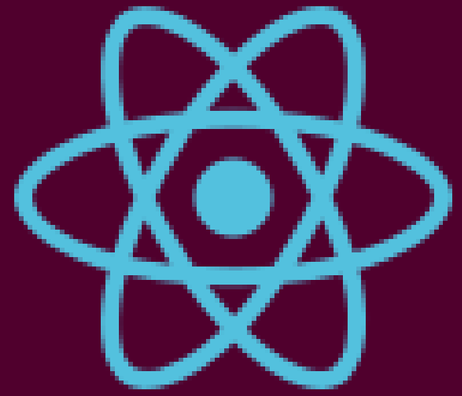
React.js basics



Feruza Makhmudova

Software Engineer (4+ years prof. experience)

- Mcs in Computer science and software engineering
- **email:** feruza.maxmudova@ventionteams.com

Agenda:

- Introduction to React
- JSX and Rendering Elements
- Components and Props
- State and Lifecycle
- Handling Events
- Conditional Rendering
- Lists and Keys
- Basic Routing with React Router

What is React?

What is React?

- **Declarative**: Instead of manipulating the DOM directly to update an element, React allows you to declare how the UI should look for a given state.
- **Component-Based**: In React, the UI is built using components, which are small, reusable, and self-contained pieces of code. Each component manages its own state and renders the UI accordingly.

Library and Framework:

- **Library**: React is a library, meaning it focuses on providing specific functionality (UI rendering) rather than enforcing a particular structure or set of rules for how you build your entire application.
- **Framework**: Frameworks like Angular provide a comprehensive solution, including routing, state management, and more, which dictate how your application is organized.

```
Traditional DOM Manipulation:
-----------------------------------
const button = document.querySelector('button');
button.addEventListener('click', () => {
  document.body.style.backgroundColor = 'blue';
});

React Approach:
-----------------------------------
function App() {
  const [color, setColor] = useState('white');

  return (
    <div style={{ backgroundColor: color }}>
      <button onClick={() => setColor('blue')}>Change Color</button>
    </div>
  );
}
```

Set up react

1. Install node.js
2. npx create-react-app ‘my-app’
3. npm start

Set up react

1. Install node.js
2. npm create vite@latest my-app -- --template react-ts

Folder structure

```
my-app/
├── public/         // Static assets like index.html, favicon, etc.
├── src/            // Application code, including components, CSS, etc.
├── node_modules/   // Installed dependencies
├── package.json    // Project configuration file
└── README.md       // Project documentation
```

```jsx
import React from 'react';

function App() {
  return (
    <div>
      <h1>Hello, World!</h1>
    </div>
  );
}

export default App;
```

Key concept

- Components
- JSX
- Virtual DOM

Function components

A simpler way to write components using JavaScript functions

Class components

Older way to write components using ES6 classes.

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

class Welcome extends Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

Props in React

- Props are the inputs to a React component.
- They allow you to pass data from a parent component to a child component.
- Props are read-only, meaning a component cannot change its own props; it can only use them.

State in React

- State is private and fully controlled by the component.
- You can use state to store data like user input, fetched data, or the current UI view.

state, useState

```
const [stateVariable, setStateFunction] = useState(initialState);
```

```
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
}
```

state, useState

- State is Asynchronous:
  - updates are not immediate
  - compute a new state based on the previous state

```
setCount(prevCount => prevCount + 1);
```

- State Should Not Be Modified Directly
  - this.state.count++

Feature	State	Props
Definition	Internal data controlled by the component	External data passed from parent to child
Mutability	Mutable (can be changed within the component)	Immutable (read-only)
Responsibility	Managed within the component	Managed by the parent component
Updates	Can trigger a re-render when updated	Cannot be updated directly by the component
Usage	For dynamic data that changes over time	For passing static or unchanging data

The component lifecycle

- **Mounting**: When the component is first created and inserted into the DOM.
- **Updating**: When the component's state or props change, causing it to re-render.
- **Unmounting**: When the component is removed from the DOM.

Mounting

- **constructor**(): Initializes the component’s state and binds methods.
- **componentDidMount**(): Called once the component is rendered. It’s often used to fetch data or start timers.
    -

Updating

- **componentDidUpdate():** Called after the component has been re-rendered due to state or prop changes. It’s often used to react to state changes or make further updates.

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    // React to state changes
  }
}
```

Unmounting

- **componentWillUnmount():** Called just before the component is removed from the DOM. It’s often used to clean up resources like timers or network requests.

- The useEffect hook allows you to perform side effects in function components.
- It runs after the component renders and can be used to simulate lifecycle methods like **componentDidMount**, **componentDidUpdate**, and **componentWillUnmount**.

```
useEffect(() => {
  // Effect code here
}, [dependencies]);
```

UseEffect

```
useEffect(() => {
  const timer = setInterval(() => {
    console.log('Timer running');
  }, 1000);

  return () => clearInterval(timer); // Cleanup when the component unmounts
}, []);
```

Handling Events

```jsx
function ClickHandler() {
  const handleClick = () => {
    alert("Button clicked!");
  };

  return <button onClick={handleClick}>Click Me</button>;
}

export default ClickHandler;
```

```jsx
function ArgumentHandler() {
  const handleClick = (name) => {
    alert(`Hello, ${name}`);
  };

  return <button onClick={() => handleClick("Alice")}>Greet</button>;
}

export default ArgumentHandler;
```

Handling Events

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

function FormExample() {
  const [inputValue, setInputValue] = useState("");

  const handleChange = (event) => {
    setInputValue(event.target.value);
  };

  return (
    <div>
      <input type="text" value={inputValue} onChange={handleChange} />
      <p>You typed: {inputValue}</p>
    </div>
  );
}

export default FormExample;
```

Conditional Rendering

```jsx
function Greeting({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn ? <h1>Welcome back!</h1> : <h1>Please sign in.</h1>}
    </div>
  );
}
```

Conditional Rendering

```
function WelcomeMessage({ isLoggedIn }) {
  return (
    <div>
      {isLoggedIn && <h1>Welcome back!</h1>}
    </div>
  );
}
```

Task: Conditional Rendering Based on State

Create a button, which on click toggles visible state of

Solution

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

function ToggleParagraph() {
  const [isVisible, setIsVisible] = useState(false);

  const toggleVisibility = () => {
    setIsVisible(prevState => !prevState); // Toggle the state
  };

  return (
    <div>
      <button onClick={toggleVisibility}>
        {isVisible ? 'Hide' : 'Show'} Paragraph
      </button>
      {isVisible && <p>This is a toggled paragraph.</p>}
    </div>
  );
}

export default ToggleParagraph;
```

List and keys

To render a list of items in React, you typically map over an array and return JSX for each item.

```
function TodoList({ todos }) {
  return (
    <ul>
      {todos.map((todo) => (
        <li key={todo.id}>{todo.text}</li>
      ))}
    </ul>
  );
}

export default TodoList;
```

Why to use keys

- Keys should be stable and unique among siblings, meaning each element in the list should have a distinct key.
- Keys help React optimize the re-rendering process by minimizing unnecessary updates.

Key Pest practises

- Always provide **unique keys** for each item in a list. A unique identifier, such as an id field, is preferable.
- Avoid using array indices as keys, as this can lead to issues with state management and performance, especially if the list is reordered or modified.
- Keys must be stable—meaning that the same key should be used for the same item between renders.
- Keys are used internally by React to optimize rendering, so they do not appear in the rendered HTML.

Key Features of React Router

- **Declarative routing:** Routes are defined in JSX, making the routing logic easier to manage and understand.
- **Dynamic route matching**: React Router can dynamically match URL patterns to display the appropriate components.
- **Browser history management**: React Router can manage the browser’s history API, enabling back and forward navigation.

```jsx
import { BrowserRouter, Routes, Route } from 'react-router-dom';
import Home from './components/Home';
import About from './components/About';
import Contact from './components/Contact';

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

```jsx
import { Link } from 'react-router-dom';

function Navbar() {
  return (
    <nav>
      <ul>
        <li>
          <Link to="/">Home</Link>
        </li>
        <li>
          <Link to="/about">About</Link>
        </li>
        <li>
          <Link to="/contact">Contact</Link>
        </li>
      </ul>
    </nav>
  );
}

export default Navbar;
```

Navigating Between Pages with Link

Route Parameters

```jsx
import { BrowserRouter, Routes, Route, useParams } from 'react-router-dom';

function UserProfile() {
  const { userId } = useParams(); // Extracts userId from the URL
  return <h1>User Profile for User ID: {userId}</h1>;
}

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/user/:userId" element={<UserProfile />} />
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

404 Pages (Not Found)

```jsx
function NotFound() {
  return <h1>404 - Page Not Found</h1>;
}

function App() {
  return (
    <BrowserRouter>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
        <Route path="*" element={<NotFound />} /> {/* Catch-all route */}
      </Routes>
    </BrowserRouter>
  );
}

export default App;
```

Navigation with useNavigate

```jsx
import { useNavigate } from 'react-router-dom';

function Login() {
  const navigate = useNavigate();

  const handleLogin = () => {
    // Perform login logic
    navigate('/dashboard'); // Navigate to the dashboard after login
  };

  return (
    <div>
      <button onClick={handleLogin}>Login</button>
    </div>
  );
}

export default Login;
```

```jsx
import { createBrowserRouter, RouterProvider } from 'react-router-dom';
import Home from './pages/Home';
import About from './pages/About';
import Contact from './pages/Contact';

// Define routes as plain objects
const routes = [
  {
    path: '/',
    element: <Home />,
  },
  {
    path: '/about',
    element: <About />,
    loader: async () => {
      const response = await fetch('/api/about-data');
      return response.json();
    },
  },
  {
    path: '/contact',
    element: <Contact />,
    action: async ({ request }) => {
      const formData = await request.formData();
      const name = formData.get('name');
      await fetch('/api/contact', {
        method: 'POST',
        body: JSON.stringify({ name }),
      });
      return { success: true };
    },
  },
];

// Create a router from the route objects
const router = createBrowserRouter(routes);

function App() {
  return <RouterProvider router={router} />;
}

export default App;
```

Create Plain Object Routes

What is an Action?

```jsx
import { Form } from 'react-router-dom';

function Contact() {
  return (
    <div>
      <h1>Contact Us</h1>
      <Form method="post">
        <label>
          Name:
          <input type="text" name="name" required />
        </label>
        <button type="submit">Submit</button>
      </Form>
    </div>
  );
}

export default Contact;
```

```jsx
export async function contactAction({ request }) {
  // Parse form data
  const formData = await request.formData();
  const name = formData.get('name');

  // Simulate sending data to a backend API
  const response = await fetch('/api/contact', {
    method: 'POST',
    body: JSON.stringify({ name }),
    headers: { 'Content-Type': 'application/json' },
  });

  if (!response.ok) {
    throw new Error('Failed to submit contact form');
  }

  // You can also return data or redirect
  return { success: true };
}
```

```jsx
import Contact from './pages/Contact';
import { contactAction } from './actions/contactAction';

const routes = [
  {
    path: '/contact',
    element: <Contact />,
    action: contactAction, // Link the action to the route
  },
];
```

Any questions?

Thank you very much!