Unit - 3

Components

- We stopped last unit at *stateful* and *stateless* components. Just to recap, these are components that we make in react to break down our website into multiple pieces.
- *Stateless* components have no *state*. This means that they are just a function that returns some HTML. They are used for things like headers, footers, and other things that don't change.
- *Stateful* components have *state*. This means that they are a class that extends `React.Component` and has a `render()` method. They are used for things like forms, and other things that change.
- We'll see later that we can kind of cheat through this and make our stateful components too as functional components, but for now, we'll stick to the rules.
- In modern react, everybody writes stateful components as functions instead of classes.

```
// This is a stateful component
class MyComponent extends React.Component {
  // some state logic here. we'll talk about what comes here in
the next section
  render() {
    return (
      <div>
        <h1>Hello World!</h1>
      </div>
    );
  }
}

// This is a stateless component
function MyComponent() {
  return (
    <div>
      <h1>Hello World!</h1>
    </div>
  );
}
```

States

- States are the way we store data in react. They are the way we make our components dynamic. They are the way we make our components *react* to changes in the data.
- It allows us to make components like forms, where the data changes as the user types, or a counter, where the data changes as the user clicks on the increment button.
- States are stored in the `state` property of the component. This is an object that we can access in the `render()` method of the component.
- To make a function component, this is what we do:

```
class MyComponent extends React.Component {
  // this is the important part
  constructor(props) {
    super(props); // this is required | why? I'll tell you in a
second
    this.state = {
      // this is where we define our state. it's just an object
that holds any values we want dynamic. In this case, name and age.
Now, these two don't really change in this example, but we'll see
how to change them in the next section.
      name: "John Doe",
      age: 20,
    };
  }
  // until here

  render() {
    return (
      <div>
        <h1>Hello {this.state.name}!</h1> // we can access the
state here
        <p>You are {this.state.age} years old.</p>
      </div>
    );
  }
}
```

Info

- The `constructor()` method is a special method that is called when the component is created. It is used to initialize the component. Anything you

want to run *once* when the component initializes goes here. The `super(props)` is a method that calls the constructor method of the parent component.
- Here, our parent component is `React.Component`. We need to explicity call the constructor of React.Component to initialize the component. If we don't do this step, we get errors while using States in the component.
- in `super(props)` , the `props` are just the props that are passed to our component. This is passed to the parent component's constructor as well to initialise it.

`setState()`

- We just made an object to hold the state, but how do we change it?
- We cannot just do `this.state.name = "John Smith"` because that will not trigger a re-render of the component.
- We must let react know when our state variables change so that it can figure out what to do with the contents on the screen and if a rerender is required.
- We use the `setState()` method to do this. This is how we use it:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "John Doe",
      age: 20,
    };
    this.handleClick = this.handleClick.bind(this); // we have to
bind the handleClick function to use the this context of the
component instance. Since handleClick is a common class method,
it'd be using the this context of the class instead of the
instance of the class when it's called. Please refer to the
previous chapter for what 'this' means
  }

  // we'll use this function to change the state
  handleClick() {
    this.setState({ name: "John Smith" }); // set state takes an
object as an argument. This object contains all the state
variables that have changed along with their new data. In this
case, I'm changing only name to "John Smith"
  }

  render() {
```

```
    return (
      <div>
        <h1>Hello {this.state.name}!</h1>
        <p>You are {this.state.age} years old.</p>
        <button onClick={handleClick}>Change Name</button>
        // onClick of the button, call the handleClick function.
This will change the state.
      </div>
    );
  }
}
```

Info

- The `onClick` attribute in JSX is a special attribute that is used to call a function when the element is clicked. This is similar to the `addEventListener()` method in javascript.
- You can pass an `event` object to these handler Functions. We don't need it here so we don't pass it in. In case you need the event object to do stuff like `event.preventDefault()` and stuff, you can just take the `event` object as a parameter just like how we did in HTML.
- This is just a wrapper for the actual onclick function we use in HTML.
- In HTML, we passed the onclick function as a string, and here we pass it in curly braces as a function.

Creating a Digital Clock With States

- A digital clock has to be updating in real time. This means that we have to change the state every other second.
- But before that, how do we make a react app? Until now, we just put stuff in an html file and imported react through CDNs.
- This is not really the way to do it. It gets confusing and messy when you have multiple pages and a more complex project.
- It also doesn't properly containerise components. You gotta write everything in one file. That isn't great for large projects where you'd have multiple people working

Info

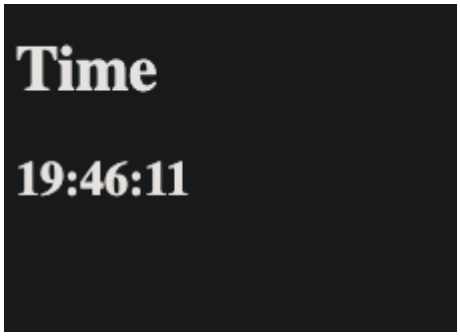
- The slides recommend using `create-react-app` to create a react app. This is the old way of doing things and it's *deprecated*.

> - The modern way of doing it, is to use frameworks like nextjs, gatsby, etc. This is recommended by the react team themselves
> - Unfortunately, we study at PES and PES likes to stick to the old ways. You can use `npx create-react-app` but I'd be using a similar but more modern alternative called `vite`. You can find the docs for it here
> - To install a react app using vite, run `npm create vite@latest` and follow the prompts on the screen.
> - After you're done with that, cd into the created project and then run `npm run dev` to start the development server. Make sure you have NodeJS and npm installed before you do this.

- Now that we have our react app, let's make a digital clock.
- Here's the code:

```
// App.jsx (this is the file that contains the main component)
import React from "react";

export default class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      time: new Date().toLocaleTimeString(), // this is our state.
We initialise it to the current time.
    };
  }

  // the render method is rerun everytime a rerender is required.
  render() {
    // we use the setTimeout function to change the state every
second. This will change the time and rerender the component.
    // you might intuitive think to use a setInterval because you
need this to happen every second.
    // We use setTimeout here because this function runs everytime
a render is required.
    // The code inside the setTimeout causes a rerender, which
calls this function again, which creates a new setTimeout for it
to update in the next second
    // If we had used setInterval, it'd create a new setInterval
every second which would soon lead to an infinite loop. It's kinda
like recursion.
    setTimeout(() => {
      this.setState({
```

```
        time: new Date().toLocaleTimeString(), // every second,
update the time to the current time.
      });
    }, 1000);
    return (
      <div>
        <h1>Time</h1>
        <h2>{this.state.time}</h2>
      </div>
    );
  }
}
```



Another Form Of setState()

- Imagine you have a normal counter, which does previous state's count + 1.
- You might intuitive think to do something like this:

```
this.setState({
    counter: this.state.counter + 1
})
```

- However, this might not work consistently.
- This is because React *batches* updates to the state sometimes for performance optimisation
- So `this.state.counter` is not guaranteed to be updated instantly.
- This difference might not be seen when you're triggering this with a button manually but if you're running this counter every second with a `setInterval`, it might appear as if it's skipping numbers
- To fix this, you can use another form of `setState()` that takes a callback function as an argument:

```
this.setState((prevState) => {
    return {
```

```
        counter: prevState.counter + 1,
    }
})
```

- This callback function takes in the parameter of *previous state*. This is guaranteed to be the previous state before setState is called.
- This function then does whatever logic you want to perform and returns an object representing all the state variables that have changed. In our case, we are just doing `prevState.counter + 1`

Component Lifecycle

- The component lifecycle describes the way a component is created, updated, and destroyed.
- It's kinda like the lifecycle of a human. We are born, we grow up, we die.
- The component lifecycle is similar. It has 3 phases:
  - Mounting (born)
  - Updating (grow up)
  - Unmounting (die)
- You can define a couple of functions that are *implicitly* called during these phases. These functions are called *lifecycle methods*.
- I'm gonna divide the lifecycle into these three phases and explain the lifecycle methods called in each phase.

Mounting

- Mounting is the phase where the component is created and added to the DOM.
- The methods that are called during this phase are:
  - `constructor()` - This is the first method that is called when the component is created. This is where we initialize the component.
  - `render()` - This is the method that is called to render the component. This is where we return the HTML that we want to render.
  - `componentDidMount()` - This is the method that is called after the component is added to the DOM. This is where we do things like API calls, etc.
- The important method here is `ComponentDidMount()`. This method is called after the component mounts, like the name suggests. We do things like API calls here to populate data in our component.
- Here's an example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
```

```
      data: null,
    };
  }

  componentDidMount() {
    // this is where we do our API calls
    fetch("https://jsonplaceholder.typicode.com/todos/1")
      .then((response) => response.json())
      .then((json) => this.setState({ data: json.title }));
  }

  render() {
    return (
      <div>
        <h1>Hello World!</h1>
        <p>{this.state.data}</p>
      </div>
    );
  }
}
```

- Here, we do an API call to get the title of a todo from the jsonplaceholder API. If you go to this link on your browser, you'll see a json that looks like this:

```
{
  "userId": 1,
  "id": 1,
  "title": "delectus aut autem",
  "completed": false
}
```

- This is how APIs talk usually, in JSON. We are setting the title property of this json into our `data` state variable. This will cause a rerender and the title will be displayed on the screen.
- To get the data from the API, we use the `fetch()` method provided by the browser. We use promise chaining to have a series of `then()` calls to extract the title from the returned JSON. You can refer to the previous chapter to know more about the `fetch()` method
- Once we have our JSON, We use the `setState()` method to change the state variable `data` to `json.title`. This will cause a rerender of the component.

- The text 'detectus aut autem' is the title of the todo that we got from the API.

There's also another method - `componentWillMount()`. This is executed on both the user's browser and the server you're hosting the website on just before the component mounts. This is not recommended to use because it's not consistent. It's better to use `componentDidMount()` instead.

Updating

- The updating phase is when the component is updated. This is when the state is changed.
- This is when our component *grows up* and has new state
- The methods that are called during this phase are:
    - `render()` - This is the method that is called to render the component. This function is called again and again when the contents on the screen have to be re rendered with new data
    - `componentDidUpdate()` - This is the method that is called after the component is updated. You can put anything you want here. Anything you want to run after every update of the component.

  There are also a couple other methods that have very descriptive names:
  `componentWillReceiveProps()`
    - When you have nested components, you usually pass props from the parent to the child component.
    - Now when data changes in the parent component, and you want the data in the child also to update, you just change the props given to the child in the parent.
    - For example,

```
// this is the child component
class Child extends React.Component {
    constructor(props) {
      super(props);
    }
    componentWillReceiveProps() {
        // logic for whatever you want to do when component
receives props goes here. Ima just console log for now
        console.log("Received props!");
    }
```

```
    render() {
        return <div>Name in props is: {this.props.name}</div>
    }
}
```

// this is the parent component
class Parent extends React.Component {
constructor(props) {
super(props);
this.state = {
name: "John Doe",
};
}

```
render() {
  return (
    <div>
      <h1>Hello {this.state.name}!</h1>
      <Child name={this.state.name} /> // child component that we
defined earlier is used here.
    </div>
  );
}
```

}

```
- Now when I change the state variable `name` in the parent, the
props to the child will also change.
- This is where the `componentWillReceiveProps()` method comes in.
We define this in the child component and it is called everytime
the props change. We can use this to update the state of the child
component.
- The child automatically rerenders on props change but if you
want to make other changes in the child in response to this props
change, you put that logic here.

### `shouldComponentUpdate()`

- This is a method that is called before every rerender of the
component. This is used to optimize the component.
```

- Sometimes when the states change, you might not want to rerender the component.
- A real world example of this is when you have a form and you want to validate the form before submitting it. You don't want to rerender the component everytime the user types a letter. You want to rerender it only when the user clicks the submit button.
- In `shouldComponentUpdate()`, you can compare the `nextState` with the current state and decide if you want to rerender or not.
- This function should return a true or false. You can define your logic here and return true or false accordingly.
- The function signature looks something like this:
- `shouldComponentUpdate(nextProps, nextState)`
- You can access the current state using `this.state` and the next state using `nextState`.
- current props using `this.props` and the next props using `nextProps`.
- Then you can make whatever logic you want, make comparisons between these two and return true or false. If you return true, the component will rerender. If you return false, it won't rerender.
- Here's an example:

```jsx
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      name: "John Doe",
      age: 69,
    };
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.state.name =/= nextState.name) {
      return true;
    } else {
      return true;
    }
  }

  render() {
    return (
      <div>
```

```
        <h1>Hello {this.state.name}!</h1>
        <button onClick={() => this.setState({ name: "John Smith"
})}>Change Name</button>
      </div>
    );
  }
}
```

- In this example, I'm only rerendering the component when the name changes. I'm not rerendering it when the age changes, since I'm not displaying the age on the screen.

`ComponentWillUpdate()` and `ComponentDidUpdate()`

- These two methods are called just before and just after an update is triggered.
- They are *not* called on the first render of the component, that's when the component is born.
- If `shouldComponentUpdate()` returns false, these two methods are not called since an update doesn't happen.
- You should not use the `setState()` method inside these two functions or you'd cause an infinite loop of updates. This is because these two functions are called everytime an update is triggered. If you change the state inside these two functions, you'd trigger another update, which would call these two functions again, which would trigger another update, and so on.
- I know these functions seem like they have no use in the real world. That's because you're right. They don't. Modern react, doesn't even use these functions. They are deprecated and we use *React Hooks*.

Unmounting

- This is the phase where the component is removed from the DOM.
- There's only one method that's called in this phase: `componentWillUnmount()`
- This method is called before the component dies.
- It's giving a chance for the component to say any last words before it dies.
- This is where we do things like clearing intervals, clearing timeouts, and other cleanup work.

Stateless Components and Functional Components

- These are big complicated names but they are super simple.
- All this time, we wrote components as classes. We extended the `React.Component` class and wrote a `render()` method.
- This is the old way of doing things. The new way of doing things is to write components as functions. Because functions are simple.

- A functional component is just a javascript function that returns some JSX (HTML). That's it.
- Here's an example:

```
function MyComponent() {
  return (
    <div>
      <h1>Hello World!</h1>
    </div>
  );
}
```

- The problem is, we can't use all those lifecycle methods that we just told about. We can't use states. We can't use any of that.
- BUT, we can do thoda cheating through `React Hooks`. More about that later.
- These functions are called stateless components because they don't have any state. *yet*
- Since these don't have state, they are called `stateless` components.
- You can access props in functional components though. props would just be a javascript object that is passed as a parameter into the function

```
function MyComponent(props) {
  return (
    <div>
      <h1>Hello {props.name}!</h1>
    </div>
  );
}

// you use this component like this
<MyComponent name="John Doe" />;
```

- This props object is often desctructured to access the props directly, like this:

```
function MyComponent({ name }) {
  return (
    <div>
      <h1>Hello {name}!</h1>
    </div>
```

```
  );
}
```

- You only expect to receive a JS object with one property(name) in it, so you destructure it to get that property directly.
- A table is given in the slides that shows the difference between stateful and stateless components:

Stateful Components	Stateless Components
You define it using a class only. Although you can use functions to make a stateful component using hooks	Only functions
Created by extending React.Component	Created by writing a function
It has state management and can change it's own state	It doesn't have any state
It's also called as *smart* components or *containers*	It's also called as *dumb* components or *presentational* components. Because they are dumb

React Hooks

- These let you use state and other react features in *functional components*, circumventing the need to use *class components*.
- Hooks are functions that let you “hook into” React state and lifecycle features from function components.
- Hooks don’t work inside classes — they let you use React without classes.
- We'd be learning 2 important hooks in this unit, `useState()` and `useEffect()`.

`useState()`

- This is the hook that lets you *use state* in functional components.
- That's why it's called `useState()`.
- Let's start with an example, and then we'll break it down:

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

function Counter() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  function handleClick() {
    setCount(count + 1);
  }
```

```
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>Click me</button>
    </div>
  );
}
```

- This example just implements a counter that increments on click of a button.
- The first new line is this:

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

- This is where we declare our state variable. We do it by calling the useState function from react.
- It takes in the initial value of count as a parameter. In this case, it's 0.
- It returns an array of 2 elements. The first element is the state variable itself, and the second element is a function that we use to change the state variable.
- You can call these two whatever you want but by convention, if the state variable is `x`, we call the function to change it `setx`, like `count` and `setCount`.
- The next new line is what we put inside the handleClick function:

```
setCount(count + 1);
```

- We used the `setCount()` function (this was returned to us from useState) to change the state variable. This will cause a rerender of the component.
- You see how we never used a class component in this entire process and it also makes our code more readable.
- The idea of hooks is a little hard to grasp at first, but once you get it, it's super simple.
- You can declare multiple state variables just like this:

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(18);
  const [fruit, setFruit] = useState("banana");
  const [todos, setTodos] = useState(["learn hooks", "write esa",
"get an S grade"]); // here we use an array as a state variable.
This is perfectly valid.
```

```
  // ...
}
```

`useEffect()`

- This hook lets you perform side effects or actions in response to changes in the state.
- It's like the `componentDidMount()` and `componentDidUpdate()` and `componentWillUnmount` lifecycle methods combined into one.
- Imagine you want to change the title of the page (the website name that's displayed on the browser tab) everytime the count changes from our counter, you'd use a useEffect hook like this:

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

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

  function handleClick() {
    setCount(count + 1);
  }

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>Click me</button>
    </div>
  );
}
```

- The `useEffect()` hook takes in a function as a parameter. This function is called everytime the component is rendered, including the first render.
- `useEffect()` also accepts a second parameter, which is an array of dependencies. This is an optional parameter.
- Currently, this function is called everytime an update happens. We want this piece of code to run only when the count changes.

- Running it on every update would be a waste of resources.
- We can do this by passing in the count as a dependency to the `useEffect()` hook like this:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // this is the dependency array, [count]
```

- Now, this function will only run when the count changes.
- If you want to run it only on the first render, you can pass in an empty array as the second parameter:

```
useEffect(() => {
  // this will run only on the first render
  // generally used to do API calls
}, []); // this is the dependency array, [] (empty array)
```

Here's a summary:

- No dependency array passed => runs on every render
- Empty dependency array passed => runs only on first render
- Dependency array with some values passed => runs only when those values change
- The function you pass into `useEffect()` can also return a function.
- This function that you return is called when the component is unmounted. This is similar to the `componentWillUnmount()`.
- Here's where you can do cleanup work like clearing intervals, timeouts, etc.

```
useEffect(() => {
  let interval = setInterval(() => {
    // do something
  }, 1000); // runs every second
  return () => {
    clearInterval(interval); // this is called when the component
is unmounted, you gotta clear the interval here
  };
}, []); // this is the dependency array, [] (empty array) so it's
only run once on the first render
```

- Notice how since we used an empty dependency array. This avoids the infinite loop of updates that we talked about earlier.

Info

In all of these examples, I've used the `import React, { useState, useEffect } from "react";` syntax. This is the modern way of importing react.
You can also use the old way of importing react like this: `import React from "react";` and then use the hooks like this: `React.useState()` and `React.useEffect()`.
This is the method given in the slides but it's deprecated. I'd recommend using the modern way of importing react, but don't get thrown off if you see the old way in the question paper

Refs

- Refs are a way to access DOM elements directly from react.
- It's a substitute for using `document.getElementById()` and other DOM methods.
- Since your react app is just a bunch of javascript, you can't use DOM methods directly. You need to use refs.
- Here's how you create a ref:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.divRef = React.createRef();
  }
  render() {
    return <div ref={this.divRef}>Hello!</div>;
  }
}
```

- Here, we created a ref called `divRef` and assigned it to the div using the `ref` attribute.
- Now, we can access this div using `this.divRef.current`. This is the DOM element.
- You can use this to do things like focus on an input element, chage it's contents, etc. You can do anything that you could do with normal DOM methods.
- Here's an example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.divRef = React.createRef();
    this.handleClick = this.handleClick.bind(this); // we have to
bind the handleClick function to use the this context of the
component instance. Since handleClick is a common class method,
it'd be using the this context of the class instead of the
instance of the class when it's called. Please refer to the
previous chapter for what 'this' means
  }

  handleClick() {
    const node = this.divRef.current;
    // now we have our div element in the node variable
    node.textContent = "Hello! this new content was added using
refs";
  }

  render() {
    return (
      <div>
        <div ref={this.divRef}>Hello!</div>
        <button onClick={this.handleClick}>Change Text</button>
      </div>
    );
  }
}
```

- You can use the `useRef()` hook to achieve the same thing in a functional component:

```
import React, { useRef } from "react";
function MyComponent() {
  const divRef = useRef(null);

  function handleClick() {
    const node = divRef.current;
    // now we have our div element in the node variable
    node.textContent = "Hello! this new content was added using
refs";
  }

  return (
    <div>
```

```
      <div ref={divRef}>Hello!</div>
      <button onClick={handleClick}>Change Text</button>
    </div>
  );
}
```

- Notice how we used the `useRef()` hook to create a ref. We passed in `null` as the initial value. This is because we don't have a DOM element yet. We'll get it when the component renders.
- We can access the DOM element using `divRef.current` just like we did in the class component.

Callback Refs

- You can use a callback function to create a ref.
- This allows for more fine grained control over what element the ref refers to.
- Here's an example:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);

    // here's the new addition
    this.divRef = null;
    // this is callback function that we pass to the ref attribute
    this.setDivRef = (element) => {
      // an arrow function that accepts the element as a parameter
      this.divRef = element; // set the divRef to the element
    };

    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    const node = this.divRef.current;
    // now we have our div element in the node variable
    node.textContent = "Hello! this new content was added using
refs";
  }

  render() {
    return (
      <div>
```

```
        <div ref={this.setDivRef}>Hello!</div>
        <button onClick={this.handleClick}>Change Text</button>
      </div>
    );
  }
}
```

- The `ref` property we pass into the div this time is a function, not a ref created by `React.createRef()`.
- This function is called when the component is rendered. It is passed the DOM element as a parameter, in this case, the div is passed into setDivRef.
- This function then sets the `divRef` variable. That's why it's called `setDivRef()`.
- Now, we can access the div using `this.divRef.current` just like we did in the previous example.

Keys

- Imagine you're making a todo app. This component has a list of todos that you display on the screen.
- This list of todo's might come from an API that connects to your database, but for now, let's define it as an array in the component itself.
- It's not practical to statically define JSX for each todo in the array. We need to dynamically generate the JSX for each todo.

The `map()` method

- The `map()` method is a method that is available on arrays. It takes in a function as a parameter and returns a new array.
- This function is called on each element of the array and the return value of this function is added to the new array.
- Here's an example:

JS

```
const arr = [1, 2, 3, 4, 5];
const newArr = arr.map((element) => element * 2); // we pass an
arrow function that multiplies each element by 2
```

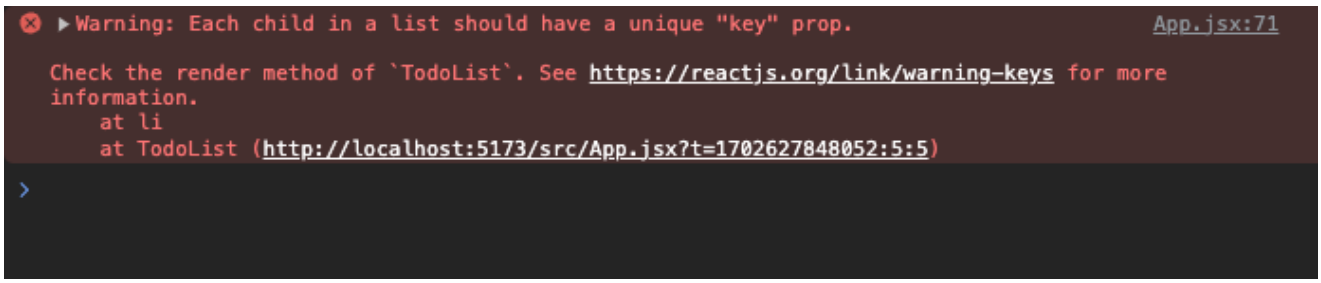
- The `newArr` variable will now be `[2, 4, 6, 8, 10]`.
- We will now use this to dynamically generate JSX for each todo in our todo list.
- Here's the code:

```
class TodoList extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.state = {
      todos: [
        { id: 1, text: "Learn React" },
        { id: 2, text: "Get an S Grade" },
        { id: 3, text: "Profit" },
      ],
    };

    this.todosJSX = this.state.todos.map((todo) => <li>{todo.text}
</li>); // we make a new array that'll have an array of JSX
elements.
    // we can then directly insert this into our JSX we return
  }

  render() {
    return (
      <div>
        <h1>Todo List</h1>
        <ul>{this.todosJSX}</ul>
      </div>
    );
  }
}
```

Here's the output of the above code:

- Now, this works but if you open up the console, you'd see React screaming about something:

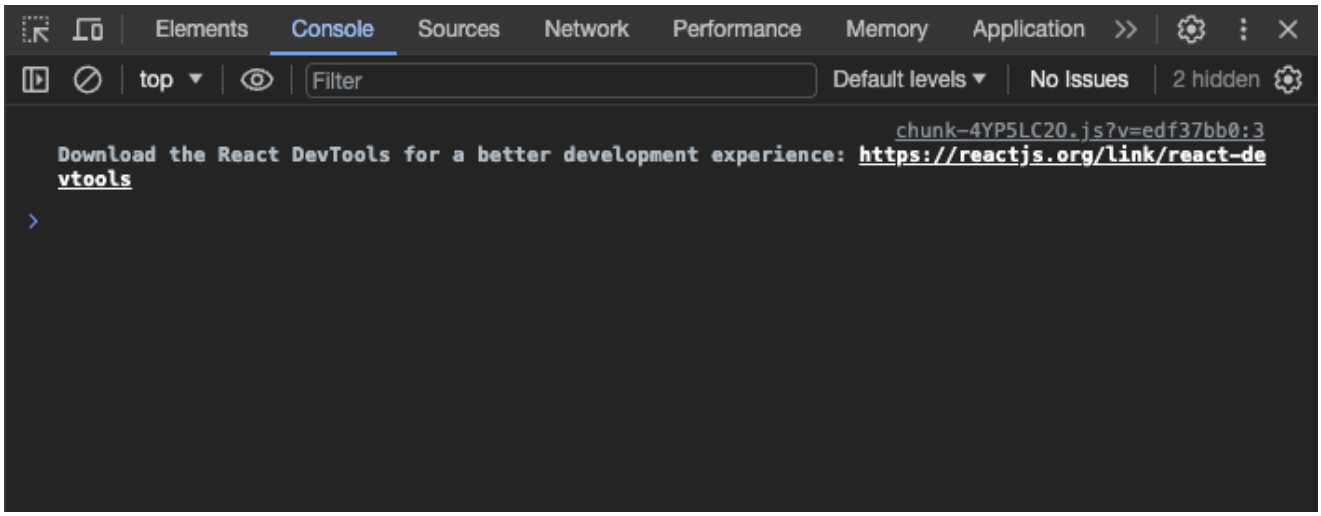


- This is because we didn't define a `key` attribute for each todo.
- React uses this `key` attribute to identify each element in the list.
- This is important because if we change the todos, React will not know which todo is which and it'll have to rerender the entire list.
- This is not efficient. We want React to only rerender the todos that changed.

Here's the new code with the key attribute:

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      todos: [
        { id: 1, text: "Learn React" },
        { id: 2, text: "Get an S Grade" },
        { id: 3, text: "Profit" },
      ],
    };

    this.todosJSX = this.state.todos.map((todo) => <li key=
{todo.id}>{todo.text}</li>); // we make a new array that'll have
an array of JSX elements.
    // we can then directly insert this into our JSX we return
  }

  render() {
    return (
      <div>
        <h1>Todo List</h1>
        <ul>{this.todosJSX}</ul>
      </div>
    );
  }
}
```

Now React is happy. Here's the console:



- Now, if we change the todos, React will know which todo is which and it'll only rerender the todos that changed.

Frontend is Done, No More React, Now We Do Backend!

NodeJS

- All of the javascript we've written until now, it's been running on the browser.
- The browser has a javascript engine that runs our javascript code.
- This is great for making websites, but what if we want to make a backend for our website too in javascript?
- This is where NodeJS comes in.
- NodeJS is *not* a programming language. It's just a javascript engine, just like an interpretor for python or java.
- It's a javascript runtime environment that lets you run javascript code on your computer directly, without a browser.
- NodeJS is built on top of the V8 engine. That's the engine under the hood of google chrome.
- NodeJS is written in C++ and it's open source. This just means that you can see the source code of NodeJS and you can contribute to it if you want.
- If you like javascript, this is a great thing because you can write both your frontend and backend in the same language.
- If not, like me, you just have to shove it down your throat because PES decided it was a good idea.
- Infact the creator of NodeJS himself apologised publicly for creating it- here's the link

> Dahl apologized to his audience, with a pained look, for decisions that he said "cannot be corrected now, because there's just so much software that uses it."

(Okay he didn't apologise for creating it but he apologised for the state it is in right now)

- NodeJS also provides us with a bunch of modules that we can use to make our lives easier. These are like libraries we import in other languages.

The Nature Of NodeJS

- NodeJS is *non blocking* by default.
- This just means that almost all of the functions that NodeJS uses are asynchronous.
- Since JS is a single threaded language, this is important. If we had blocking functions, the entire program would halt until the function is completed. This would waste CPU cycles.
- Instead, NodeJS will make a request (like reading a file) and then pause and it is free again to do other stuff. When the request is completed, it'll handle it accordingly.
- This is called *event driven* programming. This is the nature of NodeJS.
- This is also important when your server has multiple requests coming in.
- Imagine you have a server that's responsible for returning some data from the database. In a blocking paradigm, the server would send the database request, wait for it to come back, and then send it back to the client as a response.
- In NodeJS, it just sends a request to the database, and then it's free to handle other requests. When the database request comes back, it'll send it back to the client as a response.
- This increases the capacity of our server to handle requests.

Setting Up NodeJS

- idk if this is in syllabus but it is given in the slides so I'm gonna cover it.
- You just gotta download NodeJS for your system from here.
- Then you can run `node -v` in your terminal to check if it's installed properly.
- You can also run `npm -v` to check if npm is installed properly. npm is the package manager for NodeJS. It's like pip for python or apt for ubuntu, pacman for Arch, etc.
- You can use npm to install packages that you can use in your NodeJS project.
- Then you create a file with a `.js` extension. Let's call ours main.js.

```JS
// main.js

console.log("Hello World!");
```

- Then you can run `node main.js` in your terminal to run the file.

Starting a New NodeJS Project

1. Make a folder for your project

2. Open a terminal in that folder
3. Run `npm init` and follow the prompts on the screen
4. This will create a `package.json` file in your folder. This is where all the dependencies of your project are stored.

> **Info**
>
> - Unlike package managers like pip, where you have to memorise what dependencies your project relies on and what stuff you have to pip install, npm stores all of this in the `package.json` file.
> - This is great because you can just copy your project folder to another computer and run `npm install` and it'll install all the dependencies for you.
> - `package.json` also contains other information about your project like the name, version, author, etc.

Inbuilt Modules

- NodeJS comes with a bunch of inbuilt modules that you can use in your project.
- You can find the documentation for all of them here
- You can use some of these modules directly and some of them you have to import.

Timer Functions

- NodeJS has a bunch of timer functions that you can use to do things like run a function after a certain amount of time, make a function run every second etc.
- You might have seen some of these being used in the previous sections as well.

`setTimeout()`

- This is a function that takes in a function as a parameter and a time in milliseconds.
- It runs the function after the specified time.
- Here's an example:

```JS
setTimeout(() => {
  console.log("Hello World!");
}, 1000); // this will print hello world after 1 second
```

- For some reason, if you want to cancel this happening before the 1 second elapses, you can do something like this:

```JS
const timeout = setTimeout(() => {
  console.log("Hello World!");
}, 1000); // this will print hello world after 1 second

clearTimeout(timeout); // this will cancel the timeout
```

- You can use `.ref()` and `.unref()` to decrease and increase the reference count of a timeout.
- If the reference count is 0, the timeout will be cancelled.
- Decreasing the reference count tells the event loop that this timeout is not important and it can be cancelled if needed.
- What an *event loop* is is explained in the next chapter, idk why they put this topic here.
- Here's an example:

```JS
const timeout = setTimeout(() => {
  console.log("Hello World!");
}, 1000); // this will print hello world after 1 second

timeout.unref(); // this will decrease the reference count
timeout.ref(); // this will increase the reference count
```

`setInterval()`

- The syntax is the same as `setTimeout()` but this function runs the function every time the specified time elapses.
- Here's an example:

```JS
let interval = setInterval(() => {
  console.log("Hello World!");
}, 1000); // this will print hello world every second

// if you want to cancel this:
clearInterval(interval); // this will cancel the interval
```

`setImmediate()`

- This is a function that takes in a function as a parameter and runs it immediately after the current function is done executing.
- Here's an example:

```js
console.log("Hello World!");
setImmediate(() => {
  console.log("Hello World!");
}); // this will print hello world immediately after the first hello world
// this is used for scheduling a function to run after the current function is done executing

// to cancel this:
clearImmediate(); // this will cancel the immediate
```

`npm` And Installing Modules

- `npm` is the package manager for NodeJS. It's used to install/remove packages
- To install a package, run `npm install <package-name>` in a terminal
- To remove a package, run `npm uninstall <package-name>` in a terminal
- As an example in the slides, they have installed and used the `validator` module. So let's go ahead with that.
- The `validator` module is used to validate things like emails, urls, etc.

> **Info**
>
> We used `npm` before to create a react app using vite. You can also use `create-react-app`.
>
> - When you ran `npm run dev`, it starts a development *server* on your system itself. Then you can visit `localhost:3000` or `localhost:5173` to see your app running.
> - This uses NodeJS to make a server on your system itself.
>
> You could also use the traditional HTML pages approach to make a react app but nobody does it that way anymore. It's too much work.

The Flow Of Installing And Using A Module

- Let's say you want to use the `validator` module in your project.
- You'd first run `npm install validator` in your terminal. This will install the module in your project.
- After running this, if you check your `package.json`, you'd see that the `validator` module is added as a dependency.

- You can check the version installed by running `npm version validator` in your terminal.
- Then you'd import it in your project with the `require` function like this:

```js
const validator = require("validator");
```

- Now you can use the functions in the `validator` module like this:

```js
const validator = require("validator");
const email = "johndoe@example.com";
console.log(validator.isEmail(email)); // this will print true if the email is valid and false if it's not

let string = "HELLO";
console.log(validator.isLowercase(string)); // false, pretty self explatory
console.log(validator.isUppercase(string)); // true

string = " ";
console.log(validator.isEmpty(string)); // true

// There are other functions available like isBoolean(), isCurrency(), isJSON(), isDecimal(), etc.
// you can't possibly remember the functions of all the modules you use, so you can always refer to the documentation
```

`require()` And `module.exports`

- This is fun and all but how do you make your *own* modules?
- You'd want to make your own module because you'd have multiple people working on your project. Putting functions in different files is a good idea then. You can then import these functions in your main file.
- To create a module, just add it the `exports` object in your module file.
- The `exports` object is an object that is available in every module. You can add functions to this object and then import them in other files.
- Here's an example:

```js
// myModule.js

exports.hello = () => {
  console.log("Hello World!");
```

```
};

// you can add multiple exports too. Just add them to the exports
object

exports.add = (x, y) => {
  return x + y;
};

exports.sub(x, y) => {
    return x - y;
}
```

- Now I can import this in my main file like this:

```JS
// main.js

const myModule = require("./myModule.js"); // this is how you
import a module. The string inside require is the path to the
module file
myModule.hello(); // this will print hello world
console.log(myModule.add(1, 2)); // 3
console.log(myModule.sub(1, 2)); // -1
```

- You run this JS file using `node main.js` in your terminal. (assuming your main file is called `main.js`)

The `fs` Module

- `fs` stands for file system. It lets you read and write to files.
- It's an inbuilt module in NodeJS, so you don't have to install it. However, you have to import it.
- You import it the usual way: `const fs = require("fs");`

First We Open The File

```JS
const fs = require("fs");

fs.open("file.txt", "r", (err, fd) => {
  // this is the callback function that's called when the file is
opened
  // err is the error, if any. it'll be null if no error.
```

```
  // fd is the file descriptor. This is used to read/write to the
file
  // r is the mode. r means read. You can also use w for write, a
for append, the standard stuff, etc.

  // gotta close it after we're done
  fs.close(fd, (err) => {
    // this is the callback function that's called when the file
is closed
  });
});
```

- Observe how everything is `callback` based in NodeJS. The callback is called when the file is opened. It doesn't block the main thread waiting for the file to open.
- This is the non blocking nature of NodeJS that we talked about earlier.
- `fs.open` also takes an optional `mode` parameter. This is used when the file is created. The `mode` parameter is a number that specifies the permissions of the file. This is related to unix. Just google what these file permissions mean, they are super simple
- Here's an example:

```JS
const fs = require("fs");

fs.open("file.txt", "w", 0o666, (err, fd) => {
  // 0o666 is the mode. This is the permissions of the file. This
is a number. 0o666 is the default mode. it means read and write
permissions for everyone on the system
  // 0o means that it's an octal value, just like 0x means it's a
hexadecimal value
});
```

Reading From a File

- Now that we have the file descriptor, we can read from the file.
- We use the `fs.read()` function to read from the file.
- Here's the syntax:

```JS
// fs.read(fd, buffer, offset, length, position, callback);
// fd is the file descriptor returned by fs.open()
// buffer is the buffer that we read into. We'll talk about
buffers later
```

```
// offset is the offset in the buffer to start writing at
// length is the number of bytes to read
// position is the position in the file to start reading from
// callback is the callback function that's called when the read
is done

let buffer = Buffer.alloc(1024); // this is the buffer that we
read into. It's a buffer of size 1024 bytes. We'll talk about
buffers later
fs.open("text.txt", "r", (err, fd) => {
  fs.read(fd, buffer, 0, 1024, 0, (err, bytesRead, buffer) => {
    // err is the error, if any. it'll be null if no error.
    // bytesRead is the number of bytes read
    // buffer is the buffer that we read into
  });
});
```

- If this sounds too complicated to read a file, you're right. It is. That's why we have the `fs.readFile()` function.
- It reads directly from a file without using all the `fd` and `buffer` stuff.

JS
```
fs.readFile("text.txt", (err, data) => {
  // err is the error, if any. it'll be null if no error.
  // data is the data read from the file. It's a buffer
  // you can convert it to a string using data.toString()
});
```

- You can also specify the encoding of the file in the second parameter. The default is `utf-8`.

JS
```
fs.readFile("text.txt", "utf-16", (err, data) => {
  // err is the error, if any. it'll be null if no error.
  // data is the data read from the file. It's a string
});
```

Writing To a File

- We use the `fs.write()` function to write to a file.
- Here's the syntax:

```JS
// fs.write(fd, buffer, offset, length, position, callback);
// fd is the file descriptor returned by fs.open()
// buffer is the buffer that we write from.
// offset is the offset in the buffer to start reading at
// length is the number of bytes to write
// position is the position in the file to start writing from
// callback is the callback function that's called when the write
is done

let buffer = Buffer.from("Hello World!"); // this is the buffer
that we write from. It has "Hello World!" in it
fs.open("text.txt", "w", (err, fd) => {
  fs.write(fd, buffer, 0, 12, 0, (err, bytesWritten, buffer) => {
    // called after writing is done
    // err is the error, if any. it'll be null if no error.
    // bytesWritten is the number of bytes written
    // buffer is the buffer that we wrote from
  });
});
```

- Again, this is too complicated. We have the `fs.writeFile()` function for this.

```JS
fs.writeFile("text.txt", "Hello World!", (err) => {
  // called after writing is done
  // err is the error, if any. it'll be null if no error.
});
```

- You can also use an options object to send in options about the file like the encoding, mode, etc.

```JS
fs.writeFile("text.txt", "Hello World!", { encoding: "utf-16" },
(err) => {
  // called after writing is done
  // err is the error, if any. it'll be null if no error.
});
```

- Note that this `options object` can also be passed into the `fs.readFile()` function.

Deleting a File

- You can use the `fs.unlink()` function to delete a file.
- Here's the syntax:

```JS
fs.unlink("text.txt", (err) => {
  // called after deleting is done
  // err is the error, if any. it'll be null if no error.
});
```

Truncating A File

- You can use the `fs.truncate()` function to truncate a file.
- Truncating a file means that you can change the size of the file.
- Here's the syntax:

```JS
// we truncate the file to 10 bytes. This means that the file will be 10 bytes long after this. Only the first 10 bytes will be kept.
// the default is 0, which means that the file will be empty after this
fs.truncate("text.txt", 10, (err) => {
  // called after truncating is done
  // err is the error, if any. it'll be null if no error.
});
```

Synchronous Vs Asynchronous

- Every `fs` method we talked about until now has a synchronous counterpart.
- The synchronous counterpart is the same function name but with `Sync` appended to it.
- For example, `fs.readFile()` has a synchronous counterpart called `fs.readFileSync()`.
- The synchronous functions are blocking. This means that the main thread will wait for the function to complete before moving on.
- This is not good because it'll block the main thread and waste CPU cycles.
- You should always use the asynchronous functions.

Other Methods

- There are a bunch of other methods in the `fs` module that you can use.
- You can refer to the documentation here for more info.
- A table is given in the slides about them:

Method Name	Description
fs.rename(oldPath, newPath, callback)	Renames a file
fs.chown(path, uid, gid, callback)	Changes the owner of a file
fs.stat(path, callback)	Returns the statistics of a file
fs.link(srcPath, destPath, callback)	Creates a new link to a file
fs.symlink(srcPath, destPath, type, callback)	Creates a new symbolic link to a file
fs.rmdir(path, callback)	Removes a directory
fs.mkdir(path, mode, callback)	Creates a directory
fs.readdir(path, callback)	Reads the contents of a directory
fs.utimes(path, atime, mtime, callback)	Changes the timestamps of a file
fs.exists(path, callback)	Checks if a file exists
fs.access(path, mode, callback)	Checks permissions for a file
fs.appendFile(path, data, options, callback)	Appends data to a file

Buffers

- Buffers are temporary storage areas in memory.
- They are used to store data while it's being transferred from one place to another, like when it's being read from a file or being sent over a network.
- Buffers are used because they are faster than arrays.
- Buffers are used to store raw binary data.
- Buffers are instances of the `Buffer` class.

Creating Buffers

- You can create a Buffer using `Buffer.alloc()`

```JS
// Buffer.alloc(size[, fill[, encoding]]);
// size is the size of the buffer in bytes
// fill is the value to fill the buffer with. The default is 0
// encoding is the encoding to use. The default is utf-8
// The [] just mean that the parameter is optional
const buf1 = Buffer.alloc(1024); // this creates a buffer of size 1024 bytes
```

- You can also use `Buffer.allocUnsafe()` to create a buffer. This is faster but it's unsafe because it might contain data written by other programs previously

Writing To Buffers

- You can write to buffers using the `write()` method.

- Here's the syntax:

```JS
// buffer.write(string[, offset[, length]][, encoding]);
// string is the string to write to the buffer
// offset is the offset in the buffer to start writing at
// length is the number of bytes to write
// encoding is the encoding to use. The default is utf-8
// it returns the number of bytes written

buf1.write("Hello"); // this will write "Hello" to the buffer
```

> **Important**
>
> If there is not enough space in the buffer to write the string, it'll be truncated. Only the part that fits will be written.
> If there is more space than required, and you convert the string to a buffer later, the string will just repeat itself in the buffer.

Reading From Buffers

- You can read from buffers using the `toString()` method.
- Here's the syntax:

```JS
// buffer.toString([encoding[, start[, end]]]);
// encoding is the encoding to use. The default is utf-8
// start is the start index to read from
// end is the end index to read to
// it returns the string read from the buffer

// !Unintuitive behaviour alert!
console.log(buf1.toString()); // this will print
"HelloHelloHelloHelloHello...around 205 times"
// this is because because the buffer is 1024 bytes long and the
string is 5 bytes long
```

Comparing 2 Buffers

- You can compare 2 buffers using the `compare()` method.
- Here's the syntax:

```JS
// buffer.compare(otherBuffer, [targetStart], [targetEnd], [sourceStart], [sourceEnd]);
// otherBuffer is the buffer to compare to
// targetStart is the start index of the target buffer to compare
// targetEnd is the end index of the target buffer to compare
// sourceStart is the start index of the source buffer to compare
// sourceEnd is the end index of the source buffer to compare
// it returns a number indicating the difference between the 2 buffers

const buf2 = Buffer.from("Hello"); // you can use Buffer.from to just convert a string to a buffer directly

console.log(buf1.compare(buf2)); // this will print 0 because the 2 buffers are the same
// it's like strcmp in C
// it returns 1 if buf1 is greater than than buf2
// it returns -1 if buf1 is less than than buf2
// It's essentially doing buf1 - buf2
```

Copying Buffers

- You can copy a buffer using the `copy()` method.
- Here's the syntax:

```JS
// buffer.copy(targetBuffer[, targetStart[, sourceStart[, sourceEnd]]]);
// targetBuffer is the buffer to copy to
// targetStart is the start index of the target buffer to copy to
// sourceStart is the start index of the source buffer to copy from
// sourceEnd is the end index of the source buffer to copy from
// it returns the number of bytes copied

const buf3 = Buffer.alloc(1024);
buf1.copy(buf3); // this will copy buf1 to buf3
```

Streams

- Streams are used to read and write data in chunks.

- They are used to read and write data in chunks because it's faster than reading and writing the entire file at once into your RAM.
- This makes streams useful when you're dealing with large files - imagine Gigabytes of Data. You can't read all of that into your RAM at once. You have like 16GB of RAM-
- So streams provide 2 advantages:
    1. Memory: You don't have to read the entire file into your RAM at once
    2. Time: You can start processing the data as soon as you get it. You don't have to wait for the entire file to be read.

Types Of Streams

- There are 4 types of streams:

    1. Readable: Used for reading data
    2. Writable: Used for writing data
    3. Duplex: Used for both reading and writing data
    4. Transform: Used when you're reading and writing data, while you're transforming it in some way. For example, file compression: you'd be reading the file and writing the compressed data to it.

  Examples Of Readable and Writable Streams:

Readable Streams	Writable Streams
HTTP Responses, being received, usually on the browser	HTTP Requests being sent, usuelly on the server
Reading a file	Writing to a file
TCP Sockets, when they are being received, usually on the server	TCP Sockets, when they are being sent, usually on the browser
STDIN, when it's being received, usually on the terminal	STDOUT, when it's being sent, usually on the terminal
HTTP Requests, when they are being received, usually on server	HTTP Responses, being sent, usually on the browser

Stream As Events

- Streams are also event emitters. They emit events when they are opened, closed, data is available, etc.
- You can listen to these events and perfrom actions accordingly.
- Some of the commonly used events are:
    1. `data`: This event is fired when there is data available to read
    2. `end`: This event is fired when there is no more data to read
    3. `error`: This event is fired when there is an error

4. `finish`: This event is fired when all the data has been flushed to the underlying system

- For example:

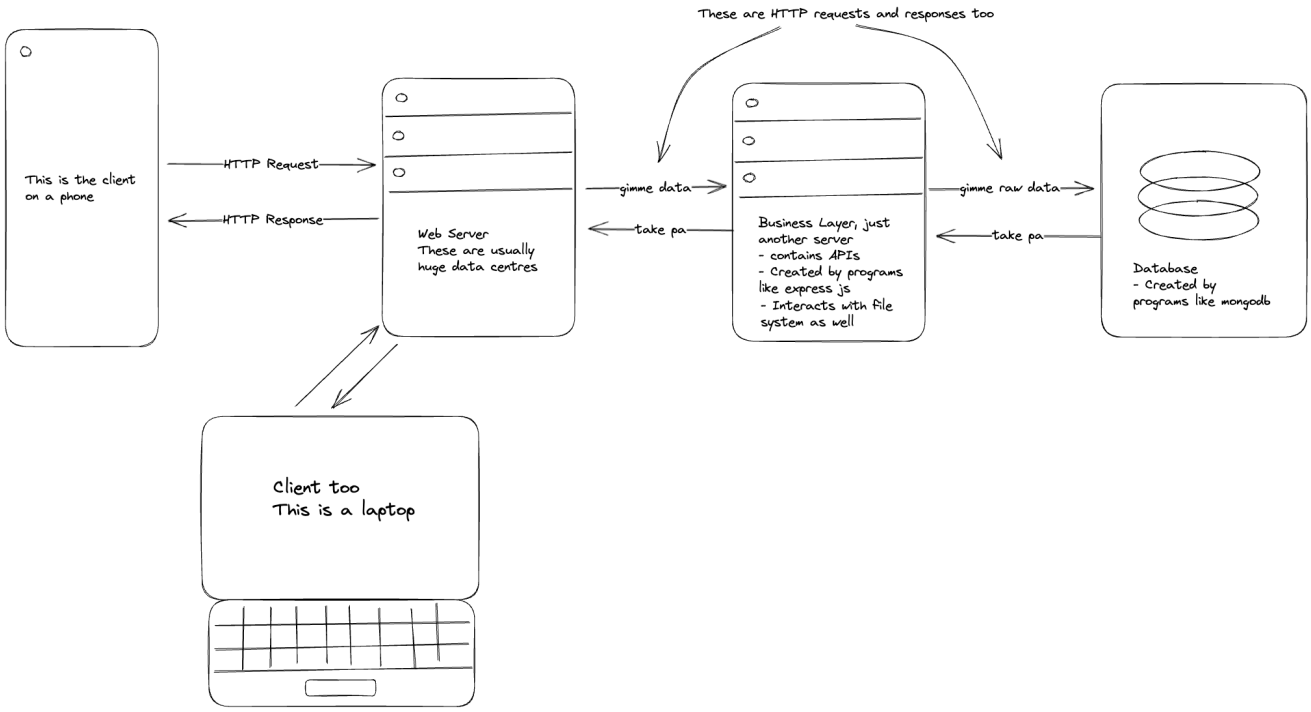
```js
const fs = require("fs");

let data = ""; // this is where we'll store the data
const readStream = fs.createReadStream("input.txt"); // this
creates a readable stream
readStream.setEncoding("utf-8"); // this sets the encoding to utf-
8

readStream.on("data", (chunk) => {
  // this is called when there is data available to read
  // chunk is the data that's available to read
  data += chunk; // we just append the data to our variable
});

readStream.on("end", () => {
  // this is called when there is no more data to read
  console.log(data); // this will print the entire file
});

readStream.on("error", (err) => {
  // this is called when there is an error
  console.log(err); // this will print the error
});
```

Creating A Basic HTTP Server Of Your Own

- Remember when we did `npm run dev` or `npm run start`, it created a server on our system itself? It uses this module called `http` to do that under the hood
- We can do that too. We can create a server on our system itself.
- Taking a step back and trying to get a clear picture, here's how a web app works usually:

1. There's a client, or user who's accessing your website on a browser
2. The client accesses your website that is hosted on a server. This is the server that we're going to create.
3. This server might have to talk to a database to get some data. This is where the `mongodb` in the next chapter comes in.

4. You might also have a `business logic` in between the backend and the database. This is where the `express` module in the next chapter comes in. You can also make files and put other logic in here.
5. Finally server then sends the data to the client as a response.



- We're going to create the server in step 2.
- Here's how you do it:

```JS
// main.js
const http = require("http"); // import http module

const server = http.createServer((req, res) => {
  // this creates a server
  // req is the request object
  // res is the response object
  // this is the callback function that's called when a request is made to the server
  // this is where you write your logic
  reseponse.writeHead(200, { "Content-Type": "text/plain" }); // this is the header of the response. It's a status code and some metadata
  response.end("Hello World!"); // this is the body of the response. It's the data that you send back to the client
});

// now we make the server listen on port 3000
```

```
server.listen(3000);
console.log("Server listening on port 3000");
```

- Now if you run this file using `node main.js` and visit `localhost:3000` in your browser, you'll see `Hello World!` on the screen.

The `url` Module

- This topic should have been up there with the other modules but it's given here so I'm gonna cover it here.
- The `url` module is used to parse URLs.
- It's an inbuilt module in NodeJS, so you don't have to install it. However, you have to import it.
- You import it the usual way: `const url = require("url");`
- Then you parse a URL using the `parse()` method.
- Here's an example:

```JS
const url = require("url");
var url1 = url.parse("http://localhost:3000/hello?
name=John&age=20"); // this is an example url with query
parameters of name and age. Query parameters have been explained
in the previous chapters
console.log(url1.host); // this will print localhost:3000
console.log(url1.pathname); // this will print /hello
console.log(url1.search); // this will print ?name=John&age=20
console.log(url1.query); // this will return an object with the
query parameters as keys and their values as values
// in this case, it'll be:
// {
//     name: "John",
//     age: "20"
// }
// you can also assign it to a variable and access the individual
query parameters like this:
const query = url1.query;
console.log(query.name); // John
```

Web Client Using `http`

- You can also act like a client and make requests to other websites using the `http` module.
- This is useful when you're making a web scraper or something.

- Here's an example:

```JS
const http = require("http");
let options = {
  host: "http.badssl.com", // let's scrape this webpage
  // can give additional info here like port, path, etc.
};

let callbackFunction = (response) => {
  // this is the callback function that's called when the response
is received
  let data = "";
  response.on("data", (chunk) => {
    // this is called when there is data available to read
    // chunk is the data that's available to read
    // This is the stream we were talking about earlier
    data += chunk; // we just append the data to our variable
  });

  response.on("end", () => {
    // this is called when there is no more data to read, aka the
request is over
    console.log(data); // this will print the entire file
  });

  response.on("error", (err) => {
    // this is called when there is an error
    console.log(err); // this will print the error
  });
};

let request = http.request(options, callbackFunction); // this
makes the request. You could have defined the options and callback
function here itself but I've defined it separately for clarity
request.end(); // this is important. This is what actually makes
the request. If you don't do this, the request will never be made
```

It prints a bunch of HTML form the website. Here's the output:

```
<!DOCTYPE html>
<html>
<head>
```

```
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-
scale=1">
  <link rel="shortcut icon" href="/icons/favicon-red.ico"/>
  <link rel="apple-touch-icon" href="/icons/icon-red.png"/>
  <title>http.badssl.com</title>
  <link rel="stylesheet" href="/style.css">
  <style>body { background: red; }</style>
</head>
<body>
<div id="content">
  <h1 style="font-size: 8vw;">
    http.badssl.com
  </h1>
</div>

</body>
</html>
```

- You won't be using the `http` module to make requests in real life, you'd rather be using the `fetch()` function. This was given in the slides and that's why I'm putting it here
- istg, PES is soo good at teaching shit that's useless stuff

Edit: Readline Module

- This module is not there in the slides and wasn't covered in my notes until now.
- But it has been asked in the ISA (was out of syllabus but pes is not gonna do anything about it, obviously-)
- I also saw it in one of the previous year question ESA question papers so I'm gonna write about it here just in case it comes in the exam
- The readline module is used to take input streams from anywhere, including the `stdin`.
- Here's how you import the `readline` module:

```
JS
const readline = require("readline");
```

- It's a module built into JS so you don't need to `npm install` it.
- Next, you need to create an interface with the readline module:

```
JS
const readline_interface = readline.createInterface(process.stdin,
```

```
process.stdout);
```

- The `createInterface()` method takes in two parameters, the input stream and the output stream
- Here, the input and output stream happen to be the `stdin` and `stdout` (we get these from the process object, this is a global object containing information about our program's process)
- Next, we can use the syntax of stream events to read from this interface we created (See above section on *Streams as Events*):

```JS
readline_interface.on("line", () => { // refer the above section on 'streams as events' for the signature of this function
    console.log(data); // this will be contents of one line. This function is called again and again for every line
});
```

- The callback function passed to the `on()` method would be called everytime there's data from the specified input, in this case, `stdin`.
- Once we get it, we can do anything with it. Here, I'm just console logging it.
- You can also ask the user a question on the terminal by doing something like this:

```JS
readline_interface.question("What's your name? ", (name) => {
    console.log("Name received is: ", + name);
});
```

- The input received from the user will be passed to the callback function, and we can do anything with it.
- The answer to the ISA question asked would be something like this:

```JS
// read from stdin.
// write to stdout.
// stdin will contain a bunch of lines that are comma separated
// print only the odd words
// for example,
// input:
// hello, pes, is, a, great, college
// there, couldn't, be, a, worse, lie
// output:
// pes a college
// couldn't a lie
```

```
const readline = require("readline");


const interface = readline.createInterface(process.stdin,
process.stdout); // interface with stdin and stdout

interface.on("line", (line) => {
    // on every line, we need to console log the odd words
    const all_words = line.split(',');
    const odd_words = all_words.filter((word, index) => {
        return index % 2; // returns true if the index is odd
    })
    odd_words.map((word) => {
        process.stdout.write(word + " "); // this does the same
thing as console.log but without a new line after each word.
    })
    console.log(); // to get a new line character
})
```

Note

The input and output streams could be any stream, even a file. You can create an interface with the input being the file and output being `stdout` like this:
`const interface = readline.createInterface(fs.createReadStream("input.txt"), process.stdout)`

Made with ♡ by Anurag Rao