



Expertise
and insight
for the future

Phuoc Nguyen

How JavaScript ecosystem and open-source tooling enable a modern era of Single-Page Applications

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

20th April 2021

| | |
|---|---|
| Author Title Number of Pages Date | Phuoc Nguyen How JavaScript ecosystem and open-source tooling enable a modern era of Single-Page Applications 45 pages 20th April 2021 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information Technology |
| Professional Major | Software Engineering |
| Instructors | Janne Salonen, Supervisor |
| <p>Over the years, the term "single-page application" has meant both a specific type of website and a development model for web programming. A website could be regarded as a single-page application when it is designed to replicate the behaviors of a desktop application more than acting as a traditional static web document. It achieved the additional smoothness and dynamicity to your average web experience by taking advantage of structured JavaScript to interact with server-side services and modifies the view as needed.</p> <p>Regarding JavaScript, it has recently become extremely popular and powerful; it can be used to build web applications, servers, mobile applications, games and IoT devices. Huge adoption leads to a broad ecosystem, the community has been building plenty of tools to ease the developer experience.</p> <p>The purpose of this thesis is to study the origins of Single-Page Applications and follow along the evolution of JavaScript and its ecosystem on how they redefine the norms of building modern web applications. To demonstrate the functionalities of the mentioned JavaScript tooling, the thesis will include examples of SPAs as subjects of research, and explore how they are being orchestrated in harmony under-the-hood by tools and libraries such as React.js, Next.js, Webpack.js, TypeScript. In addition, different types of SPA will be taken into account, for example server-side rendering, client-side rendering and static site generation, to measure impact of JavaScript toolkit on the process of creating those applications.</p> <p>In conclusion, when a web application is being developed as a SPA, it can isolate communicating with the server and updating its views into two separate concerns, thus accomplishing huge gains in user experience and preserving bandwidth. However, in order to reach such achievement, modern JavaScript engineering tools remain essential factors in paving the path to build a user-friendly, performant and scalable application.</p> | |

| | |
|----------|---|
| Keywords | Single-Page Application, JavaScript, open-source, tooling, server-side rendering, client-side rendering, static site generation |
|----------|---|

Contents

List of Abbreviations

| | | |
|----------|--|-----------|
| 1 | <i>Introduction</i> | 1 |
| 2 | <i>JavaScript Ecosystem</i> | 2 |
| 2.1 | From JavaScript to ECMAScript | 2 |
| 2.1.1 | The origins of JavaScript | 2 |
| 2.1.2 | JavaScript standardization: ECMAScript | 3 |
| 2.2 | Open-source tooling | 4 |
| 2.2.1 | Package Managers | 4 |
| 2.2.2 | User Interface Libraries | 5 |
| 2.2.3 | Code Formatters and Linters | 7 |
| 2.2.4 | Static Type Checkers | 9 |
| 2.2.5 | Transpilers | 10 |
| 2.2.6 | Module Bundlers | 11 |
| 3 | <i>Single-Page Application</i> | 12 |
| 3.1 | Traditional Web Applications and Single-Page Applications | 12 |
| 3.2 | SPA Implementation Strategy | 13 |
| 3.2.1 | Client-side Rendering | 13 |
| 3.2.2 | Server-side Rendering | 15 |
| 3.2.3 | Static Rendering and Prerendering | 17 |
| 4 | <i>Jamstack Technology</i> | 19 |
| 4.1 | What is Jamstack? | 19 |
| 4.1.1 | JavaScript | 20 |
| 4.1.2 | APIs | 21 |
| 4.1.3 | Markup | 21 |
| 4.1.3.1 | Prebuilding Markup in Jamstack | 22 |
| 4.1.4 | Type of Jamstack Projects | 23 |
| 4.1.4.1 | HTML Content | 23 |
| 4.1.4.2 | Web Applications | 23 |
| 4.1.5 | Advantages of Jamstack compared to traditional and monolith applications | 23 |
| 4.1.5.1 | Cost | 24 |
| 4.1.5.2 | Scale | 25 |
| 4.1.5.3 | Performance | 25 |
| 4.1.5.4 | Security | 27 |
| 4.2 | Jamstack Enablers | 27 |
| 4.2.1 | Introduction of Next.js, the hybrid server-side rendering framework | 27 |
| 4.2.2 | Comparison of Next.js to Gatsby.js and traditional React.js applications | 29 |
| 4.2.3 | Headless CMS | 31 |
| 5 | <i>Implementation</i> | 33 |
| 5.1 | Introduction of SY Store, a local fashion store based in Vietnam | 33 |

| | | |
|------------|--|-----------|
| 5.2 | Applying Jamstack to build a webstore | 34 |
| 5.2.1 | Presentation Layer with Next.js Commerce | 34 |
| 5.2.2 | Content Management with BigCommerce's User Interface | 41 |
| 6 | Conclusion | 44 |
| 7 | References | 45 |

List of Abbreviations

| | |
|------|---|
| SPA | Single-Page Application. |
| API | Application Programming Interface |
| ECMA | European Computer Manufacturers Association |
| AJAX | Asynchronous JavaScript and XML |
| URL | Unique Resource Locator |
| FP | First Paint |
| FCP | First Contentful Paint |
| TTI | Time to Interactive |

1 Introduction

As technology evolves, web applications have become gradually more accessible, and are undoubtedly replacing the legacy desktop applications. Modern web applications have higher standards and greater requirements from users than ever before. Today's web apps are expected to be highly available worldwide and can be used virtually on any platform or screen size. To tackle increasingly complex user experience scenarios, there are two major ways of building websites nowadays: as traditional multi-page applications (MPAs) or as single-page applications (SPAs). SPAs are considered as the norm in the modern web, since they allow a much more linear user experience, fast and responsive interactions, and support rich user interfaces with many features. Compared to the traditional web apps, where functionalities that can be built are quite limited, and often only contain simple operations.

In order to build a full-fledged and efficient SPA, familiarity with JavaScript and client-side programming techniques and libraries are required. With that notion set in stone, JavaScript has a lot to offer to web developers with a variety of different toolkits. Each of them can be categorized extensively based on their responsibility in the process of building a web app, ranging from module bundlers, static type checkers to user interface libraries. Those tools may call for a significant effort to be proficient at them, but ultimately, they are created in order to serve towards a common goal: enable the creation of the best user experiences software has to offer.

The objective of this thesis is to research the origins of Single-Page Applications and demonstrate its effectiveness in enhancing user experience in modern software. Different ways of building SPAs will be constructed, and then assessed and compared to provide the advantages and disadvantages between those methods. In addition, the thesis evaluates several JavaScript open-source toolkits to measure their impacts on the process of implementing such applications, and thus offer an overview of why SPA and JavaScript are so tightly coupled with each other.

The thesis is structured as follow. The first chapter introduces the general background and the goal of the project. The second chapter describes an overview of JavaScript ecosystem and its toolkits applications. The third chapter provides history context of Single Page Applications and how they have affected the standard user experience in modern web application, followed by the fourth chapter which discusses the practical implementations SPAs through Jamstack.

The fifth chapter gives details of the practical case of applying Jamstack and SPA principles to build a modern webstore. Finally, the last chapter states the conclusion of this project.

2 JavaScript Ecosystem

2.1 From JavaScript to ECMAScript

2.1.1 The origins of JavaScript

The first popular web browser in 1993 was NCSA Mosaic. In 1994, a company named Netscape was established to leverage the power of the emerging World Wide Web. Netscape developed the exclusive Netscape Navigator web browser, which dominated the majority of the 1990s. Many of the initial Mosaic developers went to work on the Navigator but the two deliberately had no common code.

At that time in the web, in order to do an operation as simple as checking if the form values are entered correctly by the user must be done by sending the data to the server side to evaluate them. This raises a lot of complications, and Netscape decided that the web needs to have its own scripting language. In 1995, Netscape recruited Brendan Eich agreeing to let him implement Scheme (Lisp-inspired language) in the browser. Before Brendan started his job, Netscape has already partnered with software company Sun (which later got acquired by Oracle) to integrate its Java programming language in the Navigator. [2] As a result, an intensely discussed question at Netscape was why the web needed two programming languages: a scripting language and Java. The advocates of the scripting language offered the following explanation:

We aimed to provide a “glue language” for the Web designers and part time programmers who were building Web content from components such as images, plugins, and Java applets. We saw Java as the “component language” used by higher-priced programmers, where the glue programmers—the Web page designers—would assemble components and automate their interactions using [a scripting language]. [3]

By then, Netscape management had decided to go with the idea to have the core interactive parts of the browser to be implemented in a scripting language that is similar to Java. With that context, popular languages at the time such as Scheme or Python are not the suitable solutions for JavaScript to be based on anymore. Netscape needed a prototype to protect the JavaScript concept against competing propositions. In May 1995, a proposal was completed by

Eich in 10 days. JavaScript's initial code name was Mocha, conceived by Marc Andreessen. It got later renamed to LiveScript, as to align with other Netscape's products that already have the prefix "Live". In early December 1995, Java's influence continues to grow exponentially, and to match the popularity of it, the language was changed to its final name: JavaScript. [4]

2.1.2 JavaScript standardization: ECMAScript

In late 1995, when Microsoft started noticing on the competitive threat the Web posed, the Internet Explorer project was launched in an all-out effort to take ownership of the emerging platform from Netscape. After the release of JavaScript, Microsoft introduced the same language under the different name of JScript in Internet Explorer 3.0. Partially to prevent Microsoft from taking control of the JavaScript language, Netscape decided to begin the JavaScript standardization process and asked the standards organization European Computer Manufacturers Association International (ECMA International) to host the standard. Because at that time, the name JavaScript was trademarked by Sun so it cannot be the name for the standardized language. Hence, the term *ECMAScript* was coined, combined from *JavaScript* and *ECMA*. In essence, JavaScript and ECMAScript have the same meaning, though some distinctions can be made as follows:

- JavaScript means the language and its implementation.
- ECMAScript means the language standard specifications.

ECMAScript specifications are managed and developed by Ecma's Technical Committee 39 (TC39). Its members are companies such as Microsoft, Mozilla, Google, Facebook, Apple, Twitter and others, which appoint employees to participate in committee work. These companies usually compete with each other intensely on the web platform but comes together in the committee to work for a better future of the language. TC39 holds meetings every two months at which delegates elected by members and open-source experts participate. The details of those meetings are open to public in [TC39 GitHub repository](#).

The following is a list of ECMAScript versions and their key features:

- ECMAScript 1 (June 1997): First edition of the standard.
- ECMAScript 2 (June 1998): Editorial changes to keep ECMA-262 in sync with the ISO standard.
- ECMAScript 3 (December 1999): Many new core features are implemented– "do-while, switch, regular expressions, try/catch exception, better string handling"

- ECMAScript 4 (abandoned in July 2008): Might have been a major update but ended up being too ambitious and cause disagreements between the language's administrators.
- ECMAScript 5 (December 2009): Strict mode supported, new array methods, support for JSON, getters and setters, and more.
- ECMAScript 5.1 (June 2011): Minor change to keep Ecma and ISO standards align.
- ECMAScript 6 (June 2015): A major update that carry out many of the planned features of ECMAScript 4. This version is the first one whose official name – *ECMAScript 2015* – is based on the year of publication.
- ECMAScript 2016 (June 2016): First yearly release. The shorter release life cycle resulted in fewer new features compared to the large ES6.
- ECMAScript 2017 (June 2017). Second yearly release.
- Subsequent ECMAScript versions (from ECMAScript 2018 onwards) are always released in June and named after the year it was released. [1]

2.2 Open-source tooling

This section will focus on exploring the open-source tooling that the JavaScript ecosystem offers to its developers.

2.2.1 Package Managers

Packages are collections of code that you can publish and reuse like low-level components, libraries or frameworks. These packages are versioned and installed based on semantic versioning. Applications can use these packages as dependencies, and each package can be either dependent or independent on other packages.

Package managers are software tools that help you automate the process of managing packages as dependencies of individual applications, or of the whole computer's system through global package registry. These tools utilize manifest files to keep track of application's metadata and relevant dependencies, and lock files to control deterministic installations.

In the Node ecosystem, a directory named “node_modules” will contain all the dependencies of the application. The installation process of dependencies goes as follow:

- The package manager starts with resolving the dependencies by making request to the registry.
- Then, it recursively looks up each dependency in the registry, and fetches the relevant package tarballs.
- Finally, it links all the dependencies together based on the metadata in the registry files.

There are several package managers that handle JavaScript packages, and the major three that are actively developed and used by the developers are:

- Node Package Manager, or npm is the most popular JavaScript package manager. It consists of a website to discover packages, a CLI to interact with the packages through terminal, and a global registry to share both private and public packages.
- Yarn is a JavaScript package manager release by Facebook in 2016, compatible with the npm registry. Its major advantages over npm are ultra-fast, consistent and secure CLI client, with easy to memorize shortcuts terminal commands.
- PNPM was released in 2016 to solve the problem of huge “node_modules” folder size. Its way of working is different compared to npm/yarn as it caches all previously installed dependencies in a central folder, and any applications/projects that use those packages will contain links to them, ensuring no duplicate packages are installed.

2.2.2 User Interface Libraries

User Interface libraries are packages that help JavaScript, or web developers in general to easily build a consistent and interactive applications. In the earliest day, the most prominent JavaScript UI library was jQuery. In fact, several useful features of jQuery are incorporated into JavaScript itself at the present, and jQuery has served its purpose to be a supportive bridge that helps millions of web developers to build clean and consistent UI.

Nowadays, the modern methodology to build a rich and interactive application is component-based approach, where UI libraries like React, Angular or Vue is most famous for.

- React is a JavaScript library, with a focus on building reusable components. It is developed and maintained by Facebook and an open-source community of developers. It is introduced in May 2013 and is extensively used and battle-tested by Facebook products (Instagram, Messenger, new Facebook). React can be used to build both web and mobile development. However, apps written with React itself is not sufficient. For complex application, state management, routing or form validation require additional libraries, built on-top of React.
- Angular is full-fledge UI framework, TypeScript-based and is maintained by Google's Angular team and the Angular open-source community. It was first released in 2010, though there is a major shift in philosophy between the first version and second version of Angular. It can be also used to build both web and mobile applications. Compared to React, Angular offer its developers all the necessary tool from the beginning. However, come with that is the steep learning curve, and developers may not use all the feature Angular offer.
- Vue is the youngest member of the group, introduced in 2014 by ex-Google employee Evan You. The framework used to be a one-man project, but nowadays it got a dedicated community of core contributors. Similar to Angular, it also offers a semi-accomplished kit to its developers to build a rich and complex application from the start.

A short comparison table that shows differences of the three major UI Libraries are shown in figure 1.

| Feature | Angular | React | Vue |
|----------------------------|---------|-------|-----|
| UI / DOM Manipulation | ✓ | ✓ | ✓ |
| State Management | ✓ | ✓ | ✓ |
| Routing | ✓ | ✗ | ✓ |
| Form Validation & Handling | ✓ | ✗ | ✗ |
| Http Client | ✓ | ✗ | ✗ |

Figure 1. Feature comparison of Angular, React and Vue. Retrieved from <https://academind.com/tutorials/angular-vs-react-vs-vue-my-thoughts/>

2.2.3 Code Formatters and Linters

Code linter are tools that analyse source code to detect inconsistency and discrepancy in code formatting based on pre-written rules and then output them as warnings or errors. They are used to enhance code quality, configured manually and are run automatically on code changes.

Code formatters are tools to format codebase deterministically based on formatting rules to enforce a unified coding style. They can act as linters in term of formatting rules but have no values when it comes to code quality rules. They are often integrated into workflow together with linters to do both formatting and linting.

Even though JavaScript compilers or static type checkers have evolved to include many of linting functions, linters have also evolved to detect even a wider range of incorrect behaviours: warning about syntax errors, undeclared variables, misuse of scope.

The most popular JavaScript linters are ESLint and JSLint. They provide online Graphical User Interface to try out formatting rules with live coding. They also offer plugins extensions for text editors like VS Code, Sublime Text and Atom.

- ESLint is a highly configurable linter that also support JSX. It can auto format your code to match the preferred formatting style based on pre-written rules. However, its upside is also its downside as too much customisation causes the tool hard to pick up for beginners. A typical ESLint

configuration for a React application can be referenced from figure 2.

```

1  {
2    "parserOptions": {
3      "ecmaVersion": 2021,
4      "ecmaFeatures": {
5        "jsx": true
6      },
7      "sourceType": "module"
8    },
9
10   "settings": {
11     "react": {
12       "version": "detect"
13     }
14   },
15
16   "plugins": [
17     "react"
18   ],
19
20   "rules": {
21     "react/jsx-no-bind": ["error", {
22       "allowArrowFunctions": true,
23       "allowBind": false,
24       "ignoreRefs": true
25     }],
26     "react/no-did-update-set-state": "error",
27     "react/no-unknown-property": "error",
28     "react/no-unused-prop-types": "error",
29     "react/prop-types": "error",
30     "react/react-in-jsx-scope": "error"
31   }
32 }

```

Figure 2. ESLint configuration file for a React project. Retrieve from <https://github.com/standard/eslint-config-standard-react/blob/master/eslintrc.json>

- JSLint is an opinionated linter, based on the book How JavaScript Works by Douglas Crockford. It is super straightforward to start using but come with that is the low level of configuration.

In addition to linters, the most famous code formatters are Prettier and StandardJS, where they are both able to integrate seamlessly with linters to provide an efficient workflow:

- Prettier is no doubt the best code formatter that have ever been created in the JavaScript ecosystem. It supports all major languages like JavaScript, JSX, HTML, CSS and Markdown. It is also able to integrate with popular text editors, easy to use and have a huge ecosystem of plugins.

Difference in formatting when using Prettier can be shown in figure 3.

```

1 function HelloWorld({greeting = "hello", greeted = "World", silent = false, onMouseOver,}) {
2
3   if(!greeting){return null};
4
5   // TODO: Don't use random in render
6   let num = Math.floor(Math.random() * 1E+7).toString().replace(/\.d+/ig, "");
7
8   return <div className="HelloWorld" title={`You are visitor number ${ num }`} onMouseOver={onMouseOver}>
9     {greeting.endsWith(",") ? " " : <span style={{color: 'grey'}}>"}</span> }
10
11     <em>
12       { greeted }
13     </em>
14     { (silent
15       ? " "
16       : "1")}
17
18   </div>;
19 }

```

```

1 function HelloWorld({
2   greeting = "hello",
3   greeted = "World",
4   silent = false,
5   onMouseOver,
6 }) {
7   if (!greeting) {
8     return null;
9   }
10
11   // TODO: Don't use random in render
12   let num = Math.floor(Math.random() * 1e7)
13     .toString()
14     .replace(/\.d+/gi, "");
15
16   return (
17     <div
18       className="HelloWorld"
19       title={`You are visitor number ${num}`}
20       onMouseOver={onMouseOver}
21     >
22       {greeting.endsWith(",") ? (
23         " "
24       ) : (
25         <span style={{ color: "grey" }}>"}</span>
26       )}
27       <em>{greeted}</em>
28       {silent ? " " : "1"}
29     </div>
30   );
31 }
32

```

Figure 3. Code without Prettier vs Code with Prettier. Retrieved from live GUI <https://prettier.io/playground/>

- StandardJS is an all-in-one JavaScript style guide, linter and formatter. Its core principles are no configuration, automated code formatting and catch style issues and programming errors early. It is available both as a npm package and text editor plugins.

2.2.4 Static Type Checkers

JavaScript is a dynamic type checking language, which translate to type safety is only validated at runtime. It brings flexibility to the language, but also cause unexpected errors at runtime. This is where static type checkers come in handy.

Static type checking is the process of checking type safety of source code at compile-time. It offers a lot of great benefits compared to bare bone dynamic type checking: catching errors early, limiting type errors, providing auto-completion, code documentation. JavaScript static type checkers are developed by extending JavaScript with type systems, which will at the end be removed at compile-time. This results in many productive development tools and practices like code refactoring and static checking.

There are two major static type checkers in the JavaScript ecosystem: Flow and TypeScript. They both have their own strength and weaknesses. Using them can greatly improve the confidence in producing high quality code, but it comes with a cost like steep learning curve, or increased verbosity of your code.

- Flow is a static type checker for JavaScript, developed and maintained by Facebook. Flow checks code for errors through static type

annotations, which in turns allow Flow to understand how the code works and ensure it works that way. It is lightweight and easy to set up, but its ecosystem is not as developed as TypeScript's.

- TypeScript is an open-source language which builds on JavaScript, by extending it with static type definitions. It is developed and maintained by Microsoft and is open source. It can act as static type checker, compile to JavaScript at the end, and is supported by major text editors and Integrated Development Environments. Similar to Flow, writing type with it is optional as type inference allows a lot of power without writing additional code.

2.2.5 Transpilers

Transpilers in JavaScript are source-to-source translators that convert JavaScript variants (ClojureScript, ReasonML, TypeScript) or modern JavaScript versions (ES2015+) to equivalent vanilla JavaScript that meets pre-defined constraint like browser compatibility, uglification, minification or strict.

Compiler and transpiler are often used interchangeably in JavaScript world. However, a transpiler translates between programming languages that function at the same degree of generalization, meanwhile a standard compiler will translate from higher-level programming language to a lower-level language like Java to WebAssembly or C++ to binary. [5]

- Babel is the de facto most common JavaScript transpiler. It is a open-source toolkit that is mainly used to transform modern ECMAScript standards into a backward compatible version of JavaScript in targeted browser environments. Babel provides a set of awesome features including: syntax transformers, polyfill features that are missing in the targeted environment, and source code transformations (codemods).[6]
- TypeScript can also act as a transpiler, as it is strictly a syntactical superset of JavaScript with optional static type. As a superset of JavaScript, it also transpiles the code to match targeted ECMAScript standard, which allows developers to support multiple browsers with minimal effort, or to take advantage of new ECMAScript standards early on. In figure 4. is shown how TypeScript acts as a transpiler to compile TypeScript to relevant version JavaScript with pre-defined attributes.

```
1 = {
2 =   "include": [
3 =     "./src/**/*"
4 =   ],
5 =   "compilerOptions": {
6 =     "strict": true,
7 =     "esModuleInterop": true,
8 =     "lib": [
9 =       "dom",
10 =      "es2015"
11 =    ],
12 =    "jsx": "react"
13 =  }
14 = }
```

Figure 4. A typical transpiler configuration for TypeScript project

2.2.6 Module Bundlers

Module bundlers are tools that pack JavaScript applications into one or more bundles, typically used to optimize deployment steps. It internally builds a dependency tree which maps every module that your project needs and generate, based on configuration, one or multiple bundles. The main difference across various module bundlers is how many types of non-JavaScript file they can process (such as CSS, JSON, PNG, JPEG, XML), and whether they will bundle a npm package, a web app on the browser, or back-end apps to run on Node.js.

They are not general-purpose task runners but can be used like one with limited configuration (such as Webpack is used for automating front-end building tasks). Module bundlers are often released as CLI applications which developers can use to run development server, serving static content, building production files or watching changes.

- Webpack is the most popular and powerful module bundler for JavaScript applications. It can process all type of files, and can generate either package, a modern web app or server app. It has a dynamic architecture, with highly customizable configuration, extensive plugins and loaders ecosystem. It also has default support for production optimization that cannot be configured manually such as lazy loading, minification, uglified JavaScript, tree shaking or scope hoisting.
- Rollup is a module bundler that can generate both npm packages and JavaScript applications. It has default support for ES Modules (zero configuration) and famous for tree shaking optimization. It also has an extensive plugin system to customize its behaviour. Rollup is often used to bundle JavaScript libraries, as it optimizes tree shaking much better compared to Webpack. However, for normal JavaScript application, Webpack

is a better alternative since it offers much wider build plugins and default configurations.

- Parcel is the rising dark horse in the module bundler competition, with a focus on offering bundling web application with no configuration. It has out of the box support bundling capability for CSS, HTML, JS file assets with no external plugins required.

3 Single-Page Application

3.1 Traditional Web Applications and Single-Page Applications

Figure 5. blows describe a typical operation flow that happened in a Traditional Web Applications (a) compared to a Single-Page Applications (b), which is popularized through the introduction of AJAX technique (Asynchronous JavaScript and XML)

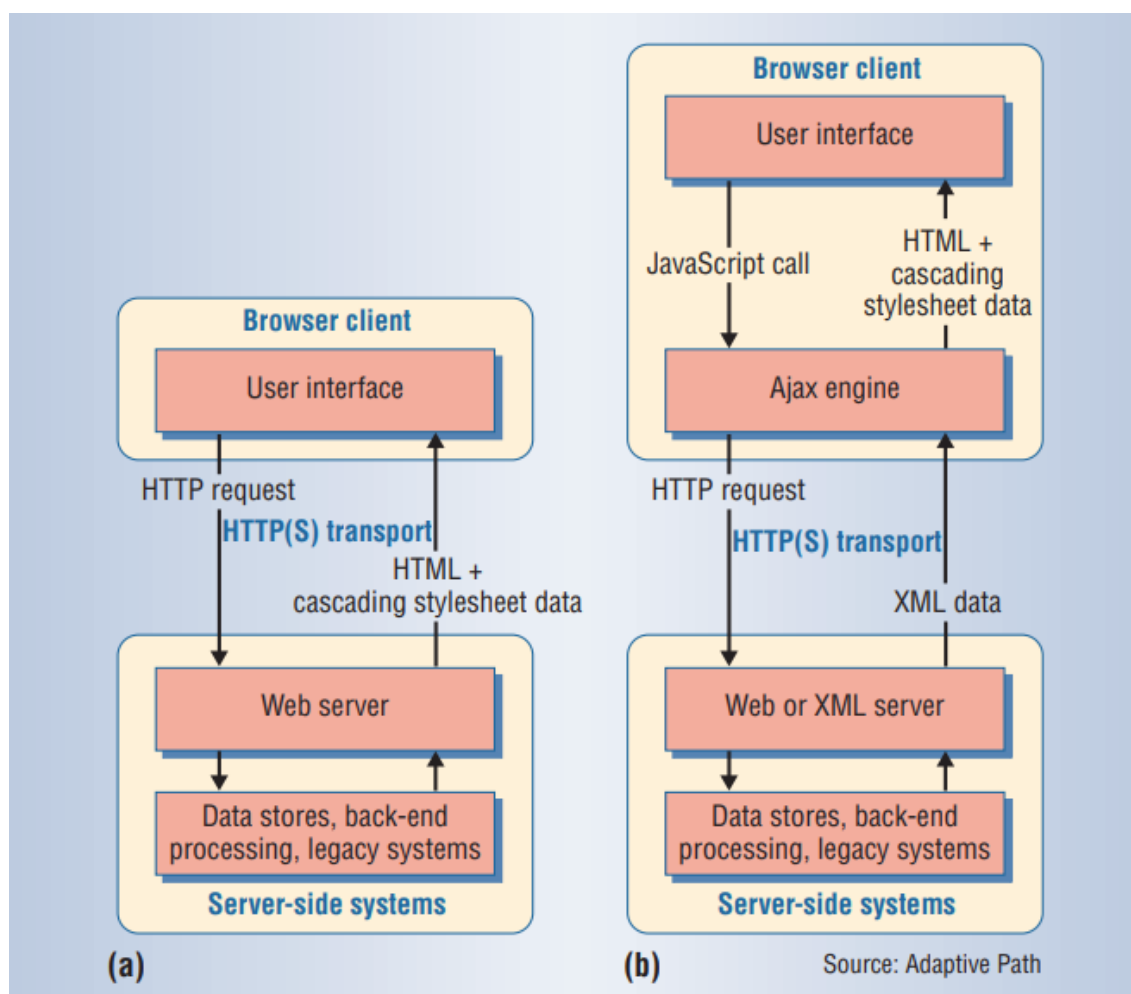


Figure 5. Traditional Web application and Ajax Web application. Retrieve from

<https://www.semanticscholar.org/paper/Building-Rich-Web-Applications-with-Ajax-Paulson/a4777cea6758969602098785d10a599a77fd0d30>

Traditional Web Applications required little to no client-side behaviour, and heavily relied on the server for all actions from navigation, queries to updating the app with relevant information. Each operation on the app would result in a HTTP request to the web server and interact with the server-side systems to handle proper data pieces, which then translate into a full reload on the end user's browser (a HTML page response to the original HTTP request). Classic Model-View-Controller (MVC) frameworks generally adopt this approach, with each request corresponding to a different controller action, which in turn would interact with a model and return a view.

On the other hand, Single-Page Applications involve very few dynamically generated server-side page loads. AJAX technique is the core technology that allows SPA to shine: this technique allows web applications to make dynamic requests to the server without loading a new page, hence the term Single-Page. SPA is typically deployed as a HTML static file that then loads necessary JavaScript libraries to initialize and run the app. These apps make heavy usage of browser's APIs to handle their data requests and in turns provide a much richer user experience as it allows user to interact with the app without reload while the data is being exchanged with the server-side systems.

However, it is worth noting that when building a modern application, a hybrid approach can be achieved through the combination of Traditional Web Application behaviour, mainly for content, and SPAs, for interactivity.

3.2 SPA Implementation Strategy

It is expected for modern web applications to be mostly built with a SPA approach. With that perspective in mind, the web community has developed multiple ways to achieve the interactivity of SPAs with a set of different qualities are being taken into consideration: fast user experience, Search Engine Optimization (SEO) or both. In this section, different methods that are used to build SPA and its trade-offs will be analysed and compared to provide the most suitable technique for a application's needs

3.2.1 Client-side Rendering

Client-side rendering (CSR) in the purest form means HTML pages are rendered directly on the browser using JavaScript. Application's business logic, data accessing, navigation are handled on the client-side (browser) rather than on the server.

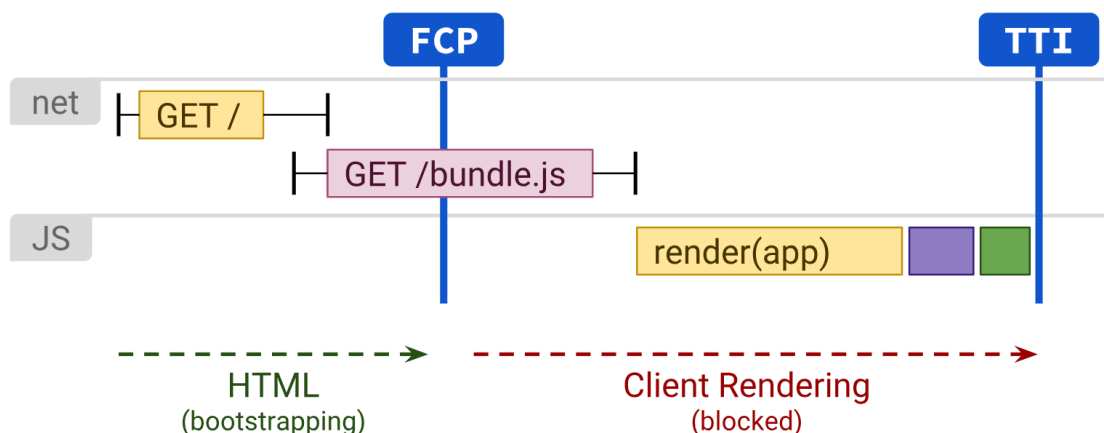


Figure 6. How a request is processed in CSR applications. Retrieve from <https://developers.google.com/web/updates/2019/02/rendering-on-the-web>

With CSR solution, the application redirects the request to a single HTML file and the server will deliver the blank page without any content until the request to fetch all the necessary JavaScript, or First Contentful Paint (FCP) is completed. Then, after the browser compiles every required JavaScript files, the application can be considered fully rendered and ready for Time to Interactive (TTI). [7]

Under a stable and reliable internet connection, this approach can work well. However, its downside was that the size of necessary JavaScript will grow exponentially as the application grows. The initial loading time before FCP will be significantly larger than needed because of the additional new JavaScript polyfills or libraries, which compete for the browser's processing capability and must often be handled before the presentation layer can be fully rendered. [7]

Applications that chose this approach with large JavaScript bundles must consider solutions like aggressive code-splitting and lazy-loading – “serve only what you need, when you need it”. Especially when building a SPA, technique like Application Shell caching can be utilized to cache common parts of User Interface share by most views in the application. An application shell means the core HTML, CSS and JS needed to power a common User Interface [7]. Example of Application Shell caching is shown in figure 7.

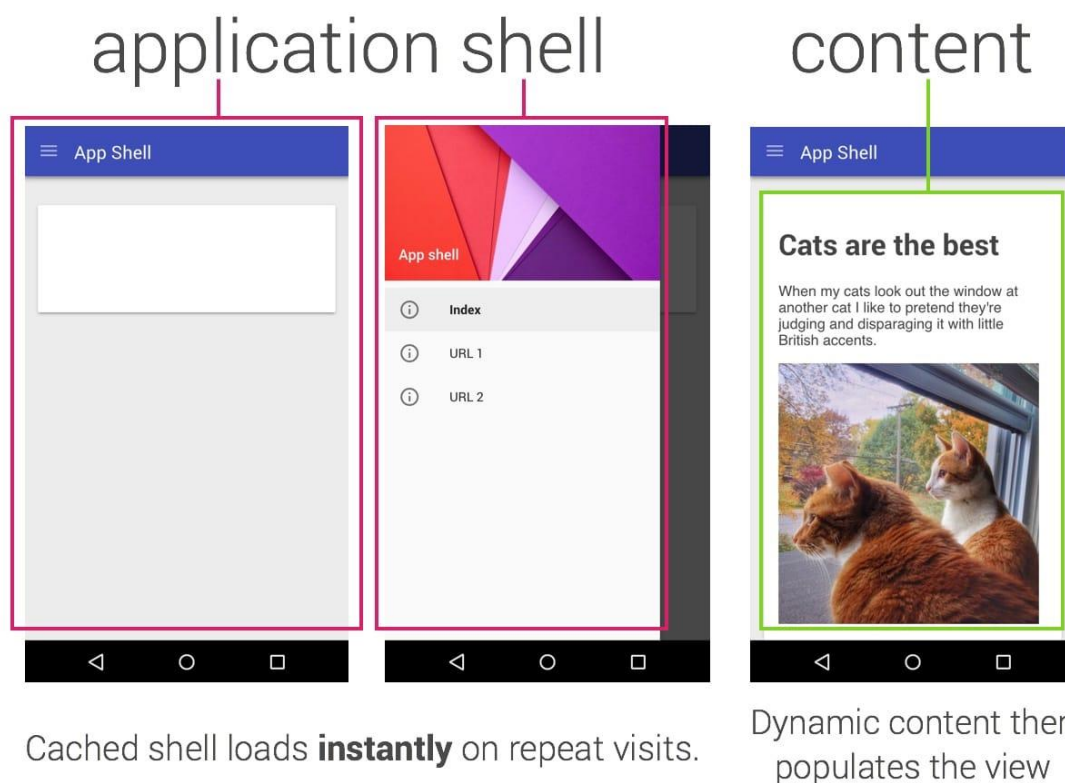


Figure 7. A practical example of Application Shell caching technique in action. Retrieve from <https://developers.google.com/web/updates/2015/11/app-shell>

3.2.2 Server-side Rendering

Server-side rendering compiles a full HTML page on the server corresponding to each user's action. With this approach, additional requests to fetch data or template on the client can be avoided, as those actions are processed before

the browser interacts with the response.

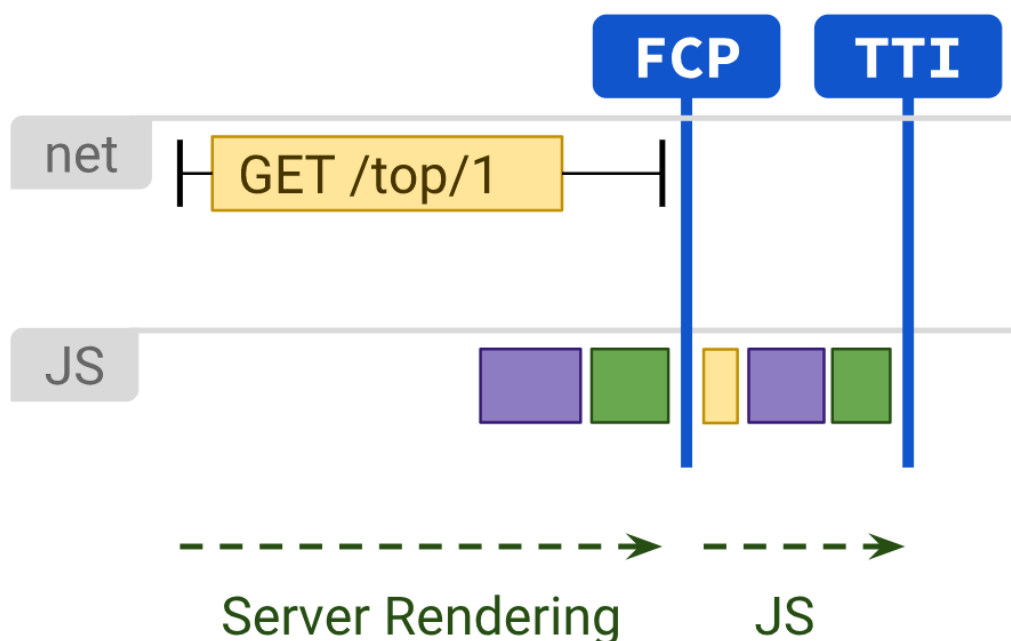


Figure 8. Operation flow of Server Rendering solution. Retrieve from <https://developers.google.com/web/updates/2019/02/rendering-on-the-web>

Figure 8. demonstrates how server-side applications are rendered in response to each user's request. Typically, server rendering can provide a quick First Paint (FP) and FCP meaning requested content can become instantly visible to user. By combining rendering and business logic handling on the server, it stops the process of sending large JavaScript files to the client side, which help achieves a fast TTI.

With server rendering, users can ignore the client-side JavaScript processes before accessing the site. Even when subsidiary JavaScript like ads, analytics, trackers, social media cannot be prevented, using server-rendering to reduce your own primary JavaScript sizes can give more room and performance power for the rest of your application. However, there is one key disadvantage to this approach: generating pages with fully loaded JavaScript take time and can lead to a larger Time to First Byte (TTFB), meaning the time between the client making an HTTP request and the first byte of the page being received by the user's browser will be increased. [7]

Whether server-side rendering approach is the suitable method to build out a web application mainly depends on what kind of experience the developers want to deliver to their users. There has been an enduring discussion over the optimal way to build one's applications of client-side rendering versus server rendering, but there is one critical factor to take in mind is that for some certain pages, server-rendering can be utilized while some others cannot. Some sites have picked up a hybrid technique with success, such as Netflix. On Netflix, it

server-renders the core static pages, while on the background with hybrid approach, prefetching all the JavaScript for highly-interactive pages, enable these client-heavy pages to load quicker compared to traditional approach.

Nowadays, there are many modern architectures and frameworks that allow the possibility to render one's application in a hybrid way, both on the server and the client. These methods can be used to server-side rendering as well, but there is one crucial factor about this approach was that structures of these applications where rendering happens in a hybrid way is its own degree of solution with distinct features and trade-offs. In React, developers can utilize the "[renderToString\(\)](#)" method or solutions built on top of React like [Next.js](#) for either server-side rendering or hybrid approach via rehydration. For Vue there is a complete server-rendering guide on [the Vue official documentation page](#), or [Nuxt.js](#). For Angular, there is [Universal](#).

One important factor when it comes to choosing render strategy in the web is the impact of SEO on the application. More than often enough, server rendering is the most complete technique for SEO as crawler can easily interpret one's application. JavaScript can also be analysed by crawlers, but there are often restrictions and problems bound to JavaScript that should be taken into consideration on how they can affect the parse ability of crawler. Thus, this often influences pure client-side rendering strategy where the application is mostly comprised of JavaScript. However, hybrid approach can be considered to improve your application's SEO if it relied heavily on client-side JavaScript.

3.2.3 Static Rendering and Prerendering

With static rendering, applications generally compile a single HTML file for every page that the user can navigate to ahead of time. Then, these pages can be served through a cloud service like Amazon S3 instances or from a running customized server like nginx. Figure 9. shows how a static rendering site operates: when user navigates to a new Unique Resource Locator (URL), a HTML file will be produced ahead of time as a response to user's actions.

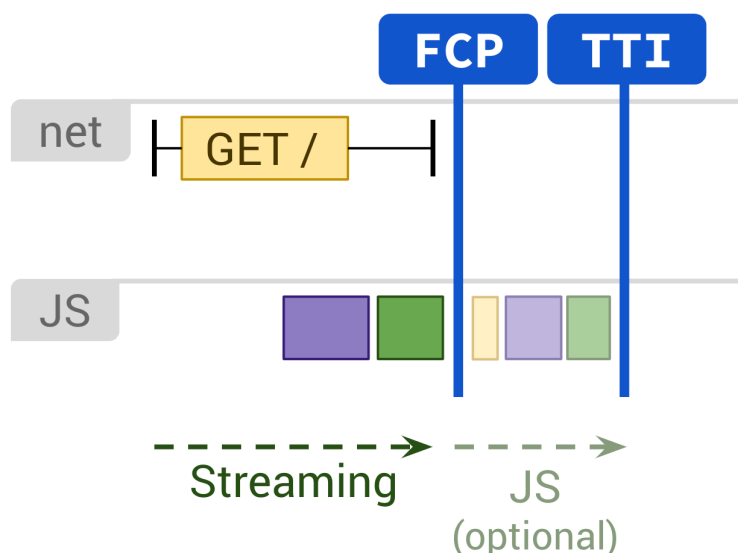


Figure 9. Static Rendering flow. Retrieve from <https://developers.google.com/web/updates/2019/02/rendering-on-the-web>

Static rendering is typically utilized at build-time, which lead to a fast FP, FCP and TTI— assuming the size of JavaScript files is reasonable. Compared to server rendering, static rendering can achieve a quick Time to First Byte since it can avoid generate HTML pages on the fly. With HTML responses are compiled ahead of time, static rendering applications are often deployed to Content Delivery Networks (CDNs) to make use of edge-caching.

One major downside to static rendering is each HTML file must be compiled in response to any available URL. This problem can be difficult and infeasible since more than often developers cannot predict all the possible URLs ahead of time, or the application has many unique pages.

React developers may be used to libraries like [Gatsby](#) or [Next.js static export](#) – these libraries and methods make it favourable to develop static rendering applications or components. However, there is one major distinction between static rendering and prerendering: users can interact with static rendered pages regardless of the client-side JavaScript, compared to prerendering where it improves the FP or FCP of a SPA that must be powered on the client with JavaScript for pages to achieve interactivity.

For static rendered pages, most functionality of the web application are available without JavaScript. For prerendered pages, some basic operations like navigation through link are available, but the page will be mainly inactive. Prerendering mainly requests more JavaScript to achieve dynamic actions, and those bits of JavaScript tend to be more intricate than the [Progressive Enhancement](#) method utilized by static rendering. [7]

4 Jamstack Technology

4.1 What is Jamstack?

The Jamstack at its fundamental is an effort to name a range of popular architecture practices. The term was coined in order to summarize a wide range of architectural decisions. It originated from discussions between communities of static site generators, SPA libraries, build tools and API-driven services as they perceived that the changes they are making in their own domain are interconnected.

The initial wave of static site generators found its popularity when it was supported natively by GitHub Pages. These tools allow a simplicity in control over the convoluted data-heavy approach Content Management Systems (CMSs) that they replaced.

Simultaneously, SPA frameworks like React and Vue initiate a process of decoupling the front-end from the back-end, with the assistance of new JavaScript toolkits like Gulp, Grunt, and later, Webpack, Parcel and Rollup, which all contributed to the notion of a innovative front-end with its own integration and delivery pipeline. This results into a conventional software architecture for front-end applications and allows front-end engineers to develop much quicker on the interactivity layer.

A new API economy starts to emerge when these approaches to build new website and applications intersect. Tools like Google Map, Stripe, Disqus allow features such as map direction, payment and comment available for the front-end to consume through API-driven services.

A new set of expressions that describe these innovative software compositions began to materialize: progressive web apps, static hosting, front-end continuous deployment, SPAs and serverless function. However, none of these terms really captured the big picture of the rising new software structure that incorporates content-driven static site, web applications and all the fascinating combinations between the two. [8, p.8]

The Jamstack name set a cohesive conclusion to all those rising trends.

In modern web architecture, the “stack” has ascended a tier. Before, the common discussion around the “stack” would be about operating system, database and the web server (such as Linux Apache Mysql PHP (LAMP) stack or MongoDB Express Angular Node (MEAN) stack), however, with the new trends of architecture practices, the new paradigm emerges as follow: [8, p.8]

- JavaScript in the browser as runtime. [8, p.8]
- Reuseable HTTP APIs services rather than application's business logic programs. [8, p.8]
- Prebuilt Markup served through CDN as the distribution procedure for Jamstack applications. [8, p.8]

These components are the JAM in the JAMstack. Each of these elements is an important piece of the way to build a modern Jamstack architecture, and they will be analysed in detail in the following chapters.

4.1.1 JavaScript

Nowadays, JavaScript is undoubtedly the most common programming language, especially dominant in web-based application. It has grown from a simplistic, scripting language with Java-like syntaxes into the most highly optimized language in the world. ECMAScript is the official committee behind the advancement of the language, includes of various open-source leaders in the web community. Elegant features are being reviewed and added continuously to the language from time to time, ranging from state-of-the-art constructs for asynchronous action, agile class and module system, to refined syntactic constructs such as destructuring assignment, and an object syntax named JavaScript Object Notation (JSON) that has become the most popular data exchange format on the web. [8, p.9]

JavaScript has evolved from a standalone programming language into a compilation target for transpilers, which convert JavaScript variants into pure operational JavaScript in the targeted browsers, as well as compilers for new or existing languages such as ClojureScript, ReasonML or TypeScript. As a result, JavaScript has transformed into the universal runtime on the web that Sun Microsystems envisioned when they first build the Java Virtual Machine (JVM). [8, p.9]

This web-based "Virtual Machine" is the runtime platform of the Jamstack. It is the place where developers can make modifications when they need to establish dynamic workflows that are not limited to the content and presentation layer, such as fully functional applications or adding extra potent features to a content-based website. As the browsers have developed to become the functional system of the web, JavaScript evolve into the same role in Jamstack as C has in Unix [8, p.9]

4.1.2 APIs

The World Wide Web at its core is just a user interface and presentation level on top of a stateless protocol called the HyperText Transfer Protocol (HTTP). The web's foundational functionality is the Universal Resource Locator (URL). [8, p.9]

The most basic user experience flow of a web application goes as typing a URL into a browser or follows a link from website and finally land on a different website. URL architecture and structure of modern web application should be built with a careful mindset. In addition, URL also allow programs running in the browser the ability to reach any programmatic resource that has been exposed to the web. [8, p.9]

Originally, browser APIs were created only to be utilized in server-side applications. Without techniques like proxying through a mediator server or using external plugins like Flash, it is impossible for a program running in regular browser to consume any browser API outside of its own space [8, p.10]. Since JavaScript emerged from just a scripting language designed mainly to perform minor progressive upgrades for server-side rendered applications into a complete runtime layer, new standards such as WebSocket and CORS appeared. Together with other modern standards such as OAuth2 and JSON Web Token (JWT) for authorization and stateless authentication, any latest browser API has become unexpectedly accessible from any JavaScript client running in the browser. [8, p.10]

This massive innovation on the web has enabled JAMstack emergence as one of the most important architectural guideline for websites and applications. Immediately, the whole web ecosystem has morphed into a massive operating system. A new API economy evolution began to appear—from payments or subscription through advanced machine learning model on the cloud, to services that affect the physical daily life such as traffic direction or shipping services, or anything else people can imagine. Possibly for every practical problem exists, a solution with API-driven services will exist to combat such issue.

4.1.3 Markup

The web's foundational element is the HyperText Markup Language (HTML). HTML, parsed and interpreted by all browsers, represents how content should be distributed and structured on the web. It manages the accessibility to resources and assets for a website and presents a Document Object Model (DOM) that can be parsed, exhibited, and handled by anything from the common web browser to search engine crawler, to mobile devices or smart watches. [8, p.10]

In the beginning days of the web, a website simply consists of a folder of HTML files exposed over HTTP by a web server [8, p.10]. As the web matured, a running program on the server would compile the HTML directly correspond to each user's navigation, normally after communicating with the database. This approach as we discussed in chapter 3 was the traditional way of how web applications are being operated. It was a very slow and complex approach, compared to just serving static assets, but at the time due to the limited technology this is the only viable way to make a simple document viewers web application dynamic. Because of this document-focused software architecture, web experiences are underwhelmed and much less interactive compared to desktop applications.

However, this approach was only the most common way before the emergence of JavaScript and the revolution of modern web APIs available in the browser. Most modern web applications have move away from such legacy architecture, adopting a more progressive approach such as the Jamstack.

4.1.3.1 Prebuilding Markup in Jamstack

Markup is delivered in a different paradigm in Jamstack, compared to the traditional way of frontend web servers build HTML pages at runtime. Alternatively, the Jamstack way was to build all the static assets ahead of time and serve them directly on the browser through CDN. This approach typically involves a build tool such as static site generator like Gatsby or Hugo; or front-end toolkits like Webpack, Parcel or Rollup where all relevant assets are bundled into HTML, scripting files are either compiled or transpiled into vanilla JavaScript, and CSS files are run via pre-processor and post-processor.

This strategy creates a clear distinction and decouples the front-end from any back-end services. It allows a much more simplicity architecture in infrastructure and live system, which results in a much better isolation considering the front-end and individual APIs. In a lot of ways, this decoupled architecture resembles the architecture of mobile apps. Unlike the traditional building model of web application where the full UI would be refreshed from the server each time users interacts with apps, iOS applications are built in such a way that each app is distributed to the users as an app store package with fully functional UI and compiled together with additional assets to interact with JSON or XML-based browser API. This strategy is the same as the Jamstack approach. The front-end is the iOS app. It is distributed to users on browsers through CDN and take advantages of the JavaScript on the browser to communicate with web-based APIs.

4.1.4 Type of Jamstack Projects

4.1.4.1 HTML Content

The simplest form of a Jamstack site is a pure static site: a folder with HTML and supporting assets (JS, CSS, images, fonts); plain-text files that can be kept under version control and be modified on the fly in the chosen editor [8, p.12].

With application that has more than a single page, common elements like headers, navigation, footers or any repeat elements in general can be extracted into their own template and HTML components files. If the application requires JavaScript, modules from npm can be use as well as ES6 modern features available directly from the browser's APIs. If the application's stylesheet starts to grow complex, techniques like pre-processing or post-processing CSS can be applied in order to manage the scalability of the stylesheet.

A proper build toolkit such as Webpack can bundle all these extra complex steps as static assets while maintaining the basic simplicity of the site. The whole basis of such application binds to simple text files that lives in Git-repository, under the effect of version control and can be modified to likings with any text-centric developer tools. In addition, combined with continuous deployment (CD) workflow, publishing these applications becomes as simple as a Git commit and push.

4.1.4.2 Web Applications

True SPAs come to live when JavaScript performance on the web becomes reasonable enough so developers can invert the legacy existing model of relying refreshing pages after each's user navigation to handle all pages transition directly in the browser. The first generation of SPAs was developed as a frontend inside a large, monolith database-driven applications. As JavaScript becomes more and more mature, modern SPAs architectures start to separate the front-end and back-end thoroughly. The emergence of a new generation of front-end build tools improves the SPAs building experience and make working with front-end in isolation extremely satisfied. Webpack, again, is the core enabler for this new wave of building the web: it offers a complete workflow that supports JavaScript and Stylesheet transpiling, pre/post processing, code splitting & lazy loading, as well as watching live changes.

4.1.5 Advantages of Jamstack compared to traditional and monolith applications

In recent years, front-end architectures have proven to be rather sophisticated, but with that is the growth of browser native APIs as well as HTML and CSS. This leads to the capability to be a full-stack developer is rather difficult and

challenging, as it is not an easy task to be both fluent in client-side JavaScript functionalities as well as back-end operations such as database query optimization, cache invalidation or infrastructure work. With Jamstack, decoupling front-end from back-end allow developers to focus on one area of work without sacrificing the cost of performance or maintainability. Nowadays, front-end processes can be run locally with a live watching development server communicating directly with a production API and switching between staging or testing environment is as easy as changing an environment variable.

With Jamstack approach utilizing the microservices architecture, the service layer becomes much more small, focused and maintainable compared to one big monolithic application. This leads to a thriving API economy, with production-ready API driven services like authentication, public discussion, map direction, ecommerce, searching and so on. Jamstack applications can out-source complex domains to these third-party tools with confidence knowing these providers have thoroughly professional teams focus exclusively on tackling problems in their own space.

4.1.5.1 Cost

With traditional architectures, where page requests are highly active at every level of the stack, that volume to respond to the requests are increased through each rank, often leading to numerous, convoluted physical or virtual machines for databases, message queues and load balancers. Each of these infrastructure layers are associated with cost. The cost of these pieces can grow exponentially with software certificates, machine costs and human labor to maintain physical or virtual servers. In addition, in a practical web development project, each application will have multiple environments from testing to staging and production, meaning for each environment, these pieces of infrastructure need to be duplicate to provide suitable requirements, lead to even more absurd costs.

Jamstack sites do not have to deal with such costs as they take advantages of much more straightforward technical architecture. To deal with peak traffics problem, Jamstack sites often offload the difficulty to Content Delivery Network that is serving the application's resources instead of having to scale each level of the stack to deal with such issue. Even if a Jamstack application does not adopt the highly flexible functionalities of CDN, the infrastructure layer would still be significantly rationalized compared to traditional approach. When the operations such as data fetching and page routing are decoupled from the invocations for these operations, it translates to the requirement for these fractions of the infrastructure does not correlate to the traffic of the website at all. Major parts of traditional architecture do not need to be scaled or possibly might not even exist in the Jamstack approach.

In conclusion, the cost to build, develop and maintain website and applications are greatly reduced with the Jamstack.

4.1.5.2 Scale

Even with the most meticulous capacity planning, applications will, at certain points in time, receive heavy spikes in traffic loads. In order to tackle this problems, high traffic sites often adopt the technique to add a caching layer or CDNs to their infrastructure.

In traditional stacks, web applications need to cope with their dynamic data by adding various caching layers and CDNs in front of the services that are dealing with those data. This is a very expensive and complex set of functions. This results in an additional level of complexity just to enable the ability to serve static assets with static hosting infrastructure in order to solve the scaling traffic problems.

Meanwhile, applications built with Jamstack approach are already bundled and prebuilt into the required static assets that are optimized to be served directly through CDN without any additional layers of complexity. Furthermore, CDN will often have nodes distributed globally. Therefore, applications that are served through CDN do not have to deal with the situation where main traffics of the application locate far away from the server. Instead, they can utilize the global characteristic of CDN to distribute the content to anywhere in the world. In addition, when sites are being served through CDN, the application's servers are no longer act as the Single Point of Failure that could block visitor from accessing the site. If a single node within a global CDN fails, the traffic would be redirected through a different healthy node, available at other places in the network.

As such, Jamstack sites prove to be capable of dealing with resiliency, redundancy and capacity in the simplest way compared to the traditional, monolithic applications.

4.1.5.3 Performance

In traditional architectures, performance optimizations are typically conducted in the server side. This leads to back-end engineering leading the performance game and leaving the front-end side behind as a less-complicated field of engineering. However, this has changed considerably in the modern way of building the web. The fact that highly organized architecture and delivery of front-end code could make a significant improvement to the performance of an application has established a crucial field where important enhancements can be conducted.

Let assess a simple request for a page in a dynamic site:

1. The browser requests for a page in response to user's navigation
2. The invocation then is processed by a web server that assess the requested URL and routes the request to the appropriate piece of logic.
3. The web server then routes the request to an application server that deals with logic that is related to data.

4. The application server queries for the relevant data from a database and forms a response with new data to the request.
5. The response is returned from the application server, and then to the web server and then passed back to the browser where it can finally be shown to the user. [8, p.27]

Figure 10. demonstrate such process in the traditional stack.

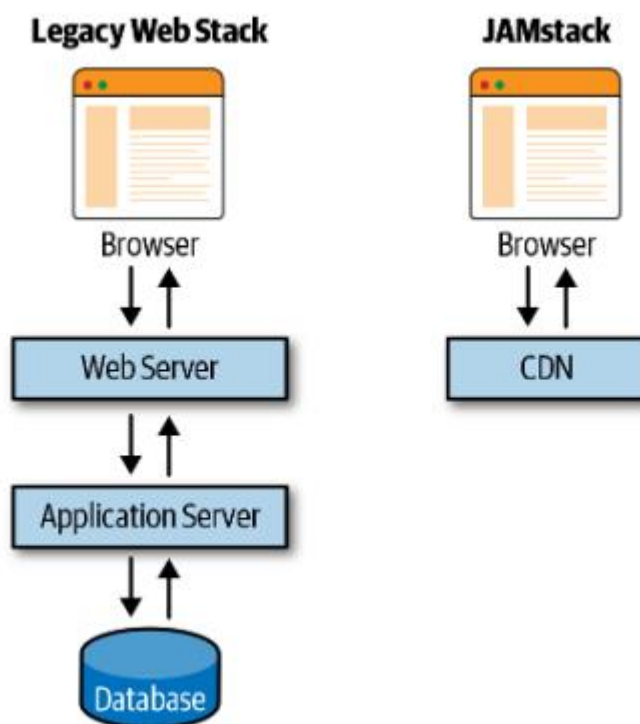


Figure 10. Request flow in Traditional Architecture versus Jamstack.

In legacy stacks like the one mentioned above, with a flexible back-end, in order to improve performance, it is common start adding layers in between to cache data. It can be either a caching level between a web server and an application server, or between the database layer and application server. Such layers have operations act like the layers are static but in fact it must be managed and modified accordingly by the application over the span of operation.

Now let assess the similar request for Jamstack applications instead, demonstrate in figure 10 as well, where the compilation of page views is prebuilt ahead of time instead of being generated on the fly:

1. The browser requests for a page in response to user's navigation
2. A CDN connects the request to a prebuilt response and returns it instantly to the browser and shows to the user.

Consequently, advantages of Jamstack approach are proven immediately with less back-end layers to take care of. There are fewer failure mistakes and fewer systems in the operation to render response for the requests. This leads to improved performance in the hosting environment by default. Resources are

instantly ready to be served to the front-end and shown to the end-user as quickly as possible in recognition of the CDN.

4.1.5.4 Security

As discussed in previous chapters, Jamstack approach provides advantages of a compact and focused stack when compared to legacy architecture with its complicated database and caching layers, each expose the capability to read and interchange data. By ignoring those layers, Jamstack also removes the impact points at which different systems can communicate and exchange data. Occasions of compromising the back-end layers are reduced. The effort needed to spend patching and protecting pieces of infrastructure is simplified. Security is improved.

Moreover, Jamstack sites enable a read-only system compared to the traditional stack where write operations for servers to execute code exists. Thus, Jamstack applications typically can avoid the attacks that normally only happen in legacy web stack. In addition, by out-sourcing certain complexity in the applications to third-party specialist tools, Jamstack advocates for a more rational separation of individual layers and underlying capabilities. This leads to a clear separation of concerns combined with security responsibilities.

4.2 Jamstack Enablers

After the discussion about benefits and details of the Jamstack architecture in previous chapter, this section will focus on the practical implementation of Jamstack and the tooling that enable to build a Jamstack application with all the advantages it offers.

4.2.1 Introduction of Next.js, the hybrid server-side rendering framework

In simple terms, Next.js is a React.js-based framework that focus on offering the static strategy to build Single-Page Applications. Besides that, it also includes out-of-the-box many great features and advantages that enhances developer experience and application's performance such as: Fast Refresh, Zero Config, Hybrid Static Site Generation and Server-side Rendering, Automatic Code Splitting and many more. Even though Next.js is open-source, it is largely maintained by a company named Vercel, which is one of the leading companies that provides the deployment platform for Jamstack applications.

The core benefits of using Next.js compared to bare bone React.js approach was that many of the complex optimization tasks are supported by default in Next.js. When building a React application from scratch, there are many details that need to be considered like code bundling and transpiling through external

build tools like Webpack or Babel, statically pre-render pages to improve performance and SEO or writing server-side code to handle data exchange. All these advantages that it provides to developers make it a perfect tool to build a Jamstack application where all the important optimization techniques are supported. In addition, with Vercel as the deployment platform, Next.js applications have built-in analytics to measure performance based on metrics like First Contentful Paint, Largest Contentful Paint, First Input Delay, which are crucial factors to determine a good Lighthouse score.

One key difference between Next.js and other ways to build React.js application is that Next.js is a React.js framework, where routing, data fetching are built-in. This allows developers to quickly prototype an application without doing research on which libraries to use to handle those features.

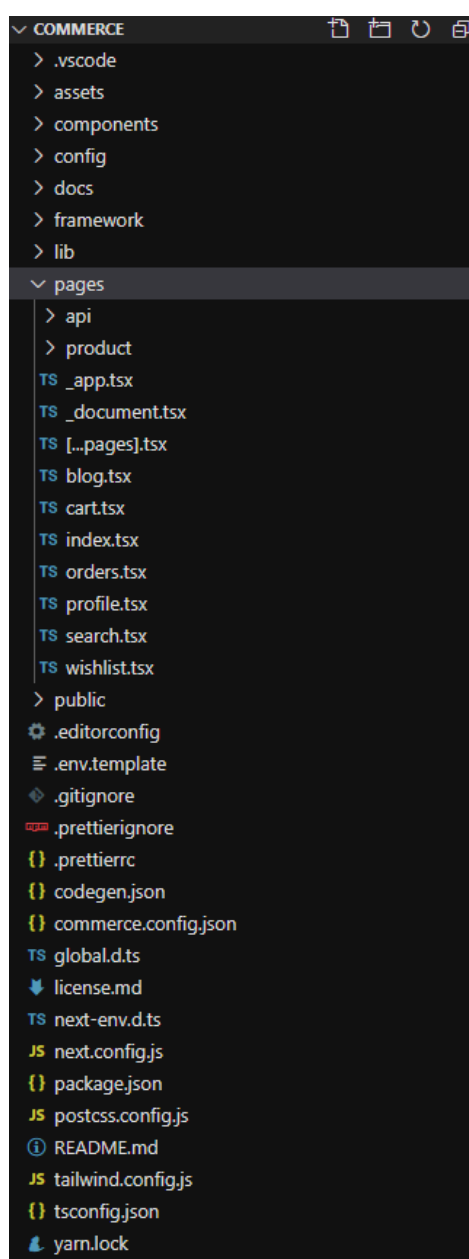


Figure 11. An example structure of a Next.js Application

In figure 11 shows a typical structure of a Next.js application where pages folder is parsed by Next.js engine to provide dynamic routing to the application without the need install additional library. Each component in “pages” folder will be redirected to the routing corresponding with the component’s name, like “pages/blog.tsx” component will take the users to path “/blog”.

The magical features of Next.js are possible thanks to the tools it is using under-the-hood like Webpack and Babel. By taking advantage of them as build tools, many optimization features are integrated out of the box. Developers that use Next.js do not need to worry about configuring Webpack and Babel since Next.js already takes care of the heavy lifting part. As such, Zero Config is the highlighted feature that Next.js proudly presents to its users. In addition, with Babel as the transpiling/compiling tool, many syntactic features from Node.js Engine V8 are supported like ES6 and async and await, allow developers to make use of the latest cutting-edge JavaScript attributes.

4.2.2 Comparison of Next.js to Gatsby.js and traditional React.js applications

Gatsby.js is also another React.js framework that focus on building static Single-Page Applications. It acts the same as Next.js as a foundation layer for a React app and offer users a clear guideline to build out the application. Compared to the traditional way to build React app with “create-react-app”, where it only provides the boilerplate and developers must choose additional libraries to handle various operations, Gatsby.js and Next.js each acts as a toolkit where it provides the full-fledge bricks and instructions for you to build a complete application.

At the time of writing this paper, Next.js has supported both to build static generating pages or pre-render pages and server side rendering as well. This in previous version of Next.js was not possible, hence, it is used to be that when developers want to build static pages, they will often choose Gatsby.js but now it is not the case anymore. However, there are still certain features that Gatsby do well compared to Next.js and will be discussed in this chapter.

The first main difference between Gatsby and Next was that how each framework handles their data exchange/fetching. With Gatsby, data access is handled through a query language named GraphQL. The benefit of GraphQL is its declarative nature where it only accepts specific data fetching, extracts only the relevant information compared to the alternative REST API where it often returns all the data, irrelevant of the information requirement. In figure 12, the requiring query is on the left, and only the specified properties define on the left are return on the right like “name” of the “country”.

```

1
2 # Welcome to GraphQL
3 #
4 # GraphQL is an in-browser tool for
5 # testing GraphQL queries.
6 #
7 # Type queries into this side of the
8 # typeaheads aware of the current Gr
9 # validation errors highlighted with
10 #
11 # GraphQL queries typically start wi
12 # with a # are ignored.
13 #
14 # An example GraphQL query might lo
15 #
16 #   {
17 #     field(arg: "value") {
18 #       subField
19 #     }
20 #   }
21 #
22 # Keyboard shortcuts:
23 #
24 #   Run Query: Ctrl-Enter (or p
25 #
26 #   Auto Complete: Ctrl-Space (or j
27 #
28 #####
29 # Default endpoint is an instance of
30 #####
31
32 query {
33   countries {
34     name
35   }
36 }
37

```

```

{
  "data": {
    "countries": [
      {
        "name": "Andorra"
      },
      {
        "name": "United Arab Emirates"
      },
      {
        "name": "Afghanistan"
      },
      {
        "name": "Antigua and Barbuda"
      },
      {
        "name": "Anguilla"
      },
      {
        "name": "Albania"
      },
      {
        "name": "Armenia"
      },
      {
        "name": "Angola"
      },
      {
        "name": "Antarctica"
      },
      {
        "name": "Argentina"
      },
      {
        "name": "American Samoa"
      },
      {
        "name": "Austria"
      },
      {

```

Figure 12. Example of a GraphQL Query and its response

On the other hand, Next.js with its flexibility to implement server-side code allows much more room to choose with how an application manage its data. It is not vendor locked in to GraphQL by default like Gatsby and results in huge gain of resilience. This also makes your application scale better, since with Gatsby if one's application requires huge amount of data, its core philosophy is static pages so there will be no extensions at runtime, leading to build time will be significantly longer with large amount of data.

In addition, Gatsby offers to its developers a very diverse ecosystem of plugins. Plugins are premade Node.js packages that allow developers to attach the already made functionality of the plugin into one's application. This in turns saves a lot of complexity and allows developers to prototype and build applications much quicker. This is not possible with Next.js application unfortunately, however this is an acceptable trade-off for Next.js when it allows much more flexibility and scalability with solutions available through either building your app in a

server-side rendering way, in a static generation way or even a hybrid approach that combine both named incremental static regeneration.

Moreover, the company behind Next.js framework, Vercel, provides one of the best Jamstack platform with seamless deployment pipeline as its core philosophy. The most impressive feature that Vercel can provide to its users was the one-click domain assigning. Everything development operation related is as simple as a single click. It utilizes the git workflow to track deployment operations and handled under the hood all the complicated deployment processes. Beside the super simple deployment pipeline, Vercel also offers analytics and user-friendly statistics so developers can focus on implementing and optimizing appropriate parts of their application. Technically considering, serverless functions are not parts of Next.js but something Vercel provides to users independently. It can be however combined into Next.js as well where users can experiment additional functionalities with their application, ranging from handling data logic, user authentication, database queries to form submission, custom slack command and more. [9]

4.2.3 Headless CMS

With Jamstack methodology, there is an innovative trending way to build an application that deliver improved development, maintenance and operational effectiveness. Jamstack also enables shortening the distance in functionality between a dynamic application and a static website, while keeping the key advantages of the static unscathed. And one of the key important factor in enabling the dynamic in Jamstack application is Headless CMS.

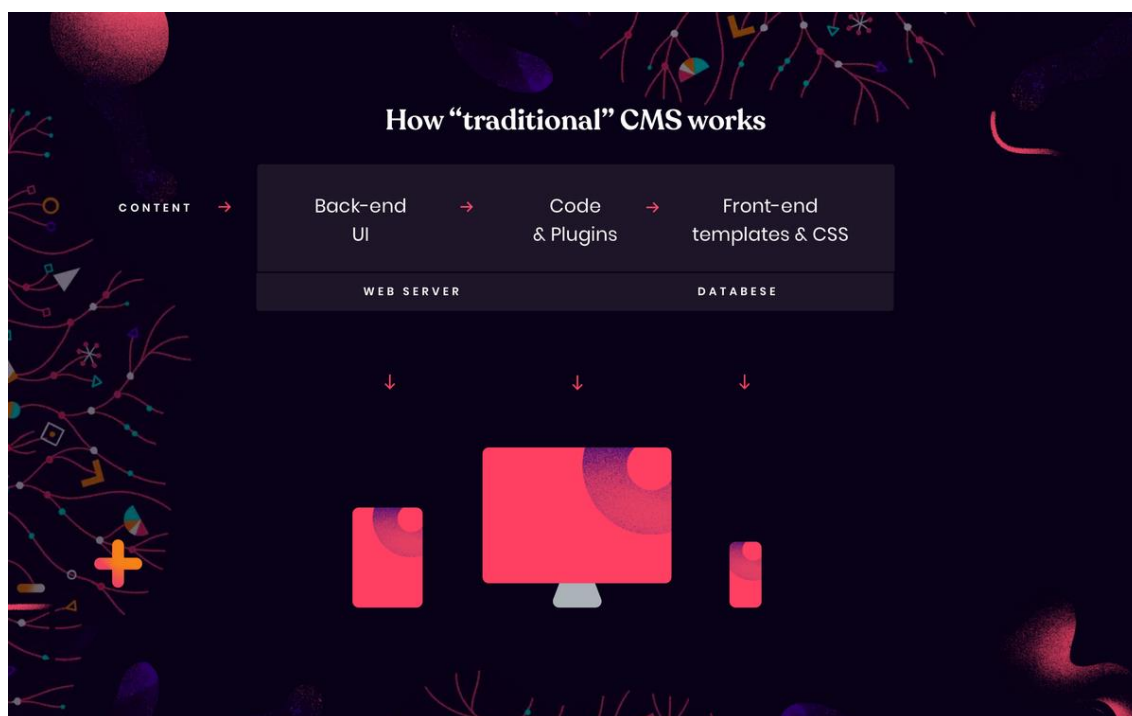


Figure 13. Operational flow of content through a traditional CMS. Retrieve from <https://bejamas.io/blog/headless-cms/>

Traditional CMS like Wordpress is bundled together with the entire stack into a single web application. It is then hosted and served together with the application whenever a page request is made. Headless CMS, instead, decouples the content layer from the presentation layer and offer them through API-driven services.

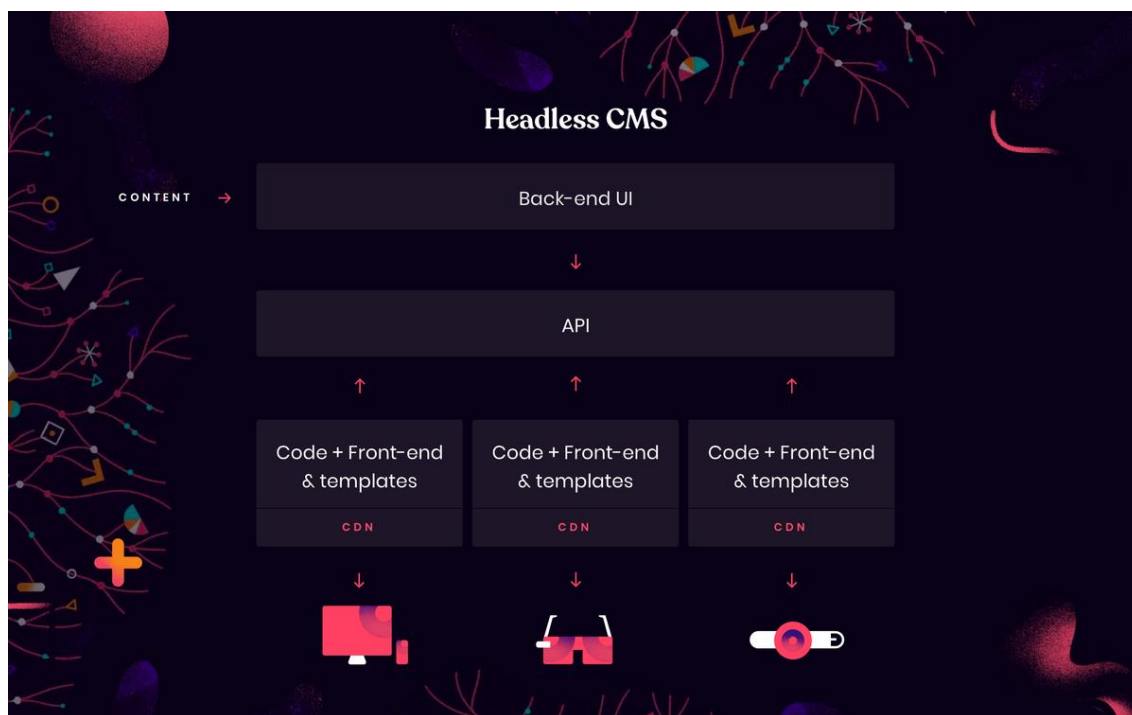


Figure 14. Operational flow of content with Headless CMS approach. Retrieve from <https://bejamas.io/blog/headless-cms/>

A headless CMS allows the content management features separate from the application's front-end and enabling content delivery through many channels beyond just pure website and apps. A headless CMS can not function on its own. It has to be built together with a site or an experience, and then takes advantages of the APIs it offers to developers to plug the content in the product.

When making changes to the application, traditional CMS must often be reimplemented completely to suit a specific part of the application. Compared to headless CMS where the content is completely decoupled from the application itself, allowing developers to make change independently to the application without affecting the content. This is huge in saving time and resources when developers do not have to deal with back-end infrastructure when they want to tweak some parts in the CMS.

Moreover, traditional CMS with its various plugins often have malicious code and bugs and the developers often must maintain those themselves if they decided to incorporate part of the plugins to the application. Meanwhile headless CMS is distributed as Software as a Service (SaaS), and since everything is delivered as external package with no maintenance required, security is also delivered as SaaS. Headless CMS also gives developer full authority on the style and theme of the content they are delivering in the application, not being limited on themes and templates like traditional CMS.

The most important aspect of using Headless CMS is its flexibility to deliver content regardless of the presentation layer. It could either be a static website, a mobile or dynamic web application, every content is delivered through APIs, not bound to a specific implementation of the headless CMSs themselves.

5 Implementation

5.1 Introduction of SY Store, a local fashion store based in Vietnam

SY Store is a fashion store based in Ho Chi Minh city, Vietnam, where its main product offered to the customers is custom-tailored clothing. It was founded in 2012 originally as an online ecommerce shop selling clothes through channels like Facebook and Instagram, and its own custom Wordpress site. Key challenges with selling products through these channels are the manual labour activities whenever there is a new collection come into sale. The managers of the store must manually update every existing product by deleting the old ones and upload new pictures and details for the new product. This could prove very labour intensive, especially through channels like Facebook and Instagram where the main method was only to upload new pictures or albums for every new product collection. And for its custom Wordpress site, the plugins used to manage the products are outdated since the managers stop hiring an external developer to maintain the site. In order to solve these challenges, the owners of the store are looking for a new way to build out a new website where it can be easily maintained through a custom content portal. This can be proven a suitable case study for Jamstack approach where content can be managed through headless CMS with one end is built as an editor for the managers, the other end is integrated APIs with the webstore application. In addition, presentation layer of the webstore can be developed independently from the content management side, allow the owners to manage the data without affecting the development of the site itself.

After taking into account key considerations, SY Store decides to quickly build out a prototype of the store with the applied Jamstack principles. Core technologies of Jamstack that the site wanted to pursue was Next.js as the full-stack framework, where data fetching can be managed through Next.js's custom data hooks and managing data from the end-user's side can be done via

BigCommerce’s user interface. In addition, Next.js with its hybrid server-side rendering and static site generation capabilities allow good SEO for an e-commerce webstore. The prototype of the webstore can be quickly built up thanks to Next.js’s starter kit name Next.js Commerce, where the foundation layer for a modern webstore is already implemented. The developers only need to study the architecture of the Next.js Commerce web application and based on that develop their own webstore with existing highly-optimized functionalities that the starter kit already provided.

5.2 Applying Jamstack to build a webstore

5.2.1 Presentation Layer with Next.js Commerce

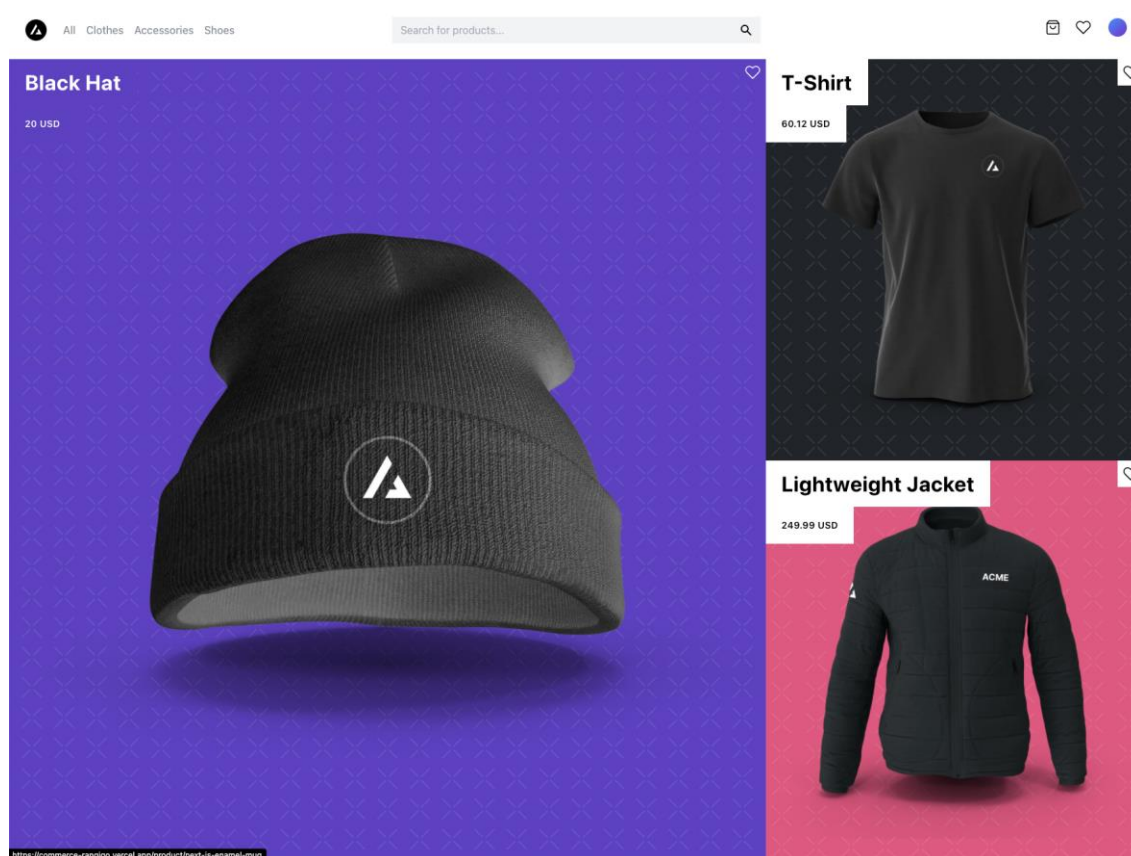


Figure 15. The homepage of webstore prototype

Upon navigating to SY Store URL <https://systore.vercel.app/>, users are welcomed to the homepage of the webstore as shown in figure 15. The page features some basic functionalities of a webstore such as Add to Cart, Favourite Product, Search Product, Filtering Product by Type, User Authentication. All UI assets of the webstore are handled in the folder “pages” and “components”, shown in figure 16.

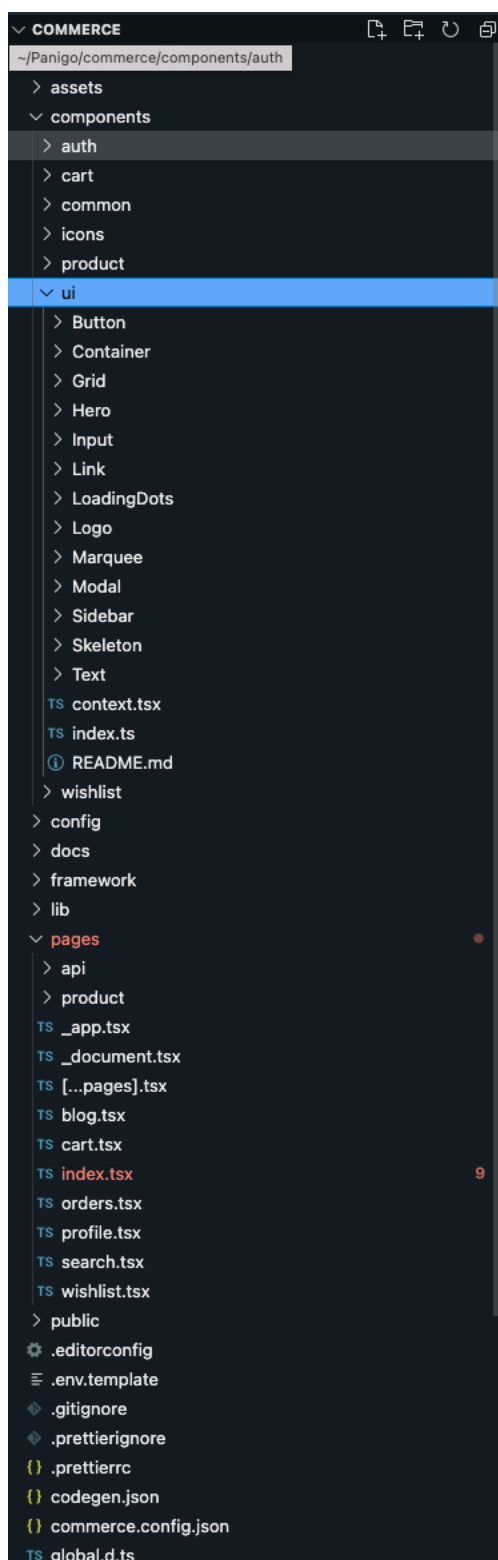


Figure 16. Directory structure of the SY Store prototype, powered by Next.js Commerce

Each component in “pages” directory corresponds to a unique path with the name of the component. For communicating with serverless functions, scripting

files within the “pages/api” folder will be the source of truth. For data handling, each Provider in the “framework” folder will be the interactive gate, shown in figure 17.

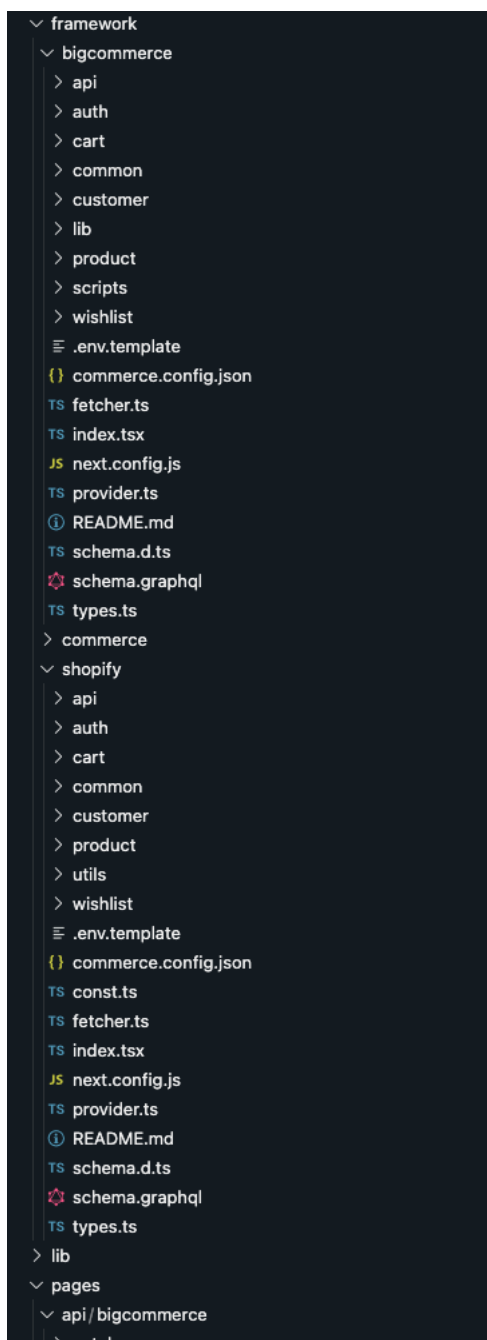


Figure 17. Each folder in the directory “framework” is a Provider

The Provider here can be interpreted as the chosen Headless CMS to handle all data exchange operations of the webstore. At the time of writing this paper, Next.js Commerce offer two examples of implementing such Provider, with BigCommerce and Shopify. In addition, if the developers want to implement custom provider, it is also possible by extending the “feature/commerce” folder,

where all the base types, helpers and functions are already implemented. As such, provider is not dependent on the UI layer of the application themselves and can be extracted into external package if needed for reuse purposes.

Let assess how the data is being fetched when user navigates to the home page, via the index component in “pages” folder.

```

import { Layout } from '@components/common'
import { Grid, Marquee, Hero } from '@components/ui'
import { ProductCard } from '@components/product'
// import HomeAllProductsGrid from '@components/common/HomeAllProductsGrid'
import type { GetStaticPropsContext, InferGetStaticPropsType } from 'next'

import { getConfig } from '@framework/api'
import getAllProducts from '@framework/product/get-all-products'
import getSiteInfo from '@framework/common/get-site-info'
import getAllPages from '@framework/common/get-all-pages'

export async function getStaticProps({
  preview,
  locale,
}: GetStaticPropsContext) {
  const config = getConfig({ locale })

  const { products } = await getAllProducts({
    variables: { first: 12 },
    config,
    preview,
  })

  const { categories, brands } = await getSiteInfo({ config, preview })
  const { pages } = await getAllPages({ config, preview })

  return {
    props: {
      products,
      categories,
      brands,
      pages,
    },
    revalidate: 14400,
  }
}

```

As the name of the function “getStaticProps” imply, the page is generated statically, with data fetching is operated at build time via the “getStaticProps” function. Methods like “getAllProduct”, “getSiteInfo” or “getAllPages” are extracted from the Provider mentioned above, in this case the chosen Provider is BigCommerce, which is enabled by a custom config file in the “framework” folder. These methods are where the data fetching happens and combines with the flexibility of the “getStaticProps” function, our Home page can access data

with ease through the props and render them dynamically as shown in the below code.

```

export default function Home({
  products,
  brands,
  categories,
}): InferGetStaticPropsType<typeof getStaticProps> {
  return (
    <>
      <Grid>
        {products.slice(0, 3).map((product, i) => (
          <ProductCard
            key={product.id}
            product={product}
            imgProps={{
              width: i === 0 ? 1080 : 540,
              height: i === 0 ? 1080 : 540,
            }}
          />
        ))}
      </Grid>
      <Marquee variant="secondary">
        {products.slice(0, 3).map((product, i) => (
          <ProductCard
            key={product.id}
            product={product}
            variant="slim"
            imgProps={{
              width: 320,
              height: 320,
            }}
          />
        ))}
      </Marquee>
      <Hero
        headline="Release Details: The Yeezy BOOST 350 V2 'Natural'"
        description="
          The Yeezy BOOST 350 V2 lineup continues to grow. We recently had the
          'Carbon' iteration, and now release details have been locked in for
          this 'Natural' joint. Revealed by Yeezy Mafia earlier this year, the
          shoe was originally called 'Abez', which translated to 'Tin' in
          Hebrew. It's now undergone a name change, and will be referred to as
          'Natural'."
      />
      <Grid layout="B">
        {products.slice(0, 3).map((product, i) => (
          <ProductCard
            key={product.id}
            product={product}
            imgProps={{
              width: i === 0 ? 1080 : 540,
              height: i === 0 ? 1080 : 540,
            }}
          />
        ))}
      </Grid>
    </>
  )
}

```

As for other pages, the same strategy is applied where data fetching happens in the “getStaticProps” function and the page component extracts and utilizes data through the props they receive.

In addition, let assess the code where data fetching is handled in the BigCommerce Provider through scripting files in the “framework/bigcommerce” directory, for example “get-all-products” hook.

```

1  import type {
2    GetAllProductsQuery,
3    GetAllProductsQueryVariables,
4  } from '../schema'
5  import type { Product } from '@commerce/types'
6  import type { RecursivePartial, RecursiveRequired } from '../api/utils/types'
7  import filterEdges from '../api/utils/filter-edges'
8  import setProductLocaleMeta from '../api/utils/set-product-locale-meta'
9  import { productConnectionFragment } from '../api/fragments/product'
10 import { BigcommerceConfig, getConfig } from '../api'
11 import { normalizeProduct } from '../lib/normalize'
12
13 export const getAllProductsQuery = /* GraphQL */ `
14   query getAllProducts(
15     $hasLocale: Boolean = false
16     $locale: String = "null"
17     $entityIds: [Int!]
18     $first: Int = 10
19     $products: Boolean = false
20     $featuredProducts: Boolean = false
21     $bestSellingProducts: Boolean = false
22     $newestProducts: Boolean = false
23   ) {
24     site {
25       products(first: $first, entityIds: $entityIds) @include(if: $products) {
26         ...productConnection
27       }
28       featuredProducts(first: $first) @include(if: $featuredProducts) {
29         ...productConnection
30       }
31       bestSellingProducts(first: $first) @include(if: $bestSellingProducts) {
32         ...productConnection
33       }
34       newestProducts(first: $first) @include(if: $newestProducts) {
35         ...productConnection
36       }
37     }
38   }
39
40   ${productConnectionFragment}
41 `

```

Due to BigCommerce API is written in GraphQL, the interactive code here that is used to communicate with BigCommerce must include GraphQL Query Language. The query has multiple dynamic variables, all to show that code written

to handle data in Next.js can be flexible with the way it is used to control data exchange operations.

```

89  async function getAllProducts({
90    query = getAllProductsQuery,
91    variables: { field = 'products', ...vars } = {},
92    config,
93  }): {
94    query?: string
95    variables?: ProductVariables
96    config?: BigcommerceConfig
97    preview?: boolean
98    // TODO: fix the product type here
99  } = {}): Promise<{ products: Product[] | any[] }> {
100    config = getConfig(config)
101
102    const locale = vars.locale || config.locale
103    const variables: GetAllProductsQueryVariables = {
104      ...vars,
105      locale,
106      hasLocale: !!locale,
107    }
108
109    if (!FIELDS.includes(field)) {
110      throw new Error(
111        `The field variable has to match one of ${FIELDS.join(', ')}`
112      )
113    }
114
115    variables[field] = true
116
117    // RecursivePartial forces the method to check for every prop in the data, which is
118    // required in case there's a custom `query`
119    const { data } = await config.fetch<RecursivePartial<GetAllProductsQuery>>({
120      query,
121      { variables }
122    })
123    const edges = data.site?.[field]?.edges
124    const products = filterEdges(edges as RecursiveRequired<typeof edges>)
125
126    if (locale && config.applyLocale) {
127      products.forEach((product: RecursivePartial<ProductEdge>) => {
128        if (product.node) setProductLocaleMeta(product.node)
129      })
130    }
131
132    return { products: products.map(({ node }) => normalizeProduct(node as any)) }
133  }
134
135  export default getAllProducts

```

In the exported `getAllProducts` function, the query and its dynamic variables are being called in a custom “`config.fetch()`” method, where it is written to handle GraphQL queries. The result return from the method was the data we are expecting to operate with. The shape of the data can be complicated, that is why the next several steps is to filter out and map the returned data to the correct “product” format that the function is expecting to return in the final step. Thus through this custom hook to handle data fetching, it proves that data operations

can be handled with ease and customized to one's satisfaction with Next.js application's structure.

Moreover, as Next.js pre-rendering or statically rendering all pages by default, better performance and SEO are guaranteed and are often the expected specifications for ecommerce applications. These pages can be cached by CDN as well with no extra configuration to ensure better performance, thus embrace the Jamstack philosophy by taking advantages of CDN to the best instance.

5.2.2 Content Management with BigCommerce's User Interface

In this section, we will inspect how the owners of SY Store or data editors can work with the content without any development knowledge about the presentation layer of the webstore. This is possible thanks to the BigCommerce CMS allow on the one hand developers accessing to data with ease through their APIs, on the other hand allow end-users to manage content through their full-fledge user interface.

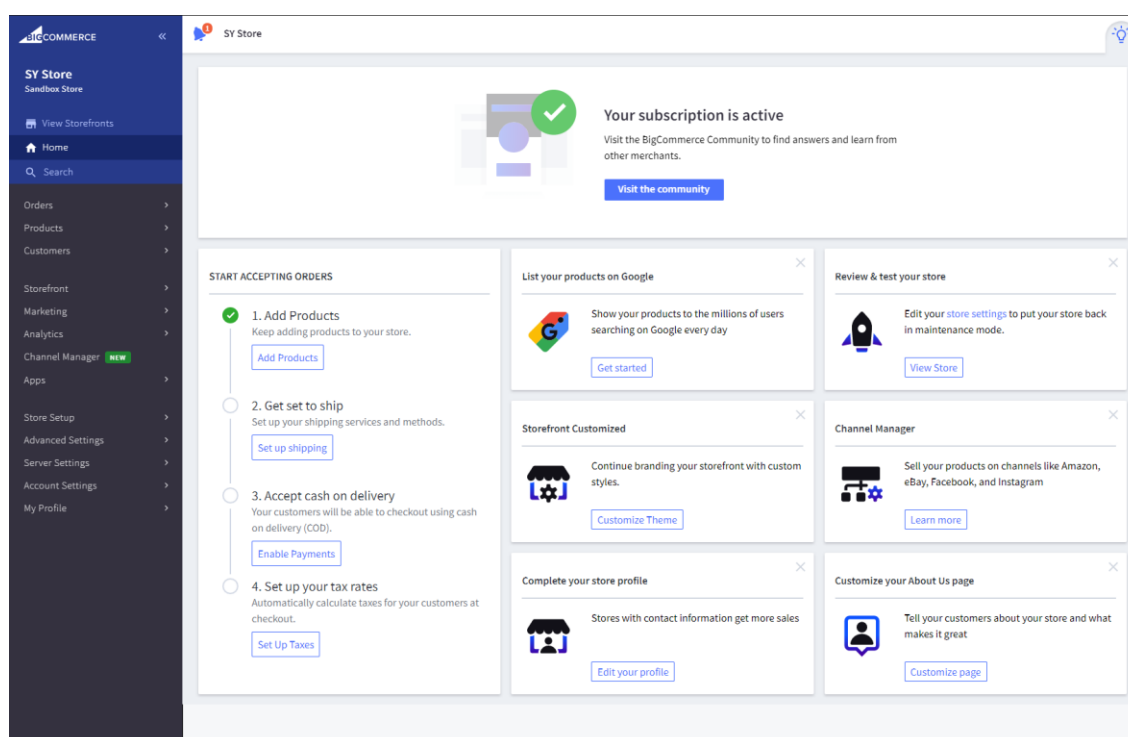


Figure 18. User Interface of BigCommerce CMS

Upon logging in the BigCommerce Editor Portal, users are greeted with the homepage and the left navigation pane where multiple content integrated features are accessible, demonstrated in figure 18. If the users want to update Products that are shown in the webstore, it can be done via the Products tab where all the existing products are displayed in the UI, shown in figure 19.

The screenshot shows the BigCommerce CMS 'Products' view. On the left is a dark sidebar with navigation links: 'View Storefronts', 'Home', 'Search', 'Products', 'View', 'Add', 'Search', 'Import', 'Export', 'Product Categories', 'Product Options', 'Product Filtering', 'Product Reviews', 'Price Lists', 'Brands', 'Import Product SKUs', and 'Export Product SKUs'. At the bottom of the sidebar is a 'Help' section with 'Support PIN: 48531308'. The main content area is titled 'Products' and features a search bar, 'Advanced Search', 'Expert Products', and 'All Products' filters. Below this is a table of 22 products, each with a checkbox, a product image, name, 'Shop All' link, SKU, Stock, Price, and Assigned to field.


















| <input type="checkbox"/> | | SKU | Stock | Price | Assigned to |
|--------------------------|---|------------------------------|-------|-------|-----------------|
| <input type="checkbox"/> |  Long Sleeve Shirt Shop All | 202967M205017 12 variants | N/A | 25VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Beanie Shop All | 202129M138042 | N/A | 22VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Sweatshirt Shop All | 202129M204032 13 variants | N/A | 28VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Piped Fleece Jacket Shop All | N/A 5 variants | N/A | 50VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Unisex Long Sleeve Tee Shop All | 202695M213084 9 variants | N/A | 26VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Desert T-shirt Shop All | 202695M213081 5 variants | N/A | 25VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Short-Sleeve Unisex T-Shirt Shop All | 202695M21302 6 variants | N/A | 21VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Unisex Fleece Hoodie Shop All | 202692M213081 8 variants | N/A | 42VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Unisex Joggers Shop All | 202695M2130D1 5 variants | N/A | 34VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Cotton T-shirt dress Shop All | 202295M213081 8 variants | N/A | 34VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Unisex Skinny Joggers Shop All | 2025M2213081 8 variants | N/A | 44VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Next.js Joggers Shop All | 202215M213081 7 variants | N/A | 54VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Short-Sleeve T-Shirt Shop All | 202695X213081 6 variants | N/A | 14VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Embroidered Champion Packable Jacket Shop All | 202695MD13081 6 variants | N/A | 52VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Letterman Jacket Shop All | 20269DM213081 5 variants | N/A | 54VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Next.js Short sleeve t-shirt Shop All | 202695M513081 12 variants | N/A | 26VND | 1 Channel ▾ ... |
| <input type="checkbox"/> |  Bomber Jacket Shop All | 2695M2130A81 | | | |

Figure 19. Product view in the BigCommerce CMS UI.

Through this view, multiple features relating to update or adding or deleting Product are supported. Content editors can click on each Product and update them with thorough details, ranging from pictures, categories, pricing, product description to product type, product code.

The screenshot displays the 'Product Information' page for a 'Long Sleeve Shirt' in the BigCommerce CMS. The interface includes a left-hand navigation menu with categories such as 'PRODUCT INFORMATION', 'PRODUCT OPTIONS', 'STOREFRONT', 'FULFILLMENT', and 'SEO & SHARING'. The main content area is titled 'Product Information' and contains a 'Basic Information' section with the following fields:

- Visible on Storefront
- Product Name: Long Sleeve Shirt
- SKU: 202967M205017
- Product Type: Physical
- Default Price (excluding tax): 25 VND
- Brand: Common Good
- Weight: 7.90 Ounces

Below these fields is a 'Categories' section with a '+ Add category' link and a list of categories: Shop All (checked), Bath, Garden, Kitchen, Publications, and Utility. At the bottom, there is a 'Description' section with a rich text editor toolbar and a text area containing the following text:

ITEM INFO
Long sleeve organic cotton jersey T-shirt in black. Logo embroidered in white at rib knit mock neck collar and cuffs. Textile logo

At the bottom right of the page, there are 'Cancel' and 'Save' buttons.

Figure 20. Product details view in the BigCommerce CMS UI

Beside managing content related to Product, editors can also access to customer's data through the Customer tab on the left navigation pane, shown in figure 21. From this view, editors can manage all the relevant information about customer, from their username, email, phone number to store credits, even changing the customer's password is possible with this feature.

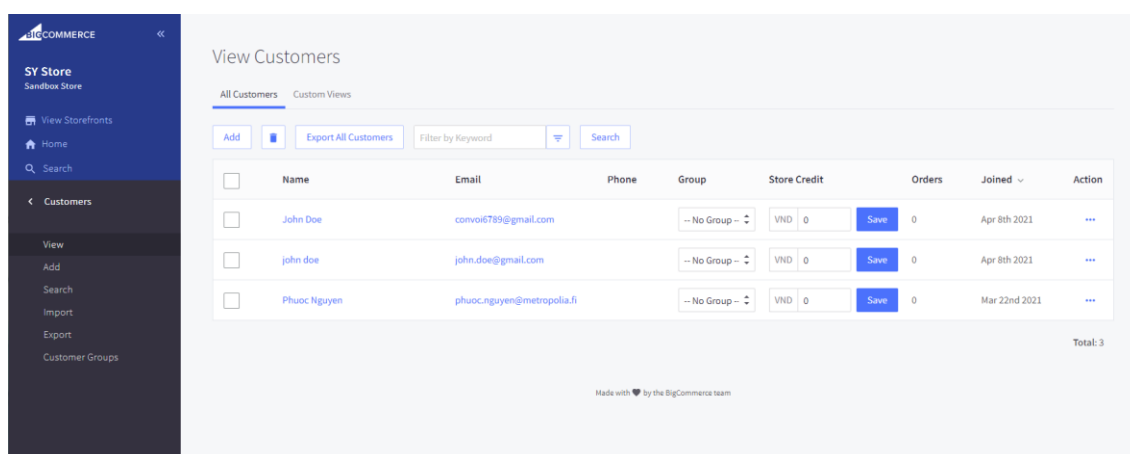


Figure 21. Customer view in the BigCommerce CMS UI

Thus, with BigCommerce CMS UI, data editors or the owners of SY Store can access and manage their data and content with ease. As they update the information on the CMS UI, relevant changes will be available immediately on the presentation layer or the store front of the webstore through the APIs. This in turns give developers a lot of power to quickly iterate out awesome user experiences without worrying messing up the data and store owners can confidently manage their content and not affecting the development progress of the webstore.

6 Conclusion

To summarize, many technical details are discussed and analysed in this paper, ranging from the origins of JavaScript and how it matures to become an important part of developing Single Page Applications, to the various tooling that the JavaScript ecosystem offer to its developers. In addition, various methods to produce a modern Single Page Application are also examined to distinguish the pros and cons of each method, and what is the most suitable way to build out the application depending on the requirements. Moreover, a rising new architecture are also discussed in the paper, namely Jamstack, where it offers many advantages over the traditional way to build out web application. It revolutionizes the front-end architecture and utilizes the power of CDN and its caching capability to provide fast and secure web application. Finally, the practical implementation of all the analysed technical details is examined and satisfy all the requirements for an ecommerce prototype that the owners of SY Store is looking for. Next.js is the framework being used to develop the prototype, and BigCommerce is the chosen headless CMS to control data flow and operations. Due to the scope of the thesis, only basic functionalities are implemented in the prototype, however the codebase is extended through the Next.js Commerce starter kit, allow for future development to be easily accomplished.

Thus, with the Jamstack methodology, and JavaScript becomes more and more developed, future of Single Page Applications is being more relevant than ever. With Jamstack advocates for simplifying the stack used to develop web applications, fast, light and secure website will become the norm when it comes to creating a modern JavaScript SPA. And with that is the rising API economy, where all the complex services are already implemented by third party developers, leading to the owners of Jamstack application can focus on delivering the best user experiences there are, without scarifying the cost of maintaining the external complicated services in their application.

7 References

1. JavaScript for impatient programmers (ES2020 edition) [online] <https://exploringjs.com/impatient-js/toc.html> [Access 1st March 2021]
2. How JavaScript was created [online] <https://brendaneich.com/2008/04/popularity> [Access 1st March 2021]
3. Interview with Brendan Eich, “A-Z programming languages: JavaScript” [online] <https://www2.computerworld.com.au/article/255293/a-z-programming-languages-javascript/> [Access 1st March 2021]
4. Paul Krill, “JavaScript Creator Ponders Past, Future,” InfoWorld, June 23, 2008, [online] <http://bit.ly/1KlpXO>; Brendan Eich, “A Brief History of JavaScript,” July 21, 2010, [online] <http://bit.ly/1KklIOM>. [Access 6th March 2021]
5. A complete guide to JavaScript tooling [online] <https://ageek.dev/javascript-tooling-overview> [Access 6th March 2021]
6. Babel.js [online] <https://babeljs.io/docs/en/index.html> [Access 14th March 2021]
7. Rendering on the Web [online] <https://developers.google.com/web/updates/2019/02/rendering-on-the-web> [Access 18th March 2021]
8. Mathias Biilmann & Phil Hawksworth, “Modern Web Development on the JAMstack”, O’Reilly Media, Inc.
9. Vercel Serverless Functions [online] <https://vercel.com/docs/serverless-functions/introduction> [Access 14th April 2021]
10. Next.js Commerce [online] <https://nextjs.org/commerce> [Access 19th April 2021]

