



DEVELOPING AN APP USING THE REST API AND WORDPRESS

BY TOM EWER

CONTENTS

CHAPTER 1: INTRODUCTION	4
A Brief Note on My Background	5
Why Now Is the Time to Embrace the REST API	6
What We'll Be Looking to Do in This ebook	6
Let's Get Started	7
CHAPTER 2: ADDING AND TESTING OUR DATA	8
Our Local WordPress Setup	9
Is This Thing On?	10
What Is the REST API Actually Doing?	11
Cranking Through Basic CRUD	12
Conclusion	15
CHAPTER 3: CHOOSING AND TESTING A FRONT END SOLUTION	16
JavaScript Is Eating the World	17
The Leading JavaScript Framework Contenders	18
Introducing React from Facebook	19
Is This Thing On? (Redux)	19
Performing A Basic React/REST API Test	21
Conclusion	22

CONTENTS

CHAPTER 4: CREATING OUR REACT-POWERED WORDPRESS SITE	23
What We're Trying to Build	24
Creating Our First Component	25
What Just Happened?	26
Sprinkling in Some Style	27
Adding Child Components	28
A Quick Word About Data In React	29
Talking To The WordPress REST API	30
Displaying External HTML Content and Adding Interaction With a Basic Button	31
Conclusion	32
CHAPTER 5: ADDING CUSTOM ENDPOINTS AND EXTRA TOUCHES	33
Introducing Custom Endpoints in the REST API	34
Adding Our Own Custom Endpoint	36
Tidying Our HTML Output From WordPress	38
Adding Some Slight Button Styling	41
Conclusion	42
CHAPTER 6: EXPLORING THE WEB WITH THIRD-PARTY APIS	43
The Wider Programmatic World Awaiting WordPress	44
Conclusion	45
ABOUT THE AUTHOR: TOM EWER	46

A man with a beard, wearing a plaid shirt, is looking at a laptop screen. The screen displays a line chart with a peak. The background is blurred, showing other people in a room. The text 'CHAPTER 1: INTRODUCTION' is overlaid on the image in white, bold, sans-serif font.

CHAPTER 1: INTRODUCTION

This ebook is geared toward the average WordPress user looking to get to grips with the next generation of the platform via a practical, exploratory project.

Excitement over the [REST API](#) has been building for what seems like an eternity, but we're still pretty much at the starting gate in terms of what it's actually going to mean for site owners and developers.

Though I've written extensively about the [potential implications](#) of the REST API, I've been holding back on actually diving in and getting my hands dirty. With the REST API finally [taxiing on the runway](#), now is a great time to grasp that nettle and really start digging into detail.

Over the course of this ebook, I'll be taking the new hotness for a spin by putting together a simple JavaScript app that uses the REST API to power its content. It promises to be an intense learning experience, but one that will hopefully serve others who are coming from a non-technical background as well.

A BRIEF NOTE ON MY BACKGROUND

Though WordPress has long been a passion of mine, I'm a writer and entrepreneur by trade. I can't pretend to be coming at this project from any type of serious development background. "Knows just about enough to be dangerous" would be how I'd charitably classify my coding experience to date.

So, this ebook won't be quite the deep dive you might expect from a theming professional [such as Jack Lenox](#), or a senior web developer [such as Ramsay Lanier](#). It should also be no great surprise that I'm not approaching this from the point of view of an established top-tier digital agency looking to [kick the tires of the latest technology](#).

With those caveats out of the way, let's briefly recap why now is a great time to roll up your sleeves and use the REST API in earnest.

The arrival of Calypso points the way to the future of WordPress.



WHY NOW IS THE TIME TO EMBRACE THE REST API

With the recent inclusion of REST API content endpoints in 4.7, and Matt Mullenweg's 2016 [State of the Word](#), it's made crystal clear which way the wind is blowing in the world of WordPress. To put it in a nutshell, the REST API is going to be at the center of the next stage of the platform's future, and developers are going to have to get on board with JavaScript sooner rather than later.

We've already seen [entire conferences](#) devoted to exploring the implications of the REST API, and [increasingly large real-world projects](#) basing themselves around it, despite its late arrival. From [Microsoft to the New York Times](#), blue-chip companies worldwide are chomping at the bit to explore its power.

If you're a theme or plugin developer, you can rest assured that the vast majority of your competition are already, at the very least, actively researching the topic. If you're a site owner, you can expect the next five years or more of your site's development to be significantly defined by the possibilities that the REST API opens up. No matter how you look at it, now is the time to get on board this particular jet.

WHAT WE'LL BE LOOKING TO DO IN THIS EBOOK

In this ebook, we're going to start from scratch with a local install, and use WordPress to house a collection of quotes from a great American original and author of [Walden](#) — [Henry David Thoreau](#). With our words of wisdom safely stored in the WordPress back end, we'll interact with them via the REST API, and build out a simple JavaScript-powered front end to display them in a variety of ways using Facebook's [React library](#).

Along the way, we'll touch on subjects such as alternative front end solutions, integration with mobile apps, design tips and tweaks, and experimenting with third-party APIs for added functionality. By the time we're finished, you should have a much more grounded and practical view of what the REST API is actually all about.


We'll be using [WordPress 4.7.3](#) running on a local development environment and [JavaScript React](#). It will also require a lot of patience and persistence to put together our finished project. Stick along for the ride and you're sure to pick up a ton of useful info along the way!

Aimed at a relatively non-technical audience, this ebook will teach users how to develop a REST API app from scratch and will take you from zero to hero in no time at all.

LET'S GET STARTED

The tools to get cracking with the REST API already exist, it's used in production by several major sites worldwide, and it won't be long until it hits the mainstream WordPress world in earnest. There's simply never been a better time to learn about it.

You won't need a computer science degree to follow along – just a little time, patience, and perseverance.

A person is shown from the side, interacting with a tablet. The tablet screen displays a data dashboard with various charts and tables. The entire image is overlaid with a teal color. The text 'CHAPTER 2: ADDING AND TESTING OUR DATA' is prominently displayed in white, bold, uppercase letters on the left side of the image.

CHAPTER 2: ADDING AND TESTING OUR DATA

It's now time to get down to business and start building the foundations of our app.

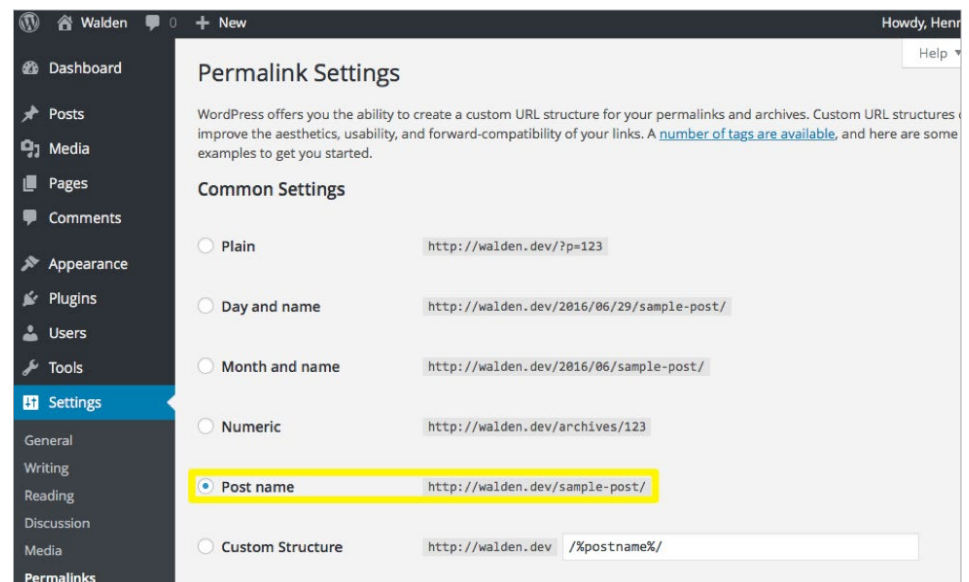
We'll kick things off by getting some core data into a local install of WordPress, and then start exploring some background concepts and the general set of options available to us. We'll do this by testing the basic reading and writing functionality of the REST API with the help of some handy tools which make it easy for non-technical users to follow along.

By the end of this chapter, we should understand what the REST API actually does, be confident we can interact with it locally, and be in good shape for taking things to the next level in future chapters. Let's start with a brief overview of the local setup being used here.

OUR LOCAL WORDPRESS SETUP

I'm running a fresh local install of [WordPress 4.7.3](#) on OS X with [Twenty Seventeen](#) installed as the theme — it's as vanilla as it gets. In honor of the man who will be providing most of the content we'll be working with, the local URL is <http://walden.dev/>.

I've also set my permalinks to use post names, as shown below.



For simplicity, I'm using the free version of [DesktopServer](#), which enables you to easily manage up to three local WordPress installs. It takes care of installing local web servers and a host of other potentially frustrating configuration issues that can occur with host files and the like behind the scenes.



DesktopServer helps keep things simple.

If you'd prefer to set up your local WordPress environment by hand, you can find instructions for doing so on Mac and Windows from [Nick Schäferhoff](#) on Torque. Fair warning: if this is your first time attempting to do this, be prepared to exercise a little patience and persistence if you run into problems!

I've also had a browse [over at Goodreads](#) and loaded a number of [Henry David Thoreau](#) quotes into the site as posts. So, at this stage, we have a site up and running, and some content loaded on to it. Let's move on to the REST API.

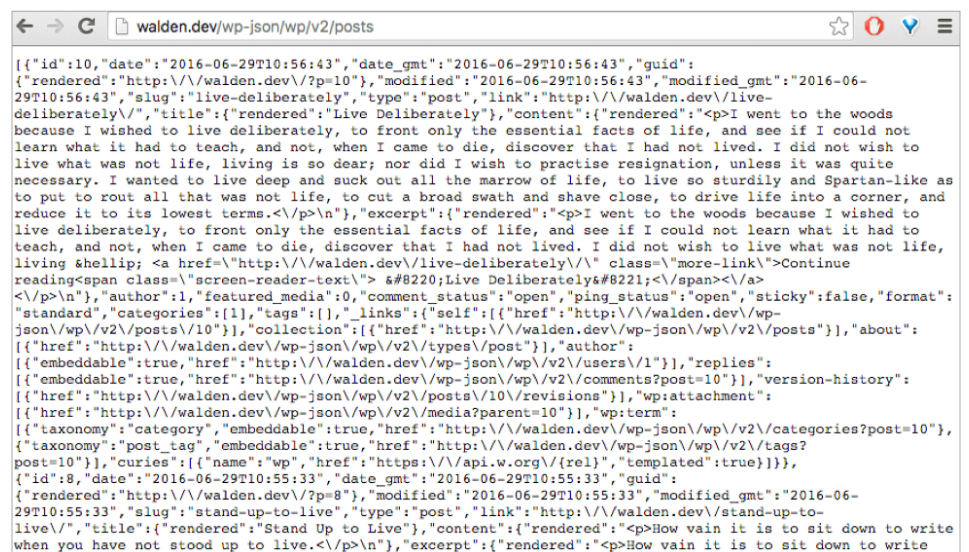
IS THIS THING ON?

We'll get into more technical detail in a second, but the first thing to take on board is that the REST API has opened up a route to our site's content. We should now be able to access data on the site directly via an HTTP request (i.e. if we type in the right URL in a browser, we should expect to see structured [JSON data](#) being returned).

Let's briefly test this in a quick and dirty way. Our local URL is <http://walden.dev/>. According to the [front page of the docs](#), it should be a piece of cake to return a list of posts.

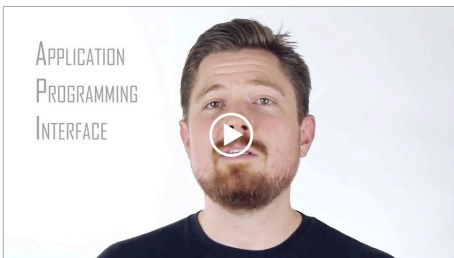
Want to get your site's posts? Simply send a `GET` request to `/wp-json/wp/v2/posts`. Update user with ID 4? Send a `POST` request to `/wp-json/wp/v2/users/4`. Get all posts with the search term "awesome"? `GET /wp-json/wp/v2/posts?search=awesome`. It's that easy.

Let's put that claim to the test and ask for a list of posts by typing in the address <http://walden.dev/wp-json/wp/v2/posts>.



There's an excellent overview of what both concepts mean in the video below.

Josh Pollock's ebook [The Ultimate Guide to the WordPress REST API](#) is another great resource if you want to dive deeper.



We have liftoff! It may look a bit of a mess in the browser, but we've successfully asked for information via the REST API, and received structured data in response. The REST API is active, and it's doing – at least on a very basic level – what it's supposed to. Now it's time to fill in some of the blanks around what we just did.

WHAT IS THE REST API ACTUALLY DOING?

Let's start with what a REST API actually is.

An [Application Programming Interface](#) (API) is simply a documented set of instructions for programmatically interacting with an application's data. That's a fancy way of saying that it enables one piece of software to talk to another. A REST API is a type of API that [follows certain rules](#) about getting data in and out of an application.

You interact with an API via an [HTTP web request](#) by sending a request to the server and getting a structured response back. Usually, your request is asking for some sort of action to be taken, and there are four standard things you might be looking to do:

1. POST (Create)
2. GET (Retrieve)
3. PUT (Update)
4. DELETE (Delete)

The acronym [CRUD](#) is typically used to describe this set of actions, and they cover the vast majority of things you might be trying to do on the average site or application. These actions are carried out by some type of resource – a thing or object that we want something to happen to.

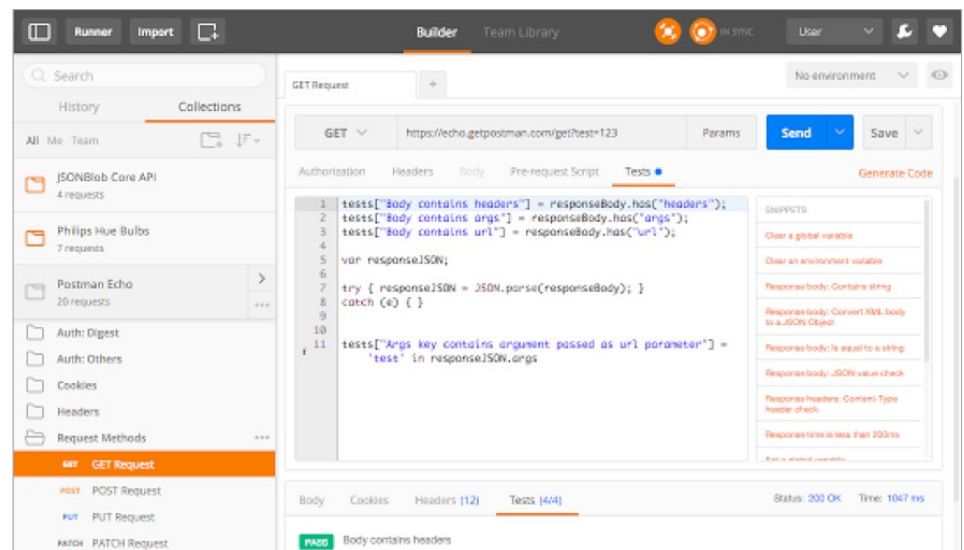
In the case of WordPress, the REST API currently enables us to interact with 12 different types of WordPress objects:

1. [Posts](#)
2. [Post Revisions](#)
3. [Pages](#)
4. [Media](#)
5. [Post Types](#)
6. [Post Statuses](#)
7. [Comments](#)
8. [Taxonomies](#)
9. [Categories](#)
10. [Tags](#)
11. [Users](#)
12. [Settings](#)

We'll be sticking almost exclusively to posts in this ebook, but as the documentation says, "chances are, if you can do it with WordPress, the WP API will let you do it." With that in mind, let's kick the tires a little further on our local site.

CRANKING THROUGH BASIC CRUD

In order to run some simple local tests, we're going to use the [Postman Chrome Extension](#). It's a handy tool that'll enable us to interact directly with the API without having to crank out any code (the [REST Easy](#) add-on does much the same on Firefox). We'll use Postman to quickly test some core CRUD functionality and make sure everything's working.

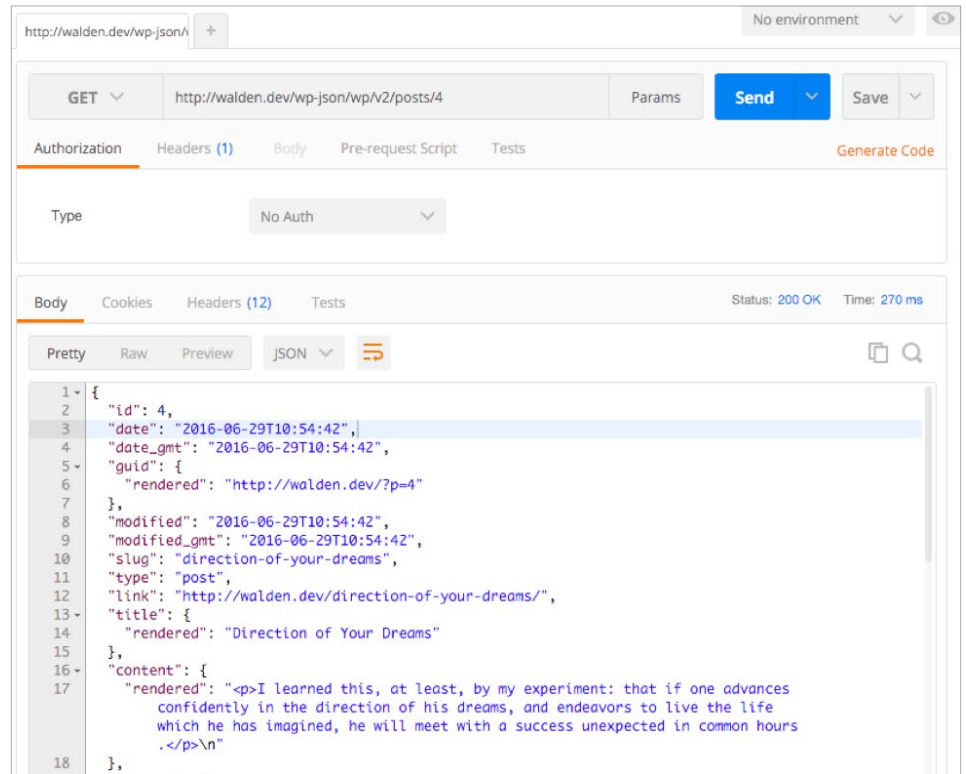


The [Postman Chrome Extension](#) helps us quickly test the REST API.

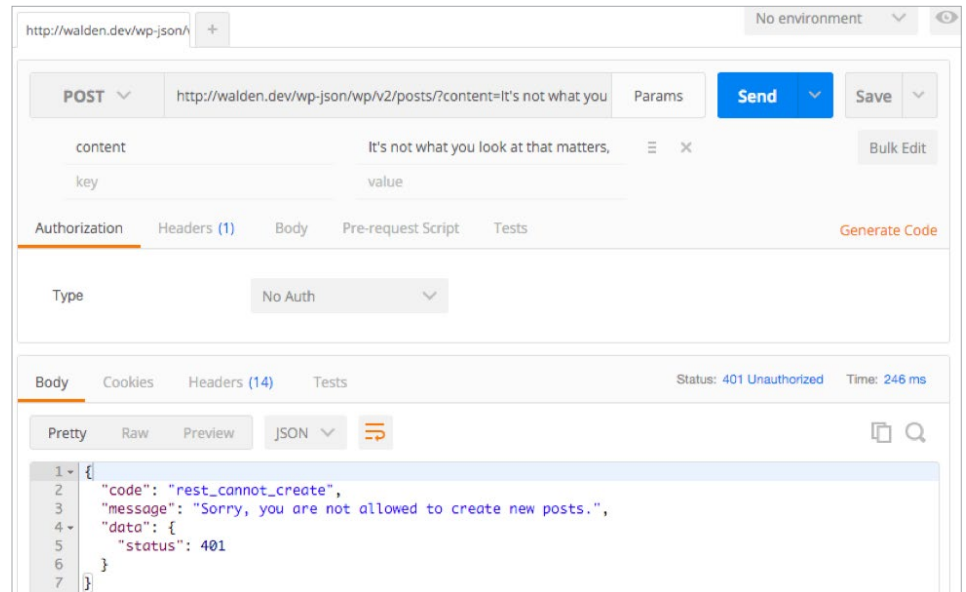
We know that the basic retrieve functionality is working, but let's double check by calling a single post. The *Retrieve a Post* section of the [reference documentation](#) shows us we can do this using a GET request and a post id: `GET /wp/v2/posts/<id>`. Post ID 4 happens to be a favorite of mine on the local site, so I've punched it into Postman:

Now let's look at creating a post. I'd like to add this pithy little number: "It's not what you look at that matters, it's what you see."

Presto! The REST API pops back some timeless wisdom and the various parts of the data are nicely broken out in the Postman interface.

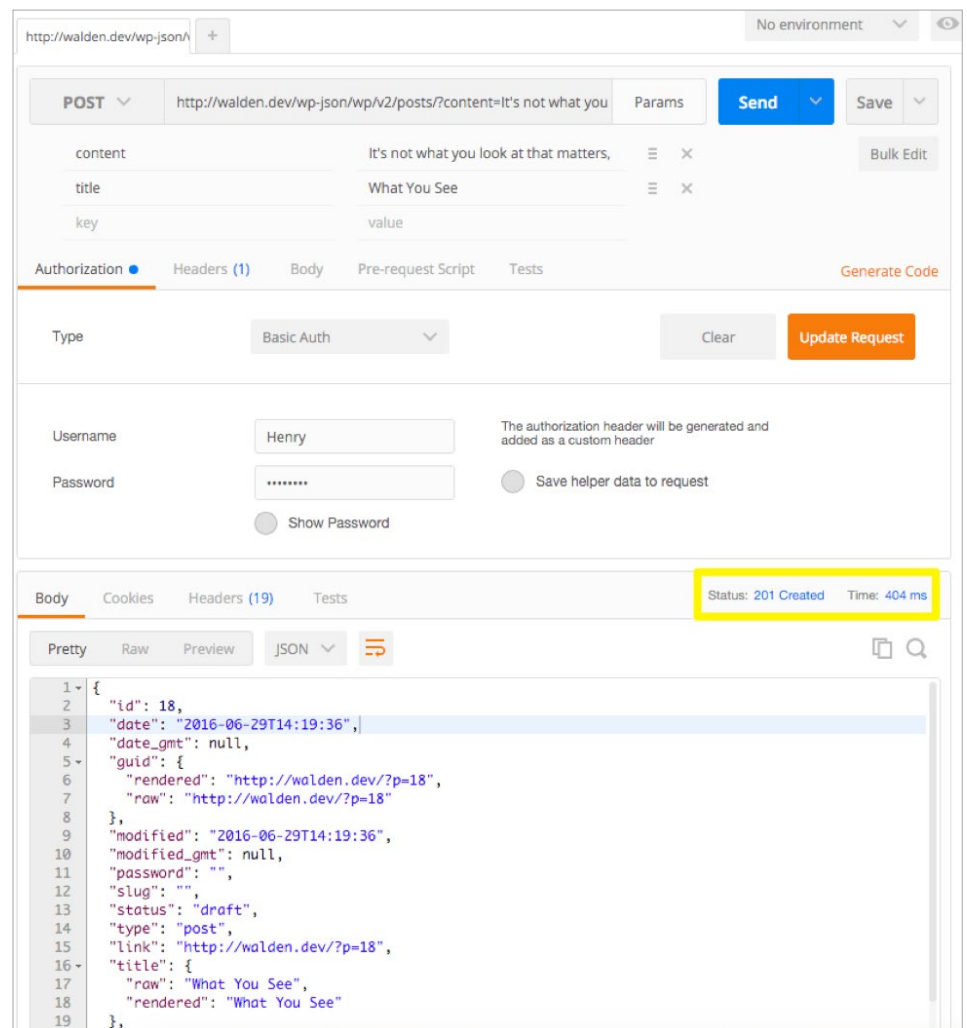


The eagle-eyed amongst you will have spotted some text saying *No Auth* under the *Authorization* tab in the screenshot above. For requests that involve ‘write’ operations (i.e. creating, updating, and deleting), we’re going to have to prove our credentials. If we try a quick test *without* authorizing, we’ll rightfully be shown the door:

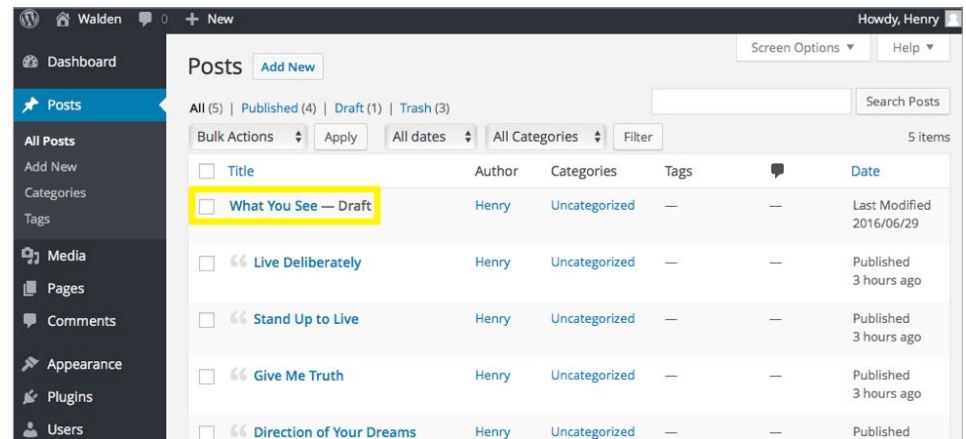


Authentication is a whole topic unto itself with the REST API, but to quickly test things locally, we'll be using the handy Basic Auth plugin to get us past the virtual bouncers.

This enables us to simply use our standard login details via Postman. In the instance below, I've passed through the content and title of a new quote via POST.



Things look good on the Postman side of things, but let's pop into the back end to make sure:

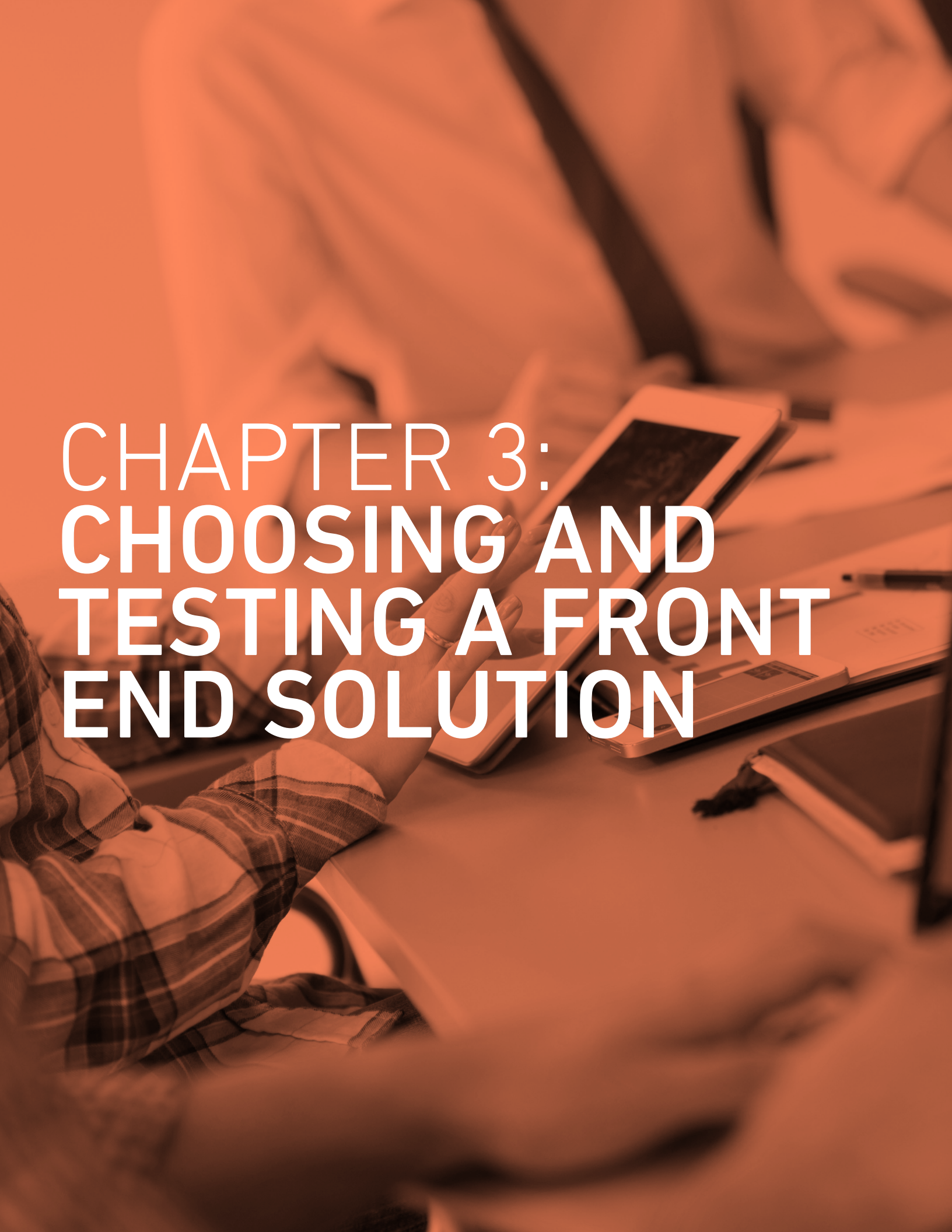


Hooray! It worked! You'll notice the status is set to Draft, as I forgot to specify it via the API call. I'll leave it as an exercise for the reader to work out how to change that via the API itself. The combination of Postman, the relevant [REST API posts documentation](#), and the Basic Auth plugin should enable you to poke around for a solution!

CONCLUSION

We've covered a lot of ground in a short space of time, and introduced some practical ways for more non-technical users to quickly interact with the REST API directly. Let's review what we have done so far:

1. Set up a simple local install and populated it with content.
2. Covered the basic concepts of REST APIs in general, and what this one can be used for in WordPress.
3. Confirmed that we can get data in and out of WordPress via the REST API

A person wearing a plaid shirt is sitting at a desk, interacting with a tablet. A laptop and a notebook are also on the desk. The entire scene is overlaid with a semi-transparent orange filter. The text 'CHAPTER 3: CHOOSING AND TESTING A FRONT END SOLUTION' is written in white, bold, sans-serif font across the center of the image.

CHAPTER 3: CHOOSING AND TESTING A FRONT END SOLUTION

In this chapter, it's time to consider how to handle things on the front end.

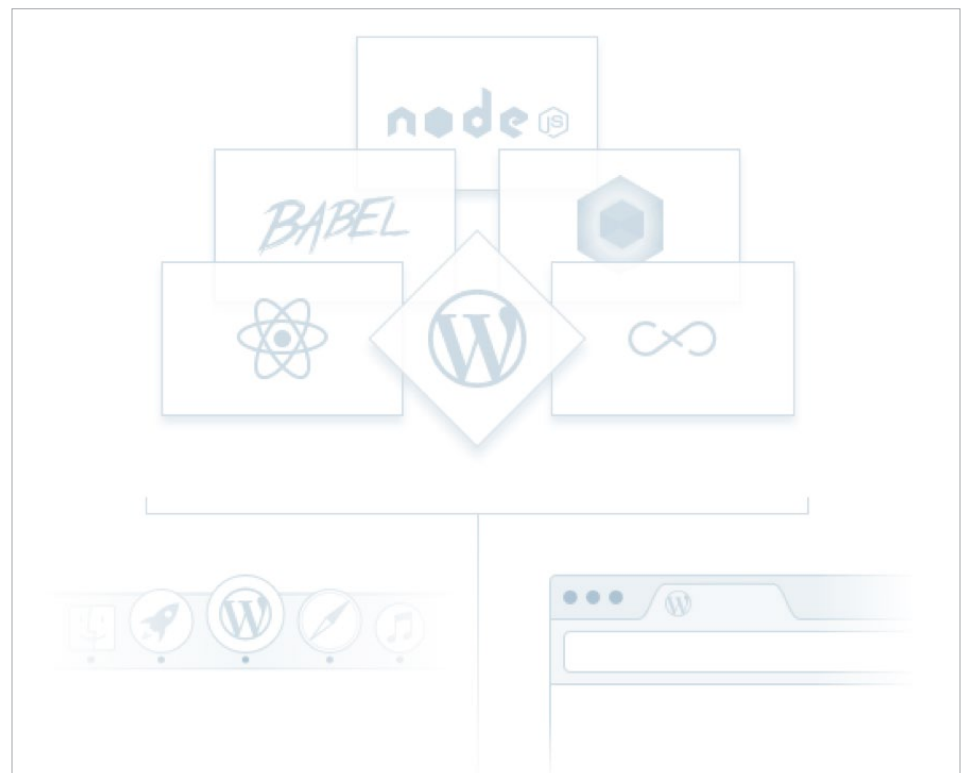
So far in this ebook, we've loaded some data into WordPress to play with and made sure we can [access it](#) via the REST API. Along the way, we've introduced a number of core concepts and simple tools you can use to start experimenting yourself, even if you're not a tech whizz.

We need a nice framework to pull data out of WordPress via the REST API and display it. It's almost certainly going to involve a JavaScript-powered solution. But which one?

That's exactly the question we'll be pondering below. We'll briefly survey the landscape of possible options, select one and introduce it, and then look at simple steps to get up and running. Let's start, though, with a quick reminder of why JavaScript makes sense in this context.

JAVASCRIPT IS EATING THE WORLD

JavaScript, as Kevin Lacker recently put it, is [eating the world](#). It's come a long way from its origins as a hastily put together [short-term solution at Netscape](#) and is now, by some measures, [the most popular programming language](#) in the global development community. In terms of front end web development, in particular, it's fair to say that JavaScript is effectively the *lingua franca* of the web these days.



Calypso is built on a modern JavaScript stack.

JavaScript is also blessed with a number of stable and mature front end frameworks that developers can use to work quickly and effectively.



It's also, as Matt Mullenweg was keen to stress in his [2015 State of the Word address](#), very much the future of WordPress. The recent [arrival of Calypso](#) dramatically pointed towards where the platform is heading – a stable and secure background WordPress core, housing data that's consumed by an ever-increasing set of wider external services, many of them powered by JavaScript.

Let's move on to look at some of the main ones.

THE LEADING JAVASCRIPT FRAMEWORK CONTENDERS

As with any other programming language, there's a huge amount to be said for sticking to [DRY](#) and [KISS](#) principles and using some sort of framework to take care of a lot of the heavy lifting when you're dealing with JavaScript.

A [baffling number of JavaScript frameworks](#) have winked in and out of existence over the last few years, but in terms of stability and active development, the practical choice here basically [boils down to one of four](#):

1. **Backbone.js** - Created by [Jeremy Ashkenas](#), [Backbone](#) was one of the first frameworks out of the starting gate back in 2010. Its combination of compactness and [flexibility](#) led to the early adoption by an [impressive list](#) of high-profile sites, and projects including [WordPress.com](#) and [WordPress core](#).
2. **Ember.js** - Where Backbone presents a deliberately stripped-down set of possibilities to build off, [Ember](#) is a substantially more ambitious and opinionated affair. The framework was created in 2011 by [Yehuda Katz](#) of [jQuery](#) and [Ruby on Rails](#) fame, and is designed to help developers tackle large-scale projects quickly and effectively. It can be seen in action [on a host of marquee sites](#) including [Discourse](#), [Groupon](#), and [LivingSocial](#).
3. **Angular.js** - Billing itself as a "Superheroic JavaScript MVW Framework", [AngularJS](#) is Google's dog in the framework fight. It offers a relatively easy learning curve and a thriving worldwide community of users. The framework's move from Version One to Version Two was [a little controversial](#), however, and it does require a bit of [performance-related hand-holding](#) when deployed at scale.
4. **React** - Arriving in 2013, Facebook's [React](#) is a more recent addition to the field but has quickly attracted an army of enthusiastic early adopters including (somewhat obviously) Facebook itself, Instagram, Flipboard, Netflix, and a host of others. React has won plaudits for its speed, comparative simplicity, and easy mobile integration in the form of [React Native](#).

The solution we'll be using to power our own humble app is, in fact, React. It's arguably the most modern of the bunch and promises an onboarding process that won't completely melt the minds of non-developers. With the ongoing dominance of services such as Facebook and Instagram who rely on it, it also won't be going out of fashion any time soon.



React can be used in conjunction with other solutions (such as Flux and Redux) to create complex, full-blown applications at truly enormous scale.

Let's move on to explore React in a little more depth.

INTRODUCING REACT FROM FACEBOOK

The first thing to clarify is that React is, strictly speaking, a library rather than a framework. As the [project homepage](#) states loud and clear, it's a "JavaScript library for building user interfaces." It also takes a different approach to the other three solutions we mentioned above.

Rather than trying to solve for every part of putting together a complete online application (as in, for example, Ember), React uses a component-based approach to focus heavily on the UI part of the problem.

It enables you to define highly modular UI components that live in their own discrete world and can be easily reused. To put it in slightly more formal terms, you can think of React [as the V in MVC](#).

If you're looking for a solid general introduction to React, the [project documentation](#) is nicely put together, with Pete Hunt's [Thinking in React](#) piece a particularly useful jumping-off point. The [React Fundamentals](#) course from [egghead.io](#) is also an excellent resource, as is the [Rapid React](#) course from [LearnCode](#).

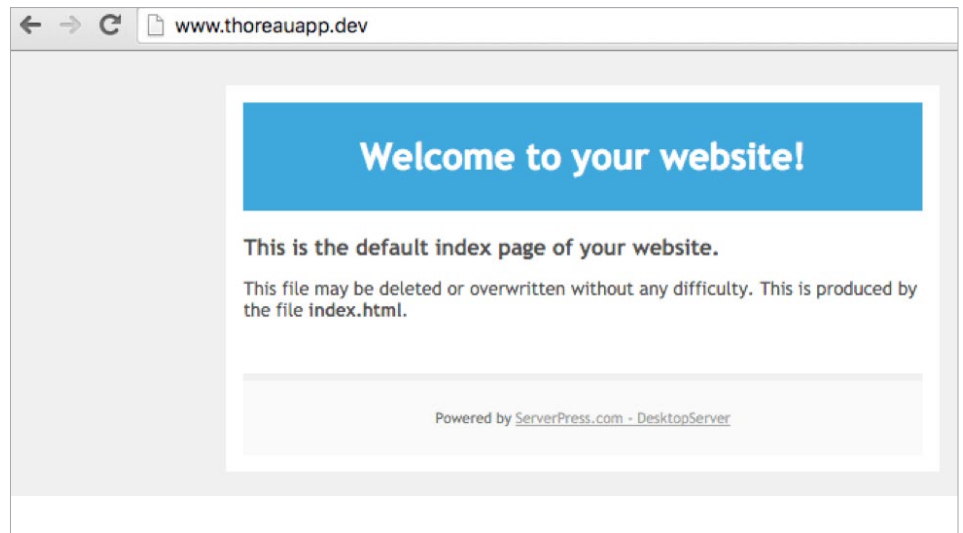
It can also be used to quickly build out native apps for both iOS and Android in the form of [React Native](#). We're not trying to build a [Doomsday Machine](#) here, however, so we'll be keeping things as vanilla as possible.

Let's start with seeing if we can get React running locally.

IS THIS THING ON? (REDUX)

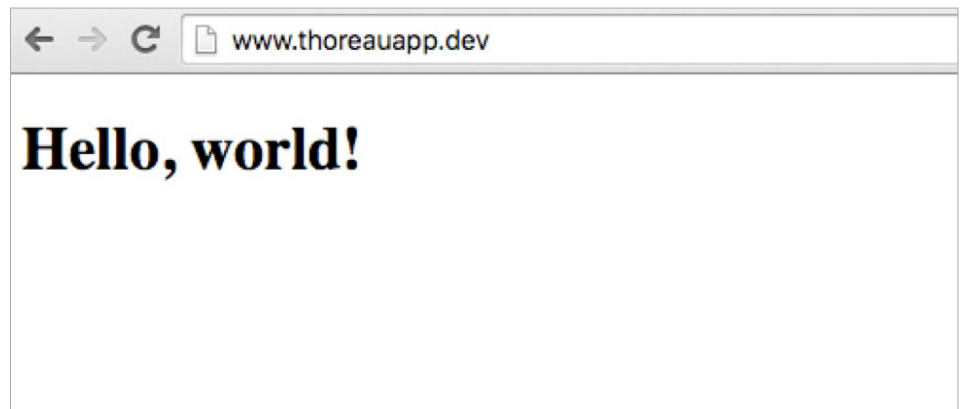
Have a quick look at many of the React tutorials online and you'll soon find yourself confronted with a wall of opinion around background tooling, involving things like [browserify](#), [Bower](#), and [webpack](#). These options are well explained in the React [package management pages](#), but we'll be looking to sidestep that rat's nest entirely by simply [downloading the React Starter Kit](#) locally.

The Starter Kit gives you a set of local files you can call directly via the browser. Using DesktopServer, I've created a new local site called [www.thoreauapp.dev](#) and simply copied the contents of Starter Kit into it.



Firing up our 'site' locally, we see just the default *index.html* page created by DesktopServer. I'm now going to replace the contents of that file with the *Hello World* script [from the Starter Kit documentation](#).

A quick refresh of the page and we see the following inspiring results:



It's not the most visually compelling page in the world, but it proves one crucial thing: we have React running locally, and it's capable of displaying content. Now let's see if we can get it talking to WordPress.

PERFORMING A BASIC REACT/REST API TEST

If you've read the previous chapter, you'll remember that we have a local REST API-enabled WordPress install running at *http://walden.dev/*. If we call *http://walden.dev/wp-json/wp/v2/posts* via HTTP, we'll get a full list of all posts in the install. I'm going to use that to do a very quick check that I can get React talking to WordPress via the REST API.

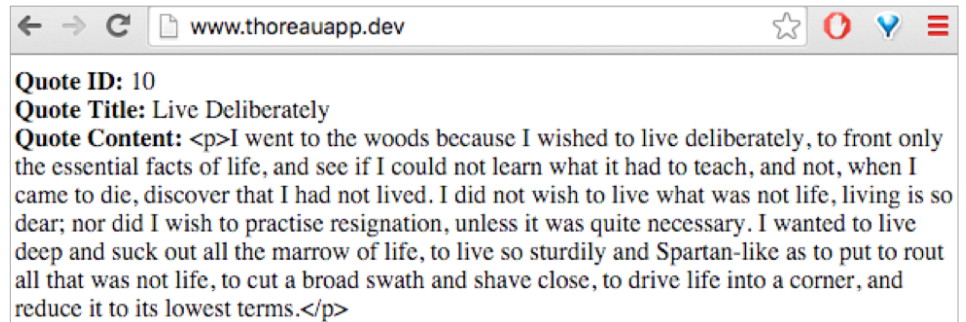
The code below is based on the React documentation [tips for loading external data](#). Don't worry about the ins and outs of it too much for now; I just want to see if we can get our separate components talking to each other by replacing my index.html file with the following code:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.0.0/
jquery.min.js"></script>
    <script src="build/react.js"></script>
    <script src="build/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">
      var ThoreauQuote = React.createClass({
        getInitialState: function() {
          return {
            quoteContent: '',
            quoteID: ''
          };
        },

        componentDidMount: function() {
          this.serverRequest = $.get(this.props.source, function
(result) {
            var firstQuote = result[0];
            this.setState({
              quoteID: firstQuote.id,
              quoteTitle: firstQuote.title.rendered,
              quoteContent: firstQuote.content.rendered
            });
          }.bind(this));
        },
        componentWillUnmount: function() {
          this.serverRequest.abort();
        },
        render: function() {
          return (
            <div>
              <b>Quote ID:</b> {this.state.quoteID}<br/>
              <b>Quote Title:</b> {this.state.quoteTitle}<br/>
              <b>Quote Content:</b> {this.state.quoteContent}<br/>
            </div>
          );
        }
      });
    </script>
  </body>
</html>
```

```
    );  
  }  
});  
ReactDOM.render(  
  <ThoreauQuote source="http://walden.dev/wp-json/wp/v2/posts" />,  
  document.getElementById('example')  
);  
</script>  
</body>  
</html>
```

Another quick refresh of the browser, *et voila!*



It's a deeply unimpressive visual result, but we've done something pretty major here – we've used React to call data from the REST API, parse it, and then display results on the screen. At this point, we can confidently say that we're in business!

CONCLUSION

JavaScript-powered front end solutions are set to be a huge part of the WordPress world very soon, so now is an excellent time to start exploring their potential. Having briefly outlined the other major contenders for building our simple app, and decided on React – a solution that's set to be around for many years to come.

We've also made quite a bit of progress towards our eventual goal. Previously, we knew that WordPress was merrily serving up content via the REST API. We also now know that we can get React running locally to consume and display that content.

A man wearing a plaid cap and glasses is drawing a wireframe on a whiteboard. The wireframe consists of several rectangular boxes of different sizes, some with arrows indicating dimensions. The dimensions are: a top box labeled '800 px', a box below it labeled '300 px', a box below that labeled '250 px', three boxes in a row at the bottom each labeled '200 px', and a wide box at the very bottom labeled '800 px'. The background is a teal-tinted image of the man.

CHAPTER 4: CREATING OUR REACT-POWERED WORDPRESS SITE

Though our test worked, we didn't exactly go to town on the details of how it was working. In this chapter, we'll look at a structured overview of putting together a simple React application and using it to display data. Let's get started.

In the previous chapter, we started looking at the front end part of the puzzle and selected [React](#) as the solution we'd be running with. The reasons why are simple: it's speedy, well-documented, and enjoys the support of one of the largest players in the business, Facebook.

In order to check that we could get WordPress and React talking to each other, we downloaded the React Starter Kit locally and cobbled together a quick API call to display some arbitrary data from our local WordPress install.

WHAT WE'RE TRYING TO BUILD

We'll be taking our inspiration here from Per Harold Borgen's [excellent introductory React article](#). If you're totally unfamiliar with React, it's also well worth checking out his [React.js in 8 Minutes](#) piece before going any further.

We'll be looking to put together a super simple, single-page app with three main moving parts: a random Thoreau quote (served up by WordPress), a suitable picture to accompany it, and a button that enables you to load another random quote.



I learned this, at least, by my experiment: that if one advances confidently in the direction of his dreams, and endeavors to live the life which he has imagined, he will meet with a success unexpected in common hours.

[Get more wisdom!](#)

Rather than thinking about this in terms of templates, we'll be using the concept of [components in React](#) to organize things. Looked at in those terms, we can break the picture above into four useful sub-components:

1. **ThoreauApp:** The component that will house everything.
2. **Picture:** Where our image will live.
3. **Quote:** Where we'll be displaying the best of Thoreau's musings.
4. **RandomButton:** An opportunity for users to load fresh wisdom.

Now, let's take it from the top and put together our first component.

CREATING OUR FIRST COMPONENT

Just a quick reminder here – we're working in an *index.html* page in the root directory of our local *http://www.thoreauapp.dev/* site. The [React Starter Kit](#) is in the same directory, and we're including relevant files in our header. You'll also notice an imaginatively named empty `<div>` on the page which we'll load our content into:

```
<!-- DOCTYPE HTML -->
<html>
  <head>
    <meta charset="UTF-8" />
    <title>React Thoreau App</title>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.0.0/
jquery.min.js"></script>
    <script src="build/react.js"></script>
    <script src="build/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-
core/5.8.34/browser.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
  </body>
</html>
```

We call the [createClass](#) method on the React object to define our first component. As you can see below, we're also passing in a specification object:

```
var ThoreauApp = React.createClass({
  render: function() {
    return (<div><p>A foolish consistency is the hobgoblin of
little minds.</p></div>);
  }
});
```

The specification object is where we'll define a number of things shortly, but we'll kick things off by creating a basic *render* method. This is what React will use each time it redraws the contents of the component.

If you excitedly refresh your browser page at this point, you'll be greeted (sadly) with absolutely nothing. That's because we've only defined a *potential* component at this point, we haven't brought it to life yet. Let's take care of that now by using `ReactDOM.render`:

```
ReactDOM.render(<ThoreauApp />, document.getElementById('content'));
```

We are telling `ReactDOM.render` two very important pieces of information:

1. The component it should render.
2. The area on the page it should render it within.

A quick refresh of the page, and we're greeted with the following inspiring sight:



Before we go any further, let's briefly take stock of what's going on.

WHAT JUST HAPPENED?

```
1 <!-- DOCTYPE HTML -->
2 <html>
3 <head>
4 <meta charset="UTF-8" />
5 <title>React Thoreau App</title>
6 <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.0.0/jquery.min.js"></script>
7 <script src="build/react.js"></script>
8 <script src="build/react-dom.js"></script> ← 1.
9 <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.34/browser.min.js"></
  script>
10 </head>
11 <body>
12 <div id="content"></div> ← 2.
13
14 <script type="text/babel"> ← 3.
15
16   var ThoreauApp = React.createClass({
17     render: function(){
18       return (<div><p>A foolish consistency is the hobgoblin of little minds</p></div>
19         );
20     }
21   });
22
23   ReactDOM.render(<ThoreauApp />, document.getElementById('content'));
24 </script> ← 5.
25
26 </body>
27 </html>
```

It's worth actually looking at the code in its entirety at this stage, just to be crystal clear about what we've actually done:

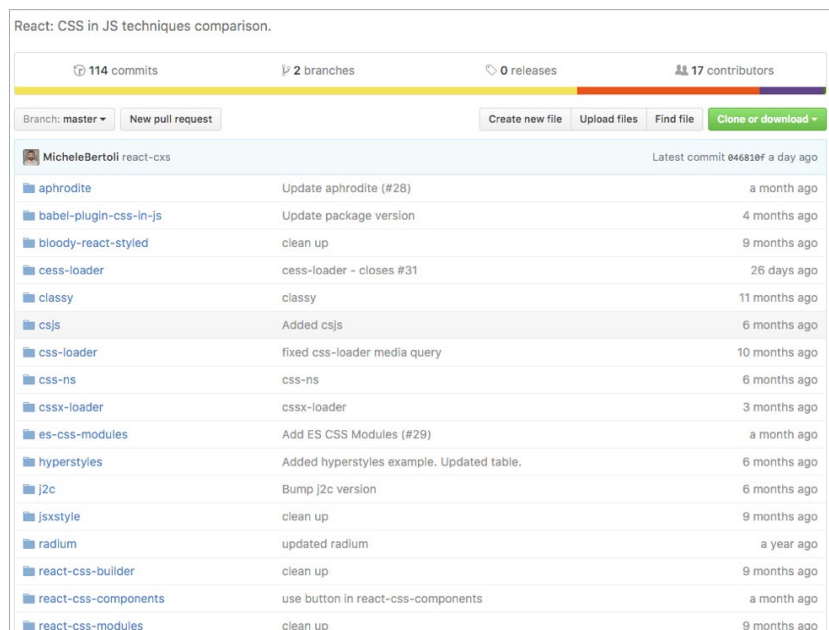
1. We loaded the relevant React libraries.
2. We have a defined area on the page to load our React content into.
3. We worked inside `<script>` tags to write our JavaScript/`JSX`.
4. We defined a component and gave it a *render* function.
5. We instantiated our component.

Now that we know roughly what's going on, let's take a brief detour to fancy things up visually.

SPRINKLING IN SOME STYLE

Our on-screen results so far are less than inspiring. It would be nice to throw at least *some* color and typography into the mix at this stage. But how? The simple answer here is by [using inline styles](#).

CSS purists will be horrified at the mere thought of this, but that pales in comparison with the despair others will feel when confronted with the [modern React CSS landscape](#).



Commit	Message	Time
MicheleBertoli react-css	Latest commit 046810f a day ago	
aphrodite	Update aphrodite (#28)	a month ago
babel-plugin-css-in-js	Update package version	4 months ago
bloody-react-styled	clean up	9 months ago
css-loader	css-loader - closes #31	26 days ago
classy	classy	11 months ago
csjs	Added csjs	6 months ago
css-loader	fixed css-loader media query	10 months ago
css-ns	css-ns	6 months ago
cssx-loader	cssx-loader	3 months ago
es-css-modules	Add ES CSS Modules (#29)	a month ago
hyperstyles	Added hyperstyles example. Updated table.	6 months ago
j2c	Bump j2c version	6 months ago
jsxstyle	clean up	9 months ago
radium	updated radium	a year ago
react-css-builder	clean up	9 months ago
react-css-components	use button in react-css-components	a month ago
react-css-modules	clean up	9 months ago

The tip of the React CSS iceberg.

There are, admittedly, all sorts of clever modular solutions (such as [Radium](#)) out there, but we're looking to keep things [as simple as possible](#) here.

With that in mind, we're going to throw some quick styles into the render function and get on with our lives:

```
var ThoreauApp = React.createClass({
  render: function(){
    var thoreauAppStyle = {
      backgroundColor: 'ffde00',
      color: '#333',
      padding: 20,
      width: 550,
      margin: '0 auto',
      fontFamily: 'Georgia',
      fontSize: 22,
      fontWeight: 'bold'
    }
    return (<div style={thoreauAppStyle}><p>A foolish consistency
is the hobgoblin of little minds.</p></div>);
  }
});
```

All of that leads to the following result:



We won't be winning any design awards, but it's enough to establish that we have some control. With that brief detour out of the way, let's break things out into components a bit more thoroughly.

ADDING CHILD COMPONENTS

Now that we've got something on the page – and a vague idea of one way to style elements – let's get a bit more organized. We'll start by creating our next two components. As you can see, the placeholder text is moved into *Quote*, and there's a placeholder image in play now courtesy of [lorempixel](#):

```
var Picture = React.createClass({
  render: function(){
    return (
      <div>
        <img src='http://lorempixel.com/550/350/' />
      </div>
    );
  }
});
```

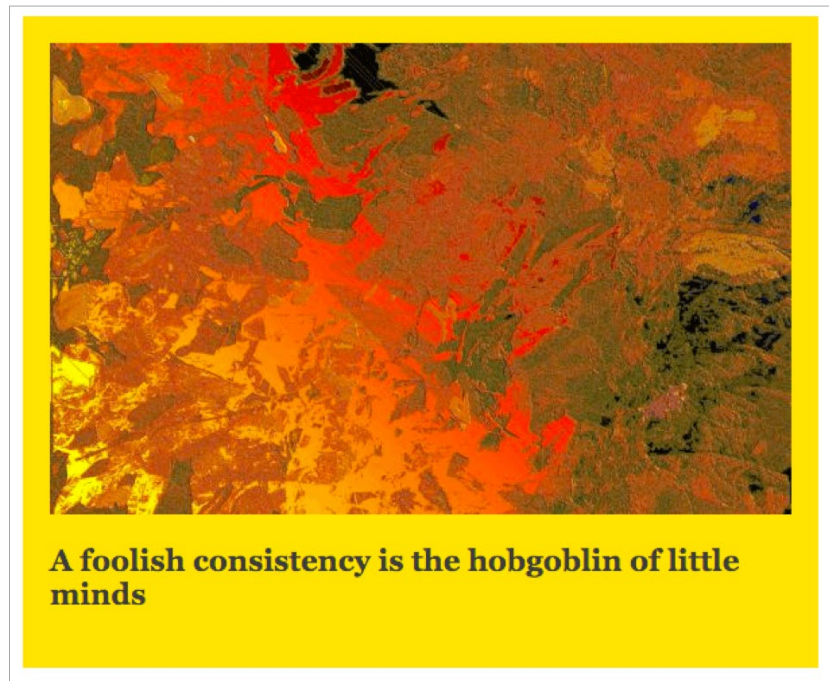
```
var Quote = React.createClass({
  render: function(){
    return (
      <p>A foolish consistency is the hobgoblin of little
minds.</p>
    );
  }
});
```

Again, we need to actually call these – this time by updating what we return from the *ThoreauApp* component:

```
return (
  <div style={thoreauAppStyle}>
    <Picture />
    <Quote />
  </div>
);
```

A quick refresh and we see the following:

It's looking a lot better! There are a lot of placeholders knocking around so far, though. Let's bring in some data.



A QUICK WORD ABOUT DATA IN REACT

Data generally in React can be handled as either 'state' or 'props'. These can be slippery concepts to grasp, so a close review of [Thinking in React](#) and the [relevant documentation](#) is recommended. The best thing about React is that changes to it will fire automatic updates to the relevant components on-screen down the line.

Once state arrives in the form of interactions, the general idea is that you want it to be handled once as high up the component chain as possible, and then move data around as props thereafter. That's broadly the approach we'll take when we hook up the REST API and introduce a button for users to play with.

TALKING TO THE WORDPRESS REST API

Let's get down to business! The first thing we'll do is make our app aware of where it can get data from:

```
ReactDOM.render(<ThoreauApp dataURL="http://walden.dev/wp-json/wp/v2/posts" />, document.getElementById('content'));
```

Before we look at loading the data, we'll make sure we're starting with a clean slate on each page load. `getInitialState` will be called once automatically on load and can be used to clear the decks:

```
getInitialState: function() {
  return {data: [], selectedQuote: ''};
},
```

Then we'll use `componentDidMount` to actually make the call to our WordPress API. In this instance, we're using `jQuery` to make the actual AJAX request. It'll store the results of the request inside the component using `setState`:

```
componentDidMount: function() {
  $.ajax({
    url: this.props.dataURL,
    dataType: 'json',
    cache: false,
    success: function(data) {
      this.setState({data: data});
      this.chooseRandomQuote();
    }.bind(this),
    error: function(xhr, status, err) {
      console.error(this.props.url, status, err.toString());
    }.bind(this)
  });
},
```

We'll also call another function here to pick out a random quote from the result set and assign it to a particular state:

```
chooseRandomQuote: function () {
  var randomNumber = Math.floor(Math.random() * this.state.data.length);
  var selectedQuote = this.state.data[randomNumber];
  this.setState({selectedQuote: selectedQuote.content.rendered});
},
```

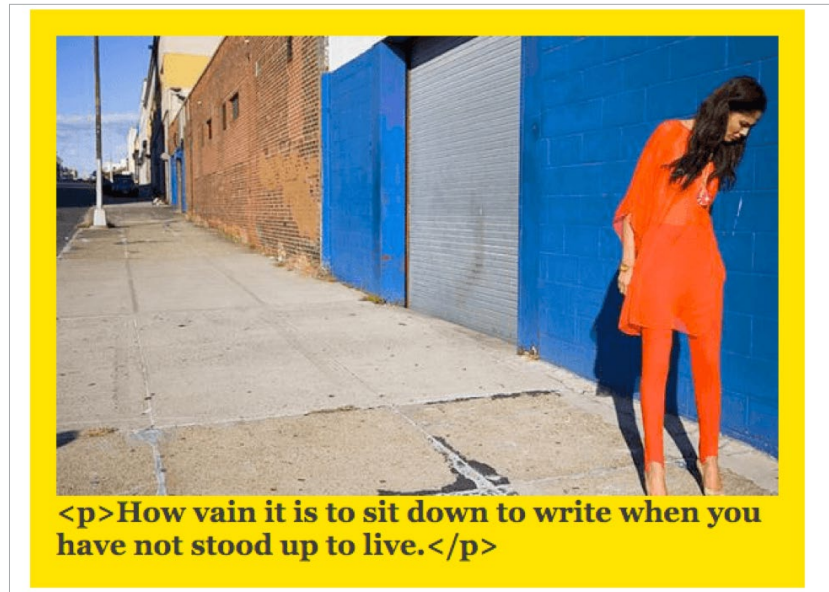
We now need to feed that information into our `Quote` component. We'll do that by passing it in as a prop called `quote`:

```
<Quote quote={this.state.selectedQuote} />
```

And then pick it up and display it inside the component:

```
<div>{this.props.quote}</div>
```

A quick refresh shows that we're *almost* there:



Let's finish things off.

DISPLAYING EXTERNAL HTML CONTENT AND ADDING INTERACTION WITH A BASIC BUTTON

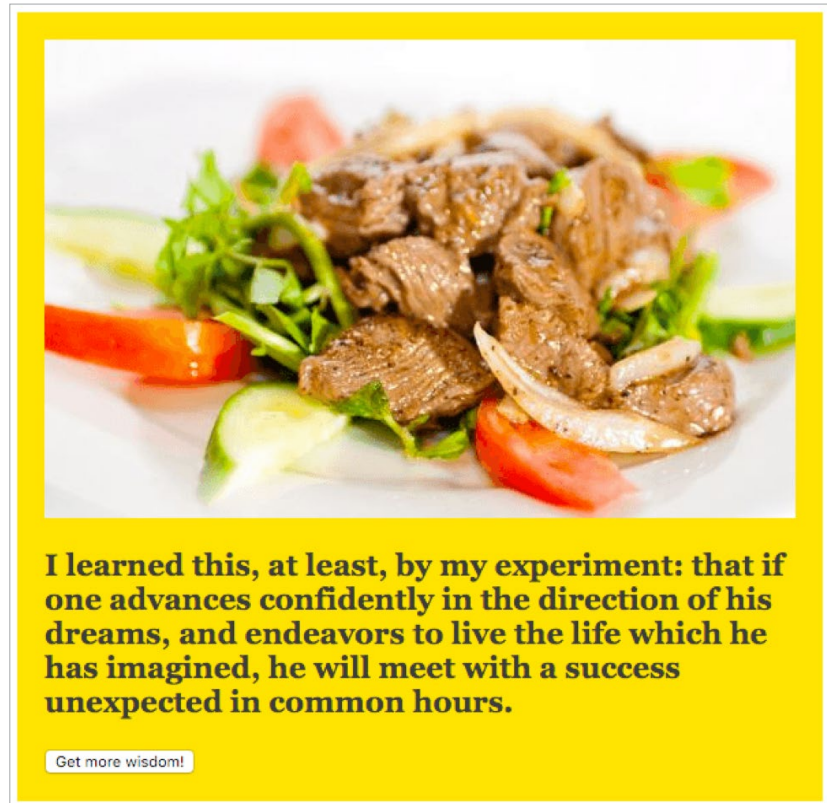
The eagle-eyed will have spotted that we have big dirty paragraph tags sitting there staring at us. This is actually by design in React, to avoid [cross-site scripting risks](#). We can get around it quickly by using the excitingly named [dangerouslySetInnerHTML](#) to get the job done:

```
<div dangerouslySetInnerHTML={{__html: this.props.quote }} />
```

Our final task in this chapter is to add a bit of basic interaction. We'll do this in the simplest way possible by adding in a quick button back in *ThoreauApp* that can call our earlier state-changing *chooseRandomQuote* function:

```
return (
  <div style={thoreauAppStyle}>
    <Picture
      imageURL='http://lorempixel.com/550/350/'
    />
    <Quote quote={this.state.selectedQuote} />
    <button onClick={this.chooseRandomQuote}>Get more wisdom!</
  button>
  </div>
```

Thanks to the magic of React, our button will now cause a re-render of the content (using the data it *already* has loaded) with a fresh quote each time it's pressed.



CONCLUSION

We've kept things as stripped down as possible in the example above, but hopefully, it's enough to walk you through the basics of React and offer several potential jumping off points for further self-study.

We've come a long way since Chapter 1! To recap, what we have so far is a one-page React app that:

1. Is split out into components.
2. Loads data from the WordPress REST API and displays a random result.
3. Enables users to display fresh content at the touch of a button, without needing further page reloads.

A close-up photograph of a person's hands working at a desk. One hand is typing on a laptop keyboard, while the other holds a white pen over a tablet. A smartphone is also visible on the desk. The scene is lit with a warm, golden light, and the background is slightly blurred.

CHAPTER 5: ADDING CUSTOM ENDPOINTS AND EXTRA TOUCHES

In Chapter 4, we stepped through building the basics of our app using [Facebook's React](#) and put together a simple solution where users could serve up nuggets of timeless wisdom on demand.

In this chapter, we'll concentrate on two main areas: adding a custom endpoint back in our WordPress site to make life a little easier when we're delivering large amounts of quotations, and adding some small extra touches on the front end in React.

As a quick reminder of the overall setup, so far we are running a local WordPress install at <http://walden.dev/> and serving up content via the [REST API](#). We're reading that content in a local React app running at <http://www.thoreauapp.dev/>. Okay – let's get down to business!

INTRODUCING CUSTOM ENDPOINTS IN THE REST API

So far we've kept things vanilla and simply used the REST API to return a list of posts that we've then parsed and displayed. We've also got the option, however, of extending the API and adding our own totally bespoke [custom endpoints](#).

As the documentation clearly outlines, this involves doing two things:

1. Creating a function in WordPress to handle our custom endpoint.
2. Registering a route to make that available via the REST API.

Let's check and see if the ["bare basics"](#) option outlined in the documentation actually works in our local setup. We'll pop into the [functions.php](#) file in our theme on <http://walden.dev/> and add the code below:

```
//Test adding basic REST API custom endpoint.
/**
 * Grab latest post title by an author!
 *
 * @param array $data Options for the function.
 * @return string|null Post title for the latest, * or null if none.
 */
function my_awesome_func( $data ) {
    $posts = get_posts( array(
        'author' => $data['id'],
    ) );

    if ( empty( $posts ) ) {
        return null;
    }

    return $posts[0]->post_title;
}
```

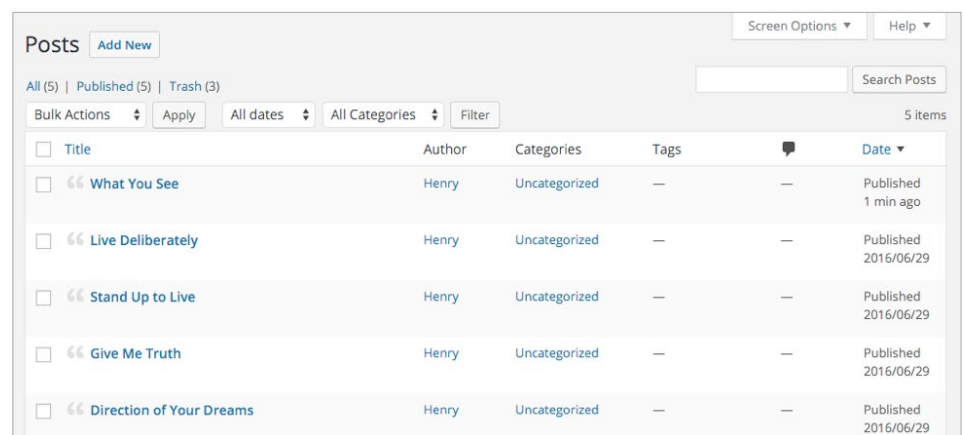
```
add_action( 'rest_api_init', function () {
    register_rest_route( 'walden/v1', '/author/(?P<id>\d+)', array(
        'methods' => 'GET',
        'callback' => 'my_awesome_func',
    ) );
} );
```

The only thing we've changed here from the sample code is using "walden" as the namespace. This code should use the `rest_api_init` hook to call `my_awesome_func`. If we feed it a valid author ID, we should expect to get the title of the first post by that author back.

In our case, we have just one author in the local WordPress setup, and his author ID is 1. Let's see what happens when we try calling the custom endpoint using the URL `http://walden.dev/wp-json/walden/v1/author/1`:



As with many of the examples so far, the on-screen results aren't breathtaking, but it establishes a key point – we can create custom endpoints and call them successfully. With a quick trip to our back end we can confirm that "What you see" is indeed the title of the first post, so we're getting the expected result.



It's worth noting here that there are several additional layers of complexity you'd add on in a real-world environment. The custom endpoints documentation does a great job of building out the basic example we've used here, so we'll simply [point you in that direction](#) for more details.

Now, let's move on to creating a useful custom endpoint for our existing app.

ADDING OUR OWN CUSTOM ENDPOINT

We'll be taking a cue from the excellent [Delicious Brains React Native REST API](#) tutorial, and streamline our data loading using a custom endpoint. As you may remember from the last chapter, we're currently loading in *all* of our data, and then displaying random individual quotes using React.

That's not a big problem considering we've only got a few quotes to deal with, but what if we had thousands? Things could get dicey quickly. What we'll do instead is use custom endpoints to get a list of all our post IDs, pick a randomized one from that list, and load in that data on its own each time.

Let's start by setting up the custom endpoint. The code below is based on a mix of the REST API documentation example and the Delicious Brains tutorial. Again, it's at the bottom of *functions.php* in our active theme's directory:

```
// Return all post IDs
function walden_get_all_post_ids() {

    $all_post_ids = get_posts( array(
        'numberposts' => -1,
        'post_type'    => 'post',
        'fields'       => 'ids',
    ) );

    return $all_post_ids;
}

// Add Walden/v1/get-all-post-ids route
add_action( 'rest_api_init', function () {
    register_rest_route( 'walden/v1', '/get-all-post-ids/', array(
        'methods' => 'GET',
        'callback' => 'walden_get_all_post_ids',
    ) );
} );
```

When we call `http://walden.dev/wp-json/walden/v1/get-all-post-ids`, we're now greeted with a nice clean array of all our post IDs:



Let's now make some quick changes in our React code to use that data instead of what we were previously doing. We'll start by passing in the new custom endpoint along with our previous REST API post's URL:

```
ReactDOM.render(<ThoreauApp dataURL="http://walden.dev/wp-json/wp/v2/posts/" idURL="http://walden.dev/wp-json/walden/v1/get-all-post-ids"/>, document.getElementById('content'));
```

We'll then move our existing code around to call for a list of IDs first, and then select a random one from that list to actually load an individual quote directly from the REST API:

```
getAllIDs: function() {
  console.log('getAllIDs called');
  $.ajax({
    url: this.props.idURL,
    dataType: 'json',
    cache: false,
    success: function(data) {
      this.setState({data: data});
      this.chooseRandomQuote();
    }.bind(this),

    error: function(xhr, status, err) {
      console.error(this.props.url, status, err.toString());
    }.bind(this)
  });
},

chooseRandomQuote: function () {
  var randomNumber = Math.floor(Math.random() * this.state.data
length);
  var selectedQuote = this.state.data[randomNumber];
  this.setState({selectedQuoteID: selectedQuote});
  this.getQuote();
},

getQuote: function(){
  $.ajax({
    url: this.props.dataURL + this.state.selectedQuoteID,
    dataType: 'json',
    cache: false,
    success: function(data) {
      this.setState({selectedQuoteContent: data.content.
rendered});
    }.bind(this),
    error: function(xhr, status, err) {
      console.error(this.props.url, status, err.toString());
    }.bind(this)
  });
},

componentDidMount: function() {
  this.getAllIDs();
},
```

We'll also make a slight change to what's passed into our *Quote* component to pick up on the changes above:

```
<Quote quote={this.state.selectedQuoteContent} />
```

A quick refresh to the main app and all appears to be well. We still have quotes being served from the REST API and the ability to load new ones, however, they're now using custom endpoints to get the job done.



It's worth pointing out at this stage that we've very much taken [the happy path](#) so far. Experienced React or WordPress developers may well have a *slew* of objections to everything from the coding style on display to the general lack of error handling, loading messages, and so on.

We're going to leave the majority of those potential improvements as exercises for the reader to tackle, but there is one minor item to address while we're at this stage.

TIDYING OUR HTML OUTPUT FROM WORDPRESS

At this stage, we're still using that slightly ominous sounding [dangerouslySetInnerHTML](#) method down in our *Quote* component. It would be ideal if we could take care of that bit of business [back in WordPress](#) rather than handling it somewhat awkwardly in React.

```
var Quote = React.createClass({
  render: function() {
    return (
      <div dangerouslySetInnerHTML={{__html: this.props.quote
    }} />
    );
  }
});
```

Fortunately, this is relatively easy to accomplish by adding an extra field on existing endpoints using `register_rest_field` as described in the [REST API documentation](#). Again leaning on the [Delicious Brains tutorial](#) we mentioned earlier, we've rejigged our existing code in `functions.php` below to add an extra field in post responses containing a nice, friendly, plaintext version of our quotes:

```
// Return plaintext content for posts
function walden_return_plaintext_content( $object, $field_name,
$request ) {
  return strip_tags( html_entity_decode( $object['content']
['rendered'] ) );
}

add_action( 'rest_api_init', 'setup_rest_route' );
add_action( 'rest_api_init', 'add_plaintext_response' );

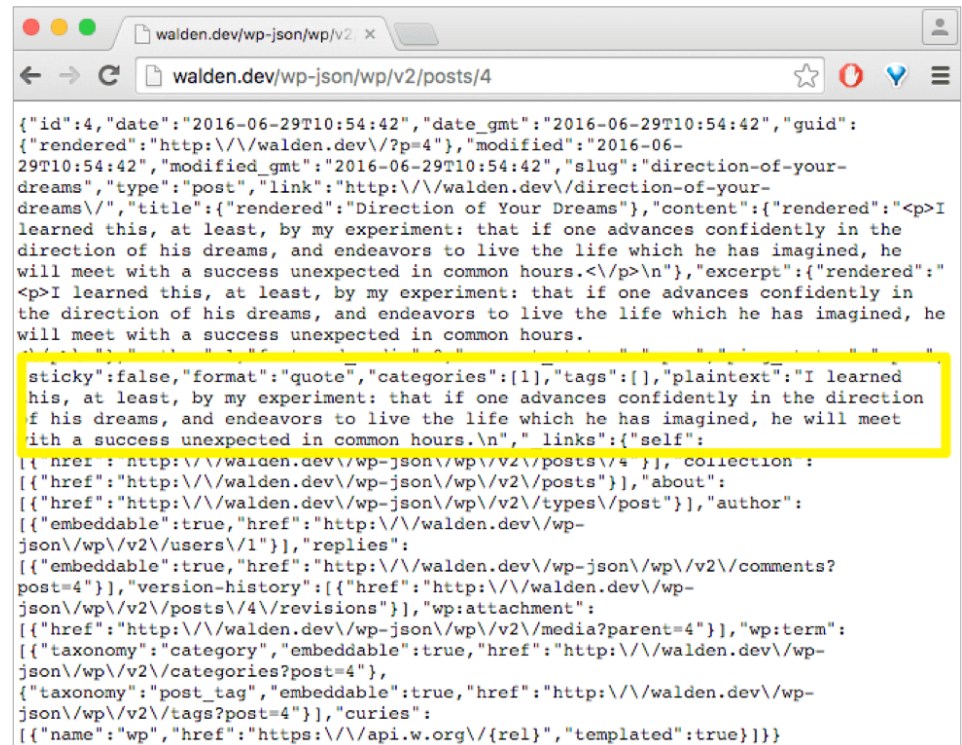
// Add Walden/v1/get-all-post-ids route
function setup_rest_route(){
  register_rest_route( 'walden/v1', '/get-all-post-ids/', array(
    'methods' => 'GET',
    'callback' => 'walden_get_all_post_ids',
  ) );
}

function add_plaintext_response() {
  // Add the plaintext content to GET requests for individual posts
  register_rest_field(
    'post',
    'plaintext',
    array(
      'get_callback' => 'walden_return_plaintext_content',
    )
  )
}
```

Just to be on the safe side, let's make sure our custom endpoint is still working at `http://walden.dev/wp-json/walden/v1/get-all-post-ids`:



Now let's see what happens if we call one post in particular with `http://walden.dev/wp-json/wp/v2/posts/4`:



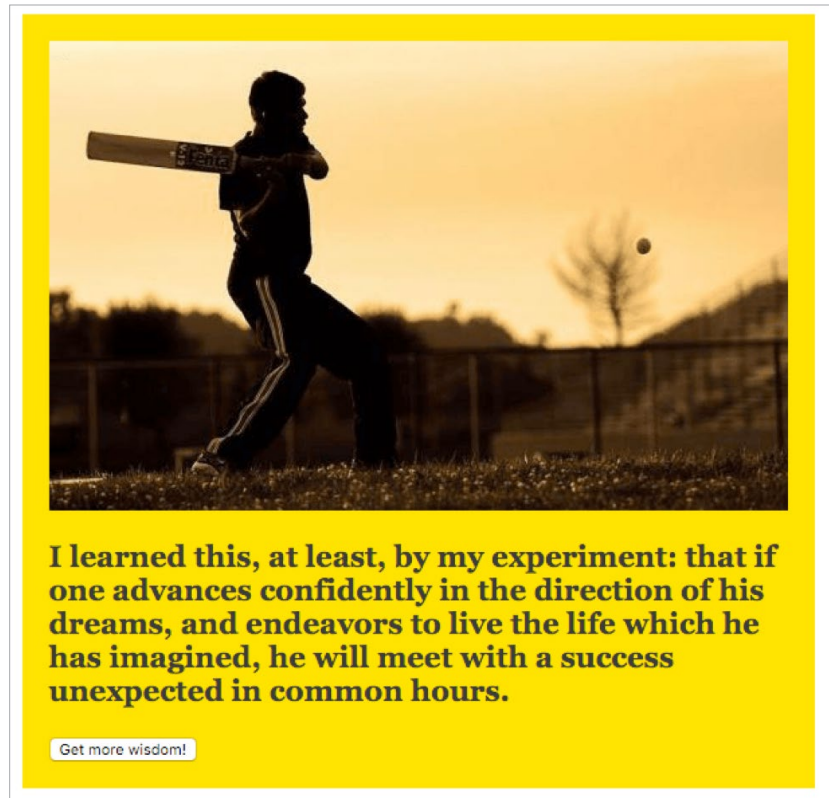
Happy days! We've now got clean text in our response. Let's use it directly in our `getQuote` function...

```
getQuote: function() {
  $.ajax({
    url: this.props.dataURL + this.state.selectedQuoteID,
    dataType: 'json',
    cache: false,
    success: function(data) {
      this.setState({selectedQuoteContent: data.plaintext});
    }.bind(this),
    error: function(xhr, status, err) {
      console.error(this.props.url, status, err.toString());
    }.bind(this)
  });
},
```

...and get rid of that worrying `dangerouslySetInnerHTML` method we were using:

```
var Quote = React.createClass({
  render: function() {
    return (
      <div><p>{this.props.quote}</p></div>
    );
  }
});
```


A final check of our front end and all appears to be well.



The “Get more wisdom!” button still looks visually unappealing, though. Let’s throw in a final bit of styling there.

ADDING SOME SLIGHT BUTTON STYLING

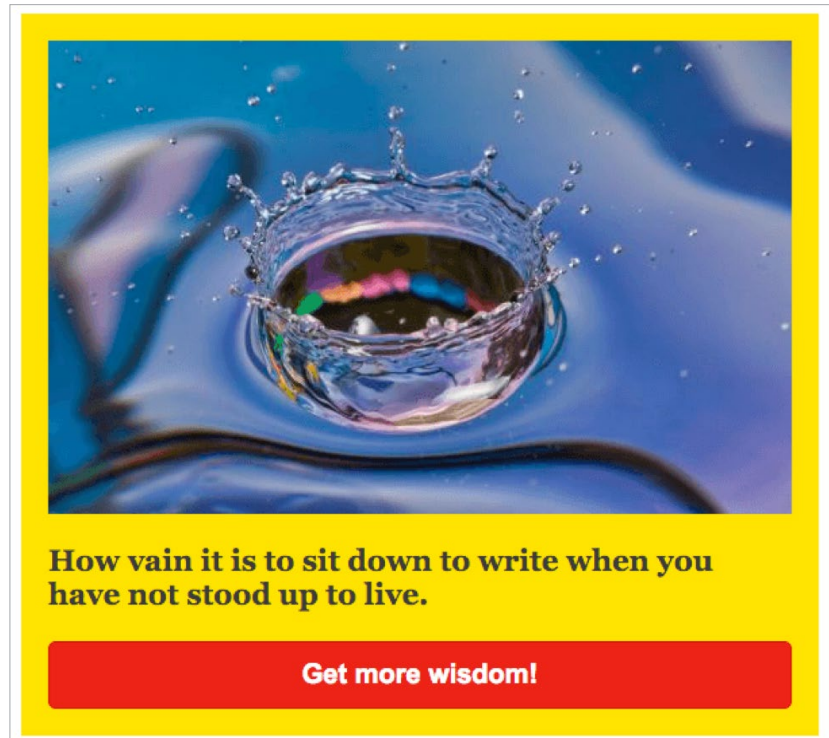
We’ll keep this last part nice and simple. First of all, we’ll add a new style object...

```
var buttonStyle = {
  height: 50,
  width: '100%',
  fontFamily: 'Arial',
  border: '1px solid #CF1111',
  borderRadius: 5,
  fontSize: 20,
  fontWeight: 'bold',
  textAlign: 'center',
  color: '#fff',
  backgroundColor: '#E82020'
}
```

...and then pass it into our button:

```
<button style={buttonStyle} onClick={this.chooseRandomQuote}>Get more wisdom!</button>
```

One final refresh on the front end, and we're done and dusted!



CONCLUSION

We haven't made *enormous* strides on the front end this time, but we've introduced a number of key concepts, and made life slightly easier for ourselves behind the scenes as a result.

The code samples shown here are very much just jumping-off points for further exploration, rather than the type of thing you'd sling into a real client-facing project.

There are two key takeaways to focus on this time if you're looking to explore further yourself:

1. Custom API endpoints can easily be added to your applications.
2. You're also free to add extra fields to standard endpoints if required.



CHAPTER 6:
EXPLORING THE
WEB WITH THIRD-
PARTY APIS

We've come a long way. From pretty much a standing start, we've managed to introduce the basic concepts of the technology, and get a small interactive app up and running (with a little help from [React](#)).

Along the way, we've proven that even non-technical WordPress users can start getting their feet wet with the programmatic power behind the next generation of the platform. The future of WordPress is not going to be about tiny Thoreau quote apps, however. It's going to largely center around how the REST API helps WordPress smoothly integrate with the rest of the online world.

To close out our ebook, let's take a peek at what that type of integration might look like.

THE WIDER PROGRAMMATIC WORLD AWAITING WORDPRESS

If you've used WordPress for any length of time at all, you've almost certainly taken advantage of existing third-party integrations in the form of plugins. You might have [added MailChimp](#) into the mix, experimented with [Google Analytics options](#), or taken advantage of any number of the other popular third-party app plugins that are out there.

Though the sophistication of many of these solutions to date has been impressive, they've always been held back by a fundamental problem on the WordPress side of the equation. From a technical perspective, the lack of a stable and reliable native API in WordPress has historically put developers on shaky ground in terms of stability, testing, and adding new features.



The REST API effectively removes those obstacles and places WordPress on an equal footing with other applications that have a stable, well-defined API. Remember, we've barely settled into the starting blocks at this stage – the real race is very much still to be run.

We can expect the REST API to power a new generation of powerful third-party integrations which enable non-technical users to create their own tailored solutions in a modular, drag-and-drop manner.

There's an excellent chance that we'll see something of a [Cambrian explosion](#) in terms of third-party integrations over the next two to five years. Put simply, if it *can* be integrated, it more than likely *will* be.

CONCLUSION

It's time to finally wrap up our ebook! We've covered a lot of resources along the way, but we'd like to point you towards a few, in particular, to close things out and help you keep pace with REST API development in general:

1. **WP REST API documentation:** The REST API is still a work in progress, and the [official docs](#) do a great job of getting people ramped up quickly.
2. **REST API-related presentations:** There's an increasingly excellent series of [deep-dive talks on WordPress.tv](#) that reward close viewing.
3. **Our own REST API reporting:** A regular review of the latest [REST API-tagged content](#) here on the site will keep you more than up to speed!

We hope you've learned as much throughout the ebook as we did putting it together!



ABOUT THE AUTHOR: TOM EWER

Tom Ewer is the founder of Leaving Work Behind and WordCandy. He has been obsessed with WordPress since he first laid eyes on it, and has been writing educational and informative content for WordPress users since 2011. When he's not running his businesses, you're likely to find him outdoors somewhere – as far away from a screen as possible!

twitter.com/tomewer

<http://wordcandy.co/>



CODE UPDATED BY: RYAN HOOVER

Ryan Hoover is a WordPress Developer for WP Engine and the lead organizer of WordCamp Austin 2017. He's been using WordPress for about 10 years. But that's only when he's not covered in sawdust or touring wineries with his wife and son.



About Torque

Torque is a news site featuring all things WordPress. We are dedicated to informing new and advanced WordPress professionals, users, and enthusiasts about the industry. Torque focuses primarily on WordPress News, Business, and Development, but also covers topics relating to open source and breakthrough technology. Torque made its debut in July 2013, at WordCamp San Francisco, and has since produced valuable content that reflects the evolution of WordPress, both as a platform and a community. Torque is a WP Engine publication, though maintains complete editorial independence. torquemag.io



About WP Engine

WP Engine powers amazing digital experiences for websites and applications built on WordPress. The company's premium managed hosting platform provides the performance, reliability and security required by the biggest brands in the world, while remaining affordable and intuitive enough for smaller businesses and individuals. Companies of all sizes rely on WP Engine's award-winning customer service team to quickly solve technical problems and create a world-class customer experience. Founded in 2010, WP Engine is headquartered in Austin, Texas and has offices in San Francisco, California, San Antonio, Texas, Limerick, Ireland and London, England.

TORQUE®

