Frontend Programming And ReactJS

Brief history of Web Applications

- Initially: static HTML files only with HTML forms for input
- Common Gateway Interface (CGI)
  - Certain URLs map to executable programs that generate web page
  - Program exits after Web page complete
  - Introduced the notion of stateless servers: each request independent, no state carried over from previous requests. (Made scale-out architectures easier)
  - Perl typically used for writing CGI programs

First-generation web app frameworks

Examples: (PHP, ASP.net, Java servlets)

- Incorporate language runtime system directly into Web server
- **Templates**: mix code and HTML - HTML/CSS describes view
- Web-specific library packages:
  - URL handling
  - HTML generation
  - Sessions
  - Interfacing to databases

Second-generation frameworks

Examples: (Ruby on Rails, Django):

- **Model-view-controller**: stylized decomposition of applications
- Object-relational mapping (**ORM**): simplify the use of databases (make database tables and rows appear as classes and objects)
  - Easier fetching of dynamic data

Third-generation frameworks

Example: AngularJS

- JavaScript frameworks running in browser - More app-like web apps
  - Interactive, quick responding applications - Don't need server round-trip
- Frameworks not dependent on particular server-side capabilities
  - Node.js - Server side JavaScript
  - No-SQL database (e.g. MongoDB)
- Many of the concepts of previous generations carry forward
  - Model-view-controller
  - Templates - HTML/CSS view description

Model-View-Controller (MVC) Pattern

- **Model**: manages the application's data
  - JavaScript objects. Photo App: User names, pictures, comments, etc.
- **View**: what the web page looks like
  - HTML/CSS. Photo App: View Users, View photo with comments
- **Controller**: fetch models and control view, handle user interactions
  - JavaScript code. Photo App: DOM event handlers, web server communication

MVC pattern been around since the late 1970's

- Originally conceived in the Smalltalk project at Xerox PARC

View Generation

- Web App: Ultimately need to generate HTML and CSS
- **Templates** are commonly used technique. Basic ideas:
  - Write HTML document containing parts of the page that are always the same.
  - Add bits of code that generate the parts that are computed for each page.
  - The template is expanded by executing code snippets, substituting the results into the document.
- Benefits of templates (Compare with direct JavaScript to DOM programming)
  - Easy to visualize HTML structure
  - Easy to see how dynamic data fits in
  - Can do either on server or browser

Controllers

- Third-generation: JavaScript running in browser

Responsibilities:

- Connect models and views
  - Server communication: Fetch models, push updates
- Control view templates
  - Manage the view templates being shown
- Handle user interactions
  - Buttons, menus, and other interactive widgets

Model Data

- All non-static information needed by the view templates or controllers
- Traditionally tied to application's database schema
  - Object Relational Mapping (ORM) - A model is a table row
- Web application's model data needs are specified by the view designers

  But need to be persisted by the database
- Conflict: Database Schemas don't like changing frequently but web application model data might (e.g. user will like this view better if we add … and lose ...)

Fourth-generation frameworks

Examples: React.js, Vue.js, Angular(v2)

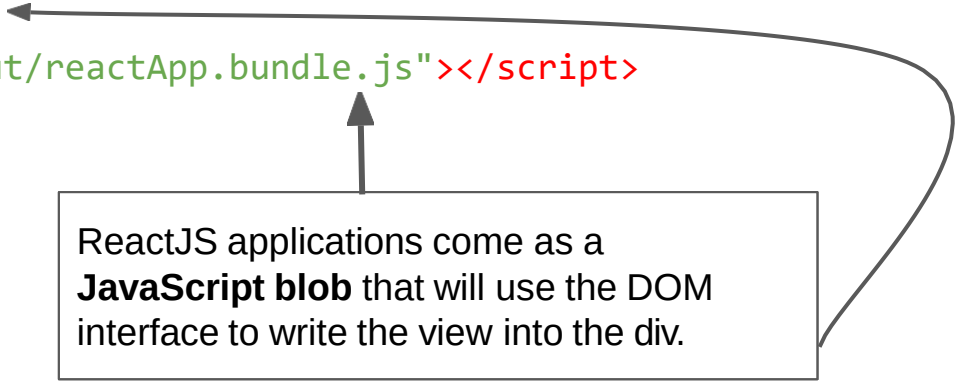
- Many of the concepts of previous generations carry forward
  - JavaScript in browser
  - Model-view-controllers
  - Templates
- Focus on JavaScript components rather than pages/HTML
  - Views apps as assembled reusable components rather than pages.
  - Software engineering focus: modular design, reusable components, testability, etc.
- Virtual DOM
  - Render view into DOM-like data structure (not real DOM)
  - Benefits: Performance, Server-side rendering, Native apps

ReactJS

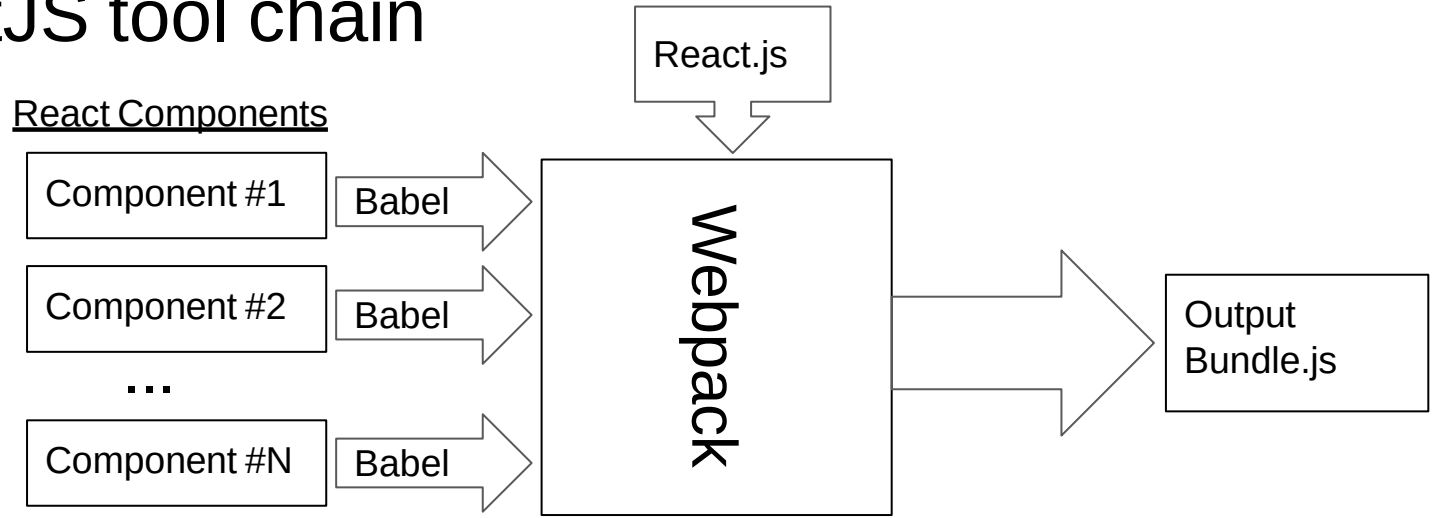
- JavaScript framework for writing the web applications
  - Like AngularJS - Snappy response from running in browser
  - Less **opinionated**: only specifies rendering view and handling user interactions
- Uses Model-View-Controller pattern
  - View constructed from Components using pattern
  - Optional, but commonly used HTML templating
- Minimal server-side support dictated
- Focus on supporting for programming in the large and single page applications
  - Modules, reusable components, testing, etc.

ReactJS Web Application Page

```
<!doctype html>
<html>
   <head>
      <title>Example</title>
   </head>
   <body>
       <div id="root"></div>
       <script src="./webpackOutput/reactApp.bundle.js"></script>
   </body>
</html>
```



ReactJS tool chain



Babel - Transpile language features (e.g. ECMAScript, JSX) to basic JavaScript

Webpack - Bundle modules and resources (CSS, images)

Output loadable with single `script` tag in any browser

index.js - Render element into browser DOM

```
import React from 'react';
import ReactDOM from 'react-dom';
import ReactAppView from './components/ReactAppView';
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <ReactAppView />
  </React.StrictMode>
);
```

ES6 Modules - Bring in React and web app React components.

Renders the tree of React elements (single component named **`ReactAppView`**) into the browser's DOM at the div with id=reactapp.

components/ReactAppView.js - ES6 class definition

```
import React from 'react';

class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    ...
  }
  render() { ...
};

export default ReactAppView;
```

Inherits from React.Component. `props` is set to the attributes passed to the component.

Require method render() - returns React element tree of the Component's view.

ReactAppView render() method

```
<div>
   <label>Name: </label>
   <input type="text" … />
   <h1>Hello!</h1>
</div>
```

```
render() {
    let label = React.createElement('label', null,'Name: ');
    let input =  React.createElement('input',{type:'text'});
   let h1 = React.createElement('h1',null ,'Hello!');

    return React.createElement('div', null, label, input, h1);
}
```

Returns element tree with `div` (`label`, `input`, and `h1`) elements

ReactAppView render() method w/o variables

```
render() {
    return React.createElement('div', null,
        React.createElement('label', null,'Name: '),
        React.createElement('input',
           { type: 'text', value: this.state.yourName,
             onChange: (event) => this.handleChange(event) }),
        React.createElement('h1',  null,
                'Hello ', this.state.yourName, '!')
      );
}
```

Use JSX to generate calls to createElement

```
render() {
   return (
      <div>
        <label>Name: </label>
        <input type="text" />
        <h1>Hello!</h1>
     </div>
    );
}
```

- JSX makes building tree look like templated HTML embedded in JavaScript.

App.js - use functions to create elements

```
function App() {
   return (
      <div>
        <label>Name: </label>
        <input type="text" />
        <h1>Hello!</h1>
     </div>
    );
}
```

- More simple and readable
- Improved performance
- Can be used with hooks and state management

JSX – JavaScript XML

Programming with JSX

- Need to remember: JSX maps to calls to `React.createElement`
  - Writing in JavaScript HTML-like syntax that is converted to JavaScript function calls
- `React.createElement(type, props, ...children);`
  - `type`: HTML tag (e.g. `h1, p`) or `React.Component`
  - `props`: attributes (e.g. `type="text"`) Uses camelCase!
  - `children`: Zero or more children which can be either:
    - String or numbers
    - A React element
    - An Array of the above

Programming with JSX

- Plain HTML

```
const myElement = React.createElement('h1', { style:{color:"green"} }, 'I do not use JSX!’);
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

- JSX

```
const myElement = <h1>I Love JSX!</h1>;
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(myElement);
```

JSX Rules

- A React component name must be capitalized (otherwise treated as built-in components)
- Return multiple HTML elements

```
const App = () => {
  return (
    <div>
      <h1>Hello World!</h1>
      <p>Tanishka here!</p>
    </div>
  );
}
```

JSX Rules

- Every tag, including self closing tags, must be closed. In case of self closing tags you have to add a slash at the end

```
const App = () => {
  return (
    <>
      <img src="./assets/react.svg" alt="" />
    </>
  );
}
```

JSX Rules

- JSX is closer to JavaScript than to HTML, the React DOM uses the camelCase naming convention for HTML attribute names.

  For example: tabIndex, onChange, and so on.

- "class" and "for" are reserved keywords in JavaScript, so use "className" and "forHTML" instead, respectively.

Styling with React/JSX - lots of different ways

```
import React from 'react';
import ‘./App.css';
function App() {
   return (
      <span className="code-name">
        ...
      </span>
    );
```

Webpack can import CSS style sheets:

```
. code-name {
  font-family: Courier New, monospace;
}
```

Must use `className=` for HTML `class=` attribute (JS keyword conflict)

Use JS in JSX

- Put plain JavaScript code in curly brackets

```
const App = () => {
  const name = "Tanishka";
  return (
    <>
      <h1>My name is {name}</h1>
    </>
  );
}
```

JSX templates must return a valid `children` param

- Templates can have JavaScript scope variables and expressions
  - `<div>{foo}</div>`
    - Valid if `foo` is in scope (i.e. if foo would have been a valid function call parameter)
  - `<div>{foo + 'S' + computeEndingString()}</div>`
    - Valid if `foo & computeEndString` in scope
- Template must evaluate to a value
  - `<div>{if (useSpanish) { … } }</div>` - Doesn't work: `if` isn't an expression
  - Same problem with "for loops" and other JavaScript statements that don't return values
- Leads to contorted looking JSX: Example: Anonymous immediate functions
  - `<div>{ (function() { if …; for ..; return val;})() }</div>`

Conditional render in JSX

- Use JavaScript Ternary operator (?:)

```
<div>{this.state.useSpanish ? <b>Hola</b> : "Hello"}</div>
```

- Use JavaScript variables

```
let greeting;
const en = "Hello";  const sp = <b>Hola</b>;
let {useSpanish} = this.prop;
if (useSpanish) {greeting = sp} else {greeting = en};

<div>{greeting}</div>
```

Iteration in JSX

- Use JavaScript array variables

```
let listItems = [];
for (let i = 0; i < data.length; i++) {
   listItems.push(<li key={data[i]}>Data Value {data[i]}</li>);
}
return <ul>{listItems}</ul>;
```

- Functional programming

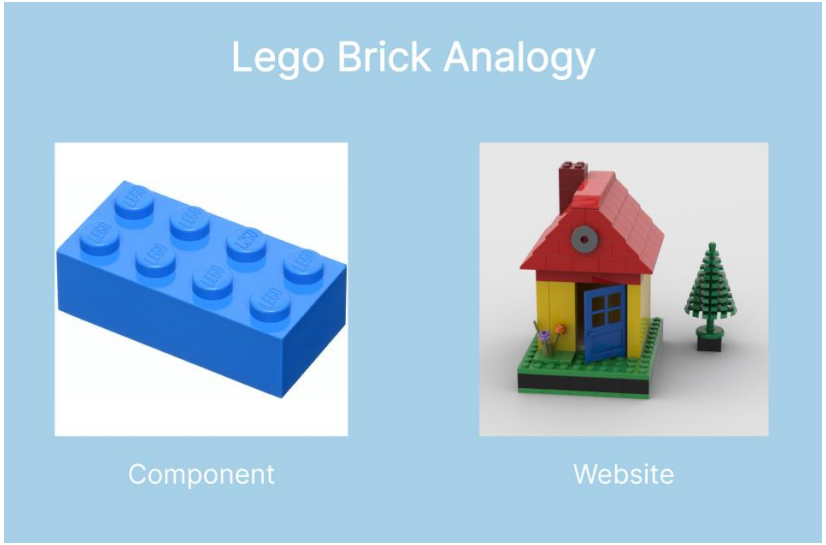
```
<ul>{data.map((d) => <li key={d}>Data Value {d}</li>)}</ul>
```

`key=` attribute improves efficiency of rendering on data change

React Component and Props

React Components

- Independent and reusable blocks of code which work in isolation
- The main advantage of components is that they help reduce redundancy.



Types of React Components

- Class components

```
class Greet extends React.Component {
        constructor (props) { … }
        render () { return <h1>Hello World!</h1>;}
}
```

- Functional components:

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

Or

```
const  Greet  =  ()  =>  <h1>Hello World!</h1>
```

Nested Components

- Creating more complex User Interfaces and getting rid of redundant code

```
const Book = () => {
      return (
            <div>
                  <h1>Book name : Cracking The Coding Interview</h1>
                  <h2>Author : Gayle Laakmann McDowell</h2>
            </div>
      );
};

const BookList = () => {
      return (
            <div>
                  <Book />
                  <Book />
            </div>
      );
};
```

Component Props

- Create components with props

```
const Book = (props) => {
   return (
      <div>
         <h1>Book name : {props.bookName}</h1>
         <h2>Author : {props.author}</h2>
      </div>
   );
};
```

```
const Book = (props) => {
   const {bookName, author} = props;
   return (
      <div>
         <h1>Book name : {bookName}</h1>
         <h2>Author : {author}</h2>
      </div>
   );
};
```

```
const Book = ({bookName, author}) => {
   return (
      <div>
         <h1>Book name : {bookName}</h1>
         <h2>Author : {author}</h2>
      </div>
   );
};
```

Pass props

- Pass props to components

```
const BookList = () => {
   return (
      <div>
         <Book bookName = "Cracking The Coding Interview"
         author = "Gayle Laakmann McDowell"/>
         <Book bookName = "The Road to Learn React"
         author = "Robert Wieruch"/>
      </div>
   );
};
```

State Management in React

Component state and input handling

```
function App() {
    let yourName = "";
    const handleChange = (event) => {
        yourName = event.target.value;
        console.log(yourName);
    }
    return (
        <div>
            <label>Name: </label>
            <input type="text" onChange = {handleChange} />
            <h1>Hello {yourName}!</h1>
        </div>
    );
}
```

Doesn’t work !

State in React

- The state is an object that holds information about a certain component.
- State allows us to manage changing data in an application.
  - It's defined as an object where we define key-value pairs specifying various data we want to track in the application.
- State change is one of the two things that make a React component re-render (the other is a change in props)
  - In this way, the state stores information about the component and also controls its behavior.

Component state and input handling

```
import React from 'react';

class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
  }
  handleChange(event) {
     this.setState({yourName: event.target.value});
  }
   ....
```

Make `<h1>Hello {this.state.yourName}!</h1>` work

- Input calls to `setState` which causes React to call render() again

One way binding: Type 'D' Character in input box

- JSX statement:

```
<input type="text" value={this.state.yourName}
onChange={(event) => this.handleChange(event)} />
<h1>Hello {this.state.yourName} !</h1>
```

  Triggers `handleChange` call with `event.target.value` == "D"

- handleChange - `this.setState({yourName: event.target.value});`

  `this.state.yourName` is changed to "D"

- React sees state change and calls render again:
- Feature of React - highly efficient re-rendering

Calling React Components from events: A problem

```
class ReactAppView extends React.Component {
    ...
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
  ...
}
```

Understand why:

```
  <input type="text" value={this.state.yourName} onChange={this.handleChange} />
```

Doesn't work!

Calling React Components from events workaround

- Create instance function bound to instance

```
class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
```

Calling React Components from events workaround

- Using public fields of classes with arrow functions

```
class ReactAppView extends React.Component {
  constructor(props) {
    super(props);
    this.state = {yourName: ""};
  }
  handleChange = (event) => {
    this.setState({ yourName: event.target.value });
  }
  ...
```

Calling React Components from events workaround

- Using arrow functions in JSX

```
class ReactAppView extends React.Component {
  ...
  handleChange(event) {
    this.setState({ yourName: event.target.value });
  }
  render() {
    return (
        <input type="text" value={this.state.yourName}
            onChange={(event) => this.handleChange(event)} />
     );
  }
```

A digression: `camelCase` vs `dash-case`

Word separator in multiword variable name

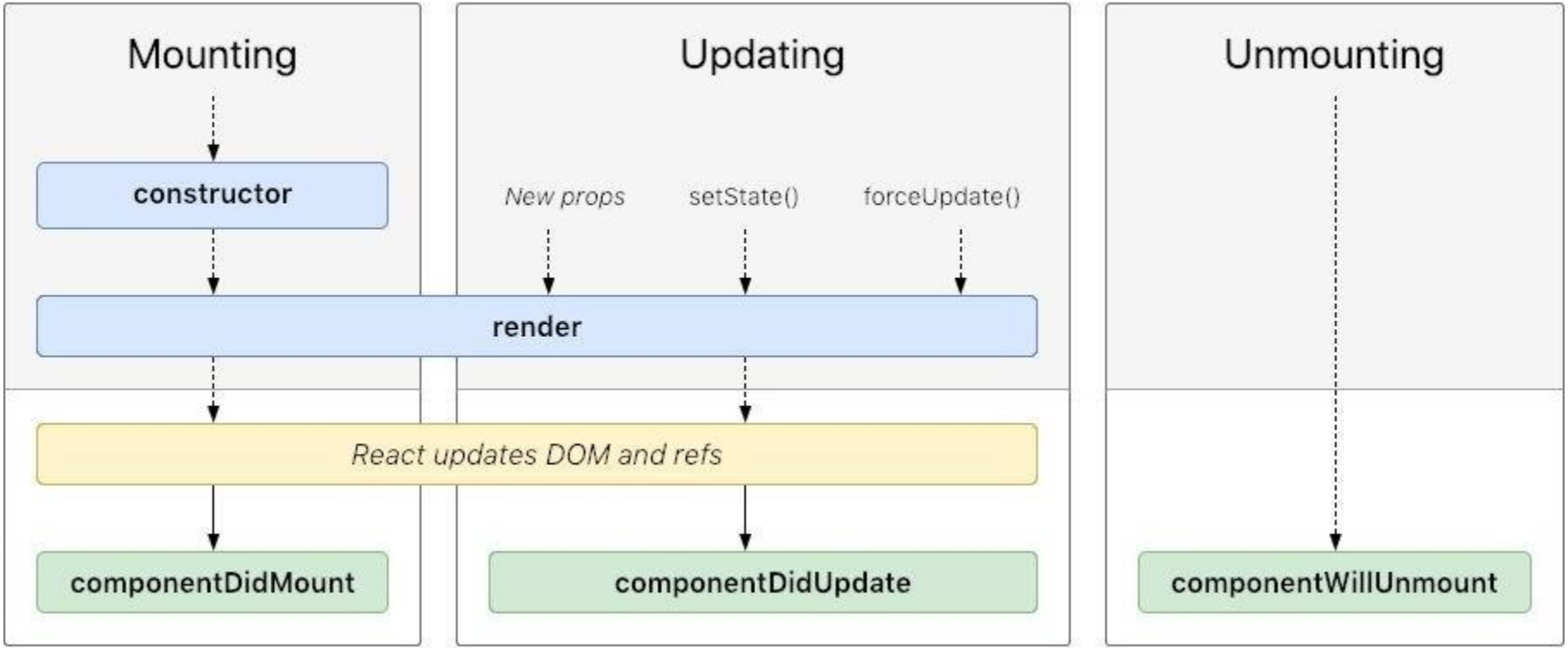
- Use dashes: `active-buffer-entry`
- Capitalize first letter of each word: `activeBufferEntry`

Issue: HTML is case-insensitive but JavaScript is not.
ReactJS's JSX has HTML-like stuff embedded in JavaScript.

ReactJS: Use camelCase for attributes

AngularJS: Used both: dashes in HTML and camelCase in JavaScript!

Component lifecycle and methods



http://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

Example of lifecycle methods - update UI every 2s

```
class Example extends React.Component {
     ...
    componentDidMount() {    // Start 2 sec counter
       const incFunc =
         () => this.setState({ counter: this.state.counter + 1 });
       this.timerID = setInterval(incFunc, 2 * 1000);
    }

    componentWillUnmount() {  // Shutdown timer
       clearInterval(this.timerID);
    }
   ...
```

Stateless Components

- React Component can be function (not a class) if it only depends on props

```
function MyComponent(props) {
  return <div>My name is {props.name}</div>;
}
```

Or using destructuring...

```
function MyComponent({name}) {
  return <div>My name is {name}</div>;
}
```

- Much more concise than a class with `render` method
  - But what if you have one bit of state…

React Hooks

Introduction to React hooks

- 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
- Hooks allow us to use stateless (functional) components together with all the more complex functionalities of class components.
- React provides a few built-in Hooks like useState, useEffect etc.
  - You can also create your own Hooks to reuse stateful behavior between different components

React Hooks - Add state to stateless components

- Inside of a "stateless" component add state: `useState(initialStateValue)`
  - `useState` parameter: `initialStateValue` - the initial value of the state
  - `useState` return value: An two element polymorphic array
    - 0th element - The current value of the state
    - 1st element - A set function to call (like `this.setState`)
- Example: a bit of state:

```
const [bit, setBit] = useState(0);
```

- How about lifecycle functions (e.g. componentDidUpdate, etc.)?
  - `useEffect(lifeCycleFunction, dependency array)`
    - `useEffect` parameter `lifeCycleFunction` - function to call when something changes

React Hooks Example - useState

```
import React, { useState} from 'react';
function App() {
  const [yourName, setName] = useState("");
  return (
      <div>
        <label>Name: </label>
        <input type="text" value={yourName}
        onChange={(event) => setName(event.target.value)} />
        <h1>Hello {yourName} !</h1>
      </div>
  );
}
```

React Hooks Example - useState

```
import React, { useState} from 'react';
function Example() {
  const [count, setCount] = useState(0);
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
           Click me
      </button>
    </div>
  );
}
```

React Hooks Example - `useEffect`

- UseEffect allows you to run a side effect on your component.
- The function passed to useEffect is a callback function. This will be called after the component renders.
- The second argument is an array, called the dependencies array. This array should include all of the values that our side effect relies upon.

```
import { useEffect } from 'react’;
function User({ name }) {
    useEffect(() => {
        document.title = name;
    }, [name]);
    return <h1>{name}</h1>;
}
```

React Hooks Example - `useEffect`

- A typical use case is to fetch data once the component has been mounted.
- Let's say we have a function called fetchData which is responsible for that – our useEffect hook might look like this:

```
useEffect(() => { fetchData() }, [])
```

- Note: if the second argument (array) is empty, the effect will run after every re-renders, otherwise it is only run when the variables in the array has changed.

Cleanup function in useEffect

- To use the cleanup function, we need to return a function from within the useEffect function.

```
function Timer() {
  const [time, setTime] = useState(0);
  useEffect(() => {
    let interval = setInterval(() => setTime(1), 1000);
    return () => {
      // setInterval cleared when component unmounts
      clearInterval(interval);
    }
  }, []);
}
```

Communicating between React components

- Passing information from parent to child: Use props (attributes)

```
<ChildComponent param={infoForChildComponent} />
```

- Passing information from child to parent: Callbacks

```
this.parentCallback = (infoFromChild) =>
        { /* processInfoFromChild */};

<ChildComponent callback={this.parentCallback}> />
```

- React Context (https://reactjs.org/docs/context.html)
  - Global variables for subtree of components

React Form Handling

Controlled component: Using useState hook

- handleChange: set input data to formData state variable.
- handleSubmit: process the formData.

```
export default function Form() {
    const [formData, setFormData] = useState({name: "",email: "",message: ""});
    const handleChange = (event) => {
        const { name, value } = event.target;
        setFormData((prevFormData) => ({ ...prevFormData, [name]: value }));
    };
    const handleSubmit = (event) => {
        event.preventDefault();
        alert(`Name: ${formData.name}, Email: ${formData.email}, Message: ${formData.message}`
        );
    };
//...
```

Using useState hook

```
    return (
        <form onSubmit={handleSubmit}>
            <label htmlFor="name">Name:</label>
            <input type="text" id="name" name="name" value={formData.name} onChange={handleChange}/>

            <label htmlFor="email">Email:</label>
            <input type="email" id="email" name="email" value={formData.email} onChange={handleChange}/>

            <label htmlFor="message">Message:</label>
            <textarea id="message" name="message" value={formData.message} onChange={handleChange}/>

            <button type="submit">Submit</button>
        </form>
    );
}
```

Using useState hook: input validation

```
function MyForm() {
    const [inputValue, setInputValue] = useState('');
    const [inputError, setInputError] = useState(null);

    function handleInputChange(event) {
        const value = event.target.value;
        setInputValue(value);

        if (value.length < 5) {
            setInputError('Input must be at least 5 characters');
        } else {
            setInputError(null);
        }
    }
```

Using useState hook: input validation

```
function handleSubmit(event) {
        event.preventDefault();
        if (inputValue.length >= 5) {
            // submit form
        } else {
            setInputError('Input must be at least 5 characters');
        }
    }

    return (
        <form onSubmit={handleSubmit}>
            <label>
                Fruit:
                <input type="text" value={inputValue} onChange={handleInputChange} />
            </label>
            {inputError && <div style={{ color: 'red' }}>{inputError}</div>}
            <button type="submit">Submit</button>
        </form>
    );
}
```

Uncontrolled component: Using useRef hook

- Use ref to get the current value of the input
- handleSubmit: process the input value

```
import { useRef } from "react";

export default function Uncontrolled() {
    const selectRef = useRef(null);
    const checkboxRef = useRef(null);
    const inputRef = useRef(null);

    function handleSubmit(event) {
        event.preventDefault();
        console.log("Input value:", inputRef.current.value);
        console.log("Select value:", selectRef.current.value);
        console.log("Checkbox value:", checkboxRef.current.checked);
    }
```

Using useRef hook

```
return (
        <form onSubmit={handleSubmit}>
            <label>
                <p>Name:</p>
                <input ref={inputRef} type="text" />
            </label>
            <label>
                <p>Favorite color:</p>
                <select ref={selectRef}>
                    <option value="red">Red</option>
                    <option value="green">Green</option>
                    <option value="blue">Blue</option>
                </select>
            </label>
            <label>
                Do you like React?
                <input type="checkbox" ref={checkboxRef} />
            </label>
            <button type="submit">Submit</button>
        </form>
    );
}
```

Use component library: React hook form

- Need to install the library: `npm install react-hook-form`
- The `useForm` hook provides several functions and properties that you can use to manage your form:
  - `register`: This function is used to register form fields with React Hook Form.
  - `handleSubmit:` This is used to handle form submissions. It takes a callback function that is called when the form is submitted.
  - `errors:` This represents an object containing any validation errors that occur when a form is submitted.
  - `watch:` This function is used to watch for changes to specific form fields. It takes an array of form field names and returns the current value of those fields.

Use component library: React hook form

```
import { useForm } from 'react-hook-form';
function LoginForm() {
    const { register, handleSubmit, formState: { errors } } = useForm();
    const onSubmit = (data) => {
        console.log(data);
    };

    return (
        <form onSubmit={handleSubmit(onSubmit)}>
            <label>Email</label>
            <input type="email" {...register("email", { required: true, pattern: /^\S+@\S+$/i })} />
            {errors.email && <p>Email is required and must be valid</p>}

            <label>Password</label>
            <input type="password" {...register("password", { required: true })} />
            {errors.password && <p>Password is required</p>}

            <button type="submit">Submit</button>
        </form>
    );
}
```

Data Handling

What is a API?

- API stands for Application Programming Interface. It enables the exchange of information and functionality between different systems
  - E.g between a website and a server or between different software applications
- An API functions as a waiter for software applications. It is a set of rules that lets one program ask another for something it needs.
- Why are APIs important in web development?:
  - Web applications need APIs to get data from various sources, like databases or websites.
  - APIs are a scalable option for managing high data or request volumes.
  - Developers use APIs to leverage existing features and services. This saves them from reinventing the wheel.
  - They keep things safe by ensuring that only authorized individuals or programs can use them.
  - An API makes a website or mobile app more enjoyable to use by integrating data.

Restful API

- REST (Representational State Transfer) is an architectural style that defines a set of constraints to be used when creating web services
  - RESTful APIs follow several key principles, including statelessness, uniform interface, and resource-based interactions.
- Anatomy of a RESTful API:
  - A RESTful API consists of resources, each identified by a unique URI (Uniform Resource Identifier).
  - These resources can be manipulated using standard HTTP methods such as GET, POST, PUT, PATCH, and DELETE.
  - The API responses typically include data in a format like JSON (JavaScript Object Notation) or XML (eXtensible Markup Language).
- RESTful API Endpoints: Endpoints are specific URLs that represent the resources exposed by a RESTful API.
  - For example, a simple blog API might have endpoints like /posts to retrieve all blog posts and /posts/{id} to retrieve a specific post by its unique identifier.

How to Make API Requests in React

- Axios is a popular JavaScript library for making HTTP requests.
  - Need to install axios library: `npm install axios`
- The Fetch API: A modern interface for making HTTP requests in the browser
  - The `useEffect` hook is used to fetch data when the component mounts
  - The `fetch` function is used to make a GET request to the specified API endpoint (`'https://api.example.com/posts'`)
  - the response is converted to JSON using `response.json()`

How to Make API Requests in React

```
import React, { useState, useEffect } from 'react’;
const ApiExample = () => {
    const [data, setData] = useState([]);
    useEffect(() => {
        const fetchData = async () => {
            try {
                const response = await fetch('https://api.example.com/posts');
                const result = await response.json();
                setData(result);
            } catch (error) {
                console.error('Error fetching data:', error);
            }
        };
        fetchData();
    }, []);

    return (
        <div>
            <h1>API Data</h1>
            <ul>
                {data.map((item) => (
                    <li key={item.id}>{item.title}</li>
                ))}
            </ul>
        </div>
    );
};
```

Making GET Requests: Enhance previous example

```
const ApiExample = () => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(true);

    useEffect(() => {
        const fetchData = async () => {
            try {
                // Simulating a delay to show loading state
                setTimeout(async () => {
                    const response = await fetch('https://api.example.com/posts?userId=1');
                    const result = await response.json();
                    setData(result);
                    setLoading(false);
                }, 1000);
            } catch (error) {
                console.error('Error fetching data:', error);
                setLoading(false);
            }
        };

        fetchData();
    }, []);
```

Making GET Requests: Enhance previous example

- A loading state is introduced to provide feedback to users while the data is being fetched

```
return (
    <div>
        <h1>API Data</h1>
        {loading ? (
            <p>Loading...</p>
        ) : (
            <ul>
                {data.map((item) => (
                    <li key={item.id}>{item.title}</li>
                ))}
            </ul>
        )}
    </div>
);
};
```

Handling Asynchronous Operations with `async/await`

- The use of `async/await` syntax makes asynchronous code more readable and easier to work with
- The `fetchData` function is declared as an asynchronous function using the `async` keyword. This allows the use of `await` inside the function

```
useEffect(() => {
    const fetchData = async () => {
        try {
            const response = await fetch('https://api.example.com/posts?userId=1');
            const result = await response.json();
            setData(result);
            setLoading(false);
        } catch (error) {
            console.error('Error fetching data:', error);
            setLoading(false);
        }
    };

    fetchData();
}, []);
```

Error Handling When Fetching Data

```
const ApiExample = () => {
    const [data, setData] = useState([]);
    const [loading, setLoading] = useState(true);
    const [error, setError] = useState(null);
    useEffect(() => {
        const fetchData = async () => {
            try {
                const response = await fetch('https://api.example.com/posts?userId=1');
                if (!response.ok) {
                    throw new Error(`HTTP error! Status: ${response.status}`);
                }
                const result = await response.json();
                setData(result);
                setLoading(false);
            } catch (error) {
                console.error('Error fetching data:', error);
                setError('An error occurred while fetching the data. Please try again later.');
                setLoading(false);
            }
        };
        fetchData();
    }, []);
```

Error Handling When Fetching Data

- The `response.ok` property is checked to determine if the HTTP request was successful. If not, an error is thrown with information about the HTTP status

```
return (
    <div>
        <h1>API Data</h1>
        {loading ? (
            <p>Loading...</p>
        ) : error ? (
            <p>{error}</p>
        ) : (
            <ul>
                {data.map((item) => (
                    <li key={item.id}>{item.title}</li>
                ))}
            </ul>
        )}
    </div>
 );
};
```

Display API Data in React Components

```
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const DisplayData = () => {
    const [apiData, setApiData] = useState(null);

    useEffect(() => {
        const fetchData = async () => {
            try {
                const response = await axios.get('https://api.example.com/data');
                setApiData(response.data);
            } catch (error) {
                console.error('Error fetching data:', error);
            }
        };

        fetchData();
    }, []);
```

Display API Data in React Components

```
return (
        <div>
            <h2>API Data Display</h2>
            {apiData ? (
                // Render your component using the fetched data
                <MyComponent data={apiData} />
            ) : (
                // Render a loading state or placeholder
                <p>Loading...</p>
            )}
        </div>
    );
};

const MyComponent = ({ data }) => {
    return (
        <div>
            <p>{data.message}</p>
            {/* Render other components based on data */}
        </div>
    );
};
```

CRUD Operations with RESTful APIs: Creating Data

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

const CreateData = () => {
    const [newData, setNewData] = useState('');
    const handleCreate = async () => {
        try {
            await axios.post('https://api.example.com/data', { newData });
            alert('Data created successfully!');
            // Optionally, fetch and update the displayed data
        } catch (error) {
            console.error('Error creating data:', error);
        }
    };
    return (
        <div>
            <h2>Create New Data</h2>
            <input
                type="text"
                value={newData}
                onChange={(e) => setNewData(e.target.value)}
            />
            <button onClick={handleCreate}>Create</button>
        </div>
    );
};
export default CreateData;
```

CRUD Operations with RESTful APIs: Updating Data

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

const UpdateData = () => {
    const [updatedData, setUpdatedData] = useState('');
    const handleUpdate = async () => {
        try {
            await axios.put('https://api.example.com/data/1', { updatedData });
            alert('Data updated successfully!');
            // Optionally, fetch and update the displayed data
        } catch (error) {
            console.error('Error updating data:', error);
        }
    };
    return (
        <div>
            <h2>Update Data</h2>
            <input
                type="text"
                value={updatedData}
                onChange={(e) => setUpdatedData(e.target.value)}
            />
            <button onClick={handleUpdate}>Update</button>
        </div>
    );
};
export default UpdateData;
```

CRUD Operations with RESTful APIs: Deleting Data

```jsx
import React from 'react';
import axios from 'axios';

const DeleteData = () => {
    const handleDelete = async () => {
        try {
            await axios.delete('https://api.example.com/data/1');
            alert('Data deleted successfully!');
            // Optionally, fetch and update the displayed data
        } catch (error) {
            console.error('Error deleting data:', error);
        }
    };

    return (
        <div>
            <h2>Delete Data</h2>
            <button onClick={handleDelete}>Delete</button>
        </div>
    );
};

export default DeleteData;
```