Introduction to React.js Using Next.js

Learning Outcomes

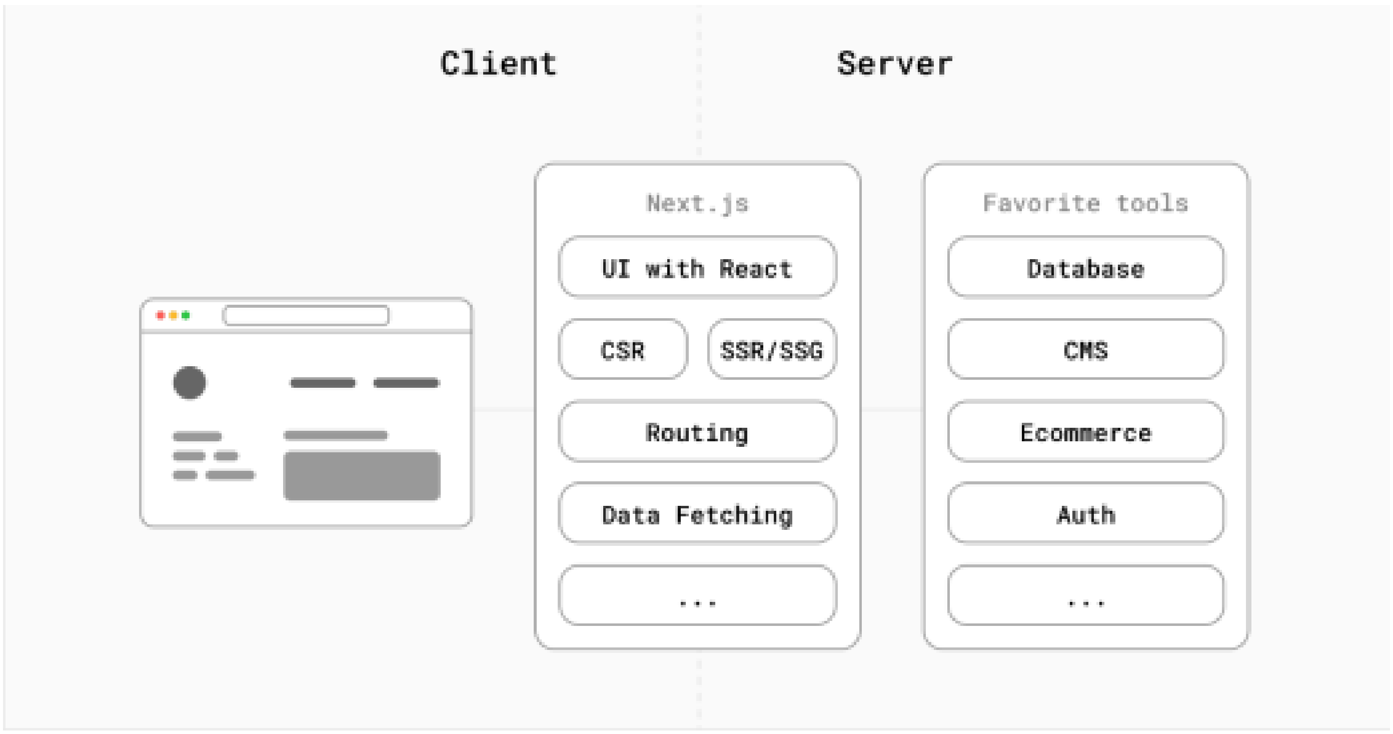
The building blocks that we will be focusing on in the next couple of days include:

- **User Interface** - how users will consume and interact with your application.
- **Routing** - how users navigate between different parts of your application.
- **Rendering** - when and where you render static or dynamic content.
- **Performance** - how to optimise your application for end-users.
- **Developer Experience** - your team's experience building and maintaining your application.

Next.js

What is Next.js?

Next.js is a React **framework** that gives you building blocks to create web applications. This means that Next.js handles the tooling and configurations needed for React, and provides additional structure, features, and optimizations for a website or web application.



For the purposes of this React module, we will be learning React by using the Next.js framework. This allows us to forget about the boilerplate of a web project and rather focus on the core concepts of React.

Create a Next.js Project

Initial Setup

Next.js is powered by a very powerful runtime engine called Node.js. It is a requirement to be able to use

Next.js. Accordingly, let's make sure that your development environment is properly set up.

> If you don't have **Node.js** installed, please follow [this link][https://nodejs.org/en/] and install it from there. You'll need Node.js version **10.13** or later.
> If you don't have Visual Studio Code installed, please follow [this link][https://code.visualstudio.com/download] and install it from there.

> If you are on Windows, I recommend downloading Git for windows and use Git Bash that comes with it.

Create a Next.js App

To create a Next.js App, open your terminal (on Mac) or Git Bash (on Windows), run the `cd` command into the directory you'd like to create the app in, which would typically be your desktop, and run the following command:

```
npx create-next-app [PROJECT_NAME] --use-npm --example
"https://github.com/vercel/next-learn/tree/master/basics/learn-starter"
```

Where `[PROJECT_NAME]` can be replaced by a name of your choice, **without the brackets**. For example, if you want to call your project `tutorial`, then you would run the following command:

```
npx create-next-app tutorial --use-npm --example "https://github.com/vercel/next
learn/tree/master/basics/learn-starter"
```

> Under the hood, this uses the tool called `create-next-app`, which bootstraps a Next.js app for you. It uses this template through the --example flag.

Start the Development Environment

You should now have a directory called `tutorial`, or whichever name you chose. For the sake of the explanation, I will assume the `tutorial` name. Let's `cd` into the project:

```
cd tutorial
```

Then, run the following command:

```
code .
```

This will open your project in Visual Studio Code.

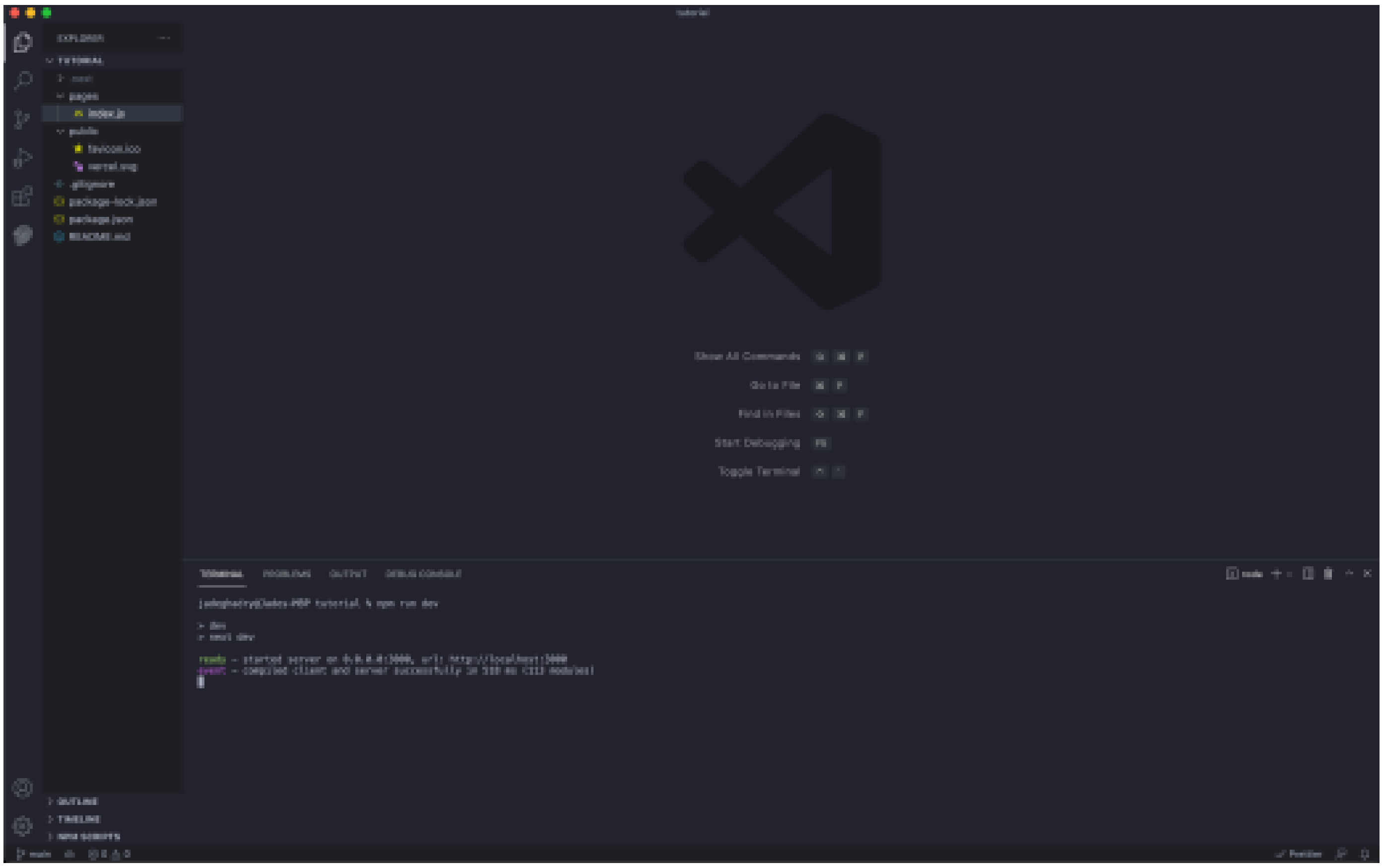
If you are using a Windows machine, you shouldn't encounter any problems with the above command. On a Mac, however, you have to install the command globally first. To do so, open a random Visual Studio Code window, then open the command palette using the **CMD+P** shortcut, and search for **Shell Command: Install 'code' command in PATH**. After running this from the command palette, you should be able to use the **code .** command.

The beauty about Visual Studio Code is that it is **very comprehensive and modular**. This means that you can customize it to whatever extent you like, but also provides you with a lot of tools to facilitate your development environment. One of these tools is an **integrated terminal** that you can access using the **CTRL+J** on Windows or **CMD+J** on Mac. Once open, run the following command:

```
npm run dev
```

This will start your development server on port **3000**, and you should now be able to see the starter Next.js project when you open [http://localhost:3000/][http://localhost:3000/] from your preferred browser.

Your Visual Studio Code window should look approximately like this

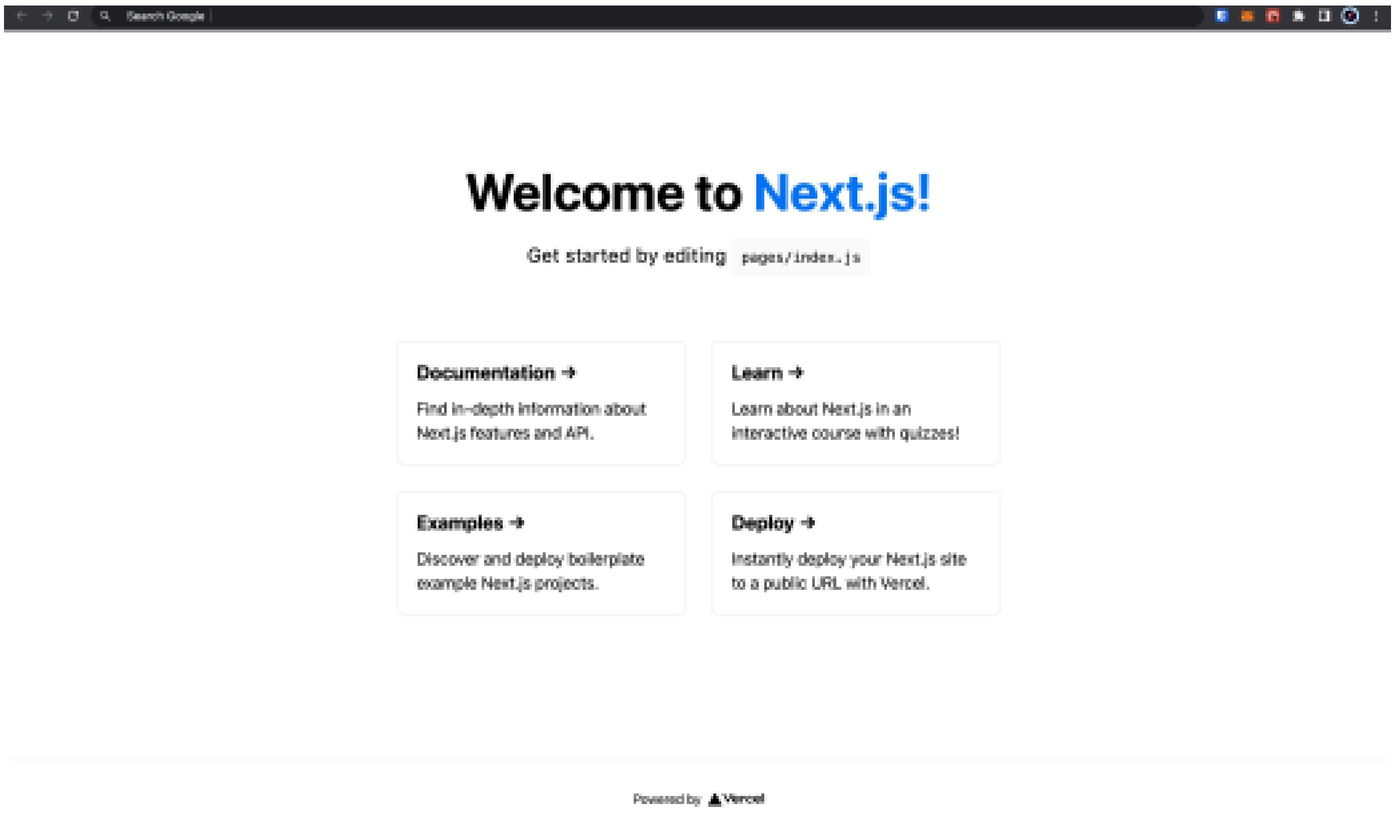


```
[illegible] tutorial % npm run dev

> dev
> next dev

ready - started server on 0.0.0.0:3000, url: http://localhost:3000
event - compiled client and server successfully in [illegible] ms ([illegible] modules)
```

And your browser window should look approximately like this


Welcome to Next.js!

Get started by editing `pages/index.js`

Documentation →
Find in-depth information about Next.js features and API.

Learn →
Learn about Next.js in an interactive course with quizzes!

Examples →
Discover and deploy boilerplate example Next.js projects.

Deploy →
Instantly deploy your Next.js site to a public URL with Vercel.

Powered by Vercel

Core Concepts in Next.js

Starter Project Structure

Before we start developing components and dealing with React, it is important to understand the structure of our starter Next.js project.

> The *pages* directory is where you nest your actual web pages. Typically, a *page* in Next.js would be the equivalent of an HTML file. Every file of code you create under the *pages* directory will be translated into a URL that you can access from your browser. For example, if you create a file called `products.js` inside of the *pages* folder, then you will be able to access that page at http://localhost:3000/products automatically. You can [read more][https://nextjs.org/docs/basic-features/pages] about pages in details and understand the power behind it, however unnecessary for the scope of our bootcamp. The *public* directory is where you nest all of your assets such as `.jpeg`, `.png`, custom fonts, or other resources. The `.gitignore` file is used by Git to stop tracking the changes in some of the redundant or large files of our project. If you open this file, you should see that it has already been preconfigured for you. The `README.md` file used to document your website or project for external users. You can disregard this file for now.

Please note that **the above** bulletpoints are pertinent to Next.js, and **not** to React. If you create a pure React project without the Next.js framework, then you won't have access to the above

The `package.json` and `package-lock.json` are used by NPM to create, read, link, and update your project's dependencies with other libraries. You don't need to worry too much about them, but

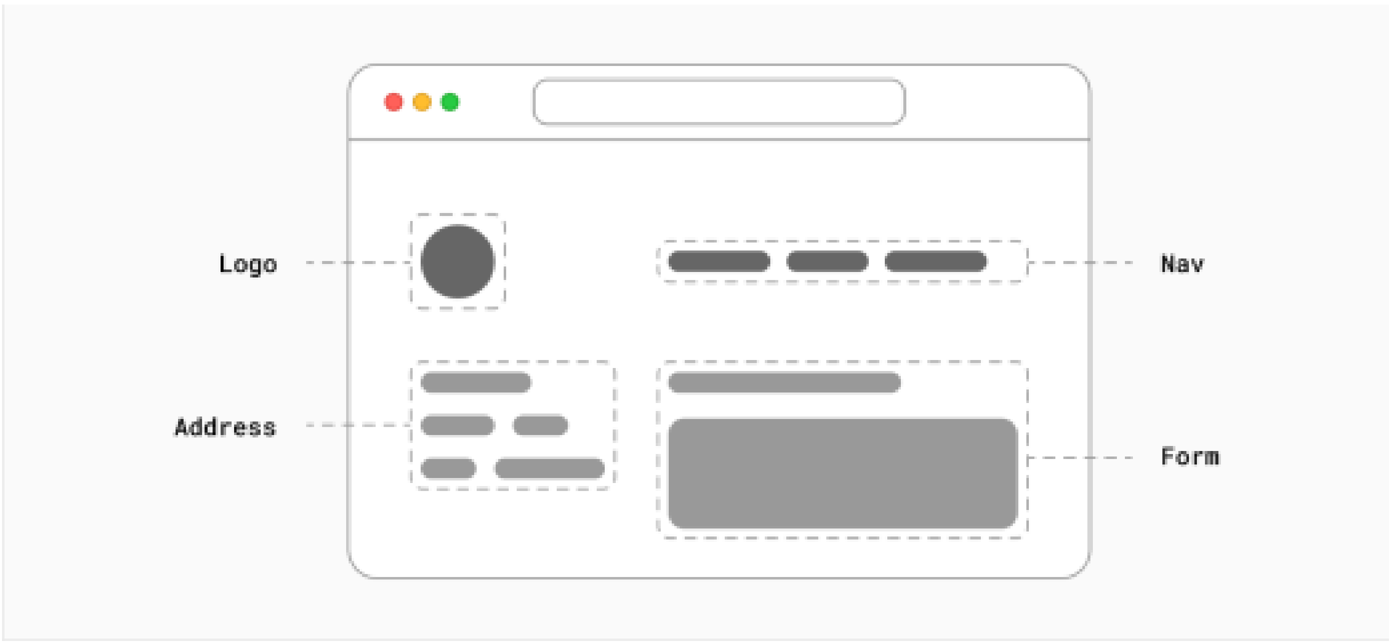
you absolutely cannot remove those files.

React.js

What is React?

[React][https://reactjs.org] is a JavaScript **library** for building interactive and compartmentalized **user interfaces** (UI).

User interfaces are usually built with the elements that can be **seen** by users, and those who can be **interacted with** on-screen.



Libraries such as React typically provide helpful functions to build user interfaces, but leaves it up to the developer where to use those functions in their application.

Part of React's success is that it is relatively unopinionated [1]. This means that it does not enforce a way of thinking on the developer, which resulted in a flourishing ecosystem of third-party tools and solutions.

It also means that biulding a complete React application from the group up requires some effort. Developers need to put in a lot of time trying to configure tools and reinvent solutions for common website and web application requirements.

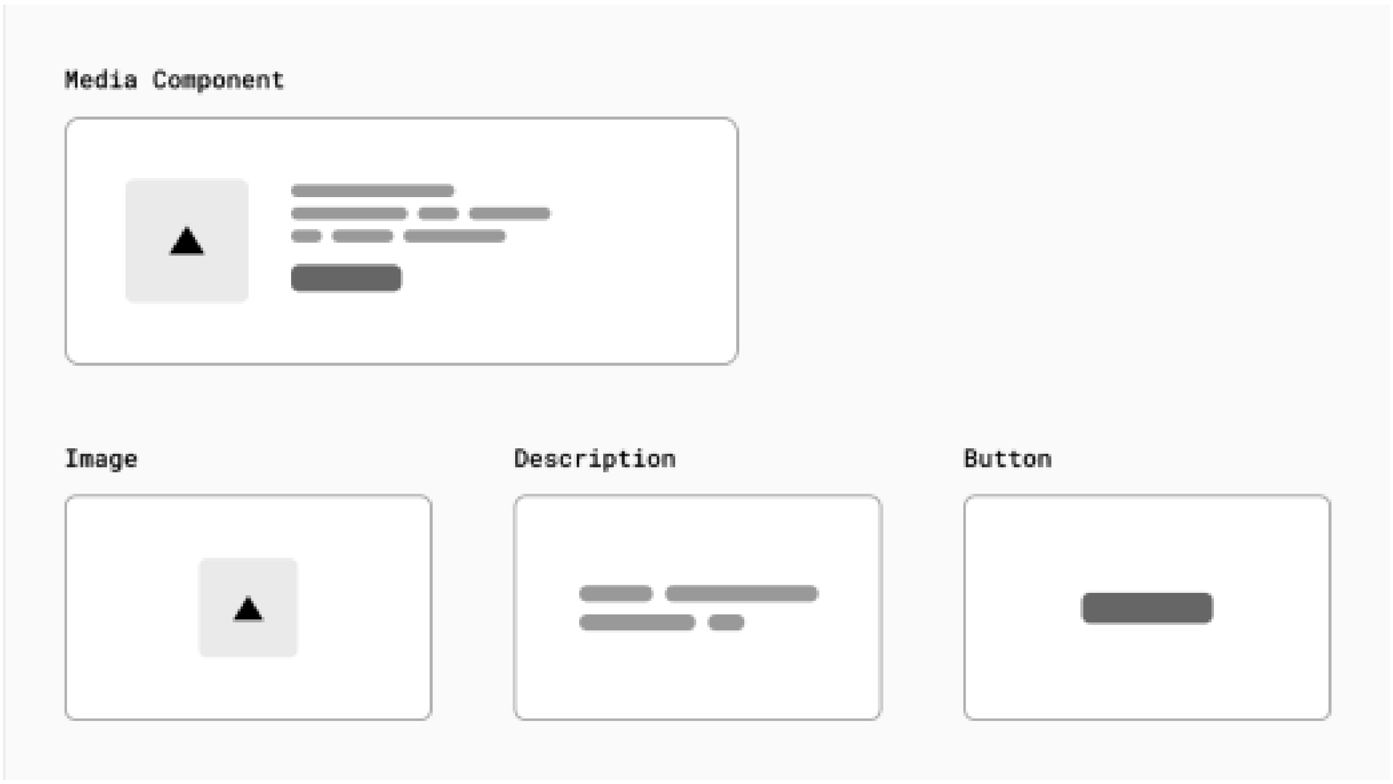
Core Concepts in React

There are two core concepts in React that you'll need to be familiar with to start building React-based websites and web applications. These concepts include:

- Components
- Props

Building UI with Components

User interfaces can be broken down into smaller building blocks, also referred to as **components**. They allow you to build **self-contained**, **reusable** snippets of code. If you think of components as **LEGO bricks**, you can take these individual bricks and combine them together to form larger structures. If you need to update a piece of the UI, you can update the specific component or brick.



This philosophy of modularity and compartmentalisation allows your code to be more maintainable as it grows, because you can easily add, update, and delete components without touching the rest of your application.

The cool thing about React components is that they are entirely built using a JavaScript extension called JSX.

Introducing JSX

Consider this variable declaration using the typical JavaScript notation that you know so

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

This weird syntax is neither considered a string, since it doesn't have any quotation marks, nor plain HTML, since we are associating it to a constant that we called `element` .

This is called JSX, a syntax extension to vanilla JavaScript that allows you to produce React *elements*, also commonly referred to as *components*.

Why JSX?

React embraces the fact that no matter how hard we try to separate things, rendering logic is inherently coupled with other UI logic, such as event handling, hovering/unhovering states, and how the DOM is consequently manipulated.

Instead of artificially separating *technologies* by putting markup and logic in separate files, React separates *concerns* with loosely coupled components that contain both.

Embedding Expressions in JSX

In the below example, we declare a variable called `name` and then use it inside JSX by wrapping it in curly braces:

```
const name = "Josh Perez";
const element = <h1>Hello, {name}</h1>
```

You can put *any* valid JavaScript expression inside the curly braces in JSX. In the example below provided by React, the result of calling a JavaScript function `formatName(user)` is embedded into an `<h1>` element.

```
function formatName(user) {
  return user.firstName + ' ' +
user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
   Hello, {formatName(user)}!
  </h1>
);
```

JSX Elements are Expressions

After being compiled, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects. This means that you can use JSX inside of `if` statements, `for` loops, assign it to variables, accept it as arguments, and return it from functions.

```
function getGreeting(user) {
  if (user) {
      return <h1>Hello,
{formatName(user)}!</h1>;
    }
    return <h1>Hello, Stranger.</h1>;
  }
```

Notice how the **else** statement was omitted from the above code. This is not a mistake. When a function executes a **return** statement, it also exists from itself, which means that no other line of code in that function will ever get executed.

Specifying Attributes with JSX

You may use quotation marks to specify string literals as attributes:

```
const element = <a href="https://www.augment.lu">Augment</a>;
```

And you may also use **curly braces** to **embed** a JavaScript expression in an

attribute:
```
const element = <img src={user.avatarUrl}></img>
```

If you put quotation marks around the curly braces when embedding a JavaScript expression, then it will be considered as a string. For example:

```
const element = <img src="{user.avatarUrl}"></img>
```

Will treat `{user.avatarUrl}` as a **string**.

Since JSX is closer to JavaScript than to HTML, React DOM uses the *camelCase* property naming convention instead of HTML attribute names. For example, class becomes className in JSX, and tabindex becomes tabIndex.

Worlds Collide

Now's that we've covered the basics, it's time to merge our knowledge in both Next.js and React to kickstart our development.

Creating a React Component

In React, one can create **class** components or **function** components. For the purposes of this bootcamp, we will be focusing on the latter.

The simplest way to define a component is to write a JavaScript function:

```
function Welcome(props) {
 return <h1>Hello, {props.name}</h1>;
}
```

This function is a valid React component because it accepts a single `props` (which is short for *properties*) argument and returns a React element. We call such components **function components** because they are literally JavaScript functions.

You can also rewrite the above using an arrow function:

```
const Welcome = (props) => {
 return <h1>Hello, {props.name}</h1>
}
```

> **Always start component names with a capital letter.** React treats components starting with lowercase letters as DOM tags. For example, <div/> represents an HTML div tag, but <Welcome/> represents a component and requires Welcome to be available in scope. This means that if the component was created in another file, then it has to be imported in the current one you are trying to use it.

When working with Next.js, it is always important to keep in mind your project structure. Accordingly, it is best to create a `components` folder at the top-level directory, in parallel of the Next.js `pages` directory.

> If you nest your components directory under the pages directory of your Next.js project, then you would be creating a new route in your website that corresponds to /components/[FILE_NAME] where [FILE_NAME] represents the name of the files you nest under components. This is fundamentally **wrong**, as the intended purpose is to create a standalone **component**, and not a website page.

Props

Let's take a deeper dive at how **props** are passed around.

Assume you are calling the above `Welcome` function, or in other words rendering the `Welcome` component, then your call would typically look like an HTML tag:

```
<Welcome />
```

If you want to pass a `firsName` and `lastName` arguments to your `Welcome` component's call, then you do so using the HTML attributes syntax that you already know:

```
<Welcome firstName="Chicken" lastName="Wings" />
```

This is literally the equivalent of calling the function in the following manner:

```
const props = {
  firstName: "Chicken",
  lastName: "Wings",
}
Welcome(props);
```

Except that React preserves the HTML syntax to keep things consistent with the rest of our HTML structure.

React Components are Pure Functions

Components must **never** modify its own props.

Consider the below `sum` function:

```
const sum = (a, b) => {
  return a + b;
}
```

Such functions are called *pure* because they do not attempt to change their inputs, and always return the same result for the same inputs.

In contrast, the below function is impure because it changes its own input:

```
const withdraw = (account, amount) => {
  account.total = account.total - amount;
}
```

React is pretty flexible in its philosophy, but has a single strict rule:

All React components must act like pure functions with respect to their props.

This means that when a component receives a set of props, the component itself should **never** change their values. It can use those props to calculate new values, but never update the prop itself.

###

1. Not opinionated; without strong opinions about something, typically in a development paradigm. ↩