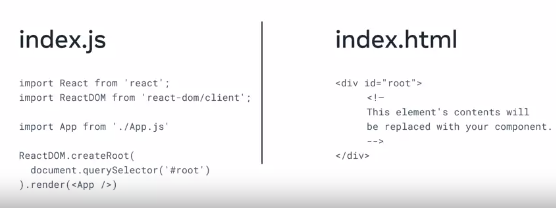
**BASIC TYPES OF NAVIGATION**

Linking and Routing

Stephen Krug's famous book on user experience’ Don't Make Me Think’ sums up the rule that developers are following today

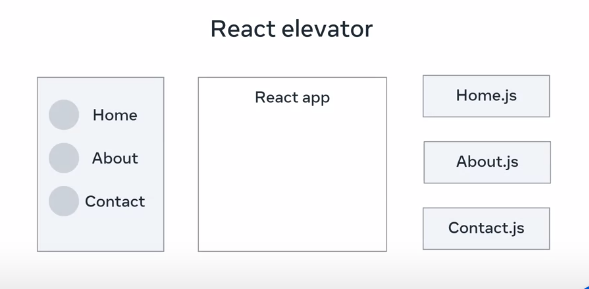
Navigation components

React navigation

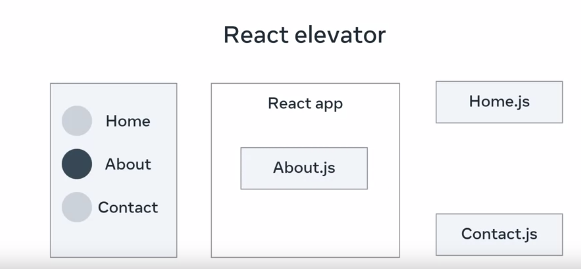


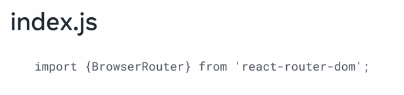
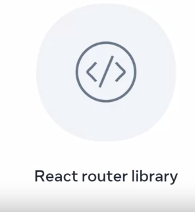






If you click about it moves to react app





**Navigation**

In this reading, you’ll learn about the differences between traditional web pages and React-powered web pages (SPAs – single page applications).

Once you understand the difference between these two ways of building web pages, you will be able to understand the necessary difference between how navigation works in traditional web apps versus how it works in modern SPA websites.

**Before Single-Page Apps**

Before the advent of modern JavaScript frameworks, most websites were implemented as multi-page applications. That is, when a user clicks on a link, the browser navigates to a new webpage, sends a request to the web server; this then responds with the full webpage and the new page is displayed in the browser.

This can make your application resource intensive to the Web Server. CPU time is spent rendering dynamic pages and network bandwidth is used sending entire webpages back for every request. If your website is complex, it may appear slow to your users, even slower if they have a slow or limited internet connection.

To solve this problem, many web developers develop their web applications as Single Page Applications.

**Single-Page Apps**

You’re using many Single Page Applications every day. Think of your favorite social network, or online email provider, or the map application you use to find local businesses. Their excellent user experiences are driven by Single Page Applications.

A Single Page Application allows the user to interact with the website without downloading entire new webpages. Instead, it rewrites the current webpage as the user interacts with it. The outcome is that the application will feel faster and more responsive to the user.

**How Does a Single-Page App Work?**

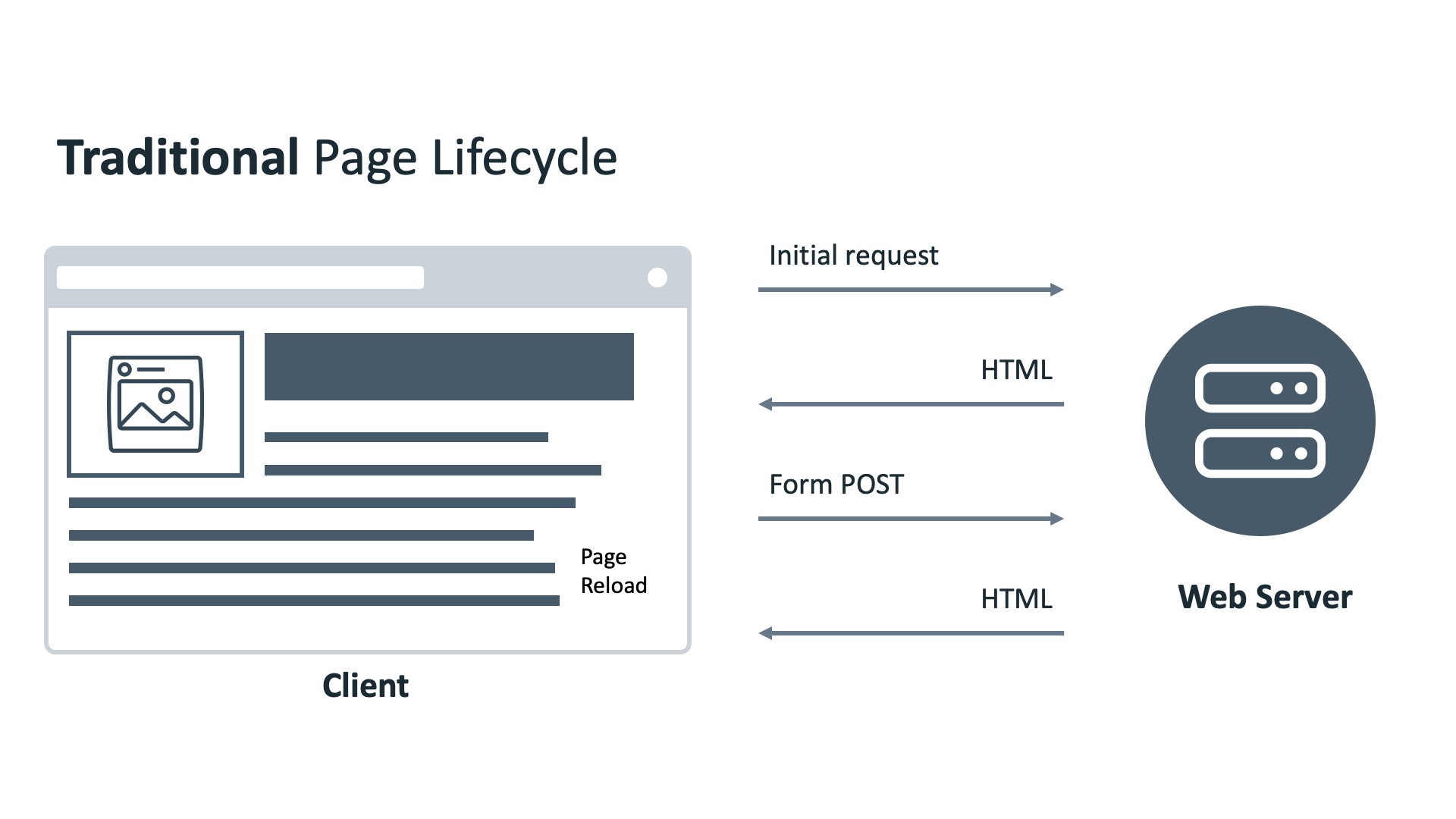
When the user navigates to the web application in the browser, the Web Server will return the necessary resources to run the application. There are two approaches to serving code and resources in Single Page Applications.

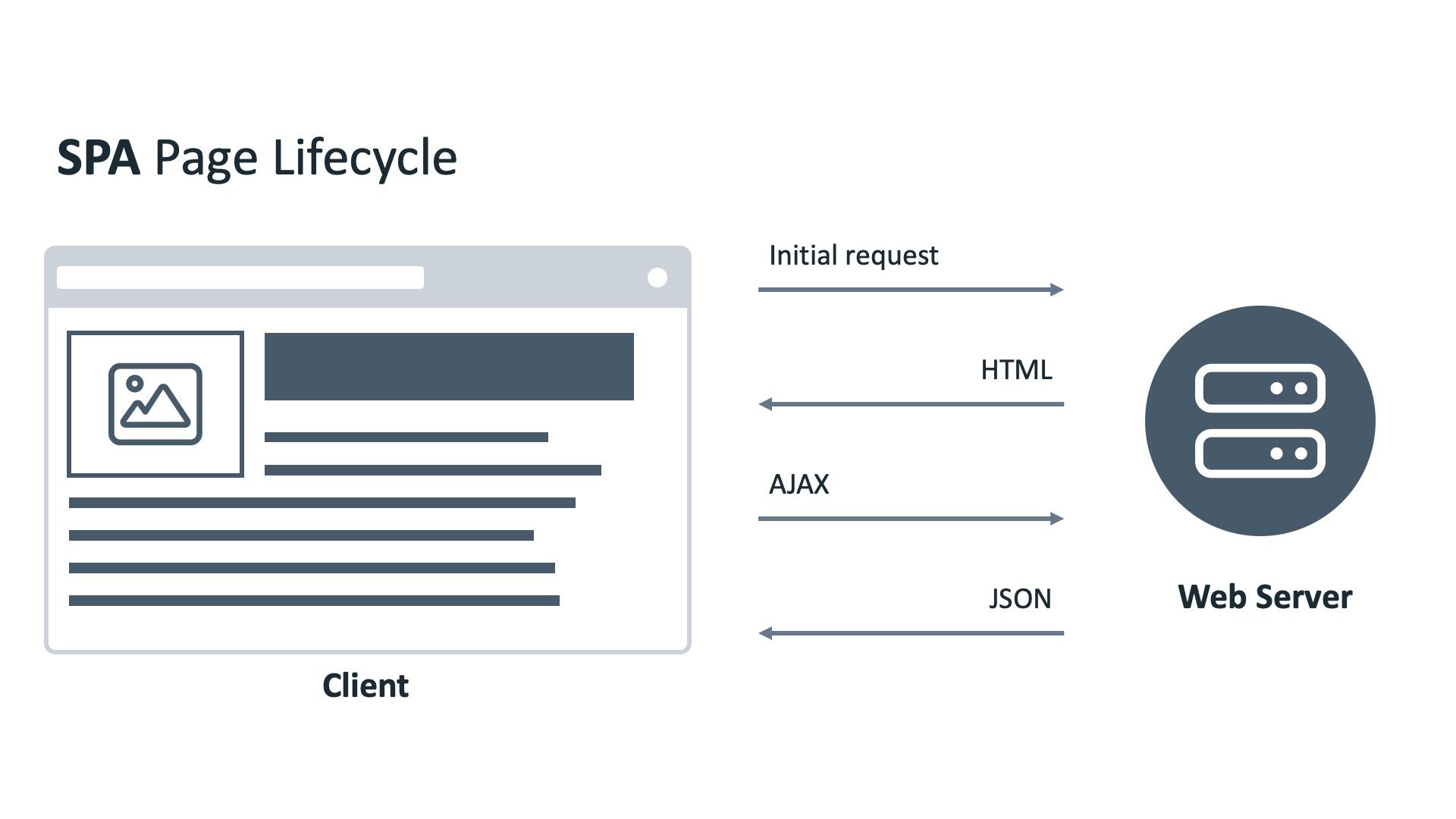
1. When the browser requests the application, return and load all necessary HTML, CSS and JavaScript immediately. This is known as *bundling*.
2. When the browser requests the application, return only the minimum HTML, CSS and JavaScript needed to load the application. Additional resources are downloaded as required by the application, for example, when a user navigates to a specific section of the application. This is known as *lazy loading* or *code splitting*.

Both approaches are valid and are used depending on the size, complexity and bandwidth requirements of the application. If your application is complex and has a lot of resources, your bundles will grow quite large and take a long time to download – possibly ending up slower than a traditional web application!

Once the application is loaded, all logic and changes are applied to the current webpage.

Let’s look at an example.





**An Example of a Single-Page App**

Imagine there is a webpage that has a Label and a Button. It will display a random movie name when the button is clicked.

In a traditional website, when the button is clicked, the browser will send a POST request to the web server. The web server will return a new web page containing the button and movie name, and the web browser renders the new page.

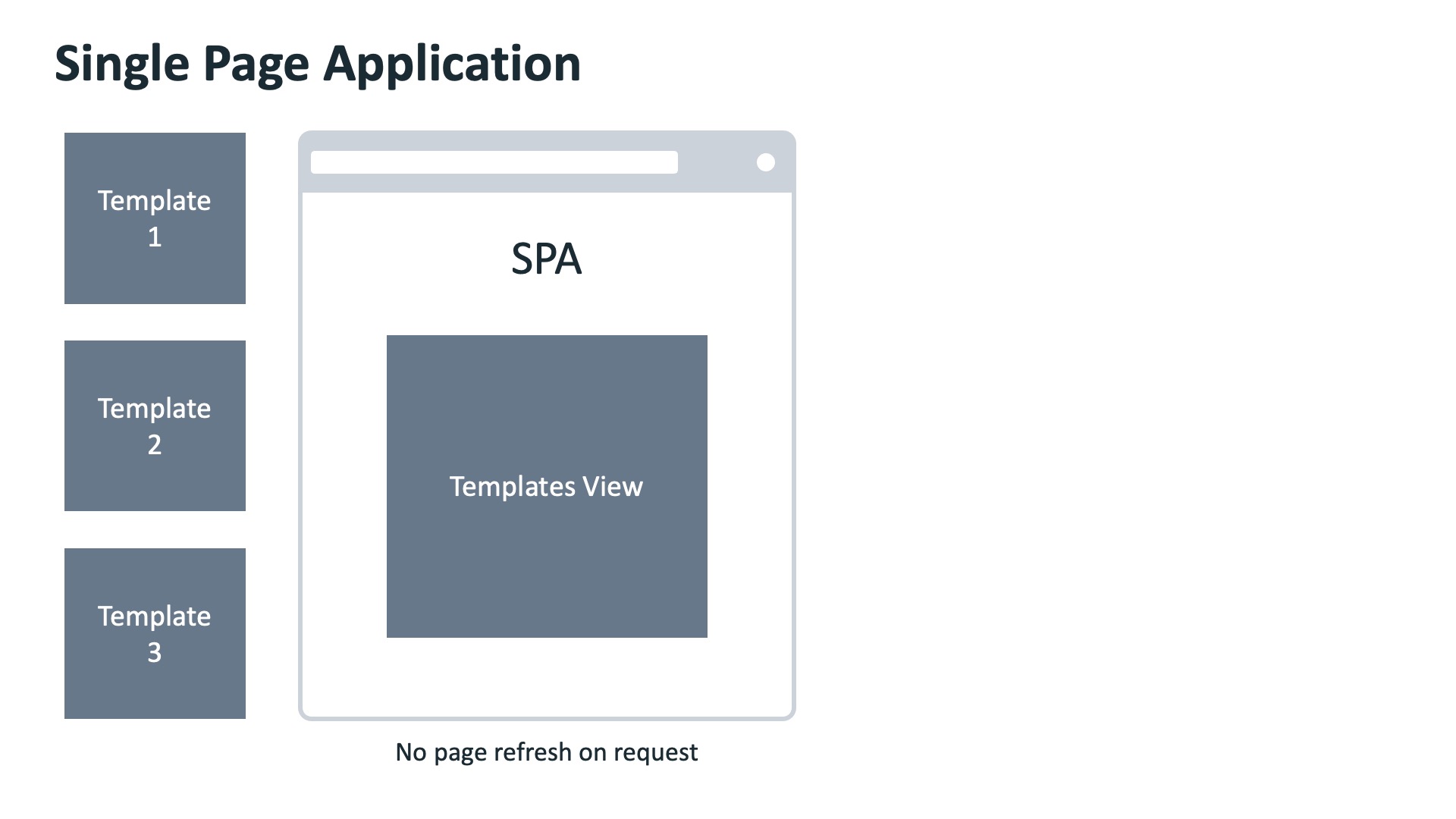
In a Single Page Application, when the button is clicked, the browser will send a POST request to a web server. The web server will return a JSON object. The application reads the object and updates the Label with the movie name.

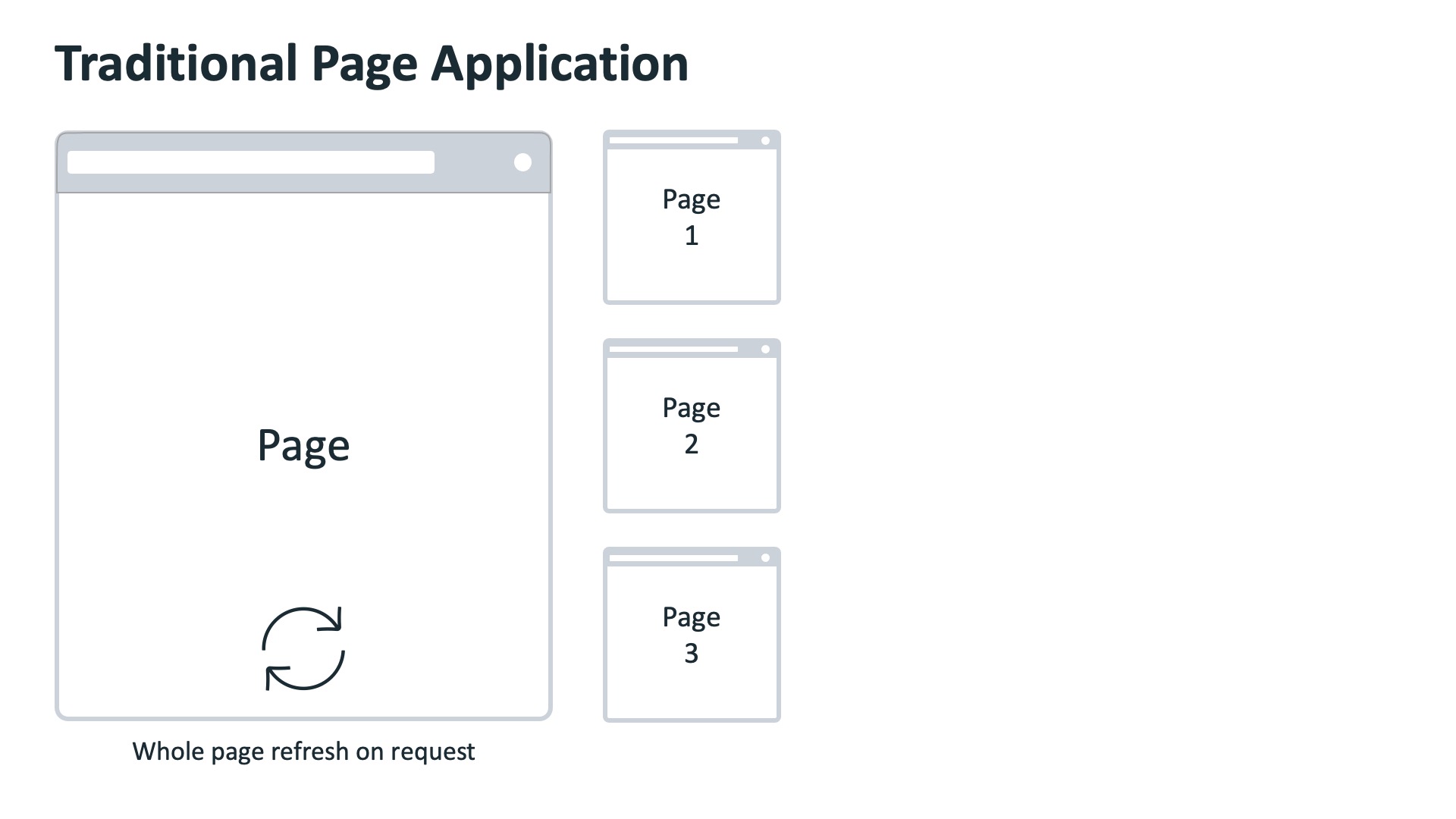
See, more efficient!

But what if we need to have multiple pages with different layouts in our application?

Let’s look at another example.

**Practical Differences Between Single-Page Apps and Multi-Page Apps**





You have a web application that has a navigation bar on top and two pages. One page shows the latest news, and the other shows the current user’s profile page. The navigation bar contains a link for each page.

In a traditional website, when the user clicks the Profile link, the web browser sends the request to the web server. The web server generates the HTML page and sends it back to the web browser. The web browser then renders the new web page.

In a Single Page Application, different pages are broken into templates (or views). Each view will have HTML code containing variables that can be updated by the application.

The web browser sends the request to the web server, and the web server sends back a JSON object. The web browser then updates the web page by inserting the template with the variables replaced by the values in the JSON object.

**Anchor Tag Elements in Single-Page Elements**

A single-page application can’t have regular anchor tag elements as a traditional web app can.

The reason for this is that the default behavior of an anchor tag is to load another HTML file from a server and refresh the page. This page refresh is not possible in a SPA that's powered by a library such as React because a total page refresh is not the way that a SPA works, as explained earlier in this lesson item.

Instead, a SPA comes with its own special implementation of anchor tags and links, which only give an illusion of loading different pages to the end user when in fact, they simply load different components into a single element of the real DOM into which the virtual DOM tree gets mounted and updated.

That's why navigation in a single-page app is fundamentally different from its counterpart in a multi-page app. Understanding the concepts outlined in this lesson item will make you a more well-rounded React developer.

**Applying conditional rendering**

State is all the data your app is currently working with. With this in mind, you can decide to conditionally render specific components in your app, based on whether specific state data has specific values. To make this possible, React works with the readily available JavaScript syntax and concepts.

Consider a minimalistic productivity app.

The app takes the client computer’s current datetime, and based on the data, displays one of two messages on the screen:

1. For workdays, the message is: “Get it done”
2. For weekends, the message is: “Get some rest”

There are a few ways you can achieve this in React.

One approach would include setting a component for each of the possible messages, which means you’d have two components. Let’s name them **Workdays** and **Weekends**.

Then, you’d have a **CurrentMessage** component, which would render the appropriate component based on the value returned from the **getDay()** function call.

Here’s a simplified **CurrentMessage** component:

function CurrentMessage() {

    const day = new Date().getDay();

    if (day >= 1 && day <= 5) {

        return <Workdays />

    }

    return <Weekends />

}

Instead of calculating it directly, you could use some historical data instead, and perhaps get that data from a user via an input, from a parent component.

In that case, the **CurrentMessage** component might look like this:

function CurrentMessage(props) {

    if (props.day >= 1 && props.day <= 5) {

        return <Workdays />

    }

    return <Weekends />

}

**Conditional rendering with the help of element variables**

To further improve your **CurrentMessage** component, you might want to use element variables. This is useful in some cases, where you want to streamline your render code - that is, when you want to separate the conditional logic from the code to render your UI.

Here’s an example of doing this with the **CurrentMessage** component:

function CurrentMessage({day}) {

    const weekday = (day >= 1 && day <= 5);

    const weekend = (day >= 6 && day <= 7);

    let message;

    if (weekday) {

        message = <Workdays />

    } else if (weekend) {

        message = <Weekends />

    } else {

        message = <ErrorComponent />

    }

    return (

        <div>

            {message}

        </div>

    )

}

The output of the **CurrentMessage** component will depend on what the received value of the day variable is. On the condition of the day variable having the value of any number between 1 and 5 (inclusive), the output will be the contents of the **Workdays** component. Otherwise, on the condition of the day variable having the value of either 6 or 7, the output will be the contents of the **Weekends** component.

**Conditional rendering using the logical AND operator**

Another interesting approach in conditional rendering is the use of the logical **AND** operator **&&**.

In the following component, here's how the **&&** operator is used to achieve conditional rendering:

function LogicalAndExample() {

    const val = prompt('Anything but a 0')

    return (

        <div>

            <h1>Please don't type in a zero</h1>

            {val &&

                <h2>Yay, no 0 was typed in!</h2>

            }

        </div>

    )

}

There are a few things to unpack here, so here is the explanation of the **LogicalAndExample** component, top to bottom:

1. First, you ask the user to type into the prompt, specifying that you require anything other than a zero character; and you save the input into the **val** value.
2. In the return statement, an **h1** heading is wrapped inside a **div** element, and then curly braces are used to include a JSX expression. Inside this JSX expression is a single **&&** operator, which is surrounded by some code both on its left and on its right sides; on the left side, the val value is provided, and on the right, a piece of JSX is provided.

To understand what will be output on screen, consider the following example in standard JavaScript:

true && console.log('This will show')

If you ran this command in the browser’s console, the text ‘This will show’ will be output.

On the flip side, consider the following example:

false && console.log('This will never show')

If you ran *this* command, the output will just be the boolean value of **false**.

In other words, if a prop gets evaluated to **true**, using the **&&** operator, you can render whatever JSX elements you want to the right of the **&&** operator.

# Conditional components

Have you ever visited a website that required a user account? To log in you click on a **Log in** button and once you’ve logged in, the **Log in** button changes to a **Log out** button.

This is often done using something called conditional rendering.

In a previous course, you’ve already learned about simple conditions using if and switch statements.

Using these statements allows you to change the behaviour of code based on certain conditions being met.

For example, you can set a variable to a different value based on the result of a condition check.

let name;

if (Math.random() > 0.5) {

    name = "Mike"

} else {

    name = "Susan"

}

let name;

let newUser = true;

if (Math.random() > 0.5 && newUser) {

    name = "Mike"

} else {

    name = "Susan"

}

Conditional rendering is built on the same principle. By using conditions, you can return different child components. This is often done using the props that are passed into the parent component, but can also be done based on component state.

Let’s take a look at a simple example.

Let’s say you have two child components called **LoginButton** and **LogoutButton**; each displaying their corresponding button.

In the parent component, named **LogInOutButton**, you can check the props passed into the parent component and return a different child component based on the value of the props.

In this example, the props contains a property named **isLoggedIn**. When this is set to **true**, the **LogoutButton** component is returned. Otherwise, the **LoginButton** component is returned.

function LogInOutButton(props) {

const isLoggedIn = props.isLoggedIn;

  if (isLoggedIn) {

    return <LogoutButton />;

  } else {

  return <LoginButton />;

}

Then when the **LogInOutButton** parent component is used, the prop can be passed in.

1

<LogInOutButton isLoggedIn={false} />

This is a simple example showing how you can change what is displayed based on a condition check. You will use this often when developing React applications.

**Bundling assets**

Earlier, you learned what assets are in React and the best practices for storing them in your project folders.

In this reading, you will learn about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. You will also learn about the trade-offs inherent in using asset-heavy apps.

The app’s files will likely be bundled when working with a React app. Bundling is a process that takes all the imported files in an app and joins them into a single file, referred to as a **bundle**. Several tools can perform this bundling. Since, in this course, you have used the **create-react-app** to build various React apps, you will focus on webpack. This is because webpack is the built-in tool for the **create-react-app**.

Let’s start by explaining what webpack is and why you need it.

Simply put, webpack is a module bundler.

Practically, this means that it will take various kinds of files, such as SVG and image files, CSS and SCSS files, JavaScript files, and TypeScript files, and it will bundle them together so that a browser can understand that bundle and work with it.

Why is this important?

When building websites, you could probably do without webpack since your project's structure might be straightforward: you may have a single CSS library, such as Bootstrap, loaded from a CDN (content delivery network). You might also have a single JavaScript file in your static HTML document. If that is all there is to it, you do not need to use webpack in such a scenario.

However, modern web development can get complex.

Here is an example of the first few lines of code in a single file of a React application:

import React from 'react';

import '@atlaskit/css-reset';

import styled from 'styled-components';

import './index.css';

import { ThemeProvider } from './contexts/theme';

import { DragDropContext } from 'react-beautiful-dnd';

import { BrowserRouter as Router, Route, Switch } from 'react-router-dom';

import Nav from './components/Nav';

import data from './data';

import Loading from './components/Loading';

The imports here are from fictional libraries and resources because the specific libraries are not necessary. All these different imports can be of various file types: .js, .svg, .css, and so on.

In turn, all the imported files might have their own imported files, and even those might have their imports.

This means that depending on other files, all of these files can create a **dependency graph**. The order in which all these files are loading is essential. That dependency graph can get so complex that it becomes almost impossible for a human to structure a complex project and bundle all those dependencies properly.

This is the reason you need tools like webpack.

So, webpack builds a dependency graph and bundles modules into one or more files that a browser can consume.

While it is doing that, it also does the following:

* It converts modern JS code - which can only be understood by modern browsers - into older versions of JavaScript so that older browsers can understand your code. This process is known as *transpiling*. For example, you can transpile ES7 code to ES5 code using webpack.
* It optimizes your code to load as quickly as possible when a user visits your web pages.
* It can process your SCSS code into the regular CSS, which browsers can understand.
* It can build source maps of the bundle's building blocks
* It can produce various kinds of files based on rules and templates. This includes HTML files, among others.

Another significant characteristic of webpack is that it helps developers create modern web apps.

It helps you achieve this using two modes: **production** mode or **development** mode.

In **development** mode, webpack bundles your files and optimizes your bundles for updates - so that any updates to any of the files in your locally developed app are quickly re-bundled. It also builds source maps so you can inspect the original file included in the bundled code.

In **production** mode, webpack bundles your files so that they are optimized for speed. This means the files are minified and organized to take up the least amount of memory. So, they are optimized for speed because these bundles are fast to download when a user visits the website online.

Once all the source files of your app have been bundled into a single bundle file, then that single bundle file gets served to a visitor browsing the live version of your app online, and the entire app’s contents get served at once.

This works great for smaller apps, but if you have a more extensive app, this approach is likely to affect your site’s speed. The longer it takes for a web app to load, the more likely the visitor will leave and move on to another unrelated website. There are several ways to tackle this issue of a large bundle.

One such approach is code-splitting, a practice where a module bundler like webpack splits the single bundle file into multiple bundles, which are then loaded on an as-needed basis. With the help of code-splitting, you can **lazy load** only the parts that the visitor to the app needs to have at any given time. This approach significantly reduces the download times and allows React-powered apps to get much better speeds.

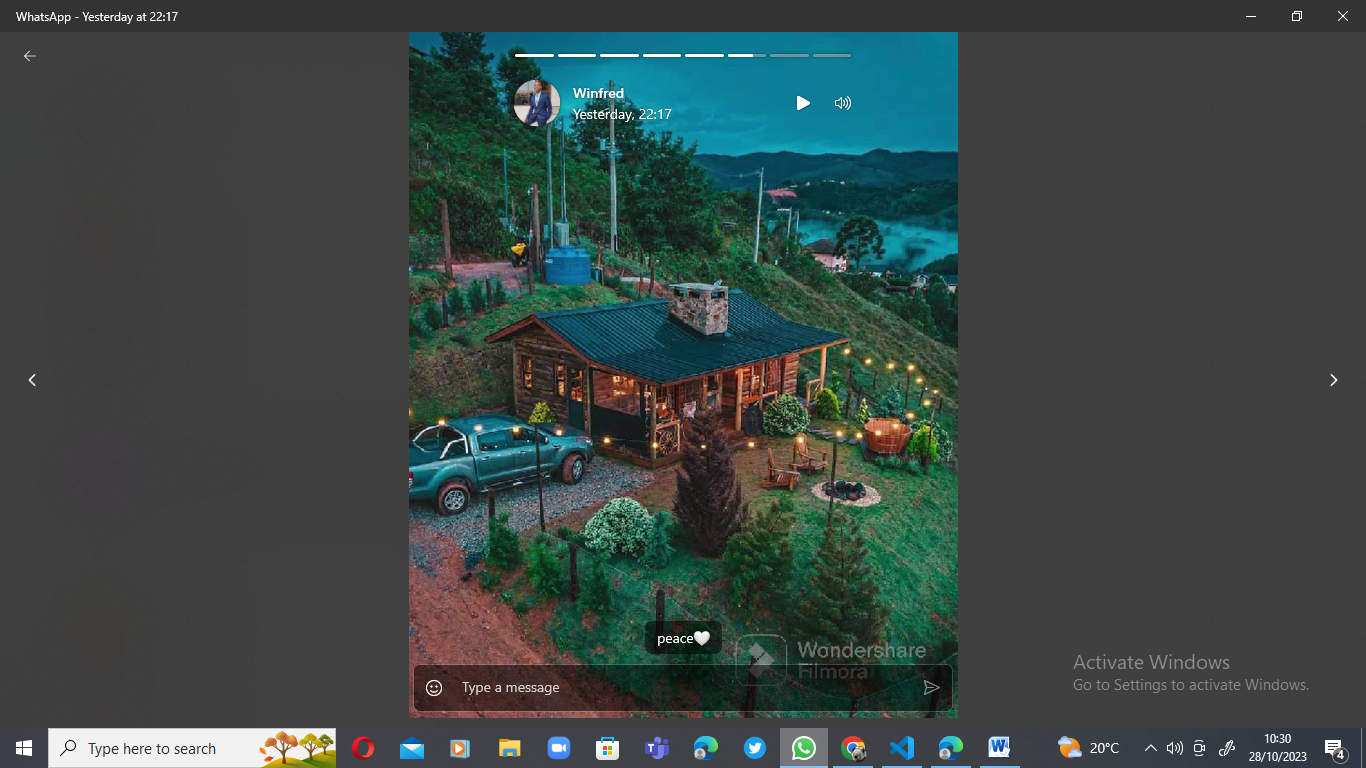
There are other ways to tackle these problems.

An example of a viable alternative is SSR (Server-side rendering).

With SSR, React components are rendered to HTML on the server, and the visitor downloads the finished HTML code. An alternative to SSR is client-side rendering, which downloads the index.html file and then lets React inject its own code into a dedicated HTML element (the **root** element in **create-react-app**). In this course, you’ve only worked with client-side rendering.

Sometimes, you can combine client-side rendering and server-side rendering. This approach results in what’s referred to as **isomorphic apps**.

In this reading, you learned about the advantages and disadvantages of embedding assets, including examples of client/server-side assets. You also learned about the trade-offs inherent in the use of asset-heavy apps.



# Media packages

In this reading, you’ll learn how to install the reactjs-media npm package.

You can find this package on the npmjs.org website at the following URL:

<https://www.npmjs.com/package/react-player>

To install this package you'll need to use the following command in the terminal:

1

npm install react-player

Once you have this package installed, you can start using it in your project.

There are a few ways that you can import and use the installed package. For example, to get the entire package's functionality, use the following import:

1

import ReactPlayer from "react-player";

If you are, for example, only planning to use videos from a site like YouTube, to reduce bundle size, you can use the following import:

import ReactPlayer from "react-player/youtube";

Here’s an example of using the react-player packaged in a small React app:

const MyVideo = () => {

  return (

    <ReactPlayer url='https://www.youtube.com/watch?v=ysz5S6PUM-U' />

  );

};

export default App;

    </div>

  );

};

    <div>

      <MyVideo />

const App = () => {

  return (

import React from "react";

import ReactPlayer from "react-player/youtube";

In this reading, you learned how to install and use the react-player npm package.