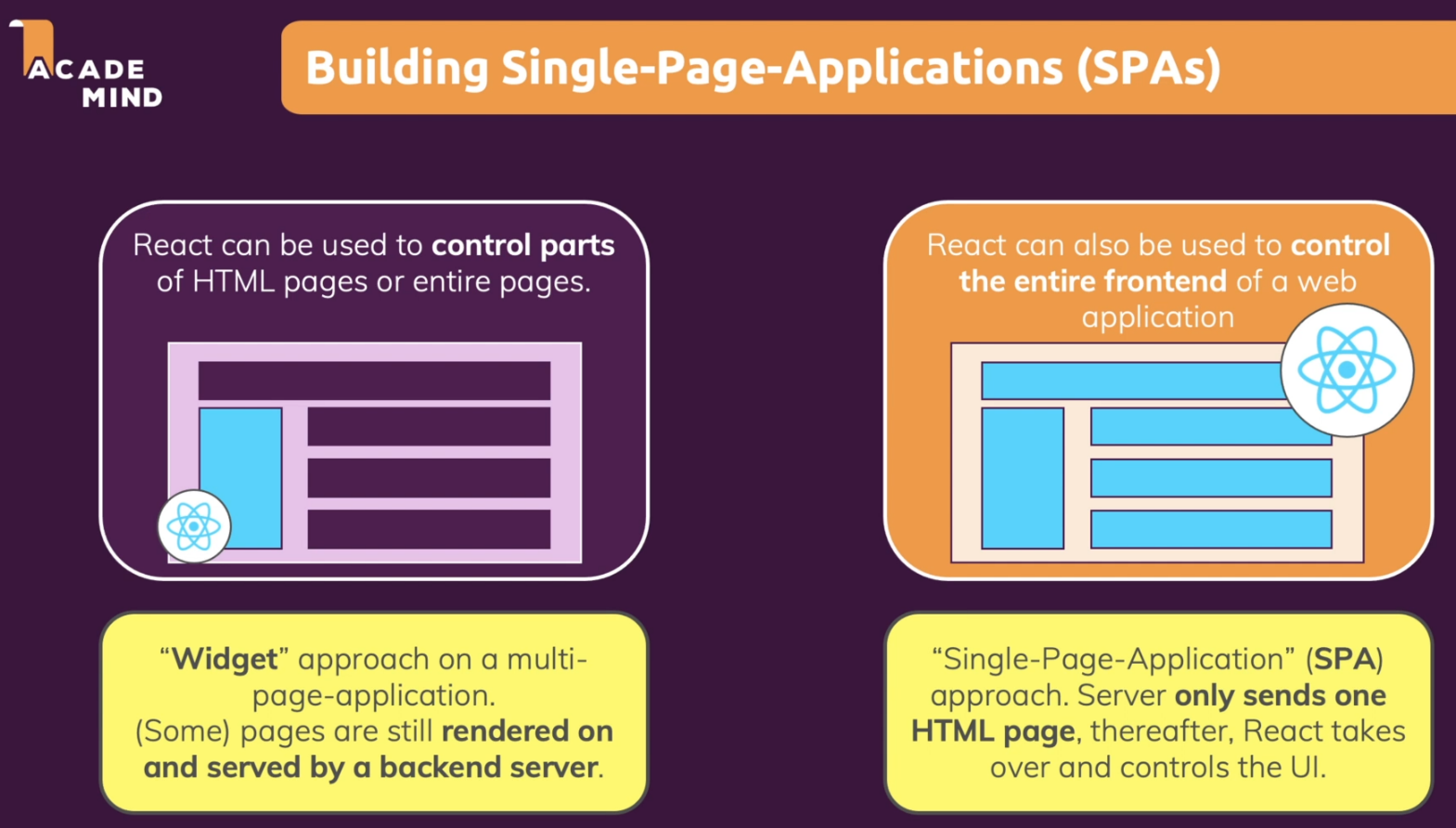
**React Notes**

React

* React is a client – side JavaScript library for building modern, reactive user interface for the Web using declarative, component – focused approach.
* JavaScript is a programming language that allows a developer to run logic in the browser.
* JavaScript runs in the browser – on the loaded page. We can manipulate the HTML Structure (DOM) of the page.
* When working with React, we often build so-called Single Page Applications.



Topics

1. React Basics & Working with Components
   1. Components
   2. Declarative & Imperative ways
   3. React project creation
   4. Intro to JSX (JavaScript XML)
   5. Custom Component
   6. Props concept
   7. Splitting Components into Multiple Components
   8. Composition concept
   9. Alternative Function syntax (Arrow function)
2. React State & Working with Events
   1. Listening to events & working with events Handlers
   2. Component function Execution
   3. Working with “State” & “useState” Hook
   4. Ways to update state
   5. Adding Form inputs, Listening to User Input & Handling form submission
   6. Working with Multiple states Vs Using One state
   7. Updating state that depends on the previous state
   8. Adding 2-way binding
   9. Child to Parent Component Communication (Bottom-up), Lifting the State Up
   10. Controlled Vs Uncontrolled Components, Stateless Vs Stateful Components
3. Rendering Lists & Conditional Content
   1. Rendering List of Data
   2. Using Stateful Lists
   3. Understanding “Keys”
   4. Outputting conditional content
   5. Adding conditional return statements
   6. Adding Dynamic styles

**1. React Basics & Working with Components**

**1.1. Components**

* React applications are built from isolated pieces of UI called **Components**.
* A component is **a piece of the UI (User interface) that has its own logic & appearance**.
* A component can be as small as a button, or as large as an entire page.
* React components are JavaScript functions that return markup (HTML or JSX)
* React components are regular JavaScript functions except:

1. Their names always begin with a capital letter.
2. They return JSX markup.

|  |
| --- |
| function MyButton() {    return (<button>I'm a button</button>);  }  export default function MyApp() {    return (      <div>        <h1>Welcome to my app</h1>        <MyButton />      </div>    );  } |

* Why Components?
* **Reusability** (Having reusable building blocks helps us avoid repetition)
* **Separation of Concerns** (Having a separation of concerns helps us with keeping our code base small & manageable instead of having one large file which holds all the HTML code & all the JavaScript logic.
* Recommended Approach
* For the entire user interface, we should create small separated units or components & each component should have one clear concern, one focus, one specific task that it should focus on.

**Steps for defining a component**

**Step 1:** Export the component

* The **export default** prefix is a standard JavaScript syntax.
* It lets us mark the main function in a file so that we can later import it from other files.

**Step 2:** Define the function

* React components are regular JavaScript functions, but their names must start with a Capital letter or they won’t work.
* We can define component using “function” keyword or using “arrow” function.

**Step 3:** Add markup

* The components return the JSX i.e., embedded XML inside JavaScript.
* We must wrap the JSX code in a pair of parentheses.

**Nesting & organizing components**

* Components are regular JavaScript functions, so we can keep multiple components in the same file.
* Only the main component in a file is declared with export default. Others must be declared as normal functions.
* We can export our function component from the source file to destination using either **default export** or **named exports.**

|  |
| --- |
| import Gallery from './Gallery.js'; // default export (valid for main component)  import { Profile } from './Gallery.js'; // Named export (valid for other than main component) |

* This is convenient when components are relatively small or tightly related to each other.
* We can create separate components to promote loose coupling & then import those components based on requirements.
  1. **Declarative & Imperative ways**
* React allows you to create re – usable & reactive components consisting of HTML & JavaScript (and CSS)
* React uses something called a declarative approach for building these components.
* In **Imperative programming approach**, we need to write the exact step by step instructions to manipulate the UI depending on what just happened i.e., we have to “command” each element telling the computer how to update the UI.
* In **Declarative programming approach**, we describe the UI for each visual state rather than micromanaging the UI (imperative).
* **React follows declarative approach i.e., define the desired target state(s) & let React figure out the actual JavaScript DOM instructions.**
* Declarative approach is basically mean that with React, we will not tell React that a certain HTML element should be created & inserted in a specific place on the UI as we would be doing that with Vanilla JavaScript.
* Instead with React, we will always define the desired end state (the target state) or conditional target state & it’s then React’s job to figure out which element on the actual webpage might need to be added or removed or updated & we don’t write these concrete DOM updating instructions on our own as we would be doing that with Vanilla JS.
* Manipulating the UI imperatively works well enough for isolated examples, but it gets exponentially more difficult to manage in more complex systems. Adding a new UI element or a new interaction would require carefully checking all existing code to make sure we haven’t introduced a bug.
* React was built to solve this problem. In React, we don’t directly manipulate the UI – meaning we don’t enable, disable, show, or hide components directly. Instead, we declare what we want to show, & React figures out how to update the UI.
* **When developing a component:**

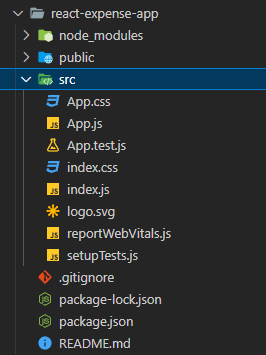
1. Identify all its visual states.
2. Determine the human & computer triggers for state changes.
3. Model the state with useState
4. Removal non – essential state to avoid bugs & paradoxes.
5. Connect the event handlers to set state
   1. **React project creation**

* Reference to get started with React: <https://github.com/facebook/create-react-app#creating-an-app>
* **Steps**

1. Download latest Node.js version: <https://nodejs.org/en> (Node JS is a technology that is not directly related to React, Node js is a runtime for JavaScript that allows us to run JS code outside the browser but React code is a JS code that runs on the browser.
2. To run following commands, we need Node JS runtime as pre-requisite

|  |
| --- |
| **// Way for creating new project**  npx create-react-app my-app  cd my-app  npm start  or  **// Way to import already created project**  npm install  npm start |

* **Newly created Project Structure**

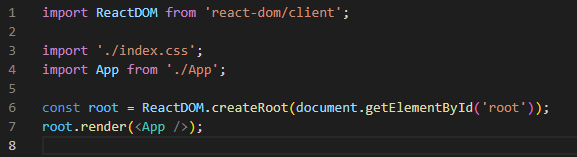


**React Project Structure Details**

**1. index.js**

* index.js file is the first code file which will be executed (<http://localhost:3000/>). It’s not exactly index.js file code but a transformed version of that code will be executed on the browser.
* Behind the scene, this **npm start** process not only start the development server & watch our code but also then takes the code, does the required transformation & deliver it to the browser.
* If we try to run index.js code directly, it won’t run in the browser (Invalid syntax)

**index.js**



**Step 1:** Here first we’re importing **ReactDOM** object from the react-dom 3rd party library which is one of our dependencies present in **package. json** which is downloaded & installed locally. Here we can see 2 React dependencies with different responsibilities i.e., react & react-dom (we can think of both together as React library)

**Step 2:** We call **createRoot ()** method on **ReactDOM** object. This creates the main entry point, the main hook of the overall UI we’re about to build with React.

**ReactDOM.createRoot ()** tells React where this React application (UI we built) should be placed in the web page that will be loaded. This leads us to one other file i.e., index.html.

**package. json**



**2. index.html**

* **index.html** is the single HTML file which is loaded by the browser in the end. This is basically the only HTML file that is being used by this overall React application because it’s a so-called **single page application (SPA)** & all subsequent changes on the webpage will be handled by React.

**index.html** is the single HTML file or the entry point where the React-driven UI should be rendered in. We can see HTML document body structure (<div id=”root”></div>) i.e., div with id as root that doesn’t hold any content but that’s the div where we want to inject or render our React-driven UI.

Following code will be used for this purpose

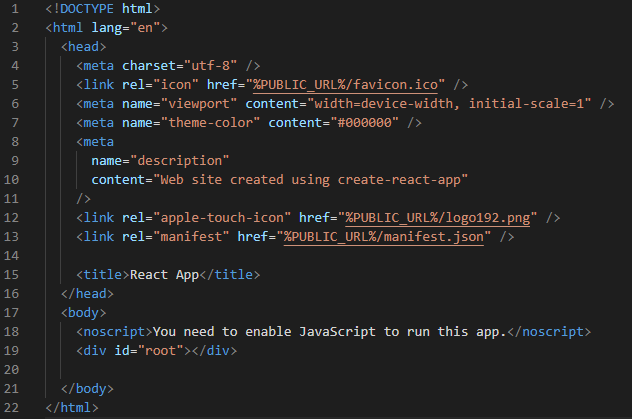




**root.render(<App />)** tells the React to render that div with id as root with the <App /> component present in App.js file.

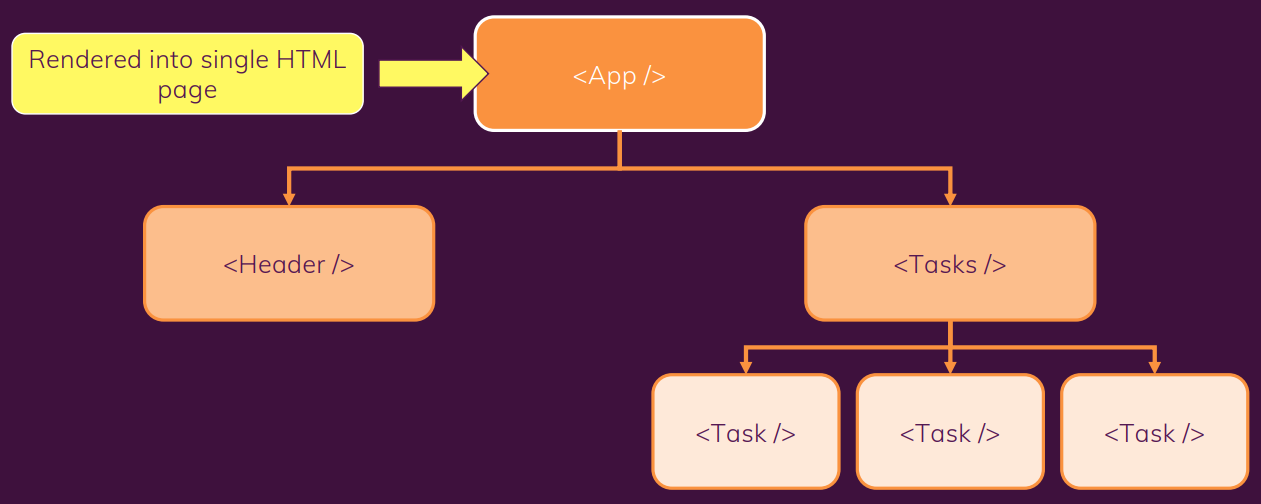
<App /> isn’t a regular JavaScript syntax, it’s **JSX syntax (JavaScript XML)**

**index.html** present in public folder



**3. App.js**

* This app component will be a special component called Root Component i.e., it’s the main component that will be render in our starting file index.js & all other components will be either nested inside App.js or nested inside of other components because ultimately with React, we build a Component Tree.



* 1. **Intro to JSX (JavaScript XML)**
* JSX is a special syntax introduced by the React team, & it works because of the transformation steps which are running behind the scenes.
* JSX is basically HTML code inside of JavaScript. JSX stands for JavaScript XML because HTML in the end is XML.
* We can see the transformed code by inspecting page source code in browser. It will contain entire react code along with our code.
* In order to output dynamic data, we put variables/constants/expressions inside curly braces {}
* JSX is stricter than HTML. We have to close tags like <br />. Our Component also can’t return multiple JSX tags. We have to wrap them into a shared parent like a <div>….</div> or empty <>….</> wrapper.

**Adding styles**

* In React, we specify a CSS class with “className”.

|  |
| --- |
| <img className="avatar" />  /\* In your CSS \*/  .avatar {  border-radius: 50%;  } |

**Displaying data (JSX with Curly Braces)**

* JSX lets us put markup into JavaScript. Curly braces {} let us “escape back” into JavaScript so that we can embed some variable from our code & display it to the user. For e.g.,

|  |
| --- |
| return (  <h1>  {user.name}  </h1>  ); |

* Curly braces let us bring JavaScript logic & variables into our markup.
* {{ and }} is not special syntax; it’s a JavaScript object tucked inside JSX curly braces.

**Conditional rendering**

* There are 3 ways for conditional rendering

1. **Using if / else statement** to conditionally include JSX
2. **Using ternary operator** {condition ? <A /> : <B />} mean if condition, render <A /> otherwise <B />.
3. **Using logical && syntax** {condition && <A /> } means if condition, render <A /> otherwise nothing.

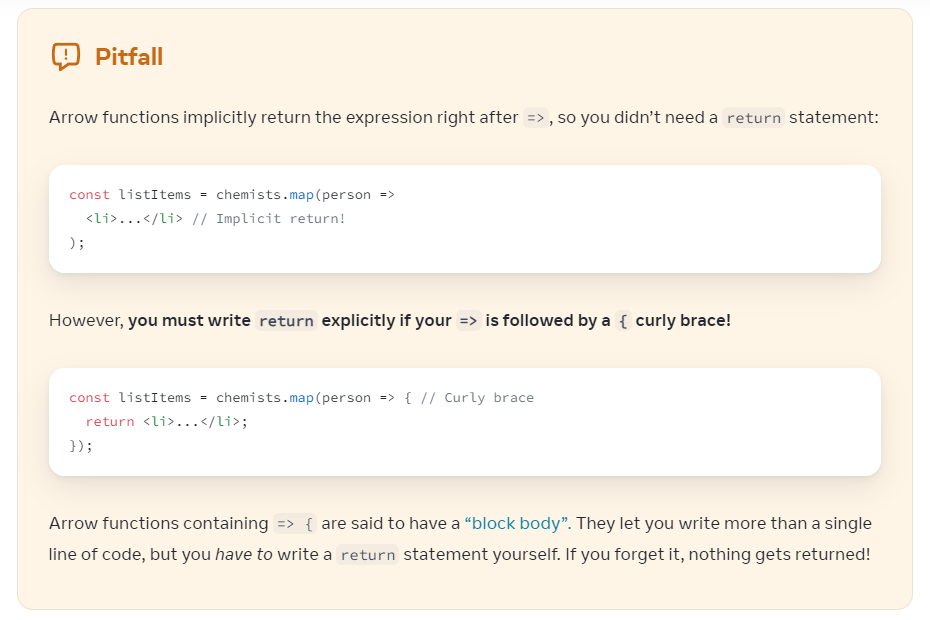
* We can put default statement in “let” variable & then update “let” variable based on the condition. Then embed the variable with curly braces in the returned JSX tree.
* In some situations, a component returns null if we don’t want to render anything at all. In practice, returning null from a component isn’t recommended because it might surprise a developer trying to render it. More often, we would conditionally include or exclude the component in the parent component’s JSX.

|  |  |  |
| --- | --- | --- |
| **Using if statement** | **Using ternay operator** | **Using logical && syntax** |
| let content;  if (isLoggedIn) {  content = <AdminPanel />;  } else {  content = <LoginForm />;  }  return (  <div>  {content}  </div>  ); | <div>  {isLoggedIn ? (  <AdminPanel />  ) : (  <LoginForm />  )}  </div> | <div>  {isLoggedIn && <AdminPanel />}  </div> |

**Rendering Lists**

* We will often want to display multiple similar components from a collection of data.
* In some case we need to show several instances of the same component using different data when building interfaces like from lists of comments to galleries of profile images. In these situations, we can store that data in JS objects & arrays and use methods like map() & filter () to render lists of components from them.
* We can use the JavaScript array methods to manipulate an array of data.
* Some commonly used array methods
* filter(), map(), forEach(), indexOf(), reduce(), reverse()

Notes:



**Keys**

* We need to give each array item a key – a string or a number that uniquely identifies it among other items in the array.
* JSX elements directly inside a map() call always need keys. Keys tell React which array item each component corresponds to, so that it can match them up later.
* Key becomes important if our array items can move (e.g., due to sorting), get inserted, or get deleted. A well – chosen key helps React infer what exactly has happened, & make the correct updates to the DOM tree.

**Where to get our key**

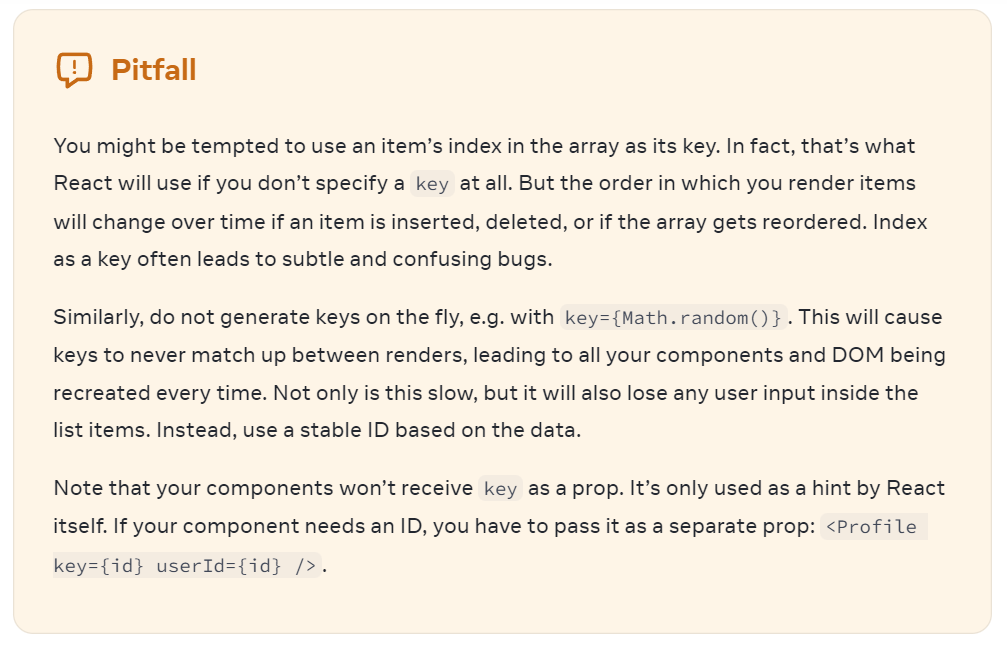
* Different sources of data provide different sources of keys:
* **Data from a database:** If our data is coming from a database, we can use the database keys/IDs, which are unique by nature.
* **Locally generated data:** If our data is generated & persisted locally (e.g., notes in a note – taking app), use an incrementing counter, crypto.randomUUID() or a package like uuid when creating items.

**Rules of keys**

* Keys must be unique among siblings. However, it’s okay to use the same keys for JSX nodes in different arrays.
* Keys must not change or that defeats their purpose! Don’t generate them while rendering.

**Why does React need keys?**

* Imagine that files on our desktop didn’t have names. Instead, we’d refer to them by their order – the 1st file, the 2nd file & so on. We could get used to it, but once we delete a file, it would get confusing. The 2nd file would become 1st, the 3rd file would be the 2nd file & so on.
* File names in a folder & JSX keys in an array serve a similar purpose. They let us uniquely identify an item between its siblings. A well – chosen key provides more information than the position within the array. Even if the position changes due to reordering, the key lets React identify the item throughout its lifetime.



**Keeping Components Pure**

* In Computer science (especially the world of functional programming), a pure function is a function with the following characteristics:
* It minds its own business i.e., It doesn’t change any objects or variables that existed before it was called.
* Same input, Same output i.e., Given the same input, a pure function should always return the same result.

Wrapper Components (Concept of composition)

* 1. **Custom Component**
* **Rule**: Components defined as Lowercase elements are built-in HTML elements while Components defined as Uppercase elements are custom component.
  1. **Props concept**
* React components use props to communicate with each other.
* Every parent component can pass some information to its child components by giving them props.
* Props might remind us of HTML attributes, but we can pass any JavaScript value through them including objects, arrays or functions.
  1. **Alternative Function syntax (Arrow function)**
* We can use arrow function instead of creating a function with “function” keyword.

|  |  |
| --- | --- |
| const ExpenseItem = (props) => {    return (        <Card>          <ExpenseDate date={props.date} />          <div>            <h2>{props.title}</h2>            <div>{props.amount}</div>          </div>        </Card>    ); | function ExpenseItem(props) {        return (        <Card>          <ExpenseDate date={props.date}/>          <div>            <h2>{props.title}</h2>            <div>{props.amount}</div>          </div>        </Card>      ); |

**2. React State & Working with Events**

**2.1 Listening to events & working with events Handlers**

* React lets us add event handlers to our JSX. Event handlers are our own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs, & so on.
* To add an event handler, we will first define a function & then pass it as a prop to the appropriate JSX tag.

e.g., We can define the clickHandler in 3 ways (All of these styles are equivalent.):

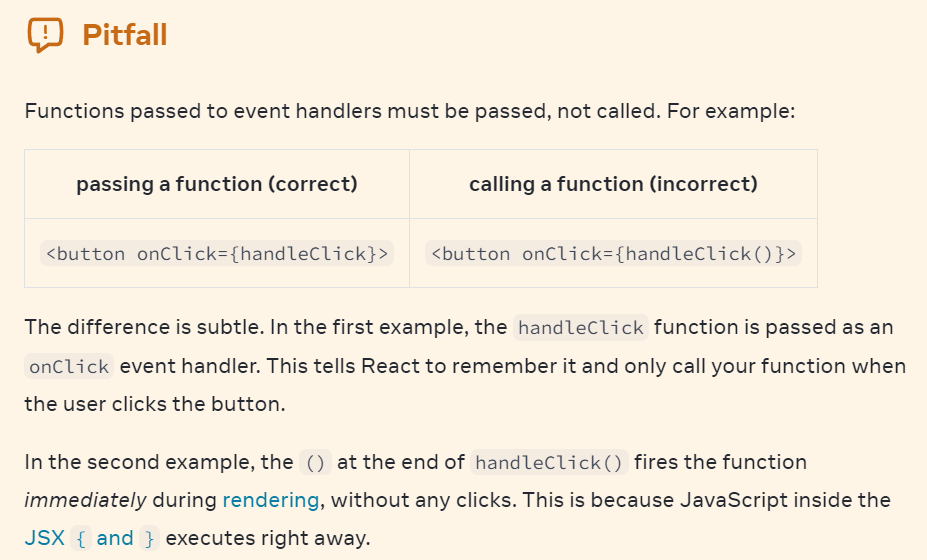
1. Normal function definition
2. Using Inline function definition
3. Using Arrow function definition

* By convention, it’s common to name event handlers as “**handle**” followed by the event name or Event name followed by “**handler**”.

e.g., onClick = { handleClick } **or** onClick = { clickHandler }

|  |
| --- |
| // Normal function definition  export default function ClickButton() {    const clickHandler = () => {      alert("Button has been clicked!!");    };    return (      <div>        <button onClick={clickHandler}>Click me!!</button>      </div>    );  } |
| // Using Inline function definition  export default function ClickButton() {    return (      <div>        <button          onClick={function handleClick() {            alert("Online Button has been click!!");          }}        >          Click me Inline!!        </button>      </div>    );  } |
| // Using Arrow function definition  export default function ClickButton() {    return (      <div>        <button          onClick={() => {            alert("Arrow function Button has been click!!");          }}        >          Click me arrow function        </button>      </div>    );  } |

**Notes:**



**Passing event handlers as props**

* Often, **we’ll want the parent component to specify a child’s event handler.**
* E.g., Consider buttons: depending on where we’re using a Button component, we might want to execute a different function – perhaps one plays a movie & another uploads an image.
* To do this, we pass the event handler as a prop to the component from its parent.
* Built – in components like <button> & <div> only support browser event names like onClick etc. However, when we’re building our own components, we can name our event handler props anyway that we like
* By convention, event handler props should start with “on”, followed by a capital letter like **onSmash, onPlayMovie, onUploadImage** etc.



* It’s common for components like buttons to contain styling but not specify behavior. Instead, components like **PlayButton** & **UploadButton** will pass event handlers down.

**Event Propagation**

* Event handlers will also catch events from any children our main component have. We say that an event “bubbles” or “propagates” up the tree; it starts with where the event happened & then goes up the tree.

e.g.,





* If we click on either button, it’s onClick will run first, followed by the parent <div>’s onClick & if we click the toolbar itself, only the parent <div>’s onClick will run.

**Stopping Propagation**

* Event handlers receive an event object as their only argument.
* By convention, it’s usually called “e”, which stands for “event”. We can use this object to read information about the event.
* That event object also lets us stop the propagation. If we want to prevent an event from reaching parent components, we need to call **e.stopPropagation()**





* When we click on a button (**PlayMovie**):

1. React calls the **clickHandler** passed to <button>.
2. clickHandler does the following:

* Calls **event.stopPropagation()**, preventing the event from bubbling or propagating up.
* Calls the **onSmash()** function, which is a prop passed from the Toolbar component

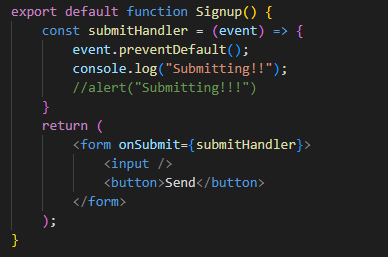
1. **onSmash** defined in the Toolbar component, displays the button’s own alert.
2. Since the propagation was stopped, the parent <div>’s onClick handler doesn’t run.

* As a result of **event.stopPropagation()**, clicking on the buttons now only show a single alert (from the <button>) rather than two of them (from the <button> & the parent toolbar <div>).
* We can add more code to the **clickHandler** before calling the parent **onSmash** event handler. This pattern provides an alternative to propagation. It lets the child component handle the event, while also letting the parent component specify some additional behavior. Unlike propagation, it’s not automatic
* The benefit of this pattern is that we can clearly follow the whole chain of code that executes as a result of some event.
* If we rely on propagation & it’s difficult to trace which handlers execute & why, try this approach instead.

**Preventing default behavior**

* Some browser events have default behavior associated with them. For e.g., a <form> submit event, which happens when a button inside of it is clicked, will reload the whole page by default.

**e.g.,**



* Both **event.stopPropagation() & event.preventDefault()** are unrelated:
* **event.stopPropagation()** stops the event handlers attached to the tags above from firing.
* **event.preventDefault()** prevents the default browser behavior for the few events that have it.
* Unlike rendering functions, event handlers don’t need to be pure, so it’s a great place to change something.

**e.g.,** Change an input’s value in response to typing, or change a list in response to a button press.

* However, in order to change some information, we first need some way to store it. In React, this is done by using **state**, a component’s memory.

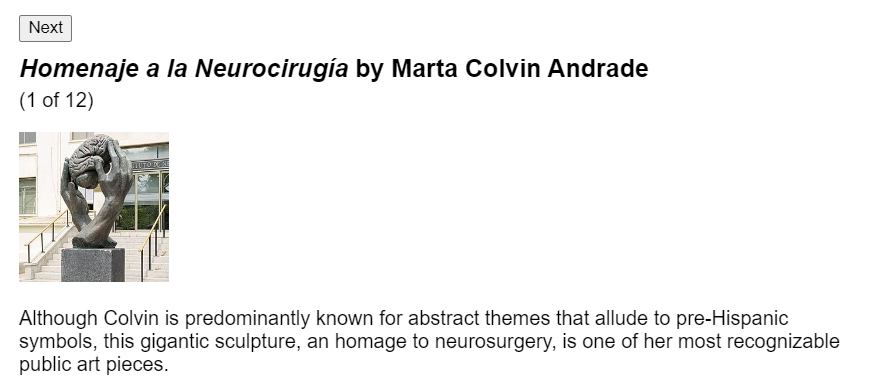
**2.2 Working with “State” & “useState” Hook**

* Components often need to change what’s on the screen as a result of an interaction.

e.g., Typing into the form should update the input field, clicking “next” on an image carousel should change which image is displayed, clicking “buy” should put a product in the shopping cart.

* Components need to “remember” things: the current input value, the current image, the shopping cart. In React, this kind of component – specific memory is called **State**.





**Imp. Point (above example)**

* The “**clickHandler**” is updating a local variable, index. But 2 things prevent that change from being visible:

1. **Local variables don’t persist between renders:** When React renders this component a second time, it renders it from scratch – it doesn’t consider any changes to the local variables.
2. **Changes to local variables won’t trigger renders:** React doesn’t realize it need to render the component again with the new data.

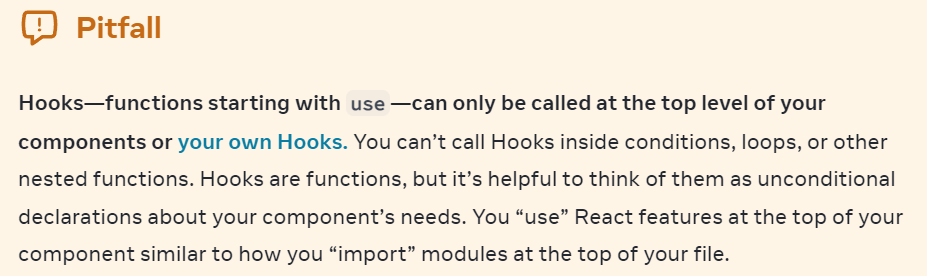
* To update a component with new data, 2 things need to happen:

1. **Retain** the data between renders.
2. **Trigger** React to render the component with new data (re – rendering)

* The **useState** Hook provides those 2 things:

1. A **state variable** to retain the data between renders.
2. A **state setter function** to update the variable & trigger React to render the component again.

* In React, **useState**, as well as any other function starting with “**use**”, is called a **Hook**. **Hooks are special functions that are only available while React is rendering.** They let us “hook into” different React features.



* The **[ & ] syntax** here is called **array Destructuring** & it lets us read values from an array. The array returned by useState always has exactly 2 items.



* When we call **useState**, we’re telling React that we want this component to remember the index.
* The only argument to **useState** is the initial value of your state variable.

e.g., Above, the index’s initial value is set to 0 with useState (0)

* Every time our component renders, useState gives us an array containing 2 values:

1. **The state variable** (**index**) with the value we stored.
2. The **state setter function** (**setIndex**) which can update the state variable & trigger React to render the component again.

**Giving a component multiple state variables**

* We can have multiple state variable in one component. It’s good to have multiple state variables if their state is unrelated, like index & showMore.
* Internally, React matches them up by their order.

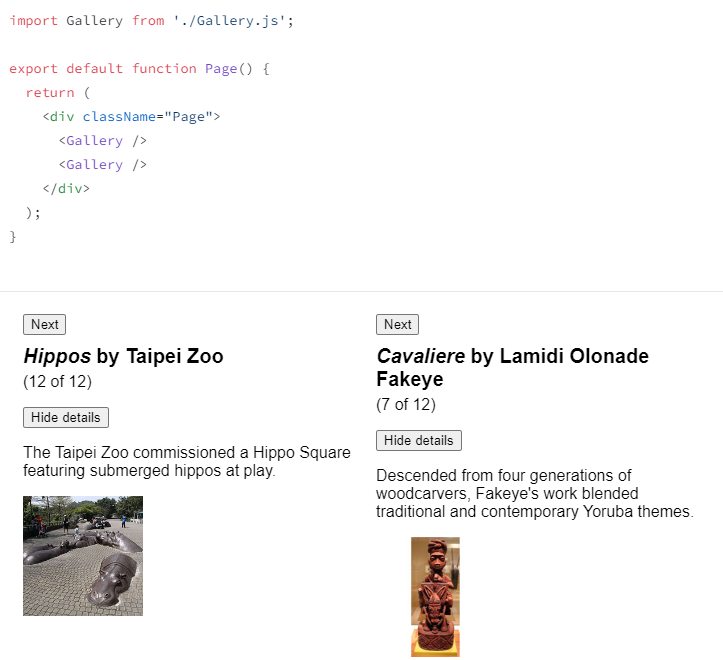
e.g., In below example, the component has 2 state variables, (a number index & a boolean showMore)



**State is isolated & private**

* State is local to a component instance on the screen. In other words, if we render the same component twice, each copy will have completely isolated state! Changing one of them will not affect the other.

For e.g., In this example, the Gallery component is rendered twice with no changes to its logic.



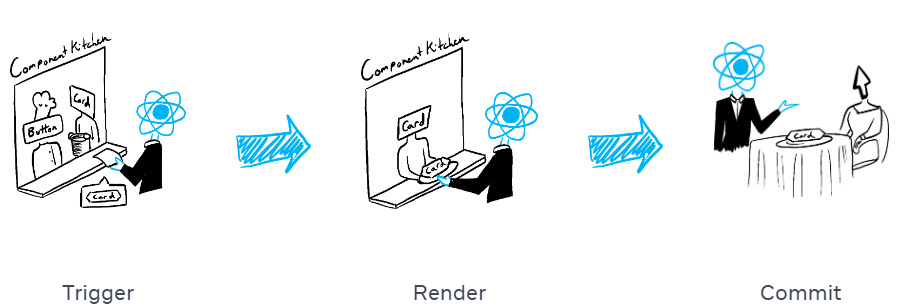
* This is what makes state different from regular variables that we might declare at the top of our module.
* State is not tied to a particular function call or a place n the code, but it’s “local” to the specific place on the screen.
* Unlike props, state is fully private to the component declaring it. The parent component can’t change it. This lets us add state to any component or remove it without impacting the rest of the components.

**2.3 Render & Commit [How React works internally]**

* Before our components are displayed on screen, they must be rendered by React. Understanding the steps in this process will help us think about how our code executes & explain its behavior.
* **Analogy**: Imagine that our **Components are COOKS** in the kitchen, assembling tasty dishes from ingredients.

In this scenario, **React is the WAITER** who puts in requests from customers & brings them their orders. This process of requesting & serving UI has 3 steps:

1. **Triggering a render** (delivering the guest’s order to the kitchen)
2. **Rendering the component** (preparing the order in the kitchen)
3. **Committing to the DOM** (placing the order on the customer’s table)



**Step 1: Triggering a render**

* There are 2 reasons for a component to render:

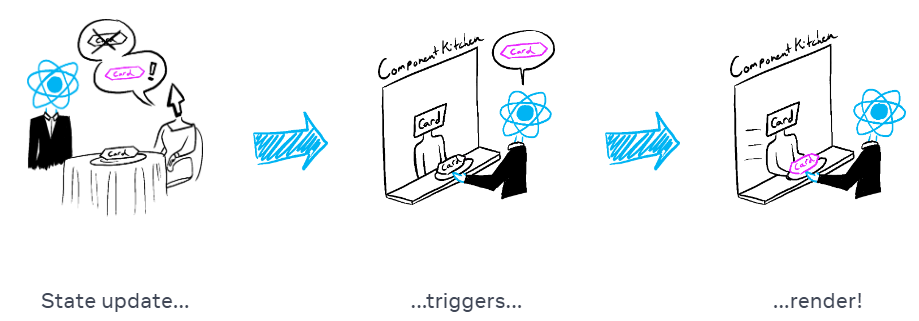
1. It’s the component’s **initial render.**
2. The component’s (or one of its ancestor’s) **state has been updated.**

* **Initial render:** When our app starts, we need to trigger the initial render. Frameworks & sandboxes sometimes hide this code, but it’s done by calling **createRoot()** with the target DOM node, & then calling its **render()** method with our component.



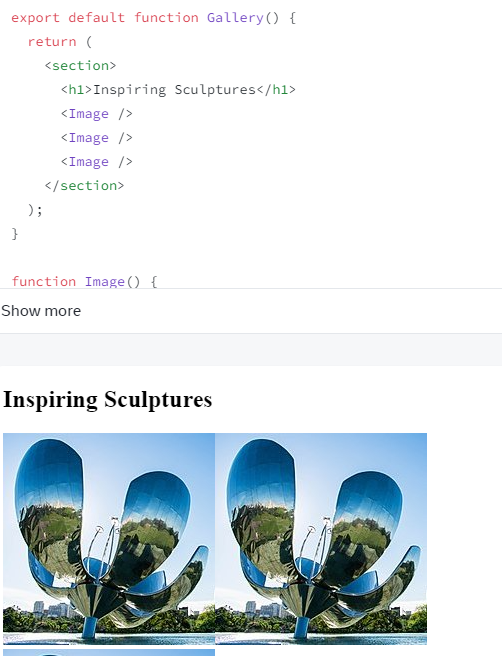
* **Re – renders when state updates:** Once the component has been initially rendered, we can trigger further renders by updating its state with the **set() function**. Updating our component’s state automatically queues a render.

(We can image these as a restaurant guest ordering tea, dessert & all sort of things after putting in their first order, depending on the state of their thirst or hunger.)



**Step 2: React renders our components (“Rendering” is React calling our components)**

* After we trigger a render, React calls our components to figure out what to display on screen.
* **On initial render,** React will call **the root component**.
* **For subsequent renders,** React will call the function component whose state update triggered the render.
* This process is recursive: If the updated component returns some other component, React will render that component next, & so on… This process will continue until there is no more nested components & React knows exactly what should be displayed on screen.



* **During the initial render,** React will create the DOM nodes for <selection>, <h1>, & 3 <img> tags.
* **During a re – render,** React will calculate which of their properties, if any, have changed since the previous render. It won’t do anything with that information until the next step, the commit phase.

**Step 3: React commits changes to the DOM**

* After rendering (calling) our components, React will modify the DOM
* **For the initial render,** React will use the **appendChild()** DOM API to put all the DOM nodes it has created on screen.
* **For re – renders,** React will apply the minimal necessary operations (calculated while rendering!) to make the DOM match the latest rendering output.
* React only changes the DOM nodes if there’s a difference between renders. For e.g., Here is a component that re – renders with different props passed from its parent every second.

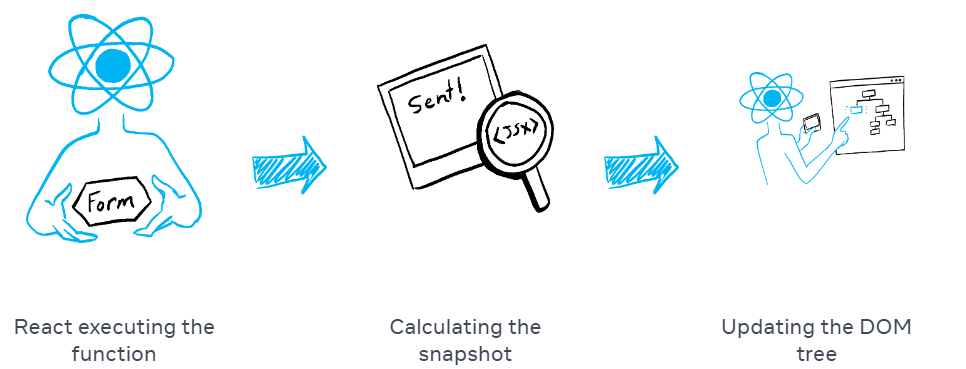
**2.4 State as a Snapshot**

* State variables might look like regular JavaScript variables that we can read & write to. However, state behaves more like a snapshot.
* Setting state doesn’t change the state variable we already have, but instead triggers a re – render.

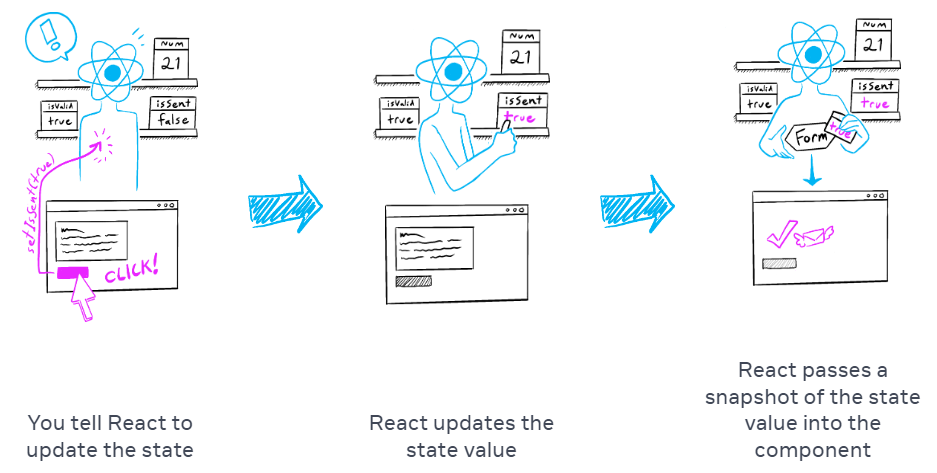
**Imp. Note**

* “**Rendering**” means that React is calling our component, which is a function. **The JSX we return from that function is like a snapshot of the UI in time**. Its props, event handlers, & local variables were all calculated using its state at the time of the render.
* Unlike a photograph or a movie frame, the UI “snapshot” we return is interactive. It includes logic like event handlers that specify what happens in response to inputs. React updates the screen to match this snapshot & connects the event handlers. As a result, pressing a button will trigger the click handler from our JSX.
* When React re – renders a component:

1. React calls our function again.
2. Our function returns a new JSX snapshot.
3. React then updates the screen to match the snapshot we’ve returned.



* As a component’s memory, state is not like a regular variable that disappears after function returns. State actually “lives” in React itself outside of our function. When React calls our component, it gives us a snapshot of the state for that particular render. Our component returns a snapshot of the UI with a fresh set of props & event handlers in its JSX, all calculated using the state values from that render.

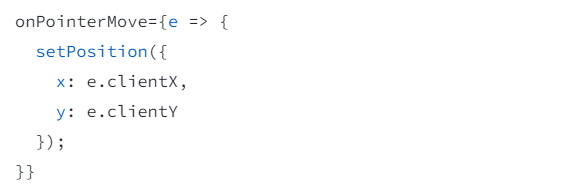


* A state variable’s value never changes within a render, even if its event handler’s code is asynchronous. We don’t need to worry whether the state has changed while the code is running.

**2.5 Batching**

**2.6 Updating Objects in State**

* State can hold any kind of JavaScript value, including objects. But we shouldn’t change objects that we hold in the React state directly. Instead, when we want to update an object, we need to create a new one (or make a copy of an existing one) & then set the state to use that copy.
* Although objects in React state are technically mutable, we should treat them as if they were immutable – like numbers, Booleans & strings i.e., we should treat any JavaScript object that we put into state as read – only.
* To actually trigger a re – render in this case, create a new object & pass it to the state setting function.



**Copying objects with the Spread syntax (…)**

* We can use spread syntax in case we want to update only one field in a form, but keep the previous values for all other fields.

|  |
| --- |
|  |
|  |

**Note**: (…) spread syntax is “shallow” – it only copies things one level deep. This makes it fast, but it also means that if we want to update a nested property, we’ll have to use it more than once.

**Using a single event handler for multiple fields**

* We can also use the [ & ] braces inside our object definition to specify a property with dynamic name.



**Updating a nested object**

* To update a nested object, we need to create copies all the way up from the place we’re updating.

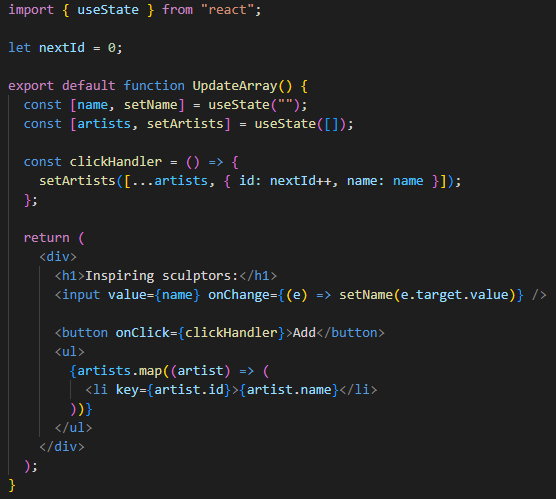
|  |
| --- |
| // To change city |

**2.7 Updating Arrays in State**

* Arrays are mutable in JavaScript, but we should treat them as immutable when we store them in state. Just like with objects, when we want to update an array stored in state, we need to create a new one ( or make a copy of an existing one), and then set state to use the new array.
* In order to update arrays without mutation, we will have to pass a new array to our state setting function. To do that, we can create a new array from the original array in our state by calling its non – mutating methods like filter() & map(), then we can set our state to the resulting new array.

|  |  |  |  |
| --- | --- | --- | --- |
| **No.** | **Operation** | **Avoid (mutates the array)** | **Prefer (returns a new array)** |
| **1.** | Adding | push, unshift | concat, […arr] (spread syntax) |
| **2.** | Removing | pop, shift, splice | filter, slice |
| **3.** | Replacing | splice, arr[i] = …assignment | map |
| **4.** | Sorting | reverse, sort | Copy the array first |

**E.g.1,**



**2.8 Principles for structuring state**

When we write a component that holds some state, we’ll have to make choices about how many state variables to use & what the shape of their data should be.

While it’s possible to write correct programs even with a suboptimal state structure, there are few principles that can guide us to make better choice:

1. **Group related state:** If we always update 2 or more state variables at the same time, consider merging them into a single state variable.

**Note**: If our state variable is an object, remember that we can’t update only one field in it without explicitly copying the other fields.

1. **Avoid contradictions in state:** When the state is structured in a way that several pieces of state may contradict & disagree with each other, we leave room for mistakes. Try to avoid this.
2. **Avoid redundant state:** If we can calculate some information from the component’s props or its existing state variables during rendering, we should not put that information into that component’s state.

**e.g.,** We can always calculate **fullName** from **firstName** & **lastName** during render, so remove it from state.

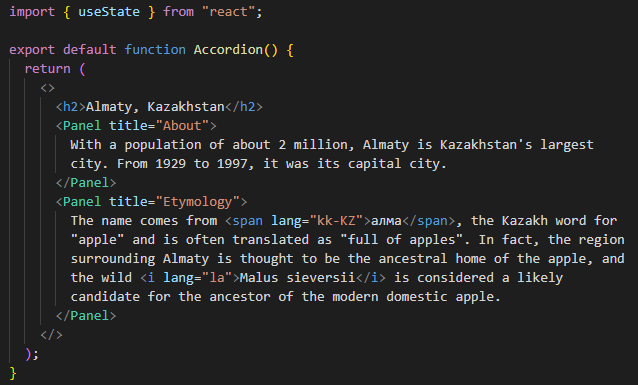
1. **Avoid duplication in state:** When the same data is duplicated between multiple state variables, or within nested objects, it’s difficult to keep them in sync. Reduce duplication when we can.
2. **Avoid deeply nested state:** Deeply hierarchical state is not very convenient to update. When possible, prefer to structure state in a flat way.

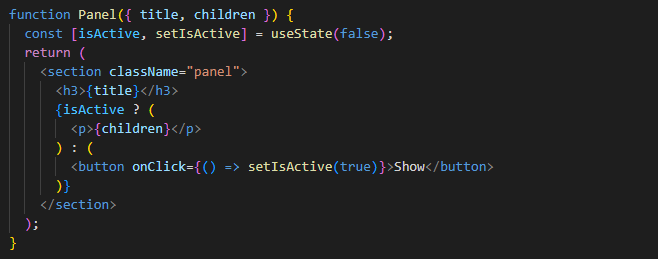
The goal behind these principles is to make state easy to update without introducing mistakes. Removing redundant & duplicate data from state helps ensure that all its pieces stay in sync.

**2.9 Sharing State between Components**

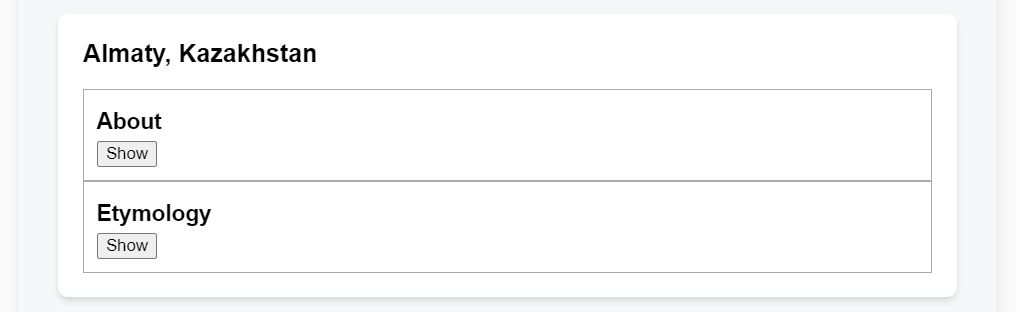
* Sometimes, we want the state of 2 components to always change together.
* To do it, remove state from both of them, move it to their closest common part, & then pass it down to them via props. This is known as **Lifting State up**, & it’s one of the most common things we will do writing React code.

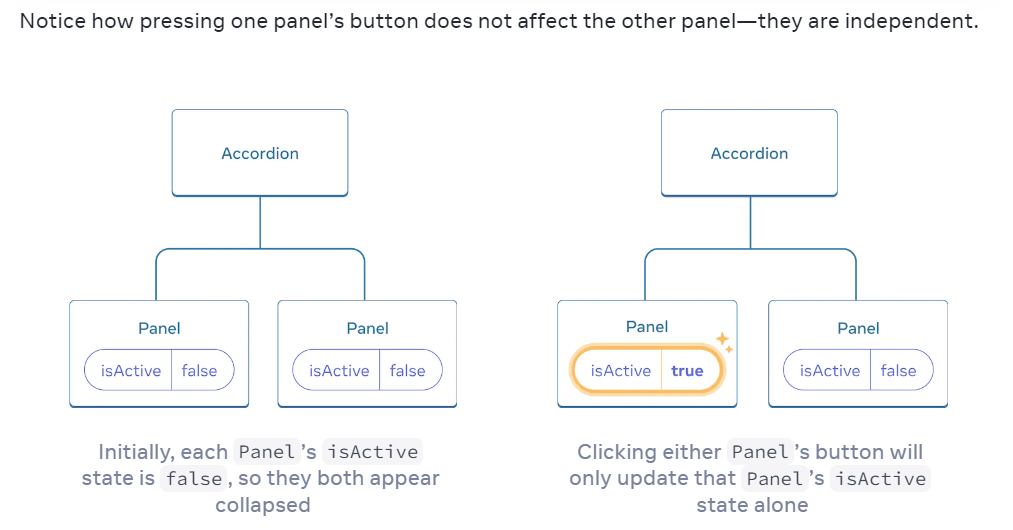
**e.g.,**





**Output:**



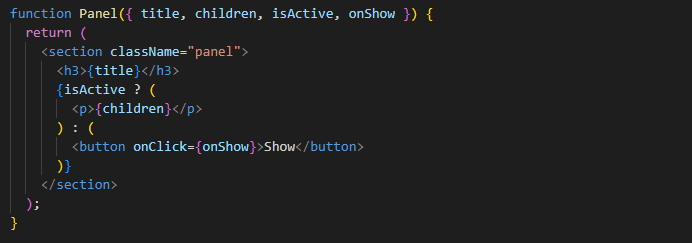


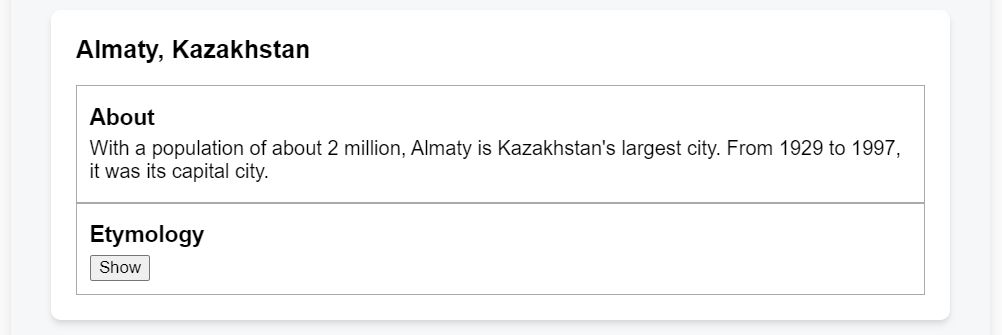
* Let’s say we want to change the Accordion so that only one panel is expanded at any given time. To coordinate these 2 panels, we need to “lift their state up” to a parent component in 3 steps:

1. Remove state from the child components.
2. Pass hardcoded data from the common parent.
3. Add state to the common parent & pass it down together with the event handlers.

**e.g.,**







**2.10 Controlled Vs Uncontrolled components**

|  |  |  |
| --- | --- | --- |
| **No.** | **Controlled Component** | **Uncontrolled Component** |
| **1.** | A component whose important information is driven by props rather than its own local state is called as “Controlled” component. This lets the parent component fully specify its behavior. | A component with some local state is called as “Uncontrolled” component. |
| **2.** | **e.g.,** The final Panel (in previous example) with the **isActive** prop is controlled by the Accordion component. | **e.g.,** the original Panel component (in previous example) with an **isActive** state variable is uncontrolled because its parent can’t influence whether the panel is active or not. |
| **3.** | Controlled components are maximally flexible, but they require the parent components to fully configure them with props. | Uncontrolled components are easier to use within their parents because they require less configuration. But they’re less flexible when you want to coordinate them together. |

**Note:**

* In practice, “**controlled**” & “**uncontrolled**” aren’t strict technical terms – each component usually has some mix of both local state & props. However, this is useful way to talk about how components are designed & what capabilities they offer.
* When writing a component, consider which information in it should be controlled (via props), & which information should be uncontrolled (via state). But we can always change our mind & refactor later.

**2.11 Single Source of Truth for each State**

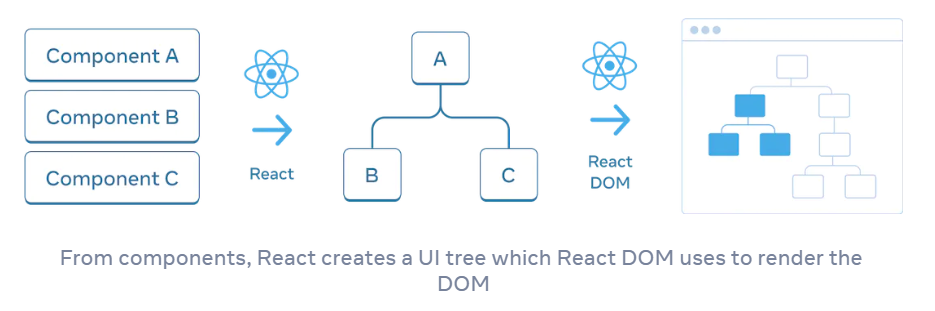
* In a React application, many components will have their own state. Some state may “live” close to the leaf components (components at the bottom of the tree) like inputs. Other state may “live” closer to the top of the app.
* For e.g., even client – side routing libraries are usually implemented by storing the current route in the React state, & passing it down by props.
* The principle of “**Single source of truth for each state”** says that **For each unique piece of state, we will choose the component that “owns” it**.
* It doesn’t mean that all state lives in one place – but that for each piece of state, there is a specific component that holds that piece of information, lift it up to their common shared parent, & pass it down to the children that need it.
* Our app will change as we work on it. It’s common that we will move state down or back up while we’re still figuring out where each piece of the state “lives”. This is all part of the process!

**2.12 Preserving & Resetting State**

* State is isolated between components. React keeps track of which state belongs to which component based on their place in the UI tree.
* We can control when to preserve state & when to reset it between re – renders.

**The UI tree**

* Browsers use many tree structures to model UI. The **DOM** (Document Object Model) represents HTML elements, the CSSOM does the same for CSS.
* React also uses tree structures to manage & model the UI we make.
* React makes UI trees from our JSX. Then React DOM updates the browser DOM elements to match that UI tree. (React Native translates these trees into elements specific to mobile platforms.)



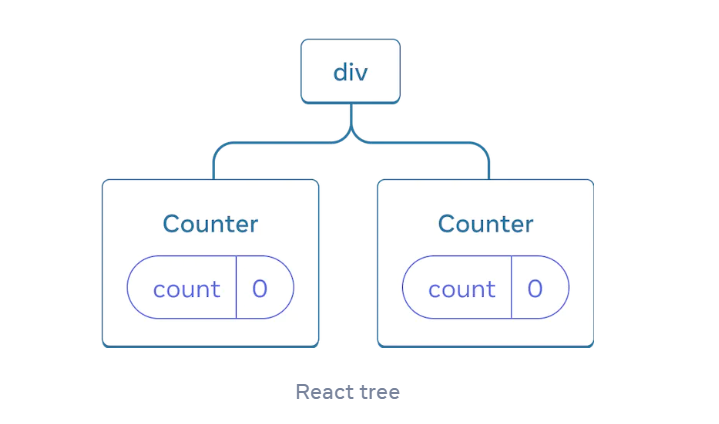
**State is tied to a position in the tree**

* When we give a component state, we might think the state “lives” inside the component. But the state is actually held inside React.
* React associates each piece of state it’s holding with the correct component by where that component sits in the UI tree.
* In React, each component on the screen has fully isolated state. For e.g., if we render 2 counter components side by side, each of them will get its own, independent (**score** & **hover** states in below example)

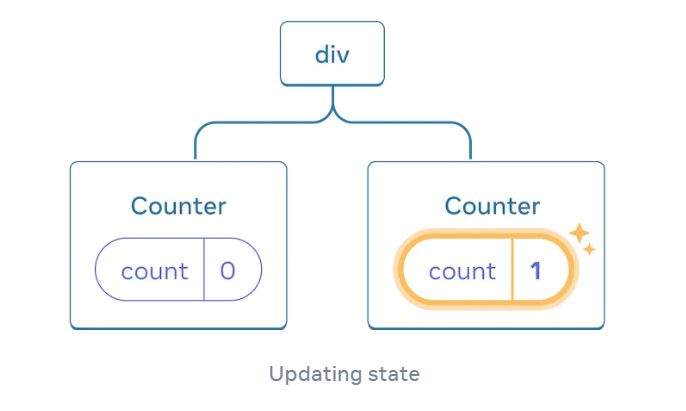
**e.g.,1 –** Here, there are 2 separate counters because each is rendered at its own position in the tree. We don’t usually have to think about these positions to use React, but it can be useful to understand how it works.







* As we can see, when one counter is updated, only the state for that component is updated:

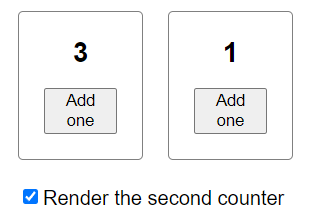
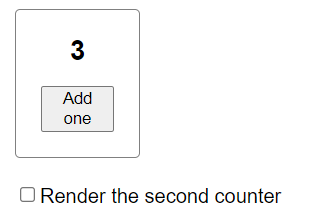
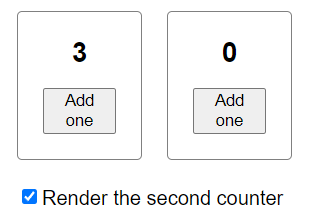


* React will keep the state around for as long as we render the same component at the same position. To see this, increment both counters, then remove the second component by unchecking “Render the second counter” checkbox, & then add it back by ticking it again.

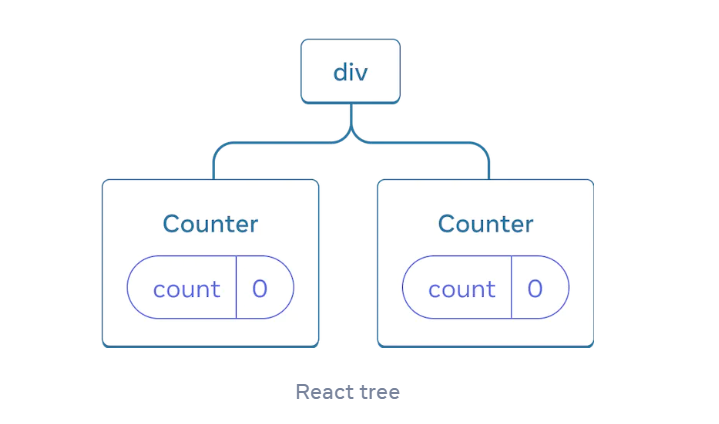
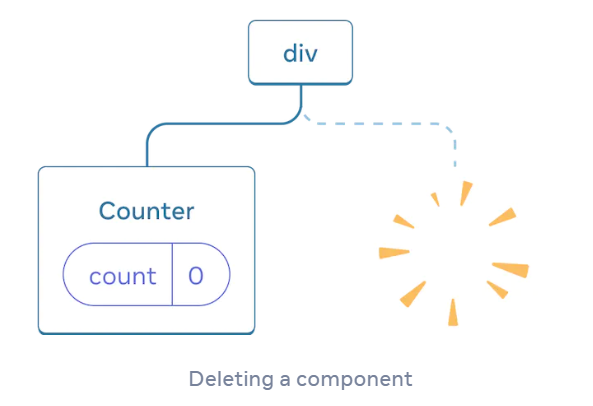
e.g.,2 –



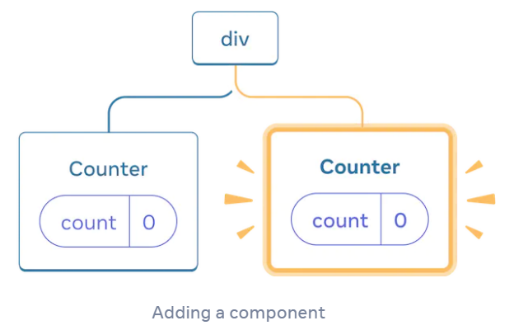
**Output:**

* Here we can notice that we stop rendering the second counter, its state disappears completely. That’s because when React removes a component, it destroys its state.

* When we tick “Render the second counter”, a second **Counter** & its state are initialized from scratch (**score = 0**) & added to the DOM.



* React preserves a component’s state for as long as it’s being rendered at its position in the UI tree. If it gets removed, or a different component gets rendered at the same position. React discards its state.

**Same component at the same position preserves state**

**3. Styling React components**

3.1 Conditional & Dynamic Styles

3.2 Styled Components

3.3 CSS Modules