets an optional `optimizationLevel` argument.
7. The compressed images are output to the Gulp `dest` folder set by `out`.

Save `gulpfile.js` and place a few images in your project's `src/images` folder before running the task from the command line:

```
gulp images
```

All images are compressed accordingly and you'll see output such as:

```
Using file gulpfile.js
Running 'imagemin'...
Finished 'imagemin' in 5.71 ms
gulp-imagemin: image1.png (saved 48.7 kB)
gulp-imagemin: image2.jpg (saved 36.2 kB)
gulp-imagemin: image3.svg (saved 12.8 kB)
```

Try running `gulp images` again and nothing should happen because no newer images exist.

## HTML TASK

We can now create a similar task which copies files from the source HTML folder. We can safely minify our HTML code to remove unnecessary whitespace and attributes using the `gulp-htmlclean` plugin:

```
npm install gulp-htmlclean --save-dev
```

This is then referenced at the top of `gulpfile.js`:

```
var
  // modules
  gulp = require('gulp'),
  newer = require('gulp-newer'),
  imagemin = require('gulp-imagemin'),
  htmlclean = require('gulp-htmlclean'),
```

We can now create an `html` task at the end of `gulpfile.js`:

```
// HTML processing
gulp.task('html', ['images'], function() {
  var
    out = folder.build + 'html/',
    page = gulp.src(folder.src + 'html/**/*.*)
      .pipe(newer(out));

  // minify production code
  if (!devBuild) {
    page = page.pipe(htmlclean());
  }

  return page.pipe(gulp.dest(out));
});
```

This reuses `gulp-newer` and introduces a couple of concepts:

1. The `['images']` argument states that our `images` task must be run before processing the HTML (the HTML is likely to reference images). Any number of dependent tasks can be listed in this array and all will complete before the task function runs.
2. We only pipe the HTML through `gulp-htmlclean` if `NODE_ENV` is set to `production`. Therefore, the HTML remains uncompressed during development which may be useful for debugging.

Save `gulpfile.js` and run `gulp html` from the command line. Both the `html` and `images` tasks will run.

## JAVASCRIPT TASK

Too easy for you? Let's process all our JavaScript files by building a basic module bundler. It will:

1. Ensure dependencies are loaded first using the `gulp-deporder` plugin. This analyses comments at the top of each script to ensure correct ordering. For example, `// requires: defaults.js lib.js`.
2. Concatenate all script files into a single `main.js` file using `gulp-concat`.
3. Remove all `console` and debugging statements with `gulp-strip-debug` and minimize code with `gulp-uglify`. This step will only occur when running in production mode.

Install the plugins:

```
npm install gulp-deporder gulp-concat gulp-strip-debug gulp-uglify --save-dev
```

Reference them at the top of `gulpfile.js`:

```
var
  ...
  concat = require('gulp-concat'),
  deporder = require('gulp-deporder'),
  stripdebug = require('gulp-strip-debug'),
  uglify = require('gulp-uglify'),
```

Then add a new `js` task:

```
// JavaScript processing
gulp.task('js', function() {

  var jsbuild = gulp.src(folder.src + 'js/**/*.js')
    .pipe(deporder())
    .pipe(concat('main.js'));

  if (!devBuild) {
    jsbuild = jsbuild
      .pipe(stripdebug())
      .pipe(uglify());
  }

  return jsbuild.pipe(gulp.dest(folder.build + 'js/'));

});
```

Save then run `gulp js` to watch the magic happen!

## CSS TASK

Finally, let's create a CSS task which compiles Sass `.scss` files to a single `.css` file using `gulp-sass`. This is a Gulp plugin for `node-sass` which binds to the superfast `LibSass C/C++ port of the Sass engine` (*you won't need to install Ruby*). We'll presume your primary Sass file `scss/main.scss` is responsible for loading all partials.

Our task will also utilize the fabulous `PostCSS` via the `gulp-postcss` plugin. PostCSS requires its own set of plugins and we'll install these:

- `postcss-assets` to manage assets. This allows us to use properties such as

`background: resolve('image.png');` to resolve file paths or `background: inline('image.png');` to inline data-encoded images.

- autoprefixer to automatically add vendor prefixes to CSS properties.
- css-mqpacker to pack multiple references to the same CSS media query into a single rule.
- cssnano to minify the CSS code when running in production mode.

First, install all the modules:

```
npm install gulp-sass gulp-postcss postcss-assets autoprefixer css-mqpacker cssnano
```

Then reference them at the top of `gulpfile.js`:

```
var
  ...
  sass = require('gulp-sass'),
  postcss = require('gulp-postcss'),
  assets = require('postcss-assets'),
  autoprefixer = require('autoprefixer'),
  mqpacker = require('css-mqpacker'),
  cssnano = require('cssnano'),
```

We can now create a new `css` task at the end of `gulpfile.js`. Note the `images` task is set as a dependency because the `postcss-assets` plugin can reference images during the build process. In addition, most plugins can be passed arguments (refer to their documentation for more information):

```
// CSS processing
gulp.task('css', ['images'], function() {

  var postCssOpts = [
    assets({ loadPaths: ['images/'] }),
    autoprefixer({ browsers: ['last 2 versions', '> 2%'] }),
    mqpacker
  ];

  if (!devBuild) {
    postCssOpts.push(cssnano);
  }

  return gulp.src(folder.src + 'scss/main.scss')
    .pipe(sass({
```

```
        outputFile: 'nested',
        imagePath: 'images/',
        precision: 3,
        errLogToConsole: true
    )))
    .pipe(postcss(postCssOpts))
    .pipe(gulp.dest(folder.build + 'css/'));

});
```

Save the file and run the task from the command line:

```
gulp css
```

## Step 6: Automate Tasks

We've been running one task at a time. We can run them all in one command by adding a new `run` task to `gulpfile.js`:

```
// run all tasks
gulp.task('run', ['html', 'css', 'js']);
```

Save and enter `gulp run` at the command line to execute all tasks. Note that I omitted the `images` task because it's already set as a dependency for the `html` and `css` tasks.

Is this still too much hard work? Gulp offers another method — `gulp.watch` — which can monitor your source files and run an appropriate task whenever a file is changed. The method is passed a folder and a list of tasks to execute when a change occurs. Let's create a new `watch` task at the end of `gulpfile.js`:

```
// watch for changes
gulp.task('watch', function() {

    // image changes
    gulp.watch(folder.src + 'images/**/*', ['images']);

    // html changes
    gulp.watch(folder.src + 'html/**/*', ['html']);

    // javascript changes
    gulp.watch(folder.src + 'js/**/*', ['js']);

    // css changes
    gulp.watch(folder.src + 'scss/**/*', ['css']);
```

```
});
```

Rather than running `gulp watch` immediately, let's add a default task:

```
// default task
gulp.task('default', ['run', 'watch']);
```

Save `gulpfile.js` and enter `gulp` at the command line. Your images, HTML, CSS and JavaScript will all be processed, then Gulp will remain active watching for updates and re-running tasks as necessary. Hit `Ctrl/Cmd + C` to abort monitoring and return to the command line.

## Step 7: Profit!

Other plugins you may find useful:

- `gulp-load-plugins` — load all Gulp plugin modules without `require` declarations
- `gulp-preprocess` — a simple HTML and JavaScript `preprocessor`
- `gulp-less` — the `Less` CSS `preprocessor` plugin
- `gulp-stylus` — the `Stylus` CSS `preprocessor` plugin
- `gulp-sequence` — run a series of gulp tasks in a specific order
- `gulp-plumber` — error handling which prevents Gulp stopping on failures
- `gulp-size` — displays file sizes and savings
- `gulp-nodemon` — uses `nodemon` to automatically restart Node.js applications when changes occur.
- `gulp-util` — utility functions including logging and color coding.

One useful method in `gulp-util` is `.noop()` which passes data straight through without performing any action. This could be used for cleaner development/production processing code. For example:

```
var gutil = require('gulp-util');

// HTML processing
gulp.task('html', ['images'], function() {
  var out = folder.src + 'html/**/*.html';
```

```
return gulp.src(folder.src + 'html/**/*')
    .pipe(newer(out))
    .pipe(devBuild ? gutil.noop() : htmlclean())
    .pipe(gulp.dest(out));

});
```

Gulp can also call other Node.js modules, and they don't necessarily need to be plugins. For example:

- [browser-sync](#) — automatically reload assets or refresh your browser when changes occur
- [del](#) — delete files and folders (perhaps clean your `build` folder at the start of every run).

Invest a little time and Gulp could save many hours of development frustration. The advantages:

- [plugins are plentiful](#)
- configuration using pipes is readable and easy to follow
- `gulpfile.js` can be adapted and reused in other projects
- your total page weight can be reduced to improve performance
- you can simplify your deployment.

Useful links:

- [Gulp home page](#)
- [Gulp plugins](#)
- [npm home page](#)

Applying the processes above to a simple website reduced the total weight by more than 50%. You can test your own results using [page weight analysis tools](#) or a service such as [New Relic](#), which provides a range of sophisticated application performance monitoring tools.

Gulp can revolutionize your workflow. I hope you found this tutorial useful and consider Gulp for your production process.



# Chapter 4: A Side-by-side Comparison of Express, Koa and Hapi.js

**BY OLAYINKA OMOLE**

If you're a Node.js developer, chances are you have, at some point, used Express.js to create your applications or APIs. Express.js is a very popular Node.js framework, and even has some other frameworks built on top of it such as Sails.js, kraken.js, KeystoneJS and many others. However, amidst this popularity, a bunch of other frameworks have been gaining attention in the JavaScript world, such as Koa and hapi.

In this article, we'll examine Express.js, Koa and hapi.js — their similarities, differences and use cases.

## Background

Let's firstly introduce each of these frameworks separately.

### EXPRESS.JS

Express.js is described as the standard server framework for Node.js. It was created by TJ Holowaychuk, acquired by StrongLoop in 2014, and is currently maintained by the Node.js Foundation incubator. With about 170+ million downloads in the last year, it's currently beyond doubt that it's the most popular Node.js framework.

### KOA

Development began on Koa in late 2013 by the same guys at Express. It's referred to as the future of Express. Koa is also described as a much more modern, modular and minimalistic version of the Express framework.

### HAPI.JS

Hapi.js was developed by the team at Walmart Labs (led by Eran Hammer) after they tried Express and discovered that it didn't work for their requirements. It was originally developed on top of Express, but as time went by, it grew into a full-fledged framework.

*Fun Fact: hapi is short for Http API server.*

## Philosophy

Now that we have some background on the frameworks and how they were created, let's compare each of them based on important concepts, such as their philosophy, routing, and so on.

### Compatibility

All code examples are in ES6 and make use of version 4 of Express.js, 2.4 of Koa, and 17 for hapi.js.

## EXPRESS.JS

Express was built to be a simple, unopinionated web framework. From its GitHub README:

The Express philosophy is to provide small, robust tooling for HTTP servers, making it a great solution for single page applications, web sites, hybrids, or public HTTP APIs.

Express.js is minimal and doesn't possess many features out of the box. It doesn't force things like file structure, ORM or templating engine.

## KOA

While Express.js is minimal, Koa can boast a much more minimalistic code footprint — around 2k LOC. Its aim is to allow developers be even more expressive. Like Express.js, it can easily be extended by using existing or custom plugins and middleware. It's more futuristic in its approach, in that it relies heavily on the relatively new JavaScript features like generators and async/await.

## HAPI.JS

Hapi.js focuses more on configuration and provides a lot more features out of the box

than Koa and Express.js. Eran Hammer, one of the creators of hapi, described the reason for building the framework properly in his blog post:

hapi was created around the idea that configuration is better than code, that business logic must be isolated from the transport layer, and that native node constructs like buffers and stream should be supported as first class objects.

## Starting a Server

Starting a server is one of the basic things we'd need to do in our projects. Let's examine how it can be done in the different frameworks. We'll start a server and listen on port 3000 in each example.

### EXPRESS.JS

```
const express = require('express');
const app = express();

app.listen(3000, () => console.log('App is listening on port 3000!'));
```

Starting a server in Express.js is as simple as requiring the `express` package, initializing the `express` app to the `app` variable and calling the `app.listen()` method, which is just a wrapper around the native Node.js `http.createServer()` method.

### KOA

Starting a server in Koa is quite similar to Express.js:

```
const Koa = require('koa');
const app = new Koa();

app.listen(3000, () => console.log('App is listening on port 3000!'));
```

The `app.listen()` method in Koa is also a wrapper around the `http.createServer()` method.

### HAPI.JS

Starting a server in hapi.js is quite a departure from what many of us may be used to from Express:

```
const Hapi = require('hapi');
```

```

const server = Hapi.server({
  host: 'localhost',
  port: 3000
});

async function start() {
  try {
    await server.start();
  }
  catch (err) {
    console.log(err);
    process.exit(1);
  }
  console.log('Server running at:', server.info.uri);
};

start();

```

In the code block above, first we require the `hapi` package, then instantiate a server with `Hapi.server()`, which has a single config object argument containing the host and port parameters. Then we start the server with the asynchronous `server.start()` function.

Unlike in `Express.js` and `Koa`, the `server.start()` function in `hapi` is not a wrapper around the native `http.createServer()` method. It instead implements its own custom logic.

The above code example is from the `hapi.js` website, and shows the importance the creators of `hapi.js` place on configuration and error handling.

## Routing

Routing is another key aspect of modern web applications. Let's define a `/hello` route for a simple Hello World app in each framework to have a feel of how routing works for them.

### EXPRESS.JS

```

app.get('/hello', (req, res) => res.send('Hello World!'));

```

Creating routes in `Express` is as simple as calling the `app` object with the required HTTP method. The syntax is `app.METHOD(PATH, HANDLER)`, where `PATH` is the path on the server and `HANDLER` is function which is called when the path is matched.

## KOA

Koa doesn't have its own router bundled with it, so we'll have to use a router middleware to handle routing on Koa apps. Two common routing options are koa-route and koa-router. Here's an example using koa-route:

```
const route = require('koa-route');

app.use(route.get('/hello', ctx => {
  ctx.body = 'Hello World!';
}));
```

We can see immediately that Koa needs each route to be defined as a middleware on the app. The `ctx` is a context object that contains Node's request and response objects. `ctx.body` is a method in the response object and can be used to set the response body to either a string, Buffer, Stream, Object or null. The second parameter for the route method can be an async or generator function, so the use of callbacks is reduced.

## HAPI.JS

```
server.route({
  method: 'GET',
  path: '/hello',
  handler: function (request, h) {
    return 'Hello world!';
  }
});
```

The `server.route()` method in hapi takes a single config object with the following parameters: `method`, `path` and `handler`. You can see the documentation on routing in hapi here.

The `request` parameter in the handler function is an object which contains the user's request details, while the `h` parameter is described as a response toolkit.

## Middleware

One of the major concepts Node developers are used to is working with middleware. **Middleware functions** are functions that sit in between requests and responses. They have access to the `request` and `response` objects and can run the next middleware after they're processed. Let's take a look at how they're defined in the

different frameworks by implementing a simple function that logs the time a request is made to the server.

## EXPRESS.JS

```
app.use((req, res, next) => {
  console.log(`Time: ${Date.now()} `);
  next();
})
```

Registering middleware in Express.js is as simple as binding the middleware to the `app` object by using the `app.use()` function. You can read more on middleware in Express.js [here](#).

## Koa

```
app.use(async (ctx, next) => {
  console.log(`Time: ${Date.now()} `);
  await next();
});
```

Middleware registration in Koa is similar to Express.js. The major differences are that the `context object (ctx)` is used in place of the `request` and `response` objects in Express.js and Koa embraces the modern `async/await` paradigm for defining the middleware function.

## HAPI.JS

```
server.ext('onRequest', (request, h) => {
  console.log(`Time: ${Date.now()} `);
  return h.continue;
});
```

In hapi.js there are certain extension points in the `request lifecycle`. The `server.ext()` method registers an extension function to be called at a certain point in the request life cycle. You can read more about it [here](#). We make use of the `onRequest` extension point in the example above to register a middleware (or extension) function.

## Usage

From the comparisons and code examples we've seen above, it's clear that Express and Koa are the most similar, with hapi.js being the framework to deviate from the norm

that Node.js devs are used to. Hence hapi.js may not be the best choice when trying to build a quick and easy app, as it will take a bit of time to get used to.

In my opinion, Express is still a great choice when building small- to medium-sized applications. It can become a bit complicated to manage for very large applications, as it doesn't possess the modularity hapi.js has built into it, with support for custom plugins and its unique routing method. However, there's been some speculation in recent times regarding the future of Express.js as TJ announced he's no longer working on it and the reduced rate at which updates are shipped. But it's pretty stable and won't be going away any time soon. It also has a large community of developers building various extensions and plugins for it.

Like Express.js, Koa is well suited for many simple Node.js projects. It only consists of the bare minimum (it has zero in-built middleware) and encourages developers to add what they need to it by building or making use of available external middleware. It makes use of modern JavaScript generator functions and `async/await` heavily, which makes it sort of futuristic in its approach. Its middleware cascading pattern is also great, as it makes implementing and understanding the flow of middleware in your applications very easy. Koa probably won't be a great choice for you if you aren't yet ready to embrace new shiny things like generator functions, or if you're not willing to spend some time building out all the middleware you need. The community support for Koa is rapidly growing, as it has a good amount of external middleware already built for it (some by the core Koa team) for common tasks such as routing, logging and so on.

Hapi.js is the definite choice if you and your team prefer to spend more time configuring than actually coding out features. It was built to be modular and for large applications with large teams. It encourages the micro-service architecture, as various parts of your app can be built as plugins and registered in your server before starting it up. Hapi.js is backed by large companies such as Auth0 and Lob, so it has a pretty good future ahead of it and won't be going away anytime soon. It's also trusted by some big names, as seen on their community page.

Hapi.js has a whole lot more features out of the box than Koa and Express.js, such as support for authentication, caching, logging, validation and so on, which makes it feel more like a full-fledged framework. You can check out their tutorials page to get a good feel of the features they provide. There aren't yet very many open-source projects and plugins built on and for hapi.js, so a lot of work might need to be done by developers using it if they plan to extend its core functionality.

## Conclusion

All three frameworks are great choices when starting up new projects, but ultimately your choice will be based on the project requirements, your team members and the level of flexibility you're looking for.



# Chapter 5: An Introduction to Sails.js

BY AHMED BOUCHEFRA

**Sails.js** is a Node.js MVC (model–view–controller) framework that follows the “convention over configuration” principle. It’s inspired by the popular Ruby on Rails web framework, and allows you to quickly build REST APIs, single-page apps and real-time (WebSockets-based) apps. It makes extensive use of code generators that allow you to build your application with less writing of code—particularly of common code that can be otherwise scaffolded.

The framework is built on top of Express.js, one of the most popular Node.js libraries, and Socket.io, a JavaScript library/engine for adding real-time, bidirectional, event-based communication to applications. At the time of writing, the official stable version of Sails.js is *0.12.14*, which is available from [npm](https://www.npmjs.com/package/sails). Sails.js version 1.0 has not officially been released, but [according to Sails.js creators](#), version **1.0** is already used in some production applications, and they even recommend using it when starting new projects.

## Main Features

Sails.js has many great features:

- it’s built on Express.js
- it has real-time support with WebSockets
- it takes a “convention over configuration” approach
- it has powerful code generation, thanks to Blueprints
- it’s database agnostic thanks to its powerful Waterline ORM/ODM
- it supports multiple data stores in the same project
- it has good documentation.

There are currently a few important cons, such as:

- no support for JOIN query in Waterline
- no support for SQL transactions until Sails v1.0 (in beta at the time of writing)
- until version 1.0, it still uses Express.js v3, which is EOL (end of life)
- development is very slow.

## Sails.js vs Express.js

Software development is all about building abstractions. Sails.js is a high-level abstraction layer on top of Express.js (which itself is an abstraction over Node's HTTP modules) that provides routing, middleware, file serving and so on. It also adds a powerful ORM/ODM, the MVC architectural pattern, and a powerful generator CLI (among other features).

You can build web applications using Node's low-level HTTP service and other utility modules (such as the filesystem module) but it's not recommended except for the sake of learning the Node.js platform. You can also take a step up and use Express.js, which is a popular, lightweight framework for building web apps.

You'll have routing and other useful constructs for web apps, but you'll need to take care of pretty much everything from configuration, file structure and code organization to working with databases.

Express doesn't offer any built-in tool to help you with database access, so you'll need to bring together the required technologies to build a complete web application. This is what's called a stack. Web developers, using JavaScript, mostly use the popular MEAN stack, which stands for MongoDB, ExpressJS, AngularJS and Node.js.

MongoDB is the preferred database system among Node/Express developers, but you can use any database you want. The most important point here is that Express doesn't provide any built-in APIs when it comes to databases.

## The Waterline ORM/ODM

One key feature of Sails.js is Waterline, a powerful ORM (object relational mapper) for SQL-based databases and ODM (object document mapper) for NoSQL document-based databases. Waterline abstracts away all the complexities when working with databases and, most importantly, with Waterline you don't have to make the decision of choosing

a database system when you're just starting development. It also doesn't intimidate you when your client hasn't yet decided on the database technology to use.

You can start building your application without a single line of configuration. In fact, you don't have to install a database system at all initially. Thanks to the built-in `sails-disk` NeDB-based file database, you can transparently use the file system to store and retrieve data for testing your application functionality.

Once you're ready and you have decided on the convenient database system you want to use for your project, you can then simply switch the database by installing the relevant adapter for your database system. Waterline has official adapters for popular relational database systems such as MySQL and PostgreSQL and the NoSQL databases, such as MongoDB and Redis, and the community has also built numerous adapters for the other popular database systems such as Oracle, MSSQL, DB2, SQLite, CouchDB and neo4j. In case when you can't find an adapter for the database system you want to use, you can develop your own custom adapter.

Waterline abstracts away the differences between different database systems and allows you to have a normalized interface for your application to communicate with any supported database system. You don't have to work with SQL or any low-level API (for NoSQL databases) but that doesn't mean you can't (at least for SQL-based databases and MongoDB).

There are situations when you need to write custom SQL, for example, for performance reasons, for working with complex database requirements, or for accessing database-specific features. In this case, you can use the `.query()` method available only on the Waterline models that are configured to use SQL systems (you can find more information about `query()` from the docs).

Since different database systems have common and database-specific features, the Waterline ORM/ODM can only be good for you as long as you only constrain yourself to use the common features. Also, if you use raw SQL or native MongoDB APIs, you'll lose many of the features of Waterline, including the ability to switch between different databases.

## Getting Started with Sails.js

Now that we've covered the basic concepts and features of Sails.js, let's see how you can quickly get started using Sails.js to create new projects and lift them.

## PREREQUISITES

Before you can use Sails.js, you need to have a development environment with Node.js (and npm) installed. You can install both of them by heading to the [official Node.js website](#) and downloading the right installer for your operating system.

### Downloads

Latest LTS Version: 8.9.4 (includes npm 5.6.0)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.

LTS Recommended For Most Users	Current Latest Features	
 Windows Installer <small>node-v8.9.4-x86.msi</small>	 macOS Installer <small>node-v8.9.4.pkg</small>	 Source Code <small>node-v8.9.4.tar.gz</small>

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binaries (.tar.gz)

Linux Binaries (x86/x64)

Linux Binaries (ARM)

Source Code

32-bit		64-bit	
32-bit		64-bit	
64-bit			
64-bit			
32-bit		64-bit	
ARMv6	ARMv7		ARMv8
node-v8.9.4.tar.gz			

### Additional Platforms

SunOS Binaries

Docker Image

Linux on Power Systems

Linux on System z

AIX on Power Systems

32-bit	64-bit
Official Node.js Docker Image	
64-bit le	
64-bit	
64-bit	

- [Signed SHASUMS for release files](#)

Make sure, also, to install whatever database management system you want to use with Sails.js (either a relational or a NoSQL database). If you're not interested by using a full-fledged database system, at this point, you can still work with Sails.js thanks to `sails-disk`, which allows you to have a file-based database out of the box.

## INSTALLING THE SAILS.JS CLI

After satisfying the working development requirements, you can head over to your terminal (Linux and macOS) or command prompt (Windows) and install the Sails.js Command Line Utility, globally, from npm:

```
sudo npm install sails -g
```

If you want to install the latest *1.0* version to try the new features, you need to use the beta version:

```
npm install sails@beta -g
```

You may or may not need *sudo* to install packages globally depending on your npm configuration.

## SCAFFOLDING A SAILS.JS PROJECT

After installing the Sails.js CLI, you can go ahead and scaffold a new project with one command:

```
sails new sailsdemo
```

This will create a new folder for your project named `sailsdemo` on your current directory. You can also scaffold your project files inside an existing folder with this:

```
sails new .
```

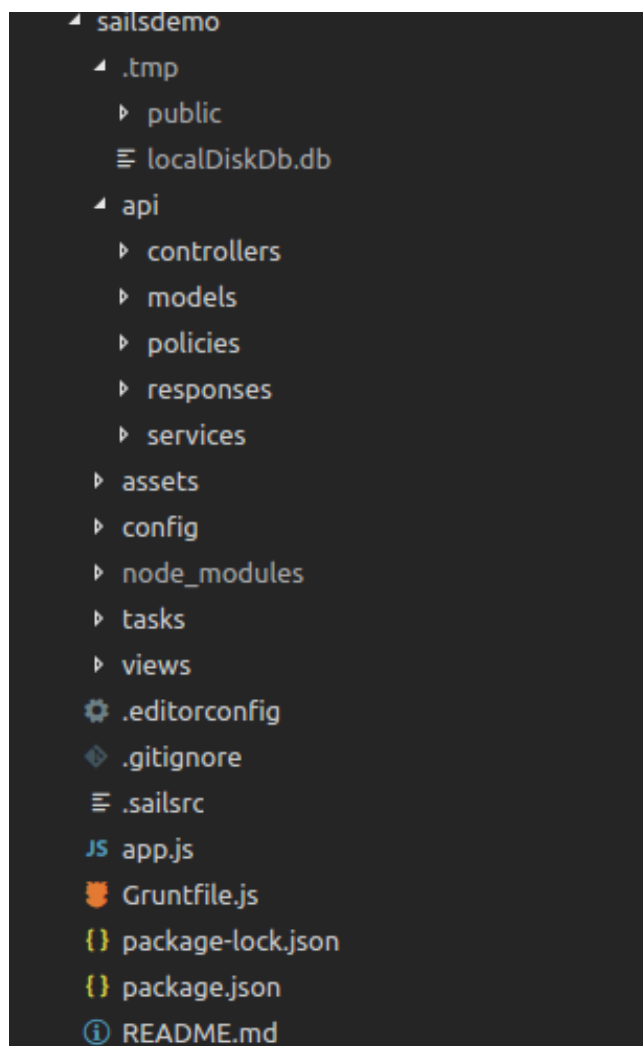
You can scaffold a new Sails.js project without a front end with this:

```
sails new sailsdemo --no-frontend
```

Find more information about the features of the CLI from the docs.

## THE ANATOMY OF A SAILS.JS PROJECT

Here's a screenshot of a project generated using the Sails.js CLI:



A Sails.js project is a Node.js module with a `package.json` and a `node_modules` folder. You may also notice the presence of `Gruntfile.js`. Sails.js uses Grunt as a build tool for building front-end assets.

If you're building an app for the browser, you're in luck. Sails ships with Grunt — which means your entire front-end asset workflow is completely customizable, and comes with support for all of the great Grunt modules which are already out there. That includes support for Less, Sass, Stylus, CoffeeScript, JST, Jade, Handlebars, Dust, and many more. When you're ready to go into production, your assets are minified and gzipped automatically. You can even compile your static assets and push them out to a CDN like CloudFront to make your app load even faster. (You can read more about these points [on the Sails.js website](#).)

You can also use Gulp or Webpack as your build system instead of Grunt, with custom generators. See the [sails-generate-new-gulp](#) and [sails-webpack](#) projects on GitHub.

For more community generators, see [this documentation page](#) on the Sails.js site.

The project contains many configuration files and folders. Most of them are self explanatory, but let's go over the ones you'll be working with most of the time:

- `api/controllers`: this is the folder where controllers live. Controllers correspond to the *C* part in *MVC*. It's where the business logic for your application exists.
- `api/models`: the folder where models exist. Models correspond to the *M* part of *MVC* architecture. This is where you need to put classes or objects that map to your SQL/NoSQL data.
- `api/policies`: this is the folder where you need to put policies for your application
- `api/responses`: this folder contains server response logic such as functions to handle the 404 and 500 responses, etc.
- `api/services`: this where your app-wide services live. A service is a global class encapsulating common logic that can be used throughout many controllers.
- `./views`: this folder contains templates used for displaying views. By default, this folder contains the *ejs* engine templates, but you can configure any Express-supported engine such as EJS, Jade, Handlebars, Mustache and Underscore etc.
- `./config`: this folder contains many configuration files that enable you to configure every detail of your application, such as CORS, CSRF protection, i18n, http, settings for models, views, logging and policies etc. One important file that you'll frequently use is `config/routes.js`, where you can create your application routes and map them to actual actions in the controllers or to views directly.
- `./assets`: this is the folder where you can place any static files (CSS, JavaScript and images etc.) for your application.

## RUNNING YOUR SAILS.JS PROJECT

You can start the development server by running the following command from your project's root:

```
sails lift
```

This will prompt you to choose a migration strategy, and then will launch the dev server.

```
Terminal File Edit View Search Terminal Help
info:
info:      .-.-.-.
info:
info:  Sails      <|      .-.-.-.
info:  v0.12.14
info:
info:      / \
info:     /  \
info:    /    \
info:   /      \
info:  /==\  / \
info: /-----\
info:
info:  -----
info:  -----
info:
info: Server lifted in `~/home/ahmed/glib/demos/nodejs/sails.js/sailsdemo`
info: To see your app, visit http://localhost:1337
info: To shut down Sails, press <CTRL> + C at any time.
debug: -----
debug: :: Sun Jan 07 2018 00:19:01 GMT+0000 (WET)
debug: Environment : development
debug: Port       : 1337
debug: -----
```

You can then use your web browser to navigate to [<http://localhost:1337/>] (<http://localhost:1337/>). If you've generated a Sails.js project with a front end (i.e without using the `--no-frontend` option) you'll see this home page:

## A brand new app.

You're looking at: `views/homepage.ejs`

- 1 Generate a REST API.**

Run `sails generate api user`. This will create two files: a [model](#) and a [controller](#).
- 2 Lift your app.**

Run `sails lift` to start up your app server. If you visit <http://localhost:1337/user> in your browser, you'll see a [WebSocket-compatible](#) user API.
- 3 Dive in.**

Blueprints are just the beginning. You'll probably also want to learn how to customize your app's [routes](#), set up [security policies](#), configure your [data sources](#), and build custom [controller actions](#). For more help getting started, check out the links on this page.

Docs

- [App Structure](#)
- [Reference](#)
- [Supported Databases](#)

Tutorials

- [Sails 101](#)

Community

- [StackOverFlow](#)
- [GitHub](#)
- [Google Group](#)
- [IRC \(#sailsjs on freenode\)](#)



A **model** is an abstraction, usually represented by an object or a class in a general-purpose programming language, and it refers/maps either to an SQL table in a relational database or a document (or key-value pairs) in a NoSQL database.

You can create models using the Sails.js CLI:

```
sails generate model product
```

This will create a `Product.js` model in `api/models` with the following content:

```
/**
 * Product.js
 *
 * @description :: TODO: You might write a short summary of how this model
 * @docs         :: http://sailsjs.org/documentation/concepts/models-and-orm
 */

module.exports = {

  attributes: {

  }

};
```

You can then augment your model with attributes. For example:

```
module.exports = {

  attributes: {
    name: {
      type: 'string',
      defaultsTo: '',
      required: 'true'
    },
    description: {
      type: 'string',
      defaultsTo: ''
    },
    quantity: {
      type: 'integer'
    },
    user: { model: 'User' }
  }
};
```

Notice how we can define the association (one-to-many or belongsTo relationship) with the model *User*. You can see all supported associations and how to create them via this [Sails.js associations page](#).

For more information about the available model attributes, see the [Sails.js attributes page](#).

You can also add configurations per model or model settings by adding top-level properties in the model definition, which will override the global models settings in `config/models.js`. You can override settings related to the model's attributes, database connections etc.

For example, let's specify a different datastore for the *product* model other than the global one used throughout the project:

```
module.exports = {
  connection: 'mysqlcon'
  attributes: { /*...*/ }
}
```

This will instruct Sails.js to use a connection named *mysqlcon* to store this model data. Make sure you add the *mysqlcon* connection to the *connections* object in `config/connections.js`:

```
module.exports.connections = {
  // sails-disk is installed by default.
  localDiskDb: {
    adapter: 'sails-disk'
  },
  mysqlcon: {
    adapter: 'sails-mysql',
    host: 'YOUR_MYSQL_HOST',
    user: 'YOUR_MYSQL_USER',
    password: 'YOUR_MYSQL_PASSWORD',
    database: 'YOUR_MYSQL_DB'
  }
};
```

You also need to install the *sails-mysql* adapter from npm:

```
npm install sails-mysql@for-sails-0.12
```

You can find the available model settings that you can specify from the [Sails.js model](#)

[settings page](#).

## SAILS.JS CONTROLLERS

**Controllers** hold your app's business logic. They live in `api/controllers` and provide a layer which glues your app's models and views. Controllers contain actions that are bound to routes and respond to HTTP requests from web/mobile clients.

A controller is a JavaScript object that contains methods called the *controller actions*, which take two parameters: a request and a response.

You can find more information about controllers on the [Sails.js controllers page](#).

You can generate a controller using the Sails.js CLI:

```
sails generate controller product
```

This command will generate a controller named `api/controllers/ProductController.js`, with the following content:

```
/**
 * ProductController
 *
 * @description :: Server-side logic for managing products
 * @help        :: See http://sailsjs.org/#!/documentation/concepts/Contro
 */

module.exports = {

};
```

The code exports an empty JavaScript object where you can add new actions or override the default (automatically added) controller actions.

At this point, you can actually execute CRUD operations against your server without further adding any code. Since Sails.js follows convention over configuration, it wires your controllers to their corresponding routes and provides default actions for handling the common HTTP POST, GET, PUT and DELETE requests etc.

## TESTING WITH POSTMAN

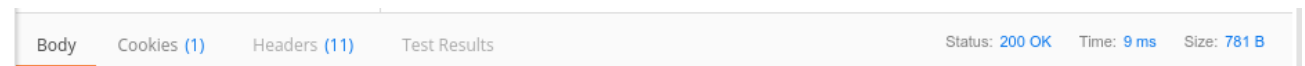
Using Postman, you can send POST, GET and other requests to test your API, so go

ahead and grab the Postman version for your operating system. Next, enter the product endpoint URL `http://localhost:1337/product`. Then choose the HTTP method to send — POST in this case, because we want to create a Product. Next, you need to provide data, so click on the *Body tab*, select the *Raw* option, then enter the following:

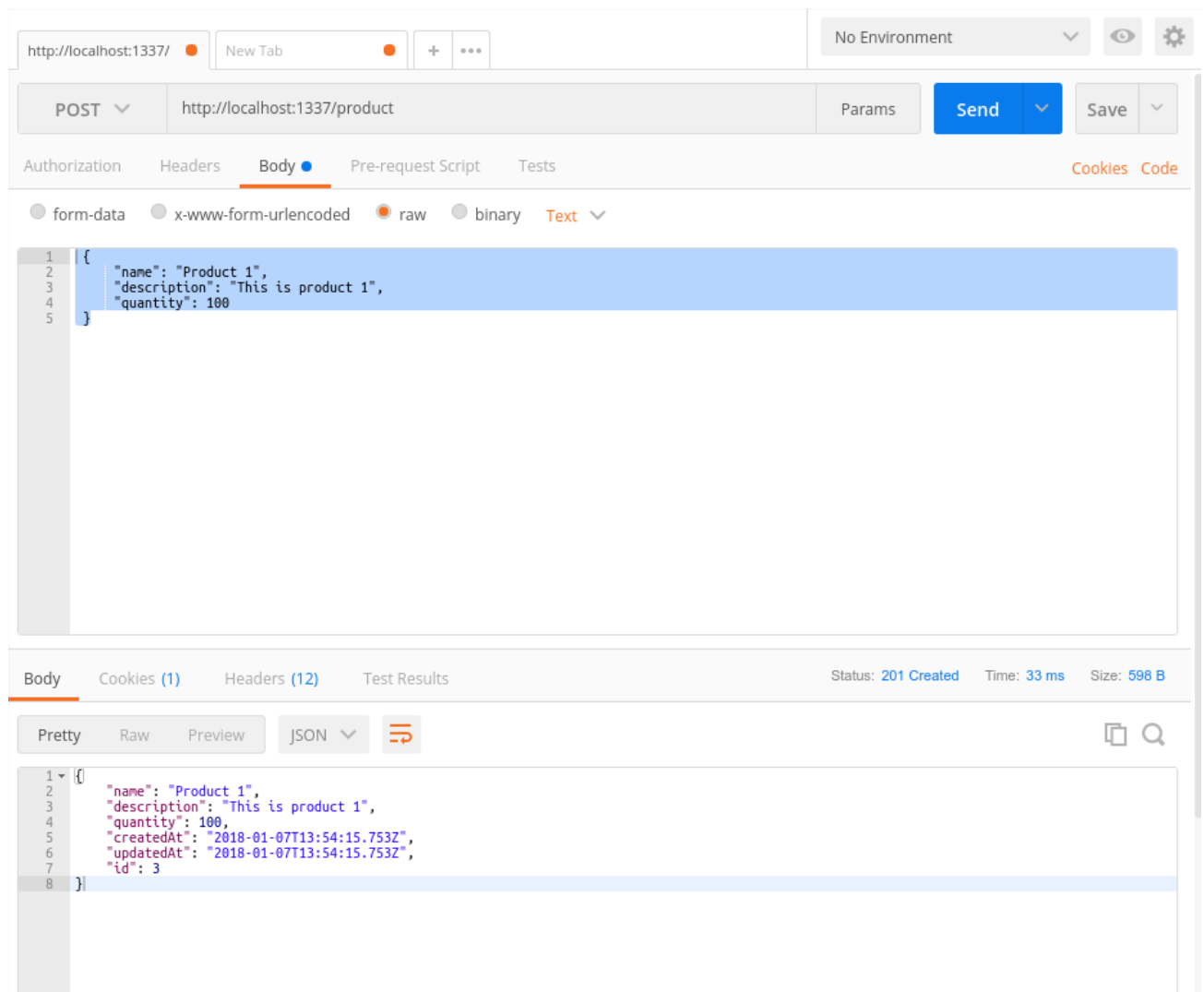
```
{
  "name": "Product 1",
  "description": "This is product 1",
  "quantity": 100
}
```

Then hit the Send button.

You should pay attention to the returned status code: 200 OK means the product was successfully created.



You can then verify if products are created by sending a GET request:



You can also update a product by its id by sending a PUT request:

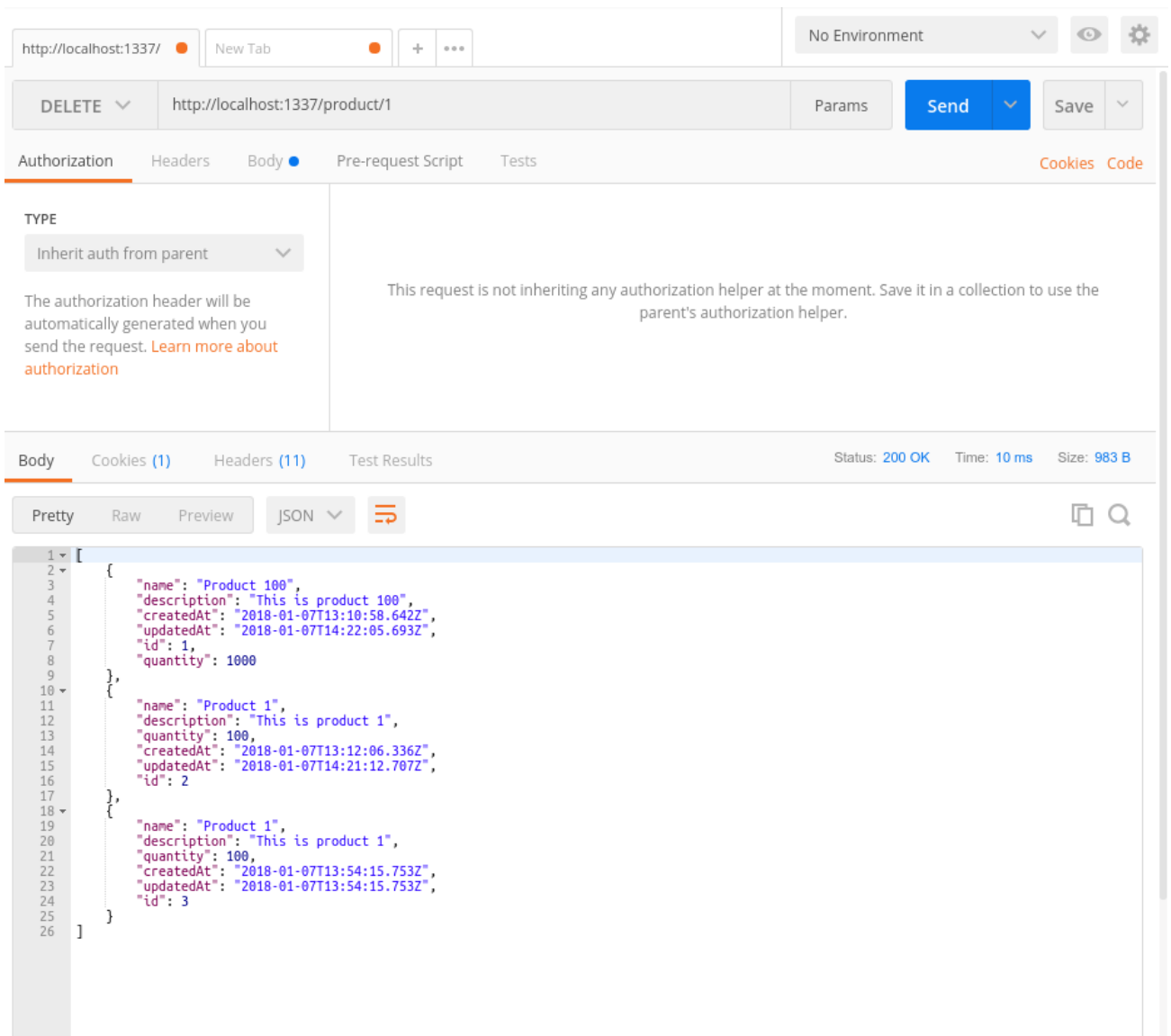
The screenshot shows a REST client interface with the following details:

- URL:** `http://localhost:1337/product/1`
- Method:** PUT
- Body Type:** raw (selected)
- Request Body (raw):**

```
1 {  
2   "name": "Product 100",  
3   "description": "This is product 100",  
4   "quantity": 1000  
5 }
```
- Response Status:** 200 OK
- Response Time:** 30 ms
- Response Size:** 568 B
- Response Body (Pretty JSON):**

```
1 {  
2   "name": "Product 100",  
3   "description": "This is product 100",  
4   "createdAt": "2018-01-07T13:10:58.642Z",  
5   "updatedAt": "2018-01-07T14:22:05.693Z",  
6   "id": 1,  
7   "quantity": 1000  
8 }
```

Finally, you can delete a product by its id by sending a DELETE request:



For custom logic, you can also override these actions and implement your own.

When you create an API (i.e. a controller and a model) Sails.js automatically adds eight default actions, which are:

- `add to`
- `create`
- `destroy`
- `find one`
- `find where`
- `populate where`
- `remove from`
- `update`

Find where and find one, create, update and destroy are normal CRUD actions that needs to be present in most APIs. The others are related to foreign records:

- `add to`: used to add a foreign record to another record collection (e.g a product to a user's products).
- `populate where`: used to populate and return foreign record(s) for the given association of another record. [Read more information here.](#)
- `remove from`: used to remove a foreign record (e.g. a product) from one of a related record collection association (e.g. user's products). See [more information and examples here.](#)

For customizing the behavior of the default actions, you can do either of these:

- Override the action in a specified controller. That is, create an action with the same name as one of these actions: `find`, `findOne`, `create`, `update`, `destroy`, `populate`, `add` or `remove`.
- Override the default action for all controllers. You can do this by creating an `api/blueprints` folder, where you need to add files with lowercase names for a default action (e.g. `find.js`, `findone.js`, `create.js`, etc.). You can find more information about blueprints by checking the [Sails.js Blueprint API docs.](#)

## ROUTING IN SAILS.JS

[Routes](#) allow you to map URLs to controllers or views. Just like the default actions, Sails.js automatically adds default routes for the default actions, allowing you to have an automatic API by just creating models and controllers.

You can also add custom routes for your custom actions or views. To add a route, open the `config/routes.js` file and add this:

```
module.exports.routes = {  
  '/products': {  
    view: 'products'  
  }  
};
```

This maps `/products` to the template named *products* in views folder.

You can optionally add an HTTP verb to the URL. For example:

```
module.exports.routes = {  
  'get /': {  
    view: 'homepage'  
  }  
};
```

You can also specify a controller action for a route. For example:

```
module.exports.routes = {  
  'post /product': 'ProductController.create',  
};
```

This tells Sails.js to call the *create* action of the *ProductController* controller when a client sends a POST request to the `/product` endpoint.

## Conclusion

In this chapter, you were introduced to Sails.js. We looked at the basic concepts of Sails.js, and how to generate a new Sails.js project, and then created an API by just generating models, adding some attributes then generate controllers. Sails.js has other advanced concepts such as services, policies, blueprints and hooks. These you can further discover on your own, once you grasp and get familiar with the basic concepts in this introduction.



# Chapter 6: Building Apps and Services with the Hapi.js Framework

BY MARK BROWN

**Hapi.js** is described as “a rich framework for building applications and services”. Hapi’s smart defaults make it a breeze to create JSON APIs, and its modular design and plugin system allow you to easily extend or modify its behavior.

The recent release of version 17.0 has fully embraced `async` and `await`, so you’ll be writing code that appears synchronous but is non-blocking *and* avoids callback hell. Win-win.

## The Project

In this article, we’ll be building the following API for a typical blog from scratch:

```
# RESTful actions for fetching, creating, updating and deleting articles
GET      /articles                articles#index
GET      /articles/:id           articles#show
POST     /articles                articles#create
PUT      /articles/:id           articles#update
DELETE   /articles/:id           articles#destroy

# Nested routes for creating and deleting comments
POST     /articles/:id/comments  comments#create
DELETE   /articles/:id/comments  comments#destroy

# Authentication with JSON Web Tokens (JWT)
POST     /authentications        authentications#create
```

The article will cover:

- Hapi’s core API: routing, request and response

- models and persistence in a relational database
- routes and actions for Articles and Comments
- testing a REST API with HTTPie
- authentication with JWT and securing routes
- validation
- an HTML View and Layout for the root route /.

## The Starting Point

Make sure you've got a recent version of Node.js installed; `node -v` should return 8.9.0 or higher.

Download the starting code from here with git:

```
git clone https://github.com/markbrown4/hapi-api.git
cd hapi-api
npm install
```

Open up `package.json` and you'll see that the "start" script runs `server.js` with `nodemon`. This will take care of restarting the server for us when we change a file.

Run `npm start` and open `http://localhost:3000/`:

```
[{"so": "hapi!"}]
```

Let's look at the source:

```
// server.js
const Hapi = require('hapi')

// Configure the server instance
const server = Hapi.server({
  host: 'localhost',
  port: 3000
})

// Add routes
server.route({
  method: 'GET',
  path: '/',
  handler: () => {
```

```

    return [{ so: 'hapi!' }]
  }
})

// Go!
server.start().then(() => {
  console.log('Server running at:', server.info.uri)
}).catch(err => {
  console.log(err)
  process.exit(1)
})

```

## The Route Handler

The route handler is the most interesting part of this code. Replace it with the code below, comment out the return lines one by one, and test the response in your browser.

```

server.route({
  method: 'GET',
  path: '/',
  handler: () => {
    // return [{ so: 'hapi!' }]
    return 123
    return `<h1><marquee>HTML <em>rules!</em></>`
    return null
    return new Error('Boom')
    return Promise.resolve({ whoa: true })
    return require('fs').createReadStream('index.html')
  }
})

```

To send a response, you simply return a value and Hapi will send the appropriate body and headers.

- An Object will respond with stringified JSON and Content-Type: application/json
- String values will be Content-Type: text/html
- You can also return a Promise or Stream.

The handler function is often made `async` for cleaner control flow with Promises:

```

server.route({
  method: 'GET',

```

```
path: '/',
handler: async () => {
  let html = await Promise.resolve(`<h1>Google</h1>`)
  html = html.replace('Google', 'Hapi')

  return html
}
```

It's not *always* cleaner with `async` though. Sometimes returning a Promise is simpler:

```
handler: () => {
  return Promise.resolve(`<h1>Google</h1>`)
    .then(html => html.replace('Google', 'Hapi'))
}
```

We'll see better examples of how `async` helps us out when we start interacting with the database.

## The Model Layer

Like the popular Express.js framework, Hapi is a minimal framework that doesn't provide any recommendations for the Model layer or persistence. You can choose any database and ORM that you'd like, or none — it's up to you. We'll be using SQLite and the Sequelize ORM in this tutorial to provide a clean API for interacting with the database.

SQLite comes pre-installed on macOS and most Linux distributions. You can check if it's installed with `sqlite -v`. If not, you can find installation instructions at the SQLite website.

Sequelize works with many popular relational databases like Postgres or MySQL, so you'll need to install both `sequelize` and the `sqlite3` adapter:

```
npm install --save sequelize sqlite3
```

Let's connect to our database and write our first table definition for `articles`:

```
// models.js
const path = require('path')
const Sequelize = require('sequelize')
```

```
// configure connection to db host, user, pass - not required for SQLite
const sequelize = new Sequelize(null, null, null, {
  dialect: 'sqlite',
  storage: path.join('tmp', 'db.sqlite') // SQLite persists its data directly
})

// Here we define our Article model with a title attribute of type string,
const Article = sequelize.define('article', {
  title: Sequelize.STRING,
  body: Sequelize.TEXT
})

// Create table
Article.sync()

module.exports = {
  Article
}
```

Let's test out our new model by importing it and replacing our route handler with the following:

```
// server.js
const { Article } = require('./models')

server.route({
  method: 'GET',
  path: '/',
  handler: () => {
    // try commenting these lines out one at a time
    return Article.findAll()
    return Article.create({ title: 'Welcome to my blog', body: 'The happiest place on earth' })
    return Article.findById(1)
    return Article.update({ title: 'Learning Hapi', body: `JSON API's a breeze` }, { where: { id: 1 } })
    return Article.findAll()
    return Article.destroy({ where: { id: 1 } })
    return Article.findAll()
  }
})
```

If you're familiar with SQL or other ORM's, the Sequelize API should be self explanatory, It's built with Promises so it works great with Hapi's `async` handlers too.

*Note: using `Article.sync()` to create the tables or `Article.sync({ force: true })` to drop and create are fine for the purposes of this demo. If you're wanting to use this in production you should check out sequelize-cli and write Migrations for*

*any schema changes.*

## Our RESTful Actions

Let's build the following routes:

GET	/articles	fetch all articles
GET	/articles/:id	fetch article by id
POST	/articles	create article with `{ title, body }` params
PUT	/articles/:id	update article with `{ title, body }` params
DELETE	/articles/:id	delete article by id

Add a new file, `routes.js`, to separate the server config from the application logic:

```
// routes.js
const { Article } = require('./models')

exports.configureRoutes = (server) => {
  // server.route accepts an object or an array
  return server.route([
    {
      method: 'GET',
      path: '/articles',
      handler: () => {
        return Article.findAll()
      }
    }, {
      method: 'GET',
      // The curly braces are how we define params (variable path segments in
      path: '/articles/{id}',
      handler: (request) => {
        return Article.findById(request.params.id)
      }
    }, {
      method: 'POST',
      path: '/articles',
      handler: (request) => {
        const article = Article.build(request.payload.article)

        return article.save()
      }
    }, {
      // method can be an array
      method: ['PUT', 'PATCH'],
      path: '/articles/{id}',
      handler: async (request) => {
        const article = await Article.findById(request.params.id)
        article.update(request.payload.article)

        return article.save()
      }
    }
  ])
}
```

```

    }
  }, {
    method: 'DELETE',
    path: '/articles/{id}',
    handler: async (request) => {
      const article = await Article.findById(request.params.id)

      return article.destroy()
    }
  })
}
}

```

Import and configure our routes before we start the server:

```

// server.js
const Hapi = require('hapi')
const { configureRoutes } = require('./routes')

const server = Hapi.server({
  host: 'localhost',
  port: 3000
})

// This function will allow us to easily extend it later
const main = async () => {
  await configureRoutes(server)
  await server.start()

  return server
}

main().then(server => {
  console.log('Server running at:', server.info.uri)
}).catch(err => {
  console.log(err)
  process.exit(1)
})

```

## Testing Our API Is as Easy as HTTPie

HTTPie is great little command-line HTTP client that works on all operating systems. Follow the installation instructions in the documentation and then try hitting the API from the terminal:

```

http GET http://localhost:3000/articles
http POST http://localhost:3000/articles article='{ "title": "Welcome to m
http POST http://localhost:3000/articles article='{ "title": "Learning Hap

```

```
http GET http://localhost:3000/articles
http GET http://localhost:3000/articles/2
http PUT http://localhost:3000/articles/2 article:='{ "title": "True happine
http GET http://localhost:3000/articles/2
http DELETE http://localhost:3000/articles/2
http GET http://localhost:3000/articles
```

Okay, everything seems to be working well. Let's try a few more:

```
http GET http://localhost:3000/articles/12345
http DELETE http://localhost:3000/articles/12345
```

*Yikes!* When we try to fetch an article that doesn't exist, we get a 200 with an empty body and our destroy handler throws an `Error` which results in a 500. This is happening because `findById` returns `null` by default when it can't find a record. We want our API to respond with a 404 in both of these cases. There's a few ways we can achieve this.

## DEFENSIVELY CHECK FOR `NULL` VALUES AND RETURN AN ERROR

There's a package called `boom` which helps create standard error response objects:

```
npm install --save boom
```

Import it and modify `GET /articles/:id` route:

```
// routes.js
const Boom = require('boom')

{
  method: 'GET',
  path: '/articles/{id}',
  handler: async (request) => {
    const article = await Article.findById(request.params.id)
    if (article === null) return Boom.notFound()

    return article
  }
}
```

## EXTEND `SEQUELIZE.MODEL` TO THROW AN ERROR

`Sequelize.Model` is a reference to the prototype that all of our Models inherit from,



so we can easily add a new method `find` to `findById` and throw an error if it returns `null`:

```
// models.js
const Boom = require('boom')

Sequelize.Model.find = async function (...args) {
  const obj = await this.findById(...args)
  if (obj === null) throw Boom.notFound()

  return obj
}
```

We can then revert the handler to its former glory and replace occurrences of `findById` with `find`:

```
{
  method: 'GET',
  path: '/articles/{id}',
  handler: (request) => {
    return Article.find(request.params.id)
  }
}
```

```
http GET http://localhost:3000/articles/12345
http DELETE http://localhost:3000/articles/12345
```

*Boom.* We now get a 404 *Not Found* error whenever we try to fetch something from the database that doesn't exist. We've replaced our custom error checks with an easy-to-understand convention that keeps our code clean.

*Note: another popular tool for making requests to REST APIs is Postman. If you prefer a UI and ability to save common requests, this is a great option.*

## Path Parameters

The routing in Hapi is a little different from other frameworks. The route is selected on the *specificity* of the path, so the order you define them in doesn't matter.

- `/hello/{name}` matches `/hello/bob` and passes `'bob'` as the *name* param
- `/hello/{name?}` — the `?` makes name optional and matches both `/hello` and `/hello/bob`

- `/hello/{name*2}` — the `*` denotes multiple segments, matching `/hello/bob/marley` by passing `'bob/marley'` as the *name* param
- `/args*` matches `/any/route/imaginable` and has the lowest specificity.

## The Request Object

The request object that's passed to the route handler has the following useful properties:

- `request.params` — path params
- `request.query` — query string params
- `request.payload` — request body for JSON or form params
- `request.state` — cookies
- `request.headers`
- `request.url`

## Adding a Second Model

Our second model will handle comments on articles. Here's the complete file:

```
// models.js
const path = require('path')
const Sequelize = require('sequelize')
const Boom = require('boom')

Sequelize.Model.find = async function (...args) {
  const obj = await this.findById(...args)
  if (obj === null) throw Boom.notFound()

  return obj
}

const sequelize = new Sequelize(null, null, null, {
  dialect: 'sqlite',
  storage: path.join('tmp', 'db.sqlite')
})

const Article = sequelize.define('article', {
  title: Sequelize.STRING,
  body: Sequelize.TEXT
})
```

```

const Comment = sequelize.define('comment', {
  commenter: Sequelize.STRING,
  body: Sequelize.TEXT
})

// These associations add an articleId foreign key to our comments table
// They add helpful methods like article.getComments() and article.createComment
Article.hasMany(Comment)
Comment.belongsTo(Article)

// Create tables
Article.sync()
Comment.sync()

module.exports = {
  Article,
  Comment
}

```

For creating and deleting comments we can add nested routes under the article's path:

```

// routes.js
const { Article, Comment } = require('./models')

{
  method: 'POST',
  path: '/articles/{id}/comments',
  handler: async (request) => {
    const article = await Article.find(request.params.id)

    return article.createComment(request.payload.comment)
  }
}, {
  method: 'DELETE',
  path: '/articles/{articleId}/comments/{id}',
  handler: async (request) => {
    const { id, articleId } = request.params
    // You can pass options to findById as a second argument
    const comment = await Comment.find(id, { where: { articleId } })

    return comment.destroy()
  }
}

```

Lastly, we can extend GET `/articles/:id` to return both the article *and* its comments:

```

{
  method: 'GET',

```

```

    path: '/articles/{id}',
    handler: async (request) => {
      const article = await Article.find(request.params.id)
      const comments = await article.getComments()

      return { ...article.get(), comments }
    }
  }
}

```

article here is the *Model* object; `article.get()` returns a plain object with the model's values, on which we can use the spread operator to combine with our comments. Let's test it out:

```

http POST http://localhost:3000/articles/3/comments comment:='{ "commenter": "John", "text": "This is a great article!" }'
http POST http://localhost:3000/articles/3/comments comment:='{ "commenter": "John", "text": "This is a great article!" }'
http GET http://localhost:3000/articles/3
http DELETE http://localhost:3000/articles/3/comments/2
http GET http://localhost:3000/articles/3

```

Our blog API is almost ready to ship to production, just needing a couple of finishing touches.

## Authentication with JWT

JSON Web Tokens are a common authentication mechanism for APIs. There's a plugin `hapi-auth-jwt2` for setting it up, but it hasn't yet been updated for Hapi 17.0, so we'll need to install a fork for now:

```
npm install --save salzhrani/hapi-auth-jwt2#v-17
```

The code below registers the `hapi-auth-jwt2` plugin and sets up a *strategy* named `admin` using the `jwt` *scheme*. If a valid JWT token is sent in a header, query string or cookie, it will call our `validate` function to verify that we're happy to grant those credentials access:

```

// auth.js
const jwtPlugin = require('hapi-auth-jwt2').plugin
// This would be in an environment variable in production
const JWT_KEY = 'NeverShareYourSecret'

var validate = function (credentials) {
  // Run any checks here to confirm we want to grant these credentials access
}

```

◀ | | | ▶

```
// server.js
const { configureAuth } = require('./auth')

const main = async () => {
  await configureAuth(server)
  await configureRoutes(server)
  await server.start()

  return server
}
```

```
http GET localhost:3000/articles
http GET localhost:3000/articles Authorization:yep
http GET localhost:3000/articles Authorization:eyJ0eXAiOiJKV1QiLCJhbGciOiJ
```

```
{
  method: 'GET',
  path: '/articles',
  handler: (request) => {
```

```
    return Article.findAll()
  },
  config: { auth: false }
}
```

Make these three routes public so that anyone can read articles and post comments:

GET	/articles	articles#index
GET	/articles/:id	articles#show
POST	/articles/:id/comments	comments#create

## Generating a JWT

There's a package named `jsonwebtoken` for signing and verifying JWT:

```
npm install --save jsonwebtoken
```

Our final route will take an email / password and generate a JWT. Let's define our login function in `auth.js` to keep all auth logic in a single place:

```
// auth.js
const jwt = require('jsonwebtoken')
const Boom = require('boom')

exports.login = (email, password) => {
  if (!(email === 'mb4@gmail.com' && password === 'bears')) return

  const credentials = { email }
  const token = jwt.sign(credentials, JWT_KEY, { algorithm: 'HS256', expires: 1000000000 })

  return { token }
}
```

```
// routes.js
const { login } = require('./auth')

{
  method: 'POST',
  path: '/authentications',
  handler: async (request) => {
    const { email, password } = request.payload.login

    return login(email, password)
  },
}
```

```
    config: { auth: false }
  }
```

```
http POST localhost:3000/authentications login='{ "email": "mb4@gmail.com" }
```

Try using the returned token in your requests to the secure routes!

## Validation with `joi`

You can validate request params by adding config to the route object. The code below ensures that the submitted article has a body and title between three and ten characters in length. If a validation fails, Hapi will respond with a 400 error:

```
const Joi = require('joi')

{
  method: 'POST',
  path: '/articles',
  handler: (request) => {
    const article = Article.build(request.payload.article)

    return article.save()
  },
  config: {
    validate: {
      payload: {
        article: {
          title: Joi.string().min(3).max(10),
          body: Joi.string().required()
        }
      }
    }
  }
}
```

In addition to payload, you can also add validations to path, query and headers. Learn more about validation in the [docs](#).

## Who Is Consuming this API?

We could serve a single-page app from /. We've already seen — at the start of the tutorial — one example of how to serve an HTML file with streams. There are much

better ways of working with Views and Layouts in Hapi, though. See [Serving Static Content](#) and [Views and Layouts](#) for more on how to render dynamic views:

```
{
  method: 'GET',
  path: '/',
  handler: () => {
    return require('fs').createReadStream('index.html')
  },
  config: { auth: false }
}
```

If the front end and API are on the same domain, you'll have no problems making requests: `client -> hapi-api`.

If you're serving the front end from a *different* domain and want to make requests to the API directly from the client, you'll need to enable CORS. This is super easy in Hapi:

```
const server = Hapi.server({
  host: 'localhost',
  port: 3000,
  routes: {
    cors: {
      credentials: true
      // See options at https://hapijs.com/api/17.0.0#-routeoptionscors
    }
  }
})
```

You could also create a *new* application in between the two. If you go down this route, you won't need to bother with CORS, as the client will only be making requests to the front-end app, and it can then make requests to the API on the server without any cross-domain restrictions: `client -> hapi-front-end -> hapi-api`.

Whether that front end is another Hapi application, or Next, or Nuxt ... I'll leave that for you to decide!



# Chapter 7: Create New Express.js Apps in Minutes with Express Generator

BY PAUL SAUVE

**Express.js** is a Node.js web framework that has gained immense popularity due to its simplicity. It has easy-to-use routing and simple support for view engines, putting it far ahead of the basic Node HTTP server.

However, starting a new Express application requires a certain amount of boilerplate code: starting a new server instance, configuring a view engine, setting up error handling.

Although there are various starter projects and boilerplates available, Express has its own command-line tool that makes it easy to start new apps, called the `express-generator`.

## What is Express?

Express has a lot of features built in, and a lot more features you can get from other packages that integrate seamlessly, but there are three main things it does for you out of the box:

1. **Routing.** This is how `/home` `/blog` and `/about` all give you different pages. Express makes it easy for you to modularize this code by allowing you to put different routes in different files.
2. **Middleware.** If you're new to the term, basically middleware is “software glue”. It accesses requests before your routes get them, allowing them to handle hard-to-do stuff like cookie parsing, file uploads, errors, and more.
3. **Views.** Views are how HTML pages are rendered with custom content. You pass in the data you want to be rendered and Express will render it with your given view engine.

# Getting Started

Starting a new project with the Express generator is as simple as running a few commands:

```
npm install express-generator -g
```

This installs the Express generator as a global package, allowing you to run the `express` command in your terminal:

```
express myapp
```

This creates a new Express project called `myapp` which is then placed inside of the `myapp` directory.

```
cd myapp
```

If you're unfamiliar with terminal commands, this one puts you inside of the `myapp` directory.

```
npm install
```

`npm` is the default Node.js package manager. Running `npm install` installs all dependencies for the project. By default, the `express-generator` includes several packages that are commonly used with an Express server.

## OPTIONS

The generator CLI takes half a dozen arguments, but the two most useful ones are the following:

- **-v ejs|hbs|hjs|jade|pug|twig|vash.** This lets you select a view engine to install. The default is `jade`. Although this still works, it has been deprecated in favor of `pug`.
- **-c less|stylus|compass|sass.** By default, the generator creates a very basic CSS file for you, but selecting a CSS engine will configure your new app with middleware to compile any of the above options.

Now that we've got our project set up and dependencies installed, we can start the new server by running the following:

```
npm start
```

Then browse to `http://localhost:3000` in your browser.

## Application Structure

The generated Express application starts off with four folders.

### BIN

The `bin` folder contains the executable file that starts your app. It starts the server (on port 3000, if no alternative is supplied) and sets up some basic error handling. You don't really need to worry about this file because `npm start` will run it for you.

### PUBLIC

The `public` folder is one of the important ones: *everything* in this folder is accessible to people connecting to your application. In this folder, you'll put JavaScript, CSS, images, and other assets that people need when they connect to your website.

### ROUTES

The `routes` folder is where you'll put your router files. The generator creates two files, `index.js` and `users.js`, which serve as examples of how to separate out your application's route configuration.

Usually, here you'll have a different file for each major route on your website. So you might have files called `blog.js`, `home.js`, and/or `about.js` in this folder.

### VIEWS

The `views` folder is where you have the files used by your templating engine. The generator will configure Express to look in here for a matching view when you call the `render` method.

Outside of these folders, there's one file that you should know well.

The `app.js` file is special because it sets up your Express application and glues all of the different parts together. Let's walk through what it does. Here's how the file starts:

```
var express = require('express');
var path = require('path');
var favicon = require('serve-favicon');
var logger = require('morgan');
var cookieParser = require('cookie-parser');
var bodyParser = require('body-parser');
```

These first six lines of the file are `requires`. If you're new to Node, be sure to read [Understanding module.exports and exports in Node.js](#).

```
var routes = require('./routes/index');
var users = require('./routes/users');
```

The next two lines of code are requiring the different route files that the Express generator sets up by default: `routes` and `users`.

```
var app = express();
```

After that, we create a new app by calling `express()`. The `app` variable contains all of the settings and routes for your application. This object glues together your application.

```
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

Once the `app` instance is created, the templating engine is set up for rendering views. This is where you'd change the path to your view files if you wanted.

After this, you'll see Express being configured to use middleware. The generator installs several common pieces of middleware that you're likely to use in a web application.

```
//app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));
```

- **favicon.** This one is pretty self-explanatory: it just serves your `favicon.ico` out

of your public directory.

- **logger.** When you run your app, you might notice that all the routes that are requested are logged to console. If you want to disable this, you can just comment out this middleware.
- **bodyParser.** You might notice that there are two lines for parsing the body of incoming HTTP requests. The first line handles when JSON is sent via POST request and it puts this data in `request.body`. The second line parses query string data in the URL (e.g. `/profile?id=5`) and puts this in `request.query`.
- **cookieParser.** This takes all the cookies the client sends and puts them in `request.cookies`. It also allows you to modify them before sending them back to the client, by changing `response.cookies`.
- **express.static.** This middleware serves static assets from your `public` folder. If you wanted to rename or move the public folder, you can change the path here.

Next up is the routing:

```
app.use('/', routes);  
app.use('/users', users);
```

Here the example route files that were required are attached to our app. If you need to add additional routes, you'd do it here.

All the code after this is used for error handling. You usually won't have to modify this code unless you want to change the way Express handles errors. By default, it's set up to show the error that occurred in the route when you're in development mode.

## A Useful Tool

Hopefully you now have a clear idea of how the `express-generator` tool can save you time writing repetitive boilerplate when starting new Express-based projects.

By providing a sensible default file structure, and installing and wiring up commonly needed middleware, the generator creates a solid foundation for new applications with just a couple of commands.

# Chapter 8: An Introduction to MongoDB

BY MANJUNATH M

**MongoDB is an open-source, document-oriented, NoSQL database program. If you've been involved with the traditional, relational databases for long, the idea of a document-oriented, NoSQL database might indeed sound peculiar. "How can a database not have tables?", you might wonder. This tutorial introduces you to some of the basic concepts of MongoDB and should help you get started even if you have very limited experience with a database management system.**

## What's MongoDB?

Here's what the [official documentation](#) has to say about MongoDB.

MongoDB is an open-source database developed by MongoDB Inc. that's scalable and flexible. MongoDB stores data in JSON-like documents that can vary in structure.

Obviously, MongoDB is a database system. But why do we need another database option when the existing solutions are inexpensive and successful? A relational database system like MySQL or Oracle has tables, and each table has a number of rows and columns. MongoDB, being a document-oriented database system, doesn't have them. Instead, it has a JSON-like document structure that is flexible and easy to work with. Here's an example of what a MongoDB document looks like:

```
{
  "_id": ObjectId(3da252d3902a),
  "type": "Article",
  "title": "MongoDB Tutorial Part One",
  "description": "First document",
  "author": "Manjunath",
  "content": "This is an..."
}
```

```
{
  "_id": ObjectId("8da21ea3902a"),
  "type": "Article",
  "title": "MongoDB Tutorial Part Two",
  "description": "Second document",
  "author": "Manjunath",
  "content": "In the second..."
}
```

A document in a NoSQL database corresponds to a row in an SQL database. A group of documents together is known as a **collection**, which is roughly synonymous with a table in a relational database. For an in-depth overview of both NoSQL and SQL databases and their differences, we have that covered in the [SQL vs NoSQL tutorial](#).

Apart from being a NoSQL database, MongoDB has a few qualities of its own. I've listed some of its key features below:

- it's easy to install and set up
- it uses a BSON (a JSON-like format) to store data
- it's easy to map the document objects to your application code
- it claims to be highly scalable and available, and includes support for out-of-the-box replication
- it supports MapReduce operations for condensing a large volume of data into useful aggregated results
- it's free and open source.

MongoDB appears to overcome the limitations of the age-old relational databases and NoSQL databases. If you aren't yet convinced about why it's useful, check out our [article on Choosing Between NoSQL and SQL](#).

Let's go ahead and install MongoDB.

## Installing MongoDB

Installing MongoDB is easy and it doesn't require much configuration or setup. MongoDB is supported by all major platforms and distributions, and you can check out their documentation if you're in doubt.

I've covered the instructions for installing MongoDB on macOS below. For Linux, MongoDB recommends installing the software with the help of your [distribution's](#)

package manager. If you're on Windows, it's as easy as downloading the latest release from the MongoDB website and running the interactive installer.

## Installing MongoDB on macOS using Homebrew

Homebrew is a package manager for macOS, and this tutorial assumes you've already installed Homebrew on your machine.

1. Open the terminal application and run

```
$ brew update
```

1. Once brew is updated, install the mongodb package as follows

```
$ brew install mongodb
```

1. Create a data directory so that the `mongod` process will be able to write data. By default, the directory is `/data/db`:

```
$ mkdir -p /data/db
```

2. Make sure your user account has permission to read and write data into that directory:

```
$ sudo chown -R `id -un` /data/db  
> # Enter your password
```

3. Initiate the mongo server. Run the `mongod` command to start the server.
4. Start the Mongo shell. Open up another terminal window and run `mongo`. The mongo shell will attempt to connect to the mongo daemon that we initiated earlier. Once you're successfully connected, you can use the shell as a playground to safely run all your commands.
5. Exit the Mongo shell by running `quit()` and the mongo daemon by pressing `control + C`.



Now that we have the mongo daemon up and running, let's get acquainted with the MongoDB basics.

## Basic Database Operations

Enter the Mongo shell if you haven't already:

```
[mj@localhost ~]$ mongo
MongoDB shell version v3.6.1
connecting to: mongodb://127.0.0.1:27017
MongoDB server version: 3.6.1

connecting to: test
```

You can see my version of MongoDB shell when you log in. By default, you're connected to a test database. You can verify the name of the current database by running `db`:

```
> db
test
```

That's awesome, but what if we want a new database? To create a database, MongoDB has a `use DATABASE_NAME` command:

```
> use exampledb
switched to db exampledb
```

To display all the existing databases, try `show dbs`:

```
> show dbs

local          0.078GB
prototype      0.078GB
test           0.078GB
```

The `exampledb` isn't in the list because we need to insert at least one document into the database. To insert a document, you can do

`db.COLLECTION_NAME.insertOne({"key": "value"})`. Here's an example:

```
> db.users.insertOne({'name': 'Bob'})
{
  "acknowledged" : true,
```

```
"insertedId" : ObjectId("5a52c53b223039ee9c2daaec")
}
```

MongoDB automatically creates a new `users` collection and inserts a document with the key-value pair `'name' : 'Bob'`. The `ObjectId` returned is the id of document inserted. MongoDB creates a unique `ObjectId` for each document on creation and it becomes the default value of the `_id` field:

```
> show dbs
exampledb  0.078GB
local      0.078GB
prototype  0.078GB
test       0.078GB
```

Similarly, you can confirm that the collection was created using the `show collections` command:

```
> show collections
users
```

We've created a database, added a collection named `users` and inserted a document into it. Now let's try dropping it. To drop an existing database, use the `dropDatabase()` command as exemplified below:

```
> db.dropDatabase()
{ "ok" : 1 }
```

`show dbs` confirms that the database was indeed dropped.

```
> show dbs
local      0.078GB
prototype  0.078GB
test       0.078GB
```

For more database operations, see MongoDB reference page on [Database Commands](#).

## MongoDB CRUD Operations

As you might already know, the CRUD acronym stands for Create, Read, Update, and Delete. These are the four basic database operations that you can't avoid while building

an application. For instance, any modern application will have the ability to create a new user, read the user data, update the user information and, if needed, delete the user account. Let's accomplish this at the database level using MongoDB.

## CREATE OPERATION

Creation is the same as inserting a document into a collection. In the previous section, we had inserted a single document using the `db.collection.insertOne()` syntax. There's another method called `db.collection.insertMany()` that lets you insert multiple documents at once. Here's the syntax:

```
> db.COLLECTION_NAME.insertMany(
  [ <document 1> , <document 2>, ... ]
)
```

Let's create a `users` collection and populate it with some actual users:

```
> db.users.insertMany([
  { "name": "Tom", "age": 33, "email": "tom@example.com" },
  { "name": "Bob", "age": 35, "email": "bob@example.com" },
  { "name": "Kate", "age": 27, "email": "kate@example.com" },
  { "name": "Watson", "age": 15, "email": "watson@example.com" }
])

{
  "acknowledged" : true,
  "insertedIds" : [
    ObjectId("5a52cb2451dd8b08d5a22cf5"),
    ObjectId("5a52cb2451dd8b08d5a22cf6"),
    ObjectId("5a52cb2451dd8b08d5a22cf7"),
    ObjectId("5a52cb2451dd8b08d5a22cf8")
  ]
}
```

The `insertMany` method accepts an array of objects and, in return, we get an array of `ObjectIds`.

## READ OPERATION

Read operation is used to retrieve a document or multiple documents from a collection. The syntax for the read operation is as follows:

```
> db.collection.find(query, projection)
```

To retrieve all user documents, you can do this:

```
> db.users.find().pretty()
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf5"),
  "name" : "Tom",
  "age" : 33,
  "email" : "tom@example.com"
}
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf6"),
  "name" : "Bob",
  "age" : 35,
  "email" : "bob@example.com"
}
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf7"),
  "name" : "Kate",
  "age" : 27,
  "email" : "kate@example.com"
}
{
  "_id" : ObjectId("5a52cb2451dd8b08d5a22cf8"),
  "name" : "Watson",
  "age" : 15,
  "email" : "watson@example.com"
}
```

This corresponds to the `SELECT * FROM USERS` query from an SQL database.

The `pretty` method is a cursor method, and there are many others too. You can chain these methods to modify your query and the documents that are returned by the query.

What if you need to filter queries and return a subset of the collection? Say, find all users who are below 30. You can modify the query like this:

```
> db.users.find({ "age": { $lt: 30 } })
{ "_id" : ObjectId("5a52cb2451dd8b08d5a22cf7"), "name" : "Kate", "age" : 27, "email" : "kate@example.com" }
{ "_id" : ObjectId("5a52cb2451dd8b08d5a22cf8"), "name" : "Watson", "age" : 15, "email" : "watson@example.com" }
```

`$lt` is a query filter operator that selects documents whose `age` field value is less than 30. There are many comparison and logical query filters available, and you see the entire list in the [Query Selector documentation](#).

## UPDATE OPERATION

The update operation modifies the document in a collection. Similar to the create operation, MongoDB offers two methods for updating a document. They are:

1. `db.collection.updateMany(filter, update, options)`
2. `db.collection.updateMany(filter, update, options).`

If you need to add an extra field, say `registration`, to all the existing documents in a collection, you can do something like this:

```
> db.users.updateMany({}, {$set: { 'registration': 'incomplete'}})
{ "acknowledged" : true, "matchedCount" : 4, "modifiedCount" : 4 }
```

The first argument is an empty object because we want to update all documents in the collection. The `$set` is an update operator that sets the value of a field with the specified value. You can verify that the extra field was added using `db.users.find()`.

To update the value of documents that match certain criteria, `updateMany()` accepts a filter object as the first argument. For instance, you might want to overwrite the value of `registration` to `complete` for all users who are aged 18+. Here is what you can do:

```
> db.users.updateMany(
  {'age':{ $gt: 18 } },
  {$set: { 'registration': 'complete'}}
)

{ "acknowledged" : true, "matchedCount" : 3, "modifiedCount" : 3 }
```

To update the registration details of a single user, you can do this:

```
> db.users.updateOne(
  {'email': 'watson@example.com' },
  {$set: { 'registration': 'complete'}}
)

{ "acknowledged" : true, "matchedCount" : 1, "modifiedCount" : 1 }
```

## DELETE OPERATION

Delete operation removes a document from the collection. To delete a document, you can use the `db.collection.deleteOne(filter, options)` method, and to

delete multiple documents, you can use the `db.collection.deleteMany(filter, options)` method.

To delete documents based on certain criteria, you can use the filter operators that we used for the read and update operation:

```
db.users.deleteMany( { status: { $in: [ "dormant", "inactive" ] } } )

{ "acknowledged" : true, "deletedCount" : 1 }
```

This deletes all documents with a status of “dormant” or “inactive”.

That’s it.

## An Overview of MongoDB Drivers

For an application to communicate with the `mongodb-server`, you have to use a client-side library called a **driver**. The driver sits on top of the database server and lets you interact with the database using the driver API. MongoDB has official and third-party drivers for all popular languages and environments.

In this tutorial, we’re going to look at our options for drivers for Node.js. Some drivers add lots of good features, like like schema support and validation of business logic, and you can choose the one that matches your style. The popular drivers for Node.js include the native MongoDB driver and Mongoose. I’ll briefly discuss their features here.

### MONGODB NODE.JS DRIVER

This is the official MongoDB driver for Node.js. The driver can interact with the database using either promises or callbacks, and also supports the ES6 `async/await` functions. The example below demonstrates connecting the driver to the server:

```
const MongoClient = require('mongodb').MongoClient;
const assert = require('assert');

// Connection URL
const url = 'mongodb://localhost:27017/exampledb';
// Database Name
const dbName = 'exampledb';

(async function() {

  let client;
```

```

try {
  // Attempts to connect to the server
  client = await MongoClient.connect(url);

  console.log("Successfully connected to the server.");

  const db = client.db(dbName);
} catch (err) {
  //Log errors to console
  console.log(err.stack);
}

if (client) {
  client.close();
}
})();

```

The `MongoClient.connect` returns a promise, and any error is caught by the `try ... catch` block. You'll be implementing the CRUD actions inside the `try ... catch` block, and the API for that is similar to what we've covered in this tutorial. You can read more about it in the official documentation page.

## MONGOOSE DRIVER

Another popular Node.js driver for MongoDB is Mongoose. Mongoose is built on top of the official MongoDB driver. Back when Mongoose was released, it had tons of features that the native MongoDB driver didn't have. One prominent feature was the ability to define a schema structure that would get mapped onto the database's collection. However, the latest versions of MongoDB have adopted some of these features in the form of JSON schema and schema validation.

Apart from schema, other fancy features of Mongoose include Models, Validators and middlewares, the `populate` method, plugins and so on. You can read more about these in the Mongoose docs. Here's an example to demonstrate `mongoose.connect()` for connecting to the database:

```

var mongoose = require('mongoose');

// Connection URL
const url = 'mongodb://localhost:27017/exampleDb';

mongoose.connect(url);

var db = mongoose.connection;

```

```
db.on('error',
console.error.bind(console, 'connection error:'));
db.once('open', function() {
  // we're connected!
});
```

## Conclusion

MongoDB is a popular NoSQL database solution that suits modern development requirements. In this tutorial, we've covered the basics of MongoDB, the Mongo shell and some of the popular drivers available. We've also explored the common database operations and CRUD actions within the Mongo shell. Now it's time for you to head out and try what we've covered here and more. If you want to learn more, I recommend creating a REST API with MongoDB and Node to acquaint yourself with the common database operations and methods.