Intro to React

Intro

ReactJs is “a javascript library for building user interfaces” is the official one-liner introduction to ReactJs.

The user interface is a multi-option playground where the user can do different things, and libraries like React helps us create this playground.

What is React

Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes. Declarative views make your code more predictable and easier to debug

Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs. Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

Some features

React is an abstraction away from the DOM

Encourages you to think of your application and UI in terms of state, rather than UI manipulations

Allows a simplified mental model for data flow

Mix and match components to build UIs

SPAs

https://www.bloomreach.com/en/blog/2018/07/what-is-a-single-page-application.html

https://dzone.com/articles/how-single-page-web-applications-actually-work

https://www.adcisolutions.com/knowledge/whats-difference-between-single-page-application-and-multi-page-application

SPAs

a Single Page Application is a type of web application that requires only a single page to be loaded into the browser.

You might be thinking, what does this even mean? How could an application with just a single-page be of use to anyone in the world?

The answer is simple. Single page web applications are built around the concept of dynamically rewriting the contents of that single page. This is different from loading pre-rendered pages from the server.

SPAs

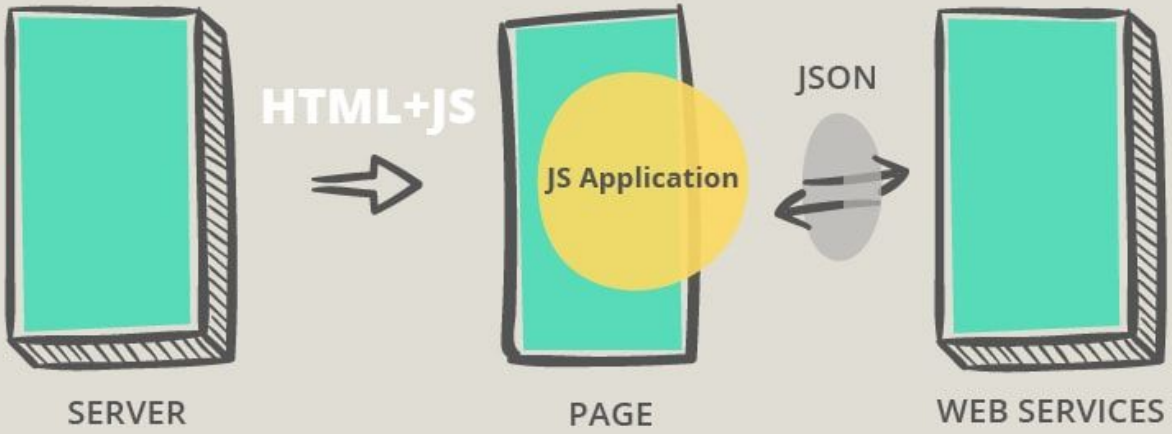
In single-page web applications, when the browser makes the first request to the server -

1. the server sends back the index.html file. And that's basically it. That's the only time an HTML file is served.

2. The HTML file has a script tag for the .js file which is going to take control of the index.html page.

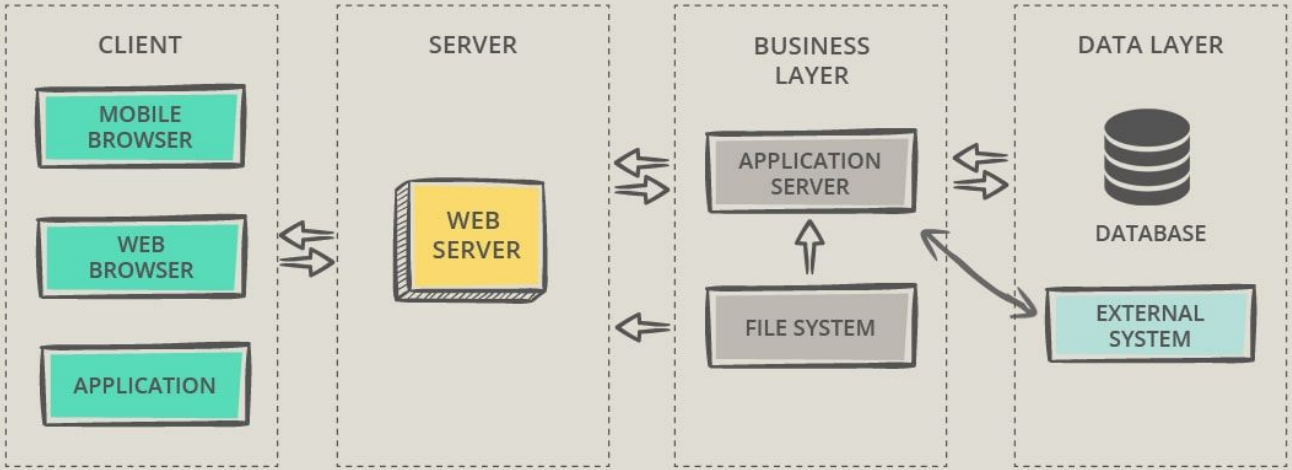
3. All subsequent calls return just the data, usually in JSON format. The application uses this JSON data to update the page dynamically. However, t**he page never reloads.**

SINGLE PAGE WEB APP
ARCHITECTURE



EXISTEK

NODE.JS WEB APPLICATION ARCHITECTURE



EXISTEK

SPAs - some advantages

SPA is fast, as most resources (HTML+CSS+Scripts) are only loaded once throughout the lifespan of application. Only data is transmitted back and forth. It avoids loading duplicate HTML over and over again.

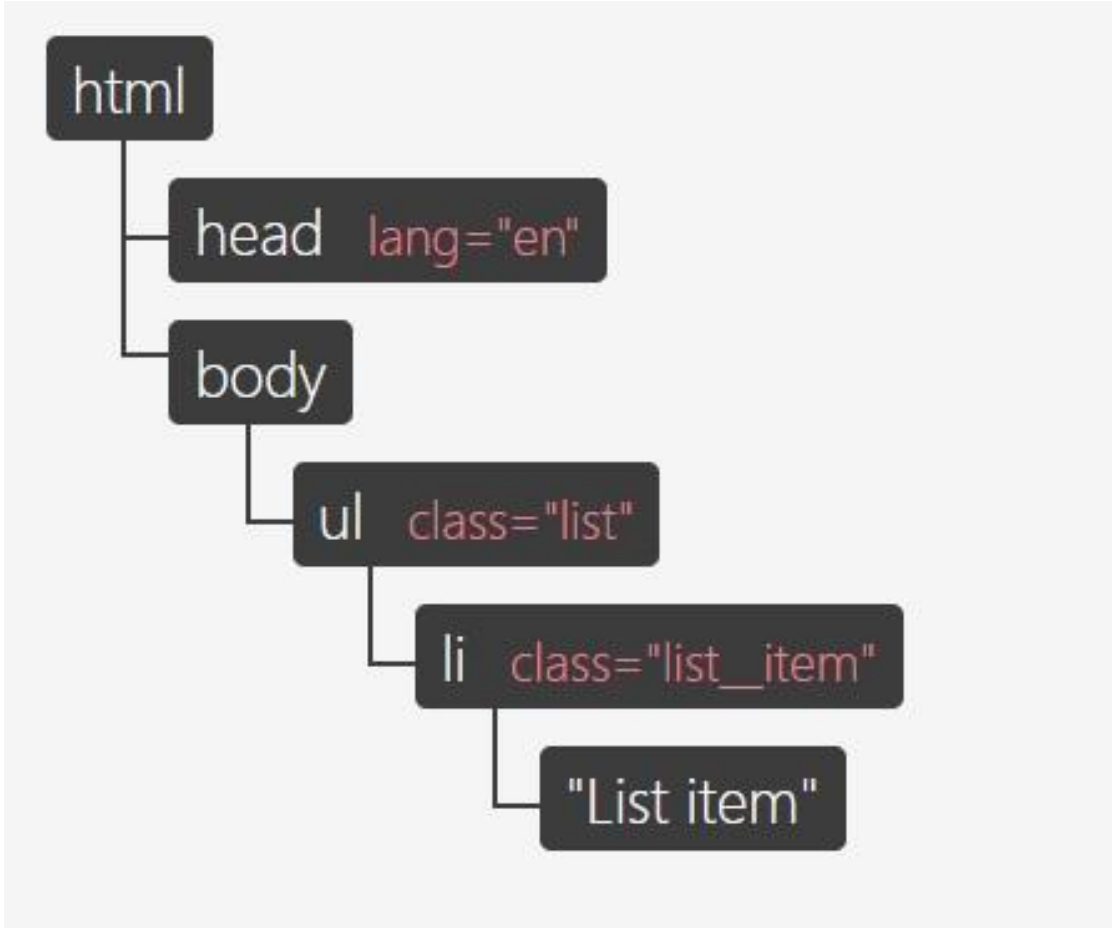
SPAs are easy to debug with Chrome, as you can monitor network operations, investigate page elements and data associated with it.

It’s easier to make a mobile application because the developer can reuse the same backend code for web application and native mobile application.

SPA can cache any local storage effectively. An application sends only one request, store all data, then it can use this data and works even offline.

Another overlooked benefit is production deployment of Single Page Applications. SPAs are super easy to deploy. When you build an SPA for production, you typically end up with one HTML file, one CSS bundle, and a JavaScript bundle.

```
<!doctype html>
<html lang="en">
 <head></head>
 <body>
    <ul class="list">
        <li class="list__item">List item</li>
    </ul>
  </body>
</html>
```


html
head lang="en"
body
ul class="list"
li class="list__item"
"List item"

```
const listItemOne = document.getElementsByClassName("list__item")[0];
listItemOne.textContent = "List item one";

const list = document.getElementsByClassName("list")[0];
const listItemTwo = document.createElement("li");
listItemTwo.classList.add("list__item");
listItemTwo.textContent = "List item two";
list.appendChild(listItemTwo);
```

```
const vdom = {
    tagName: "html",
    children: [
        { tagName: "head" },
        {
            tagName: "body",
            children: [
                {
                    tagName: "ul",
                    attributes: { "class": "list" },
                    children: [
                        {
                            tagName: "li",
                            attributes: { "class": "list__item" },
                            textContent: "List item"
                        } // end li
                    ]
                } // end ul
            ]
        } // end body
    ]
} // end html
```

Virtual DOM

DOM manipulation is the heart of the modern, interactive web. Unfortunately, it is also a lot slower than most JavaScript operations.

This slowness is made worse by the fact that most JavaScript frameworks update the DOM much more than they have to.

Inefficient updating has become a serious problem.

Virtual DOM

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.

A virtual DOM object has the same properties as a real DOM object, but it lacks the real thing’s power to directly change what’s on the screen.

Manipulating the DOM is slow. Manipulating the virtual DOM is much faster, because nothing gets drawn onscreen. Think of manipulating the virtual DOM as editing a blueprint, as opposed to moving rooms in an actual house.

Virtual DOM

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

1. Every single virtual DOM object gets updated. (This sounds incredibly inefficient, but the cost is insignificant because the virtual DOM can update so quickly.)
2. The virtual DOM gets compared to a snapshot that was taken right before the update. React figures out which objects have changed. This is called "diffing".
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.

Components

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

Components

Components === State Machines

React thinks of UIs as simple state machines.In React, you simply update a component's state, and then render a new UI based on this new state. React takes care of updating the DOM for you in the most efficient way

Components === Functions

Given the same input values, a component will return the same UI output. Often described as UI = f(state)

Props and State

The main responsibility of a Component is to translate raw data into rich HTML. With that in mind, the props and the state together constitute the raw data that the HTML output derives from.

You could say props + state is the input data for the render() function of a Component, so we need to zoom in and see what each data type represents and where does it come from.

Props and State

props (short for properties) are a Component's configuration, its options if you may. They are received from above and immutable as far as the Component receiving them is concerned.

A Component cannot change its props, but it is responsible for putting together the props of its child Components.

The **state** starts with a default value when a Component mounts and then suffers from mutations in time (mostly generated from user events).

A Component manages its own state internally, but— has no business fiddling with the state of its children. You could say the state is private.

Lifecycle methods

Lifecycle methods are various methods which are invoked at different phases of the lifecycle of a component.

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

One-way data flow in React:

For each piece of state in your application:

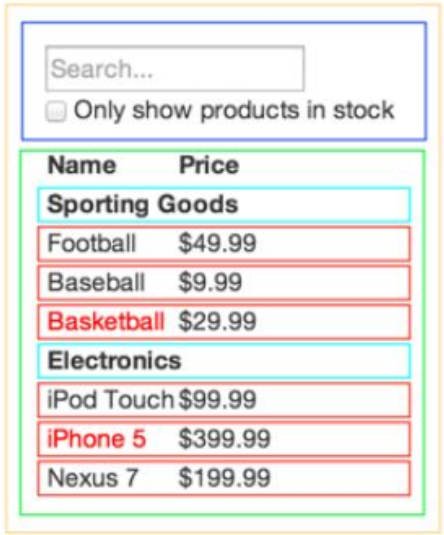
Identify every component that renders something based on that state.

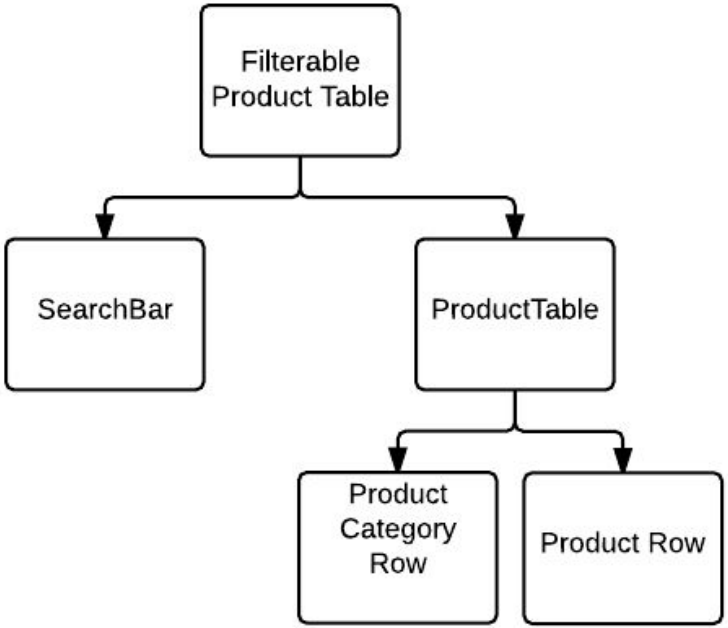
Find a common owner component (a single component above all the components that need the state in the hierarchy).

Either the common owner or another component higher up in the hierarchy should own the state.

If you can’t find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common owner component.

Thinking in React





Forms and Controlled Components

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

Forms and Controlled Components

This means your data (state) and UI (inputs) are always in sync. The state gives the value to the input, and the input asks the Form to change the current value.

This also means that the form component can respond to input changes immediately; for example, by:

- in-place feedback, like validations
- disabling the button unless all fields have valid data
- enforcing a specific input format, like credit card numbers

But if you don’t need any of that you can use uncontrolled components.-

Controlled text - https://codepen.io/gaearon/pen/VmmPgp?editors=0010

Controlled select - https://codepen.io/gaearon/pen/JbbEzX?editors=0010

Multiple inputs - https://codepen.io/gaearon/pen/wgedvV?editors=0010

Presentational components

Are concerned with how things look.

May contain both presentational and container components** inside, and usually have some DOM markup and styles of their own.

Have no dependencies on the rest of the app, such as Redux stores.

Don’t specify how the data is loaded or mutated.

Receive data and callbacks exclusively via props.

Good to use for reusable/generic components.

Rarely have their own state (when they do, it’s UI state rather than data). (e.g. tabs)

Container components

Are concerned with how things work.

May contain both presentational and container components** inside but usually don’t have any DOM markup of their own except for some wrapping divs, and never have any styles.

Provide the data and behavior to presentational or other container components.

Provide functionality in the form of callbacks to the presentational components.

Are often stateful, as they tend to serve as data sources.

Functional vs. Class based components

Before Hooks, functional components lacked state and lifecycle methods. So they were used along with class-based components.

If you choose to use class-based components, remember it is still a good idea to use functional (stateless) components where possible. Follow the rule: If you ever have a class component with **only a render method** – you should always make it a functional component.

Or you could just use Hooks and have no class-based components at all!

Functional vs. Class based components

You can choose whether to learn class-based components or Hooks first. But for the foreseeable future it would be a good idea to know how both work.

It doesn’t even have to be one or the other - an application can be migrated to functional components, one component at a time.

The general rule, no matter what paradigm you start out with - don’t add anything to your component until you need it. And always be looking for ways to make your code more modular.

Resources

Official React Docs - https://reactjs.org/docs/getting-started.html

Good interactive course for beginners - https://scrimba.com/playlist/p7P5Hd

Udemy courses (Stephen Grider)

Misc:

Roadmap (to explore React Ecosystem) - https://cdn-images-1.medium.com/max/906/0*DWU1i2q8jPrOdXsB.png

Props and State - https://lucybain.com/blog/2016/react-state-vs-pros/
https://github.com/uberVU/react-guide/blob/master/props-vs-state.md

Forms - https://goshakkk.name/on-forms-react/

More about Virtual DOM - https://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/
https://bitsofco.de/understanding-the-virtual-dom/

React Ecosystem

Entirely possible to build a React app with no other dependencies, but most apps use a variety of additional libraries for specific capabilities.

Good news: can pick exactly the libraries you need for your use cases.

Bad news: need to pick exactly the libraries you need for your use cases.

Codepen examples

Hello world - https://codepen.io/airen/pen/yzLOjK

Simple button with click counter - https://codepen.io/lbain/pen/ENpzBZ

State + Conditional rendering example + events - https://codepen.io/CharlesKenney/pen/eRBgmZ

Lists - https://codepen.io/gaearon/pen/jrXYRR?editors=0011, https://codepen.io/BigNom/pen/BZZOgX

Complex example to show component organization - https://codepen.io/ahmetaksungur/pen/abzGvrY

Thanks.