PREV
13 Setting Up Your React Dev Environment Easily

NEXT
15 Building an Awesome Todo List App in React

14 Working with External Data in React

Dealing with external data is pretty much standard in web apps today. This “dealing” typically looks as follows:

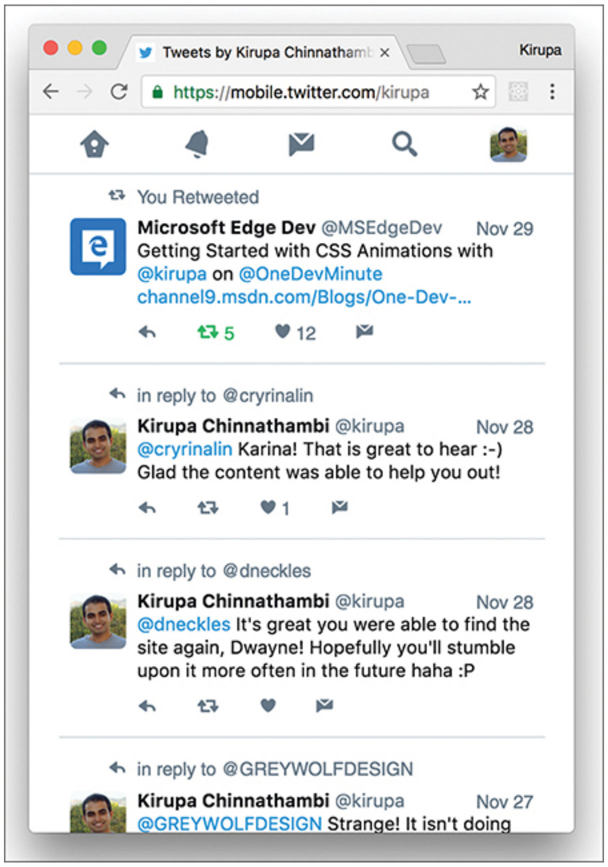
1. Your app makes a request for some data to a remote service.

2. The remote service receives the request and sends back some data.

3. Your app receives the data.

4. Your app formats and displays the data to the user.

Whether or not you realize it, almost all your favorite websites follow these four steps...Facebook, Amazon, Twitter, Instagram, Gmail, KIRUPA, and so on. When you starting loading a page on any of these sites, they all display some data initially.


Tweets by Kirupa Chinnathamb ×
Kirupa
https://mobile.twitter.com/kirupa

You Retweeted

Microsoft Edge Dev @MSEdgeDev Nov 29
Getting Started with CSS Animations with @kirupa on @OneDevMinute channel9.msdn.com/Blogs/One-Dev-...
5 12

in reply to @cryrinalin

Kirupa Chinnathambi @kirupa Nov 28
@cryrinalin Karina! That is great to hear :-) Glad the content was able to help you out!
1

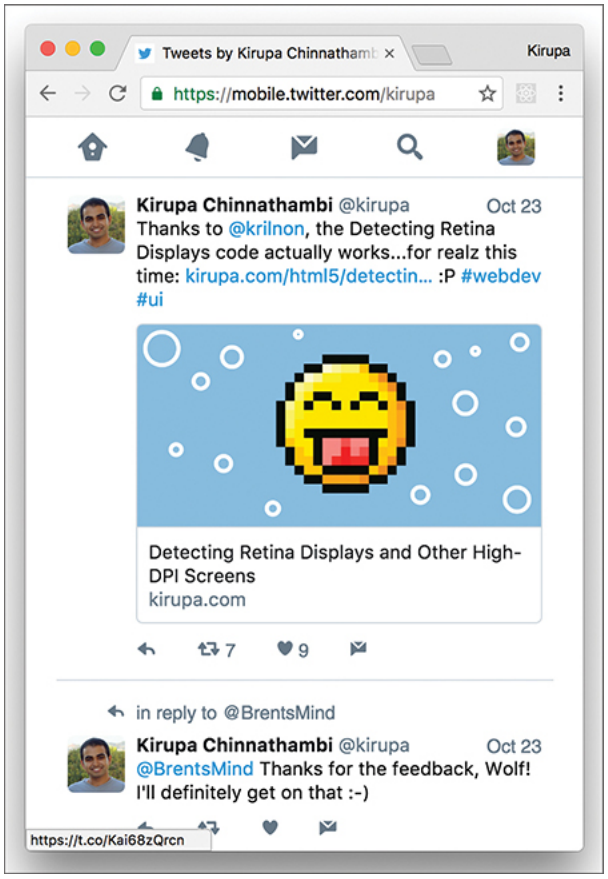
in reply to @dneckles

Kirupa Chinnathambi @kirupa Nov 28
@dneckles It's great you were able to find the site again, Dwayne! Hopefully you'll stumble upon it more often in the future haha :P

in reply to @GREYWOLFDESIGN

Kirupa Chinnathambi @kirupa Nov 27
@GREYWOLFDESIGN Strange! It isn't doing

To keep your initial page size low, not everything is downloaded at once. After your page has fully loaded or you interact with your page, the page downloads additional data from the server and displays it.


Tweets by Kirupa Chinnathamb ×
Kirupa
https://mobile.twitter.com/kirupa

Kirupa Chinnathambi @kirupa Oct 23
Thanks to @krilnon, the Detecting Retina Displays code actually works...for realz this time: kirupa.com/html5/detectin... :P #webdev #ui

Detecting Retina Displays and Other High-DPI Screens
kirupa.com
7 9

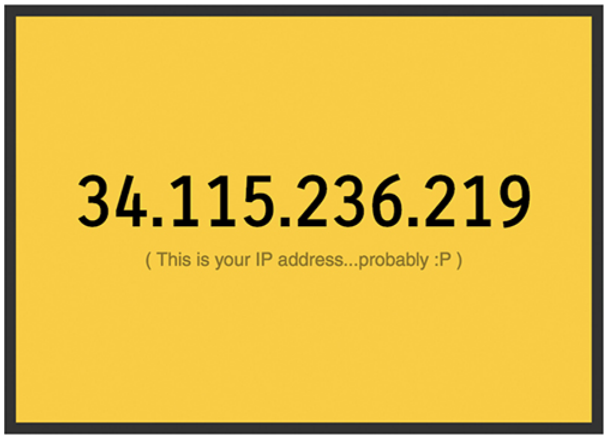
in reply to @BrentsMind

Kirupa Chinnathambi @kirupa Oct 23
@BrentsMind Thanks for the feedback, Wolf! I'll definitely get on that :-)

https://t.co/Kai68zQrcn

This is all done without requiring you to refresh the page or lose any state your page is in. The magic behind it all is a little bit of JavaScript that handles the four steps we looked at earlier. In this chapter, you'll learn all about the JavaScript needed to do that and how to make it all work inside a React app.

By the end, you'll have created a simple React app that looks as follows (view in your browser here: https://www.kirupa.com/react/examples/ipaddress.htm):



Here you're seeing your device's IP address displayed. That's it. I realize that this isn't very complicated, as examples go (especially if you were all excited seeing Twitter highlighted in the screenshots), but it contains just the right amount of complexity and relevant details to ensure that you know how to deal with external data from inside a React app.

Onward!

WEB REQUEST 101

As you probably know very well by now, the Internet is made up of a bunch of interconnected computers, called servers. When you're surfing the web and navigating between web pages, you're really telling your browser to request information from any of these servers. It kind of looks as follows: Your browser sends a request, waits awkwardly for the server to respond to the request, and (once the server responds) processes the request. All of this communication is made possible because of something known as the **HTTP protocol**.

The HTTP protocol provides a common language that allows your browser and a bunch of other things to communicate with all the servers that make up the Internet. The requests your browser makes on your behalf using the HTTP protocol are known as **HTTP requests**, and these requests go well beyond simply loading a new page as you are navigating. A common (and whole lot more exciting!) set of use cases revolves around updating your *existing* page with data resulting from a HTTP request.

For example, you might have a page where you'd like to display some information about the currently logged-in user. This is information your page might not have initially, but it is information your browser will request when you're interacting with the page. The server will respond with the data and update your page with that information. All of this probably sounds a bit abstract, so I'm going to go a bit weird for a few moments and describe a possible HTTP request and response for this example.

To get information about the user, here's our HTTP request:

Click here to view code image

```
GET /user
Accept: application/json
```

For that request, here's what the server might return:

Click here to view code image

```
200 OK
Content-Type: application/json

{
  "name": "Kirupa",
  "url": "http:https://www.kirupa.com"
}
```

This back-and-forth happens a bunch of times, and it's all fully supported in JavaScript. This ability to asynchronously request and process data from a server without requiring a page navigation/reload has a term: **Ajax** (or **AJAX**, if you want to shout). This acronym stands for Asynchronous JavaScript and XML. If you were around web developers a few years ago, Ajax was the buzzword everybody threw around to describe the kind of web apps we take for granted today (Twitter, Facebook, Google Maps, Gmail, and more) that constantly fetch data as you interact with the page, without requiring a full page reload.

In JavaScript, the object that is responsible for allowing you to send and receive HTTP requests is the weirdly named `XMLHttpRequest`. This object allows you to do several things that are important to making web requests:

1. Send a request to a server

2. Check on the status of a request

3. Retrieve and parse the response from the request

4. Listen for the `readystatechange` event that helps you react to the status of your request

`XMLHttpRequest` does a few more things, but those aren't important to deal with right now.

> **Why Not Use Third-Party Libraries?**
>
> A bunch of third-party libraries wrap and simplify how you can work with the `XMLHttpRequest` object. Feel free to use them if you want, but using the `XMLHttpRequest` object directly isn't very complicated, either. It's only a few lines of code, and (compared to everything you've been learning in React) they're some of the easiest lines of code you'll encounter.

IT'S REACT TIME!

Now that you have a good enough understanding of how HTTP requests and the `XMLHttpRequest` object work, it's time to shift our focus to the React side. I should warn you, though, that React brings very little to the table when it comes to working with external data. React is primarily focused on the presentation layer (a.k.a. the V in MVC). We'll be writing regular, boring JavaScript inside *a React component whose primary purpose is to deal with the web requests we'll be making*. We'll talk more about that design choice in a little bit, but let's get the example up and running first.

Getting Started

The first step is to create a new React app. From your command line, navigate to the folder where you want to create your new project and enter the following:

Click here to view code image

```
create-react-app ipaddress
```

Press Enter/Return to run that command. A few moments later, a brand new React project will be created. You want to start from a blank slate, so you're going to delete a lot of things. First, delete everything under your `public` folder. Next, delete everything inside your `src` folder. Don't worry: You'll fill them back with content you care about in a few moments, starting with your HTML file.

Inside the `public` folder, create a new file called `index.html`. Add the following content into it:

Click here to view code image

```
<!DOCTYPE html>
<html>

<head>
  <title>IP Address</title>
</head>

<body>
  <div id="container">

  </div>
</body>

</html>
```

All we have going here is a `div` element named `container`. Next, go to your `src` folder and create a new file called `index.js`. Inside this file, add the following:

Click here to view code image

```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";
import IPAddressContainer from "./IPAddressContainer";

var destination = document.querySelector("#container");

ReactDOM.render(
    <div>
        <IPAddressContainer/>
    </div>,
    destination
);
```

This is the script entry point for our app, and it contains the boilerplate references to React, ReactDOM, a nonexistent CSS file, and a nonexistent `IPAddressContainer` component. We also have the `ReactDOM.render` call that is responsible for writing our content to the `container` `div` element we defined in our HTML a few moments ago.

There's just one more thing to do before we get to the really interesting stuff. Inside the `src` folder, create the `index.css` file and add the following style rule into it:

Click here to view code image

```
body {
  background-color: #FFCC00;
}
```

Save all these changes if you haven't done so already. We sort of have the beginnings of our app started. In the next section, we're going to make our app really useful—or at least get really close!

GETTING THE IP ADDRESS

Next on our plate is to create a component whose job it is to fetch the IP address from a web service, store it as **state**, and then share that state as a **prop** to any component that requires it. Let's create a component to help. Inside your `src` folder, add a file called `IPAddressContainer.js` and then add the following lines inside it:

Click here to view code image

```
import React, { Component } from "react";

class IPAddressContainer extends Component {
  render() {
    return (
      <p>Nothing yet!</p>
    );
  }
}

export default IPAddressContainer;
```

The lines you just added don't do a whole lot. They just print the words `Nothing yet!` to the screen. That's not bad for now, but let's go ahead and modify the code to make the HTTP request by adding the following changes:

Click here to view code image

```
var xhr;

class IPAddressContainer extends Component {
  constructor(props) {
    super(props);


    this.state = {
      ip_address: "..."
    };

    this.processRequest = this.processRequest.bind(this);
  }
  componentDidMount() {
    xhr = new XMLHttpRequest();
    xhr.open("GET", "https://ipinfo.io/json", true);
    xhr.send();

    xhr.addEventListener("readystatechange", this.processRequest, fals
  }

  processRequest() {
    if (xhr.readyState === 4 && xhr.status === 200) {
      var response = JSON.parse(xhr.responseText);

      this.setState({
        ip_address: response.ip
      });
    }
  }

  render() {
    return (
      <div>Nothing yet!</div>
    );
  }
};
```

Now we're getting somewhere! When our component becomes active and the `component-DidMount` lifecycle method gets called, we make our HTTP request and send it off to the `ipinfo.io` web service:

Click here to view code image

```
        .
        .
        .
        componentDidMount() {


          xhr = new XMLHttpRequest();
          xhr.open('GET', "https://ipinfo.io/json", true);
          xhr.send();

          xhr.addEventListener("readystatechange", this.processRequest
        }



        .
        .
        .
```

When we hear a response back from the `ipinfo` service, we call the `processRequest` function to help us deal with the result:

Click here to view code image

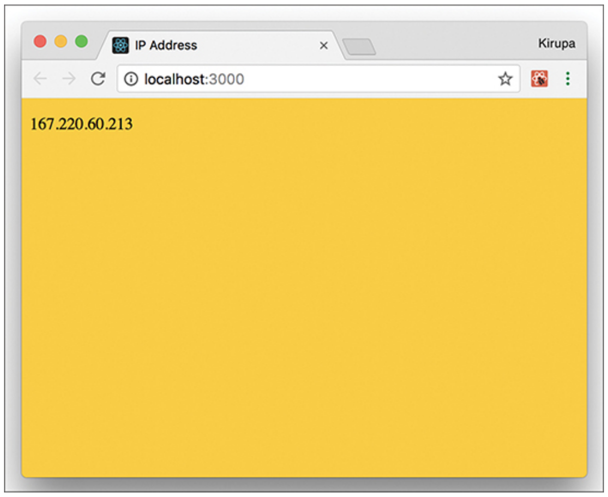
```
        .
        .
        .
   processRequest() {
    if (xhr.readyState === 4 && xhr.status === 200) {
      var response = JSON.parse(xhr.responseText);

      this.setState({
        ip_address: response.ip
      });
    }
        .
        .
        .
```

Next, modify the `render` call to reference the IP address value stored by our state:

Click here to view code image

```
var xhr;

class IPAddressContainer extends Component {
  constructor(props) {
    super(props);
    this.state = {
      ip_address: "..."
    };

    this.processRequest = this.processRequest.bind(this);
  }

  componentDidMount() {
    xhr = new XMLHttpRequest();
    xhr.open("GET", "https://ipinfo.io/json", true);
    xhr.send();

    xhr.addEventListener("readystatechange", this.processRequest, fals
  }

  processRequest() {
    if (xhr.readyState === 4 && xhr.status === 200) {
      var response = JSON.parse(xhr.responseText);

      this.setState({
        ip_address: response.ip
      });
    }
  }

  render() {
    return (
      <div>{this.state.ip_address}
</div>
    );
  }
}
```

If you preview your app in your browser, you should see an IP address displayed. If you need a reminder, you can preview your app by navigating into your `ipaddress` folder via your command line and entering `npm start`. When your app launches, it will look something like the following:



167.220.60.213

Our app currently doesn't look like much, but we'll fix that in the next section.

Kicking the Visuals Up a Notch

The hard part is done! We created a component that handles all the HTTP requesting shenanigans, and we know that it returns the IP address when called. Now we're going to format the output a bit so that it doesn't look as plain as it does now.

To do that, we won't add HTML elements and styling-related details to our `IPAddressContainer` component's `render` method. Instead, we'll create a new component whose only purpose will be to deal with all of that.

Add a new file called `IPAddress.js` in your `src` folder. Then edit it by adding the following content into it:

Click here to view code image

```
import React, { Component } from "react";

class IPAddress extends Component {
  render() {
    return (
      <div>
        Blah!
      </div>
    );
  }
}

export default IPAddress;
```

Here we're defining a new component called `IPAddress` that will be responsible for displaying the additional text and ensuring that our IP address is visually formatted exactly the way we want. It doesn't do much right now, but that will change really quickly.

We first want to modify this component's `render` method to look as follows:

Click here to view code image

```
class IPAddress extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.ip}</h1>
        <p>
( This is your IP address...probably :P )
</p>
      </div>
    );
  }
}

export default IPAddress;
```

The highlighted changes should be self-explanatory. We're putting the results of a prop value called `ip` inside an `h1` tag, and we're displaying some additional text using a `p` tag. Besides making the rendered HTML a bit more semantic, these changes ensure that we can style them better.

To get these elements styled, add a new CSS file to the `src` folder called `IPAddress.css`. Inside this file, add the following style rules:

Click here to view code image

```
h1 {
  font-family: sans-serif;
  text-align: center;
  padding-top: 140px;
  font-size: 60px;
  margin: -15px;
}
p {
  font-family: sans-serif;
  color: #907400;
  text-align: center;
}
```

With the styles defined, we need to reference this CSS file in our `IPAddress.js` file. To do that, add the following highlighted line:

Click here to view code image

```
import React, { Component } from "react";
import "./IPAddress.css";

class IPAddress extends Component {
  render() {
    return (
      <div>
        <h1>{this.props.ip}</h1>
        <p>
( This is your IP address...probably :P )
</p>
      </div>
    );
  }
}

export default IPAddress;
```

All that remains is to use our `IPAddress` component and pass in the IP address. The first step is to ensure that the `IPAddressContainer` component is aware of the `IPAddress` component by referencing it. At the top of `IPAddressContainer.js`, add the following highlighted line:

Click here to view code image

```
import React, { Component } from "react";
import IPAddress from "./IPAddress";
    .
    .
    .
```

The second (and last!) step is to modify the `render` method as follows:

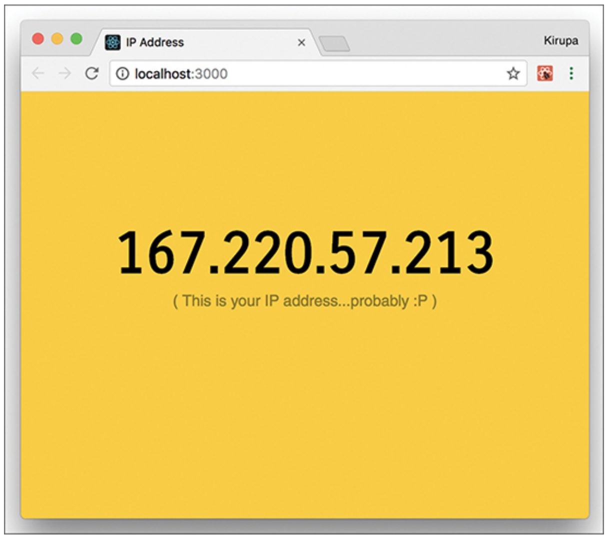
Click here to view code image

```
class IPAddressContainer extends Component {
    .
    .
    .
```

```
  render() {
    return (
      <IPAddress ip=
{this.state.ip_address}/>
    );
  }
}
```

In our highlighted line, we call our `IPAddress` component, define a prop called `ip`, and set its value to the `ip_address` state variable. This is done to ensure that our IP address value travels all the way back to the `IPAddress` component's `render` method, where it gets formatted and displayed.

If you preview the app in your browser now, you should see something identical to the example we set out to create in the beginning.



167.220.57.213

(This is your IP address...probably :P)

At this point, you're done with the app...and almost done with this tutorial. You just need to know one more thing about these awesome components that you've added.

Presentational vs. Container Components

Given what we've seen here so far, it seems like a good time to talk about a design choice that we've been indirectly following not just in this tutorial, but in other tutorials as well. In our React apps, we have been primarily dealing with two types of components:

1. **Components that deal with how things look.** These are better known as presentational components.

2. **Components that perform some under-the-covers processing.** Examples of this processing include routing, increasing a counter, fetching data via a HTTP request, and so on. You will see these components referred to as container components.

Thinking about your components in terms of whether they display something (presentational) or whether they feed data to other components (container) helps you better organize your React app. For the full low-down on how to deal with these two types of components, check out this article by React's Dan Abramov: https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0.

CONCLUSION

At this point, you're probably wondering what was made special because of React. All we really did here was use a boring old JavaScript API inside a component, hook up some events, and do the same state- and prop-related tasks you've done several times already. Here's the thing: You've already learned almost everything there is to learn about the basics of React. Going forward, nothing should surprise you. The only new things we'll be looking at fall into the category of repurposing and repackaging the basic concepts you already know into newer and cooler situations. After all, isn't that what programming is all about?

Note: If you run into any issues, ask!

If you have any questions or your code isn't running like you expect, don't hesitate to ask! Post on the forums at https://forum.kirupa.com and get help from some of the friendliest and most knowledgeable people the Internet has ever brought together!