Module 15

Day 1: Intro to React

Lesson 1.1

1. Introduction to React

- What is React?
  A powerful JavaScript library for building user interfaces, particularly single-page applications where you need a fast, interactive user experience.
- Single Page Applications (SPAs):
  Web applications that load a single HTML page and dynamically update content as the user interacts with the app.
- Key React Concepts:
  - Components
  - Virtual DOM
  - One-way data binding

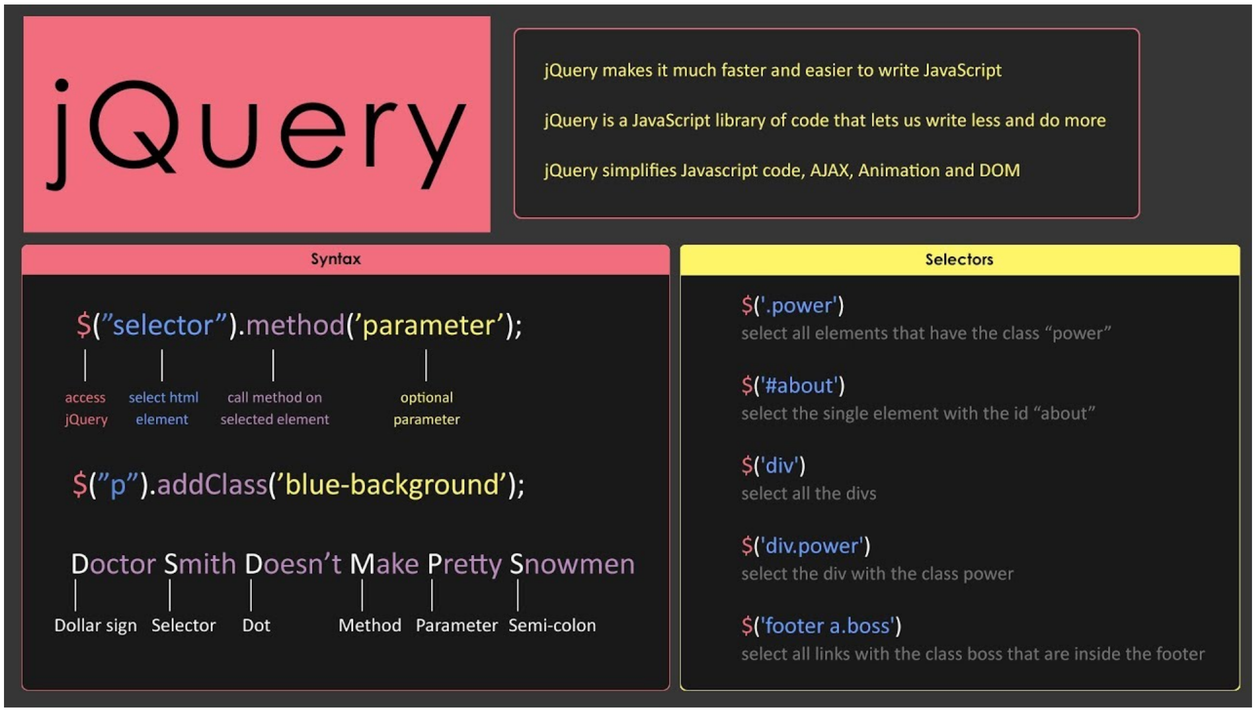
2. A Brief History of Frontend Development

- Evolution from Vanilla JS to Libraries/Frameworks:
  Development has evolved from writing plain JavaScript to using libraries and frameworks that streamline and enhance the development process.

Vanilla JS

Source: https://unclebigbay.com/difference-between-vanilla-javascript-and-javascript-explained

jQuery



jQuery

jQuery makes it much faster and easier to write JavaScript

jQuery is a JavaScript library of code that lets us write less and do more

jQuery simplifies Javascript code, AJAX, Animation and DOM

Syntax

```
$("selector").method('parameter');
|      |        |          |
access select html  call method on   optional
jQuery element  selected element     parameter

$("p").addClass('blue-background');

Doctor Smith Doesn't Make Pretty Snowmen
|      |     |       |    |      |
Dollar sign Selector Dot Method Parameter Semi-colon
```

Selectors

`$('.power')`
select all elements that have the class "power"

`$('#about')`
select the single element with the id "about"

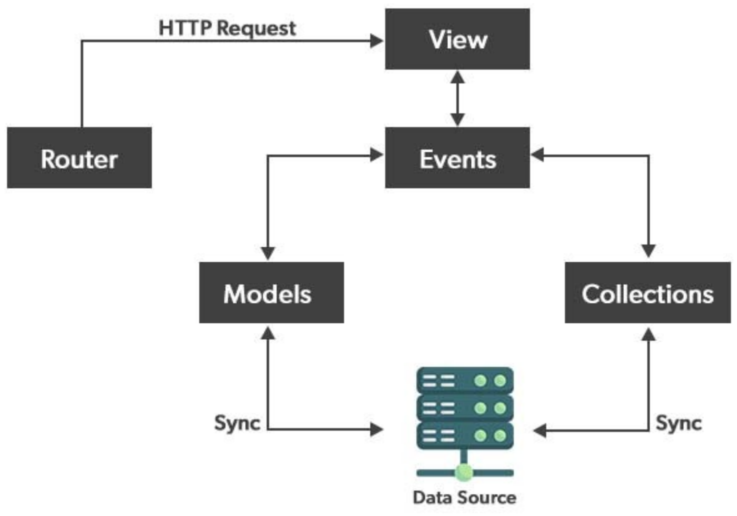
`$('div')`
select all the divs

`$('div.power')`
select the div with the class power

`$('footer a.boss')`
select all links with the class boss that are inside the footer

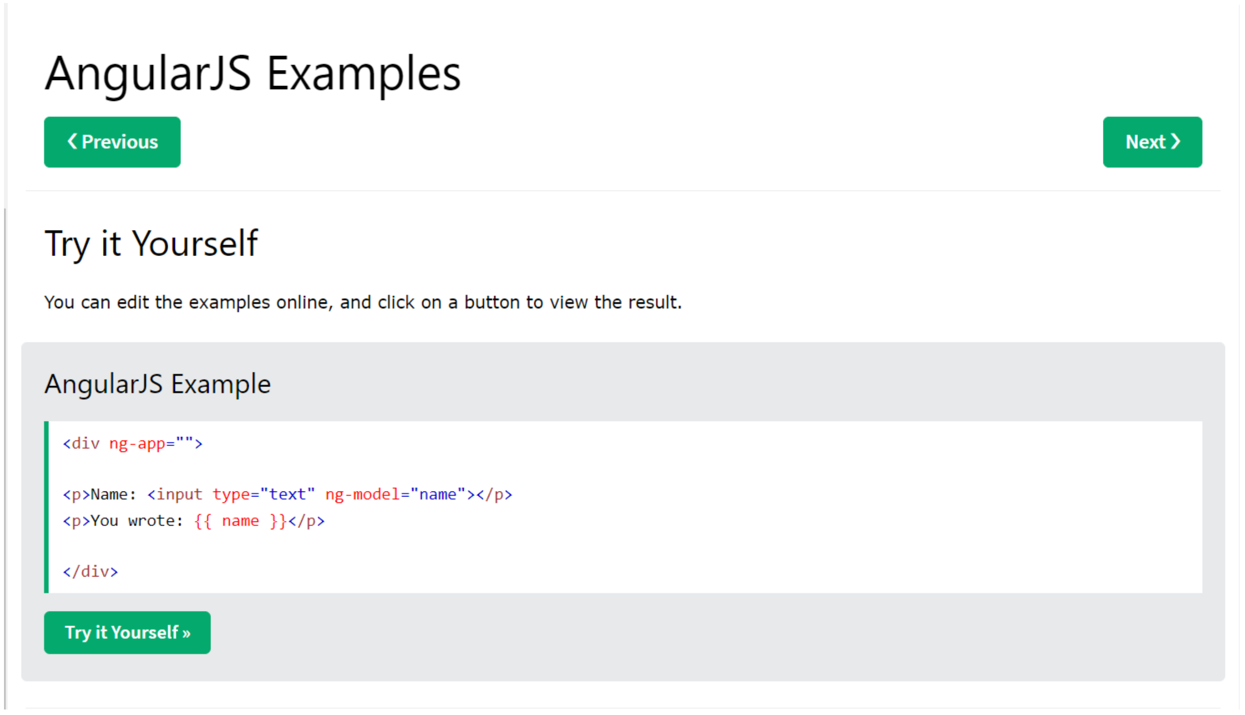
Source: https://www.youtube.com/watch?app=desktop&v=DZFugzFg4Ak

Backbone.js



Source: https://www.geeksforgeeks.org/explain-the-architecture-of-backbone-js/

Angular.js



AngularJS Examples

Try it Yourself

You can edit the examples online, and click on a button to view the result.

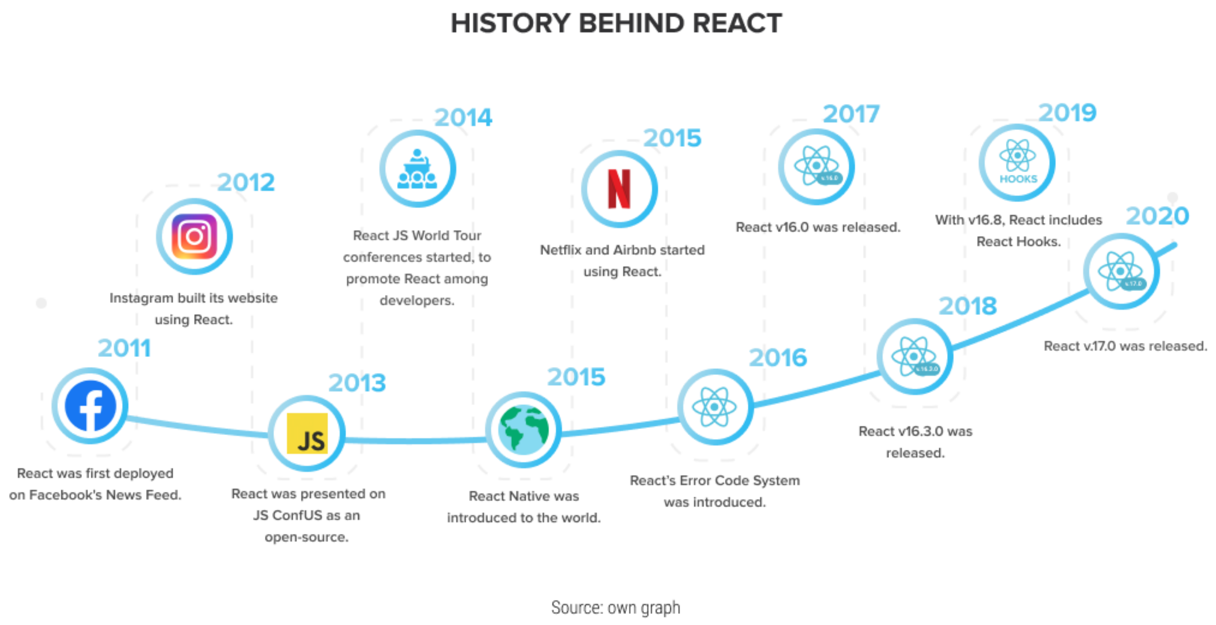
AngularJS Example

```
<div ng-app="">

<p>Name: <input type="text" ng-model="name"></p>
<p>You wrote: {{ name }}</p>

</div>
```

Source: https://www.w3schools.com/angular/default.asp

3. The Birth of React

- Origin at Facebook:
  Created by Jordan Walke and the engineering team at Facebook to solve the challenges of maintaining large-scale applications.
- Open-Sourced in 2013:
  Made available to the wider development community, leading to widespread adoption and contribution.
- Strengths of React:
  Efficiently handles rapidly changing data and updates the UI seamlessly, crucial for real-time applications and enhancing user interactions.

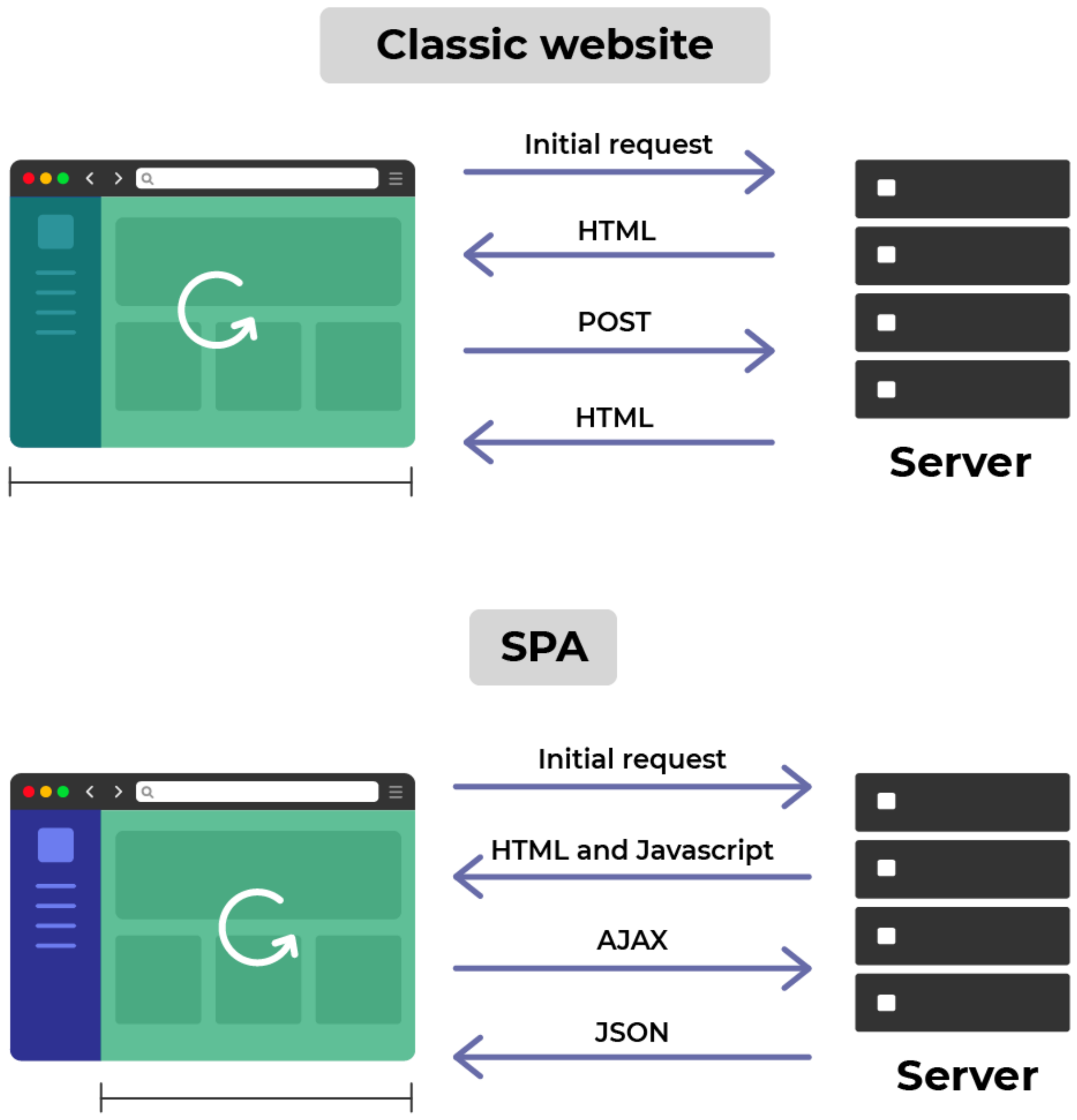

HISTORY BEHIND REACT
2011
React was first deployed on Facebook's News Feed.
2012
Instagram built its website using React.
2013
React was presented on JS ConfUS as an open-source.
2014
React JS World Tour conferences started, to promote React among developers.
2015
React Native was introduced to the world.
2015
Netflix and Airbnb started using React.
2016
React's Error Code System was introduced.
2017
React v16.0 was released.
2018
React v16.3.0 was released.
2019
With v16.8, React includes React Hooks.
HOOKS
2020
React v.17.0 was released.
Source: own graph

Source: https://2muchcoffee.com/blog/best-examples-of-websites-built-with-react/

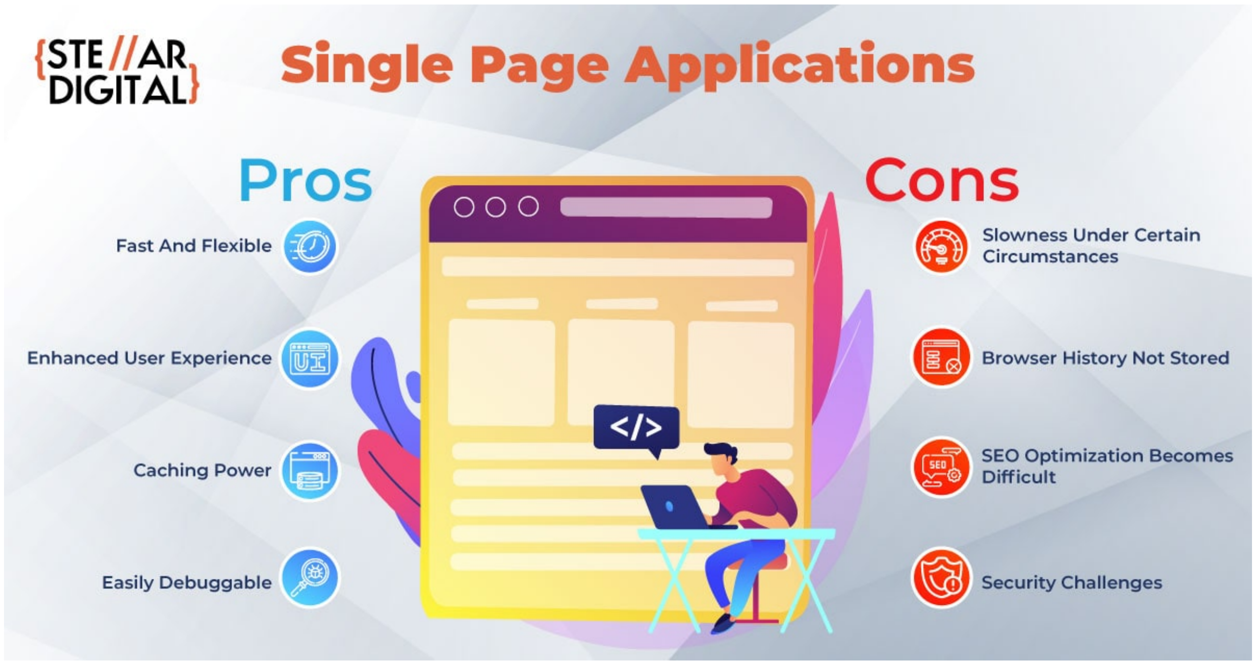
4. Single Page Applications (SPAs)

- Traditional vs. SPA Model:
  - Traditional: Full-page refreshes for each user interaction or navigation event.

    - SPA: A single HTML page, with dynamic updates for smoother user experiences.
- Advantages of SPAs:
    - Improved performance.
    - Simplified deployment and maintenance.
    - Enhanced user experience.
- Challenges and Solutions:
    - SEO: Requires additional considerations.
    - Initial Load Time: Potentially longer but leads to faster subsequent interactions.
    - Browser History Management: Managed via client-side routing libraries.



Source:
https://openclassrooms.com/en/courses/7315991-intermediate-react/7572037-turn-your-application-into-a-single-page-app-with-react-router



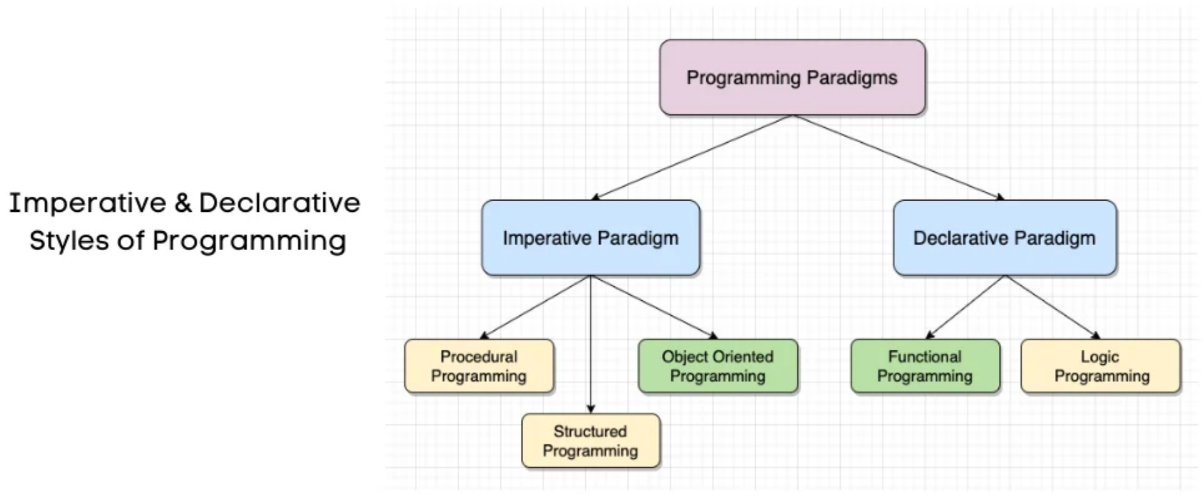
Source:
https://www.stellardigital.in/blog/what-are-the-pros-and-cons-of-single-page-applications/

5. Core Concepts in React

- Declarative vs. Imperative Programming:
  - Declarative (React's approach): Focus on what the application should do.
  - Imperative: Detailed step-by-step instructions.
- Virtual DOM:
  - An in-memory representation of the actual DOM for improved performance.
  - Minimizes actual DOM manipulations.
- Reactive Updates:
  - Automatic UI updates in response to state changes.
- Component-based Architecture:
  - Building UIs with reusable, isolated components.

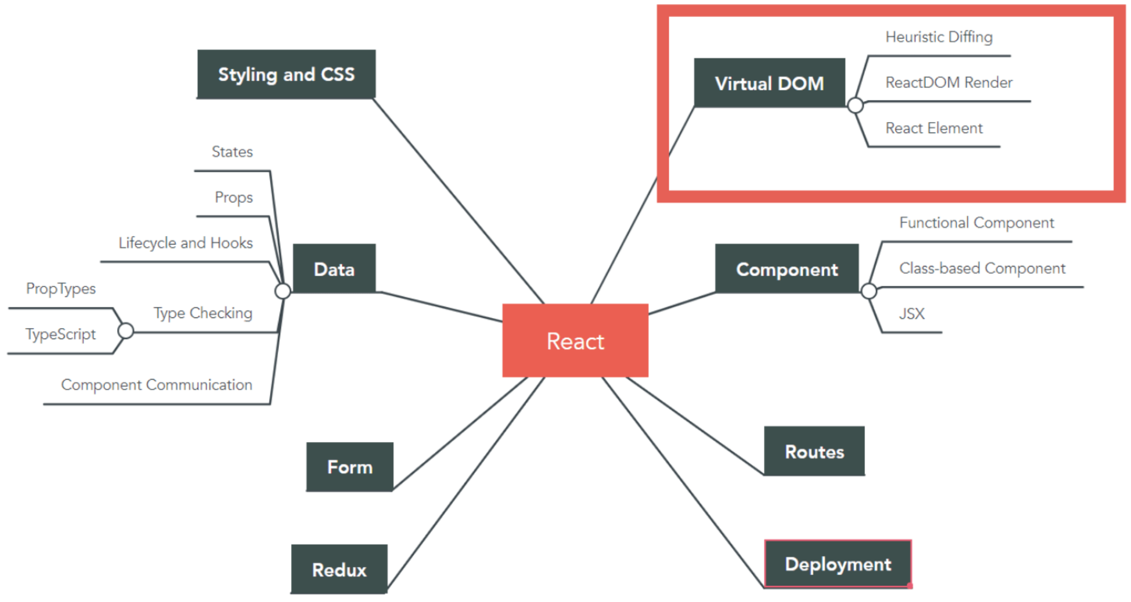
- One-way Data Binding:
    - Data flows from parent components to child components, simplifying data tracking and debugging.

Declarative vs Imperative Programming



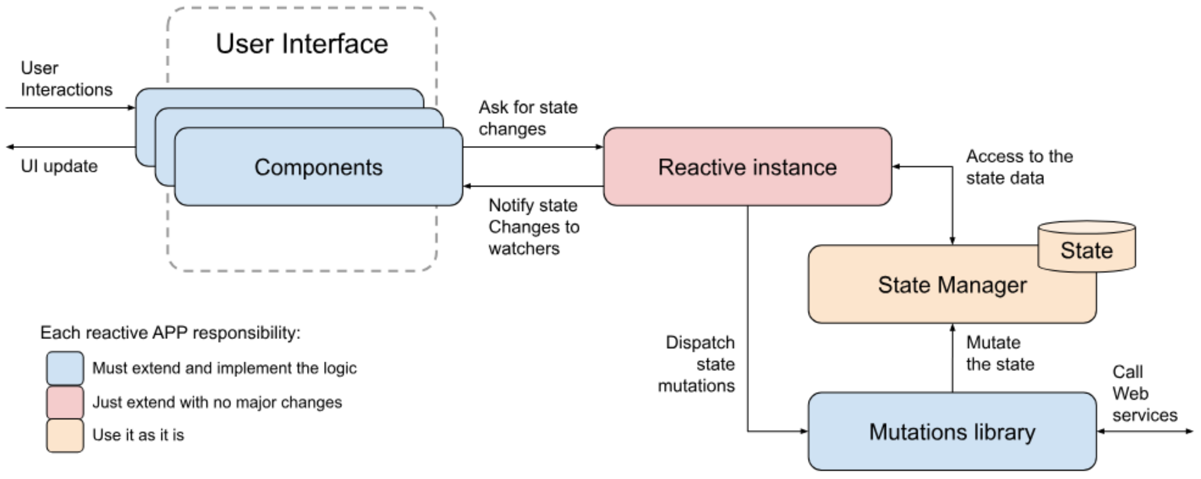
Source: https://dev.to/ggorantala/imperative-and-declarative-styles-of-programming-186d

Virtual DOM



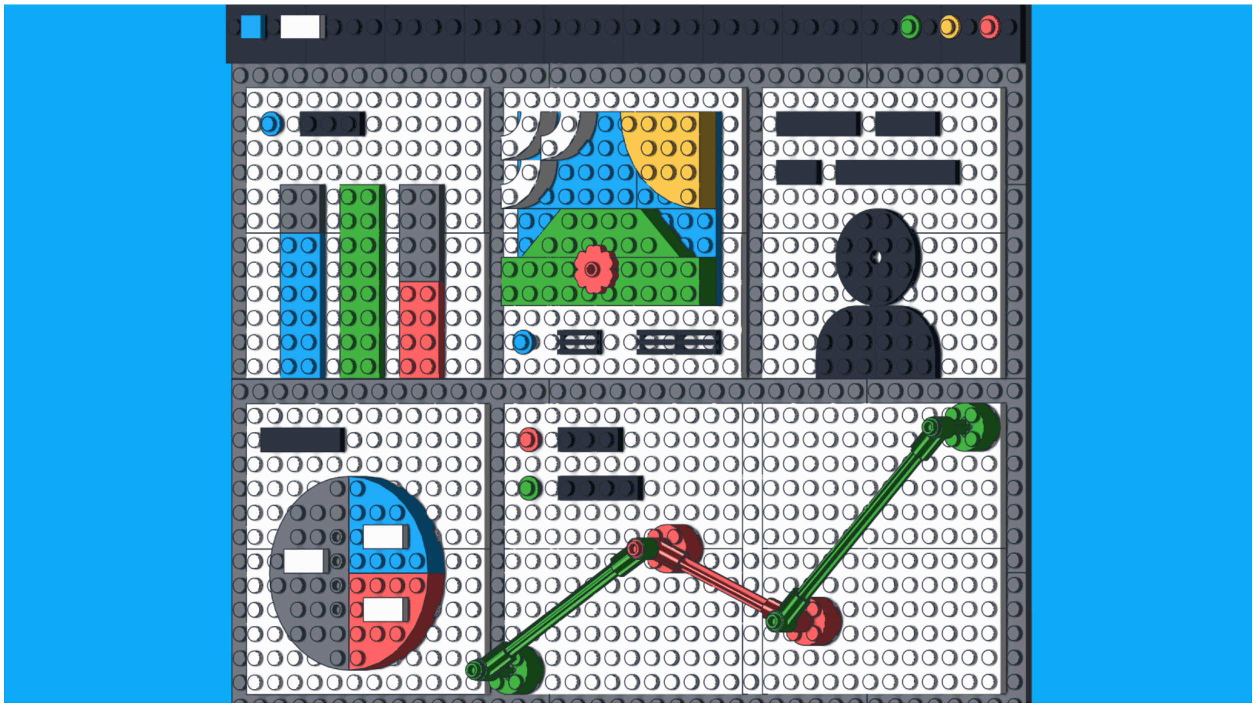
Source: https://www.underthehoodlearning.com/react-meet-virtual-dom/

Reactive Updates



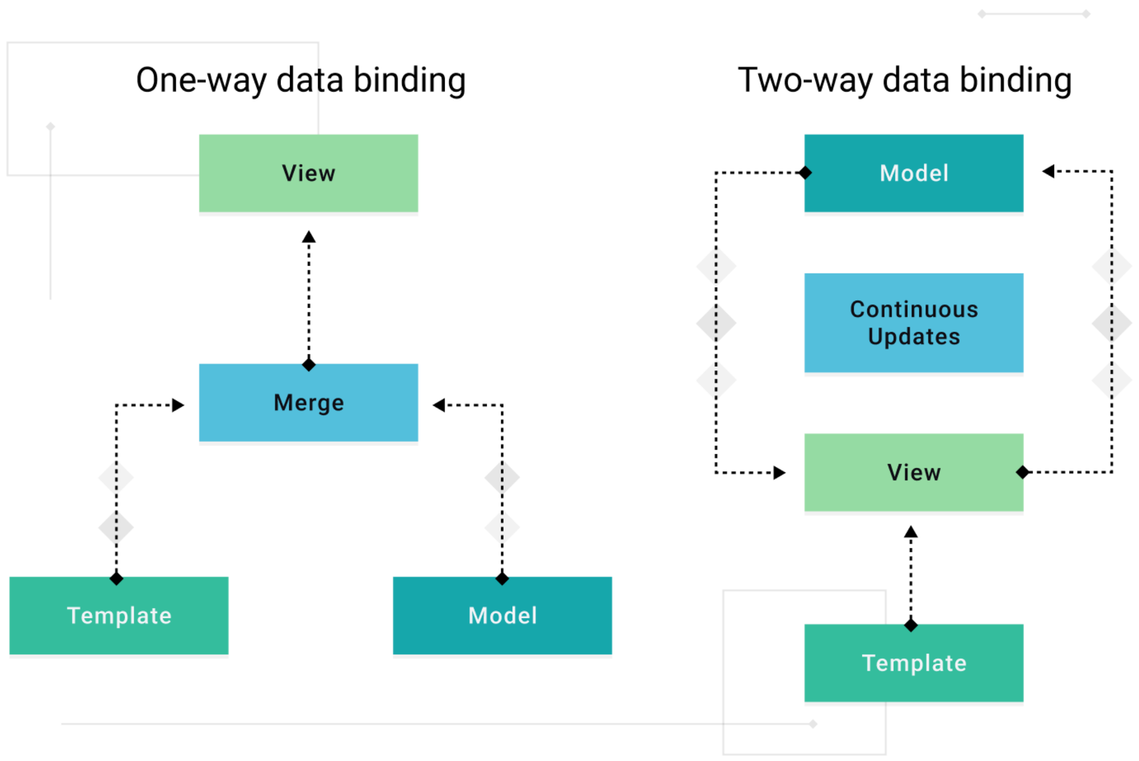
Source: https://moodledev.io/docs/guides/javascript/reactive

Component Based Architecture



Source: https://www.mendix.com/blog/what-is-component-based-architecture/

One-way Data Binding



Source: https://handsontable.com/blog/understanding-data-binding-in-react

Lesson 1.2

Lesson 2: React Basics Part 1

1. Setting Up a React Project with CRA (Create React App)

- CRA: A command-line tool that sets up a new React project with a good default configuration.
- Installation: Run `npx create-react-app my-app`.
- Running the App: Navigate to the project folder and run `npm start`.

2. CRA Boilerplate Files and Folders

- src: Contains all the JavaScript and CSS.
- public: Houses static files like images and HTML.
- node_modules: Stores all the project dependencies.
- package.json: Keeps track of project settings and dependencies.
- Others: Various configuration files.

3. Setting Up a React Project with Vite

- Vite: A newer build tool that offers faster development and build times.
- Installation: Run `npm create vite`.
- Running the App: Run `npm install` followed by `npm run dev`.

4. Vite Boilerplate Files and Folders

- src: Main directory for JavaScript and CSS.
- public: Contains static assets.
- Others: Similar to CRA but with some differences in configuration and script names.

5. CRA vs Vite: Pros and Cons

- CRA: Mature, stable, and well-documented. A bit slower in development.
- Vite: Faster with Hot Module Replacement. Newer and less mature.

6. A Primer on JSX

- JSX: A syntax extension for JavaScript to write HTML-like code.
- Transformation: JSX gets compiled into JavaScript function calls.
- Dynamic Content: Embed JavaScript expressions in JSX using curly braces.

7. Syntax of Components: Classes vs Functions

- Class Components:
  - ES6 classes extending `React.Component`.
  - Include a `render` method.
  - Offer lifecycle methods like `componentDidMount`.
- Function Components:
  - Simpler, just JavaScript functions returning JSX.
  - Can use hooks for state and lifecycle features.
  - Recommended for new code.

8. Function Components

- Simpler Syntax: Just a function returning JSX.
- Hooks: Enable state and lifecycle features.
- Conciseness: Less boilerplate than class components.
- Usage: Now the standard and recommended way to create components.

Lesson 2: Resources

1. Setting Up a React Project with CRA (Create React App)

- Create React App: Getting Started
- Introduction to Create React App (MDN)

2. CRA Boilerplate Files and Folders

- Folder Structure in Create React App

3. Setting Up a React Project with Vite

- Getting Started with Vite

4. Vite Boilerplate Files and Folders

- Vite Documentation

6. A Primer on JSX

- Writing Markup with JSX – React

7. Syntax of Components: Classes vs Functions

- Built-in React Components

8. Function Components

- A Complete Guide to useEffect (Overreacted by Dan Abramov)

Lesson 1.3

Lesson 3: React Basics Part 2

3.1 The Concept of Props

- Props (Properties): A way to pass data from a parent component to a child component.
- Usage: Make components reusable and dynamic.
- Read-Only: Props should not be modified inside the component.

3.2 Overview of Hooks

- Hooks: Functions that let you use state and lifecycle features in functional components.
- Introduced in React 16.8: Prior to this, state and lifecycle features were only available in class components.

3.3 useState Hook

- Purpose: To add state to functional components.
- Syntax: `const [state, setState] = useState(initialState);`
- Usage: Handling user input, toggling UI elements, etc.

3.4 Passing State as Props

- Data Flow: Parent to Child.
- Usage: Pass state from a parent component to a child component as props.

3.5 Which Components Should Have State

- Stateful Components: Components that manage state.
- Stateless Components: Components that receive data through props and render UI.
- Best Practice: Minimize the number of stateful components.

3.6 useEffect Hook

- Purpose: To perform side effects in functional components.
- Syntax: `useEffect(() => { /* side effects */ }, [dependencies]);`
- Usage: Fetching data, subscriptions, manual DOM manipulations.

3.7 useRef Hook

- Purpose: To access and interact with a DOM element.
- Syntax: `const refContainer = useRef(initialValue);`
- Usage: Managing focus, text selection, triggering imperative animations.

3.8 Styles and SCSS Modules

- Styles: Enhancing the visual appeal and user experience of the application.
- SCSS Modules: Leveraging SCSS syntax in CSS modules for styling React components.

3.9 Resources

- React Reference Overview
- SCSS Official Documentation

React Fundamentals: Key Concepts

Functional Components:

Functional components are the building blocks of React applications. They are JavaScript functions that return React elements, which describe what should be rendered on the screen. Functional components are the preferred way to create components in modern React. Here's an example of a functional component:

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

The Concept of State:

State is a critical concept in React. It represents the data that a component maintains and can change over time. State allows components to be dynamic and interactive. To use state, we typically use the `useState` hook (introduced in React 16.8). Here's how you declare and update state:

```
import { useState } from 'react';

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

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

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

Hooks:

Hooks are functions that allow you to “hook into” React state and lifecycle features from functional components. They were introduced to manage stateful logic in functional components. The most common hooks are `useState` (for state management) and `useEffect` (for side effects). Here’s an example of how `useEffect` can be used:

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

function ExampleComponent() {
  const [data, setData] = useState([]);

  useEffect(() => {
    // This code will run after the component renders.
    fetchData().then((result) => {
      setData(result);
    });
  }, []); // The empty dependency array means it runs once on component mount.

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

useState Hook

`useState` hook involves declaring a state variable and a corresponding function for updating that variable. Here’s the syntax:

```
const [stateVariable, setStateFunction] = useState(initialValue);
```

- `stateVariable`: This is the variable that holds the current state value. It represents the piece of data you want to manage in your component’s state.

- `setStateFunction`: This is a function that allows you to update the `stateVariable`. When you call `setStateFunction(newValue)`, it updates the `stateVariable` to `newValue`, and React re-renders the component with the updated state.

- `initialValue`: This is the initial value you want to assign to your state variable. It can be of any data type, such as a number, string, boolean, object, or even an array.

Here's an example using this syntax:

```
import { useState } from 'react';

function Counter() {
  // Declare a state variable 'count' and a corresponding setter function
'setCount'
  const [count, setCount] = useState(0);

  // This button click handler increments the 'count' state
  function increment() {
    setCount(count + 1);
  }

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
    </div>
  );
}
```

In this example, we're managing a single state variable `count` initialized to `0`. When the "Increment" button is clicked, the `increment` function is called, which in turn calls `setCount` to update the `count` state.

This standard syntax is commonly used for managing state in functional components in React and provides a clean and straightforward way to work with component state.

Props:

Props (short for properties) are a way to pass data from parent to child components in React. They make it easy to share information between components. Here's how you use props in a functional component:

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

// Usage
<Greeting name="Alice" />
```

Component Lifecycle:

React components have a lifecycle, which describes the sequence of events a component goes through. Understanding lifecycle methods (e.g., `componentDidMount`, `componentWillUnmount`) can help manage side effects and component updates.

React Router:

React Router is a popular library for handling routing in React applications. It allows you to navigate between different parts of your app while maintaining a single-page application feel. Here's how you can use React Router:

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

function App() {
  return (
    <Router>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
        </ul>
      </nav>
      <Route path="/" exact component={Home} />
      <Route path="/about" component={About} />
    </Router>
  );

}
```

Day 2: React Deep Dive

Lesson 2.1

Lesson Overview:

In this lesson, we covered essential concepts and practical examples related to React, a popular JavaScript library for building user interfaces. We explored the following topics:

Conditional Rendering:

In React, you can render content conditionally based on certain conditions. Here's an example using the ternary operator:

```
{ condition ? <ComponentIfTrue /> : <ComponentIfFalse /> }
```

Lists and Keys:

To render lists efficiently in React, use the `map()` function and provide unique keys to list items. For example:

```
{items.map((item) => (
  <li key={item.id}>{item.name}</li>
))}
```

Forms and Controlled Components:

To create interactive forms in React, use controlled components. Here's an example of a controlled input field:

```
<input
  type="text"
  value={name}
  onChange={(e) => setName(e.target.value)}
/>
```

Lesson Resources:

- React Tutorial: [React Tutorial]
- React Patterns: [React Patterns]
- React for Beginners: [React for Beginners]
- React Router Documentation: [React Router]

Lesson 2.2

Lesson 2: Routing, Route Params, Using APIs, Environment Variables, Deploying a React App

Routing:

- React Router 6: A popular library for handling routing and navigation in React applications.
- Installation: Install React Router with `npm install react-router-dom@6`.
- Components: Create different components for each route, and use the `Link` component to navigate between them.
- BrowserRouter: Wrap your routes with a `BrowserRouter` component to enable routing.

Resources:

- [React Router on GitHub]()

Route Params:

- Route Params: Capture values from the URL using route parameters.
- :id Syntax: Define route parameters with `:id` syntax.
- useParams Hook: Access route parameters in your component using the `useParams` hook.

Nested Routes:

- Hierarchical Routing: Create nested routes to build hierarchical routing structures.
- Outlet Component: Use the `Outlet` component to act as a placeholder for child routes.
- Nested Components: Organize nested components and routes within the parent route.

Protected Routes:

- Authentication/Authorization: Implement protected routes to restrict access to specific parts of your application.
- Security Considerations: Secure sensitive data and information in your application.

Using APIs:

- Data Fetching: Fetch data from APIs using the `fetch` method.
- Response Handling: Parse JSON responses using `response.json()`.
- useState and useEffect: Use state and the `useEffect` hook to manage fetched data.
- Modifying Data: Modify and format data as needed before rendering.

Resources:

- [MDN Fetch API Documentation]()
- [JSONPlaceholder - A Fake REST API]()

Environment Variables:

- Configuration Management: Use environment variables to manage configuration settings and sensitive information.
- .env Files: Store environment variables in a `.env` file in the project's root.
- Prefixing: Prefix environment variables with `REACT_APP_`.

Git-ignoring .env Files:

- Gitignore: Add `.env` files to the `.gitignore` file to prevent accidental commits, especially in public repositories.

Handling Sensitive Information:

- Security: Be cautious when storing sensitive information in environment variables.
- Client-Side Limitations: Understand that client-side applications expose environment variables, so use proxy servers or serverless functions for added security.

Resources:

- Create React App - Environment Variables
- GitHub Gitignore Documentation

Deploying a React app:

- Deployment Platforms: Options like Netlify and Vercel are popular choices for deploying React apps.
- Creating a Production Build: Use the `npm run build` script to create a production-ready build of your app.
- Platform-specific Instructions: Follow the deployment instructions provided by your chosen platform.

Resources:

- Netlify Deployment Guide
- Vercel Deployment Guide

Lesson 2.3

Performance Optimization in React

Performance optimization is crucial for delivering a smooth and responsive user experience in React applications. This lesson introduces various techniques and hooks to optimize your React applications.

Key Techniques for Performance Optimization:

1. React.memo:
    - Purpose: Prevents unnecessary re-renders of functional components when their props have not changed.
    - Use Cases: Useful for optimizing components that are computationally expensive or frequently updated.
    - Example: `React.memo(MyComponent)`
2. Lazy-Load Components:
    - Purpose: Loads components only when they are needed, reducing the initial load time of your application.
    - Use Cases: Ideal for optimizing large applications with many routes.
    - Example: `const MyComponent = React.lazy(() => import('./MyComponent'))`
3. useCallback:
    - Purpose: Memoizes callback functions to prevent unnecessary re-renders when parent components update.
    - Use Cases: Effective for optimizing components that receive callback functions as props.

- Example: `const memoizedCallback = useCallback(() => { /* callback logic */ }, [dependencies])`

4. useMemo:
   - Purpose: Memoizes expensive calculations to avoid recomputation when dependencies remain the same.
   - Use Cases: Useful for optimizing calculations or data transformations.
   - Example: `const memoizedValue = useMemo(() => { /* calculation logic */ }, [dependencies])`
5. useTransition (React version 18):
   - Purpose: Improves perceived performance by coordinating updates to multiple state pieces according to priority.
   - Use Cases: Enhances user experience when handling concurrent state updates.
   - Example: `const [isPending, startTransition] = useTransition()`

Additional Resources:

1. Mobx for React State Management:
   - Mobx Official Documentation
   - Learn about Mobx, an external state management library that can improve the performance of React applications.
2. Redux Toolkit for State Management:
   - Redux Toolkit Documentation
   - Explore Redux Toolkit, a simplified way to manage complex state in React applications.

By implementing these performance optimization techniques and choosing the right state management approach, you can create efficient and responsive React applications that provide a seamless user experience.