\*\*\*\*\* JavaScript \*\*\*\*\*

1. Design patterns in JavaScript   
<https://addyosmani.com/resources/essentialjsdesignpatterns/book/#constructorpatternjavascript>

=> Design patterns are reusable solutions to commonly occurring problems in software design.

* **Creational** design patterns: - These patterns deal with object creation mechanisms which optimize object creation compared to a basic approach.
* **Structural** design patterns
* **Behavioral** design patterns
* **Concurrency** design patterns
* **Architectural** design patterns

*// either of the following ways can be used to create a new object*

**Var** instance = {};

*// or*

**Var** instance = Object.create (Object.prototype);

*// or*

**Var** instance = **new** Object ();

Constructor Pattern

*// we define a constructor for Person objects*

**function** **Person**(name, age, isDeveloper) {

**this**.name = name;

**this**.age = age;

**this**.isDeveloper = isDeveloper || false;

**this**.writesCode = **function**() {

console.log(**this**.isDeveloper? "This person does write code" : "This person does not write code");

}

}

*// creates a Person instance with properties*

**var** person1 = **new** Person("Bob", 38, true);

*// creates a Person instance with properties***var** person2 = **new** Person ("Alice", 32);

Module Pattern

*// through the use of a closure we expose an object*

*// as a public API which manages the private objects array*

**var** collection = (**function**() {

*// private members*

**var** objects = [];

*// public members*

**return** {

addObject: **function**(object) {

objects.push(object);

},

removeObject: **function**(object) {

**var** index = objects.indexOf(object);

**if** (index >= 0) {

objects.splice(index, 1);

}

},

getObjects: **function**() {

**return** JSON.parse(JSON.stringify(objects));

}

};

})();

collection.addObject("Bob");

collection.addObject("Alice");

collection.addObject("Franck");

*// prints ["Bob", "Alice", "Franck"]*

console.log(collection.getObjects());

collection.removeObject("Alice");  
  
Singleton Pattern

2. Slice and Splice methods in JavaScript

The **splice()** method changes the original array and **slice()** method doesn’t change the original array.

The **splice()**method returns the removed item(s) in an array and **slice()**method returns the selected element(s) in an array, as a new array object.

var array=[1,2,3,4,5];

array.splice(2, 1);

array // (4) [1, 2, 4, 5]

array.slice(1, 3);

[2, 4]

array //(4) [1, 2, 4, 5]

3. Merge two arrays

var hege = ["Cecilie", "Lone"];  
var stale = ["Emil", "Tobias", "Linus"];  
var children = hege.concat(stale);

merger without duplicate

let array1 = ['a','b','c']

let array2 = ['c','c','d','e'];

let array3 = array1.**concat**(array2);

array3 = array3.**filter**((item,index)=>{

return (array3.**indexOf**(item) == index)

});

4. Typescript basics  
<https://www.typescriptlang.org/docs/handbook/>

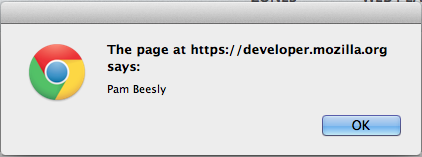
5. Closures and its uses

a closure is simply a function defined within another function. However, the power of closures is derived from the fact that the inner function remembers the environment in which it was created. In other words, the inner function has access to the outer function’s variables and parameters.

## **What’s it look like?**

**Below is an example of a closure**

function pam() {  
 var name = "Pam Beesly";  
 function displayName() {  
 alert (name);  
 }  
 displayName();   
}  
pam();



Our outer function — pam — does three things:

1. Define a local variable, name
2. Define a function, displayName
3. Call displayName

6. Program to reverse string, program to find if the string is anagram

const isAnagram = (str1, str2) => {

const normalize = str =>

str

.toLowerCase()

.replace(/[^a-z0-9]/gi, '')

.split('')

.sort()

.join('');

return normalize (str1) === normalize(str2);

};

console.log (isAnagram ('iceman', 'cinema'));

console.log (isAnagram ('madam', 'madam'));

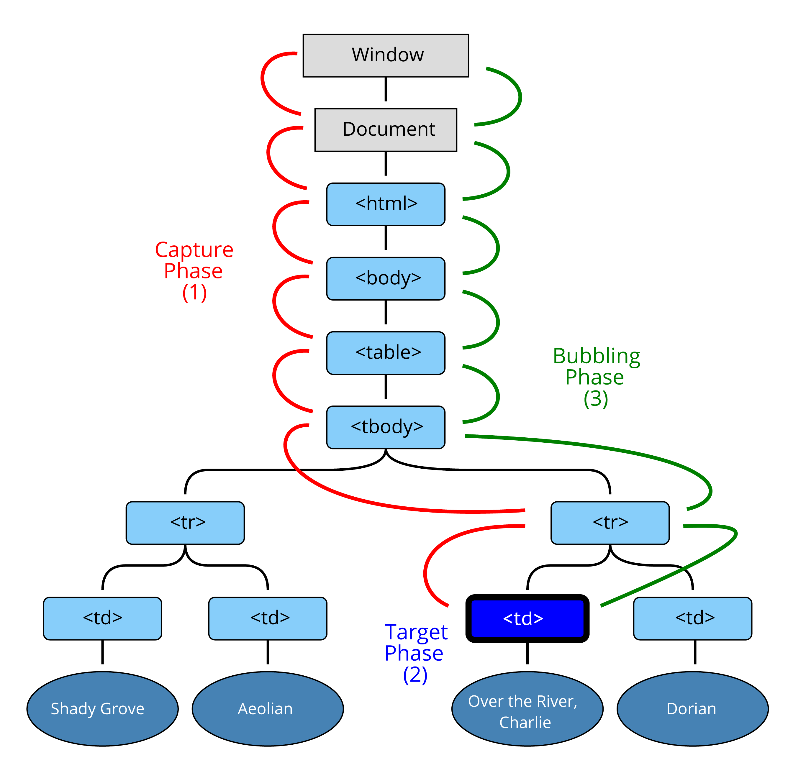
7. HTTP1/HTTP2

These are the high-level differences between HTTP1 and HTTP2:

* HTTP2 is binary, instead of textual
* HTTP2 is fully multiplexed, instead of ordered and blocking
* HTTP2 can, therefore, use one connection for parallelism
* HTP2 uses header compression to reduce overhead
* HTTP2 allows servers to “push” responses proactively into client caches

8. Event bubbling and event capturing

<https://javascript.info/bubbling-and-capturing>



## **Bubbling**

The bubbling principle is simple.

**When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.**

Let’s say we have 3 nested elements FORM > DIV > P with a handler on each of them:

A click on the inner <p> first runs onclick:

1. On that <p>.
2. Then on the outer <div>.
3. Then on the outer <form>.
4. And so on upwards till the document object.

## **Capturing**

There’s another phase of event processing called “capturing”. It is rarely used in real code, but sometimes can be useful.

The standard [DOM Events](http://www.w3.org/TR/DOM-Level-3-Events/) describes 3 phases of event propagation:

1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.
3. Bubbling phase – the event bubbles up from the element.

<script>

for(let elem of document.querySelectorAll('\*')) {

elem.addEventListener("click", e => alert(`Capturing: ${elem.tagName}`), true);

elem.addEventListener("click", e => alert(`Bubbling: ${elem.tagName}`));

}

</script>

1. Rxjs library and its usage in Angular

<https://angular.io/guide/rx-library>

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

**Create an observable from a promise:**

content\_copyimport { from } from 'rxjs';

// Create an Observable out of a promise

const data = from(fetch('/api/endpoint'));

// Subscribe to begin listening for async result

data.subscribe({

next(response) { console.log(response); },

error(err) { console.error('Error: ' + err); },

complete() { console.log('Completed');

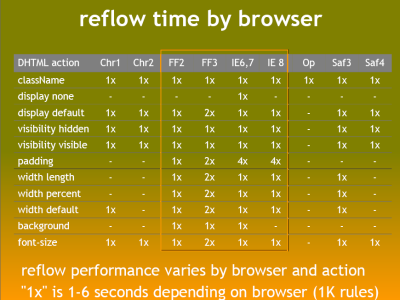
});

3. CSS floats and grid system

<https://css-tricks.com/snippets/css/complete-guide-grid/>

7. CSS reflow

Reflow is the name of the web browser process for re-calculating the positions and geometries of elements in the document, for the purpose of re-rendering part or all of the document. Because reflow is a user-blocking operation in the browser, it is useful for developers to understand how to improve reflow time and also to understand the effects of various document properties (DOM depth, CSS rule efficiency, different types of style changes) on reflow time. Sometimes reflowing a single element in the document may require reflowing its parent elements and also any elements which follow it.



**8. Reconciliation in React**

**Reconciliation** is the process through which **React** updates the DOM. When a component's state changes, **React** has to calculate if it is necessary to update the DOM. It does this by creating a virtual DOM and comparing it with the current DOM. In this context, the virtual DOM will contain the new state of the component.

<https://reactjs.org/docs/reconciliation.html>

**9. Shadow Dom in Angular**

Shadow DOM basically allows group of DOM implementation to be hidden inside a single element (which is the basic idea of components) and encapsulate styles to the element.  
OR  
Shadow DOM allows us to hide DOM logic behind other elements. 10. Virtual DOM in React

<https://angular.io/guide/component-styles>

<https://dev.to/ummeq/shadow-dom-in-angular-j08#:~:text=Shadow%20DOM%20basically%20allows%20group,encapsulate%20styles%20to%20the%20element.&text=Shadow%20DOM%20allows%20us%20to%20hide%20DOM%20logic%20behind%20other%20elements.>

**11. Positions in CSS**

**12. Specificity in CSS**

<https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity>

The following list of selector types increases by specificity:

1. [Type selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/Type_selectors) (e.g., h1) and pseudo-elements (e.g., ::before).
2. [Class selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/Class_selectors) (e.g., .example), attributes selectors (e.g., [type="radio"]) and pseudo-classes (e.g., :hover).
3. [ID selectors](https://developer.mozilla.org/en-US/docs/Web/CSS/ID_selectors) (e.g., #example).

**13. CSS combinators**

## Descendant combinator

The **descendant combinator** — typically represented by a single space ( ) character — combines two selectors such that elements matched by the second selector are selected if they have an ancestor (parent, parent's parent, parent's parent's parent, etc) element matching the first selector. Selectors that utilize a descendant combinator are called descendant selectors.

body article p

In the example below, we are matching only the <p> element which is inside an element with a class of .box.

**14. getDerivedstate form props**

<https://reactjs.org/docs/react-component.html#static-getderivedstatefromprops>

**static getDerivedStateFromProps()**

static getDerivedStateFromProps(props, state)

getDerivedStateFromProps is invoked right before calling the render method, both on the initial mount and on subsequent updates. It should return an object to update the state, or null to update nothing.

This method exists for [rare use cases](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#when-to-use-derived-state) where the state depends on changes in props over time. For example, it might be handy for implementing a <Transition> component that compares its previous and next children to decide which of them to animate in and out.

Deriving state leads to verbose code and makes your components difficult to think about.  
[Make sure you’re familiar with simpler alternatives:](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html)

* If you need to **perform a side effect** (for example, data fetching or an animation) in response to a change in props, use [componentDidUpdate](https://reactjs.org/docs/react-component.html#componentdidupdate) lifecycle instead.
* If you want to **re-compute some data only when a prop changes**, [use a memoization helper instead](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#what-about-memoization).
* If you want to **“reset” some state when a prop changes**, consider either making a component [fully controlled](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-controlled-component) or [fully uncontrolled with a key](https://reactjs.org/blog/2018/06/07/you-probably-dont-need-derived-state.html#recommendation-fully-uncontrolled-component-with-a-key) instead.

This method doesn’t have access to the component instance. If you’d like, you can reuse some code between getDerivedStateFromProps() and the other class methods by extracting pure functions of the component props and state outside the class definition.

Note that this method is fired on *every* render, regardless of the cause. This is in contrast to UNSAFE\_componentWillReceiveProps, which only fires when the parent causes a re-render and not as a result of a local setState.

**15. Access DOM elements in angular**

We can access the DOM in Angular using different reference types like ElementRef , TemplateRef , ViewRef , ComponentRef and ViewContainerRef . These reference types can be queried from templates using @ViewChild and @ContentChild . Browser's native DOM element can be accessed via ElementRef

import {ViewChild} from '@angular/core'

@ViewChild([#template-reference ], {read: [reference type]});

**16. Fragments in react**

A common pattern in **React** is for a component to return multiple elements. **Fragments** let you group a list of children without adding extra nodes to the DOM.

<https://reactjs.org/docs/fragments.html#:~:text=A%20common%20pattern%20in%20React,extra%20nodes%20to%20the%20DOM.>

**17. Pure components in React**

<https://blog.logrocket.com/pure-functional-components-in-react-16-6/#:~:text=A%20React%20component%20can%20be,are%20treated%20as%20pure%20components.>

## **Pure components**

Based on the concept of purity in functional programming paradigms, a function is said to be pure if:

* Its return value is only determined by its input values
* Its return value is always the same for the same input values

A React component can be considered pure if it renders the same output for the same state and props. For class components like this, React provides the PureComponent base class. Class components that extend the React.PureComponent class are treated as pure components.

**18. Diff between service and factory**

**19. Event loop**

<https://javascript.info/event-loop>

# **Event loop: microtasks and macrotasks**

Browser JavaScript execution flow, as well as in Node.js, is based on an event loop.

Understanding how event loop works is important for optimizations, and sometimes for the right architecture.

In this chapter we first cover theoretical details about how things work, and then see practical applications of that knowledge.  
  
[Event Loop](https://javascript.info/event-loop#event-loop)

The event loop concept is very simple. There’s an endless loop, where the JavaScript engine waits for tasks, executes them and then sleeps, waiting for more tasks.

The general algorithm of the engine:

While there are tasks:

1. Execute them, starting with the oldest task.
2. Sleep until a task appears, then go to 1.

That’s a formalization for what we see when browsing a page. The JavaScript engine does nothing most of the time, it only runs if a script/handler/event activates.

20. v8 engine

V8 is Google’s open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others. It implements [ECMAScript](https://tc39.es/ecma262/) and [WebAssembly](https://webassembly.github.io/spec/core/), and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. V8 can run standalone, or can be embedded into any C++ application.

21. Async and non-blocking

***ASYNC***

*Async* is widely considered to be non-blocking. As we shall see, this isn’t entirely true!

In the case of asynchronous JS, the browser does not interrupt the parsing of the HTML code, it downloads the script in parallel. Once the download is complete, however, it **executes** the script immediately and blocks the parsing of the remaining HTML code, the execution of the other scripts and the rendering of the page.

As such, *async* can cause the following problem: the scripts are not executed in the same order as they appear in the code, and, depending on the volume of JavaScript to execute, this can block the main thread and, therefore, increase your [Time to Interactive](https://www.fasterize.com/en/blog/understand-and-improve-the-time-to-interactive/).

22. Multithreading in nodejs

Multithreading can be achieve by using Spawn, Fork, Exec, execute and then we had execFile execute and there are synchronous way also of creating all those like spawnSync exceSync executeSync then execute sync but these are not suggested as it will block your main thread and the whole purpose of asynchronous functionality on nodejs.

23. psedo elements and pseudo classed in CSS

selector::pseudo-element {  
  property: value;  
}

::after, ::before, ::first-letter, ::first-line, ::selection

selector:pseudo-class {  
  property: value;  
}

:hover, a:link , : a:visited , a:active , a:hover

**24. shouldComponent update in React**

<https://developmentarc.gitbooks.io/react-indepth/content/life_cycle/update/using_should_component_update.html>

**Preventing unnecessary renders**

The shouldComponentUpdate() method is the first real life cycle optimization method that we can leverage in React. We can look at our current and new props & state and make a choice if we should move on. [React's PureRenderMixin](https://facebook.github.io/react/docs/pure-render-mixin.html) does exactly this. It checks the current props and state, compares it to the next props and state and then returns true if they are different, or false if they are the same.

25. box-sizing in CSS

26. Code splitting in React  
<https://reactjs.org/docs/code-splitting.html>

## **Bundling**

Most React apps will have their files “bundled” using tools like [Webpack](https://webpack.js.org/), [Rollup](https://rollupjs.org/) or [Browserify](http://browserify.org/). Bundling is the process of following imported files and merging them into a single file: a “bundle”. This bundle can then be included on a webpage to load an entire app at once.

If you’re using [Create React App](https://create-react-app.dev/), [Next.js](https://nextjs.org/), [Gatsby](https://www.gatsbyjs.org/), or a similar tool, you will have a Webpack setup out of the box to bundle your app.

If you aren’t, you’ll need to setup bundling yourself. For example, see the [Installation](https://webpack.js.org/guides/installation/) and [Getting Started](https://webpack.js.org/guides/getting-started/) guides on the Webpack docs.

27. Shallow rendering in react  
<https://reactjs.org/docs/shallow-renderer.html>

When writing unit tests for React, shallow rendering can be helpful. Shallow rendering lets you render a component “one level deep” and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered. This does not require a DOM.

### **shallowRenderer.render()**

You can think of the shallowRenderer as a “place” to render the component you’re testing, and from which you can extract the component’s output.

shallowRenderer.render() is similar to [ReactDOM.render()](https://reactjs.org/docs/react-dom.html#render) but it doesn’t require DOM and only renders a single level deep. This means you can test components isolated from how their children are implemented.

**28. Error Boundaries example in React**<https://reactjs.org/docs/error-boundaries.html#where-to-place-error-boundaries>

React 16 introduces a new concept of an “error boundary”.

Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

**Note**

Error boundaries do **not** catch errors for:

* Event handlers ([learn more](https://reactjs.org/docs/error-boundaries.html#how-about-event-handlers))
* Asynchronous code (e.g. setTimeout or requestAnimationFrame callbacks)
* Server side rendering
* Errors thrown in the error boundary itself (rather than its children)

**29. Error first in node.js**

<https://nodejs.org/api/errors.html#errors_error_first_callbacks>

### **Error-first callbacks**[**#**](https://nodejs.org/api/errors.html#errors_error_first_callbacks)

Most asynchronous methods exposed by the Node.js core API follow an idiomatic pattern referred to as an error-first callback. With this pattern, a callback function is passed to the method as an argument. When the operation either completes or an error is raised, the callback function is called with the Error object (if any) passed as the first argument. If no error was raised, the first argument will be passed as null.

**30. setState and replaceState in React**

setState the current and previous states are merged. With replaceState, it throws out the current state, and replaces it with only what you provide. Usually setState is used unless you really need to remove keys for some reason; but setting them to false/null is usually a more explicit tactic.

While it's possible it could change; replaceState currently uses the object passed as the state, i.e. replaceState(x), and once it's set this.state === x. This is a little lighter than setState, so it could be used as an optimization if thousands of components are setting their states frequently.

31. AV testing

32. Security aspects in Front end  
<https://medium.com/@manojsingh047/understanding-frontend-security-ff6585395534>

33. CSS positions

34. Flexbox in detail

35. Promises in detail and difference between promise and observable  
<https://medium.com/javascript-everyday/javascript-theory-promise-vs-observable-d3087bc1239a#:~:text=a%20Promise%20is%20always%20asynchronous,get%20a%20new%20tailored%20stream.>

* a *Promise*is always asynchronous, while an *Observable*can be either synchronous or asynchronous,
* a *Promise*can provide a single value, whereas an *Observable*is a stream of values (from 0 to multiple values),
* you can apply RxJS operators to an *Observable*to get a new tailored stream.

36. Remove duplicate values from array in JavaScript

var arr = ["apple", "bannana", "orange", "apple", "orange"];

arr = arr.filter( function( item, index, inputArray ) {

return inputArray.indexOf(item) == index;

});

Output: ["apple", "bannana", "orange"]

**37. map function parameters**

let newArray = arr.map (callback(currentValue[, index[, array]]) {

// return element for newArray, after executing something

}[, thisArg]);

***Callback***

Function that is called for every element of *arr*. Each time *callback* executes, the returned value is added to *new\_array*.

The *callback* function accepts the following arguments:

***currentValue***

The current element being processed in the array.

***Index*** Optional

The index of the current element being processed in the array.

***Array*** Optional

The array map was called upon.

**thisArg** Optional

Value to use as this when executing *callback*.

38. useMemo vs useCallback in React

<https://medium.com/@jan.hesters/usecallback-vs-usememo-c23ad1dc60#:~:text=useCallback%20and%20useMemo%20both%20expect,function%20and%20returns%20the%20result.>

## **API**

The APIs of useCallback and useMemo look similar. They both take in a function and an array of dependencies.

useCallback(fn, deps);  
useMemo(fn, deps);

So what is the difference? **useCallback returns its function uncalled** so you can call it later, while **useMemo calls its function and returns the result**.

|  |  |
| --- | --- |
|  | Function foo() {  return 'bar'; |
|  | } |
|  |  |
|  | const memoizedCallback = useCallback(foo, []); |
|  | const memoizedResult = useMemo(foo, []); |
|  |  |
|  | memoizedCallback; |
|  | // ƒ foo() { |
|  | // return 'bar'; |
|  | // } |
|  | memoizedResult; // 'bar' |
|  | memoizedCallback(); // 'bar' |
|  | memoizedResult(); // 🔴 Type Error |

39. Remove duplicate values from array

|  |
| --- |
| var newarray = [{  Email: "test1@gmail.com",  ID: "A"  }, {  Email: "test2@gmail.com",  ID: "B"  }, {  Email: "test3@gmail.com",  ID: "A"  }, {  Email: "test4@gmail.com",  ID: "C"  }, {  Email: "test5@gmail.com",  ID: "C"  }];    // Array to keep track of duplicates  var dups = [];  var arr = newarray.filter(function(el) {  // If it is not a duplicate, return true  if (dups.indexOf(el.ID) == -1) {  dups.push(el.ID);  return true;  }  return false;    });  console.log(arr);  //Other way around  var names = ["Mike","Matt","Nancy","Adam","Jenny","Nancy","Mike", "Adam", "Matt"];  uniqueArray = names.filter(function(item, pos) {  return names.indexOf(item) == pos;  }) |

40. Controlled and Uncontrolled components in React

<https://itnext.io/controlled-vs-uncontrolled-components-in-react-5cd13b2075f9>

|  |  |
| --- | --- |
| **Controlled Component** | **Un Controlled Component** |
| In a controlled component, the form data is handled by the state within the component. The state within the component serves as “the single source of truth” for the input elements that are rendered by the component.  Let’s look at a code example. | Uncontrolled components act more like traditional HTML form elements. The data for each input element is stored in the DOM, not in the component. Instead of writing an event handler for all of your state updates, you use a ***ref*** to retrieve values from the DOM.  If you’re wondering what a ***ref*** is, good question! [From the React docs](https://reactjs.org/docs/refs-and-the-dom.html):  **Note:-**  *Refs provide a way to access DOM nodes or React elements created in the render method.* |
| import React, { Component } from 'react';  class App extends Component {  state = {  message: ''  }  updateMessage = (newText) => {  console.log(newText);  this.setState(() => ({  message: newText  }));  }  render() {  return (  <div className="App">  <div className="container">  <input type="text"  placeholder="Your message here."  value={this.state.message}  onChange={(event) => this.updateMessage(event.target.value)}  />  <p>the message is: {this.state.message}</p>  </div>  </div>  );  }  }  export default App; | import React, { Component } from 'react';  class App2 extends Component {  constructor(props){  super(props);  this.handleChange = this.handleChange.bind(this);  this.input = React.createRef();  }    handleChange = (newText) => {  console.log(newText);  }  render() {  return (  <div className="App2">  <div className="container">  <input type="text"  Placeholder="Your message here."  ref={this.input}  onChange={(event) => this.handleChange(event.target.value)}  />  </div>  </div>    );  }  }  export default App2; |
|  |  |

**41. Pure components in React**

<https://medium.com/technofunnel/working-with-react-pure-components-166ded26ae48>

**Pure Components** in React are the components which do not re-renders when the value of state and props has been updated with the same values. If the value of the previous state or props and the new state or props is the same, the component is not re-rendered. Pure Components restricts the re-rendering ensuring the higher performance of the Component

**Features of React Pure Components**

* Prevents re-rendering of Component if props or state is the same
* Takes care of “shouldComponentUpdate” implicitly
* State and Props are Shallow Compared
* Pure Components are more performant in certain cases

**42. Content security policy (CSP)**

<https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP>

**Content Security Policy** ([CSP](https://developer.mozilla.org/en-US/docs/Glossary/CSP)) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross Site Scripting ([XSS](https://developer.mozilla.org/en-US/docs/Glossary/XSS)) and data injection attacks. These attacks are used for everything from data theft to site damage to distribution of malware.

CSP is designed to be fully backward compatible (except CSP version 2 where there are some explicitly-mentioned inconsistencies in backward compatibility; more details [here](https://www.w3.org/TR/CSP2) section 1.1). Browsers that don't support it still work with servers that implement it, and vice-versa: browsers that don't support CSP simply ignore it, functioning as usual, defaulting to the standard same-origin policy for web content. If the site doesn't offer the CSP header, browsers likewise use the standard [same-origin policy](https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy).

To enable CSP, you need to configure your web server to return the [Content-Security-Policy](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy) HTTP header. (Sometimes you may see mentions of the X-Content-Security-Policy header, but that's an older version and you don't need to specify it anymore.)

Alternatively, the [<meta>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/meta) element can be used to configure a policy, for example:  
  
<meta http-equiv="Content-Security-Policy" content="default-src 'self'; img-src https://\*; child-src 'none';">

43. HOC component example with login component

|  |
| --- |
| import React, { PropTypes } from 'react';  import { connect } from 'react-redux';  import { push } from 'react-router-redux';  export default function (ComposedComponent) {  class Authenticate extends React.Component {  static propTypes = {  isAuthenticated: PropTypes.boolean,  redirect: PropTypes.func.isRequired  };  componentDidMount() {  this.\_checkAndRedirect();  }  componentDidUpdate() {  this.\_checkAndRedirect();  }  \_checkAndRedirect() {  const { isAuthenticated, redirect } = this.props;  if (!isAuthenticated) {  redirect();  }  }  render() {  return (  <div>  {this.props.isAuthenticated ? <ComposedComponent {...this.props} /> : null }  </div>  );  }  }  const mapStateToProps = (state) => {  return {  isAuthenticated: state.auth.isAuthenticated  };  };    const mapDispatchToProps = dispatch => bindActionCreators({  redirect: () => push('/signin')  }, dispatch)    Authenticate.propTypes = propTypes  return connect(  mapStateToProps,  mapDispatchToProps  )(Authenticate); |

**44. Life cycle methods in react**<https://reactjs.org/docs/state-and-lifecycle.html>

The componentDidMount () method runs after the component output has been rendered to the DOM. This is a good place to set up a timer:

componentDidMount () {

this.timerID = setInterval (() => this.tick (), 1000);

}

Note how we save the timer ID right on this (this.timerID).

We will tear down the timer in the componentWillUnmount() lifecycle method:

componentWillUnmount () {

clearInterval(this.timerID);

}

Finally, we will implement a method called tick () that the Clock component will run every second

function FormattedDate (props) {

return <h2>It is {props.date.toLocaleTimeString ()}.</h2>;

}

class Clock extends React.Component {

constructor (props) {

super (props);

this.state = {date: new Date ()};

}

componentDidMount () {

this.timerID = setInterval(

() => this.tick(),

1000

);

}

componentWillUnmount() {

clearInterval(this.timerID);

}

tick() {

this.setState({

date: new Date()

});

}

render() {

return (

<div>

<h1>Hello, world!</h1>

<FormattedDate date={this.state.date} />

</div>

);

}

}

function App() {

return (

<div>

<Clock />

<Clock />

<Clock />

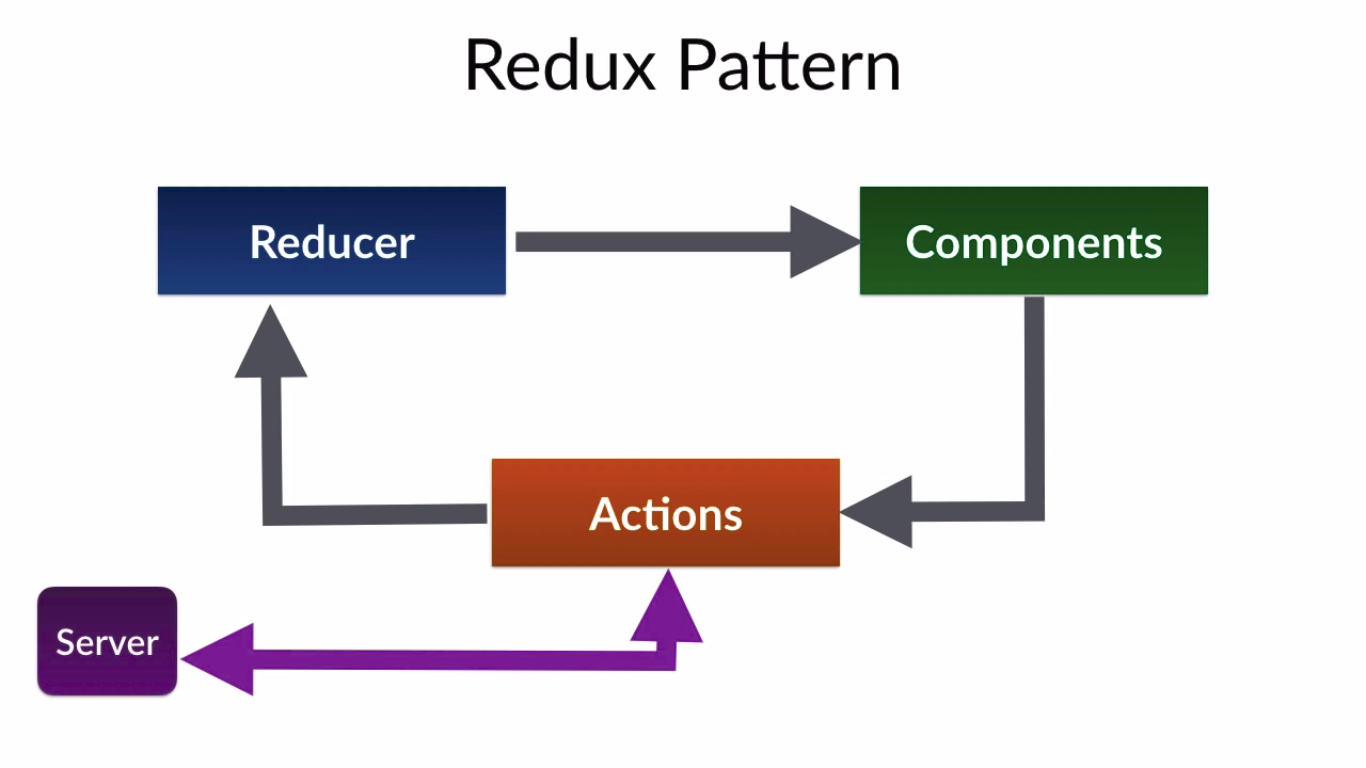
</div>

);

}

ReactDOM.render (<App />, document.getElementById('root'));

45. Redux architecture and use of connect function  
<https://redux.js.org/api/store#getState>



**Use of Connect ()**

The connect() function connects a React component to a Redux store.

It provides its connected component with the pieces of the data it needs from the store, and the functions it can use to dispatch actions to the store.

It does not modify the component class passed to it; instead, it returns a new, connected component class that wraps the component you passed in.

function connect(mapStateToProps?, mapDispatchToProps?, mergeProps?, options?)

The mapStateToProps and mapDispatchToProps deals with your redux store’s state and dispatch, respectively. state and dispatch will be supplied to your mapStateToProps or mapDispatchToProps functions as the first argument.

**46. useEffect with componentDidMount, componentWillUnmount, componentDidUpdate example**

<https://reactjs.org/docs/hooks-effect.html>

47. **Explain componentWillUnmount with example**

<https://stackblitz.com/edit/react-component-willunmout?file=src/App.js>

import React from "react";

import "./style.css";

export default class App extends React.Component {

  constructor() {

    super();

    this.state = {

      show: false

    };

  }

  render() {

    return (

      <div>

        <h1>Component will unmount</h1>

        {this.state.show ? <Child /> : null}

        <button onClick={() => this.setState({ show: !this.state.show })}>

          Toggle Child

        </button>

      </div>

    );

  }

}

class Child extends React.Component {

  componentWillUnmount() {

    console.warn("component is hidden");

  }

  render() {

    return (

      <div>

        <h3>Child Component</h3>

      </div>

    );

  }

}

Using Hooks

<https://stackblitz.com/edit/react-hooks-componentwillunmount?file=src%2FMouseContainer.js>

APP.js

import React, { useState } from "react";

import "./style.css";

import MouseContainer from "./MouseContainer";

export default function App() {

  const [display, setDisplay] = useState(true);

  const textFlic = () => {

    console.log("test");

  };

  return (

    <>

      {display && <MouseContainer />}

      <button onClick={() => setDisplay(!display)}>Toggle Display</button>

    </>

  );

}

MouseContainer.js

import React, { useState, useEffect } from "react";

function MouseContainer() {

  const [x, setX] = useState(0);

  const [y, setY] = useState(0);

  const logMouseEvent = e => {

    console.log("Mouse Position");

    setX(e.clientX);

    setY(e.clientY);

  };

  useEffect(() => {

    window.addEventListener("mousemove", logMouseEvent);

    return () => {

      console.log("Component unmounting code");

      window.removeEventListener("mousemove", logMouseEvent);

    };

  }, []);

  return (

    <>

      <h3>Capture Mouse Position</h3>

      <p>

        Hooks X {x} - Y {y}

      </p>

    </>

  );

}

export default MouseContainer;

48. Const Root = () => {

const update = (type) => {

console.log (type)

} return <Parent update={update} />

}

Const Parent = ({update}) => {useEffect(() => {

update (‘parent’)

}, [])return <Child update={update} />}

Const Child = () => {useEffect (() => {

update (‘child’)

}, []) return <div></div>}

49. useEffect with async await

<https://stackblitz.com/edit/react-useeffect-asycn-await?file=src%2FApp.js>

50. Lazy loading in React

<https://stackblitz.com/edit/react-7hthvp?file=src%2FApp.js>

51. Suspense in React

<https://stackblitz.com/edit/react-7hthvp?file=src%2FApp.js>

51. Fragments and Portal in React

<https://slinky.dev/docs/fragments-and-portals/>

## **Fragments** [Fragments](https://reactjs.org/docs/fragments.html) make it possible to return multiple elements from a component. To create a fragment simply return a list of elements in your render method.

**Portals**

Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.

ReactDOM.createPortal (child, container)

The first argument (child) is any [renderable React child](https://reactjs.org/docs/react-component.html#render), such as an element, string, or fragment. The second argument (container) is a DOM element.

## **Usage**

## Normally, when you return an element from a component’s render method, it’s mounted into the DOM as a child of the nearest parent node:

render () {

// React mounts a new div and renders the children into it

return (

<div> {this.props.children}

</div>);

}

However, sometimes it’s useful to insert a child into a different location in the DOM:

render() {

// React does \*not\* create a new div. It renders the children into `domNode`.

// `domNode` is any valid DOM node, regardless of its location in the DOM.

return ReactDOM.createPortal(

this.props.children,

domNode );

}

A typical use case for portals is when a parent component has an overflow: hidden or z-index style, but you need the child to visually “break out” of its container. For example, dialogs, hovercards, and tooltips.

52. Difference between npm and yarn

53. Create table dynamically

<https://stackblitz.com/edit/js-dynamically-create-table?file=index.js>

\*\*\*\*\* Database \*\*\*\*\*

53. Difference between MySQL and MongoDB

|  |  |
| --- | --- |
| **MYSQL** | **MongoDB** |
| MySQL is a relational database management system (RDBMS) from the Oracle Corporation. Like other relational systems, MySQL stores data in tables and uses structured query language (SQL) for database access.  When MySQL developers need to access data in an application, they merge data from multiple tables together in a process called a join.   In MySQL, you predefine your database schema and set up rules to govern the relationships between fields in your table | MongoDB is a [NoSQL database](https://www.mongodb.com/nosql-explained) that stores data as JSON-like documents. Documents store related information together and use the MongoDB query language (MQL) for access. |

**54. In MySQL storage engines**

MySQL supported storage engines:

* InnoDB
* MyISAM
* Memory
* CSV
* Merge
* Archive
* Federated
* Blackhole

55. Difference between MyISAM and InnoDB

|  |  |
| --- | --- |
| MyISAM | InnoDB |
| * MYISAM supports Table-level Locking * MyISAM designed for need of speed * MyISAM does not support foreign keys hence we call MySQL with MYISAM is DBMS * MyISAM stores its tables, data and indexes in disk space using separate three different files. (tablename.FRM, tablename.MYD, tablename.MYI) * MYISAM not supports transaction. You cannot commit and rollback with MYISAM. Once you issue a command it’s done. * MYISAM supports fulltext search * You can use MyISAM, if the table is more static with lots of select and less update and delete. | * InnoDB supports Row-level Locking * InnoDB designed for maximum performance when processing high volume of data * InnoDB support foreign keys hence we call MySQL with InnoDB is RDBMS * InnoDB stores its tables and indexes in a tablespace * InnoDB supports transaction. You can commit and rollback with InnoDB |

56. Full text search in MySQL

SELECT \* FROM table\_name WHERE MATCH(col1, col2)

AGAINST('search terms' IN NATURAL LANGUAGE MODE)

**57. What are indexes and purpose of it**

**Indexes** are used to quickly locate data without having to search every row in a database table every time a database table is accessed.

Indexes are used to find rows with specific column values quickly. Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows.

58. Type of indexes in MySQL

59. Stored procedures in MySQL

60. Functions in MySQL

61. group\_concat in MySQL

62. Triggers in MySQL

63. How to take backup of MySQL database

64. Database replication in MySQL

**65. What kind of indexes are available in MongoDB**

* Single Field indexes
* Compound Indexes
* Multi key indexes
* Text Indexes
* Wildcard indexes
* 2dSphere indexes
* 2d Indexes
* geoHaystack Indexes
* Hashed Indexes

**66. Join like queries in MongoDB**

<https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/#syntax>

$lookup  
*new in version 3.2.***Definition: -** Performs a left outer join to an unshared collection in the same database to filter in documents from the “joined” collection for processing. To each input document, the [$lookup](https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/#pipe._S_lookup) stage adds a new array field whose elements are the matching documents from the “joined” collection. The [$lookup](https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/#pipe._S_lookup) stage passes these reshaped documents to the next stage.

{

$lookup:

{

from: <collection to join>,

localField: <field from the input documents>,

foreignField: <field from the documents of the "from" collection>,

as: <output array field>

}

}

**67. Different joins in MySQL with example**

<https://www.javatpoint.com/mysql-join>

* MySQL INNER JOIN (or sometimes called simple join)
* MySQL LEFT OUTER JOIN (or sometimes called LEFT JOIN)
* MySQL RIGHT OUTER JOIN (or sometimes called RIGHT JOIN)

68. Can we create an index on array field in MongoDB?

**Multikey Indexes**

To index a field that holds an array value, MongoDB creates an index key for each element in the array. These multikey indexes support efficient queries against array fields. Multikey indexes can be constructed over arrays that hold both scalar values [[1]](https://docs.mongodb.com/manual/core/index-multikey/#scalar) (e.g. strings, numbers) and nested documents.

## **Create Multikey Index**

To create a multikey index, use the [db.collection.createIndex ()](https://docs.mongodb.com/manual/reference/method/db.collection.createIndex/#db.collection.createIndex) method:

db.coll.createIndex ({<field> : < 1 or -1 >})

MongoDB automatically creates a multikey index if any indexed field is an array; you do not need to explicitly specify the multikey type.

**69. MongoDB query example**

db.collection.find (query, projection)

## **Select All Documents in a Collection**

To select all documents in the collection, pass an empty document as the query filter parameter to the find method. The query filter parameter determines the select criteria:

**const** cursor = db.collection ('inventory').find({});

**70. Aggregation in MongoDB**

<https://www.tutorialspoint.com/mongodb/mongodb_aggregation.htm>

## **The aggregate () Method**

For the aggregation in MongoDB, you should use aggregate () method.

### **Syntax**

Basic syntax of aggregate () method is as follows −

db.COLLECTION\_NAME.aggregate (AGGREGATE\_OPERATION)

**71. Logs in MongoDB**

<https://docs.mongodb.com/manual/reference/command/getLog/>

### **Retrieve Available Log Filters** The following operation, run from the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell, returns the available log filters for passing to [getLog](https://docs.mongodb.com/manual/reference/command/getLog/#dbcmd.getLog):

db.adminCommand ( { getLog: "\*" } )

The operation returns the following document:

{"names”: [ "global", "startupWarnings" ], "ok" : 1 }

### **Retrieve Recent Events from Log**

The following operation, run from the [mongo](https://docs.mongodb.com/manual/reference/program/mongo/#bin.mongo) shell, retrieves the most recent global events for the [mongod](https://docs.mongodb.com/manual/reference/program/mongod/#bin.mongod):

db.adminCommand ( { getLog : "global" } )

**72. Get query which is time consuming in logs**

mongod --profile=1 --slowms=1 &

**73. Records in paging format in MySQL and parameters in LIMIT**

<https://www.codementor.io/@arpitbhayani/fast-and-efficient-pagination-in-mongodb-9095flbqr>

// Page 1

db.students.find ().limit (5)

// Page 2

db.students.find ().skip (5).limit (5)

// Page 3

db.students.find ().skip (5).limit (5)

74. Partition mechanism in MySQL

75. Maria DB --- diff between MySQL and maridb

76. Normalization in database and its benefits

\*\*\*\*\* React \*\*\*\*\*

**1. Virtual DOM explanation**

<https://reactjs.org/docs/faq-internals.html>

2. JSX in React

<https://reactjs.org/docs/introducing-jsx.html>

It is a syntax extension to JavaScript. React to describe what the UI should look like.   
JSX may remind you of a template language, but it comes with the full power of JavaScript. JSX produces React “elements”. We will explore rendering them to the DOM in the [next section](https://reactjs.org/docs/rendering-elements.html). Below, you can find the basics of JSX necessary to get you started.

**3. Difference between elements and components**

**React Element** - It **is** a simple object that describes a DOM node and its attributes or properties you can say. It **is** an immutable description object and you cannot apply any methods on it.

**React Component** - It **is** a function or class that accepts an input and returns a **React element**.

4. When to use class components and when to use functional component

5. Difference between state and props

6. How to bind methods/events in component

7. Inline conditional expression

8. Refs in React and forward refs

9. What is shadow DOM and how it is different from virtual DOM

**10. What is React Fiber?**

<https://github.com/acdlite/react-fiber-architecture>

Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM.

React Fiber is an ongoing reimplementation of React's core algorithm.

The goal of React Fiber is to increase its suitability for areas like animation, layout, and gestures.

Its headline feature is **incremental rendering**: the ability to split rendering work into chunks and spread it out over multiple frames.

11. What are controller and uncontrolled components?

12. HOC components

13. Why we use className in React

14. Decorators in React

15. How to enable production mode in react

16. setState is called 5 times on button click, how many times the component will be retendered.

17. Redux architecture

18. Middleware in redux - redux-thunk or redux-saga

\*\*\*\*\* Cloud/Deployment \*\*\*\*\*

1. What are the services used in AWS

2. How to deploy full stack on AWS infra

3. Docker and how to deploy FE app through docker

4. How container is designed for FE and BE app

5. Benefits of using pm2 and how to deploy a node based application

\*\*\*\*\* Programs \*\*\*\*\*

1. Merge two string arrays remove duplicate and sort

\*\*\*\*\* API/Web Services \*\*\*\*\*

1. Difference between PUT and POST

2. PATCH example

\*\*\*\*\* \*\*\*\*\*

1. Deploy React/NodeJs application on production

2. Performance and security measures for front end application

3. Eslint in react application

4. AWS services used

5. Experience of designing hybrid GUI for the applications (Desktop based and Mobile based applications). This will be a plus

Good to have knowledge of data marshalling/de-marshalling using JSON and XML

1. OAuth specifications

2. Difference between sharing and replication in MongoDB

3. CORS issue in angular application

4. What are microservices?

5. Difference between monolithic and microservices

1. How to chain promises

2. How to remove duplicate values from an array

3. Reverse a string

4. How to write IE specific css

5. Advantages of SCSS

6. HTML5 new features

7. Mobile first approach

8. JavaScript defer async

9. How to chain promises

10. Pass a common header for each and every http request in react

11. Promise.all and Promise.race

12. Why we need to use bind method in react

13. How to use es6 functions not working in IE

14. Test performance of react application

15. Await in nodejs

16. Diff between asynch await and Promise

17. Make this.setState asycn

18. Sequence of component lifecycle

19. Re-render a component in react

20. Why to use redux thunk and how it works

21. Lifecycle of redux

22. Pass data from child to parent in react