Implementation Notes

This project is meant to be an introduction to Next.JS for building a modern website - the end goal is to design a professional looking blog/cv site which can be hosted using github pages.

Introduction

We start by looking at the Next.js Foundations course to guide through the prerequisite knowledge for Next.js. We run through a simple project - starting a Javascript application and then migrating it to React and Next.js.

Next.js is a flexible React framework that provides building blocks to create fast web applications.

Building Blocks of Web Application

There are several things to consider when building modern applications including:

- User Interface - how users will consume and interact with the application
- Routing - how users navigate between different parts of the application
- Data Fetching - where the data lives and how to get it
- Rendering - when and where you render static or dynamic content
- Integrations - what third-party services will be used (CMS, auth, payments) and how you connect to them
- Infrastructure - where you deploy, store and run the application code (serverless, cdn, edge)
- Performance - how to optimize your 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

For each part of the application, we need to decide whether to build a solution ourselves or use other tools such as libraries and frameworks

What is React?

React is a JavaScript library for building interactive user interfaces. By user interfaces we mean the elements that users see and interact with on-screen.

Library means that React provides helpful functions to build UI but leaves it up to the developer where to use those functions in their application.

Part of the success of react is that it is relatively unopinionated about the other aspects of building applications. This has resulted in a flourishing ecosystem of third-party tools and solutions.

It also means that building a complete React application from the ground up requires so effort - developers need to spend time configuring tools and reinventing solutions for common application requirements.

What is Next.js?

Next.js is a react framework that gives you building blocks to create we applications - by framework we mean that Next.js handles the tooling and configuration needed for React, and provides additional structure, features and optimizations for your application.

You can use React to build the UI then incrementally adopt Next.js features to solve common application requires such as routing, data fetching, integrations - all while improving the developer and end-user experience.

Whether you are a individual developer or part of a larger team, you can leverage React and Next.js to build fully interactive, highly dynamic and performant web applications.

Next we discuss how to get started with React and Next.js.

From JS to React

Rendering User Interfaces

To see how React works we first will go over how browsers interpret your code to create interactive user interfaces (UI).

When a user visits a web page, the server returns an HTML file to the browser - the browser then reads the HTML and constructs the Document object Model (DOM).

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.

You can use DOM methods and a programming language like JavaScript to listen to user events and manipulate the DOM by selecting, adding, updating and deleting specific elements in the user interface. DOM manipulation allows you to not only target specific elements but also change their style and content.

We can see how Javascript and DOM methods can be used below:

```html
<!-- index.html -->
<html>
    <body>
        <div id="app"></div>
    </body>
</html>
```

The div is given a unique id so that we can target it later.

```
<script type="text/javascript">
    const app = document.getElementById('app');
    const header = document.createElement('h1');
    const headerContent = document.createTextNode(
        'Develop. Preview. Ship.'
    );

    head.appendChild(headerContent);
    app.appendChild(header);
</script>
```

Javascript can be written inside a HTML file by adding a script tag - inside it we can use a DOM method `getElementById()` to select the `<div>` element by its `id`. We can continue to use DOM methods to create a new `<h1>` element.

HTML vs the DOM

If we look at the DOM elements using the browser developer tools, we will notice the DOM includes the `<h1>` element. The DOM of the page is different from the source code - or in other words, the original HTML file you created

This is since the HTML represents the initial page content, whereas the DOM represents the updated page content which was changed by the Javascript code we wrote.

Updating the Dom with plain Javascript is very powerful but verbose. A lot of code was written just to write an `<h1>` element with some text. As the team or the size of the application grows it can become increasinly challenging to build applications this way.

This way, developers spend a lot of time writing instructions to tell the computer how it should do things. But wouldn't it be nice to describe what you want to show and let the computer figure out how to update the DOM!

Imperative vs Declarative Programming

The code above is a good example of imperative programming. You are writing the steps for how the user interface should be updated. But when it comes to building user interfaces, a declarative approach is often preferred because it can speed up the development process. Instead of having to write DOM methods, it would be helpful if developers were able to declare what they want to show (in this case, an 'h1' tag with some text).

In other words imperative programming is like giving a chef step-by-step instructions on how to make a pizza. Declarative programming is like ordering a pizza without being concerned about the steps it takes to make the pizza.

A popular declarative library that helps developers build user interfaces is React.

React: A declarative UI library

As a developer you can tell React what you want to happen to the user interface, and React will figure out the steps of how to update the DOM on your behalf. Next, we will explore how we can get started with React.

Getting Started with React

To use react in a project we can load two React scripts from an external website called unpkg.com:

- react is the core react library
- react-dom provides DOM-specific methods that enable you to use React with the DOM

```
<!-- index.html -->

<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>
```

Instead of directly manipulating the DOM with plain Javascript you can use `ReactDOM.render()` from `react-dom` to tell React to render our `<h1>` title inside our app element.

```
<script type="text/javascript">
    const app = document.getElementById('app');
    ReactDOM.render(<h1>Develop. Preview. Ship. </h1>, app);
</script>
```

When this code is run in the browser we end up with a syntax error because `<h1>...</h1>` is not valid Javascript - it is JSX

What is JSX?

JSX is a syntax extension for Javascript that allows you to describe your UI in a familiar HTML-like syntax. The nice thing about JSX is that a part from the three JSX rules there is no need to learn any new symbols or syntax outside of HTML and JavaScript:

- Return a single root element: to return multiple elements from a component, wrap them with a single parent tag - for example you can use a

- Close all the tags: JSX requires tags to be explicitly closed: self-closing tags like must become

- camelCase most of the things: JSX turns into JavaScript and attributes written in JSX become keys of JavaScript objects. In components you will often want to read these attributes into variables. JavaScript has limitations on variable names - their names cannot contain dashes or be reserved words like class. This is why many HTML and SVG attributes are written in camelCase, so instead of stroke-width we use strikeWidth

Note that browsers do not understand JSX out of the box, so we need a JavaScript compiler such as Babel, to transform JSX code into regular JavaScript

Adding Babel to the project

To add Babel to the project, copy the following script in `index.html` file:

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

In addition we need to inform Babel what code to transform by changing the script type to `type=text/jsx`.

We can see that comparing the declarative React code with the imperative Javascript code, React allows us to cut down on a lot of repetitive code.

Essential JavaScript for React

Would be helpful to brush up on core JavaScript principles before diving into React

React Core Concepts

There are three core concepts of React that you need to be familiar with to start building React applications. These are:

- Components
- Props
- State

Building UI with Components

User interfaces can be broken down into smaller building blocks called components. Components allow you to build self-contained, reusable snippets of code. If you think of components as LEGO bricks, you can take 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 modularity allows the code to be more maintanable as it grows since you can easily add, update and delete components without touching the rest of the application.

The nice thing about React components if that they are just JavaScript. THe following is an example of a React component:

Creating components

In React, components are functions - a component is a function that returns UI elements. Inside the return statement of the function you can write JSX:

```
function Header() {
    return (<h1>Develop. Preview. Ship. </h1>)
}
```

To render this component to the DOM, you can pass it as an argument in the `ReactDOM.render()` method:

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

function Header() {
...
}

ReactDOM.render(<Header/>, app)
```

Note that React components should be capitalized to distinguish them from plain HTML and JavaScript. React components should also be used the same way we would use regular HTML tags with angle brackets `<>`.

Nesting Components

Applications usually include more content than a single component - you can nest React components inside each other like you would with regular HTML elements.

This could look something like:

```
function Header() {
    return <h1>Develop. Preview. Ship.</h1>
}

function HomePage() {
    return (
        <div>
            {/* Nesting the Header component */}
            <Header />
        </div>
    );
}

ReactDOM.render(<Header />, app);
```

Component Trees

You can keep nesting React components this way to form component trees. For example a HomePage component could hold a `Header`, an `Article` and a `Footer` component. These components can in turn have their own child components and so on. The `Header` could contain a `Logo`, `Title` and `Navigation` component.

This modular format allows you to reuse components in different places inside your app. In the previous example, since Homepage is the top-level component, you can pass it to the ReactDOM.render() method. Next we discuss props and how to use them to pass data between components.

Displaying Data with Props

If we were to reuse the `<Header />` component, it would display the same content both times:

```
function Header() {
    return <h1>Develop. Preview. Ship.</h1>
}

function HomePage() {
    return (
        <div>
            <Header />
            <Header />
        </div>
    );
}
```

What if we want to pass different text or you do not know the information ahead of time because we are fetching data from an external source?

Regular HTML elements have attributes that we can use to pass pieces of information that change the behavior of those elements. For example, changing the `src` attribute of an `<img>` element changes the image that is shown. Changing the `href` attribute of an `<a>` tag changes the destination of the link.

In the same way, we can pass pieces of information as properties to React components. These are referred to as `props`.

Similar to a JavaScript function, you can design components that accept custom arguments (or props) that change the component's behavior or what is visibly shown when it is rendered to the screen. These props can then be passed from parent components to child components.

> In React, data flows down the component tree. This is referred to as one-way data flow. State, which will be discussed below, can be passed from parent to child components as props.

Using props

In the `Homepage` component, we can pass a custom `title` prop to the `Header` component just like you would pass HTML attributes.

```
function HomePage() {
    return (
        <div>
            <Header title="React" />
        </div>
    );
}

function Header(props) {

}
```

If we use `console.log()` on props, we can see that it is an object with a title property. Since props is an object, we can use object destructuring to explicitly name the values of props inside your function parameters. We can replace the content of the `<h1>` tag with your title variable.

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

If this is opened in the browser we see that it displays the actual word title - react thinks that we are intending to render a plain text string to the DOM. We need a way to denote to React that this a JavaScript variable.

Using Variables in JSX

To use the variable defined, we can use curly braces `{}`, a special JSX syntax that allows us to write regular JavaScript directly inside of the JSX markup.

```
return <h1>{title}</h1>;
```

we can think of curly braces as a way to enter Javascript land when using JSX. Any JavaScript expression (that evalues to a single value) can be used inside curly braces. This includes:

1. An object property with dot notation

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

2. A template literal:

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

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

```javascript
function createTitle(title){
    if (title){
        return titiel;
    } else {
        return `Default title`;
    }
}
~~~

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

```javascript
function createTitle(title){
    if (title){
~~~~

~~~
        return titiel;
    } else {
        return `Default title`;
    }
}

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators
~~~

~~~
3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

function Header({ title }) {
~~~

~~~
    return <h1>{createTitle(title)}</h1>;
}

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

```javascript
function createTitle(title){
    if (title){
        return titiel;
    } else {
~~~

~~~~
        return `Default title`;
    }
}

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function
~~~~

~~~
```javascript
function createTitle(title){
    if (title){
        return titiel;
    } else {
        return `Default title`;
    }
}

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

~~~
4. Ternary operators

3. The returned value of a function

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

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

~~~
4. Ternary operators

3. The returned value of a function

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

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

~~~
4. Ternary operators

3. The returned value of a function

```javascript
function createTitle(title){
    if (title){
        return titiel;
    } else {
        return `Default title`;
    }
~~~

~~~
}

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

```javascript
function createTitle(title){
~~~

~~~
    if (title){
        return titiel;
    } else {
        return `Default title`;
    }
}

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators
~~~

~~~
3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

```javascript
function createTitle(title){
    if (title){
        return titiel;
    } else {
        return `Default title`;
    }
}
~~~

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

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

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

4. Ternary operators

3. The returned value of a function

```javascript
function createTitle(title){
    if (title){
        return titiel;
~~~~

~~~
    } else {
        return `Default title`;
    }
}

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

4. Ternary operators

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

You can pass any string to the title prop since we have accounted for the default case in the component with the ternary operator. The component now accepts a generic title prop which we can reuse in different parts of the application - all we need to do is change the title.

Iterating through lists