INTERVIEW QUESTIONS

**1.**

**Explain the Virtual DOM, and a pragmatic overview of how React renders it to the DOM.**

Hide answer

The Virtual DOM is an interesting concept; it’s a complex idea that boils down into a much simpler algorithm.

In React, if we create simple ES6 class and print it out, we have a function (as all functions can be used as a constructor in JavaScript):

**const** **app** = () => {

**let** **React** = react,

{**Component**} = **React**,

DOM = reactDom

**class** **Comments** **extends** **Component** {

**constructor**(props){ super(props) }

**render**(){ **return** <**div**>test</**div**> }

}

console.**log**(**Comments**)

}

require('react', 'react-dom').**then**(app)

The console.log(Comments) gives us something that looks like this (after compiled by Babel from ES6 to ES5):

**function** **Comments**(props) {

**\_classCallCheck**(this, **Comments**);

**return** **\_possibleConstructorReturn**(this, **Object**.**getPrototypeOf**(**Comments**).**call**(this, props));

}

When we write something to draw a React Component to the screen, we might have something like the following:

DOM.**render**(<**Comments** />, document.body)

The JSX gets transpiled into ES5 by Babel as well:

DOM.**render**(**React**.**createElement**(**Comments**, null), document.body);

We can see that <Comments /> is transpiled directly into React.createElement(Comments, null). This is where we can see what a Virtual DOM object actually *is*: a plain JavaScript Object that represents the tag to be rendered onto the screen.

Let’s inspect the output of React.createElement():

console.**log**(<**div**/>)

// or

console.**log**(**React**.**createElement**('div', null))

This gives us:

{"type":"div","key":null,"ref":null,"props":{},"\_owner":null,"\_store":{}}

See how the type is a string? DOM.render({...}) gets this object above and looks at the type, and decides whether or not to reuse an existing <div> element on the DOM or create a new <div> and append it.

The Virtual DOM is not a simple Object – it is a recursive structure. For example, if we add two elements beneath the <div/>:

console.**log**(<**div**><**span**/><**button**/></**div**>)

// or

console.**log**(**React**.**createElement**(

'div',

null,

**React**.**createElement**('span', null),

**React**.**createElement**('button', null)

))

What we get is a nested Object-tree:

{

"type":"div",

"key":null,

"ref":null,

"props":{

"children": [

{"type":"span","key":null,"ref":null,"props":{}},

{"type":"button","key":null,"ref":null,"props":{}}

]

}

}

This is why, in a React Component’s code, we can access the child and ancestor elements via this.props.children. What React will do is walk down a very deep tree of nested Objects (depending on your UI complexity), each sitting in their parent element’s children.

One thing to note is that the type so far has just been a string. When a React Element is made from a custom Component (like Comments above), the type is a function:

console.**log**(<**Comments** />)

// or

console.**log**(**React**.**createElement**(**Comments**, null))

gives us:

{

"key":null,

"ref":null,

"props":{},

“type”: **function** **Component**() { ... }

}

You can play around with a web version of this code [at Matthew Keas’ github](https://goo.gl/HZZMjv).

**2.**

**Explain the standard JavaScript toolchain, transpilation (via Babel or other compilers), JSX, and these items’ significance in recent development. What sort of tools might you use in the build steps to optimize the compiled output React code?**

Hide answer

The bleeding edge JavaScript toolchain can seem quite complex, and it’s very important to feel confident in the toolchain and to have a mental picture of how the pieces fit together.

There are a couple primary pillars in the JavaScript toolchain: Dependency Management, Linting, Style-checking, Transpilation, and Compilation, Minification, Source-Mapping.

Typically, we use build tools like Gulp, Watchify/Browserify, Broccoli, or Webpack to *watch the filesystem* for file events (like when you add or edit a file). After this occurs, the build tool is configured to carry out a group of *sequential or parallel tasks*.

This part is the most complex piece, and is the center of the development process.

The rest of the tools belong in that group of sequential or parallel tasks:

* Style linting - typically a linter like JSCS is used to ensure the source code is following a certain structure and style
* Dependency Management - for JavaScript projects, most people use other packages from npm; some plugins exist for build systems (e.g. Webpack) and compilers (e.g. Babel) that allow automatic installation of packages being imported or require()‘d
* Transpilation - a specific sub-genre of compilation, transpilation involves compiling code from one source version to another, only to a similar runtime level (e.g. ES6 to ES5)
* Compilation - specifically separate from transpiling ES6 and JSX to ES5, is the act of including assets, processing CSS files as JSON, or other mechanisms that can load and inject external assets and code into a file. In addition, there are all sorts of build steps that can analyze your code and even optimize it for you.
* Minification and Compression - typically part of – but not exclusively controlled by – compilation, is the act of minifying and compressing a JS file into fewer and/or smaller files
* Source-Mapping - another optional part of compilation is building source maps, which help identify the line in the original source code that corresponds with the line in the output code (i.e. where an error occurred)

For React, there are specific build tool plugins, such as the [babel-react-optimize presets](https://github.com/thejameskyle/babel-react-optimize) that involves compiling code into a format that optimizes React, such as automatically compiling any React.createElement() calls into a JavaScript Object that inlines right into the source code:

**class** **MyComponent** **extends** **React.Component** {

**render**() {

**return** (

<**div** className={this.props.className}>

<**span**>Hello World</**span**>

</**div**>

);

}

}

becomes

**class** **MyComponent** **extends** **React.Component** {

**render**() {

**return** (

**\_jsx**('div', { className: this.props.className }, **void** 0,

**\_jsx**('span', {}, **void** 0, 'Hello World')

)

);

}

}

See also:

* How compilers can help [optimize React](https://medium.com/doctolib-engineering/improve-react-performance-with-babel-16f1becfaa25#.ql2io6j01)
* How to bootstrap a modern toolchain with [Create React App](https://create-react-app.dev/)
* The [Next.js](https://nextjs.org/) and [GatsbyJS](https://www.gatsbyjs.org/) frameworks built on top of React

**3.**

**How would you create Higher Order Components (HOCs) in React?**

Hide answer

Higher Order Components (HOCs) are the coined term for a custom Component that accepts dynamically provided children. For example, let’s make <LazyLoad /> Component that takes child image tags as children, waits until the <LazyLoad /> Component is scrolled into view, and then loads the images they point to in the background (before rendering them to the DOM).

An HOC accepts children via props:

DOM.**render**(

<**LazyLoad**>

<**img** src="https://media.giphy.com/media/HhvUpQhBWMtwc/200.gif"/>

<**img** src="https://media2.giphy.com/media/3oEduUDvycvu3GYkdG/200w.gif"/>

<**img** src="https://media0.giphy.com/media/142UITjG5GjIRi/200w.gif" />

</**LazyLoad**>,

document.body)

Creating an HOC means handling this.props.children in the Component’s code:

interactive example can be found at <https://goo.gl/ns0B6j>

**class** **LazyLoad** **extends** **Component** {

**constructor**(p){

super(p)

this.state = { loaded:0 }

this.\_scroll = this.\_scroll.**bind**(this)

}

**\_scroll**(){

**let** el = DOM.**findDOMNode**(this)

**let** {top} = el.**getBoundingClientRect**()

**let** viewportHeight = **Math**.**max**(document.documentElement.clientHeight, window.innerHeight || 0)

**if**(top < (viewportHeight + this.props.top)) {

window.**removeEventListener**('scroll', this.\_scroll)

this.**setState**({loaded:1})

}

}

**componentDidMount**(){

window.**addEventListener**('scroll', this.\_scroll)

this.**\_scroll**()

}

**componentWillUnmount**(){

window.**removeEventListener**('scroll', this.\_scroll)

}

**render**(){

**let** {children} = this.props,

{loaded} = this.state

**return** <**div**>

{loaded && children}

</**div**>

}

}

**LazyLoad**.defaultProps = {

top: 100

}

Noting a few things about this code:

1. We set up initial state (this.state = {loaded: 0}) in the constructor(). This will be set to 1 when the parent container is scrolled into view.
2. The render() returns the props.children as child elements when this occurs. Extract the src by using ES6 destructuring, where {props:{src}} creates a variable src with the appropriate value.
3. We used a single componentDidMount() lifecycle method. This is used because on mount, we’d like the component to check if the HOC is visible.
4. The largest function of our component, \_scroll(), grabs the HOC Component’s DOM element with DOM.findDOMNode() and then gets the elements position. This position is compared to the height of the browser window, and if it is less than 100px from the bottom, then the scroll listener is removed and loaded is set to 1.

This technique is called HOC (Higher Order Component) because we pass in elements as this.props.children when we nest those elements inside the container component:

<HOC>

<**div**>some</**div**>

<**span**>children</**span**>

<**Props**/>

</HOC>

All of these nested elements (which can be custom components) are nested under <HOC/>, thus HOC’s code will be able to access them as this.props.children.

**Apply to Join Toptal's Development Network**

and enjoy reliable, steady, remote [Freelance React.js Developer Jobs](https://www.toptal.com/freelance-jobs/developers/react)

[**Apply as a Freelancer**](https://www.toptal.com/talent/apply)

**4.**

**What is the significance of keys in React?**

Hide answer

Keys in React are used to identify unique VDOM Elements with their corresponding data driving the UI; having them helps React optimize rendering by recycling existing DOM elements. Let’s look at an example to portray this.

We have two <TwitterUser> Components being rendered to a page, drawn in decreasing order of followers:

-----------

**| A - 103 |**

**-----------**

-----------

**| B - 92 |**

**-----------**

Let’s say that B gets updated with 105 Twitter followers, so the app re-renders, and switches the ordering of A and B:

-----------

**| B - 105 |**

**-----------**

-----------

**| A - 103 |**

**-----------**

Without keys, React would primarily re-render both <TwitterUser> Elements in the DOM. It would re-use DOM elements, but React won’t **re-order** DOM Elements on the screen.

With keys, React would actually re-order the DOM elements, instead of rendering a lot of nested DOM changes. This can serve as a huge performance enhancement, especially if the DOM and VDOM/React Elements being used are costly to render.

Keys themselves should be a unique number or string; so if a React Component is the only child with its key, then React will repurpose the DOM Element represented by that key in future calls to render().

Let’s demonstrate this with a simple list of todos rendered with React:

Interactive code sample available on [Matthew Keas’ github](https://goo.gl/fpAvSc).

**class** **List** **extends** **Component** {

**constructor**(p){

super(p)

this.state = {

items: **Array**(5).**fill**(1).**map**((x,i) => ({id:i}))

}

}

**componentDidMount**(){

**const** **random** = (a,b) => **Math**.**random**() < .5 ? -1 : 1

setInterval(() => {

this.**setState**({

items: this.state.items.**sort**(random)

})

}, 20)

}

**render**() {

**let** {items} = this.state

**return** <**ul**>

{items.map(item =>

<**li** key={item.id}>{item.id}</**li**>)}

</**ul**>

}

}

DOM.**render**(<**List** />, document.body)

The setInterval() occurring on mount reorders the items array in this.state every 20ms. Computationally, if React is reordering the items in state, then it would manipulate the DOM elements themselves instead of “dragging” them around between positions in the <ul>.

It is worth noting here that if you render a homogenous array of children – such as the <li>’s above – React will actually console.warn() you of the potential issue, giving you a stack trace and line number to debug from. You won’t have to worry about React quietly breaking.

**5.**

**What is the significance of refs in React?**

Hide answer

Similarly to keys, refs are added as an attribute to a React.createElement() call, such as <li ref="someName"/>. The ref serves a different purpose, it provides us quick and simple access to the DOM Element represented by a React Element.

Refs can be either a string or a function. Using a string will tell React to automatically store the DOM Element as this.refs[refValue]. For example:

**class** **List** **extends** **Component** {

**constructor**(p){

super(p)

}

**\_printValue**(){

console.**log**(this.refs.someThing.value)

}

**render**() {

**return** <**div** onClick={e => this.\_printValue()}>

<**p**>test</**p**>

<**input** type="text" ref="someThing" />

</**div**>

}

}

DOM.**render**(<**List** />, document.body)

this.refs.someThing inside componentDidUpdate() used to refer to a special identifier that we could use with React.findDOMNode(refObject) – which would provide us with the DOM node that exists on the DOM at this very specific instance in time. Now, React automatically attaches the DOM node to the ref, meaning that this.refs.someThing will directly point to a DOM Element instance.

Additionally, a ref can be a function that takes a single input. This is a more dynamic means for you assign and store the DOM nodes as variables in your code. For example:

**class** **List** **extends** **Component** {

**constructor**(p){

super(p)

}

**\_printValue**(){

console.**log**(this.myTextInput.value)

}

**render**() {

**return** <**div** onClick={e => this.\_printValue()}>

<**p**>test</**p**>

<**input** type="text" ref={node => this.myTextInput = node} />

</**div**>

}

}

DOM.**render**(<**List** />, document.body)

**6.**

**[Legacy projects only, < circa 2016] In a general overview, how might React Router and its techniques differ from more traditional JavaScript routers like Backbone’s Router?**

Hide answer

“Traditional” routers like the ever-popular [Backbone.Router](http://backbonejs.org/" \l "Router) establish a predefined set of routes, in which each route defines a series of actions to take when a route is triggered. When combining Backbone.Router with React, you may have to mount and unmount React Components when the route changes:

**var** **MyRouter** = **Backbone**.Router.**extend**({

routes: {

'home': 'showHome',

'search/:q': 'showSearch',

'\*default': 'show404'

},

**showHome**(){

DOM.**unmountComponentAtNode**(document.body)

DOM.**render**(<**Home** />, document.body)

},

**showSearch**(q){

DOM.**unmountComponentAtNode**(document.body)

DOM.**render**(<**Search** query={q} />, document.body)

},

**show404**(){

DOM.**unmountComponentAtNode**(document.body)

DOM.**render**(<**Error** />, document.body)

}

})

The router exists externally of the React Components, and the VDOM has to mount and unmount potentially frequently, introducing a possible slew of problems. React Router focuses on not just “single-level” routing, but enables - nay, *empowers* - the creation of HOCs that can “decide for themselves” what to render within them.

This is where the advanced HOC implementations can really help simplify a seemingly complex notion. Let’s look at using a tiny router to assess some of the beauty of embedding application routers inside React HOCs. Here, we define a Component that wraps it’s own routing mechanism (router() not provided here, see [universal-utils](https://github.com/matthiasak/universal-utils/blob/master/src/router-alt.js)):

// router(routesObject, callback) --> when a route event occurs, we invoke callback() with

// the React Element and the props passed via the route params

**class** **Router** **extends** **Component** {

**constructor**(...a){

super(...a)

**let** p = this.props

this.state = {

routes: p.routes || {},

default: p.default || '/'

}

this.router = **router**(this.state.routes, (el, props) => {

this.current = el

})

this.router.**trigger**(this.state.default)

}

**render**(){

**return** this.**current**()

}

}

DOM.**render**(<**Router** routes={{

'/': () => <**Home**/>,

'/search/:q': ({q}) => <**Search** query={q} />,

'\*': () => <**Error** />

}}/>, document.body)

This Router Component opts for parsing the routes object passed into this.props instead of reading over an array of React Components passed as this.props.children. React Router opts for the latter technique. Need proof? Take a look at this example code provided by [React Router’s docs](https://github.com/reactjs/react-router):

DOM.**render**(

<**Router** history={browserHistory}>

<**Route** path="/" component={App}>

<**Route** path="about" component={About}/>

<**Route** path="users" component={Users}>

<**Route** path="/user/:userId" component={User}/>

</**Route**>

<**Route** path="\*" component={NoMatch}/>

</**Route**>

</**Router**>

, document.body)

A <Router /> Component has one or more <Route /> Components as items in this.props.children, and <Route />s can have sub-<Route />s. React Router’s code recursively walks down the tree of children until there are no more to process, allowing the developer to recursively declare routes in a structure that encapsulates sub-routes, instead of having to implement a Backbone-esque flat list of routes (i.e. "/", "/about", "/users", "/users/:id", etc).

**7.**

**Why do class methods need to be bound to a class instance, and how can you avoid the need for binding?**

Hide answer

In JavaScript, the value of this changes depending on the current context. Within React class component methods, developers normally expect this to refer to the current instance of a component, so it is necessary to bind these methods to the instance. Normally this is done in the constructor—for example:

**class** **SubmitButton** **extends** **React.Component** {

**constructor**(props) {

super(props);

this.state = {

isFormSubmitted: false

};

this.handleSubmit = this.handleSubmit.**bind**(this);

}

**handleSubmit**() {

this.**setState**({

isFormSubmitted: true

});

}

**render**() {

**return** (

<**button** onClick={this.handleSubmit}>Submit</**button**>

)

}

}

There are several common approaches used to avoid this binding:

1. Define Your Event Handler as an Inline Arrow Function

For example:

**class** **SubmitButton** **extends** **React.Component** {

**constructor**(props) {

super(props);

this.state = {

isFormSubmitted: false

};

}

**render**() {

**return** (

<**button** onClick={() => {

this.setState({ isFormSubmitted: true });

}}>Submit</**button**>

)

}

}

Using an arrow function like this works because arrow functions do not have their own this context. Instead, this will refer to the context in which the arrow function was defined—in this case, the current instance of SubmitButton.

2. Define Your Event Handler as an Arrow Function Assigned to a Class Field

**class** **SubmitButton** **extends** **React.Component** {

state = {

isFormSubmitted: false

}

handleSubmit = () => {

this.**setState**({

isFormSubmitted: true

});

}

**render**() {

**return** (

<**button** onClick={this.handleSubmit}>Submit</**button**>

)

}

}

*Note: As of September 2019, class fields are a Stage 3 ECMAScript proposal and are not yet part of the published ECMAScript specification. However, they are available for use in both Google Chrome and Mozilla Firefox and are commonly used in React projects.*

3. Use a Function Component with Hooks

Using the hooks functionality in React it is possible to use state without using this, which simplifies component implementation and unit testing.

For example:

**const** **SubmitButton** = () => {

**const** [isFormSubmitted, setIsFormSubmitted] = **useState**(false);

**return** (

<**button** onClick={() => {

setIsFormSubmitted(true);

}}>Submit</**button**>

)

};

**8.**

**Explain the positives and negatives of shallow rendering components in tests.**

Hide answer

Positives:

* It is faster to shallow render a component than to fully render it. When a React project contains a large number of components, this performance difference can have a significant impact on the total time taken for unit tests to execute.
* Shallow rendering prevents testing outside the boundaries of the component being tested—a best practice of unit testing.

Negatives:

* Shallow rendering is less similar to real-world usage of a component as part of an application, so it may not catch certain problems. Take the example of a <House /> component that renders a <LivingRoom /> component. Within a real application, if the <LivingRoom /> component is broken and throws an error, then <House /> would fail to render. However, if the unit tests of <House /> only use shallow rendering, then this issue will not be identified unless <LivingRoom /> is also covered with unit tests.

**9.**

**If you wanted a component to perform an action only once when the component initially rendered—e.g., make a web analytics call—how would you achieve this with a class component? And how would you achieve it with a function component?**

Hide answer

Using a Class Component

The componentDidMount() lifecycle hook can be used with class components:

**class** **Homepage** **extends** **React.Component** {

**componentDidMount**() {

**trackPageView**('Homepage');

}

**render**() {

**return** <**div**>Homepage</**div**>;

}

}

Any actions defined within a componentDidMount() lifecycle hook are called only once when the component is first mounted.

Using a Function Component

The useEffect() hook can be used with function components:

**const** **Homepage** = () => {

**useEffect**(() => {

**trackPageView**('Homepage');

}, []);

**return** <**div**>Homepage</**div**>;

};

The useEffect() hook is more flexible than the lifecycle methods used for class components. It receives two parameters:

1. The first parameter it takes is a callback function to be executed.
2. The optional second parameter it takes is an array containing any variables that are to be tracked.

The value passed as the second argument controls when the callback is executed:

* If the second parameter **is undefined**, the callback is executed every time that the component is rendered.
* If the second parameter **contains an array of variables**, then the callback will be executed as part of the first render cycle and will be executed again each time an item in the array is modified.
* If the second parameter **contains an empty array**, the callback will be executed only once as part of the first render cycle. The example above shows how passing an empty array can result in similar behaviour to the componentDidMount() hook within a function component.

**10.**

**What are the most common approaches for styling a React application?**

Hide answer

CSS Classes

React allows class names to be specified for a component, like class names are specified for a DOM element in HTML.

When developers first start using React after developing traditional web applications, they often use CSS classes for styling because they are already familiar with the approach.

Inline CSS

Styling React elements using inline CSS allows styles to be completely scoped to an element using a well-understood, standard approach. However, there are certain styling features that are not available with inline styles. For example, the styling of :hover pseudo-classes.

Pre-processors Such as Sass, Stylus, and Less

Pre-processors are often used on React projects. This is because, like CSS, they are well understood by developers and are often already in use if React is being integrated into a legacy application.

CSS-in-JS Modules Such as Styled Components, Emotion, and Styled-jsx

CSS-in-JS modules are a popular option for styling React applications because they integrate closely with React components. For example, they allow styles to change based on React props at runtime. Also, by default, most of these systems scope all styles to the respective component being styled.

**11.**

**If you were working on a React application that was rendering a page very slowly, how would you go about investigating and fixing the issue?**

Hide answer

If a performance issue such as slow rendering is seen within a React app, the first step is to use the Profiler tool provided within the React Developer Tools browser plugin, which is available for Google Chrome and Mozilla Firefox. The Profiler tool allows developers to find components that take a long time to render or are rendering more frequently than necessary.

One of the most common issues in React applications is when components re-render unnecessarily. There are two tools provided by React that are helpful in these situations:

* React.memo(): This prevents unnecessary re-rendering of function components
* PureComponent: This prevents unnecessary re-rendering of class components

Both of these tools rely on a shallow comparison of the props passed into the component—if the props have not changed, then the component will not re-render. While both tools are very useful, the shallow comparison brings with it an additional performance penalty, so both can have a negative performance impact if used incorrectly. By using the React Profiler, performance can be measured before and after using these tools to ensure that performance is actually improved by making a given change.

**12.**

**At a high level, what is the virtual DOM (VDOM) and how does React use it to render to the DOM?**

Hide answer

The VDOM is a programming concept, providing a critical part of the React architecture. Rather than interacting directly with the DOM, changes are instead first rendered to the VDOM—a lightweight representation of the target state of the DOM.

Changes made to the VDOM are batched together to avoid unnecessary frequent changes to the DOM. Each time these batched changes are persisted to the DOM, React creates a diff between the current representation and the previous representation persisted to the DOM, then applies the diff to the DOM.

This abstraction layer for the DOM provides a simple interface for developers while allowing React to update the DOM in an efficient and performant manner.

**13.**

**What is prop drilling and how can you avoid it?**

Hide answer

When building a React application, there is often the need for a deeply nested component to use data provided by another component that is much higher in the hierarchy.

Consider the following example components:

* <EditUsersPage />, which includes selectedUserAddress in its component state and renders a <User /> component
* <User />, which renders a <UserDetails /> component
* <UserDetails />, which renders a <UserAddress /> component
* A <UserAddress /> component that requires the selectedUserAddress property stored in the <EditUsersPage /> state

The simplest approach is to simply pass a selectedUserAddress prop from each component to the next in the hierarchy from the source component to the deeply nested component. This is called prop drilling.

The primary disadvantage of prop drilling is that components that should not otherwise be aware of the data—in this case <User /> and <UserDetails />—become unnecessarily complicated and are harder to maintain.

To avoid prop drilling, a common approach is to use React context. This allows a Provider component that supplies data to be defined, and allows nested components to consume context data via either a Consumer component or a useContext hook.

While context can be used directly for sharing global state, it is also possible to use context indirectly via a state management module, such as Redux.

**14.**

**What is the**StrictMode**component and why would you use it?**

Hide answer

<StrictMode /> is a component included with React to provide additional visibility of potential issues in components. If the application is running in development mode, any issues are logged to the development console, but these warnings are not shown if the application is running in production mode.

Developers use <StrictMode /> to find problems such as deprecated lifecycle methods and legacy patterns, to ensure that all React components follow current best practices.

<StrictMode /> can be applied at any level of an application component hierarchy, which allows it to be adopted incrementally within a codebase.

**15.**

**What is the key architectural difference between a JavaScript library such as React and a JavaScript framework such as Angular? How would that impact the decision for a project to use one versus the other?**

Hide answer

React enables developers to render a user interface. To create a full front-end application, developers need other pieces, such as state management tools like Redux.

Like React, Angular enables developers to render a user interface, but it is a “batteries included” framework that includes prescriptive, opinionated solutions to common requirements like state management.

While there are many other considerations when comparing React and Angular specifically, this key architectural difference means that:

* Using a library such as React can give a project a greater ability to evolve parts of the system—again for example, state management—over time, when new solutions are created by the open source community.
* Using a framework such as Angular can make it easier for developers to get started and can also simplify maintenance.

**16.**

**How can automated tooling be used to improve the accessibility of a React application?**

Hide answer

There are two main categories of automated tools that can be used to identify accessibility issues:

Static Analysis Tools

Linting tools like ESLint can be used with plugins such as eslint-plugin-jsx-a11y to analyse React projects at a component level. Static analysis tools run very quickly, so they bring a good benefit at a low cost.

Browser Tools

Browser accessibility tools such as aXe and Google Lighthouse perform automated accessibility at the app level. This can discover more real-world issues, because a browser is used to simulate the way that a user interacts with a website. It is possible for many of these tools to run in a continuous integration environment such as Travis or Jenkins. Since these tools take longer to execute, many developers just run these tools within their local browser on an occasional basis, such as when reaching project milestones.

**17.**

**[Legacy projects only: React < 16.8] What are pure functional Components?**

Hide answer

Traditional React Components as we have seen thus far are creating a class with class Example extends React.Component or React.createClass(). These create stateful components if we ever set the state (i.e. this.setState(), getInitialState(), or this.state = {} inside a constructor()).

If we have no intention for a Component to need state, or to need lifecycle methods, we can actually write Components with a pure function, hence the term “pure functional Component”:

function Date(props){

let {msg="The date is:"} = props

let now = new Date()

return <div>

<span>{msg}</span>

<time>{now.toLocaleDateString()}</time>

</div>

}

This function that returns a React Element can be used whereever we see fit:

DOM.render(<div><Date msg="Today is"/><div>)

You might notice that <Date/> also takes a prop – we can still pass information into the Component.

**18.**

**How might React handle or restrict Props to certain types, or require certain Props to exist?**

Hide answer

You may recall a previous example that looked like the following (some parts of the code left out):

class LazyLoad extends Component {

constructor(p){

super(p)

this.state = { loaded:0 }

}

render(){

let {children} = this.props,

{loaded} = this.state

return <div>

{loaded && children}

</div>

}

}

When rendering the <LazyLoad/>, we can pass in props (i.e. <LazyLoad top={0}/>). Props are essentially inputs or values being passed down to one Component from the parent rendering context, and the code that passes the props to the element may not be compliant with your code. For example, top here seems to be just a number, but would I be able to verify that the prop is in-fact a number before my component is rendered? It’s certainly possible to write this code in **each and every Component that uses props**. However, React provides us a much simpler and shorter solution: Prop Types.

let p = React.PropTypes

LazyLoad.PropTypes = {

top: p.number

}

When using React’s non-minified development version (i.e. when building and testing in development), React will throw an error to alert you of any instances where a Prop is either missing or the wrong type. Above, top should always be a number.

We can make top a *required* prop by adding:

let p = React.PropTypes

LazyLoad.PropTypes = {

top: p.number.isRequired

}

PropTypes can be used [to test Props for any kind of value](https://reactjs.org/docs/typechecking-with-proptypes.html). Here’s a few quick type-checkers React has for JavaScript’s built-in types:

* React.PropTypes.array,
* React.PropTypes.bool,
* React.PropTypes.func,
* React.PropTypes.number,
* React.PropTypes.object,
* React.PropTypes.string,
* React.PropTypes.symbol,

We can also test that props are React and DOM types:

* React.PropTypes.node,
* React.PropTypes.element,

And we have the ability to test more complex types, such as “shapes”, “instances of”, or “collections of”:

* React.PropTypes.instanceOf(Message),
* React.PropTypes.oneOf(['News', 'Photos']),
* React.PropTypes.oneOfType([ React.PropTypes.string, React.PropTypes.number, React.PropTypes.instanceOf(Message)])
* React.PropTypes.arrayOf(React.PropTypes.number),
* React.PropTypes.shape({ color: React.PropTypes.string, fontSize: React.PropTypes.number })

Use these PropTypes to produce errors and track down bugs. When used effectively, PropTypes will prevent your team from losing too much time in the debugging and documentation process, ensuring stricter standards and understanding of your growing library of Components.

**19.**

**[Legacy projects only: React < 15.5] Compare and contrast creating React Components in ES5 and ES2015 (also known as ES6). What are the advantages and disadvantages of using one or the other? Include notes about default props, initial state, PropTypes, and DisplayName.**

Hide answer

Creating React Components the ES5 way involves using the React.createClass() method:

var Comments = React.createClass({

displayName: 'Comments',

getInitialState: function(){

return {comments: []}

},

getDefaultProps: function(){

return {some\_object: {a:1, b:2, c:3}}

},

\_handleClick: function(){

alert('hello world!')

},

render: function(){

return <div>

There are {this.state.comments.length} comments

<button onClick={this.\_handleClick}>click me!</button>

</div>

}

})

This Comments Component can now be rendered either inside another React Component or directly in the call to ReactDOM.render():

ReactDOM.render(<Comments />, document.querySelector('.app'))

ES5 Components have some particular qualities, which we’ll note:

1. Like the above example, to set the state to an initial value, create the getInitialState() function on the Component. What it returns will be the initial state for a Component when rendered.
2. Additionally, you can set the default props for the component to have a certain value with the getDefaultProps() method on the ES5 version.
3. The displayName is used in debugging and error reporting by React. If you use JSX, then the displayName is automatically filled out.
4. For some, it is common practice to denote a custom method added to a React Component by prefixing it with an underscore, hence \_handleClick. \_handleClick is passed as the onClick callback for a button in the code above. We can’t do this so easily in the ES6 API of React, because the ES5 version has *autobinding*, but the ES6 does not. Let’s take a look at what autobinding provides:

**Auto-binding**

Consider the following piece of code:

var thing = {

name: 'jen',

speak: function(){ console.log(this.name) }

}

window.addEventListener('keyup', thing.speak)

Invoking thing.speak() in the console will log "jen", but pressing a key will log undefined because the *context* of the callback is the global object. The browser’s global object – window – becomes this inside the speak() function, so this.name becomes window.name, which is undefined.

React in ES5 automatically does autobinding, effectively doing the following:

window.addEventListener('keyup', thing.speak.bind(thing))

*Autobinding* automatically binds our functions to the React Component instance so that passing the function by reference in the render() works seamlessly.

Creating React Components the ES6 way works a little differently, favoring the ES6 class ... extends ... syntax, and no autobinding feature:

class Comments extends React.Component {

constructor(props){

super(props)

this.state = {comments: []}

}

\_handleClick(){

alert('hello world!')

}

render(){

return <div>

There are {this.state.comments.length} comments

<button onClick={() => this.\_handleClick}>click me!</button>

</div>

}

}

Comments.defaultProps = {a:1, b:2, c:3}

Comments.displayName = 'Comments'

1. Notice that in ES6, we have a constructor() that we use to set the initial state,
2. We can add default props and a display name as properties of the new class created, and
3. The render() method, which works as normal, but we’ve had to alter how we pass in the callback function. This current approach (<button onClick={() => this.\_handleClick}>click me!</button>) will create a new function each time the component is re-rendered; so if it becomes a performance bottleneck, you can always bind manually and store the callback:

class Comments extends React.Component {

constructor(...args) {

super(...args);

this.state = { toggledOn: false };

this.\_handleClick = this.\_handleClick.bind(this);

}

\_handleClick() {

this.setState(prevState => ({ toggledOn: !prevState.toggledOn });

}

render() {

return <button onClick={this.\_handleClick}> { this.state.toggledOn ? 'ON' : 'OFF' } </button>

}

}

Or with class fields syntax:

class Comments extends React.Component {

state = { toggledOn: false };

\_handleClick = () => {

this.setState(prevState => ({ toggledOn: !prevState.toggledOn }));

};

render() {

return <button onClick={this.\_handleClick}> {this.state.toggleOn ? 'ON' : 'OFF' </button>

}

}

Many React utility libraries on npm provide a single function to bind all handlers in the constructor, just like React does.

**20.**

**[Legacy projects only: React < 15.5] Compare and contrast incorporating mixins and enforcing modularity in React Components. (**extend**,**createClass**and mixins, HOCs) Why would you use these techniques, and what are the drawbacks of each?**

Hide answer

Modularity is – in effect – something partially done with intention while coding, and partially done when refactoring afterwards.

Let’s first paint a scenario which we’ll model using each method above. Imagine we have three React Components: onScrollable, Loadable, and Loggable.

* an onScrollable Component will listen to the window.onscroll event, and use a logging mechanism to record it
* a Loadable Component will not render until one or more async requests have finished loading, and will use a logging mechanism to record when this occurs
* a Loggable Component provides a logging mechanism, be it a console, a [Winston Node.js logging setup](https://github.com/winstonjs/winston) on our own server, or some 3rd party logging service that records logs via JSON requests

First, let’s model this with React’s ES5 API and [mixins](https://facebook.github.io/react/docs/reusable-components.html" \l "mixins).

Interactive code sample at [Matthew Keas’ github](https://goo.gl/kSIJe0).

**var** onKeypress = {

**componentDidMount**(){

this.onpress && window.**addEventListener**('keyup', this.onpress)

},

**componentWillUnmount**(){

this.onpress && window.**removeEventListener**('keyup', this.onpress)

}

}

**var** **Loadable** = {

**componentDidMount**(){

**if**(this.load){

this.**setState**({loaded: false})

**Promise**.**all**([].**concat**(this.load))

.**then**(() =>

this.**setState**({loaded: true}))

}

}

}

**var** **Loggable** = {

**log**(...args) {

**alert**(args)

}

}

**var** **Example** = **React**.**createClass**({

mixins: [**Loggable**, **Loadable**, onKeypress],

**componentWillMount**(){

this.onpress = (e) => this.**log**(e.which, 'pressed!')

this.load = [**new** **Promise**((res,rej) => setTimeout(res, 3000))]

this.log = (...args) => console.**log**(...args)

},

**getInitialState**(){

**return** {}

},

**render**() {

**if**(!this.state.loaded)

**return** <**div**>loading...</**div**>

**return** <**div**>test</**div**>

}

})

DOM.**render**(<**Example** />, document.body)

Let’s note a few things about the above code:

1. There are three POJOs (Plain ol’ JS Objects) created, which hold lifecycle and/or custom methods.
2. When creating the Example Component, we add mixins: [Loggable, Loadable, onKeypress], meaning that any functions from all three objects are included in the Example class.
3. Both onKeypress and Loadable add a componentDidMount(), but this doesn’t mean the latter cancels out the prior. In fact, all componentDidMount() functions from each mixin will be invoked when the event occurs. The same is true for all lifecycle methods added to mixins. This way, both the onKeypress and Loadable mixins will work simultaneously!

Mixins are possible, but not built-in to React’s ES6 API. However, the ES6 API makes it easier to create a custom Component that extends another custom Component.

So our Components’ prototype chains would look like the following:

[Example] --- extends ---> [Loggable] --- extends ---> [Loadable] --- extends ---> [onKeypress]

This would result from Components written as such:

**class** **onKeypress** {}

**class** **Loadable** **extends** **onKeypress** {}

**class** **Loggable** **extends** **Loadable** {}

**class** **Example** **extends** **Loggable** {}

Creating anonymous classes would help here, because then Loggable would not have to extend Loadable *and* onKeypress.

**class** **Example** **extends** (**class** **a** **extends** **Loggable** **extends** ...) {

}

With a mixin() function, this could look more like:

**class** **Example** **extends** **mixin**(**Loggable**, **Loadable**, onKeypress) {

}

Let’s try to write mixin() by building a chain of anonymous classes that extend Loggable, Loadable, and onKeypress:

**const** **mixin** = (...classes) =>

classes.**reduce**((a,v) => {

**return** (**class** **temp** **extends** **a**)

}, (**class** **temp** {}))

There’s a caveat, though – if Loadable extends onKeypress and both implement componentDidMount(), Loadable’s version will be lower on the prototype chain, which means the function from onKeypress will never be invoked.

The takeaway here is that the mixin pattern isn’t easily implemented by relying only on the ES6 extends approach. Let’s try to implement mixin() again, but build a more robust function:

**const** **mixin** = (...classes) => {

**class** **\_mixin** {}

**let** proto = \_mixin.**prototype**

classes.**map**(({prototype:p}) => {

**Object**.**getOwnPropertyNames**(p).**map**(key => {

**let** oldFn = proto[key] || (() => {})

proto[key] = (...args) => {

**oldFn**(...args)

**return** p[key](...args)

}

})

})

**return** \_mixin

}

This new mixin() implementation maps over each class, and cascades function calls from a parent class’s componentDidMount() alongside the child’s componentDidMount().

There are similar implementations of mixin() available on npm, using packages like [react-mixin](https://www.npmjs.com/package/react-mixin) and [es6-react-mixins](https://www.npmjs.com/package/es6-react-mixins).

We use mixin() from above like so:

interactive code sample available at <https://goo.gl/VnQ21R>

**class** **A** {

**componentDidMount**(){

console.**log**(1)

}

}

**class** **B** {

**componentDidMount**(){

console.**log**(2)

}

}

**class** **C** **extends** **mixin**(A,B) {

**componentDidMount**(...p){

super.**componentDidMount**(...p)

console.**log**(3)

}

}

**let** c = **new** **C**()

c.**componentDidMount**() // logs 1, 2, 3

Recently, React provided support for – and documented its preference of – React Components declared with ES6 classes. ES6 classes allow us to create component heirarchies with less code, however this makes it more difficult to create a single Component that inherits properties from several mixins, instead forcing us to create prototype chains.

**21.**

**[Legacy projects only: React < 16] Compare and contrast the various React Component lifecycle methods. How might understanding these help build certain interfaces/features?**

Hide answer

There are several React lifecycle methods that help us manage the asynchronous and non-determinate nature of a Component during it’s lifetime in an app – we need provided methods to help us handle when a component is created, rendered, updates, or removed from the DOM.

Let’s first classify and define the life-cycle methods:

**The “Will’s”** - invoked right before the event represented occurs.

* componentWillMount() - Invoked once, both on the client and server, immediately before the initial rendering occurs. If you call setState within this method, render() will see the updated state and will be executed only once despite the state change.
* componentWillReceiveProps(object nextProps) - Invoked when a component is receiving new props. This method is not called for the initial render. Calling this.setState() within this function will not trigger an additional render. One common mistake is for code executed during this lifecycle method to assume that props have changed.
* componentWillUnmount() - Invoked immediately before a component is unmounted from the DOM. Perform any necessary cleanup in this method, such as invalidating timers or cleaning up any DOM elements that were created in componentDidMount.
* componentWillUpdate(object nextProps, object nextState) - Invoked immediately before rendering when new props or state are being received. This method is not called for the initial render.

**The “Did’s”**

* componentDidMount() - Invoked once, only on the client (not on the server), immediately after the initial rendering occurs. At this point in the lifecycle, you can access any refs to your children (e.g., to access the underlying DOM representation). The componentDidMount() method of child components is invoked before that of the parent component.
* componentDidUpdate(object prevProps, object prevState) - Invoked immediately after the component’s updates are flushed to the DOM. This method is not called for the initial render. Use this as an opportunity to operate on the DOM when the component has been updated.

**The “Should’s”**

* shouldComponentUpdate(object nextState, object nextProps) - Invoked before rendering when new props or state are being received. This method is not called for the initial render or when forceUpdate() is used. Use this as an opportunity to return false when you’re certain that the transition to the new props and state will not require a component update.

Having a strong understanding of how these fit together – and how setState() or forceUpdate() affect the lifecycle – will help the conscious React developer build robust UIs.