React: The Big Picture

by Cory House

Start Course

Bookmark Add to Channel Download Course

Course Overview

Course Overview

Hello. My name is Cory House, and welcome to React: The Big Picture. I'm the principal consultant at reactjsconsulting. com. Are you considering React and wondering why it's so popular? Are you curious if React is a good fit for you and your team? Or perhaps you're already using React, but you want to better understand the merits and downsides so that you can better sell React to your coworkers and leadership. If so, this course is for you. In this course, we're going to answer three questions, why should I choose React, what tradeoffs are inherent in React's design, ecosystem, and philosophy, and why shouldn't I choose React? Some of the major topics that we'll cover include the many potential use cases for React, what sets React apart from its competition, key projects to consider in the React ecosystem, approaches to mitigating React's downsides, and the five key decisions that you need to make to get started. By the end of this course, you'll understand React and its ecosystem at a high level and you'll have a clear view of React's strengths, tradeoffs, and weaknesses. I hope you'll join me to learn what makes React special in React: The Big Picture, at Pluralsight.

Why React?

Intro

Hello, and welcome to React: The Big Picture. If you're watching this, I assume that you're new to React and you're looking for a short overview of what it is and why it has become so popular so that you can decide if React is right for your team. I'm Cory House. You can catch me on Twitter @housecor or at my consulting site, reactjsconsulting. com. This course consists of three short modules. In this first module, we'll explore what makes React special and worth choosing over the competition. In the next module, we'll discuss the tradeoffs inherent in React's design so that you can understand what you're getting and what you're giving up by choosing React. And I'll close out the course by considering common concerns that I hear about React and approaches for how to mitigate these concerns. All right, let's get started.

History

Let's begin with a brief history of React. Facebook created React in 2011 for their own use on facebook. com, one of the highest-trafficked websites in the world. React was then utilized by Instagram a year later in 2012. After already using React internally for 2 years, Facebook open sourced React in 2013. Some initially dismissed it because React ran contrary to popular practices by placing markup and logic together in a single file. But as more people experimented with the library, many embraced the new component-centric philosophy for separating concerns. Each React component is a separate concern. A year later in 2014, React had grown significantly in popularity and was embraced by many notable companies outside of Facebook. This popularity led Facebook to open source React Native too in 2015. React Native is a related library that allows you to create native mobile applications for iOS and Android using React. In April of 2016, React reached another significant milestone by publishing version 15. This was a notable release because the previous version was. 14. Moving to 15. 0 finally put React's semantic versioning scheme in sync with traditional semantic versioning practices, and it also helped convey that React was now a mature and stable platform with over 5 years of active development and heavy production usage to back it up. Today there are over 30, 000 React components in production at Facebook. Facebook is deeply committed to React since it also uses React on Instagram and React Native for mobile development. Today, Facebook employs a full-time React development staff that regularly releases bug fixes, enhancements, blog posts, and documentation. And as you'll see soon, many large, well-respected Fortune 500 companies now utilize React in production.

Why React?

If you're watching this course, the biggest question on your mind likely is why should I choose React over the long list of alternatives? Well, throughout the course, we'll explore the answer in detail, but in the next few clips, let's explore six key reasons, flexibility, developer experience, corporate commitment, community support, performance, and testability.

Reason 1: Flexibility

Perhaps the most compelling reason to choose React is once you learn it, you can build user interfaces for a huge variety of platforms and use cases. React is remarkably flexible. React embeds fewer opinions than its competition, so it's more flexible than opinionated frameworks like Angular and Ember. React is a library, not a framework, and as you're about to see, React's library approach has allowed it to evolve into a remarkably flexible tool. When React was initially created, it had a single, focused use case. It was for creating components for web applications. However, as React has grown in popularity, its ecosystem has grown to cover a variety of use cases. You can generate static sites with React using popular tools like Gatsby and Phenomic. You can use React to build truly native mobile applications using React Native. And with great tools like Electron, you can even create installable desktop apps that run on Mac and Windows using web technologies like React. React also supports server rendering out of the box, and popular frameworks like Next. js make it really easy to set up. Finally, you can use React to create virtual reality websites and 360 experiences with React VR. In summary, learn React once and you can write applications just about everywhere. React is highly versatile because the render is separate from React itself. For web apps, you call react-dom to render your components to HTML. For React Native, you use react-native to render React components into native-friendly code. And react-vr is useful for rendering your React components into a virtual reality environment. In fact, there are over a dozen different renderers for React. This list is called awesome-react-renderer and lists a variety of other targets that React can render to, including canvas, WebGL, command line apps, and even PDFs and Word documents. React-dom provides a simple function called renderToString that renders your component to a string of HTML. This is useful if you want to render your React components on the server, so this means you can use React to replace your traditional server-side rendering technologies. And there are multiple popular libraries that make it easy to render React components on the server, including Next. js, Gatsby, and Phenomic. You can also generate static content with these libraries as well, so you can easily use React to deploy plain HTML and JavaScript files to any host. And since React is a lightweight library, you can use it with existing apps too, even server-side rendered apps. In fact, that's precisely what React was designed for. Facebook used React to slowly replace its server-side rendered PHP application.

You can start with small portions of the page like this, then move onto slightly larger portions of the page, and eventually, rebuild the entire page by slowy replacing larger sections with React. It's a low-risk way to migrate an existing app to React or to use React in places where it's most valuable. Finally, since React is used heavily by Facebook, you can trust that it will continue to run reliably in all popular browsers, including recent versions of Internet Explorer. Of course Facebook can't afford to have its website only run on some browsers.

Reason 2: Developer Experience

Reason number 2 is developer experience. Trust me, once your team tries React, they'll likely fall in love. The rapid feedback development experience, combined with React's small, logical API, creates a development experience that's very hard to beat. React is special because it offers a simple API that's easy to learn. There are very few concepts to master. React's API is so small and straightforward that developers rarely need to check the docs. The entire API just fits in your head. Could this get simpler? It's basically a function that returns what looks like HTML. With React components, you import React at the top using a standard JavaScript import statement, then you can declare a component using a standard JavaScript function. The function receives variables via an object called props. You can also declare a React component using a standard JavaScript class. This approach is slightly more typing, but gives you a little more power. Now, you're likely wondering what's happening here inside this render function. It looks like HTML, but it's sitting in a JavaScript file, so how does that work? Well, that's called JSX. As you can see, the JSX on top looks like HTML. There are a few minor differences between JSX and HTML, and I'll quickly show you those later. Now, JSX compiles down to JavaScript. It becomes a call to React. createElement. The function is passed the name of the tag that you created, an object that specifies the attributes you'd like to set, and finally, the markup that should sit inside. This final parameter contains calls to other elements if you have nested markup. So you can avoid JSX if you want to write this JavaScript yourself, but of course I recommend using JSX since clearly the top is easier to read and nest. Traditional libraries like Angular, Vue, and Ember seek to enhance the power of HTML by inventing their own syntax for simple operations like looping. For example, in Angular, you say *ngFor let user of users. In Vue you say v-for user in users, and in Ember, you say #each user in users. React went the exact opposite route. Instead of trying to make HTML more powerful, React just handles HTML with JavaScript. You don't need to learn new framework-specific keywords, rules, and syntax for conditionals, looping, and so on. You just use JavaScript. JavaScript already has a built-in function for iterating over an array called map, so in React you just plain JavaScript. Basically, traditional libraries put fake JavaScript in HTML. And React is

doing the exact opposite. It puts fake HTML in JavaScript. I find React's approach preferable because it creates a simpler API. React encourages you to get better at JavaScript, and in doing so, you also get better at React. Here I'm using create-react-app, which is the most popular way to do React development today. I just say create-react-app with the desired name for my app, and it creates a full working development environment on my machine. With a single command, npm start, it starts up a web server on my machine and serves my React app. In React, each component is atomic; it stands alone. This means you can work with each component in isolation, and every time I hit Save, the changes are immediately reflected in the browser. And here's the great thing. If you make a mistake, you receive a detailed error message in the browser. If I forget to close a tag within JSX, I get a compile time message about the precise line where I made a mistake. And if you need to debug your code, it's easy to set a breakpoint and view the original code in the browser. Through the power of source maps, you can see the original code that you wrote in your editor displayed within the browser. And everything I'm showing you here happens automatically with create-react-app, the most popular way to do React development today. Or you can use an online React editor like CodeSandbox to easily experiment with React, share your work, and even build your entire app. There's no configuration required. Just load the site and start coding. CodeSandbox makes it easy to experiment with React and share your work. When I'm trying an idea, I just load up this site and start coding. You can create multiple files, reference them, use modern JavaScript, and every time you hit a key, you see your changes reflected on the right.

Reason 3: Corporate Investment

Reason number 3 is that many well-respected corporations are deeply invested in React and its ecosystem today. React was created by Facebook, so of course React is heavily used on Facebook, one of the highest-trafficked apps in the world, as well as Facebook's other properties such as Instagram and WhatsApp. Facebook is deeply committed to React. Although React is an open source project, four of the top six committers to the React project are current, full-time Facebook employees. And the Facebook development team maintains an active blog that consistently outlines the details of each release and plans for the future. Because of Facebook's deep existing commitment to React in production, when breaking changes occur in React, Facebook has consistently provided a codemod that automates the change. Now, a codemod is a command line tool that you can point at your code base to automate changes. So with React codemod, you can automatically update older React components to the latest specification. Over the years when breaking changes have occurred, the Facebook team has consistently published a

codemod in order to automate updating to the latest version. So for example, when Facebook released React 15. 5, the propTypes feature was published as a separate package, and use of the propTypes feature embedded in React began throwing a warning to notify developers of the change. To make updating easy, Facebook released a codemod to automate the changes by updating imports to reference the separate propTypes packages. And it's smart enough to update relevant code below in the body of the components as well. The beauty is we can feel confident about writing React components today because of Facebook's deep investment in the production React code, means they must rely on the codemods that they create to update their own code. See, codemods exist because Facebook needs them. Facebook has over 30, 000 components in production. This is a benefit of using React because it helps assure that significant breaking changes in the future are highly unlikely. Doing so would require Facebook to deal with painful breaking changes to tens of thousands of their own components, so this helps assure the long-term stability of the project.

Reason 4: Community

The 4th reason to consider React is it boasts a huge, active community. Since 2013, React's popularity has steadily grown to over 75, 000 stars on GitHub today. This makes React one of the most popular repositories on GitHub. Today it has over 1, 000 contributors. In fact, out of over 2 million repositories, only 3 repositories have more stars on GitHub than React. At the time of this recording, the React npm package is downloaded over 1. 5 million times every single week. Now that's seriously impressive. On StackShare, a site where companies can share the technologies that they're using, nearly 5, 000 companies have reported using React. And at the time of this recording, React is the eighth most popular technology on all of StackShare. Facebook developers and a long list of open source React contributors are also involved in Reactiflux, which is an active online community of over 20, 000 React developers. There are over 55, 000 threads on Stack Overflow that are tagged reactjs, and nearly 20, 000 tagged react-native, as well as a long list of related tags for other React-related technologies. Now all this matters because it adds up to a simple fact: if you're trying to do something in React, you can almost certainly find an example of it online. And hey, what developer doesn't love a little copy and paste now and then? It's pretty awesome being able to consistently get answers from other people who have run up against the same challenge that you're trying to solve. And that's because React is embraced by far more than just Facebook. Today, many of the world's most respected companies use React, including Apple, Netflix, Amazon, Airbnb, PayPal, and many more that you see here. Many of these companies regularly open source their React-related work as well. So if you choose React,

you're certainly in great company. And you don't need to create your own components, since there's a huge list of free and mature component libraries online. Microsoft open sourced their React component library for making user interfaces that look and feel like Microsoft office. Material-UI offers a set of React components that implement Google's material design guidelines. And React-Bootstrap is a popular library that contains React components that make it easy to work with Bootstrap. Plus, there are literally hundreds of interesting standalone React components out there on GitHub that you might find useful. Check out the awesome React list on GitHub for a long list of additional components. Deep community investment has led to a wide variety of mature, related projects. Do you need routing? Well, check out React Router. Do you want to handle complex data flows using a library? Well, Redux and Mobx are two popular options to consider today. Do you want to set up automated testing? Well, check out Jest, which is also from Facebook. Want an alternative to RESTful API calls where you can declare your API calls on the client? Try out GraphQL, which is a great fit on React apps. Want to quickly set up a server-side rendered site in React with node? Then try Next. js. Of course, this just scratches the surface of the ecosystem. This list could go on and on. So I guess you could say that React is kind of a big deal right now.

Reason 5: Performance

When React was first released, its performance was striking and helped set it apart from the competition. The React team recognized that JavaScript is fast, but it's the DOM that makes it feel slow. They realized that updating the DOM is expensive, so they found that updating the DOM in the most efficient way would help enhance performance. So behind the scenes when you change data, React intelligently figures out the most efficient way to update the DOM. When React was created years ago, this was a novel design that gave React a notable performance advantage in many cases. See, before React, most libraries would unintelligently update the DOM to reflect the new state. This often led to redrawing a significant portion of the page even when only a minor change had occurred. In contract, React monitors the values of each component's state. When a component's state changes, react compares the existing DOM state to what the new DOM should look like. It then determines the least expensive way to update the DOM. This sounds complicated, but it's all handled automatically behind the scenes. And there are multiple benefits to this approach. It helps avoid layout thrashing, which is when a browser has to recalculate the position of everything when a DOM element changes. And being efficient is increasingly important in a world where many people are using mobile devices. Mobile devices vary widely in their CPU power, and conserving battery life is a concern as well. This also enables

React's simple programming model. When data changes, React efficiently updates the DOM automatically. And there's nothing extra that you have to do to enjoy this performance benefit. When you update the state of a component, it happens automatically. The comparison happens in memory, so it's typically very fast. Today, a variety of other libraries use a similar approach, but React's performance remains quite competitive. React offers various performance optimizations, but in my experience, they're rarely necessary. React is fast enough by default that many apps run great on the first try. Of course, the library's size also has an impact on performance too. React with React DOM weighs only 35K when gzipped and minified. So React lends itself well to bandwidth constraint use cases like mobile. It's significantly smaller than many of its competitors, but if you need the smallest library size possible, there are alternative React-like libraries that you can consider using as well. Inferno is a lightweight React alternative that only weighs 9K, but has a very similar API to React. And Preact is even smaller, weighing in at only 3K. Both Preact and Inferno are extremely similar to React and utilize the same API; they just omit a few features to help keep the size down. So these are great options if the bundle size is a high priority for you. And it's easy to switch to these later if desired.

Reason 6: Testability

The final reason is testability. Typically, testing a front end is hard. That's why so few teams do comprehensive front-end testing. But React is attractive and noteworthy because its design is very friendly to automated testing. As I mentioned, there's a number of reasons that teams do little to no automated UI testing today. Let's consider a few reasons why. Traditional UI testing is a hassle to configure. You have to carefully wire together multiple open source projects to get it to work. With React, there's little to no configuration required. In fact, if you use a popular boilerplate like create-react-app, testing is already configured for you out of the box. Traditional UI tests often require a browser, but you can reliably test your React components quickly in memory using node. And while traditional browser-based UI tests are often slow, testing React in memory on the command line is fast enough that you can quickly run a large test suite every time that you hit Save. Traditional UI tests are brittle because they operate more like integration tests, but with React, you can write reliable, deterministic unit tests that test a single React component in isolation. Finally, traditional UI tests are often time-consuming to write and maintain because you have to carefully interact with the browser and the DOM to test the UI. In contrast, React tests can be written quickly using popular tools like Jest and Enzyme. You can easily update your tests too, in many cases with a single keystroke after confirming that your output was changed as expected. With React, the vast majority of your components can be plain, pure functions. A pure

function always returns the same output for a given input. It has no side effects. React's functional component style makes it trivial to test your component, since it's pure. For example, if I set the message prop to world for this component, it will always output a div that contains the text Hello World. It's reliable, deterministic, and it has no side effects. It relies upon no global state. And while this is a simple example, this design can scale to highly complex user interfaces. There's a wide variety of JavaScript testing frameworks available, and since React is just JavaScript, any one of these will work just great. But for React, the most popular choice today is Jest. Jest was created by Facebook, and Jest makes it trivial to get started doing automated testing in React. Jest is built into create-react-app. Every time you hit Save, it automatically runs any affected tests. This instant feedback can rapidly speed development. Jest also offers unique features like snapshot testing that make it easy to store a text-based snapshot of your component's output. This way, if you accidently change the way your component renders, you're immediately notified. Here I'm using Jest to write an automated snapshot test. Note that I instantly see feedback on the command line as I hit Save, and it finds any new tests I write automatically since it looks for files that end in. test. js or. spec. js by default. No configuration required. And there are well-documented and powerful libraries like Enzyme from Airbnb that make it easy to manipulate and traverse your React component's output. This allows you to write tests that interact with your React components to assure that they operate as expected. And you don't even need to open up a browser since the tests will run in memory via Node. For more on testing in React, check out the testing module in Building Applications with React and Redux in ES6, as well as the testing module in Creating Reusable React Components.

Summary

In this module, we explored the many reasons that React has become so popular. It's exceptionally flexible. You can build web apps, native apps, desktops apps, virtual reality apps, and more. React provides a rapid feedback development experience that allows the developer to work with single components in isolation. You don't need the cognitive overhead and slow feedback loop that often occurs when maintaining separate interconnected files. Facebook has made a deep corporate investment in React including a full-time development staff, excellent documentation, and heavy public usage by Facebook itself on both facebook. com and Instagram. React has remarkable community support. It's one of the most popular projects on GitHub, and it's used by many of the most respected companies in the world. It offers excellent performance by default, and handy performance enhancements for the rare occasions that you need them. Finally, it's easy to test, especially with popular libraries like Jest that are built in to

popular boilerplates like create-react-apps. Okay, so this module was admittedly quite the sales pitch. Life can't all be rosy; you're right. So in the next module, let's consider the tradeoffs that are inherent in React.

Tradeoffs

Intro

I enjoy React, but I have to admit, I just gave you a one-sided sales pitch. So in this module, let me step back and be more balanced. Here's the plan. Let's consider six key tradeoffs that you accept when you choose React. These are the key tradeoffs that the React development team made when designing React. In the next few clips, let's consider the impact of these key tradeoffs so that you can better understand if React makes sense for your team,

Tradeoff 1: Framework vs. Library

The first tradeoff is framework versus library. Competitors like Angular and Ember are frameworks. React, in contrast, is generally considered a library, since it's lean and focused on components. Now a framework isn't fundamentally better than a library; it's a tradeoff. Here's a few advantages to choosing the framework approach. A framework contains more opinions, so you can avoid spending time trying to choose between many options. This reduces decision fatigue and there's often less setup overhead. Frameworks can help enforce consistency since most frameworks are more opinionated. However, React's library approach also has some clear advantages. At only around 35K gzipped, React is significantly smaller than most frameworks. This means that it's small enough that you can sprinkle it on existing applications so that you can slowly migrate an existing app to React, even a server-side rendered app. Imagine you have an existing app built in. Net, Java, Ruby, PHP, Python-- Whatever. Since React is small and flexible you can replace a single component on the page with a react component. So, you can use your React components anywhere because they're light-weight. This is precisely how Facebook slowly rendered from a server side rendered PHP application to React. And React doesn't force many decisions on you. It allows you to only pull in the features that you need to keep your app lean and fast. You're free to pick the precise technologies that you need for your project, and you're free to select the best technology for your use case as well. Decision fatigue is also largely a solved problem with React because opinionated boilerplates like create-react-app effectively turn

React into an optional framework. Now, since React is a focused component library, more comprehensive frameworks like Angular come bundled with more features, including testing, a library for HTTP calls, routing, and internationalization all built in. In contrast with React, you select the pieces that apply to your use case and you add them in. Since React is very popular, there's almost certainly a mature library that does what you need. Here's just a few of the most popular options for each use case. And the nice thing with React is your users don't have to waste time downloading and parsing features that they don't use. You can pull in only what you need from this list.

Tradeoff 2: Concise vs. Explicit

The second tradeoff to consider is concise versus explicit. React trades conciseness for predictability and explicitness. You spend a little more time explicitly wiring things up together, but that helps them not fall apart, and it also helps people better understand what the code is doing. Here's a concrete example. Frameworks like Knockout and Angular popularize two-way binding as a way to avoid typing by automatically keeping form inputs in sync with the underlying data. This approach was extremely popular until React came along. It was popular because it requires less coding. With two-way binding, JavaScript values and inputs are automatically kept in sync. In contrast, React embraces one-way binding instead. It requires a little more code. With React, you declare an explicit change handler and you reference it on your input. This extra work has some benefits. You have more control because you can declare precisely what should happen on every event. This means you can transform and validate input before updating state and perform performance optimizations as desired. Your code is more explicit since you clearly state what you want to happen when an event occurs, and this makes it easy to understand and debug when an error happens. Although React helped repopularize doing one-way binding, other popular libraries like Angular have shifted gears and embraced it as well today for these reasons. Now if your team strongly prefers two-way binding, you can use libraries that add it to React, but for the reasons I just mentioned, few do so, and I don't recommend it. Also, don't worry; you don't need to declare a separate change handler for each input. There are simple patterns for centralizing your change handlers in React, as I show in my other React courses. So in the real world, the amount of code that you write in React isn't substantially larger because of one-way binding because you typically have a single change handler for an entire form. In summary, here's the tradeoff. React requires more typing to implement than traditional two-way binding approaches, but with the benefit of easier maintenance, greater clarity, reliability, and performance.

Tradeoff 3: Template-centric vs. JavaScript-centric

The 3rd tradeoff is template-centric versus JavaScript-centric. In a previous clip, I contrasted the template-centric approach of traditional frameworks with React's JavaScript-centric approach. Angular, Vue, and Ember seek to make HTML more powerful by inventing their own unique syntax for writing code in HTML. React takes the exact opposite approach and instead utilizes the power of JavaScript to handle HTML. This fundamental difference is what makes React so elegant. Let's consider simple if logic for conditionally showing an h1 tag when a Boolean called isAdmin is true. Here I'm using Angular's unique ngIf directive, which must be prefixed with an asterisk. My conditional is written inside a string. With Vue, I use v-if, and again declare my conditional inside of a string. Ember also uses a unique syntax that looks a bit like JavaScript, but with no parentheses, and the entire statement must be wrapped in double curly braces. With React, I can use JavaScript's logical and operator. If you're not familiar with it, the right-hand side only runs when the left-hand side is true. So Hi Admin will only display when isAdmin is true. And since the code I'm writing inside the curly braces is ultimately just JavaScript in a JS file, any editor will provide autocomplete support along the way as I type and reference my functions. If I make a typo, I'll see warnings in my editor if I reference a variable that doesn't exist. Now let's consider writing a loop in each technology. With Angular, you say *ngFor, then use a syntax that looks a bit like JS, but is declared inside a string. Vue is similar, but a little less typing. Again, you write your looping code inside a string. With Ember, you use Ember's #each helper, which is rather wordy compared to the others above. Finally, with React, you use JavaScript's built-in map function and a plain JavaScript arrow function to display each user name. So the only React-specific syntax is the curly braces around user. name, so it's not just preferable because it's less code, it's preferable because the syntax is plain JavaScript. Finally, let's see how each handles clicking on a button. With Angular, you add parentheses around the event, and unlike traditional event handlers, you also put parentheses by the event handler name as though you're invoking it. This syntax wouldn't work if it were real JavaScript. With Vue, you use v-on with a colon and the event name, or you can put an at sign the front of click. Again, both of these are Vue-specific syntax. Finally, with Ember, you specify a plain onclick, which is nice, but inside, you use an Ember-specific convention to tie the click to a specific action, which is declared via a string. Finally, with React you declare an onClick handler. So in React, you use the native click handler name, but it's CamelCase'd since React's JSX uses JavaScript casing rules; otherwise, the only unique syntax is that you specify the function name inside a curly brace instead of in quotes. Place these together, and you can see the contrast. If you know JavaScript, then you know how to handle conditionals, looping, and events in React. This is why React's API is so small. So after

seeing the comparisons, let's contrast the benefits of each approach. The benefit of the template approach is that it requires less JavaScript knowledge. Template languages provide a streamlined API for performing core functions. You focus on enhancing a template with framework-specific syntax. And these unique syntaxes have some advantages like avoiding the confusion that's often caused by JavaScript's this keyword behavior. This makes templates more approachable for developers who are new to JavaScript. And in theory, a template language is preferable because of a principle called the rule of least power. It's counter-intuitive, but less powerful languages can theoretically be preferable because they can protect from misuse by only allowing the user to perform a small set of prescribed operations. For example, Angular's template syntax only supports a selected subset of JavaScript-like expressions. This helps avoid misuse. Of course, there's an obvious downside that we just saw, which is that the template-centric approach leads to framework-specific syntax. To get good at template-centric libraries like Angular and Vue, you must spend your time learning their specific syntax and rules for doing things that JavaScript can already handle. In contrast, React has very little unique syntax to learn, and since React embraces JavaScript, you don't have to learn a new vocabulary to describe the new features that other template-centric libraries like Angular add to HTML. Most of what you write in React is really just plain JavaScript. This leads to less typing and less code, which I find produces a result that's easier to read and debug. Finally, React's JavaScript-centric approach encourages improving your JavaScript skills. And admittedly, you could consider this a downside because the list of new features in ES 2015 was significant. If your team isn't yet up to speed, it can take a few weeks to get comfortable with all this. But the great news is, once you do, you won't just be better at React, you'll be better at JavaScript forevermore. In summary, to get better at React, you mostly need to learn modern JavaScript. And this is great because it means your skills will transfer to all other JavaScript code that you write, even if you move away from React in the future.

Tradeoff 4: Separate vs. Single File

The 4th tradeoff is a separate template versus a single file per component. Patterns like MVC popularize separating the model, view, and controller into three separate files. Traditionally for web apps, this means that the view is HTML. The model declares the data for the view in JavaScript, and the controller controls the interactions with the model. In contrast, with React, each component is an autonomous concern. Each component stands on its own and can be composed with other components to build rich, complex UIs. This means markup and logic are co-located in the same file. When React was introduced in 2013, people were very skeptical, and for good reason. React's design ran against the current best practice of placing HTML templates

and JavaScript logic in separate files. In React, each component contains both logic and markup in the same file, so on the surface this feels like it violates the principle of separation of concerns. However, in React you think about separation of concerns differently. Traditional separation of concerns often fixates on placing each technology in a separate file. So in web development, this means placing HTML, CSS, and JS in separate files. But React recognizes that while these are indeed separate technologies, they must be carefully composed together to do anything useful. So in React, each component is a separate concern. Examples include a button, a DatePicker, an accordion, or a TextInput. Each of these components is a separate concern. It will often embed logic, styling, and markup concerns because JavaScript, CSS, and JSX work together to create a useful component. In my experience, placing such intertwined concerns in separate files actually hinders debugging and slows feedback because you have to mentally keep these separate files in sync. And of course by composing small, simple components together, you can create more complex components like ContactForms, customer details, and so on. In my experience, components are the common concern worth separating. The old mindset of separating HTML, JavaScript, and CSS into separate files, merely separated technologies, but their concerns and interactions are actually fundamentally intertwined. If you change one file, it often requires corresponding changes to other files. Oh, and you're not forced to handle styling in the same file. With React, you're free to handle CSS in a traditional separate CSS file if you prefer. Think of component composition like Russian dolls. Russian dolls are interesting because each doll can hold a smaller doll inside. React's component model works the same way. Simple, reusable components can be composed together to build rich, complex user interfaces. With React, you can think of the page as a set of nested components. Consider an author page on Pluralsight.com. We could build this page in React by creating simple components like a StarRating, an AuthorPhoto, a navigation link, and so on. Then we could compose those simple components together to make the CourseSummary, the AuthorSummary, or the sidebar navigation. Finally, we could compose those larger components together to create even larger components like AuthorCourses. When you see all this displayed at the same time, you can see how the small components in orange are used as part of the larger components in blue. And the larger components in blue, like the CourseSummary, are composed together to create large portions of the app in green, like the AuthorCourses. As you can see, React's component approach lends itself well to building complex UIs by breaking your page down into small autonomous pieces that you can reason about and test in isolation.

Tradeoff 5: Standard vs. Non-standard

The 5th tradeoff is standard versus non-standard. React is just one of many non-standard component libraries. But the Web Components standard has actually been around for years without much usage yet. So why aren't many people building apps with standardized Web Components yet? Well, first, let's review what the Web Components standard is. The Web Components standard exists of four core technologies, templates, which contain markup; custom elements, which allow you to expand HTML with your own custom elements; the Shadow DOM; which encapsulates styling to avoid your styles leaking outside of your component; and imports, which bundle everything together into a single line that you can import onto the page. I published HTML5 Web Component Fundamentals in 2015, and I remain hopeful about standardized HTML components, but unfortunately, the standard has yet to be embraced much by the development community. Today, most developers continue to use libraries like Angular, Vue, and React instead. So the big question is, why hasn't the Web Components standard taken off yet? Well, likely the biggest reason that people aren't using plain Web Components is browser support remains spotty. The template tag isn't supported in any versions of Internet Explorer. HTML imports are only supported in Chrome, Opera, and Android. Custom elements are only supported in Chrome, Opera, and some of the newest Android browsers, and it's pretty much the same sad story with the Shadow DOM. After years of waiting, it's become clear: the browser vendors have shown little interest in implementing the full suite of HTML5 Web Component features. So you need to add polyfills to make it all work cross browser. And once you're pulling in extra JavaScript, you have to ask why you're choosing an unpopular and poorly-supported standard like Web Components instead of an extremely popular technology like React. Second, Web Components no longer enable anything new. Everything that you can do in Web Components can be accomplished today in a cross-browser-friendly way using a variety of modern JavaScript libraries including React. Let's consider the core features of the Web Components standard and contract them with React. Web Components have templates for holding your markup, and in React you use JSX to declare your markup, along with plain JavaScript. Web Components allow you to expand HTML by declaring custom elements. React components accomplish that same thing. The Shadow DOM lets you encapsulate your styles within your web component. Various technologies like CSS modules, CSS in JS, and even React's built-in inline styles assure that your styles are encapsulated automatically. Finally, Web Components offer HTML imports for bundling the component together into a single import. With React, each component is a single file, so you can always import it via a single line import. So in summary, Web Components enable nothing new compared to React today. The third reason Web Components aren't taking off is JavaScript libraries like React, Angular, and Vue offer compelling component stories that continue to improve. It's both difficult and unlikely for standards bodies to be able to keep up with the pace of JavaScript

innovation. Today, modern libraries offer a number of compelling features for binding, bundling, packing, and testing JavaScript-based components. Finally, since Web Components utilize standards built into a web browser, you can't use them elsewhere like on native mobile. React, in contrast, works on mobile and virtual reality, too, as well as a number of other interesting platforms. In summary, the Web Components standard may take off at some point, but currently, the majority of developers continue to reach for the tools on the right because they're innovating more quickly, offer a strong user experience, and run reliably cross-browser today.

Tradeoff 6: Community vs. Corporate

The final tradeoff we'll consider is community versus corporate backing. Many popular open source JavaScript projects are community-driven. React is an open source project, but it's backed and actively maintained by Facebook. This means that React is driven by Facebook's needs, so if your apps are very different from what Facebook is building, React may not be ideal for your use case. But as I alluded to earlier, React's corporate backing has some clear benefits. Facebook provides React a full-time development staff that carefully plans releases, writes documentation and blog posts, and provides ongoing support for issues on GitHub, and often even on social media. Even though React has a full-time staff, there are over a thousand contributors to React, so the community is vibrant and engaged. Facebook is currently wildly successful. At the time of this recording, it's the fifth most valuable company in the world, with a market capitalization of $445 billion. Even if Facebook were no longer involved in React, the project would certainly live on without full-time Facebook involvement for the foreseeable future. Plus, many competitors lack the benefit of the full-time corporate development staff that React boasts today. So while React's corporate tie is a potential concern, it's also a significant benefit for support and ongoing development for the foreseeable future. Finally, as I mentioned earlier, Facebook is deeply invested in React, with over 30, 000 components in production, so as long as Facebook is around, they're likely going to continue providing significant ongoing support for the project. And that's it. Those are the 6 tradeoffs to consider with React. Let's close out this module with a summary.

Summary

In this module, we looked at 6 tradeoffs to consider with React. Frameworks offer more opinion and standardization, but React's library approach allows you to select only the tools that you need, and pick the best tools for your use case. Other frameworks strive to be concise using

techniques like two-way binding and abstractions over JavaScript operations. But React is explicit, so code is more readable and scalable at the admitted expense of doing a little more typing on the keyboard. React chooses to be JavaScript-centric instead of template-centric. React's JavaScript-centric approach is easier to understand and debug and requires learning less unique syntax, but at the cost of requiring modern JavaScript knowledge. Many frameworks utilize a separate template file. In contrast, each React component is a single, autonomous file that you can work with and test in isolation. The Web Components standard has been around for years, yet it continues to lack broad adoption. Non-standard approaches like React and Angular remain more popular because they offer the same power, more rapid innovation, and a superior developer experience. And React is corporate backed, which means its design is influenced by Facebook's needs. But Facebook continues to accept input from the community, and has evolved React into a highly flexible and well-supported system. Of course with tradeoffs there's no right answer, but at least now you understand what you're getting and what you're giving up with each of these tradeoffs. In the next module, I'll be even more critical. Let's explore the common concerns that I hear about with React so that you can decide if the downsides that other people talk about matter to you.

Why Not React?

Intro

If I'm going to share the big picture with you, I owe it to you to be completely up front about the potential issues with choosing React. So in this module, I'll share the common concerns I hear about React. JSX differs from HTML, React requires a build step, there's a potential for version conflicts, and React has evolved over time so you'll find references to old features when searching the web, and since React is a lightweight library, you may feel intimidated by the number of decisions you need to make up front. Oh, and you might have heard that React's license has a patent clause. Well, great news, that's no longer an issue. With the release of React 16, Facebook relicensed React to use the standard MIT open source license, so there's nothing to worry about there anymore. That said, you'll see that some of the other concerns above are valid and others are merely misconceptions or issues that can be easily mitigated.

Concern 1: HTML and JSX Differ

The first common concern that I hear is that JSX differs from HTML. As I showed earlier, React uses an optional syntax that looks a lot like HTML called JSX. I say optional because JSX needs to be compiled down to plain JavaScript so that the browser can understand it. The code on the bottom is what is sent to the browser, so you're free to just use this syntax on the bottom if you prefer. But most people prefer using JSX since it feels extremely similar to HTML and it's easier to read. Now JSX's syntax is 99% the same as HTML, but it does differ in a few ways such as htmlFor instead of for, and className instead of class, inline styles are declared in JSON format, and finally, comments are handled in JavaScript style instead of HTML style. So as you can see, learning JSX's differences is trivial. The list of differences is quite short, and thus easy to adopt. Now maybe you're still worried. But I have a lot of existing HTML. Won't that be a hassle to convert into JXS? Well, the good news is, that's easy; there are multiple ways to handle it. Honestly, since there are only a few differences between HTML and JSX, I typically convert existing HTML with a simple find and replace. The list of differences is so small that it typically only takes me a moment. But there's also an online HTML to JSX compiler and an npm package called HTML to JSX that you can use. With this online HTML to JSX compiler, you enter your HTML on the left and it displays the resulting JSX on the right. As you can see, the two sides are quite similar, but I deliberately created an example on the left to show the core differences. Now if you have a lot of HTML to convert to JSX, you can convert it in bulk using the HTML to JSX package on npm. This is a command line tool and it uses the same code that's running behind the scenes on the online HTML to JSX compiler that I just showed you. So you can point this at a set of files and convert your HTML in bulk.

Concern 2: Build Step Required

The second common concern is closely related to the first. When you use JSX, React requires a build step. As I mentioned, you need to compile JSX down to plain JavaScript calls so that the browser can understand it. In practice, this is trivial to handle today. Frankly, worrying about needing a build step makes little sense today. These days, a build step is a critical part of just about any web application. No matter what JavaScript framework you're using, you're going to want a build step. You'll want to minify your code to save bandwidth. You'll want to transpile your code so that you can use modern JavaScript features today, even before your platform offers full support. And you'll want to lint your code and run your automated tests when you hit Save as well. So compiling JSX is just another thing that happens automatically along the way. Today's two most popular transpilers both work great with React, Babel and TypeScript, and both allow you to use modern JavaScript features today even before all browsers offer full support. And

most importantly for React developers, both transpile React's JSX for you. And there are a variety of React boilerplates that make it easy to get started and have build steps built in automatically to transpile JSX for you. Create-react-app, which I've mentioned earlier, is the most popular option today. And it also minifies and bundles your code, and even includes automated testing support so you don't even need to configure your development or production build environment to work with JSX. This all just happens automatically.

Concern 3: Version Conflicts

The third concern is potential version conflicts. As I mentioned earlier, React with React DOM weighs only around 35K minified and gzipped. That's a very reasonable size, but there's some downsides to having a runtime at all. See, you can't run two versions of React at the same time on the same page, so this means that you need to keep your React components on the same version for a given page. In contrast, if you build standardized Web Components, you don't have to worry about version conflicts at all since there's no runtime. Standard Web Components just leverage the support that's built right into the browser. However, for reasons I outlined in a previous module, I still prefer working in React over using plain Web Components, and that's partially because the Web Components standard lacks features that I've grown to know and love from React and other frameworks like efficient DOM updates, reactive data binding, and more. So there are other interesting tools to consider like SkateJS, Svelte, and Stencil, which bring those features to Web Components without the need for a framework. These options are interesting because they leverage the Web Components standard, but add extra features. They also embed their runtime within each component so you don't need to worry about version conflicts. And since React is a lean component library, you will often choose to use related libraries with React such as React Router, so you need to run compatible versions. This means that you typically need to run a recent version of React to avoid a version conflict. For example, today if you want to use the newest version of React Router with your React app, you need to run React 15 or newer. In practice though, I've found version conflicts are rarely a problem in React. Since Facebook has been consistent about releasing codemods when breaking releases occur, upgrades to your existing React components can typically be easily automated. Here are three tips to avoid version conflicts. First, agree as a team which version of React you're working with. Second, upgrade React when you're upgrading related libraries. And finally, on the rare instances where there are breaking changes in a future release, decide as a team when to upgrade.

Concern 4: Outdated Resources Online

The 4th concern is old stuff showing up in searches. React has a large, mature community, and it has evolved since it was open sourced in 2013. Search for the term react example on Google, and you get 1. 7 million results. There are over 57, 000 questions tagged with react on Stack Overflow and around 30, 000 questions on related technologies like React Native, React Router, and Redux. And there are thousands of blog posts out there on blogging platforms like Medium about React and related technologies. And hey, having many great resources online is a great thing, right? Absolutely. But there's an obvious risk. Some of this public content is outdated. React has evolved since it was released in 2013, so as you search around the web, you'll see some patterns that are no longer popular today. So what has changed? Well recently, features have been extracted from React Core to keep the library lean and simple. Since React is used for more than just the web now, React DOM was extracted to a separate package, so you'll see many posts using the style on the left, but today, reference the separate React DOM library for web development. And if you're doing development for other platforms, you'll import the renderer that's appropriate for that platform such as React Native. Second, since most people are using ES classes today, react. createClass was extracted to a separate library called create-react-class, so you need to reference this separate library if you want to declare React components using the createClass style. And since only some teams choose to use PropTypes over alternatives like TypeScript and Flow, the PropTypes library was extracted to a separate npm package, too. So you'll need to install the PropTypes library and import it like you see here on the right. Finally, mixins were initially a popular way to share functionally between components in React. However, due to various issues, mixins are no longer part of React Core. Today, alternative patterns like higher ordered components and render props serve this purpose. Now this is an advanced topic, so don't worry if that makes no sense at all. The bottom line is, when you see an old point that mentions mixins, look into these alternative patterns instead. So keep these four changes in mind. You'll see the old-style imports on the left in older blog posts and videos, but as you can see, it's trivial to update the old code to reference the separate packages on the right. And of course, when you're in doubt, check React's documentation. It's excellent, up to date, and actively maintained by Facebook.

Concern 5: Decision Fatigue

[Autogenerated] the fifth and final concern is decision fatigue. React, his lightweight. None opinionated. So there are multiple ways to do some of the same things. Okay, I'm about to run through a few decisions, but keep in mind one convert you having many decisions is a glass half empty or a glass half full. I like having lots of options because it means that I can set things up in a

way that's perfect for my team's unique needs and preferences. So I see reacts rich ecosystem as a big win getting started in react. Kim feel intimidating with so many options, but let me break this down for you. There are five key decisions to make up front. Deaf environment. Yes, class or create class types, state and styling. Let's walk through these five key decisions. The first decision is a development environment. There are over 100 boilerplate projects available on Get Hub, and Andrew Farmer was nice enough to create a searchable list of react boiler plates on his website. You can search through the list for things that you want and things that you don't when starting react. This tool is a handy way to see all the different ways that you could choose to configure your team's react development approach. But I recommend starting with Create React app, which is the official development environment that supported by Facebook. This is a mature platform for rapidly creating react applications. This project includes automated testing, trance piling, bundling lint ing and an automated build process all set up and ready to go In a poll of over 2000 react developers, 65% use create react out. The next largest group of 25% chooses to build their own dev environment. I show how to build your own environment in my react and Redox course. And if you're wanting to build a native mobile app with react, I suggest using, create react native app now admittedly create react app doesn't include all of your decisions baked in. For example, related libraries like react router for routing or reduction for state management aren't included and create react at. But you don't need these to get started. So look into these related libraries later. Okay, on decision to you need to decide whether to use E s classes or the original create class in tax. Here's what the two approaches look like side by side. Now you can accomplish the same things in both, so it really just comes down to personal preference. Each has its advantages. The create class style is friendlier to beginners, since it avoids confusion and around Java scripts. This key word by auto binding functions and doesn't require knowledge of modern JavaScript features from PS 2015. Both worked great, but I prefer using yes classes, since it doesn't require pulling in an extra library. And it's the most popular approach today. That said, I used the create class approach in my react influx course, and I used the S class approach in my react and Redox course so you can check out both styles and see which you prefer. There are three popular ways to handle types. React prop types, typescript and float. To see the difference between these options, let's consider a simple component called greeting. With proper types, you can declare the types of the data that are passed into your component here. I'm declaring that the name passed into the greeting component is a string with proper types types air checked on Lee at runtime and on Lee during development. Typescript is the second option. It's a popular project for Microsoft. Typescript is a super set of Java script that adds strong typing support and compiles down to plain JavaScript. Typescript will feel very familiar at anyone who is coated in C. Sharp or Java. Here, I'm using an interface to declare the type for the greeting components props,

and I'm saying the props argument getting passed into my greeting component has a type of props. Unlike prop types, typescript checks types at compile time, so you find out earlier about any potential type issues. The third option is Flo, a project from Facebook for adding static type. Checking to your Java script flow provides type safety in a different way than typescript. With Flo, you had type annotations to plain JavaScript and flow intelligently and first types throughout your code base. With Flo, you annotate the top of each file that you'd like it to check. Here. I'm declaring my type above the component, then specifying the type within the argument. Floegel process plain JavaScript without type imitations, and it will in for those types or, in other words, flow through your code. Now flow runs a separate process so types can be checked whenever you choose to run flow. So these air three great options. But I recommend prop types for developers getting started with react because they're simple to implement, trivial, to learn and require no special configuration to get started. Around 40% of react developers reported using proper types in a recent poll. React works great alone, but many developers prefer to enhance react with popular third party state libraries. Now what do I mean by state? While Simple state is your APS data? Popular ways to handle state and react include Plane React, flux reduction and Ma Becks. React handles state great all by itself, so these other libraries are totally optional with plain react. Your components handles state on their own. But shortly after react was open sourced Facebook. Also released flux is an optional way to handle state and react. Flux is still heavily used by Facebook, and it centralizes your application state. But today Redox is the most popular state management library for working in react like flux with Redox. Your APS data is centralized, but Redox offers a more elegant approach than flux and uses an immutable data store. Finally, Mob X is a lighter weight alternative to re ducks, but with a fundamentally different take on state management. Mob excuses, observable data structures. In short, Redox is more explicit and scalable. But Ma Becks requires less code and is easier to learn. That said, I want to emphasize react works great alone. You can build applications using just plain react. You often don't need a separate state management library like Redox, I recommend starting with plain react. As you can see, Redox is the most popular approach, but I still recommend starting with plain react and then learning Redox later. Don't feel obligated to immediately learn and use Redox or Ma Becks. You can build powerful APS using just react by itself, and this final decision is where it gets a little silly. There are over 50 different styling approaches that you could potentially use with react, but really react works great with traditional CSS sass and less to So my suggestion is just get started by using whatever you know today. And in fact, the majority of react developers continue to use traditional styling approaches with react after you've gotten comfortable with react. If you're curious about the styling options. I explore them in detail in my creating reusable react components. Course. So in summary, yes, the benefit of react being a lightweight library is

that you have options. But the obvious downside is the number of options can feel intimidating. So I'd encourage you to use these recommendations as a starting point for standardizing your approach. And don't let these decisions intimidate you. The implications of these decisions are really mostly minor anyway. Okay, let me wrap up the course by suggesting some next steps.

Next Steps

And that's a wrap. Now assuming that you're excited about moving forward with React, let me share some next steps to consider. Step 1 is to learn React and its ecosystem. I introduce React from scratch in my React and Flux course, and you don't need to know modern JavaScript since I use plain ES5 JavaScript in this course. Step 2 is to learn how to use modern JavaScript with React. I show how to work with React using modern JavaScript in my React and Redux in ES6 course. Finally, step 3 is totally optional, but for larger and more complex apps, many React developers prefer to add Redux or MobX. I explore Redux in detail in the second half of my React and Redux course. And if you're interested in building your own development environment from scratch, check out the Building a JavaScript Development Environment course. I also show how to build a React development environment from scratch using the most popular React tooling choices at the beginning of my React and Redux course as well. If you're interested in quickly moving your team to React, I specialize in remote and onsite React consulting services tailored to your team's needs. Learn more at reactjsconsulting. com.

Summary

In this module, we considered the common concerns that I hear about React today. Yes, JSX differs from HTML in some minor ways, but it's easy to convert using find and replace or handy existing tools. React typically requires a build step to compile JSX, but any modern JavaScript app should have a build step anyway to handle minification, bundling, and so on. And all popular React boilerplates such as create-react-app automatically transpile JSX for you. You can only run one version of React on a given page, so there's a risk of version conflicts, but in practice, this isn't a big deal because it's easy to upgrade your React components to the latest version using Facebook's codemod project when necessary. Now since React is popular, you will stumble across old features in searches, so I noted the key changes that you should understand to avoid getting confused. Finally, decision fatigue is definitely a valid concern with React, so my suggestion is to start simple. Just use create-react-app and my other recommendations to get

started. They're the most popular approaches to doing React today. Add the complexity of other tools like Redux only as needed later.

Course author

Cory House

Cory is the principal consultant at reactjsconsulting.com, where he has helped dozens of companies transition to React. Cory has trained over 10,000 software developers at events and businesses...

Course info

Level	Beginner
Rating	★★★★½ (465)
My rating	★★★★★
Duration	1h 11m
Released	21 Nov 2017

Share course