Introduction to React

A JavaScript library for building user interfaces

It was developed by Facebook around 2011/2012 and was released as an open source project in 2013.

Claims

- 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.
- Build encapsulated components that manage their state, then compose them to make complex UIs.
- We don’t make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

Components

- HTML and CSS focused on decomposing UI into blocks and groups of elements for content, semantics, and styling.
- The same is true for React Components.
- We can now also look at our pages as collections of React Components.
- Each of these components can be reused which will help us reduce code.

Class versus Function components

- Two styles of React components
- `Class` components came first and most documentation, tutorials, and Stack Overflow answers are using `class`
- The other style, `functions/hooks` is the future.

We will teach you hooks

Following our guide of teaching you currently deployed technolody but with an eye on the future...

... we are teaching you hooks. However, if you start a `class` based project you'll be able to pick that up.

What does a Component look like

The rules of a React component are:

- It must be named following the `PascalCase` style.
- It must be a function (`function` style or arrow-style) that returns JSX.
- The JSX returned must consist of *exactly* one main element, with other elements contained within. We'll see this more later.

What is JSX?

JSX is an extension of JavaScript that allows us to use an HTML-like syntax in our JavaScript code that will be turned into DOM at run-time. By using JSX we can dynamically generate content and use a UI as state style.

Simplest React Component

```
function HelloWorld() {
  return <p>Hello, World!</p>
}
```

HTML in our JavaScript/ TypeScript?!?

Transpiling

```
return <p>Hello, World!</p>
```

becomes

```
function HelloWorld() {
  return React.createElement('p', null, 'Hello, World')
}
```

How does our template help us use React Components?

- Our template for React is inspired by `vite` which is a tool for generating web based projects.
- React Components can represent the **entire** web page, or be mixed in with static content of the page.

```
return <p>Hello, World!</p>
```

Let's create a new React project

```
degit suncoast-devs/react-project-template ReactArticles
```

HTML file

The template SDG uses will generate an `index.html` file that looks like the following (only the `<body>` is shown and only the relevant parts)

```
<body>
  <div id="root"></div>
</body>
```

Where is the User Interface!??

If we rendered this without JavaScript it would be an empty page. It is thus up to JavaScript to connect our React code to our HTML.

User Interface all in JavaScript

In our SDG template we include an `main.tsx` -- this script loads React and a component we provide named App

```
import React from 'react'
import ReactDOM from 'react-dom'
import './index.css'
import App from './App'

ReactDOM.render(<App />, document.getElementById('root'))
```

ReactDOM is the glue between HTML and JavaScript

React connects an existing `DOM` element to a component with the `ReactDOM` method. Here we state that the element with the `id` of `root` will be replaced with the component `App` that we `import` from the `App.tsx` file.

Typically we will not have to adjust the `index.html` or the `main.tsx` files. We will start writing our code in our `App.tsx` file.

JSX Files and *transpiling*

You may have noticed that we define our React Components in files that end in `.tsx` instead of `.ts`.

The `.tsx` file name extension allows our editors and our code management tools to know we are using the `JSX` extensions.

Build a simple React application

This is some sample HTML we will work to create and learn how React Components can help simplify our code.

```
<div class="all-main-content">
  <main>
    <article class="intro-article">
      <h2 className="article-title">SDG Announces Hackathon!</h2>
      <p>
        SDG announces the 2020 Summer Hackathon. Join us for an exciting weekend
      </p>
      <a class="read-more" href="#here">
        read more about SDG Announces Hackathon!
      </a>
    </article>

    <article class="intro-article">
      <h2 className="article-title">
        Student Graduation is Right Around the Corner
      </h2>
      <p>Our next cohort of students will be graduating in just over a week.</p>
      <a class="read-more" href="#here">
        read more about Student Graduation is Right Around the Corner
      </a>
    </article>
  </main>
</div>
```

Start with `hard coded` content

We'll take our HTML and place it *ALL* inside the `render` method of our App

```
import React from 'react'

export function App() {
  return (
    <div className="all-main-content">
      <main>
        <article className="intro-article">
          <h2 className="article-title">SDG Announces Hackathon!</h2>
          <p>
            SDG announces the 2020 Summer Hackathon. Join us for an exciting
            weekend
          </p>
          <a className="read-more" href="#here">
            read more about SDG Announces Hackathon!
          </a>
        </article>

        <article className="intro-article">
          <h2 className="article-title">
            Student Graduation is Right Around the Corner
          </h2>
          <p>
            Our next cohort of students will be graduating in just over a week.
          </p>
          <a className="read-more" href="#here">
            read more about Student Graduation is Right Around the Corner
          </a>
        </article>
      </main>
    </div>
  )
}
```

`class` versus `className`

You may have noticed we used `className=` everywhere in the code instead of `class=`.

This is because `JSX` uses the `DOM` names for properties whereas `HTML` uses a more generic name.

In the `DOM` it is the `className` property that `class=` becomes. So we will be using `className` for our class names in JSX.

Don't worry, our `console` will warn us of these types of errors when we make them.

Some CSS to help

Let's use this CSS to make the app look better

```css
@import url('https://fonts.googleapis.com/css2?family=Open+Sans&display=swap');

:root {
  font: 16px / 1 sans-serif;
}

html {
  height: 100%;
}

body {
  font-family: 'Open Sans', sans-serif;

  margin: 0;
  min-height: 100%;
  background-color: #f7f0da;
  color: #2f3737;
}

article {
  max-width: 20rem;
  padding: 1rem 3rem;
  font-size: 0.8rem;
}

article .read-more {
  font-size: 0.6rem;
}

article h2 {
  font-size: 1.5rem;
  color: #5a9090;
}

main {
  display: flex;
  flex-wrap: wrap;
}

.read-more {
  text-transform: uppercase;
  font-size: 0.8rem;
  font-weight: 800;
  text-decoration: none;
}
```

Extracting common code into a "component"

- We duplicate code for the `<article>` part of our page.
- React will allow us to make this a generic *component* and reuse it!

Creating a `NewsArticle` component.

Let's create a new file in the `components` directory and name it `NewsArticle.tsx`

NOTE: If you don't have a `src/components` *directory you should create that directory first. Since this is a common pattern you may want to chose to add this directory (and an empty* `.keep` *file within) to your template.*

Import React

We'll add this as the first line to tell JavaScript we are going to use `React` and it activates the `JSX` template process.

```
import React from 'react'
```

NOTE: With the latest version of `React` *and some tools we don't need this import. You may see some tutorials and examples omit this line. For* `vite`*, at the time of writing, we still need this import.*

Next, we will make our component:

```
export function NewsArticle() {
  // Code here
}
```

The `export` at the beginning of that line, tells JavaScript we wish to share this class outside of the file. We'll use that fact in just a moment.

Add a return

For now, as a test, we'll have it just make a `<div>` with some text

```
export function NewsArticle() {
  return <div>Something</div>
}
```

Prepare App to use NewsArticle

In App.tsx add:

```
import { NewsArticle } from './components/NewsArticle'
```

Use our `NewsArticle`

Add this right inside `<main>`

```
<NewsArticle />
```

Composition

This is the idea of `composition` -- we are now defining a component, that has its own content, that we can embed into another component, in this case the App.

Update the `NewsArticle`

Let's take one example of the news article we have and make it the `render` method's JSX.

```
import React from 'react'

export function NewsArticle() {
  return (
    <article className="intro-article">
      <h2 className="article-title">
        <p>FROM THE COMPONENT</p> Student Graduation is Right Around the Corner
      </h2>
      <p>Our next cohort of students will be graduating in just over a week.</p>
      <a className="read-more" href="#here">
        read more about Student Graduation is Right Around the Corner
      </a>
    </article>
  )
}
```

Three Articles

You should notice that our app now has **THREE** articles. The first comes from our `<NewsArticle/>` and the other two from the hardcoded elements.

Cleaning up

Let's remove the other two hardcoded `<article>`s leaving only our `<NewsArticle/>`

```
import React from 'react'
import { NewsArticle } from './components/NewsArticle'

function App() {
  return (
    <div className="all-main-content">
      <main>
        <NewsArticle />
      </main>
    </div>
  )
}
```

Make many NewsArticles!

We should only see one article listed. If we repeat the `<NewsArticle/>` we can have as many of the `<article>` structures as we want.

```
import React from 'react'
import { NewsArticle } from './components/NewsArticle'

function App() {
  return (
    <div className="all-main-content">
      <main>
        <NewsArticle />
        <NewsArticle />
        <NewsArticle />
        <NewsArticle />
      </main>
    </div>
  )
}

export default App
```

Reusable but not customizable

Component properties: props

We can add properties to our components by specifying them in a very similar way we would for regular HTML elements.

```
<NewsArticle
  title="SDG Announces Hackathon!"
  body="SDG announces the 2020 Summer Hackathon. Join us for an exciting weekend"
/>
<NewsArticle
  title="Student Graduation is Right Around the Corner"
  body="Our next cohort of students will be graduating in just over a week."
/>
<NewsArticle
  title="SDG Standardizes on React"
  body="React is the best library for learning front end Web"
/>
```

Hooray TypeScript!

Notice that we immediate get warnings and errors that some of these properties are unknown to the `<NewsArticle>` component.

Using props in a component

Properties added in the *USAGE* of a component are present to us inside a component via an argument we will name `props`

So our `title` is in a variable named `props.title` and our `body` is in a variable called `props.body`

In the places where we have hard coded data we can replace with variables

Interpolation of variables in the middle of JSX

To have values appear in our JSX we use `interpolation`

In JSX we write our variables inbetween { } characters.

We can replace all the hardcoded text with values from `props`

```
import React from 'react'

export function NewsArticle(props) {
  return (
    <article className="intro-article">
      <h2 className="article-title">{props.title}</h2>
      <p>{props.body}</p>
      <a className="read-more" href="#here">
        read more about {props.title}
      </a>
    </article>
  )
}
```

TypeScript

Now our `App.tsx` has no warnings or errors, but our `NewsArticle` does.

We need to tell TypeScript the shape of the `props` variable, otherwise it will be defined as any. We will try to avoid "any" wherever we can.

Define a `type` for our `props`

We can declare a type for our `props` argument. We know it will have two object properties. One named `title` and the other named `body`. Both of these are strings.

Our type definition is:

```
type NewsArticleProps = {
  title: string
  body: string
}
```

Using our type definition

```
export function NewsArticle(props: NewsArticleProps) {
```

Inline types

Instead of declaring a separate type and thus needing to invent a name for the type, we can declare the type at the same time we declare the argument.

```
export function NewsArticle(props: { title: string; body: string }) {
```

Reusable type

The inline style is useful if the type isn't going to be reused. If we reuse the type we should choose the explicit type definition. We can prepend type with `export` so it becomes reusable.

```
export type NewsArticleProps = {
  title: string
  body: string
}

export function NewsArticle(props: NewsArticleProps) {
```

With this style we can reuse the `NewsArticleProps` type elsewhere in our code.

Reusable and Customizable

Now when each of these components is rendered on the page, the unique values for `this.props` are available and we now have a component that is:

- **reusable**
- **customizable**

Driving our application from data

This is great, and we have an application that can render any number of articles we want. However, we still must manually code these in our main application.

It would be nice to render this from a data file, or perhaps a remote API.

Importing JSON data

Let's start by making a JSON file named `articles.json` in the directory along with `App.tsx`

This JSON file will describe an array of articles we want to render.

We will also give each article an `id` as if it came from an API since that is likely the format when this data eventually comes from an API.

```
[
  {
    "id": 42,
    "title": "SDG Invents New Serialization Format!",
    "body": "SDG Brings the 'A' to JSON and invents `JASON` a next generation serialization format."
  },
  {
    "id": 99,
    "title": "SDG To Teach Haskell In Place of TypeScript",
    "body": "Taking type systems and functional programming to the next level with Haskell!"
  },
  {
    "id": 100,
    "title": "SDG Standardizes on React",
    "body": "React is the best library for learning front end Web"
  },
  {
    "id": 101,
    "title": "This data comes from JSON!",
    "body": "React works with data using tools we already know!"
  }
]
```

IMPORTing JSON data

We will be using this in our `App.tsx` so let's import it!

```
import articles from './articles.json'
```

Transforming data into components

Since we want one `<NewsArticle/>` that is related to each element of the `articles` array, we will bring out our friend `map`

```
const newsArticlesFromData = articles.map(article => (
  <NewsArticle title={article.title} body={article.body} />
))
```

NOTE: This is a very powerful line of code!

TypeScript!

Note that we get a warning/error.

```
Missing "key" prop for element in iterator. eslint(react/jsx-key)
```

Any time we dynamically generate JSX in an iterator (e.g. `map`) we need to give it a *unique* identifier so React can efficiently track/change the JSX when updating.

See this React documentation article for more details

What value to use for the `key` prop?

`id` is a great choice

We said it will be unique and unchanging for any specific article. This is perfectly what React wants!

```
const newsArticlesFromData = articles.map(article => (
  <NewsArticle key={article.id} title={article.title} body={article.body} />
))
```

Another good choice would be the `title` if we feel like that is unique enough.

What to do if there is no good choice?

`map` with `index`

If you have no other good choice, add an `index` argument to `map` and use that as the key

```
const newsArticlesFromData = articles.map((article, index) => (
  <NewsArticle key={index} title={article.title} body={article.body} />
))
```

The map essentially creates this:

```
[
  <NewsArticle title="SDG Announces ..." body="SDG announces ..." />,
  <NewsArticle title="Student Graduation ..." body="Our next cohort ..." />,
  <NewsArticle title="SDG Standardizes ..." body="React is the ..." />,
  <NewsArticle title="This data ..." body="React works ..." />,
]
```

Using the mapped data

Since we now have an array of the `<NewsArticle/>` we can simply place them where we want them in place of the hardcoded data.

```
import React from 'react'
import { NewsArticle } from './components/NewsArticle'
import articles from './articles'

function App() {
  const newsArticlesFromData = articles.map(article => (
    <NewsArticle title={article.title} body={article.body} />
  ))

  return (
    <div className="all-main-content">
      <main>{newsArticlesFromData}</main>
    </div>
  )
}

export default App
```

Conclusion

- We now have an application that allows us to add more articles to this listing.
- If we add new data to our JSON file and reload the application it *reacts* to the new data by rendering more information.