Question Bank

React

1) Describe the process of setting up the environment for React.js development. ***

1. Install Node.js and its package manager, npm, from the official Node.js website.
2. Choose a code editor that suits your workflow; popular options include Visual Studio Code, Sublime Text, and Atom.
3. For a setup, use Create React App by running npx create-react-app my-app in your terminal, replacing "my-app" with your desired project name.
4. This command generates a project directory containing all essential React files and dependencies.
5. Navigate to the project directory using cd my-app.
6. Start the development server with npm start, which launches your React app in a web browser for live coding.
7. Make edits to your React components, and the development server automatically refreshes the browser to reflect the changes.
8. This approach allows you to focus on writing React code without extensive manual setup.
9. Depending on the requirements of your project, you may need to install additional dependencies, such as Redux using npm- ‘npm install redux react-redux’

2) Explain the importance of Form Validation in React.Js forms.

In React.js applications, form validation serves a critical role in ensuring data integrity and enhancing user experience. By implementing form validation, you can:

1. Improve data quality: Validated data minimizes the chance of inconsistencies or incorrect information being stored in your application's database.
2. Guarantee expected data formats: Validation rules verify that user input adheres to specific criteria, such as required fields, email formats, or minimum password lengths.
3. Simplify server-side logic: With validated data on the client-side, server-side logic can focus on core functionalities rather than basic input checks.
4. Maintain a clean codebase: Dedicated validation logic promotes code reusability and separation of concerns within your React components.
5. Prevent server-side errors: Catching invalid data on the client-side reduces unnecessary processing and error handling on the server.
6. Boost application security: Validation can act as a preliminary defense against malicious attacks that exploit invalid form submissions.
7. Establish a foundation for complex forms: As your forms become more intricate, a robust validation strategy ensures they continue to function effectively.
8. Offer a polished user interface: Visually highlighting invalid fields and providing clear error messages leads to a more professional and user-friendly application.
9. Enhance user experience: Clear error messages guide users towards providing correct information, preventing frustration and wasted submission attempts.
10. Increase form completion rates: By informing users about errors immediately, validation helps them rectify mistakes and successfully submit forms.

3) Discuss the use of properties (props) in React.Js components.

- In React.js, props stands for properties.
- They are a powerful mechanism for passing data from parent components to child components.

Following are the uses of props in react.js:

1. Passing Data from Parent to Child Components:
   By passing props, parent components can provide specific data or behaviour to their child components.
2. Immutable and Read-Only:
   Props are immutable and read-only within the child component thus they cannot directly modify the props passed to them by their parent components.
   This immutability ensures data integrity and helps maintain a unidirectional data flow, where changes in parent components propagate down to their child components.
3. Customizing Child Components:
   Props allow for the customization and configuration of child components by passing different props to the same component.
   This can dynamically change its behaviour, appearance, or content.
4. Conditional Rendering:
   Props are often used for conditional rendering, where components render different content based on the values of their props.
   By passing props with conditional values, you can control when and how components are rendered in the UI, providing a dynamic and responsive user experience.
5. Event Handling:
   Props can also be used to pass event handlers from parent components to child components which allows child components to trigger actions or update the state of their parent components when certain events occur.
6. Type Checking with PropTypes:
   React provides a mechanism called PropTypes for type checking props to ensure that the correct types of data are passed to components.
   PropTypes help catch bugs early in the development process and provide clear documentation for the expected props of a component, making code easier to understand and maintain.

4) Define Shipping Details and Delivery Details. Discuss the Shipping Details and Delivery Details steps in a form wizard in React.js

- Shipping details refer to the information required to ship a product from the seller to the buyer and includes the recipient's name, address, contact information, and any special instructions for delivery.
- Delivery details encompass the specifics of how the purchased items will be delivered to the recipient, including shipping method, delivery timeframe, and any additional services requested by the buyer.

Following are the steps involved:

1. The Shipping Details step gathers information like name, address, and potentially a phone number for where the order will be delivered.
2. Input fields and validation rules ensure accurate data collection for shipping purposes.
3. The Delivery Details step presents users with various delivery options, such as standard, expedited, or express shipping.
4. Each option typically displays estimated delivery times and corresponding costs.
5. Users select their preferred delivery method based on their desired speed and budget.
6. The form wizard validates the chosen delivery option based on the provided shipping address or any relevant restrictions.
7. Once both steps are complete, the user can proceed to the final steps, such as payment and order confirmation.
8. This separation streamlines the checkout process and allows users to focus on specific details in each step.

5) Explain the steps to set up a React application for handling forms and discuss the necessary dependencies and configuration needed to get started. ***

8) Describe the steps involved in setting up a React.Js application for handling forms.

1. Initiate your React project using Create React App: npx create-react-app my-form-app.
2. This command establishes a project structure with essential React dependencies.
3. To manage form state, import the useState hook from the react library.
4. Define state variables using useState to hold form field values like name or email.
5. Create form elements (e.g., <input>, <textarea>) and bind their value attribute to the corresponding state variable.
6. Implement event handlers (e.g., onChange) for form elements to update state on user input.
7. For complex forms or validation, consider using a third-party library like react-hook-form or Formik.
8. With this setup, you can build dynamic and interactive forms in your React application.

6) Discuss the different lifecycle phases of a React component and the lifecycle methods associated with each phase in React.js.

React components go through a well-defined lifecycle with specific methods for different stages. Here's a breakdown of the three main phases and their associated methods:

- Mounting Phase: This phase occurs when a component is first created and inserted into the DOM. Methods involved include:

1. constructor(): Used for initializing state and binding event handlers (optional).
2. render(): This mandatory method defines the component's JSX output that gets rendered to the DOM.
3. componentDidMount(): Ideal for network requests to fetch data after the component is rendered.

- Updating Phase: This phase kicks in whenever a component's state or props change. The methods involved are:

1. getDerivedStateFromProps(props, state) (optional): Allows state updates based on changes in props before re-rendering.

2. shouldComponentUpdate(nextProps, nextState) (optional): Controls whether a component should re-render based on upcoming prop and state changes.
3. render(): This method is called again to reflect the updated state or props in the UI.
4. componentDidUpdate(prevProps, prevState, snapshot) (optional): Useful for side effects after the component re-renders, like DOM manipulation using refs.

- Unmounting Phase: This final phase happens when a component is removed from the DOM. The method involved is:

1. componentWillUnmount(): Used for cleanup tasks like removing event listeners or clearing intervals to prevent memory leaks.

7) Explain the difference between Props and State in React.js with examples.

SN	Props	State
1.	Props are read-only.	State changes can be asynchronous.
2.	Props are immutable.	State is mutable.
3.	Props allow you to pass data from one component to other components as an argument.	State holds information about the components.
4.	Props can be accessed by the child component.	State cannot be accessed by child components.
5.	Props are used to communicate between components.	States can be used for rendering dynamic changes with the component.
6.	Stateless component can have Props.	Stateless components cannot have State.
7.	Props make components reusable.	State cannot make components reusable.
8.	Props are external and controlled by whatever renders the component.	The State is internal and controlled by the React Component itself.

9) Explain the role of DOM events of React.js in traditional web development.

- In traditional web development, DOM events play a fundamental role in enabling interactivity and responsiveness within web applications.
- Without frameworks like React.js, developers typically handle DOM events directly by attaching event listeners to HTML elements using JavaScript.

DOM events in traditional web development facilitate various functionalities such as:

1. User Interaction: DOM events enable developers to capture user interactions like clicks, mouse movements, and keyboard inputs, allowing for the implementation of interactive features such as dropdown menus, sliders, and drag-and-drop functionality.
2. Form Handling: DOM events are crucial for managing form submissions, validation, and handling user input in web forms. Developers use events like submit, change, and input to respond to form interactions and update the UI accordingly.

3. Navigation: DOM events play a role in managing navigation within web applications, such as responding to anchor (<a>) tag clicks to navigate to different pages or sections of a website.
4. Asynchronous Operations: DOM events are used to trigger asynchronous operations such as fetching data from servers or performing background tasks in response to user actions without blocking the main UI thread.
5. Error Handling: Developers utilize DOM events to capture and handle errors that occur during the execution of JavaScript code, providing a mechanism for graceful error recovery and debugging.

Although handling DOM events directly can sometimes lead to complex and tightly coupled code, making it challenging to maintain and scale applications.

10) Describe a detailed example of a stateful TextArea component in React.js.

11) Describe the pros and cons of React.js.

Advantages:

1. Component-Based Architecture: It promotes code reusability and maintainability by breaking down UIs into independent, reusable components.
2. Virtual DOM: React employs a virtual DOM for efficient updates, improving performance and responsiveness of web applications.
3. JSX Syntax: JSX, a blend of JavaScript and HTML, enhances readability and makes UI development more intuitive.
4. Large Community and Ecosystem: React benefits from a vast developer community and a rich ecosystem of libraries and tools.
5. SEO Friendliness: React applications can be SEO-friendly with server-side rendering or static site generation techniques.

Disadvantages:

1. Learning Curve: While considered beginner-friendly, understanding JSX and core React concepts requires an initial learning investment.
2. Rapid Development Pace: The fast evolution of React can necessitate keeping up with frequent updates and potential library version conflicts.
3. Lack of Built-in Features: React is a library, not a full framework. Routing, state management, and other functionalities often require additional libraries.
4. Complexity for Simple Projects: For small-scale applications, React's overhead might be unnecessary compared to simpler solutions.
5. Potential Overuse of Components: Overly granular component structures can make code harder to reason about and maintain.

Node

11) Describe the types of modules in Node.js.***

Node.js offers three main types of modules to organize and share code:

1. Core Modules: These are built-in modules that come pre-installed with Node.js.
   They provide essential functionalities like file system access (fs), networking (http), and working with the operating system (os).
2. Local Modules: These are modules you create yourself for your specific project.
   They reside within your project directory and allow you to organize your codebase into reusable components.
   You can import and export functions, variables, or even entire classes from local modules.
3. Third-Party Modules: Node.js boasts a rich ecosystem of third-party modules published on the npm (Node Package Manager) registry.
   These modules offer a wide range of functionalities beyond the core modules, including database interaction, web frameworks (like Express.js), and UI libraries.
   You can easily install and use these modules in your project using npm.

12) Explain the role of npm in Node.js development.

1. Providing a Registry: It offers a vast online repository containing millions of reusable code packages for various functionalities.
2. Package Installation: npm simplifies installing these packages into your project with a single command.
3. Dependency Management: It automatically downloads and installs all dependencies required by a package, ensuring compatibility and completeness.
4. Version Control: npm allows specifying version ranges for dependencies, enabling updates while maintaining project stability.
5. Versioning and Updates: It facilitates easy management of package versions within your project and provides commands for updating them.
6. Sharing and Collaboration: npm enables developers to share their own code packages with the community, fostering collaboration and innovation.
7. Streamlined Development: By leveraging pre-built and well-tested packages, npm significantly reduces development time and effort.
8. Consistent Ecosystem: npm establishes a standardized way to manage code dependencies, promoting a cohesive development experience for Node.js projects.

13) Explain the role of web sockets in Node.js applications.

WebSockets enhance Node.js applications by enabling real-time, two-way communication between a server and connected clients.

1. Real-time data exchange: Servers can push data updates to clients instantly, fostering dynamic and responsive applications.
2. Bidirectional communication: Both clients and servers can initiate communication, enabling features like collaborative editing or live chat.
3. Efficient data transfer: WebSockets optimize data transfer by sending smaller data packets compared to full HTTP requests.
4. Reduced server load: Persistent connections eliminate the need for frequent request-response cycles, improving server performance.
5. Node.js suitability: Node.js's event-driven architecture is well-suited for handling multiple WebSocket connections simultaneously.

6. Diverse applications: WebSockets empower Node.js developers to build various interactive features, from real-time dashboards to multiplayer games.

14) Define streams in Node.js and explain how they differ from traditional I/O operations.

In Node.js, streams represent an abstract interface for handling data in a continuous, flowing manner which deal with data in smaller pieces.

It differs from traditional I/O in following aspects:

1. Data Handling: Traditional I/O functions like fs.readFileSync work with entire data buffers at once. Streams, on the other hand, process data in smaller chunks as it becomes available. This allows for more efficient memory usage and avoids loading massive datasets entirely.
2. Synchronous vs. Asynchronous: Traditional I/O operations are synchronous, meaning the program execution pauses until the operation (e.g., reading a file) finishes. This can hinder performance for larger data sets. Streams are asynchronous, enabling the program to continue execution while data processing occurs in the background.
3. Blocking vs. Non-blocking: Traditional I/O blocks the program until the operation completes. Streams are non-blocking, allowing the program to handle other tasks concurrently while data processing happens asynchronously
4. Event-Driven vs. Procedural: Traditional I/O follows a procedural approach where you write code for specific steps. Streams leverage events to notify the program when data is ready or when errors occur. This event-driven nature allows for a more reactive and flexible approach to data handling.
5. Limited Functionality vs. Flexibility: Traditional I/O typically offers basic read/write functionalities. Streams come in various types (readable, writable, duplex, transform) enabling them to handle diverse data sources and destinations, and even modify data on the fly.

15) Discuss the role of modules in Node.js. Explain how to create and use modules in a Node.js application.

In Node.js, modules are reusable blocks of code that encapsulate functionality.

They perform following roles:

1. Code Organization: Modules help break down complex applications into smaller, manageable units.
2. Reusability: Modules can be reused throughout your Node.js application, or even in other projects which saves development time and effort.
3. Dependency Management: Modules can have dependencies on other modules which allows you to leverage existing functionality from external sources (like the npm registry) without re-inventing the wheel.
4. Separation of Concerns: Modules promote separation of concerns by isolating specific functionalities within their own boundaries. This improves code organization and reduces the potential for conflicts or side effects.

Creating a Module:

Begin by creating a new JavaScript file (e.g., mymodule.js) within your project directory. Write your code within this file, encapsulating the desired functionality (functions, variables, classes). Use the exports object to specify which parts of your code should be accessible from other modules. You can assign functions, variables, or entire objects to the exports object.

```
exports.add = function(a, b) {
  return a + b;
}
```

Using a Module:

In another JavaScript file within your project, use the require function to import the module you created. Provide the path to the module file within the require function call. This path can be relative (e.g., ./myModule.js) or absolute. Once imported, the module's exported elements become available in the current file. You can access them using the variable name assigned during export.

```
const math = require('./mymodule);

const result = math.add(5, 3);
console.log(result);
```

16) Explain the purpose of the npm init command and write steps to initialize a new Node.js project.

Create a simple Node.js script that prints "Hello, World!" to the console. ***

The npm init command serves a crucial role in Node.js project initialization. It performs the following tasks:

1. It sets up the basic project structure, including a package.json file, which acts as the project manifest.
2. It guides you through a series of prompts to gather information about your project, such as its name, version, description, and entry point.
3. The package.json file helps manage project dependencies by specifying the external libraries your project relies on.

Steps to create a simple node.js project:

1. Open terminal, navigate to the desired directory and run ‘npm init -y’ command.
2. The -y flag tells npm init to accept all default values for the prompts.
3. Create a new JavaScript file named hello.js in your project directory and write the following code in it.
4. console.log(“Hello, world!”);
5. Run ‘node hello.js’ I terminal to run the code.

17)Explain the event-driven architecture of Node.js.

18) Discuss the importance of streams and buffers in Node.js. Provide examples of scenarios where streams and buffers are used.

1. Streams enable processing data in smaller chunks as it becomes available, avoiding the need to load everything into memory at once.

2. Streams operate asynchronously, allowing the program to continue execution while data processing happens in the background. This enhances overall application responsiveness as the program doesn't wait for I/O operations to complete.
3. Streams are well-suited for handling large or continuous data streams. They can be easily chained together using pipes, enabling the creation of complex data processing pipelines that handle data efficiently.
4. Streams come in various types (readable, writable, duplex, transform) catering to different data sources and processing needs. This flexibility makes them adaptable to diverse application requirements.
5. Streams form the foundation for many Node.js functionalities like working with files, network communication (HTTP), and real-time data processing (WebSockets). Understanding streams is essential for effective Node.js development.

Example: Instead of loading the entire file into memory at once, a readable stream can be used to process the file content in chunks, improving memory usage for very large files.

1. Buffers serve as temporary storage for data chunks within streams. This allows for smoother data flow by accommodating network latency or processing delays without data loss.

Example: Network communication often involves sending and receiving data packets. Buffers are used on both sending and receiving ends to temporarily store these data packets before they are transmitted or processed further

19) Explain the concept of callbacks in Node.js. Provide examples demonstrating the use of callbacks in asynchronous functions.

- In Node.js, callbacks are functions passed to asynchronous operations.
- They act as messengers, telling the program what code to execute after the asynchronous operation finishes.
- This allows your program to remain responsive while waiting for tasks like file reads or network requests to complete.
- However, callbacks can become nested and lead to "callback hell," making code complex and hard to read

```
function greet(name, callback)
{
    setTimeout(() =>
    {
        const msg = "Hello, " + name + "!";
        callback(msg);
    }
    , 2000);
}
greet("Alice", (msg) =>
{
    console.log(msg);
});
console.log("Waiting for greeting...");
```

20) How does Node.js connect to databases? Explain different methods or libraries available for connecting Node.js to databases, citing examples.

1. Native Drivers: Node.js provides built-in modules for popular databases like mysql2 for MySQL or pg for PostgreSQL.
   These offer low-level control and performance benefits.
2. Object-Relational Mappers (ORMs): Popular ORMs like Sequelize or TypeORM simplify database interaction by mapping database tables to JavaScript objects.
   They handle SQL queries and relationships, improving data manipulation.
3. Object Data Modeling (ODM): For NoSQL databases like MongoDB, libraries like Mongoose offer an ODM layer.
   They model documents as JavaScript objects and provide a schema-based approach for managing data.
4. Query Builders: Libraries like Knex.js provide a more flexible way to build SQL queries using JavaScript syntax.
   This offers control over complex queries while remaining separate from application logic.

The choice depends on your database type and project needs. For beginners, an ORM can simplify data interaction, while experienced developers might prefer native drivers or query builders for more control.

Typescript

21) Differentiate between "let" and "var" in TypeScript. Give appropriate examples to illustrate your points.

	var	**let**	**const**
origins	pre ES2015	ES2015(ES6)	ES2015(ES6)
scope	globally scoped OR **function** scoped. attached to window object	globally scoped OR **block** scoped	globally scoped OR **block** scoped
global scope	is attached to Window object.	not attached to Window object.	attached to Window object.
hoisting	**var** is hoisted to top of its execution (either global or function) and initialized as ***undefined***	**let** is hoisted to top of its execution (either global or block) and left **uninitialized**	**const** is hoisted to top of its execution (either global or block) and left **uninitialized**
redeclaration within scope	**yes**	**no**	**no**
reassigned within scope	**yes**	**yes**	**no**

22) Explain the concept of decorators in TypeScript.

Decorators are a powerful syntactic feature in TypeScript that allows you to modify the behaviour of classes, properties, methods, and other parts of your code at runtime.

1. Decorators are functions prefixed with the @ symbol.
2. They are applied to code elements like classes, methods, or properties.
3. They are typically used for adding functionality like logging, authorization checks, or dependency injection.
4. They can be chained together to apply multiple modifications to a single code element.
5. While powerful, overuse of decorators can make code harder to read and debug.
6. TypeScript decorators provide a typed and structured approach to modifying code behavior, enhancing code maintainability.

23) Explain the concept of Type Annotations and Assertion in TypeScript and also state their importance.

Type Annotations:

- Act as hints for the TypeScript compiler, specifying the expected data type for variables, functions, and arguments.
- Enhance code readability by explicitly stating what type of data a variable should hold or what a function expects and returns.
- The compiler uses these annotations for static type checking, catching potential type errors during development.

Type Assertions:

- Used to tell the compiler to trust you and treat an expression as a specific type, even if the compiler might infer it differently.
- Useful in scenarios where the compiler cannot definitively determine the type due to dynamic code or external data sources.
- Should be used judiciously, as incorrect assertions can bypass type safety checks and introduce runtime errors.

Importance:

- Both Type Annotations and Assertions contribute to better code quality.
- Annotations improve maintainability and developer experience by clarifying data types.
- Assertions, used carefully, can provide flexibility when dealing with dynamic data.
- Together, they make TypeScript a powerful tool for building robust and type-safe applications.

24) Describe the interfaces in TypeScript. Give an example of defining and implementing an interface in TypeScript code.

1. In TypeScript, interfaces define contracts for objects, outlining the properties and methods they should have, but not the implementation details.
2. They act as blueprints, specifying the expected structure and behaviour of objects.
3. By referring them, the TypeScript compiler can enforce type safety, ensuring objects stick to the defined contract.
4. They separate the object's structure from its implementation, promoting loose coupling and code reusability.

5. They can be used to define the expected types of function parameters, improving type safety and code clarity.
6. They can be extended to inherit properties and methods from other interfaces, promoting code organization and reusability.

```
interface Person {
    name: string;
    age: number;
}

function printInfo(person: Person)
{
    console.log(`Name: ${person.name}, Age: ${person.age}`);
}

let john: Person = {
    name: "John Doe",
    age: 30
};

printInfo(john);
```

25) Discuss the benefits of using TypeScript over JavaScript in web development projects.

1. Type Safety: TypeScript offers static type checking which helps catch potential type errors during development, reducing runtime bugs and improving code reliability.
2. Improved Maintainability: Type annotations in TypeScript enhance code readability by explicitly stating the expected data types for variables and functions. This makes code easier to understand and maintain for both you and other developers.
3. Better Developer Experience: TypeScript provides features like autocompletion and refactoring tools that leverage type information. This can significantly improve developer productivity and reduce the time spent debugging common type-related issues.
4. Early Error Detection: TypeScript catches type errors during development, preventing them from causing unexpected behaviour at runtime. This leads to a more robust and predictable codebase.
5. Large-Scale Application Suitability: Type safety and improved maintainability make TypeScript particularly valuable for large-scale web applications. It helps manage complexity and reduces the chances of introducing errors as the codebase grows.
6. Integration with Existing JavaScript: TypeScript is a superset of JavaScript, meaning existing JavaScript code can be used within TypeScript projects. This allows for a gradual migration from JavaScript to TypeScript without a complete rewrite.

26)Define generics in TypeScript. Discuss how generics provide a way to create reusable, type-safe functions and data structures.

27) Discuss the significance of enums in TypeScript.

1. Type Safety: Enums define a set of named constants with a specific type (numeric or string), enhancing type safety and preventing accidental assignment of incorrect values.

2. Improved Readability: Enum names are more descriptive than raw numbers or strings, making code easier to understand and maintain. For example, Color.Red is clearer than simply using the number 1.
3. Error Prevention: Enums restrict values to the defined set, preventing typos or assignment of invalid constants, which could lead to runtime errors.
4. Automatic Iteration: Enums can be easily iterated over, allowing for code that efficiently processes all defined enum values.
5. Reverse Mapping: For numeric enums, TypeScript offers reverse mapping, allowing you to retrieve the enum name from its numeric value, which can be helpful in debugging or logging situations.
6. Code Reusability: Enums can be used across different parts of your codebase, promoting consistency and making it easier to modify constant values in a centralized location.

28)Explain the concept of basic types in TypeScript. Provide examples of primitive data types such as number, string, boolean, and null. Discuss their significance and usage in TypeScript code.

TypeScript inherits basic types from JavaScript, providing a foundation for building more complex data structures. These basic types include:

1. Primitive Types: Represent fundamental data types like numbers (including integers and floating-point numbers), strings, booleans, symbols (unique and immutable identifiers), and undefined/null.
2. Literal Types: Allow representing specific string or numeric values directly, offering more precise type information (e.g., "hello" or 42).
3. Any Type: Acts as an escape hatch, allowing any data type to be assigned to a variable. It should be used cautiously as it bypasses type safety checks.

1. String: Represents textual data. Used for text labels, user input, and displaying information.
   let name: string = "Apple";
2. Number: Represents numeric values (whole numbers or decimals).
   let age: number = 30;
3. Boolean: Represents true or false values.
   let isLoggedIn: boolean = true;
4. null: Represents the intentional absence of a value. Used to indicate no value exists for a variable.
   let profilePicture: string | null = null;

Bootstrap and ES6

29) Describe the arrow functions in ES6. Implement a function convertToUpperCase using arrow function syntax that takes an array of strings as input and returns a new array with each string converted to uppercase. (use map method).

- Arrow functions, introduced in ES6, are a shorthand way to write JavaScript functions.
- They use the => symbol instead of the function keyword.

- Arrow functions inherit the value from their surrounding code, and perform the given action for it.

```
const convertToUpperCase = (arr) => arr.map(str => str.toUpperCase());
const names = ["apple", "banana", "cherry"];
const upperNames = convertToUpperCase(names);
console.log(upperNames);
```

30) Describe the MERN stack with each component.

The MERN stack refers to a popular combination of technologies used for building modern web applications. Here's a breakdown of each component:

1. MongoDB: A NoSQL document database that stores data in flexible JSON-like documents. It offers scalability and ease of use for storing diverse data structures.
2. Express.js: A lightweight web framework built on top of Node.js that provides a robust foundation for building web applications and APIs. It simplifies server-side logic and routing.
3. React.js: A JavaScript library for building user interfaces. React uses a component-based approach and virtual DOM for efficient rendering, leading to dynamic and responsive web applications.
4. Node.js: A JavaScript runtime environment that allows you to execute JavaScript code outside of a web browser. It's ideal for building server-side applications and APIs due to its asynchronous and event-driven nature.

Together, these technologies offer several advantages:

1. Scalability: Both MongoDB and Node.js are well-suited for handling large datasets and concurrent requests.
2. Flexibility: MongoDB's schema-less nature and React's component-based architecture allow for adaptability to changing requirements.
3. Rich User Interfaces: React enables building interactive and dynamic user interfaces.
4. Full-Stack JavaScript: Using JavaScript on both the front-end (React) and back-end (Node.js) simplifies development and reduces the need for context switching between different languages.

31) Explain Bootstrap Components. ***

1. Breadcrumb
    - Bootstrap Breadcrumbs provide a visual and navigational hierarchy to users.
    - They display a list of links representing the user's location within a multi-step process or website structure.
    - This helps users understand their current position and allows them to easily navigate back to previous steps.
2. Popover
    - Bootstrap Popover components are small contextual pop-up windows triggered by user interaction (like hover or click) on an element.
    - They display additional information or options related to the clicked element without requiring a fullpage load.

    - Popovers are useful for providing quick explanations, definitions, or supplementary content without disrupting the main page flow.
3. Dropdowns
    - Bootstrap Dropdowns offer a space-saving way to present a list of options or actions related to a primary button or link.
    - Clicking the button triggers a dropdown menu displaying the list items.
    - Users can then select the desired option from the dropdown menu, promoting a clean and organized interface for presenting choices.
4. Progress Bar
    - Bootstrap Progress Bars provide a visual indication of the completion status of a task or process.
    - They are customizable with different styles (e.g., striped, animated) and can display percentages or text labels.
    - Progress bars are helpful for conveying loading times, upload progress, or any ongoing operation's advancement.
5. Form
    - Bootstrap Forms offer a collection of pre-built styles and components for creating user input elements.
    - These include standard form controls like text inputs, checkboxes, radio buttons, select menus, and text areas.
    - Bootstrap forms simplify form creation with consistent styling, layout options, and validation features.

32)Explain the principles of classes, inheritance, encapsulation, and polymorphism of object-oriented programming (OOP) in ES6.

33)Define and explain the concept of functional programming using JavaScript.

34)Define symbols, iterators/generators, and map/set in JavaScript.

1. Symbols: Unique and immutable identifiers used as object property keys to avoid naming conflicts and prevent accidental property name modification.
2. Iterators/Generators: Iterators provide a way to access elements of a data structure one at a time, while generators are functions that can pause execution and yield values iteratively.
3. Map: Maps are collections that use key-value pairs for storing data, allowing for any data type as keys.
4. Sets are collections of unique values, ensuring no duplicates exist within the set

35) Explain promises, async/await, callbacks, and generators in JavaScript asynchronous programming. ***

1. Callbacks: In asynchronous JavaScript, callbacks are functions passed to other functions to be executed after an asynchronous operation (like network requests) finishes. They can become nested and lead to "callback hell," making code hard to read.
2. Promises: Promises offer a more structured approach to asynchronous operations. They represent the eventual completion (or failure) of an asynchronous task. Promises can be chained together using .then and .catch methods to handle success and error scenarios, improving code readability.

3. Async/Await: Async/await syntax builds upon Promises, offering a more synchronous-like feel for asynchronous code. You can use async before functions and await before promises to pause execution until the promise resolves. This simplifies asynchronous code and improves readability compared to traditional promise chaining.
4. Generators: Generators are functions that can be paused and resumed, yielding values at different points. They are particularly useful for creating iterators and handling complex asynchronous workflows in a more controlled manner. Generators can be combined with Promises or async/await for even more powerful asynchronous programming constructs.

37) Describe immutable data structures and their significance in functional programming. Provide an example illustrating the use of immutable data structures in JavaScript.

1. Immutable data structures ensure the state of your data remains constant, leading to predictable program behaviour and simplifying debugging.
2. Since they cannot be accidentally modified by multiple threads, they are inherently thread-safe, making them ideal for concurrent programming.
3. Functions that operate on immutable data always produce the same output for the same input, facilitating reasoning about program behaviour.
4. The immutability removes the complexity of tracking changes, allowing developers to focus on the logic of manipulating data through creating new data structures.

```
const person = {
    name: "Alice",
    age: 30,
};
const updatedPerson = {
    ...person,
    age: 31,
};
console.log(updatedPerson.age);
console.log(person.age);
```

36)Discuss the spread syntax in ES6. Provide an example demonstrating its usage to merge arrays or objects.

38)Explanation of Map as a collection of key-value pairs and Set as a collection of unique values.

39)Mention of methods for adding, removing, and iterating over elements in both Map and Set.

40)Explain the difference between Bootstrap Components and Bootstrap Advance Components. Provide examples of each type and discuss their respective use cases.

41)Discuss the significance of Bootstrap Utilities in web development. Provide examples of commonly used utilities and explain how they contribute to creating responsive and visually appealing websites.

42)Define and explain the concept of functional programming. Discuss the characteristics of pure functions, higher-order functions, currying, and immutable data structures, providing examples where applicable.