

Learning the Fundamental Concept of React Architecture

By Partha Roy

Lead Product Engineer @ Fasal

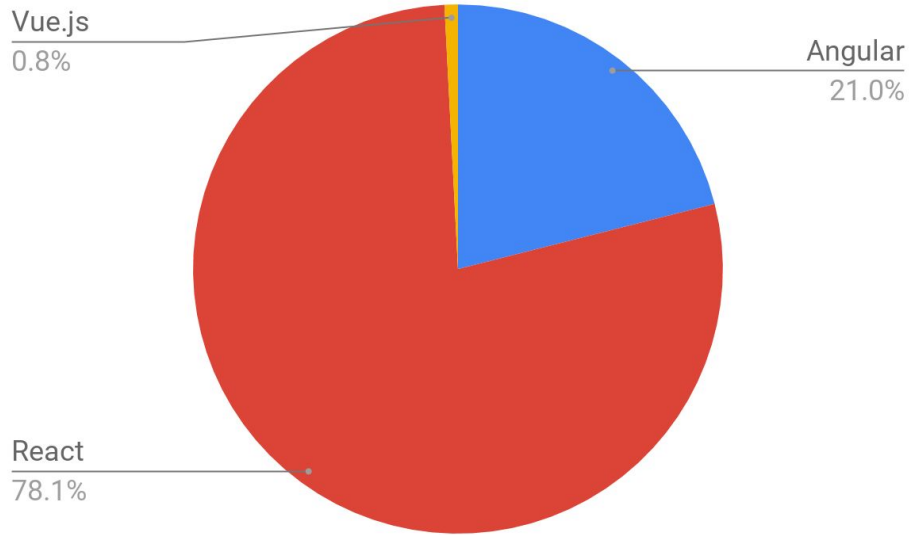
Frontend Technologies

History of Major Frontend Libraries and Frameworks

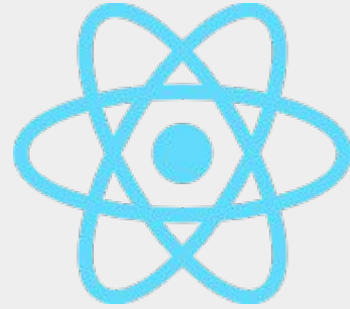
- Vanilla Javascript
- jQuery (2006)
- AngularJS (2010)
- React (2013)

Problems that React Solved

1. Code Reusability.
2. Project Maintenance.
3. Faster Rendering.
4. It is SEO friendly
5. Guarantees stable code
6. Backed by a strong community



What is React ?



A Declarative JavaScript library for building user interfaces using Component-Based Architecture.

Why React is called a library, not a framework?

Is it a Library ?

- React is distributed as a single npm install. It does one thing and does it very well.
- React by itself doesn't do all the pieces. Instead, it intends to be bolted onto an existing stack.
- React is non-invasive. For almost everything it does, if you would rather use something else, you can. Most of its features are opt in. Heck, even the render itself is opt in.

Is it a Framework ?

- About 99% of what you want to do on the client can be done with only React as a library (and react specific components distributed as part of its ecosystem).
- It is a full-featured dom manipulation and state engine
- It has some strongly opinionated ideas about how companion libraries should work (flux specification, graphql specification)

It's Depends on You.

Setup

Prerequisites

[Node JS](#)

Code IDE (VS Code / Sublime)

Using npx

npx create-react-app your-app-name

(npx is a package runner tool that comes with **npm 5.2+ and higher**, see instructions for older npm versions)

Using npm

npm init react-app my-app

npm init <initializer> is available in npm 6+

Running the Project

cd your-app-name

npm start

[Read More](#)

Important ES6 Shortcuts

```
//Arrow function
const sum = (a, b) => a + b
console.log(sum(2, 6)) // prints 8

//Default parameters
function print(a = 5) {
  console.log(a)
}
print() // prints 5
print(22) // prints 22

//scope of let
let a = 3
if (true) {
  let a = 5
  console.log(a) // prints 5
}
console.log(a) // prints 3

// use of const
// can be assigned only once:
const a = 55
a = 44 // throws an error
```

```
//Multiline string
console.log(`
  This is a
  multi | line string
`)

//Template strings
const name = 'Leon'
const message = `Hello ${name}`
console.log(message) // prints
"Hello Leon"

//Destructuring array
let [a, b] = [3, 7];
console.log(a); // 3
console.log(b); // 7

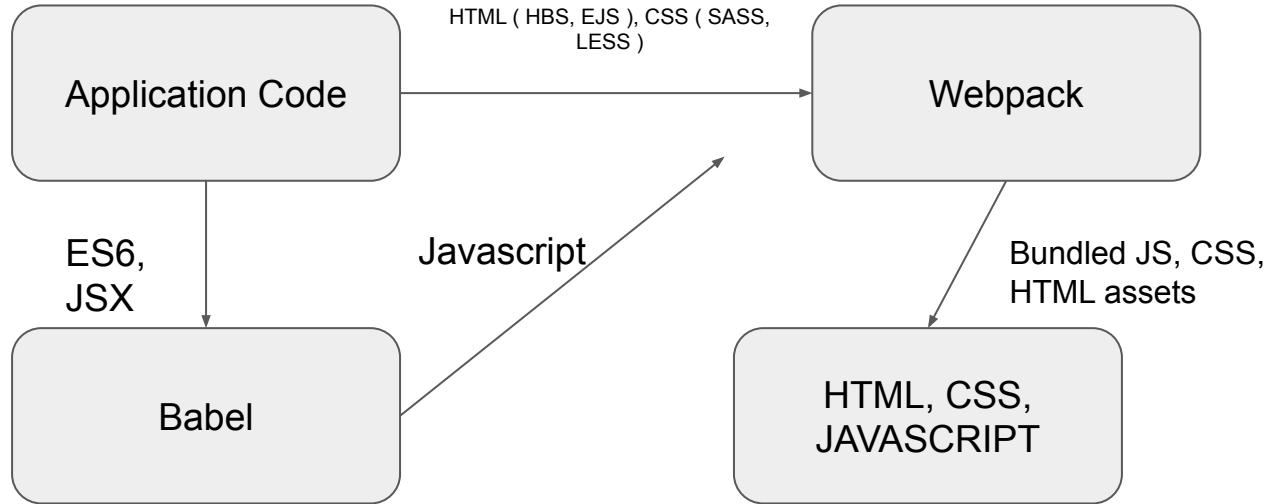
//Destructuring object
let obj = { a: 55, b: 44 };
let { a, b } = obj;
console.log(a); // 55
console.log(b); // 44
```

```
//Object property assignment
const a = 2
const b = 5
const obj = { a, b }
console.log(obj) // prints { a: 2, b: 5 }

//Spread operator (Array)
const a = [ 1, 2 ]
const b = [ 3, 4 ]
const c = [ ...a, ...b ]
console.log(c) // [1, 2, 3, 4]

//Spread operator ( Object )
const a = { firstName: "Barry", lastName:
"Manilow"}
const b = {
  ...a,
  lastName: "White",
  canSing: true,
}
console.log(a) // {firstName: "Barry",
lastName: "Manilow"}
console.log(b) // {firstName: "Barry",
lastName: "White", canSing: true}
```

Tech behind a React Application



React Devtools

React Developer Tools is a Chrome DevTools extension for the open-source React JavaScript library. It allows you to inspect the React component hierarchies in the Chrome Developer Tools.

[For Chrome](#)

[For Mozilla](#)

Components

JSX

```
const element = <h1>Hello, world!</h1>;
```

This tag syntax is neither a string nor HTML.

It is called JSX, and it is a **syntax extension to JavaScript**. JSX is somewhat like a template language, but it comes with the full power of JavaScript.

JSX produces React “elements”.

Let's Create a Component

A component represent the **part of user interface**. Components are **reusable** and can be use in anywhere in user interface.

We will learn about,

1. Creating a simple component
2. Adding Styling to that component
3. Creating a Parent & a Child Component
4. Passing Data between Parent, Child and Siblings.

How UI is structured using Components



props

```
// Heading component declaration - Class Component
class Heading extends React.Component {
  render() {
    return <h1>Hello {this.props.name}</h1>;
  }
}

// Heading component declaration - Functional Component
const Heading = (props) => {
  return <h1>Hello {props.name}</h1>;
}

// using Heading component by passing props
const element = <Heading name="Sara" />;
```

“**props**” is short for “**properties**”

props are passed into the component from parent components.

They are like **arguments passed to functions**, arguments can only be changed by what calls the functions, similarly here props can only be changed by parent classes.

A component can also have default props, so if a prop isn't passed through it can still be set.

React component that only uses props (and not state) called as “pure components”

state

```
class StateExample extends React.Component {
  constructor() {
    super();
    this.state = {
      count: 0,
    };
  }
  render(){
    return <div>
      <p>{this.state.count}</p>
      <button onClick={
        () => {
          // setting the state value
          this.setState({
            count: this.state.count + 1
          });
          // never do this
          this.state.count = this.state.count + 1;
        }
      }> Increment Counter </button>
    </div>
  }
}
```

Like props, state holds information about the component. When a component needs to **keep track of information between renderings the component itself can create, update, and use state**. State is **created** in the component, and **changeable**.

It is tempting to write,
this.state.count = this.state.count + 1.

Do not do this! React cannot listen to the state getting updated in this way, so your component will not re-render. Always use setState.

Functional Component or Stateless component

1. It is simple javascript functions that simply returns html UI
2. It is also called “stateless” components because they simply accept data and display them in some form that is they are mainly responsible for rendering UI.
3. It accept properties(props) in function and return html(JSX)
4. It gives solution without using state
5. There is no render method used in functional components.
6. These can be typically defined using arrow functions but can also be created with the regular function keyword.

```
//with arrow function
import React from "react";
const Person = (props) => (
  <div>
    <h1>Hello, {props.name}</h1>
  </div>
);
export default Person;
```

```
//without arrow function
import React from "react";
function Person(props){
  return (
    <div>
      <h1>Hello, {props.name}</h1>
    </div>
  )
};

export default Person;
```


Class Component or Stateful Component

1. It is regular ES6 classes that extends component class from react library
2. Also known as “stateful” components because they implement logic and state.
3. It must have render() method returning html
4. It has complex UI Logic
5. You pass props to class components and access them with this.props

```
import React, { Component } from "react";
class Person extends Component {
  constructor(props){
    super(props);
    this.state = {
      name: "bikash";
    }
  }

  render() {
    return (
      <div>
        <h1>Hello {this.state.name}</h1>
      </div>
    );
  }
}
export default Person;
```


Which one should we use ?

Benefits you get by using functional components in React:

1. Functional component are much **easier to read and test** because they are plain JavaScript functions without state or lifecycle-hooks
2. It has **less code** which makes it more readable
3. It will get easier to separate container and presentational components because you need to think more about your component's state if you don't have access to `setState()` in your component

If you are writing a **presentational component** which doesn't have its own state or needs to access a lifecycle hook, use functional component as much as possible. For state management you can use class component.

super()



```
class Checkbox extends React.Component {  
  constructor(props) {  
    // Can't use `this` yet  
    super(props);  
    // Now we can  
    this.state = { isOn: true };  
  }  
}
```

In JavaScript, `super` refers to the parent class constructor. (In our example, it points to the `React.Component` implementation.)

Importantly, you can't use **this** in a constructor until after you've called the parent constructor. JavaScript won't let you.

super(props)



```
class Button extends React.Component {  
  constructor(props) {  
    super(); // We forgot to pass props  
    console.log(props); // {}  
    console.log(this.props); // undefined  
  }  
}
```

Passing props down to super is necessary so that the base `React.Component` constructor can initialize `this.props`

However even if you forget to pass props to `super()`, React would still set them right afterwards.

There's a catch, React would later assign `this.props` after your constructor has run. But `this.props` would still be undefined between the super call and the end of your constructor.

So it's advisable to always use `super(props)` so that we don't get into this kind of issue.

Conditional Rendering, Loops and Virtual DOM

Conditional Rendering

We will learn about how can we hide or display elements based on certain conditions.

Also Nested Conditional Rendering.

Loops

We will learn about how we can render multiple items using a loop.

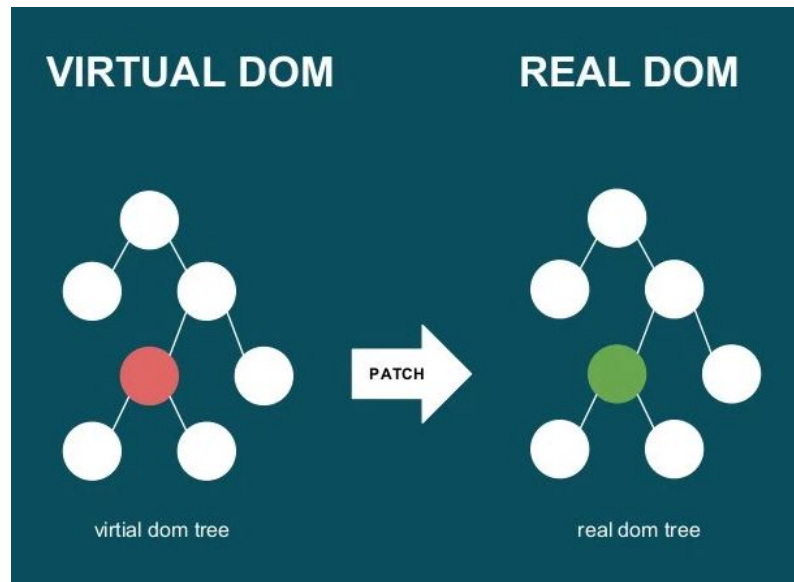
The Virtual DOM

Most JavaScript frameworks update the DOM much more than they have to. Let's say that you have a list that contains ten items. You check off the first item. Most JavaScript frameworks would rebuild the entire list. That's ten times more work than necessary! Only one item changed, but the remaining nine get rebuilt exactly how they were before.

In React, for every DOM object, there is a corresponding “virtual DOM object.” **A virtual DOM object is a representation of a DOM object, like a lightweight copy.**

Here's what happens when you try to update the DOM in React:

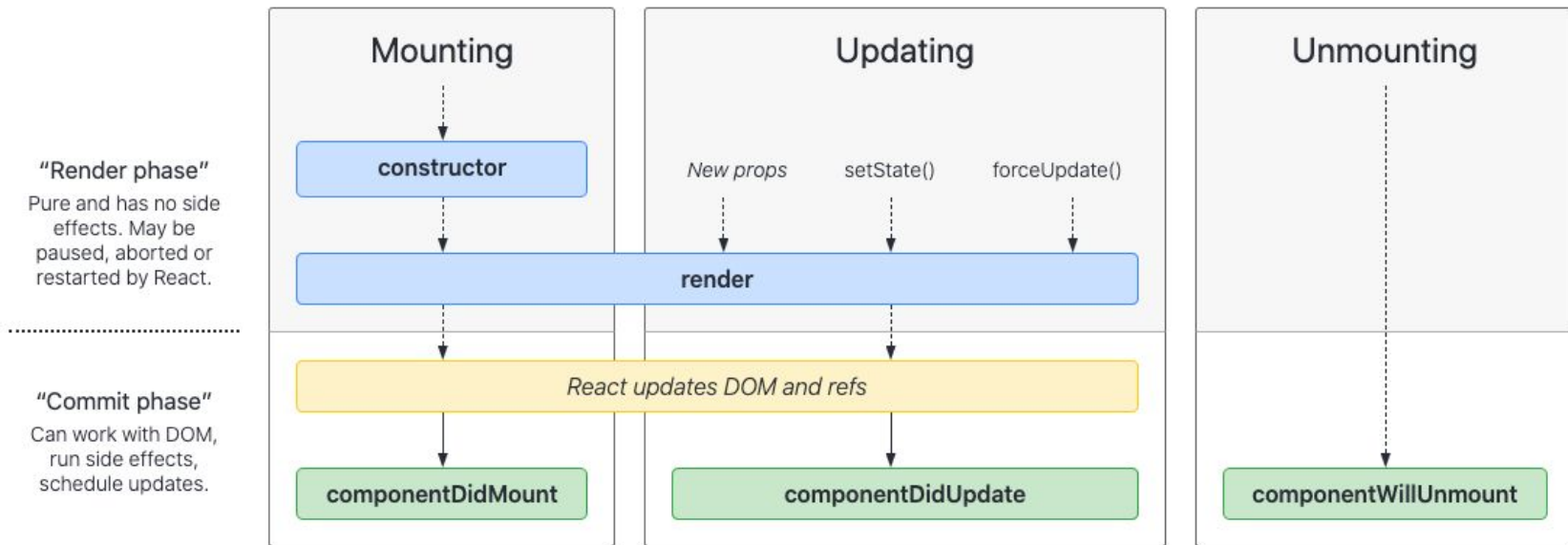
1. The entire virtual DOM gets updated.
2. The virtual DOM gets compared to what it looked like before you updated it. React figures out which objects have changed.
3. The changed objects, and the changed objects only, get updated on the real DOM.
4. Changes on the real DOM cause the screen to change.



Lifecycle

React Lifecycle

Each component has several “lifecycle methods” that you can override to run code at particular times in the process.



Lifecycle methods

Mounting

These methods are called in the following order when an instance of a component is being created and inserted into the DOM:

- ***constructor()***
- ***getDerivedStateFromProps()***
- ***render()***
- ***componentDidMount()***

Updating

An update can be caused by changes to props or state.

These methods are called in the following order when a component is being re-rendered:

- ***getDerivedStateFromProps()***
- ***shouldComponentUpdate()***
- ***render()***
- ***getSnapshotBeforeUpdate()***
- ***componentDidUpdate()***

Unmounting

This method is called when a component is being removed from the DOM:

- ***componentWillUnmount()***

Event Architecture

React Fiber Architecture

React Fiber is an ongoing reimplementation of React's core algorithm. It is the culmination of over two years of research by the React team.

reconciliation

The algorithm React uses to diff one tree with another to determine which parts need to be changed.

update

A change in the data used to render a React app. Usually the result of `setState`. Eventually results in a re-render.

[Read More](#)

Dealing with React Events

We will learn how React *reacts* when any event is triggered on the client side on the DOM.

Designing the State of Our Application

We will learn how can we implement a simple form and handle the inputs using the concepts of State, Props and Events.

Controlled Components

In a controlled component, the **form data is handled by the state within the component**. The **state within the component serves as “the single source of truth”** for the input elements that are rendered by the component.

Uncontrolled Components

Uncontrolled components act more like traditional HTML form elements. The data for each input element is stored in the DOM, not in the component. Instead of writing an event handler for all of your state updates, you use a **ref** to retrieve values from the DOM.

refs

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // create a ref to store the textInput DOM element
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // Explicitly focus the text input using the raw DOM API
    // Note: we're accessing "current" to get the DOM node
    this.textInput.current.focus();
  }

  render() {
    // tell React that we want to associate the <input> ref
    // with the `textInput` that we created in the constructor
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

Refs provide a way to access DOM nodes or React elements created in the render method.

- A few key points regarding refs:
- Refs are created using `React.createRef()`.
- Refs are attached to input elements using the `ref` attribute on the element in question.
- Refs are often used as instance properties on a component. The ref is set in the constructor (as shown above) and the value is available throughout the component.
- You cannot use the `ref` attribute on functional components because an instance is not created.
- BUT, You can use a `ref` attribute inside a functional component.

Hooks

Hooks

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

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 are JavaScript functions, but they impose two additional rules:

1. Only call Hooks at the top level. Don’t call Hooks inside loops, conditions, or nested functions.
2. Only call Hooks from React function components. Don’t call Hooks from regular JavaScript functions. (There is just one other valid place to call Hooks — your own custom Hooks. We’ll learn about them in a moment.)

Different Hooks

State Hook

We call `useState` inside a function component to add some local state to it.

React will preserve this state between re-renders. `useState` returns a pair: the current state value and a function that lets you update it. You can call this function from an event handler or somewhere else.

It's similar to `this.setState` in a class, except it doesn't merge the old and new state together. (We'll show an example comparing `useState` to `this.state` in Using the State Hook.)

Effect Hook

When you call `useEffect`, you're telling React to run your “effect” function after flushing changes to the DOM.

Effects are declared inside the component so they have access to its props and state. By default, **React runs the effects after every render — including the first render.**

Fetching Data by Making API Requests

You can use any AJAX library you like with React. Some popular ones are Axios, jQuery AJAX, and the browser built-in **window.fetch**.

You should populate data with AJAX calls in the **componentDidMount** lifecycle method. This is so you can use `setState` to update your component when the data is retrieved.

```
fetch('API SERVICE URL')  
  .then(response => response.json())  
  .then( ( data ) => {  
    // ... do stuffs  
  } );
```

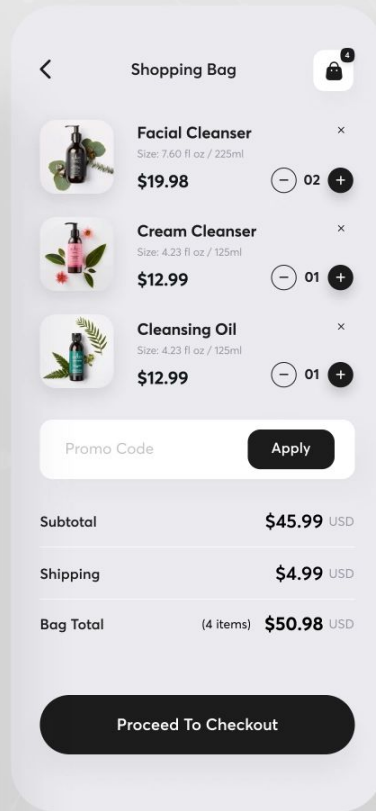
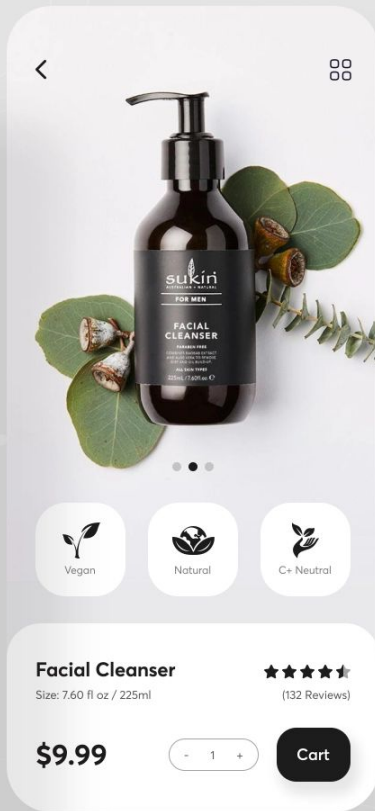
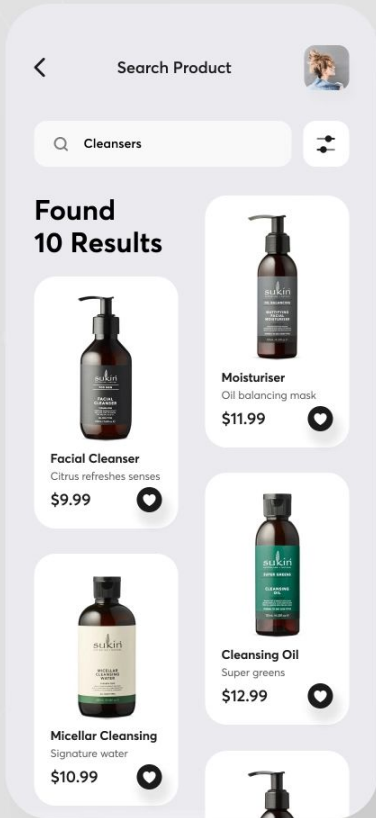
Developing a Project

Objective

Ideating and Building a Single Page Application using the concepts of React JS that we have learnt earlier.

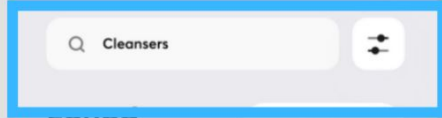
Problem Statement

*Creating a E-commerce Website
with Product Listing, Product
Details and Cart Page.*

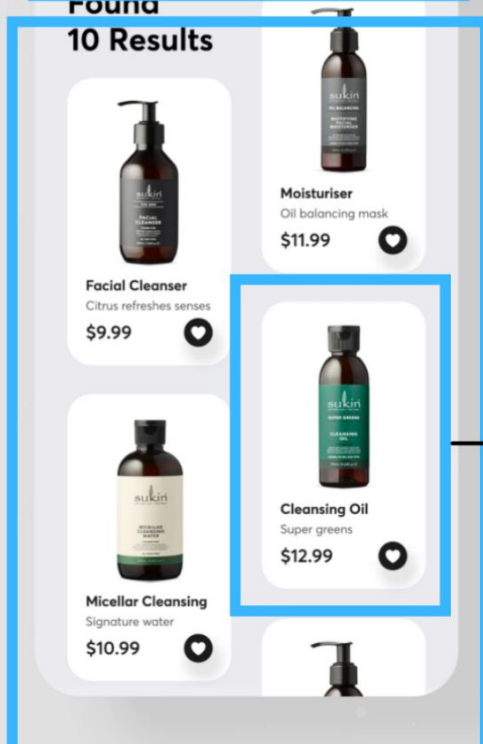




Navbar



Search & Filter



Product Listing

Product Item

Components List

Single Page Application

A single-page application is a web application or website that interacts with the user by **dynamically rewriting the current web page with new data from the web server**, instead of the default method of the browser **loading entire new pages**.

- Great User Experience
- Lacks Performance
- Lacks Search Engine Optimization
- Non-Scalable

SPA Architecture

Single Page Applications Work Differently



Choosing the right tools

- How Complex Is The Project?
- Is It An Interactive Application Or Just A Static Collection Of Web Pages?
- How Scalable Is The Proposed Application?
- How High Or Low Maintenance Is The Application?
- Is the Framework Cross Browser Compatible?
- What Do My Instincts Say?

Our Tech Stack

For building the Web App

- React - UI Library
- React Router (For Routing)
- FakeStoreAPI for Open Source Products API
- REST - Protocol

Questions ?

Let's build