Next.js

> *Next.js is a flexible **React framework** that gives you building blocks to create fast **web applications**. Next.js handles the tooling and configuration needed for React and provides additional structure, features, and optimizations for the application.*

Use React to build UI, then adopt Next.js to solve common application requirements such as **routing, data fetching, integration**...

Building Blocks of a Web Application

- **User Interface** - how users will consume and interact with your application
- **Routing** - how users navigate between different parts of the application
- **Data Fetching** - where your data lives and how to get it
- **Rendering** - when and where you render static or dynamic content
- **Integrations** - 3rd party services (CMS, auth, payments, etc)
- **Infrastructure** - where you deploy, store, and run your application code (Serverless, CDN, Edge, etc)
- **Performance** - how to optimize the application for end-users
- **Scalability** - how your application adapts as your team, data, and traffic grow.
- **Developer Experience** - your team's experience building and maintaining your application.

From JavaScript to React

> *React is a JavaScript **library** for building interactive **user interfaces**.*

> ## *What is the DOM?*
>
> *The DOM is an object representation of the HTML elements. It acts as a bridge between your code and the user interface, and has a tree-like structure with parent and child relationships.*

Inline JavaScript with DOM methods

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Next.js Exercises 01 - Intro</title>
</head>
<body>
    <div id="app">
    </div>
    <script type="text/javascript">
        const app = document.getElementById("app");
        const header1 = document.createElement("h1");
        const header1Content = document.createTextNode(
            "Develop. Preview. Ship. 🚀"
        )
        header1.appendChild(header1Content);
        app.appendChild(header1);
    </script>
</body>
</html>
```

The DOM of the page is different from the source code - or in other words, the original HTML file created. This is because the HTML represents the **initial page content**, whereas the DOM represents the **updated page content** which was changed by the JavaScript code.

Imperative vs Declarative Programming

In imperative programming, we are telling computer what to do step by step like in the above example.

React is a declarative library where developer declares what he/she needs, for example an h1 tag with some text in it and the rest is handled by React, it will figure out the steps of how to update the DOM.

Getting Started with React

React Scripts for core React library and React DOM

```
 <script src="https://unpkg.com/react@17/umd/react.development.js">
</script>
    <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js"></script>
```

```
<!DOCTYPE html>
<html lang="en">

<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>01 - JS DOM Methods</title>
</head>

<body>
    <div id="app">
    </div>

    <!-- React scripts taken from unpkg.com -->
    <script
src="https://unpkg.com/react@17/umd/react.development.js"></script>
    <script src="https://unpkg.com/react-dom@17/umd/react-
dom.development.js"></script>

    <script type="text/jsx">
        const app = document.getElementById("app");
        // ReactDOM.render() method instead of plain JavaScript DOM
methods
        ReactDOM.render(<h1>Develop. Preview. Ship. 🚀</h1>, app)
// <h1></h1> --> JSX code
    </script>
</body>

</html>
```

> ***JSX***
>
> *JSX is a syntax extension for JS that allows you to describe your UI in a familiar HTML-like syntax. Three JSX rules:*
>
> 1. *Return a single root element*
> 2. *Close all the tags*
> 3. *camelCase most of the things*

> *!!!*
>
> *Browsers don't understand JSX out of the box, so you'll need a **JavaScript compiler** such as **Babel** to **transform JSX code into regular JavaScript**.*

Adding Babel to project

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js">
</script>
```

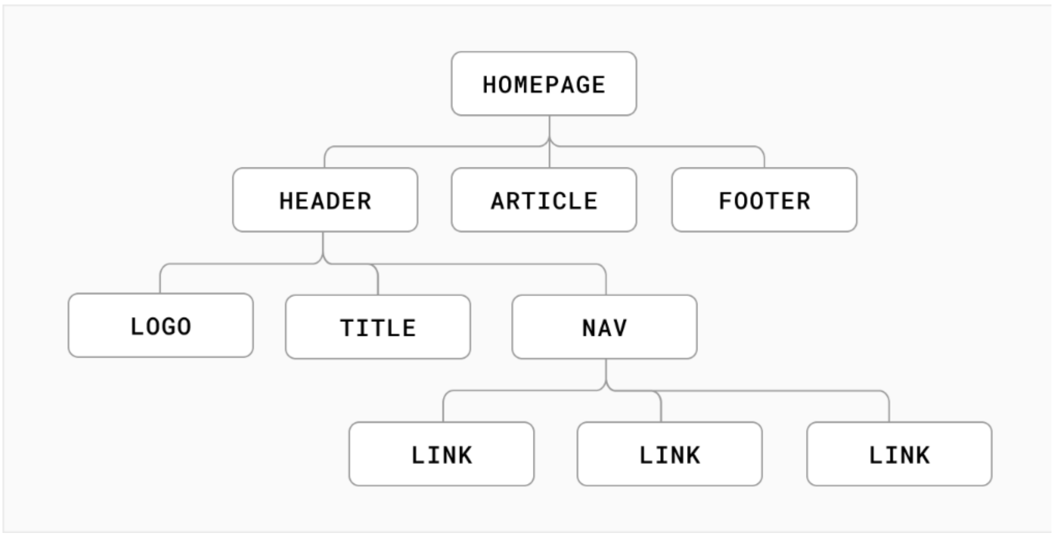
Inform Babel what code to transform by changing the script tyle to "text/jsx". ==> **<script type="text/jsx">**

```
    <script type="text/jsx">
        const app = document.getElementById("app");
        // ReactDOM.render() method instead of plain JavaScript DOM
methods
        ReactDOM.render(<h1>Develop. Preview. Ship. 🚀</h1>, app)
// <h1></h1> --> JSX code
    </script>
```

React Core Concepts

Components

Component Tree



```
<script type="text/jsx">
        const app = document.getElementById("app");

        function HomePage(){
            return (
                <>
                    <Header />
                    <hr />
                    <Main />
                    <hr />
                    <Footer />
                </>
            )
        }

        function Header(){
            return (
                <header>
                    <div className="logo">
                        <h1>Next.js</h1>
                    </div>
                    <h2>Develop. Preview. Ship. 🚀</h2>
                    <nav>
                        <ul>
                            <li>Home</li>
                            <li>About</li>
                            <li>Contact</li>
                        </ul>
                    </nav>
                </header>)
        }
        function Main(){
            return (
                <main>
                    <article>
                        <p>
                            Lorem ipsum dolor sit amet consectetur
adipisicing elit. Quod minus quaerat culpa non voluptatem animi,
quos cum repellendus fuga, error accusantium, tempore iste
asperiores quidem commodi rem doloremque vero officiis?Architecto
cum est minus nobis perspiciatis fugiat odit id accusantium
voluptatibus quod soluta consequuntur obcaecati asperiores velit ex
dicta natus, suscipit vel delectus itaque dignissimos sit tempora!
Error, quisquam sunt? Tenetur iure quae dolorum rerum, voluptatum
harum! Mollitia aperiam culpa eum quaerat earum fugit, facilis amet
similique id atque eius blanditiis voluptate aut veritatis neque
adipisci praesentium explicabo ipsum nihil?
                        </p>
                    </article>
                </main>
                )
        }

        function Footer(){
            return <footer><small>I am footer.</small></footer>
        }
        ReactDOM.render(<HomePage/>, app)
    </script>
```

Properties

Object property with dot notation

```
function Header(props) {
  return <h1>{props.title}</h1>;
}
```

Template literal

```
function Header({ title }) {
  return <h1>{`Cool ${title}`}</h1>;
}
```

The returned value of a function

```
function createTitle(title) {
  if (title) {
    return title;
  } else {
    return 'Default title';
  }
}

function Header({ title }) {
  return <h1>{createTitle(title)}</h1>;
}
```

Ternary operators

```
function Header({ title }) {
  return <h1>{title ? title : 'Default Title'}</h1>;
}
```

Iterating Through Lists

> *React is unopinionated when it comes to data fetching, meaning you can choose whichever solution best suits your needs.*

```
<script type="text/jsx">

        const app = document.getElementById("app");



        function HomePage(){
            const names = ['Ada Lovelace', 'Grace Hopper', 'Margaret
Hamilton'];

            const [sciList, setSciList] = React.useState([]);
            const [clickCount, setClickCount] = React.useState(0);

            function handleClick(e){
                setClickCount(clickCount + 1);
                setSciList([...sciList, <li key={clickCount}>
{names[clickCount]}</li>])
            }

            return (
                <>
                    <Header title="Woman Computer Scientists 💪🏻" />
                        <ul>
                            {sciList}
                        </ul>
                        <button onClick={handleClick}>Get
scientist</button>
                </>
            )
        }

        function Header({title}){

            return (
                <header>
                    <div className="logo">
                        <h1>{title ? title : 'Default Title'}</h1>
                    </div>
                </header>)
        }

        ReactDOM.render(<HomePage/>, app)
    </script>
```

IMPORTANT

Unlike props which are passed to components as the first function parameter, the state is initiated and stored within a component. ***You can pass the state information to children components as props:***

```
import React, { useState } from "react";

export function Parent() {
  const [name, setName] = useState("Dolan The Duck");

  return (
    <div>
      <h1>Dolan Family</h1>
      <Child name={name} />
    </div>
  );
}

function Child(props) {
```

> ```
>   return <h2>I am Luffy! {props.name}'s son!</h2>;
> }
> ```
>
> *But the logic for updating the state should be kept within the component where state was initially created.*
>
> ***Props** is **read-only information** that's passed to components. State is information that can change over time, usually triggered by user interaction.*

From React to Next.js

Getting Started with Next.js

Create package.json file with an empty object:

```
// package.json
{
}
```

In the terminal, run:

```
npm install react react-dom next
```

After running, package.json should look like:

```
// package.json
{
  "dependencies": {
    "next": "^12.1.0",
    "react": "^17.0.2",
    "react-dom": "^17.0.2"
  }
}
```

Jumping back to the `index.html` file, you can delete the following code:

1. The `react` and `react-dom` scripts since you've installed them with NPM.
2. The **`<html>`** and **`<body>`** tags because Next.js will create these for you.
3. The code that interacts with `app` element and `ReactDom.render()` method.
4. **The `Babel` script** because Next.js has a compiler that transforms JSX into valid JavaScript browsers can understand.

5. The `<script type="text/jsx">` tag.
6. The `React.` part of the `React.useState(0)` function

After deleting the lines above, add `import { useState } from "react"` to the top of your file.

The only code left in the HTML file is JSX, so you can change the file type from `.html` to `.js` or `.jsx`.

Now, there are three more things you need to do to fully transition to a Next.js app:

1. Move the `index.js` file to a new folder called `pages` (more on this later).
2. **Add default export** to your **main React component** to help Next.js distinguish which component to render as the main component of this page.

```
// ...
   export default function HomePage() {
   //  ...
```

3. Add a script to your `package.json` file to run the Next.js development server while you develop.

```
   // package.json
   {
    "scripts": {
        "dev": "next dev"
    },
    "dependencies": {
        "next": "^11.1.0",
        "react": "^17.0.2",
        "react-dom": "^17.0.2"
    }
   }
```

RUN APP:

```
npm run dev
```

Default port: http://localhost:3000/

How Next.js Works

Since each environment has different considerations and goals, there is a lot that needs to be done to move an application from development to production. For instance, the application code needs to be compiled, bundled, minified, and code split.

Compiling

In Next.js, compilation happens during the development stage as you edit your code, and as part of the build step to prepare your application for production.

Minifying

Minification is the process of removing unnecessary code formatting and comments (which added by the developers to optimize human readability, such as comments, spaces, indents, and multiple lines.) without changing the code's functionality. **The goal is to improve the application's performance by decreasing file sizes**.

In Next.js, JavaScript and CSS files are automatically minified for production.

Bundling

Developers break up their application into modules, components, and functions that can be used to build larger pieces of their application.

Bundling is the process of resolving the web of dependencies and merging (or 'packaging') the files (or modules) into optimized bundles for the browser, with **the goal** of **reducing the number of requests for files when a user visits a web page.**

> *Compiling is transforming code into something parsable by browser. Bundling is resolving your applications dependency graph and reducing the number of files.*

Code Splitting

Developers usually split their applications into multiple pages that can be accessed from different URLs. Each of these pages becomes a unique **entry point** into the application.

Code-splitting is the process of splitting the application's bundle into smaller chunks required by each entry point. The goal is to **improve the application's initial load time** by only loading the code required to run that page.

Next.js has built-in support for code splitting. **Each file inside your `pages/` directory will be automatically code split into its own JavaScript bundle during the build step.**

Further:

- Any code shared between pages is also split into another bundle to avoid re-downloading the same code on further navigation.
- After the initial page load, Next.js can start pre-loading the code of other pages users are likely to navigate to.
- Dynamic imports are another way to manually split what code is initially loaded.

Rendering

There is an unavoidable unit of work to convert the code you write in React into the HTML representation of your UI. This process is called **rendering**.

Rendering can take place on the server or on the client. It can happen either ahead of time at build time, or on every request at runtime.

Client-Side Rendering vs. Pre-Rendering

In a standard React application, the browser receives an empty HTML shell from the server along with the JavaScript instructions to construct the UI. This is called **client-side rendering** because the initial rendering work happens on the user's device.

In contrast, Next.js pre-renders every page by default. Pre-rendering means the HTML is generated in advance, on a server, instead of having it all done by JavaScript on the user's device.

> *You can opt to use client-side rendering for specific components in your Next.js application by choosing to fetch data with React's `useEffect()` or a data fetching hook such as useSWR.*

Types of Pre-Rendering

Server-side Rendering

With server-side rendering, the HTML of the page is generated on a server for **each** request. The generated HTML, JSON data, and JavaScript instructions to make the page interactive are then sent to the client.

On the client, the HTML is used to show a fast non-interactive page, while React uses the JSON data and JavaScript instructions to make components interactive (for example, attaching event handlers to a button). This process is called **hydration**.

In Next.js, you can opt to server-side render pages by using getServerSideProps.

> ***Note:*** *React 18 and Next 12 introduce an alpha version of* ***React server components****. Server components are completely rendered on the server and do not require client-side JavaScript to render. In addition, server components allow developers to keep some logic on the server and only send the result of that logic to the client. This reduces the bundle size sent to the client and improves client-side rendering performance. Learn more about React server components here.*

Static Site Generation

With Static Site Generation, the HTML is generated on the server, but unlike server-side rendering, there is no server at runtime. Instead, content is generated once, at build time, when the application is deployed, and the HTML is stored in a CDN and re-used for each request.

In Next.js, you can opt to statically generate pages by using getStaticProps.

> ***Note:*** *You can use Incremental Static Regeneration to create or update static pages after you've built your site. This means you do not have to rebuild your entire site if your data changes.*

To learn more about which rendering method is right for your specific use case, see the data fetching docs.

The Network

In the case of a Next.js application, your application code can be distributed to **origin servers**, **Content Delivery Networks (CDNs)**, and **the Edge**.

Origin Servers

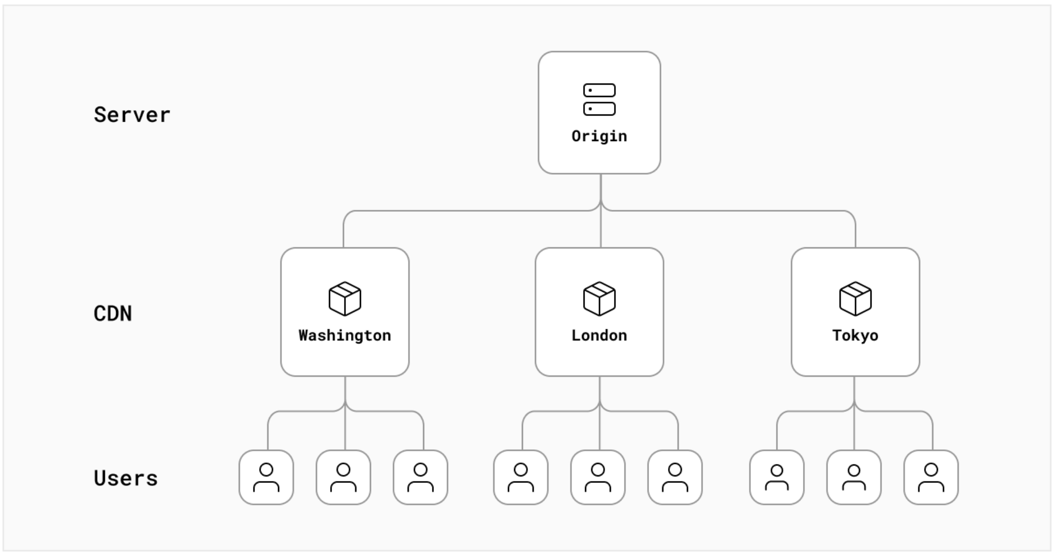
The server refers to the main computer that stores and runs the original version of your application code.

We use the term **origin** to distinguish this server from the other places application code can be distributed to, such as **CDN servers** and **Edge servers**.

When an origin server receives a request, it does some computation before sending a response. The result of this computation work can be moved to a CDN (Content Delivery Network).

Content Delivery Network (CDN)

CDNs store static content (such as HTML and image files) in multiple locations around the world and are placed between the client and the origin server.



In Next.js, since pre-rendering can be done ahead of time, **CDNs are well suited to store the static result of the work** - making **content delivery faster**.

The Edge

Similar to CDNs, Edge servers are distributed to multiple locations around the world. But unlike CDNs, which store static content, some Edge servers can run small snippets of code.

This means both **caching** and **code execution** can be done at the Edge closer to the user.

By moving some work that was traditionally done client-side or server-side to the Edge, you can make your application more performant because it reduces the amount of code sent to the client, and part of the user's request does not have to go all the way back to the origin server - thus reducing latency. See Edge examples with Next.js here.

In Next.js, you can run code at the Edge with Middleware, and soon with React Server Components.

App A - Links

Related Links
Introduction to the DOM
How to view the DOM in Google Chrome
HTML vs the DOM
How declarative UI compares to imperative
Writing markup with JSX
https://unpkg.com/
three JSX rules
Babel
UI trees
react-dom/server
Functions
Arrow Functions
Objects
Arrays and array methods
Destructuring
Template literals
Ternary Operators
ES Modules and Import / Export Syntax
Your first component
Importing and exporting components
Passing props to a component
Rendering lists
Conditional rendering
hooks
Adding Interactivity
Managing State

Related Links
State: A component’s memory
Meet your first hook
Responding to Events
How React handles renders
how to use refs
How to manage state
How to use context for deeply nested data
How to use React API hooks
Fast Refresh
Going to production
React server components here
Incremental Static Regeneration
data fetching docs
Edge Network
Next.js Starter Templates