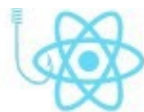# What are hooks

- *Hooks* are a new addition in React 16.8. They let you use state and other React features without writing a class.

- A Hook is a special function that lets you "hook into" React features..

- Basic Hooks
  - useState
  - useEffect
  - useContext

Additional Hooks
  - useReducer
  - useCallback
  - useMemo
  - useRef
  - useLayoutEffect

.

React Hooks Lifecycle

	Mounting	Updating	Unmounting

"Render phase"

Pure and has no side effects. May be paused, aborted or restarted by React

Mounting:
- `function () {}`
  - `useMemo()`
  - `return ()`

Updating:
- `useState()` `useReducer()` `useContext()`
  - `useCallback()` → ✗

"Commit phase"

Can work with DOM, run side effects, schedule updates.

React updates DOM and refs

`useEffect()`   `useLayoutEffect()`

When would I use a Hook?

- If you write a function component and realize you need to add some state to it, previously you had to convert it to a class. Now you can use a Hook inside the existing function component. We're going to do that right now!

Hooks Rules

- Only call Hooks **at the top level**. Don't call Hooks inside loops, conditions, or nested functions.

- Only call Hooks **from React function components**. Don't call Hooks from regular JavaScript functions.

useState

- Returns a stateful value, and a function to update it.
- During the initial render, the returned state (state) is the same as the value passed as the first argument (initialState).

```
const [state, setState] = useState(initialState);
```

- The setState function is used to update the state. It accepts a new state value and enqueues a re-render of the component.

```
setState(newState);
```

useState

- If the new state is computed using the previous state, you can pass a function to setState. The function will receive the previous value, and return an updated value. Here's an example of a counter component that uses both forms of setState:

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

Warning!

Note

Unlike the `setState` method found in class components, `useState` does not automatically merge update objects. You can replicate this behavior by combining the function updater form with object spread syntax:

```
setState(prevState => {
  // Object.assign would also work
  return {...prevState, ...updatedValues};
});
```

Another option is `useReducer`, which is more suited for managing state objects that contain multiple sub-values.

useState - Lazy initial state

- The initialState argument is the state used during the initial render. In subsequent renders, it is disregarded. If the initial state is the result of an expensive computation, you may provide a function instead, which will be executed only on the initial render:

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

.

useEffect

- `useEffect(()= { … } , [ dependencies ] );`

- A function that should be executed AFTER every component evaluation IF the specified dependencies changed.

- Dependencies of this effect – the function only runs if the dependencies changed.

useEffect

- Accepts a function that contains imperative, possibly effectful code.

```
useEffect(didUpdate);
```

- Mutations, subscriptions, timers, logging, and other side effects are not allowed inside the main body of a function component (referred to as React's *render phase*). Doing so will lead to confusing bugs and inconsistencies in the UI.

- Instead, use useEffect. The function passed to useEffect will run after the render is committed to the screen. Think of effects as an escape hatch from React's purely functional world into the imperative world.

- By default, effects run after every completed render, but you can choose to fire them only when certain values have changed.

useEffect - Cleaning up an effect

- Often, effects create resources that need to be cleaned up before the component leaves the screen, such as a subscription or timer ID. To do this, the function passed to useEffect may return a clean-up function. For example, to create a subscription:

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

- The clean-up function runs before the component is removed from the UI to prevent memory leaks. Additionally, if a component renders multiple times (as they typically do)

Did mount and unmount

- If you want to run an effect and clean it up only once (on mount and unmount), you can pass an empty array ([]) as a second argument. This tells React that your effect doesn't depend on *any* values from props or state, so it never needs to re-run. This isn't handled as a special case — it follows directly from how the dependencies array always works.

- If you pass an empty array ([]), the props and state inside the effect will always have their initial values . While passing [] as the second argument is closer to the familiar `componentDidMount` and `componentWillUnmount` mental model.

Conditional update

- You can tell React to *skip* applying an effect if certain values haven't changed between re-renders. To do so, pass an array as an optional second argument to useEffect:

```javascript
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

Week 3 - Day 3

useContext

- The **useContext**() method is an alternative to prop-drilling through the component tree and creates an internal global state to pass data.

- Accepts a context object (the value returned from React.createContext) and returns the current context value for that context. The current context value is determined by the value prop of the nearest <MyContext.Provider> above the calling component in the tree.
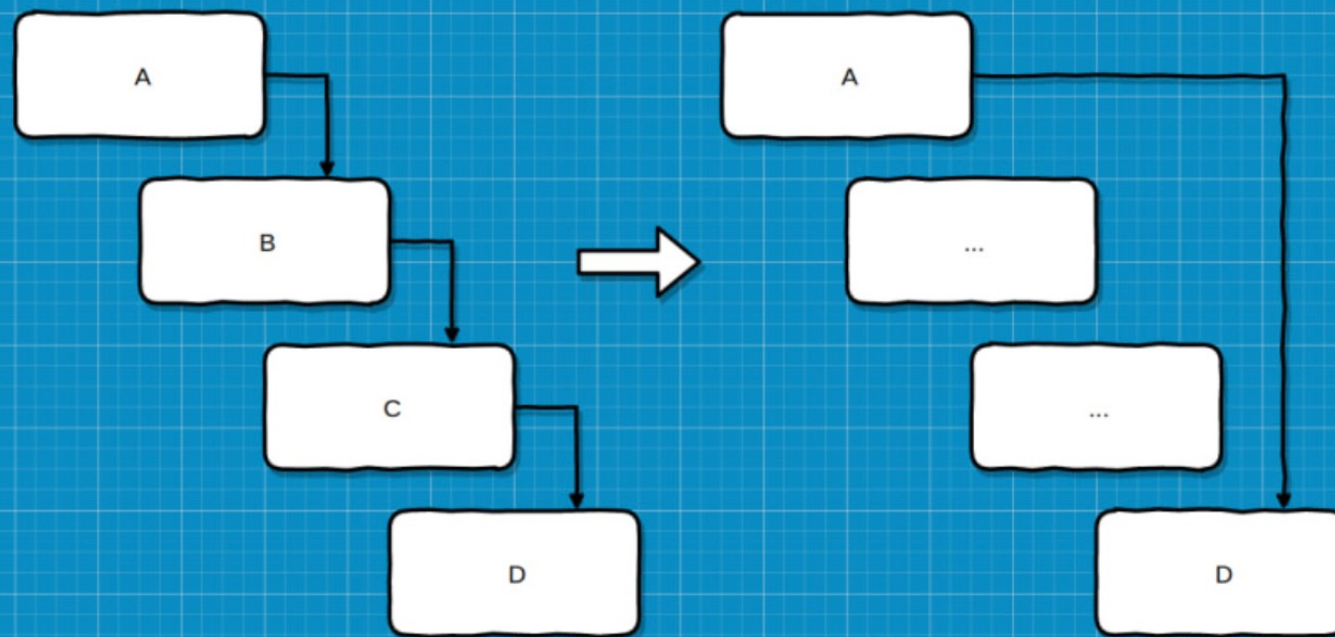
```
const value = useContext(MyContext);
```

- When the nearest <MyContext.Provider> above the component updates, this Hook will trigger a rerender with the latest context value passed to that MyContext provider. Even if an ancestor uses React.memo or shouldComponentUpdate, a rerender will still happen starting at the component itself using useContext.
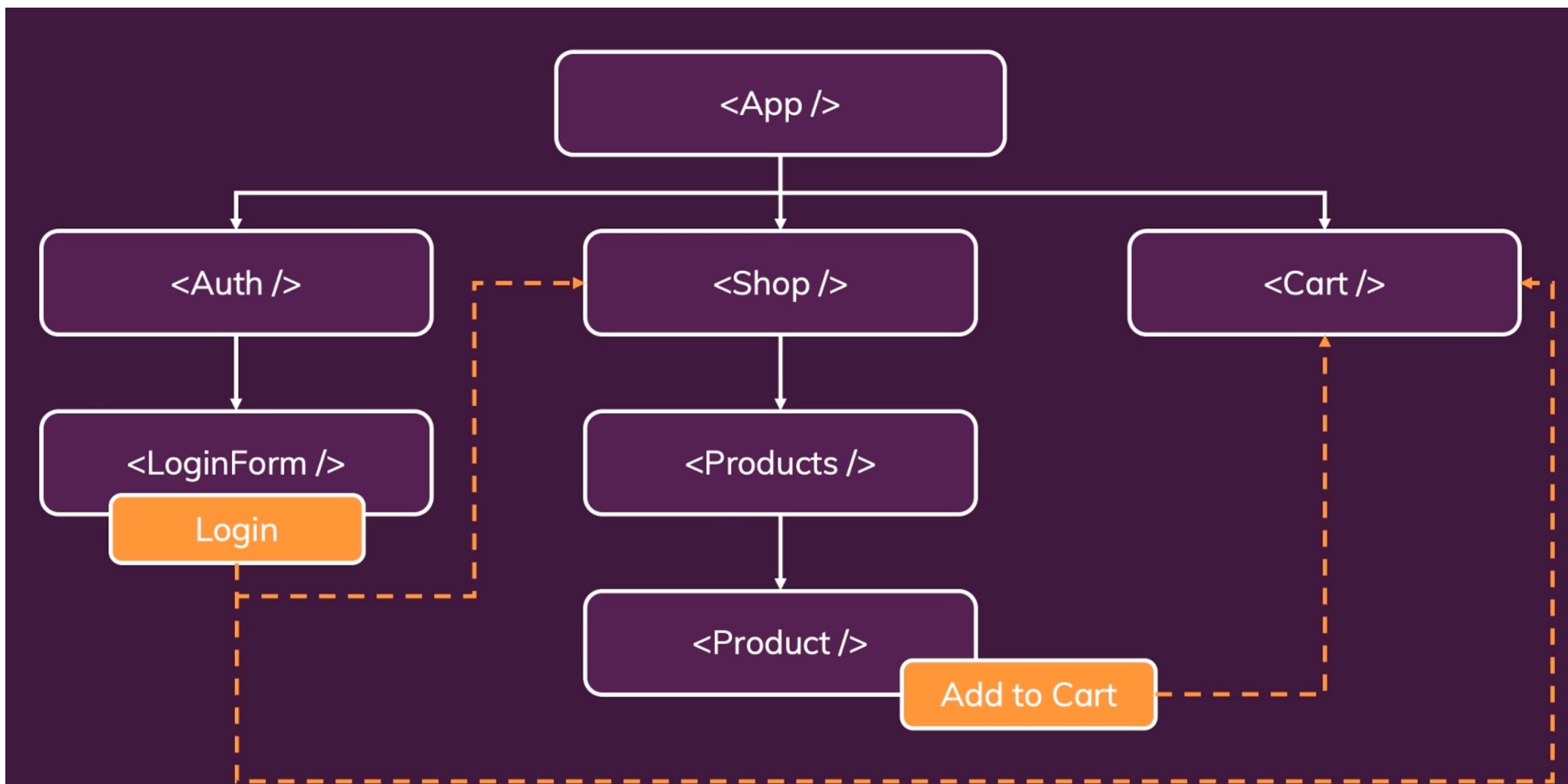
Context API

useContext - steps

- Create a context component

```
export const APIConfig = React.createContext([]);
```

- Wrap component with provider

```
<APIConfig.Provider value= 'http://localhost:8080/posts/'>
    <App />
</APIConfig.Provider>
```

- Retrieve the value from the one of the child components of <App>

```
const api = useContext(APIConfig);
```

useContext

- Don't forget that the argument to useContext must be the *context object itself*:

- **Correct:** useContext(APIConfig)

- **Incorrect:** useContext(APIConfig.Consumer)

- **Incorrect:** useContext(APIConfig.Provider)

useReducer

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

- An alternative to useState. Accepts a reducer of type (state, action) => newState, and returns the current state paired with a dispatch method. (If you're familiar with Redux, you already know how this works.)

- useReducer is usually preferable to useState when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one. useReducer also lets you optimize performance for components that trigger deep updates because you can pass dispatch down instead of callbacks.

useReducer

```jsx
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

useCallback.

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

Returns a memoized callback.

- Pass an inline callback and an array of dependencies. useCallback will return a memoized version of the callback that only changes if one of the dependencies has changed. This is useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders (e.g. shouldComponentUpdate).

- useCallback(fn, deps) is equivalent to useMemo(() => fn, deps).

useMemo

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Returns a memoized value.

- Pass a "create" function and an array of dependencies. useMemo will only recompute the memoized value when one of the dependencies has changed. This optimization helps to avoid expensive calculations on every render.

- Remember that the function passed to useMemo runs during rendering. Don't do anything there that you wouldn't normally do while rendering. For example, side effects belong in useEffect, not useMemo.

- If no array is provided, a new value will be computed on every render.

useRef

```
const refContainer = useRef(initialValue);
```

- useRef returns a mutable ref object whose .current property is initialized to the passed argument (initialValue). The returned object will persist for the full lifetime of the component.

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

useRef

- Essentially, useRef is like a "box" that can hold a mutable value in its .current property.

- You might be familiar with refs primarily as a way to access the DOM. If you pass a ref object to React with <div ref={myRef} />, React will set its .current property to the corresponding DOM node whenever that node changes.

- However, useRef() is useful for more than the ref attribute. It's handy for keeping any mutable value around similar to how you'd use instance fields in classes.

- This works because useRef() creates a plain JavaScript object. The only difference between useRef() and creating a {current: …} object yourself is that useRef will give you the same ref object on every render.

useLayoutEffect

- The signature is identical to useEffect, but it fires synchronously after all DOM mutations. Use this to read layout from the DOM and synchronously re-render. Updates scheduled inside useLayoutEffect will be flushed synchronously, before the browser has a chance to paint.

- *Prefer the standard useEffect when possible to avoid blocking visual updates.*