**********Javascript Common**********

First Order function:-> First Order function is a function that does not accept another function as argument and does not return value as a function.

Module In Javascript: JavaScript modules allow you to break up your code into separate files.

Promise: A Promise is a special JavaScript object. It produces a value after an asynchronous operation completes successfully, or an error if it does not complete successfully due to time out, network error, and so on.

Server Sent Event: A server-sent event is when a web page automatically gets updates from a server. This was also possible before, but the web page would have to ask if any updates were available. With server-sent events, the updates come automatically.

Event Bubbling : Propagate the event from innermost element to outermost element.
Event Capturing: Progate the event from outermost element to inner most element .
Event Propagation: Use event efficiently

JSON: Javascript Object Notation is used to transmitting data between server and web application.

ECMAScript: ECMAScript (often abbreviated as ES) is a standardized scripting language primarily used for writing client-side and server-side applications. It is the specification that defines the syntax, semantics, and behavior of the scripting language commonly known as JavaScript.

app shell model:The App Shell model, also known as the Application Shell Architecture, is a design pattern used in web development to optimize the performance and user experience of web applications, particularly those that are progressive web applications (PWAs)

**********Shadow DOM**********
Shadow DOM allows hidden DOM trees to be attached to elements in the regular DOM tree — this shadow DOM tree starts with a shadow root, underneath which you can attach any element, in the same way as the normal DOM.

-From the light DOM, shadow DOM elements are not visible to querySelector. Particularly, Shadow DOM elements can have ids conflicting with the elements in the light DOM.
-Shadow DOM obtains its own stylesheets. The style rules from the outer DOM will not be applied
EXP:
customElements.define('show-hello', class extends HTMLElement {

```
  connectedCallback() {
        const shadow = this.attachShadow({mode: 'open'});
        shadow.innerHTML = `<p>
        Hello, ${this.getAttribute('name')}
        </p>`;
  }
});
```

********** **Use Strict**** **********

The JavaScript strict mode is used to generate silent errors. It provides "use strict".

- Using a variable, without declaring it, is not allowed.
- Using an object, without declaring it, is not allowed.
- Deleting a variable (or object) is not allowed.
- Deleting a function is not allowed
- Duplicating a parameter name is not allowed.
- Writing to a read-only property is not allowed.

********** **Let/var/Const**** **********

Var

- Variables declared with var are in the function scope.
- Hosting allowed
- Reassign the value are allowed
- Redeclaration of the variable are allowed

Let

- Variables declared as let are in the block scope.
- Hosting are not allowed
- Reassign the value are allowed
- Redeclaration of the variable are not allowed

Const

- Variables declared as const are in the block scope
- Hosting are not allowed.
- Reassign the value are not allowed
- Redeclaration of the variable are not allowed

**.....Arrow Function ********** **

Syntax
No arguments (arguments are array-like objects)
No prototype object for the Arrow function
Cannot be invoked with a new keyword (Not a constructor function)
No own this (call, apply & bind won't work as expected)
It cannot be used as a Generator function

Duplicate-named parameters are not allowed

**.....Hoisting.....**

Hoisting is a behavior where a function or variable can be used before declaration.

**..... This keyword .....**.

In javascript this refers to an object.
In the object method this refers to an object.
Alone, this refers to a global object.
In function this refers to an object.
In a function, in strict mode this is undefined.
In event,this refers to an element that received the event.
Methods like apply,call bind this refer to any object .

**..... JS Proxies.....**

In JavaScript, proxies (proxy object) are used to wrap an object and redefine various operations into the object such as reading, insertion, validation, etc. Proxy allows you to add custom behavior to an object or a function.

Ex.

```
let target={
        name:'anil',
    email:'anil@gmail.com'
}
let handler={
        get:function(obj,prop){
        return obj[prop]?obj[prop]:'Property does not exist'
    }
}
let sethandler={
        set:function(obj,prop,value){
        obj[prop]=value;
          return ;
    }
}

let proxy = new Proxy(target,handler);
let proxy = new Proxy(target,sethandler);
proxy.age=30;
console.log(proxy);
```

**..... Object destructuring.....**

Object Destructuring is the syntax for extracting values from an object property and assigning them to a variable.

Exp.

```
let obj={
   name:'anil',
   emil:'anil@gmail.com',
   address:{
         country:'India',
         state:'up'
   }
}
let {name,emil,address:{country}}=obj;
console.log(country);
```

**..... Array Destructuring.....**

Destructuring the array in JavaScript simply means extracting multiple values from data stored in objects and arrays.

Exp.

```
let arr=[10,20,30,40];
let [a,b,c,d]=arr;
```

**.....Callback function .....**

A JavaScript callback is a function which is to be executed after another function has finished execution. A more formal definition would be - Any function that is passed as an argument to another function so that it can be executed in that other function is called as a callback function.

**.....Callback hell.....**

Callback hell refers to the situation where callbacks are nested within other callbacks several levels deep, potentially making it difficult to understand and maintain the code

**..... Promise .....**

In JavaScript, a promise is a good way to handle asynchronous operations. It is used to find out if the asynchronous operation is successfully completed or not. A promise may have one of three states. Pending. Fulfilled. and rejected .

A Promise is an object representing the eventual completion or failure of an asynchronous operation.

**..... Aync.....**

The async and await keywords allow asynchronous, promise-based behavior to be written more easily and avoid promise chains.

**..... Memory Leak.....**

A Memory leak can be defined as a piece of memory that is no longer being used or required by an application but for some reason is not returned back to the OS.
"piece of memory that is no longer being used of an application "

Exp.

```
let arr = [];
for(let i = 5; i > 1; i++){
        arr.push(i-1)
}
console.log(arr);
```

The four types of common JavaScript leaks

1.Undeclared/ Global Variables
2.Forgotten timers or callbacks
3.Out of DOM references (Event listeners)
4.Closures

**..... Lexical Scope .....**
Lexical scope is the ability for a function scope to access variables from the parent scope. We call the child function to be lexically bound by that of the parent function.

Exp:

```
var scope = "I am global";
function whatismyscope(){
  var scope = "I am just a local";
  function func() {return scope;}
  return func;
}
console.log(whatismyscope()());
```

**.....Higher Order function .....**
Higher order functions are functions that take one or more functions as arguments, or return a function as their result.
Exp.

```
function calculate(operation, initialValue, numbers) {
 let total = initialValue;
 for (const number of numbers) {
        total = operation(total, number);
 }
 return total;
}

function sum(n1, n2) {
```

```
  return n1 + n2;
}

function multiply(n1, n2) {
  return n1 * n2;
}

calculate(sum, 0, [1, 2, 4]);     // => 7
calculate(multiply, 1, [1, 2, 4]); // => 8
```

**..... Prototype.....**

Prototypes are the mechanism by which JavaScript objects inherit features from one another.

Exp.

```
let employee={
        firstName:'Anil Kumar',
    lastName:'Singh',
    fullName:function(){
        return this.firstName+" "+this.lastName;
    }
}

let student={
        firstName:'Anil Kumar',
    lastName:'Singh',
    __proto__:employee
}
console.log(student.fullName());
```

**.....Prototype and __proto__.....**

-prototype is a property of a Function object. It is the prototype of objects constructed by that function.

__proto__ is an internal property of an object, pointing to its prototype. Current standards provide an equivalent Object.getPrototypeOf(obj) method, though the de facto standard __proto__ is quicker.

exp.

```
let emp={
    email:'anil@gmail.com',
}
function Costing(name){
    this.name=name;
}
Costing.prototype.age=20;  //
```

```
let obj= new Costing('Anil');
obj.__proto__=emp;
console.log(obj.email);
```

.....Prototypal and classical inheritance.....
Classical model: Object is created from a class.
Prototypal model: Object is created directly from another object.

**..... CORS.....**
Cross-origin resource sharing (CORS) is a browser mechanism which enables controlled access to resources located outside of a given domain. It extends and adds flexibility to the same-origin policy (SOP). However, it also provides potential for cross-domain attacks, if a website's CORS policy is poorly configured and implemented. CORS is not a protection against cross-origin attacks such as cross-site request forgery (CSRF).

**..... Map Object .....**
A map in JavaScript is a data structure that stores key-value pairs, allowing efficient lookup, insertion, and deletion operations based on the keys.

A Map holds key-value pairs where the keys can be any datatype.
A Map remembers the original insertion order of the keys.
A Map has a property that represents the size of the map.

**Difference between Map/Object**

| Object | Map |
|---|---|
| Not directly iterable | Directly iterable |
| Do not have a size property | Have a size property |
| Keys must be Strings (or Symbols) | Keys can be any datatype |
| Keys are not well ordered | Keys are ordered by insertion |
| Have default keys | Do not have default keys |

**..... Set Object .....**
A JavaScript Set is a collection of unique values. Each value can only occur once in a Set. A Set can hold any value of any data type.

**.....Memory Heap .....**
It is used to store objects and functions in JavaScript. The engine doesn’t allocate a fixed amount of memory. Instead, it allocates more space as required.

**..... Memory Stack.....**

Stack: It is a data structure used to store static data. Static data refers to data whose size is known by the engine during compile time. In JavaScript, static data includes primitive values like strings, numbers, boolean, null, and undefined. References that point to objects and functions are also included. A fixed amount of memory is allocated for static data. This process is known as static memory allocation.

**..... Memory Heap Vs  Memory Stack.....**

| Stack | Heap |
| --- | --- |
| Primitive data types and references | Objects and functions |
| Size is known at compile time | Size is known at run time |
| Fixed memory allocated | No limit for object memory |

..... Call Stack .....
JavaScript is a single-threaded language as it utilizes only one “Call Stack” for the management of the “Global” and “Function“ Execution Contexts. Call Stack in JavaScript is a type of data structure that maintains the tracks of invoked and executed functions.
..... CallBack Queqe.....
Callback Queue: This is where your asynchronous code gets pushed to, and waits for the execution.

..... Event Loop .....
Event loop is just a guardian who keeps a good communication with Call Stack and Callback Queue. It checks if the call stack is free, then lets know the callback queue. Then Callback queue passes the callback function to Call stack to be executed. When all the callback functions are executed, the call stack is out and global execution context is free.

..... Web APIs .....
Web API works as JS engines assistant. When JS engine have to deal with asynchronous code, it takes the help of Web API. Web API handles the blocking behavior of JavaScript code.
In this case, from our code above, we can say Web API will take the callback function.
Exp.
setTimeout(lunch, 3000);

.....Job Queqe.....

Job Queue: Apart from Callback Queue, browsers have introduced one more queue which is “Job Queue”, reserved only for new Promise() functionality. So when you use promises in your code, you add .then() method, which is a callback method. These `thenable` methods are added to Job Queue once the promise has returned/resolved, and then gets executed.

..... Dependency Injection .....

A dependency is any external resource a program needs to work. These can be external libraries the code literally depends on or services the program functionally needs, like internet APIs and databases.

..... Design Pattern.....
A design pattern is a term used in software engineering for a general, reusable solution to a commonly occurring problem in software design.

Singleton Pattern:
Singletons are classes which can be instantiated once, and can be accessed globally. This single instance can be shared throughout our application, which makes Singletons great for managing global state in an application.

Exp:

```
let instance;
let counter=0;
class Counter{
   constructor(){
         if(instance){
               throw new Error("You can only create one instance");
         }
         instance=this;
   }
   getInstance(){
         return this;
   }
   getCount(){
         return counter;
   }
   increment(){
         return ++counter;
   }
   decrement(){
         return ++counter;
   }

}
const singletonCounter = Object.freeze(new Counter());
export default singletonCounter;
```

Proxy Pattern:

Definition:Intercept and control interactions to target objects
or
Definition2:The Proxy Pattern uses a Proxy intercept and control interactions to target objects.
Let's say that we have a person object. We can access properties with either dot or bracket notation, and modify property values in a similar fashion.

```
let proxy= new Proxy(object,handler);
```

```
const person = {
  name: "John Doe",
  age: 42,
  email: "john@doe.com",
  country: "Canada",
};
```

```
const personProxy = new Proxy(person, {
  get: (target, prop) => {
          console.log(`The value of ${prop} is ${target[prop]}`);
          return target[prop];
  },
  set: (target, prop, value) => {
          console.log(`Changed ${prop} from ${target[prop]} to ${value}`);
          target[prop] = value;
          return true;
  },
});
personProxy.age=30;
console.log(personProxy.age);
```

Reflect
JavaScript provides a built-in object called Reflect, which makes it easier for us to manipulate the target object when working with proxies.

Provider Pattern:
Definition 1: Make data available to multiple child components

Definition 2: The Provider pattern/Context API makes it possible to pass data to many components, without having to manually pass it through each component layer. It reduces the risk of accidentally introducing bugs when refactoring code.

Exp: React Provider/Context API.

Prototype Pattern:
Definition 1: Share properties among many objects of the same type.
Exp:

```
class Dog {
  constructor(name) {
        this.name = name;
  }
  bark() {
        return `Woof!`;
  }
}

const dog1 = new Dog("Daisy");
const dog2 = new Dog("Max");
const dog3 = new Dog("Spot");
Dog.prototype.play = () => console.log("Playing now!");
dog1.play();
```

Container/Presentational Pattern:
Sort Definition: Enforce separation of concerns by separating the view from the application logic.
Long Definition: In React, one way to enforce separation of concerns is by using the Container/Presentational pattern. With this pattern, we can separate the view from the application logic.

Exp:

Comp1

```
import React from "react";
import DogImages from "./DogImages";

export default class DogImagesContainer extends React.Component {
  constructor() {
        super();
        this.state = {
        dogs: []
        };
  }

  componentDidMount() {
        fetch("https://dog.ceo/api/breed/labrador/images/random/6")
        .then(res => res.json())
        .then(({ message }) => this.setState({ dogs: message }));
```

```
  }

  render() {
        return <DogImages dogs={this.state.dogs} />;
  }
}
```

Comp 2:

```
import React from "react";
export default function DogImages({ dogs }) {
  return dogs.map((dog, i) => <img src={dog} key={i} alt="Dog" />);
}
```

Observer Pattern:
Sort Definition: Use observables to notify subscribers when an event occurs
Definition: With the observer pattern, we can subscribe certain objects, the observers, to another object, called the observable. Whenever an event occurs, the observable notifies all its observers!

Exp:
Comp1

```
class Observable {
  constructor() {
        this.observers = [];
  }

  subscribe(f) {
        this.observers.push(f);
  }

  unsubscribe(f) {
        this.observers = this.observers.filter(subscriber => subscriber !== f);
  }

  notify(data) {
        this.observers.forEach(observer => observer(data));
  }
}

export default new Observable();
```

Comp 2:

```
import React from "react";
import { Button, Switch, FormControlLabel } from "@material-ui/core";
import { ToastContainer, toast } from "react-toastify";
import observable from "./Observable";

function handleClick() {
  observable.notify("User clicked button!");
}

function handleToggle() {
  observable.notify("User toggled switch!");
}

function logger(data) {
  console.log(`${Date.now()} ${data}`);
}

function toastify(data) {
  toast(data, {
        position: toast.POSITION.BOTTOM_RIGHT,
        closeButton: false,
        autoClose: 2000
  });
}

observable.subscribe(logger);
observable.subscribe(toastify);

export default function App() {
  return (
        <div className="App">
        <Button variant="contained" onClick={handleClick}>
        Click me!
        </Button>
        <FormControlLabel
        control={<Switch name="" onChange={handleToggle} />}
        label="Toggle me!"
        />
        <ToastContainer />
        </div>
  );
}
```

Module Pattern:
Sort Definition: Split up your code into smaller, reusable pieces.

Exp:
math.js

```
function add(x, y) {
  return x + y;
}
function multiply(x) {
  return x * 2;
}
function subtract(x, y) {
  return x - y;
}
function square(x) {
  return x * x;
}
```

index.js

```
console.log(add(2, 3));
console.log(multiply(2));
console.log(subtract(2, 3));
console.log(square(2));
```

Mixin Pattern:
Sort Definition: Add functionality to objects or classes without inheritance:
Definition: A mixin is an object that we can use in order to add reusable functionality to another object or class, without using inheritance. We can't use mixins on their own: their sole purpose is to add functionality to objects or classes without inheritance.

Exp:

```
class Dog {
  constructor(name) {
          this.name = name;
  }
}

const animalFunctionality = {
  walk: () => console.log("Walking!"),
  sleep: () => console.log("Sleeping!")
};
```

```
const dogFunctionality = {
  __proto__: animalFunctionality,
  bark: () => console.log("Woof!"),
  wagTail: () => console.log("Wagging my tail!"),
  play: () => console.log("Playing!"),
  walk() {
        super.walk();
  },
  sleep() {
        super.sleep();
  }
};
Object.assign(Dog.prototype, dogFunctionality);
const pet1 = new Dog("Daisy");
console.log(pet1.name);
pet1.bark();
pet1.play();
pet1.walk();
pet1.sleep();
```

Mediator/Middleware Pattern:
Sort Definition: Use a central mediator object to handle communication between components.

Exp:

```
class ChatRoom {
  logMessage(user, message) {
        const sender = user.getName();
        console.log(`${new Date().toLocaleString()} [${sender}]: ${message}`);
  }
}

class User {
  constructor(name, chatroom) {
        this.name = name;
        this.chatroom = chatroom;
  }

  getName() {
        return this.name;
  }
```

```
  send(message) {
        this.chatroom.logMessage(this, message);
  }
}

const chatroom = new ChatRoom();

const user1 = new User("John Doe", chatroom);
const user2 = new User("Jane Doe", chatroom);

user1.send("Hi there!");
user2.send("Hey!");
```

HOC Pattern:
Definition: Pass reusable logic down as props to components throughout your application.

Exp: Available to counter example in reactjs .

Render Props Pattern:
Definition: Pass JSX elements to components through props.

Exp:

```
import React from "react";
import { render } from "react-dom";
import "./styles.css";
const Title = (props) => props.render();

render(
  <div className="App">
        <Title
        render={() => (
        <h1>
        <span role="img" aria-label="emoji">
        ✨
        </span>
        I am a render prop!{" "}
        <span role="img" aria-label="emoji">
        ✨
        </span>
        </h1>
```

```
        )}
        />
  </div>,
  document.getElementById("root")
);
```

Hooks Pattern:
Sort Definition: Use functions to reuse stateful logic among multiple components throughout the app.

Exp:

```
mport React from "react";
import "./styles.css";
export default class Button extends React.Component {
  constructor() {
        super();
        this.state = { enabled: false };
  }
  render() {
        const { enabled } = this.state;
        const btnText = enabled ? "enabled" : "disabled";

        return (
        <div
        className={`btn enabled-${enabled}`}
        onClick={() => this.setState({ enabled: !enabled })}
        >
        {btnText}
        </div>
        );
  }
}
```

Flyweight Pattern:(Not Important)
Sort Definition: Reuse existing instances when working with identical objects.
Exp:

```
class Book {
  constructor(title, author, isbn) {
        this.title = title;
        this.author = author;
        this.isbn = isbn;
  }
```

```
}

const isbnNumbers = new Set();
const bookList = [];

const addBook = (title, author, isbn, availibility, sales) => {
  const book = {
        ...createBook(title, author, isbn),
        sales,
        availibility,
        isbn
  };

  bookList.push(book);
  return book;
};

const createBook = (title, author, isbn) => {
  const book = isbnNumbers.has(isbn);
  if (book) {
        return book;
  } else {
        const book = new Book(title, author, isbn);
        isbnNumbers.add(isbn);
        return book;
  }
};

addBook("Harry Potter", "JK Rowling", "AB123", false, 100);
addBook("Harry Potter", "JK Rowling", "AB123", true, 50);
addBook("To Kill a Mockingbird", "Harper Lee", "CD345", true, 10);
addBook("To Kill a Mockingbird", "Harper Lee", "CD345", false, 20);
addBook("The Great Gatsby", "F. Scott Fitzgerald", "EF567", false, 20);

console.log("Total amount of copies: ", bookList.length);
console.log("Total amount of books: ", isbnNumbers.size);
```

Factory Pattern:
Sort Definition: Use a factory function in order to create objects.
Definition: With the factory pattern we can use factory functions in order to create new objects.
A function is a factory function when it returns a new object without the use of the new keyword!

Exp:

```
const createUser = ({ firstName, lastName, email }) => ({
  firstName,
  lastName,
  email,
  fullName() {
          return `${this.firstName} ${this.lastName}`;
  }
});

const user1 = createUser({
  firstName: "John",
  lastName: "Doe",
  email: "john@doe.com"
});

const user2 = createUser({
  firstName: "Jane",
  lastName: "Doe",
  email: "jane@doe.com"
});

console.log(user1);
console.log(user2);
```

Compound Pattern:
Sort Definition: Create multiple components that work together to perform a single task.

Definition: In our application, we often have components that belong to each other. They're dependent on each other through the shared state, and share logic together. You often see this with components like select, dropdown components, or menu items. The compound component pattern allows you to create components that all work together to perform a task.

Command Pattern:
Sort Definition: Decouple methods that execute tasks by sending commands to a commander
Definition: With the Command Pattern, we can decouple objects that execute a certain task from the object that calls the method.

Exp:

```
class OrderManager {
  constructor() {
        this.orders = [];
  }

  execute(command, ...args) {
        return command.execute(this.orders, ...args);
  }
}
class Command {
  constructor(execute) {
        this.execute = execute;
  }
}
function PlaceOrderCommand(order, id) {
  return new Command(orders => {
        orders.push(id);
        console.log(`You have successfully ordered ${order} (${id})`);
  });
}
function CancelOrderCommand(id) {
  return new Command(orders => {
        orders = orders.filter(order => order.id !== id);
        console.log(`You have canceled your order ${id}`);
  });
}
function TrackOrderCommand(id) {
  return new Command(() =>
        console.log(`Your order ${id} will arrive in 20 minutes.`)
  );
}
const manager = new OrderManager();
manager.execute(new PlaceOrderCommand("Pad Thai", "1234"));
manager.execute(new TrackOrderCommand("1234"));
manager.execute(new CancelOrderCommand("1234"));
```

.....Rendering Patterns.....
Note: Rendering pattern is not most important.

Client-side Rendering(CSR):
Sort Definition: Render your application's UI on the client.

Exp:

index.html

```
<div id="root"></div>
```

index.js

```
function tick() {
  const element = (
        <div>
        <h1>Hello, world!</h1>
        <h2>It is {new Date().toLocaleTimeString()}.</h2>
        </div>
  );
  ReactDOM.render(element, document.getElementById("root"));
}

setInterval(tick, 1000);
```

Server-side Rendering:
Sort Definition: Generate HTML to be rendered on the server in response to a user request.

Exp:

index.html

```
<!DOCTYPE html>
<html>
  <head>
        <title>Time</title>
  </head>
  <body>
        <div>
        <h1>Hello, world!</h1>
        <b>It is <div id=currentTime></div></b>
        </div>
  </body>
</html>
```

Index.js

```
function tick() {
        var d = new Date();
        var n = d.toLocaleTimeString();
```

```
        document.getElementById("currentTime").innerHTML = n;
}
setInterval(tick, 1000);
```

Static Rendering:
Sort Definition: Deliver pre-rendered HTML content that was generated when the site was built.

Incremental Static Generation:
Sort Definition: Update static content after you have built your site.

Progressive Hydration:
Sort Definition: Delay loading JavaScript for less important parts of the page.

Streaming Server-Side Rendering:
Sort Definition: Generate HTML to be rendered on the server in response to a user request.

React Server Components:
Sort Definition: Server Components compliment SSR, rendering to an intermediate abstraction without needing to add to the JavaScript bundle.

Selective Hydration:
Sort Definition: How to combine streaming server-side rendering with a new approach to hydration, selective hydration.

Islands Architecture:
Sort Definition: The islands architecture encourages small, focused chunks of interactivity within server-rendered web pages

..... Performance Pattern .....

Optimize your loading sequence:
Sort Definition: Learn how to optimize your loading sequence to improve how quickly your app is usable

Static Import:
Sort Definition: Import code that has been exported by another module.

Definition: The import keyword allows us to import code that has been exported by another module. By default, all modules we're statically importing get added to the initial bundle. A module that is imported by using the default ES2015 import syntax, import module from 'module', is statically imported.

Exp:

```
import React from "react";
// Statically import Chatlist, ChatInput and UserInfo
import UserInfo from "./components/UserInfo";
import ChatList from "./components/ChatList";
import ChatInput from "./components/ChatInput";

import "./styles.css";

console.log("App loading", Date.now());

const App = () => (
  <div className="App">
        <UserInfo />
        <ChatList />
        <ChatInput />
  </div>
);

export default App;
```

Dynamic Import:
Sort Definition: Import parts of your code on demand.

Exp:

```
import React, { Suspense, lazy } from "react";
  // import Send from "./icons/Send";
  // import Emoji from "./icons/Emoji";
  const Send = lazy(() =>
        import(/*webpackChunkName: "send-icon" */ "./icons/Send")
  );
  const Emoji = lazy(() =>
        import(/*webpackChunkName: "emoji-icon" */ "./icons/Emoji")
  );
  // Lazy load EmojiPicker  when <EmojiPicker /> renders
  const Picker = lazy(() =>
        import(/*webpackChunkName: "emoji-picker" */ "./EmojiPicker")
  );

  const ChatInput = () => {
        const [pickerOpen, togglePicker] = React.useReducer(state => !state, false);
```

```
        return (
        <Suspense fallback={<p id="loading">Loading...</p>}>
        <div className="chat-input-container">
        <input type="text" placeholder="Type a message..." />
        <Emoji onClick={togglePicker} />
        {pickerOpen && <Picker />}
        <Send />
        </div>
        </Suspense>
        );
};

console.log("ChatInput loaded", Date.now());

export default ChatInput;
```

Import On Visibility:
Sort Definition: Load non-critical components when they are visible in the viewport.
Definition:Besides user interaction, we often have components that aren't visible on the initial page. A good example of this is lazy loading images that aren't directly visible in the viewport, but only get loaded once the user scrolls down.

Exp:

chatInput.js

```
import React from "react";
import Send from "./icons/Send";
import Emoji from "./icons/Emoji";
import LoadableVisibility from "react-loadable-visibility/react-loadable";

const EmojiPicker = LoadableVisibility({
  loader: () => import("./EmojiPicker"),
  loading: <p id="loading">Loading</p>
});

const ChatInput = () => {
  const [pickerOpen, togglePicker] = React.useReducer(state => !state, false);

  return (
```

```
        <div className="chat-input-container">
        <input type="text" placeholder="Type a message..." />
        <Emoji onClick={togglePicker} />
        {pickerOpen && <EmojiPicker />}
        <Send />
        </div>
  );
};

console.log("ChatInput loading", Date.now());

export default ChatInput;
```

ChatList.js

```
import React from "react";
import messages from "../data/messages";

const ChatMessage = ({ message, side }) => (
  <div className={`msg-container ${side}`}>
        <div className="chat-msg">
        <div className="msg-contents">{message}</div>
        </div>
  </div>
);

const ChatList = () => (
  <div className="chat-list">
        {messages.map(message => (
        <ChatMessage
        message={message.body}
        key={message.id}
        side={["left", "right"][Number(message.senderId === 1)]}
        />
        ))}
  </div>
);
export default ChatList;
```

webpack.config.js

```
onst path = require("path");
const HTMLWebpackPlugin = require("html-webpack-plugin");
```

```
module.exports = {
  entry: "./src/index.js",
  module: {
          rules: [
          {
          test: /.(js|jsx)$/,
          exclude: /node_modules/,
          use: ["babel-loader"]
          },
          {
          test: /.css$/i,
          use: ["style-loader", "css-loader"]
          }
          ]
  },
  resolve: {
          extensions: ["*", ".js", ".jsx"]
  },
  mode: "development",
  output: {
          path: path.resolve(__dirname, "dist"),
          filename: "[name].bundle.js"
  },
  plugins: [
          new HTMLWebpackPlugin({
          template: path.resolve(__dirname, "dist", "index.html")
          })
  ],
  devServer: {
          contentBase: path.join(__dirname, "dist"),
          compress: true,
          port: 9000
  }
};
```

Import On Interaction:
Sort Definition: Load non-critical resources when a user interacts with UI requiring it.
Definition:
Your page may contain code or data for a component or resource that isn't immediately necessary. For example, part of the user-interface a user doesn't see unless they click or scroll on parts of the page. This can apply to many kinds of first-party code you author, but this also

applies to third-party widgets such as video players or chat widgets where you typically need to click a button to display the main interface.

When the user clicks to interact with that component for the first time
Scrolls the component into view
or deferring load of that component until the browser is idle (via requestIdleCallback).

The different ways to load resources are, at a high-level:

Eager - load resource right away (the normal way of loading scripts)
Lazy (Route-based) - load when a user navigates to a route or component
Lazy (On interaction) - load when the user clicks UI (e.g Show Chat)
Lazy (In viewport) - load when the user scrolls towards the component
Prefetch - load prior to needed, but after critical resources are loaded
Preload - eagerly, with a greater level of urgency

Route Based Splitting:
Sort Definition: Dynamically load components based on the current route.
Definition: We can request resources that are only needed for specific routes, by adding route-based splitting. By combining React Suspense or loadable-components with libraries such as react-router, we can dynamically load components based on the current route

Exp:

```
import React, { lazy, Suspense } from "react";
import { render } from "react-dom";
import { Switch, Route, BrowserRouter as Router } from "react-router-dom";

const App = lazy(() => import(/* webpackChunkName: "home" */ "./App"));
const Overview = lazy(() =>
  import(/* webpackChunkName: "overview" */ "./Overview")
);
const Settings = lazy(() =>
  import(/* webpackChunkName: "settings" */ "./Settings")
);

render(
  <Router>
        <Suspense fallback={<div>Loading...</div>}>
        <Switch>
        <Route exact path="/">
        <App />
        </Route>
```

```
        <Route path="/overview">
        <Overview />
        </Route>
        <Route path="/settings">
        <Settings />
        </Route>
        </Switch>
        </Suspense>
  </Router>,
  document.getElementById("root")
);

module.hot.accept();
```

Bundle Splitting:
Sort Definition: Split your code into small, reusable pieces.
Definition: When building a modern web application, bundlers such as Webpack or Rollup take an application's source code, and bundle this together into one or more bundles. When a user visits a website, the bundle is requested and loaded in order to display the data to the user's screen.

PRPL Pattern:
Sort Definition: Optimize initial load through precaching, lazy loading, and minimizing roundtrips.

Definition:

Making our applications globally accessible can be a challenge! We have to make sure the application is performant on low-end devices and in regions with a poor internet connectivity. In order to make sure our application can load as efficiently as possible in difficult conditions, we can use the PRPL pattern.

The PRPL pattern focuses on four main performance considerations:

Pushing critical resources efficiently, which minimizes the amount of roundtrips to the server and reducing the loading time.
Rendering the initial route soon as possible to improve the user experience
Pre-caching assets in the background for frequently visited routes to minimize the amount of requests to the server and enable a better offline experience
Lazily loading routes or assets that aren't requested as frequently

Tree Shaking:

Sort Definition: Reduce the bundle size by eliminating dead code.
Definition: It can happen that we add code to our bundle that isn't used anywhere in our application. This piece of dead code can be eliminated in order to reduce the size of the bundle, and prevent unnecessarily loading more data! The process of eliminating dead code before adding it to our bundle, is called tree-shaking.

Preload:
Sort Definition: Inform the browser of critical resources before they are discovered.
Definition:
Preload (<link rel="preload">) is a browser optimization that allows critical resources (that may be discovered late) to be requested earlier. If you are comfortable thinking about how to manually order the loading of your key resources, it can have a positive impact on loading performance and metrics in the Core Web Vitals. That said, preload is not a panacea and requires an awareness of some trade-offs.

Exp:

```
<link rel="preload" href="emoji-picker.js" as="script">
...
</head>
<body>
...
<script src="stickers.js" defer></script>
<script src="video-sharing.js" defer></script>
<script src="emoji-picker.js" defer></script>
```

Prefetch:
Sort Definition: Fetch and cache resources that may be requested some time soon.
Definition:
Prefetch (<link rel="prefetch">) is a browser optimization which allows us to fetch resources that may be needed for subsequent routes or pages before they are needed. Prefetching can be achieved in a few ways. It can be done declaratively in HTML (such as in the example below), via a HTTP Header (Link: </js/chat-widget.js>; rel=prefetch), Service Workers or via more custom means such as through Webpack.

Exp:

```
port React, { Suspense, lazy } from "react";
import Send from "./icons/Send";
import Emoji from "./icons/Emoji";

const EmojiPicker = lazy(() =>
  import(/*webpackPrefetch: true,
```

```
        webpackChunkName: "emoji-picker"*/
  "./EmojiPicker")
);
const ChatInput = ({ emojiPicker, gifPicker }) => {
  const [pickerOpen, togglePicker] = React.useReducer(state => !state, false);

  return (
        <div className="chat-input-container">
        <input type="text" placeholder="Type a message..." />
        <Emoji onClick={togglePicker} />
        {pickerOpen && (
        <Suspense fallback={<p id="loading">loading</p>}>
        <EmojiPicker />
        </Suspense>
        )}
        <Send />
        </div>
  );
};
console.log("ChatInput loading", Date.now());
export default ChatInput;
```

**Optimize loading third-parties:**
**Sort Definition:**Reduce the performance impact third-party scripts have on your site.
**Definition:**

List Virtualization:
Sort Definition: Optimize list performance with list virtualization.
Definition:
In this guide, we will discuss list virtualization (also known as windowing). This is the idea of rendering only visible rows of content in a dynamic list instead of the entire list. The rows rendered are only a small subset of the full list with what is visible (the window) moving as the user scrolls. This can improve rendering performance.

**Compressing JavaScript:**
**Sort Definition:** Reduce the time needed to transfer scripts over the network.
**Definition:**

Compress your JavaScript and keep an eye on your chunk sizes for optimal performance. Overly high JavaScript bundle granularity can help with deduplication & caching, but can suffer from poorer compression & impact loading in the 50-100 chunks range (due to browser processes, cache checks etc). Ultimately, pick the compression strategy that works best for you.