React Basics

Last updated April 27, 2020

Theming Module Development 8.9.x/9.0.x

Before we start writing any React code, let's go over some basic concepts and terminology. Throughout this series we'll assume you're familiar with these things. They'll come up again and again as you work on projects that involve React, so it's worth taking the time to learn them.

In this tutorial we'll cover the following at a high level, and provide links to resources:

- Why choose React?
- What are React components?
- What are hooks, state, and JSX?
- The role of build tools when developing React applications

By the end of this tutorial you should have a firm grasp of the fundamental concepts and terminology necessary to start creating React applications.

Goal

Learn to identify important React concepts and terminology.

Prerequisites

- Familiarity with JavaScript

Essential React

The JavaScript ecosystem notoriously churns out new frameworks and libraries seemingly on a weekly basis. Blink and you may miss the next big thing. Even with all this churn, React has won the front-end race for years now. Why is that? One of the primary reasons is developer happiness. According to the State of JavaScript, in 2016, 92% of developers who had used it before would use it again, and that number rose to 93% in 2017. Those are the highest percentages among all frameworks surveyed, including Angular 1 and 2, Vue, Backbone, Polymer, Ember, and others. In 2018, it tied with Vue.js and in 2019, React regained its number one ranking with an 89% satisfaction ratio.

In this article, we will give you a taste of why developers are so happy with React. We'll also talk about how some of the concepts are similar or different from the Drupal and PHP world. But first, what is React? React is a JavaScript library for building user interfaces. React is not a framework. If you're familiar with the MVC framework, it's only the V, or view. In a typical Drupal site, a React front-end would replace your theme layer.

You can implement React in multiple ways. One is as a stand-alone front-end application that uses API endpoints from your Drupal app -- a headless CMS. You can also embed it directly in your Drupal front end, and pick and choose parts of your application to convert -- much the same way you might use jQuery with Drupal right now. The REST API in Drupal 8 Core, the JSON API module, and the GraphQL module all have made implementing React on a decoupled Drupal backend easier.

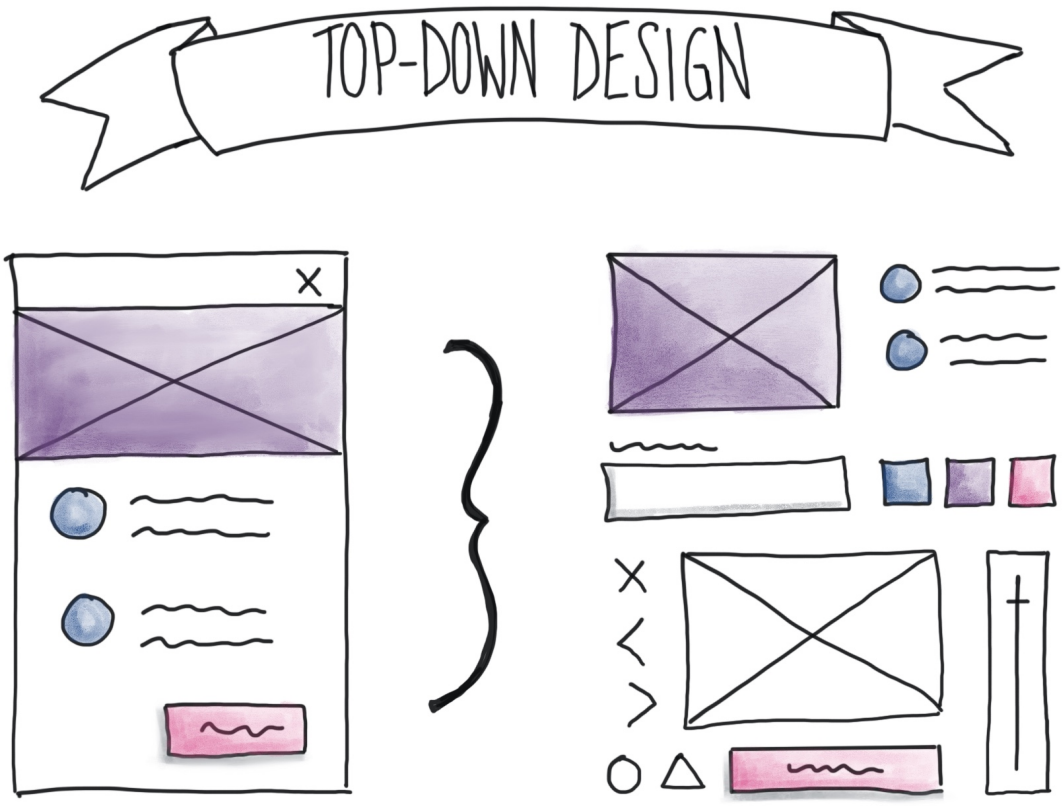
Why React?

One of the biggest reasons to use React is that it makes building the front end immensely easier than Drupal theming. It can provide more complex user experiences that are also more performant. With React you can use other non-Drupal services to extend the features of your application.

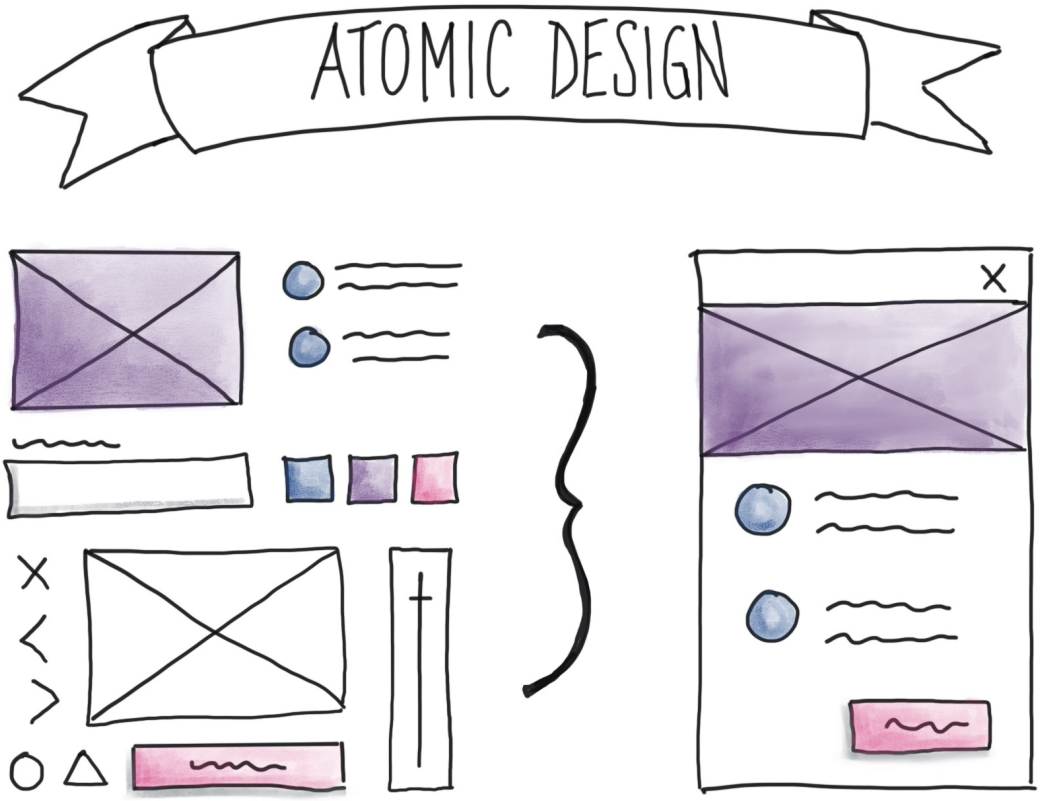
Now let's dive into the details.

Atomic Design

Most of us have probably worked with top-down designs provided by page and screen. We then took those designs and broke them down into their containers, grid systems, and elements to write our markup. This is how Drupal core and most web development works.



Atomic Design, a design-systems methodology outlined by Brad Frost, states that we should start with the basic building blocks, or atoms, of a page and work our way up to create designs:



How does all of this relate to React? React is a component-based system of building user interfaces (UI). Developers write reusable components, starting at the "atomic" level, and work their way up to collections of atoms, layouts, and screens. We refer to this as the component hierarchy. Each of these atoms can receive data and have their own state, creating a powerful set of UI building blocks. Components are similar to Drupal or Twig templates in that they can be reused with different sets of data through variables, and assembled into a tree of layouts and elements.

Pattern Lab and Emulsify have made atomic design easier in Drupal through the use of "living" style guides. React takes this one step further with more flexibility, ease of developing and modifying, and composition of applications from components. In the next section, you'll see how developing components is easier with one representation of a view.

One representation of a view

In React, we write one full representation of a view. In other words, we put both the structure (markup) and behavior (events, etc.) in one place. React calls this a Component. Often, we put the style there too, but this isn't required. For example, we might have a button with a click event. In React, the markup and click event are both in the same JavaScript file:

```
const Button = <button onClick={console.log('clicked!')}>Click Me</button>
```

However, when we use plain JavaScript or JQuery to write the same button, we need to write the structure in an HTML file:

```
<!-- HTML file -->
<button id="my-button">Click Me</button>
```

Then we need a separate JavaScript file to define the click event, find the target node, and add an event listener:

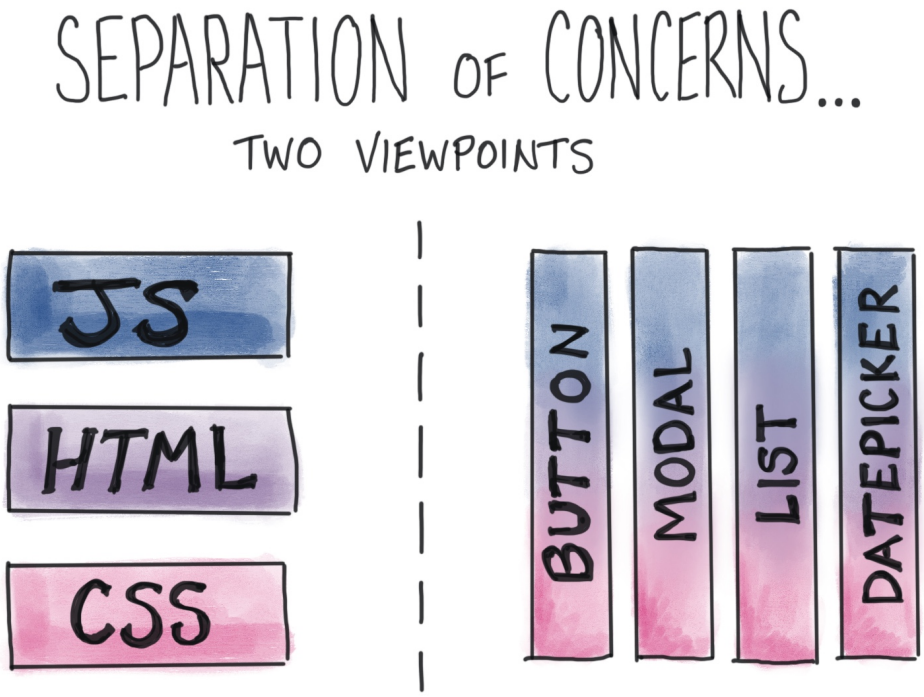
```
// JavaScript file
function onClick() {
 console.log('clicked!');
};
const button = document.getElementById('my-button');
button.addEventListener('click', onClick);
```

In React, our HTML files usually have a normal `<head>`, and the body contains a root node for our React application to target. All the rest of the markup is written in JavaScript (or JSX) alongside its behavior.

From the React documentation:

> *Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.*

Having the structure and behavior in one place makes makes it easier to reuse elements and components, making atomic design possible. Both markup and script (and sometimes style) relevant to a particular "atom" is managed in one place.



Contrast this with Drupal themes, where modifying look and feel or even behavior can become a scavenger hunt. When it's time to make changes to a theme you first need to understand the unique configuration of the site, the organization of the theme, and all the decisions that the person before you made.

> *Components and their behaviors are intimately dependant and should be encapsulated like so. --James Long on Removing User Interface Complexity or Why React is Awesome*

Unlike Drupal, React has no concept of a theme or template override. No templates, layouts, or components come with React. You either write your own or import an library of components. Either way, you have complete control over the markup, behavior, and style with less management of multiple layers of templates.

Learn more about components and props.

Side note on JSX

So what was that funny syntax we used for the button? Most React developers use JSX, or JavaScript XML, to create components that look similar to HTML but with the power of JavaScript expressions. For example, this is a simple JSX element representing an `<h1>` tag with inner text:

```
const title = <h1>Hello, JSX!</h1>
```

You're not required to use JSX with React, but the library encourages it and nearly the entire community uses it. Why? Here is the same code as above, but written without JSX:

```
const title = React.createElement(
  'h1',
  {},
  'Hello, React!'
)
```

As you can see, JSX is both easier to read and less verbose. This is especially true once you start writing element trees with children. On top of that, you can drop JavaScript expressions wherever you need them. This works like variables in Drupal or Twig templates, but is far more powerful because you can write any JavaScript you want. It is not limited to just one subset of the language's capabilities like Twig.

For example, below we have a more complex React component that outputs an input group with label and conditional error and helper text. All the curly brackets in the return statement are areas where JavaScript expressions are inserted. Sometimes it only has data like name or type, but sometimes it has conditionals or operations like Array.join:

```
const Input = (props) => {
  const { labelText, type, name, required, helperText, errors } = props
  const label = labelText ? labelText : nameToLabel(name)
  const hasErrors = !isEmpty(errors)

  return (
    <label className={hasErrors ? 'is-invalid' : ''}>
      {label} {required && <span className="required">*</span>}<br />
      <input type={type} name={name} />
      {hasErrors &&
        <span className="error-text">{errors.join(', ')}</span>
      }
      {helperText &&
        <span className="helper-text">{helperText}</span>
      }
    </label>
  )
}
```

JSX must be compiled into JavaScript in order for the browser to use it. This is typically done via a build process where tools like Babel take your JSX (and ES6+) code, transform it into valid JavaScript, and save the result as a production-ready asset, which is served to the browser. This can also be done client side, though isn't as performant. In this series we'll look at both approaches.

Learn more about JSX:

- Introducing JSX (reactjs.org)
- JSX technical documentation (jsx.github.io)
- Babel JSX compiler (babeljs.io)
- Online JSX compiler (babeljs.io)
- React without JSX (reactjs.org)
- Pros & cons of JSX (reactenlightenment.com)

State

The behavior of a component is tracked and manipulated via *state*. Change a button component's *state* from "on" to "off" and React will detect that state change, ask the button component to render its output again, and display that rendered output. Internally the render function of the component might contain logic that causes the button to render with different markup depending on whether the *state* is "on" or "off".

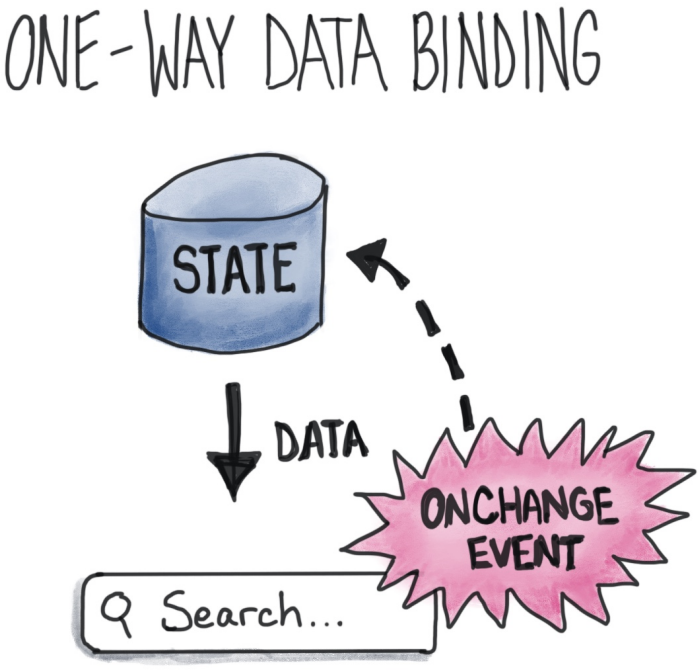
Learn more about state, and state management.

Components communicate with one another through the use of *state* and *events*.

- Component State (reactjs.org)
- A Visual Guide to State in React (daveceddia.com)

One-way data flows

In React, data only flows one way: down the component hierarchy. To update data, we send events up the hierarchy to trigger changes to data, or state, which then flow back down to the components.



To provide a little more context, React components can have 2 forms of data: props and state. Props are properties received from ancestors in the component hierarchy, and cannot be changed, or mutated. State is local to a component, and can be changed by events. Child components can receive both the values of that state and events to update that state through props.

In this example, the `<Form>` component holds the values of `firstName` and `lastName` in state. The `onChange` and `onSubmit` events are also defined in the `<Form>`. The `<Input>` component is defined elsewhere, and it accepts props of `name`, `value`, and `onChange`. The `onChange` event is triggered in the child `<Input>` component, which triggers an update to the `<Form>` state, which flows back down to the `<Input>` so we see the updated input value.

```
class Form extends React.Component {
 constructor() {
   super()
   // Set initial state
   this.state = {
     firstName: '',
     lastName: ''
   }
 }

// As the user types in an input, update the value in state for the
```

```
// property with the same `name` as the input.
 onChangeInput = (e) => {
   this.setState({
     [e.target.name]: e.target.value
   })
 }

 onSubmit = (e) => {
   // Prevent the default browser POST action
   e.preventDefault()
   console.log(`Submitted! firstName: ${this.state.firstName}, lastName: ${this.state.lastName}`);
 }

 render() {
   return (
     <form onSubmit={this.onSubmit}>
       <Input
         name='firstName'
         value={this.state.firstName}
         onChange={this.onChangeInput} />
       <Input
         name='lastName'
         value={this.state.lastName}
         onChange={this.onChangeInput} />
       <Button>Submit</Button>
     </form>
   )
 }
}
```

One-way data flow ensures that the state remains the single source of truth for a component tree. It also decouples data from the view so that updates can be batched or done asynchronously. The good news is React does this for you in an optimized way to minimize unnecessary re-renders. Our applications become less buggy and easier to maintain. Winning!

Blazing fast speed

React is blazing fast across two dimensions:

- Loads fast due to its small size
- Re-renders quickly when responding to changes due to the virtual DOM.

For size and load time, out of all the major frameworks, it ranks second only to Vue at 97.5K raw and 31.8K compressed. Not all front-end assets are created equally. A 170K image may load, decode, rasterize and paint all in less than 3.41 seconds. On the other hand, 170K of JavaScript on the same network and device would take 6.9 seconds to load, parse and compile, and execute. JavaScript is your most expensive asset so library size matters.

NAME	SIZE (MINIFIED)	SIZE (GZIPPED)
Vue 2.4.2	58.8K	20.9K

NAME	SIZE (MINIFIED)	SIZE (GZIPPED)
React 16.2.0 + React DOM	97.5K	31.8K
Angular 1.4.5	143K	51K
Ember 2.2.0	435K	111K
Angular 2	566K	111K

Source

In terms of responsiveness, React uses a virtual DOM. The virtual DOM is an ideal representation of the UI in memory. React syncs the real DOM with the virtual DOM through a process called reconciliation. The reconciliation algorithm diffs the before and after states to only re-render the parts of the DOM that changed. This results in fewer wasted renders and thus higher, or faster, responsiveness. React also provides a lifecycle method where you can override the algorithm to state whether a component should update or not.

Hooks

React hooks, much like Drupal hooks, are functions that let you hook into a React application's state and lifecycle.

Hooks are a relatively new addition to the React core API, added in version 16.8. They are quickly becoming the recommended way to implement state, interact with the component lifecycle, and handle other React features that previously required writing a class. Hooks are designed to make it easier to reuse stateful logic between components. For example, 2 different components might need to be able to determine if the current user is logged in or not and adjust their display accordingly.

The biggest win you get from using hooks is that you can write React components that are functions instead of classes and the resulting code is often easier to test and maintain.

There are currently no plans to remove classes from React, and you can always write a class component, and use the standard lifecycle methods. However, in the real-world we're seeing hooks used more and more, so we'll use them in most of our examples.

Learn more about React hooks.

ES6+

As a Drupal developer, you may or may not know JavaScript, and if you know it, you may or may not know ES6. So what is this ES6 you keep hearing about and why is it important in React?

"ES" is short for ECMAScript. ECMAScript is the standard used for ongoing development. The most popular implementation of ECMAScript is JavaScript. The number after "ES" refers to the edition of ECMAScript. You probably learned ES5 originally, which was released in 2009. ES6, also know as ES2015, was released in 2015, and included several new features. Now, development is happening more rapidly, and we also have ES7 (ES2016) and ES8 (ES2017). It can be very confusing, but for the most part, we generally refer to ES6 and above as ES6+ or ES2015+. It's still all JavaScript, but with new features that have been added through the years.

Why is this important in React? The React community has embraced ES6+ with open arms. A few years ago, tutorials and documentation were written in ES5. Currently, almost all tutorials (including ours!) and documentation are written using ES6+ syntax. To get onboard with React, you'll have to learn some of JavaScript's new features. It's like how Drupal 8 uses the latest version of PHP.

The good news...

As a PHP developer, ES6 is likely to be much more approachable than ES5 and earlier. For example, the new class syntax is very similar to PHP's.

You don't have to learn every new feature of ES6 and above. A few key components will help you get by, and once you learn them we think you'll like them.

So while you'll have to learn something new, you'll also be able to take advantage of both ease of use and performance benefits. Here's a handful of resources to get you started:

- JavaScript to Know for React (kentcdodds.com)
- Mozilla JavaScript API guide (developer.mozilla.org)
- A list of ES6 tutorials (medium.com)
- The JavaScript classes on FrontEnd Masters (requires subscription) (frontendmasters.com)

Transpiling

Since not all of the awesome features of ES6 are available to all browsers, your ES6 code can be run through a *transpiler* (translator + compiler). The most common transpiler is Babel. Babel converts code from one version of JavaScript to another, and outputs a new script that even IE 11 will understand. We'll use both the in-browser compiler and the command-line compiler while working through the examples in these tutorials.

Community

Similar to Drupal, React is open source. However, it was written by Facebook originally, which also drives the future development path of React. On the plus side, we have a huge company funding developers that work full-time on upgrades and enhancements. Open source contributors are welcomed with open arms to submit both issues and pull requests. The React team also publicly shares the meeting notes for their weekly meeting discussing future development and priorities. You can read more about contributing here.

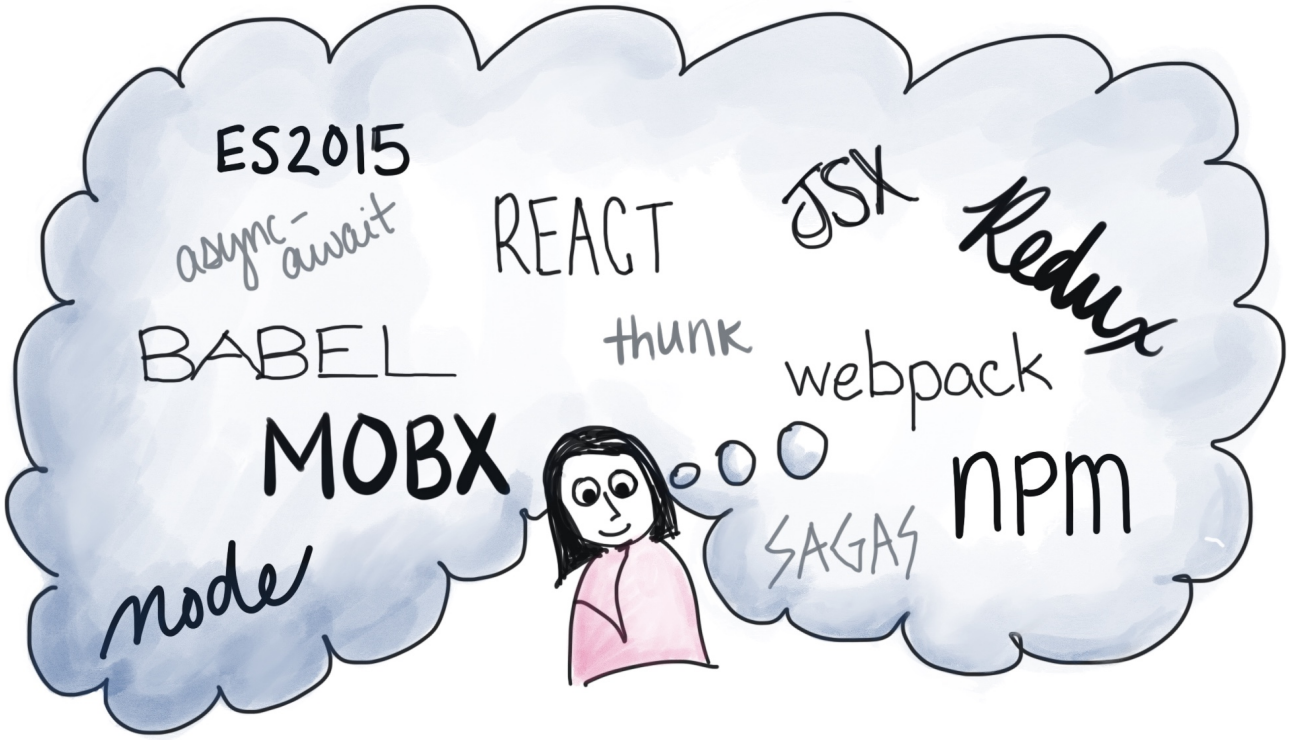
React is lightweight partially because it is not opinionated about other tools you might want to use with it. For example, it comes with state management, but many people choose to use a separate state management package such as Redux or MobX. For single-page applications (SPAs), you can build your own router or use packages such as React Router or Next.js to manage routing for you.

Unlike Drupal where popular modules and patterns are often incorporated into core, React does not incorporate or adopt popular libraries. They keep the scope narrow so that React itself remains nimble and easier to update. Development is focused on delivering better on the core, including a relentless pursuit of higher performance to improve user experience. This also means that it is more difficult to define "best practices" in React since is it more of an ecosystem rather than a monolith like Drupal that standardizes over time.

React and the JavaScript ecosystem

One complaint that new JavaScript developers often have is that the ecosystem seems so vast, new tools appear seemingly each day, and some members of the community have so many strong opinions.

Since React itself is just one small package, the ecosystem to get a full application up and running can be intimidating. From ES6, Babel, and Webpack to npm to state management, routing, and sync requests, it can be a lot to learn for newcomers.



The good news is that better tools exist now for bootstrapping new React applications that don't require you to learn how to best manage Babel and Webpack for transpiling and bundling your applications. Tools like Create React App and Next.js have been a godsend for helping new and experienced developers alike focus on React and get coding faster. The trend is to keep improving these tools that help make developers' lives easier so that they can ship code faster.

Where should I start?

Just getting started with React? We recommend taking the time to walk through the official tutorial. Note that our examples will differ from the tutorial in that we'll use React hooks. More on that later.

Styling resources

We recommend you explore some of the various options for including CSS in your components and pick the one that'll work best for you and your team. Here's some of our favorite articles on the subject:

- What to use for React styling? (javascriptstuff.com)
- Styling in React: From External CSS to Styled Components (sitepoint.com)

- The best "styling in React" tutorial you've ever seen (blog.logrocket.com)

Learning curve

It takes some time to understand the various connected technologies that are used to build JavaScript applications. This includes mastering a new way of writing JavaScript, various code compilers, asynchronous requests over a network, new kinds of CSS compilers, routers, more interactive responsive interface support, and more.

But you don't have to learn it all at once. One of our favorite features of React is that it's just JavaScript. So you can start simple, and get more and more complex as the need arises. That's exactly what we'll do as we work through the examples in this series.

Recap

In this tutorial, we introduced some of the fundamental concepts of React, including components, state, hooks, and ES6. We talked about the React community, and provided to links to some of our favorite resources for learning React. We also talked about some of the additional things you'll *eventually* want to learn as you get further into using React.

Further your understanding

- Can you find some React example code online and identify the fundamental pieces? JSX, hooks, Components, etc.
- The tutorial in the official React documentation is a great way to familiarize yourself with these concepts. Try it.

Additional resources

- React Docs (reactjs.org)
- JavaScript (MDN) (developer.mozilla.org)
- 42 JavaScript experts to follow on Twitter (techbeacon.com)

◀ Previous tutorial

Next tutorial ▶

Get Started Using React and Drupal Together

1. Introduction to React and Drupal Free
2. React Basics
3. Decoupled vs. Progressively Decoupled
4. Connect React to a Drupal Theme or Module Free
5. Create a React Component
6. Add Webpack Hot Module Replacement (HMR) to a Drupal Theme
7. Retrieve Data from an API with React
8. Use React to List Content from Drupal
9. Create, Update, and Delete Drupal Content with JavaScript
10. Build an Interface to Edit Nodes with React
11. Create a Fully Decoupled React Application Free
12. Use create-react-app to Start a Decoupled React Application
13. Make API Requests with OAuth
14. Use Fetch and OAuth to Make Authenticated Requests

Subscribe

STAY CONNECTED

Powered by: