Full-Stack Development- 2 Exam Guide CT

1. Discuss the concept of reusable components in React.

A **reusable component** is the core principle of React. The idea is to break down the UI into small, independent, and isolated pieces (e.g., <Button>, <ProductCard>, <Navbar>).

- **Independence:** Each component manages its own logic, state, and appearance.
- **Reusability (DRY Principle):** Instead of writing the same HTML/CSS/JS multiple times (like for a product card), you create *one* <ProductCard /> component. You then reuse this component, passing in different **props** (like productName or price) to configure it for each use.
- **Composition:** You build complex UIs by composing simple components together, like building blocks.

2. Describe how to pass props from a parent to a child component.

Props (properties) are passed from a parent to a child using attributes in the JSX tag, just like HTML attributes. This follows React's **one-way data flow**.

1. **Parent Defines Data:** The parent component holds the data (e.g., in a state variable).
2. **Parent Passes Prop:** The parent renders the child component and adds a prop attribute: propName={dataToPass}.
3. **Child Receives Props:** The child component receives all props as a single object, which is the first argument of the function. This is commonly destructured.

Example:

JavaScript

```
// --- Child Component ---
// Receives props and destructures them
function WelcomeMessage({ username }) {
  return <p>Welcome, {username}!</p>;
}

// --- Parent Component ---
function Dashboard() {
  const [user] = useState('Arindam');
  // 'username' is the prop being passed to the child
  return <WelcomeMessage username={user} />;
}
```

3. Analyse how props are used to pass data in React components.

This is a slight variation of the previous question, focusing on the analysis.

Props are the mechanism for passing data **downward** from a parent component to a child component. They are the *only* way to configure and customize child components.

- **Read-Only:** Props are **read-only**. A child component must **never** modify its own this.props or props. This "one-way data flow" makes the application's data flow predictable and easier to debug.
- **Configuration:** Props are used to configure a component's appearance or behavior. For example, a <Button> component might accept a color="primary" prop, or a <UserProfile> component would accept a user={userData} prop.
- **Event Handling:** Parent components also pass down *functions* as props (e.g., onClick={handleClick}). This allows a child component (like a button) to communicate *up* to the parent by calling the prop function when an event occurs.

4. Clarify the difference between state and props in React.

This is a fundamental concept in React. Both are objects that hold data, but their purpose and control are different.

Feature	**Props (Properties)**	**State**
Data Flow	Passed **into** a component from its **parent**.	Managed **inside** the component itself.
Mutability	**Read-Only (Immutable).** A component *cannot* change its own props.	**Mutable.** A component *can* and *does* change its own state.
Control	Owned and controlled by the **parent** component.	Owned and controlled by the **component itself**.
How to Change	Can only be changed by the parent component.	Changed internally using this.setState() or a useState setter function.

Feature	**Props (Properties)**	**State**
Purpose	To **configure** a component (e.g., username="Arindam").	To **"remember"** and manage data that changes over time (e.g., user input).
Analogy	Function arguments.	Local variables inside a function.

5. Explain the significance of useState in functional components.

The useState hook is a function that allows **functional components to have and manage their own internal state**.

- **Significance:** Before Hooks, only class components could hold state. useState allows functional components (which are simpler) to be stateful, making them the modern standard for building components.
- **How it Works:** You call useState with an initial value. It returns an array with two elements:
    1. The **current state value**.
    2. A **setter function** to update that value.
- **Re-Renders:** When you call the setter function, React **re-renders** the component and its children, reflecting the new state in the UI.

Example:

JavaScript

```
import React, { useState } from 'react';


function Counter() {
  // Call useState. 'count' is the state, 'setCount' is the updater.
  const [count, setCount] = useState(0);


  return (
    <div>
      <p>You clicked {count} times</p>
      {/* Call the setter function to update state and trigger a re-render */}
      <button onClick={() => setCount(count + 1)}>
```

```
        Click me
      </button>
    </div>
  );
}
```

6. Describe the rules for using state in React.

There are three critical rules you must follow when working with state:

1. **Do Not Modify State Directly:**
    - **Wrong:** this.state.count = 1; or myArray.push(item);
    - **Reason:** React only re-renders when the *setter function* (this.setState or setCount) is called. Modifying state directly will not trigger a re-render, and your UI will be out of sync.
    - **Right:** this.setState({ count: 1 }); or setMyArray([...myArray, item]); (creating a new array).
2. **State Updates May Be Asynchronous:**
    - **Reason:** React may batch multiple state updates for performance. You cannot rely on state being updated immediately after calling the setter.
    - **Problem:** setCount(count + 1); setCount(count + 1); might only result in an increment of 1, as both calls see the same old count value.
    - **Right (Functional Update):** When new state depends on the old state, pass a function: setCount(prevCount => prevCount + 1);. This guarantees it uses the most recent state value.
3. **Only Call Hooks at the Top Level (for Functional Components):**
    - **Wrong:** Do *not* call useState or useEffect inside loops, if statements, or nested functions.
    - **Reason:** React relies on the *order* of Hook calls being identical on every render to work correctly.

7. Discuss the use of prop validation in React.

Prop validation is a development-time feature to catch bugs by ensuring components receive props of the correct data type. If a component gets a prop of the wrong type (e.g., a string instead of a number), React will show a warning in the browser console.

- **Purpose:** It helps enforce correct component usage, making code more robust and easier to debug, especially in large teams.
- **Implementation:** It is implemented using the prop-types library.

Example:

JavaScript

```
import React from 'react';
import PropTypes from 'prop-types'; // 1. Import

function UserProfile({ name, age }) {
  return <p>Name: {name}, Age: {age}</p>;
}

// 2. Define the expected prop types after the component
UserProfile.propTypes = {
  // 'name' must be a string and is required
  name: PropTypes.string.isRequired,

  // 'age' must be a number
  age: PropTypes.number,
};

// 3. (Optional) Set default values
UserProfile.defaultProps = {
  age: 18,
};
```

ReactJS: Event Handling & Forms

8. Describe how event handling works in React.

React's event handling system is a wrapper around the browser's native event system.

- **Synthetic Events:** React passes a **SyntheticEvent** object to event handlers. This is a cross-browser wrapper that ensures events work identically in all browsers (e.g., Chrome, Firefox).
- **camelCase Naming:** React event props are named using camelCase (e.g., onClick, onChange) instead of HTML's lowercase (onclick).
- **Passing Functions:** You pass a **function reference** (not a string) to the event handler prop.
  - **HTML:** <button onclick="handleClick()">
  - **React:** <button onClick={handleClick}>
- **preventDefault():** You must explicitly call event.preventDefault() to stop default browser behavior (like a form submitting). Returning false does not work.

9. Demonstrate the use of arrow functions to handle a button click event in React.

Arrow functions are used for event handlers in React to automatically bind the this context (in class components) or for concise inline functions.

Example (Class Component):

In class components, using an arrow function as a class property avoids the need to .bind(this) in the constructor, as it lexically inherits this from the class instance.

JavaScript

```
import React, { Component } from 'react';


class ClickButton extends Component {
  state = { message: 'Hello!' };


  // Arrow function as a class property
  // 'this' is automatically bound to the component instance.
  handleClick = () => {
    this.setState({ message: 'Button Clicked!' });
    console.log(this); // 'this' is the ClickButton component
  };
  render() {
    return (
      <div>
        <p>{this.state.message}</p>
        <button onClick={this.handleClick}>Click Me</button>
      </div>
    );
  }
}
```

Example (Functional Component):

In functional components, you define handlers as functions (often arrow functions for consistency). There is no this binding to worry about.

JavaScript

```
import React, { useState } from 'react';

const ClickButtonFunc = () => {
  const [message, setMessage] = useState('Hello!');

  const handleClick = () => {
    setMessage('Button Clicked!');
  };

  return (
    <div>
      <p>{message}</p>
      <button onClick={handleClick}>Click Me</button>
    </div>
  );
};
```

10. Discuss the role of arrow functions in handling events.

Arrow functions play a crucial role in event handling, primarily by solving the this binding problem in JavaScript classes (like React class components).

1. **Lexical this Binding:**
   - **The Problem:** A regular function (e.g., myMethod() {}) defines its own this based on *how it's called*. When passed as an event handler (e.g., onClick={this.handleClick}), its this becomes undefined.
   - **The Solution:** Arrow functions (() => {}) do not have their own this. They **inherit this** from their surrounding (enclosing) scope. In a React class, this is the component instance itself. This allows you to reliably call this.setState().
2. **Avoiding .bind():** Before arrow functions, the solution was to manually bind this in the constructor: this.handleClick = this.handleClick.bind(this);. Arrow functions as class properties (see Q9) make this binding automatic and cleaner.
3. **Inline Handlers:** They are also used for concise inline functions, especially when passing arguments to the handler.
   - **Example:** <button onClick={() => deleteItem(item.id)}>Delete</button>
   - **Note:** This creates a new function on every render, which *can* have performance implications in complex components, but is often acceptable.

11. Explain the concept of controlled components in React forms.

A **controlled component** is a React form element (like <input>, <textarea>) whose value is **controlled by React state**.

This pattern creates a "single source of truth" where the component's state, not the DOM, manages the data.

- **value Prop:** The input's value attribute is explicitly set from a state variable (e.g., <input value={this.state.name} />).
- **onChange Handler:** A function is passed to the onChange prop. This function fires on every keystroke.
- **Data Flow:**
  1. User types into the input.
  2. The onChange handler is triggered.

3. The handler updates the React **state** with the new value (e.g., this.setState({ name: event.target.value })).
4. React re-renders.
5. The input's value is now set to the new value from the state.

This loop allows React to validate input in real-time and always have access to the form's current value.

12. Implement a React form that captures user name and email.

This example uses useState and the **controlled component** pattern. It also uses a single state object to manage multiple fields.

JavaScript

```
import React, { useState } from 'react';


function UserForm() {
  // Use a single state object to hold all form data
  const [formData, setFormData] = useState({
    name: '',
    email: ''
  });


  // A single handler to manage changes for all inputs
  const handleChange = (event) => {
    const { name, value } = event.target; // Get name/value from input


    setFormData(prevFormData => ({
      ...prevFormData, // Copy all old key-value pairs
      [name]: value      // Overwrite the one key that changed
    }));
  };
```

```
const handleSubmit = (event) => {
  event.preventDefault(); // Prevent page reload
  alert(`Form submitted! Name: ${formData.name}, Email: ${formData.email}`);
};

return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>Name:</label>
      <input
        type="text"
        name="name" // 'name' attribute must match the state key
        value={formData.name} // Controlled by state
        onChange={handleChange}
      />
    </div>
    <div>
      <label>Email:</label>
      <input
        type="email"
        name="email" // 'name' attribute must match the state key
        value={formData.email} // Controlled by state
        onChange={handleChange}
      />
    </div>
    <button type="submit">Submit</button>
  </form>
);
```

```
}
```

13. Explain the significance of onChange in form elements.

The onChange event handler is the key that makes **controlled components** work.

- **What it is:** onChange is a prop attached to form elements (e.g., <input>).
- **What it does:** It fires a function *every single time the user changes the value*. For a text input, this means it fires on *every keystroke*.
- **Significance:** Its job is to capture the new value from the DOM (via event.target.value) and immediately use it to **update the component's state**. Without onChange, a controlled component's value would be locked to the state, and the user would be unable to type.

14. Discuss the purpose of the value attribute in form controls.

In React, the value attribute on a form control (like <input>) is what *creates* a **controlled component**.

- **Purpose:** Its purpose is to **lock the input's displayed value directly to a piece of React state**.
- **In HTML:** The value attribute only sets the *initial* value.
- **In React:** value={this.state.name} *overrides* the input's internal state. The input is no longer in charge of its own value; React state is. The input will *always* display whatever the state variable contains, creating a "single source of truth."

15. Illustrate how state updates enable real-time form validation.

Because controlled components update the React state on *every keystroke* (via onChange), you can run validation logic *inside* the onChange handler. This provides immediate, real-time feedback.

Example:

We store the input's value (password) and an error message in state. The onChange handler validates the new value before setting the state.

JavaScript

```
import React, { useState } from 'react';


function PasswordForm() {
  const [password, setPassword] = useState('');
  const [error, setError] = useState(''); // State for the error message

  const handlePasswordChange = (e) => {
    const newPassword = e.target.value;
    setPassword(newPassword); // Update password state

    // Perform validation logic in real-time
    if (newPassword.length > 0 && newPassword.length < 8) {
      setError('Password must be at least 8 characters long.');
    } else {
      setError(''); // Clear the error if valid
    }
  };

  return (
    <form>
      <label>Password:</label>
      <input
        type="password"
        value={password}
        onChange={handlePasswordChange}
        style={{ borderColor: error ? 'red' : 'green' }} // Dynamic style
      />
      {/* Display the error message from state */}
      {error && <p style={{ color: 'red' }}>{error}</p>}
    </form>
  );
}
```

16. Summarize how to manage multiple input fields in a form.

The most common and scalable strategy is to use a **single state object**.

1. **Single State:** Use *one* useState call to hold an object (e.g., formData).
2. **name Attribute:** Give each <input> a name attribute that **exactly matches** the corresponding key in your state object (e.g., name="username" matches formData.username).
3. **Generic Handler:** Create *one* handleChange function for all inputs.
4. **Update Logic:** Inside the handler, use event.target.name (a string) to know *which* input changed. Use computed property syntax ([name]: value) and the spread operator (...) to update the state object immutably.

Example Handler:

JavaScript

```
const [formData, setFormData] = useState({ username: '', email: '' });


const handleChange = (e) => {
  const { name, value } = e.target;
  setFormData(prevData => ({
    ...prevData, // Copy all old values
    [name]: value  // Overwrite the one key that changed
  }));
};
```

17. Describe how handleSubmit is implemented in React forms.

handleSubmit is the function that manages the form's **submission** event.

1. **Create Handler:** Define a function, handleSubmit(event).
2. **Attach to <form>:** Pass this function to the onSubmit prop of the <form> tag: <form onSubmit={handleSubmit}>.
3. **event.preventDefault():** This is the *most important step*. You **must** call event.preventDefault() as the first line in your function. This stops the browser's default behavior of reloading the page on form submission.

4. **Access Data:** Access all the form's data, which is already available in your component's **state** (because you are using controlled components).
5. **Perform Action:** Use the data from state to send it to an API (using fetch), log it, or pass it to a parent component.

18. Explain the purpose of data binding in React.

Data binding is the process of synchronizing data between the **Model** (your component's state) and the **View** (the rendered UI).

React uses **one-way data binding**.

1. **State-to-View:** Data flows from the component's **state** *down* to the **UI**. When the state is updated, React *automatically* re-renders the UI to reflect the new data.
   - **Example:** <p>{this.state.name}</p>
2. **"Two-Way Binding" (Simulated):** In forms, React *simulates* two-way binding using the controlled component pattern:
   - **State-to-View:** value={this.state.name}
   - **View-to-State:** onChange={this.handleChange}
   - This explicit, one-way flow makes data changes predictable and easier to debug.

ReactJS: SPAs, Routing & Data Fetching

19. Analyse how ReactJS is used for building modern single-page applications.

A Single-Page Application (SPA) is a web app that loads a single HTML page and dynamically updates its content. React is ideal for building SPAs.

1. **Component-Based Architecture:** React breaks the UI into components (e.g., Navbar, Sidebar).
2. **Virtual DOM (VDOM):** When data changes, React updates a lightweight copy of the DOM in memory (the VDOM). It then calculates the *minimal* changes needed for the real DOM. This "reconciliation" process is very fast and avoids full-page reloads.
3. **Client-Side Routing:** Libraries like **React Router** intercept navigation. Instead of asking the server for a new page, React Router just *swaps* the components being displayed on the client-side (e.g., replacing <HomePage> with <AboutPage>). It also updates the browser's URL, so the user experience is seamless.

4. **Declarative UI:** You tell React *what* the UI should look like for a given state, and React handles the *how* (DOM updates).

20. Examine how React Router enhances single-page applications.

React Router is the library that adds navigation to a React SPA.

1. **App-Like Experience:** It enables navigation between "pages" (components) **without a page reload**, making the app feel fast and responsive.
2. **Declarative Routing:** It provides components like <Routes>, <Route>, and <Link> that let you declaratively map a URL path to a specific component.
   - <Link to="/about"> replaces <a> tags and prevents browser reloads.
   - <Route path="/about" element={<AboutPage />} /> defines the mapping.
3. **Browser History:** It updates the browser's URL and history stack. This means users can still use the **back/forward buttons** and **bookmark** specific URLs (e.g., /products/123).
4. **Dynamic Routes:** It allows for "parameterized" routes (e.g., <Route path="/users/:userId" ... />). The component can then access the userId from the URL to fetch the correct data.

21. Describe how React fetches data from an API.

React is a UI library and has no built-in data fetching. Instead, you use standard JavaScript mechanisms:

1. **fetch() API:** The modern, promise-based API built into browsers.
2. **axios:** A popular third-party library that simplifies requests and error handling.

The fetching is done as a **side effect**, so the correct place to do it is inside the useEffect Hook.

Standard Process:

1. **Set up State:** Use useState to store the data, a loading status, and any error.
2. **Call useEffect:** Call useEffect with an empty dependency array [] to ensure it runs **only once** when the component mounts.
3. **Fetch Data:** Inside the useEffect, make your fetch() call.
4. **Update State:** Use .then() or async/await to handle the response.
   - On success, call setData(responseData) and setLoading(false).
   - On failure, call setError(message) and setLoading(false).

5. **Render UI:** Conditionally render your component based on the loading, error, and data states.

22. Illustrate how useEffect() supports API calls in React.

The useEffect Hook is designed to handle **side effects**, and an API call is a classic side effect. It provides a way to run the fetch logic *after* the component has rendered, preventing it from blocking the UI or causing infinite loops.

- **The Hook:** useEffect(() => { ... }, [dependencies])
- **The Dependency Array []:** By passing an **empty array** [], you tell React to run the effect **only one time**, after the component's first render (similar to componentDidMount). This is perfect for initial data fetching.

Example Code:

JavaScript

```
import React, { useState, useEffect } from 'react';


function UserProfile() {
  const [user, setUser] = useState(null);
  const [loading, setLoading] = useState(true);


  // 1. useEffect is called after the component first renders.
  useEffect(() => {
    // 2. The API call is made.
    fetch('https://jsonplaceholder.typicode.com/users/1')
      .then(response => response.json())
      .then(data => {
        // 3. State is updated with the fetched data.
        setUser(data);
        setLoading(false);
      });
  // 4. The empty array [] ensures this runs ONLY ONCE.
  }, []);

  // 5. The component renders based on the state.
  if (loading) {
    return <p>Loading...</p>;
  }

  return <h1>{user.name}</h1>;
}
```

ReactJS: Components & Misc

23. Discuss the role of App.jsx in React.

App.jsx (or App.js) is, by convention, the **root component** of a React application. It serves as the main container for all other components.

- **Top-Level Container:** It is the highest-level component in the component tree. The index.js file renders the <App /> component into the DOM.
- **Application Layout:** It often defines the primary layout of the app, such as including a persistent <Navbar>, <Footer>, and the main content area.
- **Routing:** It is the most common place to define the application's **client-side routing** using React Router. It holds the <Routes> component that defines all the application's "pages."
- **Global State:** It can be used to hold or provide global application state (e.g., user auth status, theme) via React Context.

24. How is CSS implementation in a React component?

There are four primary ways to style React components:

1. **Global Stylesheets:** A standard .css file (e.g., index.css) is imported into index.js or App.jsx. These styles are global and can lead to class name conflicts.
2. **Inline Styles:** CSS is written as a JavaScript object and passed to the style prop. Properties must be camelCased (e.g., backgroundColor). This is good for dynamic styles but is verbose.
   - **Example:** <div style={{ fontSize: '16px', color: 'blue' }}>

3. **CSS Modules (Recommended):** You name your file MyComponent.module.css. When you import it (import styles from './MyComponent.module.css'), React scopes all class names uniquely to that component. This prevents class name conflicts.
    - **Example:** <div className={styles.myClassName}>
4. **CSS-in-JS (e.g., Styled Components):** Libraries that let you write actual CSS syntax inside your component files, creating components that are bundled with their own styles.

25. Analyze the advantage of functional components in React.

Since the introduction of **Hooks** (like useState, useEffect) in React 16.8, functional components are the standard.

- **Simpler Syntax:** They are just JavaScript functions. They have less boilerplate (no constructor, super, render, or this).
- **No this Binding:** They completely eliminate the confusion of the this keyword, a common source of bugs in class components.
- **Stateful Logic with Hooks:** useState and useEffect provide a simpler, more direct way to handle state and lifecycle events than the complex class lifecycle methods (componentDidMount, etc.).
- **Reusable Logic (Custom Hooks):** This is the biggest advantage. Hooks allow you to **extract and reuse stateful logic** (e.g., useFetch, useWindowSize) into custom functions, which is much cleaner than old patterns like HOCs or Render Props.
- **Easier to Test:** As they are plain functions, they are often more straightforward to test.

26. Illustrate the use of code snippets in a React project.

In React, "code snippets" are encapsulated as **reusable components**. The goal is to follow the DRY (Don't Repeat Yourself) principle.

Illustration:

Imagine you need to display a user's avatar many times.

Bad (Repeating Code):

JavaScript

```
<div>
  <img src={user1.avatarUrl} alt={user1.name} className="avatar" />
  <span>{user1.name}</span>
```

```
</div>
<div>
  <img src={user2.avatarUrl} alt={user2.name} className="avatar" />
  <span>{user2.name}</span>
</div>
```

Good (Using a Reusable Component "Snippet"):

1. Create the Snippet (Avatar.jsx):

JavaScript

```
// src/components/Avatar.jsx
import React from 'react';

// This component is a reusable snippet.
// It is made dynamic by accepting 'user' as a prop.
function Avatar({ user }) {
  return (
    <div className="avatar-container">
      <img src={user.avatarUrl} alt={user.name} className="avatar" />
      <span>{user.name}</span>
    </div>
  );
}
export default Avatar;
```

2. Use the Snippet:

JavaScript

```
// src/pages/ProfilePage.jsx
import Avatar from '../components/Avatar';

function ProfilePage({ user1, user2 }) {
  return (
    <div>
      <Avatar user={user1} />
      <Avatar user={user2} />
    </div>
  );
}
```

Angular

27. How is a controller implemented in Angular?

In modern Angular, the "Controller" (from the MVC pattern) is implemented by the **Component class**.

The component is a TypeScript class, decorated with @Component, that is responsible for all the logic of the View.

- **Model:** The data properties *within* the component class (e.g., username: string = '';).
- **View:** The HTML template (.html file) associated with the component.
- **Controller:** The **TypeScript class** (.ts file) itself.

The Component class acts as the controller by:

1. **Defining Data (Model):** Holding application data in its properties.
2. **Handling Logic:** Containing the methods (e.g., onSubmit(), loadUser()) that respond to events from the View.
3. **Connecting View/Model:** Using Angular's data binding ([], (), [()]) to link the View to the Model.
4. **Injecting Services:** Using its constructor to inject services that handle business logic (like fetching data).

28. What is a model in Angular?

In Angular, the "Model" is not a single construct but a concept that refers to the **application's data and business logic**. It is primarily implemented in two places:

1. **Component Properties:** The data properties defined inside a component's class. This is the simplest form of the model, holding the state for that specific component's view.
   - **Example:** export class UserComponent { userName: string = 'Arindam'; }
2. **Services:** For data that needs to be shared across components or for handling business logic (like API calls), Angular uses **Services**. A service is a TypeScript class marked with @Injectable().
   - Services act as the "single source of truth" for application data.
   - They contain the methods to fetch, save, and manipulate data, keeping this logic out of the component (Controller).

29. Write simple code to show the work of model, view, and controller in Angular.

This example shows a simple user list.

1. The Model (Service)

This service manages the data and logic.

TypeScript

```
// src/app/user.service.ts (Model)
import { Injectable } from '@angular/core';

@Injectable({ providedIn: 'root' })
export class UserService {
  private users = ['Arindam', 'Jane', 'John'];

  getUsers(): string[] {
    return this.users;
  }
}
```

2. The Controller (Component Class)

This class gets data from the service and provides it to the view.

TypeScript

```
// src/app/user-list/user-list.component.ts (Controller)
import { Component, OnInit } from '@angular/core';
import { UserService } from '../user.service';

@Component({
  selector: 'app-user-list',
  templateUrl: './user-list.component.html'
})
export class UserListComponent implements OnInit {

  // This property is the component's local model, for the view
  public userList: string[] = [];

  // Injects the model (UserService)
  constructor(private userService: UserService) {}

  // Controller logic, runs on initialization
  ngOnInit() {
    // Calls the service to get data
    this.userList = this.userService.getUsers();
  }
}
```

3. The View (HTML Template)

This template displays the data from the controller.

HTML

```
<h2>User List</h2>
<ul>
  <li *ngFor="let user of userList">
```

```
    {{ user }}
  </li>
</ul>
```

Angular vs. React

30. Analyse the difference between Angular and React.

Feature	**Angular**	**React**
Type	**Framework** (Complete, "batteries-included")	**Library** (UI layer only)
Scope	**Opinionated.** It provides a full solution for routing, state management, HTTP requests, etc., and dictates *how* you build.	**Unopinionated.** It gives you flexibility. You must *choose* and integrate third-party libraries for routing, state, etc.
Language	**TypeScript** (Enforced)	**JavaScript (JSX)** (TypeScript is optional but common)
Data Binding	**Two-Way Data Binding** (using [(ngModel)]) and One-Way. Changes in the View can auto-update the Model.	**One-Way Data Flow.** State flows down via props. View changes update state via onChange handlers (controlled components).
DOM	Uses a **Real DOM** with change detection.	Uses a **Virtual DOM (VDOM)**. Calculates minimal DOM updates in memory, which is generally faster.
Architecture	**MVC/MVVM.** Enforces separation with Modules, Components (Controllers), and Services (Models).	**Component-Based.** Architecture is flexible and defined by how you compose components.
Learning Curve	**Steep.** Must learn TypeScript, Dependency Injection, Modules, RxJS, and the Angular "way" all at once.	**Moderate.** The core library is small. Difficulty increases as you learn the "ecosystem" (Redux, Router).

API & Data Communication

31. Describe how to create a simple API using PHP and MySQL.

A simple PHP "GET" API connects to a MySQL database, runs a query, and returns the result as JSON.

Example (getUsers.php):

PHP

```
<?php
// 1. Set Headers: Tell the client (React) we're sending JSON
header('Content-Type: application/json');
header('Access-Control-Allow-Origin: *'); // Allow cross-origin (for dev)

// 2. Database Connection
$servername = "localhost";
$username = "root";
$password = "";
$dbname = "my_database";

$conn = new mysqli($servername, $username, $password, $dbname);

if ($conn->connect_error) {
   echo json_encode(['error' => 'Connection failed']);
   die();
}

// 3. SQL Query
$sql = "SELECT id, name, email FROM users";
$result = $conn->query($sql);

$users = []; // Array to hold results
```

```
if ($result->num_rows > 0) {
    // 4. Fetch data into the array
    while($row = $result->fetch_assoc()) {
        $users[] = $row;
    }
}

// 5. Encode array as JSON and print it
echo json_encode($users);

$conn->close();
?>
```

32. Explain the function of JSON in API communication.

JSON (JavaScript Object Notation) is a lightweight, text-based data format. Its function is to be the **universal language** for data exchange between a client (like a React app) and a server (like a PHP/MySQL API).

- **Problem:** A server (e.g., PHP) cannot send a "PHP array" to a browser. A browser (JavaScript) cannot understand a PHP array.
- **Solution (JSON):**
    1. **Server-to-Client:** The PHP server **serializes** (converts) its data (e.g., a PHP array from a database query) into a **JSON string** (using json_encode()). This string is sent in the HTTP response.
    2. **Client-to-Server:** The React app receives this text string and **parses** (converts) it into a native JavaScript object or array (using response.json()). This data is now perfectly usable in JavaScript.

JSON is used because it is human-readable, lightweight (less verbose than XML), and every major programming language (JS, PHP, Python, Java) has built-in tools to read and write it.