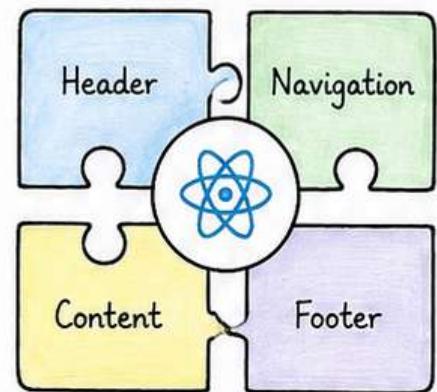


Chapter 2: React Components

2.1 Components Introduction

- * A component is a **reusable piece of UI** in a **React** application.
- * It allows us to break down the UI into **independent, reusable** and **manageable** pieces.
- * Components make the code **clean, organized** and **easy to maintain**.
- * Every React application is built using **components**.



Building UI like puzzle pieces

Key Points :

- ✓ Components are the building blocks of React applications.
- ✓ They promote reusability and modularity.
- ✓ Components can be small (like a button) or big (like a page layout).

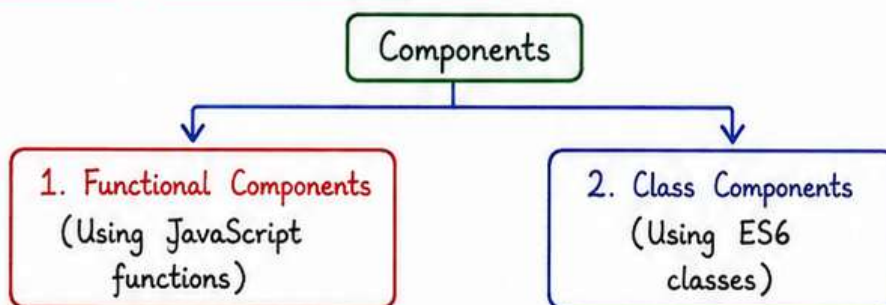


2.1.1 Why Use Components?

- * **Reusability** : Write once, use many times.
- * **Maintainability** : Easy to update and manage.
- * **Readability** : Code is well-structured and easy to understand.
- * **Scalability** : Applications can grow by combining many components.



2.1.2 Types of Components



Note :

Functional components are now more popular because they are simpler and easier to use with **Hooks**.

Key Takeaway :



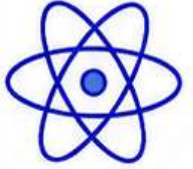
Components are the foundation of React. They help us build **reusable, maintainable** and **scalable** user interfaces.



2.2 Functional Components

- * Functional components are **JavaScript functions**.
- * They accept **props** as input and return **JSX**.
- * They are **simpler** and **easier to write**.
- * React **Hooks** can be used in functional components to add state and other features.
- * Most React components today are **functional components**.

```
function Welcome(props) {  
  return (  
    <div>  
      <h2>Hello, {props.name}</h2>  
      <p>Welcome to React!</p>  
    </div>  
  );  
}  
  
export default Welcome;
```



Key Points :

- ✓ Functional components are plain JavaScript functions.
- ✓ They return JSX to describe the UI.
- ✓ Hooks make functional components powerful.



2.3 Component Structure

- * A React component has a **simple structure**.

```
function ComponentName(props) {  
  // logic (optional)  
  return (  
    <div>  
      {/* JSX content */}  
    </div>  
  );  
}  
  
export default ComponentName;
```

Function name should start with a capital letter.

Component logic can be written here.

Return statement must return JSX.

JSX describes what should be shown on screen.

Export the component to use it in other files.



Key Points :

- ✓ Component name must start with a capital letter.
- ✓ It must return a single parent element.
- ✓ Use export default to make the component reusable.



Key Takeaway :



Functional components are **simple JavaScript functions** that return **JSX**.
They are easy to write, understand and reuse.

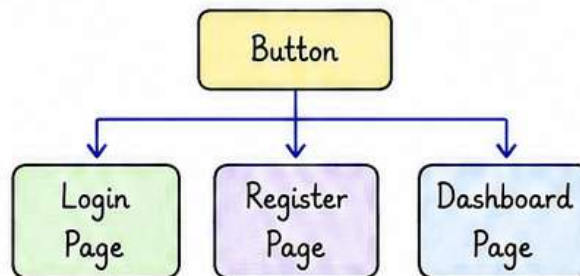


2.4 Component Reusability

- * One of the biggest advantages of components is **reusability**.
- * Once a component is created, it can be used anywhere in the application **any number of times**.
- * This saves time, reduces **code duplication** and makes maintenance easier.
- * Reusable components make the application more **consistent**.

```
function Button(props) {  
  return (  
    <button className="btn">{props.text}</button>  
  );  
}  
  
export default Button;
```

Reusing Button component



Same component used in multiple places.

Key Points :

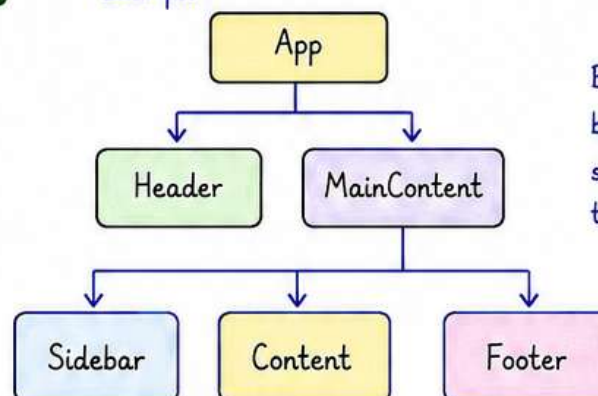
- ☒ Write a component once and reuse it multiple times.
- ☒ Reusability reduces code duplication.
- ☒ It ensures consistency across the application.



2.5 Component Composition

- * Components can be **combined together** to build complex user interfaces.
- * A component can use **other components** inside its return statement.
- * This process of combining components is called **component composition**.
- * It helps in building big applications from small and **reusable pieces**.

Example :



Big UI is built by composing small components together.

Key Points :

- ☒ Components can use other components.
- ☒ Composition helps to build complex UIs easily.
- ☒ It improves organization and readability of code.



Key Takeaway :

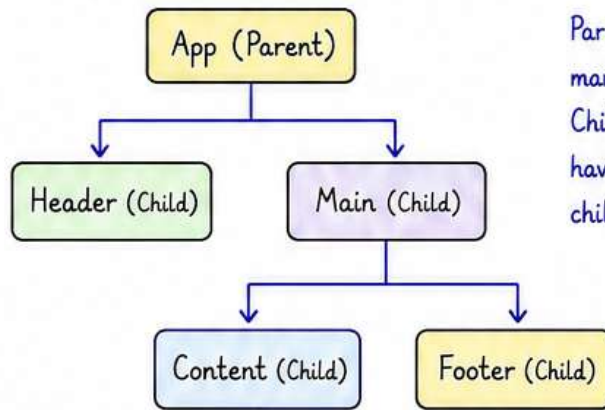
- ☆ Reusability and composition are the core principles of React.
- ☆ They help us build **scalable, maintainable** and **efficient** applications.



2.6 Parent and Child Components

- * Components in React are organized in a **tree structure**.
- * The component that contains another component is called the **parent component**.
- * The component that is inside another component is called the **child component**.
- * Data can be passed from parent to child using **props**.

Example :



Parent can have many children. Child can also have its own children.

Key Points :

- ✓ The topmost component is the root of the component tree.
- ✓ Parent components can contain one or more child components.
- ✓ Child components depend on their parent.
- ✓ Props flow from parent to child.



Key Takeaway :

-  React applications are built using a **component tree** where data flows from **parent to child**.



2.7 Props

- * Props (short for **properties**) are used to pass data from one component to another.
- * Props are **read-only**.
- * They are passed from **parent** to **child**.
- * Props help make components **dynamic** and **reusable**.

Example :

Parent Component

```
function App() {  
  return (  
    <div>  
      <Greeting name="Bhavitha" />  
    </div>  
  );  
}  
export default App;
```

Child Component

```
function Greeting(props) {  
  return (  
    <h2>Hello, {props.name}!</h2>  
  );  
}  
export default Greeting;
```

The parent **App** sends the value "Bhavitha" to the child **Greeting** through **props**. The child uses it to display the message.



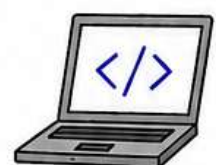
Key Points :

- ✓ Props are used to pass data from parent to child.
- ✓ Props are immutable (cannot be changed by the child).
- ✓ They make components configurable and reusable.



Key Takeaway :

-  Props are the main way to pass data between components in React. They flow from **parent to child** and help in building **dynamic UIs**.



2.8 Passing Data Through Props

- * Props are used to pass data from a **parent** component to its **child** component.
- * The parent sends the data as **attributes** when it uses the child component.
- * The child receives the data and uses it inside its **JSX**.
- * Data flows in **one direction** only (**parent to child**).

Example :

Parent Component (App.js)

```
function App() {  
  const name = "Bhavitha";  
  const age = 22;  
  return (  
    <div>  
      <User name={name} age={age} />  
    </div>  
  );  
}  
export default App;
```

Child Component (User.js)

```
function User(props) {  
  return (  
    <div>  
      <h2>{props.name}</h2>  
      <p>Age: {props.age}</p>  
    </div>  
  );  
}  
export default User;
```

The parent **App** component sends **name** and **age** to the child **User** component using **props**.



Key Points :

- ✓ Data is passed from parent to child using props.
- ✓ Props are written like HTML attributes.
- ✓ The child component receives props as an object.
- ✓ Props are read-only.

Key Takeaway :



Props are the **bridge** that allows components to share data.
They help in building **dynamic** and **interactive** React applications.



2.9 Props Destructuring

- * Instead of using **props.name**, **props.age** repeatedly, we can **destructure** props.
- * Destructuring makes the code **shorter** and **easier to read**.
- * We extract only the **needed** properties from props.

Example :

Without Destructuring

```
function User(props) {  
  return (  
    <div>  
      <h2>{props.name}</h2>  
      <p>Age: {props.age}</p>  
      <p>City: {props.city}</p>  
    </div>  
  );  
}
```

With Destructuring

```
function User({ name, age, city }) {  
  return (  
    <div>  
      <h2>{name}</h2>  
      <p>Age: {age}</p>  
      <p>City: {city}</p>  
    </div>  
  );  
}
```

We extract **name**, **age** and **city** from **props** directly inside the function parameters.

Key Points :

- ✓ Destructuring helps access props easily.
- ✓ It reduces repetition and makes code clean.
- ✓ We can destructure any number of props.



Key Takeaway :



Props destructuring is a simple technique that improves code **readability** and helps in writing **clean** and **maintainable** React components.



2.10 Children Props

- * The special prop **children** is used to pass content (JSX) between the opening and closing tags of a component.
- * It allows components to wrap and display other components or elements.
- * Children props make components more **flexible** and **reusable**.
- * **children** is **read-only** and cannot be modified.

Example :

Parent Component (App.js)

```
function App() {  
  return (  
    <div>  
      <Card>  
        <h2>Welcome to React</h2>  
        <p>This content is passed  
          as children.</p>  
      </Card>  
    </div>  
  );  
}  
export default App;
```

Child Component (Card.js)

```
function Card(props) {  
  return (  
    <div className="card">  
      {props.children}  
    </div>  
  );  
}  
export default Card;
```

The content inside **<Card>...</Card>** in App is received by Card component through **props.children**.



Key Points :

- ✓ children props allow us to pass JSX content inside a component.
- ✓ It is useful for creating wrapper components like Card, Panel, Layout, etc.
- ✓ children makes components more dynamic and reusable.
- ✓ We use {props.children} to render it in the child component.



Key Takeaway :

- ☆ children props let us pass and display any content between component tags, making components **flexible** and **easy to reuse**.



2.11 Conditional Rendering

Examples :

1. Using if statement

```
function Greeting({ isLogin }) {  
  if (isLogin) {  
    return <h2>Welcome Back!</h2>;  
  } else {  
    return <h2>Please Login.</h2>;  
  }  
}
```

2. Using Ternary Operator

```
function Greeting({ isLogin }) {  
  return (  
    <div>  
      {isLogin ? <h2>Welcome Back!</h2>  
        : <h2>Please Login.</h2>}  
    </div>  
  );  
}
```

3. Using Logical AND (&&)

```
function Notification({ message }) {  
  return (  
    <div>  
      {message && <p className="msg">{message}</p>}  
    </div>  
  );  
}
```



- * Conditional rendering means showing **different UI** based on a **condition**.
- * It helps to display components or elements only when certain conditions are **true**.
- * We use **JavaScript conditions** inside **JSX**.
- * Common methods: **if**, **ternary operator** (**? :**), **logical AND** (**&&**).

Key Points :

- ✓ Conditional rendering helps create dynamic and interactive UIs.
- ✓ We can show or hide elements based on conditions.
- ✓ Ternary operator and && are widely used for conditional rendering.



Key Takeaway :

- ☆ Conditional rendering allows us to show the **right content** to users based on different conditions, improving **user experience**.



2.12 Rendering Lists

* In React, we can render lists of data using the `map()` function.

* The `map()` function loops through an array and returns a new array of elements.

* Each item in the list must have a unique **key**.

* Lists make it easy to display **repeated data** on the UI.

Example :

Data (App.js)

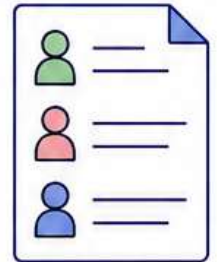
```
const students = [
  { id: 1, name: "Bhavitha" },
  { id: 2, name: "Reddy" },
  { id: 3, name: "Teja" }
];

function App() {
  return (
    <div>
      <h2>Student List</h2>
      <ul>
        {students.map(student => (
          <li key={student.id}>{student.name}</li>
        ))}
      </ul>
    </div>
  );
}
```

Output (On Screen)

Student List

- Bhavitha
- Reddy
- Teja



Key Points :

- ✓ Use `map()` to loop through arrays and render lists.
- ✓ Each item in a list must have a unique key.
- ✓ Lists help in displaying dynamic and repeated data.
- ✓ The key helps React identify which items have changed.



Key Takeaway :



Rendering lists in React is simple using `map()`, and it helps to display data in an **organized and dynamic** way.



2.13 Keys

* **Keys** are special attributes used when rendering lists.

* Keys help React **identify** which items have **changed**, been **added**, or **removed**.

* Keys must be **unique** among siblings.

* Using **index** as key is **not recommended** if the list can change order.

Example :

Using ID as Key (Recommended)

```
const items = [
  { id: 101, name: "Book" },
  { id: 102, name: "Pen" },
  { id: 103, name: "Bag" }
];

function ItemList() {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

Using Index as Key (Not Recommended)

```
function ItemList() {
  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item.name}</li>
      ))}
    </ul>
  );
}
```

Key Points :

- ✓ Keys should be unique among siblings.
- ✓ Use stable IDs from data as keys.
- ✓ Avoid using index as key if the list can change.
- ✓ Keys improve performance and help React update the UI efficiently.



Key Takeaway :

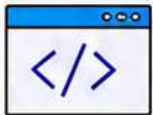


Always use **unique and stable keys** when rendering lists. It helps React **track elements** and **update the UI correctly**.



2.14 Component Best Practices

- * Following best practices helps us write **clean**, **efficient** and **maintainable** components.
- * Good practices improve code quality and make collaboration easier.
- * It also helps in building **scalable** and bug-free applications.
- * Always keep components **simple**, **focused** and **reusable**.



Best Practice	Description
1. Single Responsibility	A component should do one thing and do it well.
2. Reusability	Create reusable components and avoid code duplication.
3. Descriptive Names	Use meaningful and descriptive names for components and props.
4. Keep It Small	Keep components small and focused on a single task.
5. Use Props Properly	Pass only necessary data through props and avoid deeply nested props.
6. Avoid Inline Styles	Use CSS classes or styled-components instead of inline styles.
7. Organize Files	Keep components in separate files and use a logical folder structure.
8. Use Key Properly	Always use unique and stable keys while rendering lists.
9. Avoid Unnecessary Re-renders	Use React.memo, useMemo and useCallback to optimize performance.
10. Write Clean Code	Write clean, readable and well-commented code.

Key Points :

- ✓ Follow best practices to write **clean** and **maintainable** components.
- ✓ Keep components small, **reusable** and focused.
- ✓ Use **meaningful** names for components and props.
- ✓ Use props and keys properly.
- ✓ Optimize performance by avoiding unnecessary re-renders.



Key Takeaway :



Following best practices helps us build **high-quality** React applications that are easy to **maintain**, **scale** and **reuse**.



Summary Flow:

