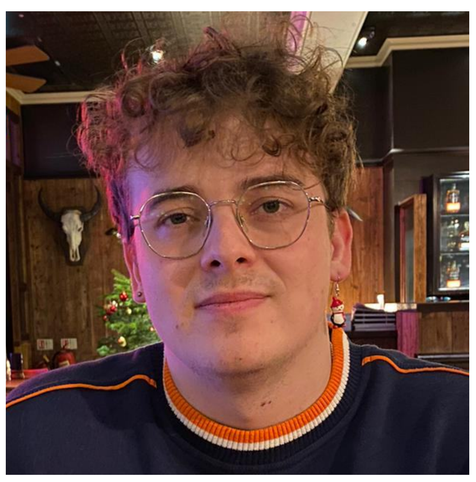
What price are you willing to pay for abstraction?

As programmers we are always seeking shortcuts. And why not? Simplifying complex tasks is a principle of our profession. But, as all principles must be asked, where do we draw the line? This presentation explores this line through the case study of **Create React App (CRA)**, a framework that hid development hell underneath its promise of abstraction. First, we will analyse why the framework was depreciated by the React team, before using this understanding to better understand CRA users' choice: start again, or break free from the framework.



Michael Keay

I am a software developer from Essex who has been coding for 8 years. Growing up, I only took interest in creative subjects, so when I chose to pursue a career in computer science, people thought I was copying my older brother. However, coding has always been a creative endeavour to me, and, whether it helps or just complicates my work, it is the reason I love software development.

My main interest is creating web applications (with a slight obsession with Spotify client apps) using React to build upon the apps and services I use day to day. I am currently studying Software Engineering in my third year at The University of Portsmouth, where I am working on my engineering project that aims to link Spotify tracks to their original samples

Very quickly, what is React?

React is a JavaScript library for building dynamic user interfaces. The defining feature of React is its ability to combine script and markup into one file so that JavaScript objects can be defined by, *what are essentially*, HTML elements.

```jsx
const myDiv = <div>Hello World!</div>;
```

We can use this to create **React components** (a function that follows React's rules and returns markup) that function as custom markup elements.

By integrating the way we render our applications into their logic, React has become a pillar of the growing trend of **single page applications (SPAs)**.

```jsx
const MyComponent = (props) => {
    const name = props.name.split(" ");
    return (
        <div>
            <p key="first">{name[0]}</p>
            <p key="last">{name[1]}</p>
        </div>
    );
};

const App = () => (
    <div>
        <MyComponent name="John Doe" />
        <MyComponent name="Jane Doe" />
    </div>
);
```

```jsx
const App = () => (
    <Routes>

        <Route path="/">
            <p>Hello World!</p>
        </Route>

        <Route path="/name">
            <MyComponent />
        </Route>

    </Routes>
);
```

React is the **second most popular JavaScript library** on the entire internet (BuiltWith, 2024). It has a massive community with over 20 million developers accessing the React docs every month (React, 2024), and has been adopted across the industry

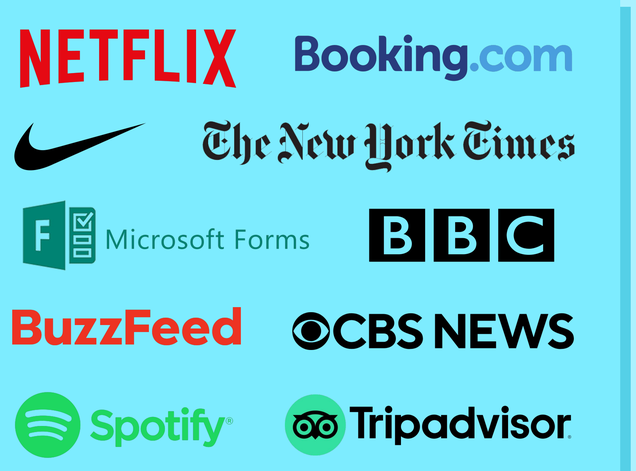


Figure 1: *Popular (top 10k) sites using React (BuiltWith, 2024).*

Nowadays React is used to create **16.54% of the top 1 million** most popular websites and **39.76% of the top 10 thousand** (BuiltWith, 2024). Simply put, if a website is dynamic enough to feel more like a native application than a web app, then it is likely to be written in React.

But straight away we are met with an issue. Whilst React is a JavaScript library, that previous code was not JavaScript. React instead would like us to write our code in a React **JavaScript XML (`.jsx`)** file. However, this syntax is not recognised outside of React, therefore, for React's JavaScript XML code to be run in the browser, it must be transpiled to standard **JavaScript (`.js`)**.

JavaScript XML (`.jsx`)

```jsx
const element = (
    <div>
        <h1>Hello, world!</h1>
        <p>Welcome to React.</p>
    </div>
);
```

JavaScript (`.js`)

```js
const element = React.createElement(
    'div',
    null,
    React.createElement('h1', null, 'Hello, world!'),
    React.createElement('p', null, 'Welcome to React.')
);
```

So that's React 101. It's a rich library with a wealth of features for front-end developers of which we have little time to get into. React has a great development community, and if you're interested in learning, there are plenty of resources out there.

For more information on what you can do with React go to the **Official React Docs**:

Create React App

Create React App (CRA) is a framework created by the React team in 2016. CRA aimed to be the starting point for new and experienced developers to begin their development of a React project without having to configure a development environment. The framework became the official way to begin a React project (Karrys, 2021) and can be seen as a step in the majority of beginner React tutorials from its release to its depreciation in 2023.

Early last year, React's endorsement of the project was removed from its official documentation, and soon CRA was officially depreciated. Its creator, Dan Abramov, released a statement around this time via a GitHub issue comment. The comment by Abramov gives deep insight into the state of React today and I recommend it to those interested in the library.

https://github.com/reactjs/react.dev/pull/5487#issuecomment-1409720741

In his statement, Abramov attributes the depreciation of the framework to a shift in the community towards server-side rendering and static site generation, or more specifically, the framework's inability to evolve with this movement as a client-side focused framework. But this interpretation paints the problem as inherent when, in reality, CRA's downfall comes from years of the framework's decline.

Why was CRA depreciated?

CRA makes its goal clear. "Set up a modern web app by running one command" (Create React App, n.d.). To use a tool like a bundler, transpiler or a testing framework we would typically have to install our chosen library and create our configuration file. You can think of these configurations as settings and instructions telling the library how to function. Only then can we start actually rendering our React components. Alternatively, Create React App replaces these steps. By installing its template, we get a development environment that transpiles, bundles, lints and tests our code out of the box. This naked application comes ready to run whatever React component's we write in the browser, both on a development server and in distributable builds. There are only a few dependencies, and no configuration files to manage whatsoever. But what if there's a specific feature we want to implement in our build process? Or we just want to open our app up and understand what's happening under the hood. Well, CRA gives us a script for that too with "eject". However, when ejected, CRA is exposed.

7 dependencies turn to 53. Zero configuration turns to 1,365 lines worth of configuration files with 14 build plugins. You could be easily mistaken for thinking this added technology is to make the framework run faster or provide more features, but CRA has grown a reputation years before its depreciation of being slow, bloated and incompatible with many different plugins, features and concepts.

Despite its depreciation, 2.94% of the 10k most popular sites are made using CRA, likely due to a mix of outdated projects and the framework's prevalence on old tutorials.

But what about those who did move on? What took CRA's place?

So, CRA is dead. What's Next?

NEXT.JS

Next.js is, to prelude a more technical explanation, Create React App's cooler younger sibling that we all like more.

A subsidiary of the cloud provider **Vercel**, Next.js has been developed with server-side operations and with optimisation for large projects in mind, making the framework a natural evolution from CRA.

Next.js, like its predecessor, predominantly uses **Webpack** to build our application with **Babel** to transpile our code. Whilst complex, Next.js's configuration is fast and powerful unlike the outdated bloat of CRA.

However, as of recent, Next.js have been experimenting with a new bundler, **Turbopack**. The option is much faster, but is still in its beta, therefore restricted to development builds only.

```
"dev": "next dev --turbo",
```

NEXT.JS

Client-Side Rendering

CRA produces a client-side app. The browser is downloading our entire application bundle, limiting its speed by the user's bandwidth. Next, the browser must run all the code, limiting its speed by the user's processing power. Then, we'll likely have to load some data, which means another download. Finally, we re-render our components with the new data, and our app is ready. This process was pretty standard back in 2016, but nowadays, it's slow.

Server Side Rendering (SSR)

SSR means that when the user requests the site, instead of just sending them our bundled app, we also render the full page for the browser. This means that, instead of the user waiting for the browser to run all of our code and fetch their data, the browser can take over the fully rendered page with all of our content ready.

This is much faster, especially for apps with user-specific content or complex single-page applications.

Static Site Generation (SSG)

But what if we don't have a complex app? What if we have a portfolio or a blog where our content isn't changing between versions?

SSG is a much simpler solution for these pages. Instead of rendering content in the browser (client-side rendering) or per the request (SSG), we just render it once during our build process. Now, we have a fully rendered static page that we can send on request, giving our client and our server very little to do.

It's a simple concept, but still cannot be done by CRA out of the box.

However, what really makes Next.js's setup so special is that we can choose between SSR and SSG on a page by page basis. User profile? SSR. Terms and conditions? SSG. This allows us to gain the benefits of both methods in our application out-of-the-box.

And finally, as previously stated, Next.js is a subsidiary of the cloud hosting platform Vercel. So what does this mean for Next.js? Firstly, Next.js is a development powerhouse thanks to the resouces Vercel have put into the project, but, far more interestingly, the framework has been designed to seamlessly integrate into Vercel's cloud hosting services, meaning that Next.js is set up, not just for development and production, but deployment also.

But what if we don't want to jump from one restrictive framework to another?
What do these frameworks really do?

Want to see these frameworks and the following code snippets in full? Go to:

github.com/mwkeay/the-cra-curse

If you have visited this presentations companion GitHub repository, you are likely as aware as my course leaders of my fascination with "the bare minimum". But here, in this environment, that fascination comes from a desire to learn.

Using **Create React App (CRA)**, we abstracted our entire development environment. In doing so, many developers got comfortable building their applications in the centre of CRA's development fog. When the React landscape changed, CRA could not, and therefore, neither could its users.

Next.js's abstraction does not deserve such bleak imagery. The framework outdoes its predecessor at almost every turn and shows hopeful signs that it will continue to build itself with the evolution of the React community in mind.

So why not stop here and herald Next.js as our solution? Because restrictive development frameworks will never truly satisfy my desire for creation and my desire for understanding. There must be a part of your developer brain, however small a piece, that wonders: How does this work? Could I make this myself? And what would happen if I did?

In their simplest form, frameworks like CRA and Next.js are just a collection of popular libraries and configuration files.

On the right, we have two incredibly simple configurations of **ESLint** (mistake and coding style checks) and **Jest** (testing framework). We must note, these configurations are a starting point to illustrate how simple they can be.

However, these are just the added development features that we expect from CRA and Next.js. What about the core of what these frameworks really provide? Bundlers.

```
$ npm i -D eslint
```

eslint.config.js

```
export default [
    {
        rules: {
            semi: "error",
            // ...
        }
    }
];
```

{...} package.json

```
"lint": "eslint src/**/*.{js,jsx}",
```

```
$ npm i -D jest
```

jest.config.js

```
export default {
    verbose: true,
    // ...
};
```

{...} package.json

```
"test": "jest",
```

You can also use the prebuild script to automatically test our app before building.

```
"prebuild": "npm run test",
```

webpack.common.js

```
/**
 * ==============
 *    Modules
 * ==============
 */

const path = require("path");

// Webpack plugin to use hashed filenames in `index.html`
const HtmlWebpackPlugin = require("html-webpack-plugin");

/**
 * ====================
 *    Configuration
 * ====================
 */

// Common configuration
const common = {

    // Entry point for running the application
    entry: "./src/index.js",

    // Output configuration
    output: {

        // Specify distributable folder
        path: path.resolve(__dirname, "dist"),

        // Add hash to filename to avoid cached `main.js` errors
        filename: "main.[contenthash].js",
    },

    // Set to use Webpack's environment specific additions.
    // Read more in Webpack's docs: https://webpack.js.org/concepts/targets/.
    target: "web",

    // Specify file types to resolve
    resolve: {
        extensions: [".js", ".jsx", ".json"],
    },

    module: {

        rules: [

            {
                // Babel loader for JavaScript and JSX files
                test: /\.(js|jsx)$/,
                exclude: /node_modules/,
                use: "babel-loader",
            },

            {
                // CSS loader for CSS files
                test: /\.css$/,
                use: [
                    "style-loader",
                    "css-loader",
                ],
            },
        ],
    },

    plugins: [

        // Replaces "main.js" with "main.[contenthash].js" in index.html
        new HtmlWebpackPlugin({
            template: path.resolve(__dirname, "public", "index.html"),
        }),
    ],
};

// Export
module.exports = common;
```

webpack.dev.js

```
/**
 * ==============
 *    Modules
 * ==============
 */

const path = require("path");

// Function to merge production config with base config
const { merge } = require("webpack-merge");

// Webpack plugin to add static files from `./public` to build
const CopyPlugin = require("copy-webpack-plugin");

// Base config
const common = require("./webpack.common.js");

/**
 * ====================
 *    Configuration
 * ====================
 */

// Production configuration
const prod = {

    mode: "production",

    // Source maps link minified build code to the original source code for debugging.
    // "hidden-source-map" hides source maps from end-users, also reducing bundle size.
    // Note: for debugging production errors, consider using "source-map" instead to ma
    devtool: "hidden-source-map",

    plugins: [

        // Webpack plugin to add static files from `./public` to build
        new CopyPlugin({

            patterns: [
                {
                    // Copy static files from the public directory
                    from: "public",

                    // Paste static files in a new directory inside the distributable
                    to: "./static",

                    // Ignores index.html as HtmlWebpackPlugin handles index.html
                    globOptions: { ignore: ["**/index.html"] },

                    // Stops errors when public folder has only index.html
                    noErrorOnMissing: true
                },
            ],
        }),
    ],
};

// Merge production config with base config
const merged = merge(common, prod);

// Export merged config
module.exports = merged
```

webpack.prod.js

```
/**
 * ==============
 *    Modules
 * ==============
 */

// Function to merge development config with base config
const { merge } = require("webpack-merge");

// Base config
const common = require("./webpack.common.js");

/**
 * ====================
 *    Configuration
 * ====================
 */

// Development configuration
const dev = {

    mode: "development",

    // Source maps link minified build code to the original source code for debugging.
    // "eval-source-map" offers a balance of build performance and debuggability, ideal fo
    devtool: "eval-source-map",

    // Development server configuration
    devServer: {

        // Server port number (e.g. "3000" serves client at http://localhost:3000)
        port: "3000",

        // Serve static files from "./public"
        static: ["./public"],

        // Prevents automatically opening the client in browser.
        // Note: Change to `true` for default behaviour.
        open: false,

        // Enables Hot Module Replacement (live reloads without a full page refresh)
        hot: true,

        // Enables automatic reloading of the page when changes are made to the source code
        liveReload: true,
    },
};

// Merge development config with base config
const merged = merge(common, dev);

// Export merged config
module.exports = merged;
```

The heart of your application's build pipeline will be the configuration of your bundler. To the left is mine. A short stripped down **Webpack** configuration.

Like CRA, this configuration groups and minimises our code, creates source maps, runs a development server, transpiles .jsx files, hashes our bundle and more.

Is it fast? No. Is it production ready? This is a question of standards. But it's my baby and I'm proud. Only through this have I really given myself an opportunity to understand its alternatives, and whilst I have no intention of deploying an app anywhere near this thing, one day I'll do it proud.

"Configuration hell" is a term developers like to throw around, but, in keeping with its grand imagery, I say, forge your understanding in that fire. See what you can make. See how fast it builds your app and see how fast that app runs. At the end of that journey, the smart decision will likely be to return to a framework, but you'll return with an understanding much deeper of why you're there.

Finally, to send you on that journey, I would lke to introduce you to something that might make it a whole lot easier, and give you a good reason to stay independent of a framework.

Everything will be all **Vite**

Vite is a build tool that has gained a lot of traction in the last few years. The library stands, for the purposes of this presentation, as a middle ground between a framework and a custom configuration. Similar to the likes of Next.js and CRA, Vite gives us a build tool that works out of the box with no configuration. However, unlike CRA and Next.js, Vite is incredibly lightweight, fast in development and concerns itself with only our builds.

```
$ npm i -D vite @vitejs/plugin-react-refresh
```

{...} package.json

```
"dev": "vite dev",
"build": "vite build"
```

vite.config.js

```
import { defineConfig } from "vite"
import reactRefresh from "@vitejs/plugin-react-refresh"

export default defineConfig({
  plugins: [reactRefresh()]
})
```

Vite is incredibly easy to setup in your environment. Your first experience of its lightweight philosphy will be the speed at which the library installs compared to alternatives.

For development builds (running your application locally through a development server) Vite uses **ESBuild**, an insanely fast bundler. ESBuild is written in Go (making it 10-100x faster than bundlers written in JavaScript) and pre-bundles dependencies (Vite, n.d.). The speed of development when using Vite is something I believe every developer should experience.

For production builds, Vite uses **Rollup**, a promising bundler that creates much lighter bundles, meaning we can serve our application faster from our server, increasing our sites Search Engine Optimisation (SEO). The Rollup team are working to match the development speed of ESBuild and stand out as one of the relatively new bundlers with a bright future.

Whilst Vite's builds cannot compete with the likes of Next.js when it comes to suitability for large to monolith applications, Vite has made a name for itself as an efficient developer-friendly build tool that every developer should have in their web development toolkit.

References

BuiltWith. (2024). [Statistics on websites libraries as of May 16, 2024]. Retrieved May 16, 2024, from https://trends.builtwith.com/javascript/React

Vite. (n.d.). *Why Vite*. Vite. Retrieved May 17, 2024, from https://vitejs.dev/guide/why.html

Meta. (2021, January 9). Getting Started. Create React App. Retrieved on April 30, 2024, from https://create-react-app.dev/docs/getting-started

Abramov, D. (2023, January 1). [Statement of Create React App's depreciation in a GitHub issue comment]. Retrieved May 17th, 2024, from https://github.com/reactjs/react.dev/pull/5487#issuecomment-1409720741