

Hieu Nguyen Dinh

Front-end structure for a Finnish cosmetics online shop

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Thesis

27 December 2018

Author Title	Hieu Nguyen Dinh Front-end structure for a Finnish cosmetics online shop
Number of Pages Date	50 pages 6 March 2019
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Professional Major	Mobile Solutions
Instructors	Patrick Ausderau (Principal lecturer)
<p>The purpose of the project was to construct a standalone front-end for a Finnish cosmetics web shop. This would be part of their migration from a custom-made framework to a new framework called Litium, to enable a more streamlined e-commerce development experience and to scale the shop smoothly in the future.</p> <p>The project was completed by observing established methodologies and approaches to frontend frameworks before devising a custom implementation. As a result, the site's frontend was built separately from the backend. It includes an independent static markup section to imitate the production site, with reactive file processing and optimization using Node.js and Gulp.js for a smooth development time. On top of that, the frontend sector also expands into a pattern library, which is to ensure a common voice between the design team and engineering team.</p> <p>In conclusion, the project was a success fulfilling the needs from both the development team with its frontend processing and the design team with the pattern library. There is still room for improvement, such as a more consistent naming scheme to cover unexpected cases or a better method to organize the global and local scripts. However, they will be assigned to a post-launch project, after the migration is complete.</p>	
Keywords	HTML, SCSS, Gulp.js, frontend, pattern library, Zurb Foundation, Litium

Table of Contents

1	<i>Introduction</i>	1
2	<i>Theoretical Background</i>	2
2.1	Client company	2
2.1.1	Client company background	2
2.1.2	Client company requirements	2
2.2	Frameworks	3
2.2.1	Introduction to web frameworks	3
2.2.2	Litium framework	3
2.2.3	Node.js and Gulp.js	5
2.2.4	SASS	6
2.2.5	jQuery & Litium scripting model	8
2.2.6	Astrum and VueJS	10
2.3	Pattern Library	10
3	<i>Implementation</i>	12
3.1	HTML structure	12
3.2	Styling structure	15
3.3	Methodology	20
3.3.1	BEM	21
3.3.2	ITCSS	23
3.3.3	SMACSS	24
3.3.4	SUITCSS	25
3.4	Custom implementation	26
3.4.1	CSS custom implementation	26
3.4.2	JS custom implementation	29
3.4.3	Asset management	31
3.5	Pattern library	32
3.5.1	Tooling	33
3.5.2	Implementation	35
4	<i>Result & Discussion</i>	37
4.1	Result	37

4.2	Discussion	38
4.2.1	Structure	38
4.2.2	Task automation	40
4.2.3	Asset optimization	42
4.2.4	Styling methodology	43
4.2.5	Pattern library	44
5	Conclusion	46
	References	47

List of Abbreviations

AJAX	Asynchronous JavaScript And XML.
API	Application Programming Interface.
BEM	Block – Element – Modifier.
CDN	Content Delivery Network
CLI	Command Line Interface
CSS	Cascading Style Sheets.
DOM	Document Object Model
ERP	Enterprise Resource Planning.
HTML	Hypertext Markup Language.
IIS	Internet Information Server.
JS	JavaScript.
JSON	JavaScript Object Notation.
KSS	Knyle Style Sheet.
MVC	Model – View – Controller.
PIM	Product Information Management.
SASS	Syntactically Awesome Style Sheets.
SCSS	Sassy CSS.
SEO	Search Engine Optimization.
SMACSS	Scalable and Modular Architecture for CSS.
SQL	Structured Query Language.
SUITCSS	Styled User Interface Tools CSS.
UI	User Interface.
XML	Extensible Markup Language.

1 Introduction

For most web projects present in the current day, whether simple or complex, there needs to be a certain structure during its development. Such an organization ensures the work is done in a professional way, saving time and efforts for the both the client and the contractor. For a functional website, the most necessary technologies are Hyper Text Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript (JS). However, as the site grows more complex, there is a need for solid architecture to keep it scalable and sustainable. The thesis presents the front-end structure and the underlying principles for Web Shop A, which has been one of the leading forces in e-commerce in Finland. Earlier in 2018, Web Shop A — or from this point on referred to as Company A — decided to adopt a new platform for their shopping business. As the previous backend had to be migrated, so did the frontend, hence the whole process of its restructuring.

Even though the world of web development is constantly expanding, the idea stays the same. Every web project frontend should have all three core technologies — HTML, CSS and JS — built up logically and consistently. Not only does the frontend have to be able to follow the design, it also must stay logical to the developers so that they can collaborate with less time and efforts.

The objective of this thesis was to describe the front-end construction process, that is, the directory structure, methodology during development and build process. Additionally, to satisfy Company A's wish for a design archive and seamless future design implementation, a pattern library is also featured.

2 Theoretical Background

2.1 Client company

2.1.1 Client company background

As business moves onto the Internet more than ever, there is a strong need for e-commerce. Company A is one of the leading forces in this field, starting out from the 90s in Finland and since then has been expanding steadily to other markets. The company has its focus on cosmetics and hygiene products such as hand lotion, body cream and showering liquid. It was awarded the title of the best web shop in Finland during 2013 and currently has its market shares in several European countries, including Finland, Sweden, Russia and Estonia.

Since its establishment, Company A has been using its own custom-made platform for 15 years (and still counting at the time of this writing). However, as new business strategic decisions were made, the company decided to migrate from its current platform to a more streamlined and scalable e-commerce platform called Litium.

The migration process started officially in the spring of 2018 and is still underway. It was decided that most of the current platform, both backend and frontend, had to be rewritten in the new platform.

2.1.2 Client company requirements

As one of the leading online cosmetic shops in Finland, Company A has many existing and upcoming clients. On top of that, since it has acquired more business in other countries, there is a growing need for customization for both visual and content aspects to match the market situation.

Traditionally, any new feature to the site, whether a new section or a simple functionality, needs to be approved by the marketing team from Company A before the designers can start working on it. After another round of approval, the feature is transferred to the development team and its display on the web has to be reviewed again before it is implemented.

Company A would usually have multiple requests at the same time, with each taking different amount of time to be assessed. Therefore, it is necessary that the frontend — which creates the static appearance — is separated from the backend. Not only will this faster the reviewing process but also help the development not so reliant on either side to operate. This way, the frontend can go closely with the backend, and at the same time, be independent of its logic. With that in mind, even though the frontend side will still be included as part of the project repository, it is a standalone sector that can operate by itself without any data from the backend.

2.2 Frameworks

2.2.1 Introduction to web frameworks

The most basic blocks of all functional websites include HTML, CSS and JS. HTML is a computer language that fabricates the essential content and structure of the site and is “the World Wide Web’s core markup language” [1]. It is the most essential piece of a page, without which the site would be invalid and unreadable. Traditionally, HTML alone is enough for a webpage, but as the technology evolves, CSS (and later, JavaScript) proves to be equally indispensable. CSS describes how the webpage appears [2]. Lastly, JavaScript holds the logic for the site, adding the possibility of interactive components [3]. One can imagine HTML as the skeleton, CSS as the skin and JavaScript as the mind of a website. Together, those three make a complete website with meaningful layouts and engagements.

For projects that requires a complicated structure and scales across several regions, even though HTML, CSS and JS still are indispensable, they need to be planned in such a way that adding new features or replacing old ones causes as little difficulty as possible. This consequently involves several web frameworks; whose core logics already cover numerous basic functionalities. Hence, developers only need to extend from those without writing from scratch. A web framework can be understood as a foundation upon which developers can build and expand into certain specifications [4].

2.2.2 Litium framework

As the whole platform migration is executed on Litium, this framework plays the major role in enabling all the required functionalities of the web shop. To give a brief introduction, Litium is an e-commerce platform built by Litium AB — a Swedish company, running on ASP.NET Model – View – Controller (MVC) framework. Structurally, Litium is simple: it operates as a ASP.NET web application, runs on Internet Information Server (IIS) and keeps data in a standard Microsoft Structured Query Language (SQL) Server database by writing files on disc [5].

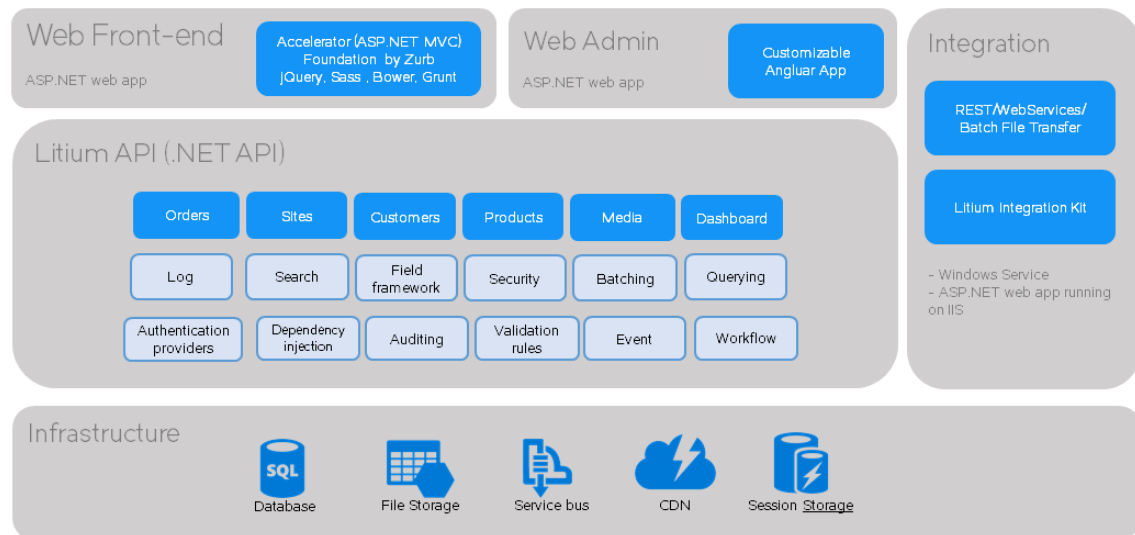


Figure 1. Litium framework three-tier architecture. Copied from Litium Documentation website (2018) [5].

As can be seen from Figure 1, Litium provides a wide range of functionalities for the business through its Application Programming Interface (API), including dashboard management, web content control, product management, sales and customer management and media handling. All these business logics are seamlessly integrated along with Integration Kit, which enables fast setup and problem-free migration to their Enterprise Resource Planning (ERP) and Product Information Management (PIM) systems.

Since Litium framework is a very complex system, this thesis only covers its front-end modules, which is structured into two main parts: the Web-Admin interface and the Web interface. The former is used to control the dashboard and behind-the-scene content of the web shop. It is built using Angular 1 framework and compiled into a complete application using Webpack. This section is supposed not to be modified by the client and only developed by the Litium team themselves. However, the latter component — the Web interface — can be orchestrated to the client's requests and is the primary working ground of the front-end team.

This web interface is built using Litium Accelerator, which is a pre-packaged application, shipped with a full-fledged front-end solution including markup, styling and script processing. The markup is made using ASP.NET MVC, that is, the Razor views render dynamic content coming from their respective controllers after accessing data from the models. As for styling, Litium uses Foundation 5 framework by Zurb. Lastly, front-end scripting follows Foundation JavaScript model with various modules being initialized separately, built using pure JavaScript with the assistance from jQuery. For task automation, Litium uses task runner Grunt.js to compile all the markup, styling and scripts into desired output.

Overall, Litium has developed an exhaustive package for front-end purposes. In the following sections, the frameworks and the necessary changes to meet the needs of development will be explained.

2.2.3 Node.js and Gulp.js

There is a series of repetitive tasks to be done in most web development projects, such as generating the entire HTML pages from templates, removing unnecessary scripts and compiling them, etc. While individually, each task can be done within seconds but altogether, they will create a huge amount of time strain. Therefore, it is necessary to introduce a utility tool that will automate those minor processes for the developers. After trials and errors, GulpJS running within Node.js was the choice for this project.

To quote from its homepage, Node.js is “a JavaScript runtime built on Chrome's V8 JavaScript engine” [6]. The creator of Node.js modifies the engine of Google Chrome browser so that it can have OS utilities and HTTP library amongst other useful functions. This results in an innovating use of Node.js — to allow JavaScript to run outside of the normal web browser environment. Previously, JavaScript is primarily used in websites, but Node.js enables it to be used by a much wider selection of applications, such as server-side. Regarding Company A's website development, Node.js allows the development team to run Gulp.js.

Gulp.js is a powerful automation tool that can perform repetitive manual tasks, mainly for front-end development [7]. Any single task in Gulp consists of three core elements: the source, the tasks and the output. The `gulp.src()` function marks the source, with its

argument being a file-matching string so Gulp can find those files. Following it is a series of tasks coming from the defined plugins at the top of the file, such as `newer`, `concat` and `prefixer`. Finally `gulp.dest()`, which also takes a file-matching string, decides where to generate the processed result. It can be seen that the Gulp system itself has little power here, but it is the collection of plugins that enable it to complete the task.

Additionally, Gulp also has `gulp.watch()` to allow actions upon file changes. Developers can create a custom task that include multiple `gulp.watch()` to react to changes in different types of files.

```
gulp.task('watch', function () {
  gulp.watch(['UI/**/*.{gif,png,svg,jpg}'], ['resources']);
  gulp.watch(['UI/html/**/*.html'], ['html']);
});
```

Listing 1. Gulp.js watch task for on-change reactions.

In the task `watch` shown in Listing 1, there are directives to detect changes in image file, for example, `gif` or `jpg`, which will trigger the task `resources`, and in HTML files, which will trigger the task `html`. Combining this powerful feature with explicit tasks, the development team created a reactive automation system that would react upon relevant files and update the browser accordingly.

2.2.4 SASS

CSS is a stylesheet language used to define the display of an HTML or eXtensible Markup Language (XML) document on different media including screen, paper and speech [8]. It is one of the core components in the open web as it enables plain-text document to become an interactive and decorated one. For example, Figure 2 demonstrates the same website with and without the use of CSS:

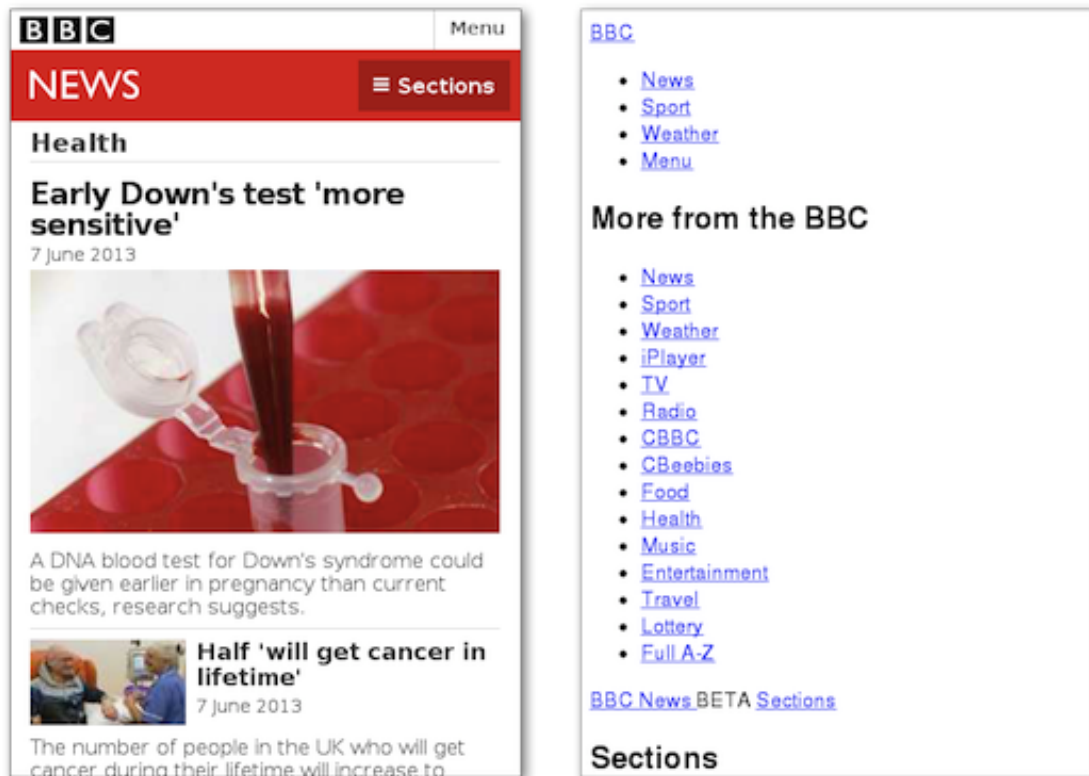


Figure 2. BBC News website before and after removing CSS from Capturing Web Page without Stylesheet (2013) [9].

There is a vast difference between a page with the aid of CSS and the one completely without. CSS can modify the appearance of HTML properties such as headlines, paragraphs and anchor links. This can be achieved by referencing the target, either by its tag, for example, `h1`, `h2`, `p`, `a`, or calling its selector, for example, giving a headline the class name of `headline` and later use `.headline`, then insert styling properties inside. By specifying the look of necessary elements in a page, a web developer can improve its visual appeal. Although it is possible to have a website with purely content, either HTML or XHTML, there is hardly any web page nowadays without some CSS styling.

Syntactically Awesome Style Sheets (SASS) is an extension to the CSS language, with more functionality and flexibility [10]. This language cannot be processed by the browser naturally, although it is possible to compile it into CSS with a proper software. Since its introduction in 2006, SASS has introduced an increasing number of features to boost the functionality of CSS. In fact, it is still under active development, with its latest release in March 2018. With SASS, it is possible to define variables, allowing for the reusability of

property values that CSS alone could not afford yet. Another interesting feature is `partial`, which allows developers to choose whether to output SASS files to CSS or simply keep it as a form of reference.

The development of SASS also introduced a syntax variation called “Sassy CSS”, abbreviated as SCSS [11]. Since it is a different syntax, it also requires a different processor and file extension to work properly. SCSS includes the entirety of SASS capabilities but has gained developer's favor over the years due to its similarity with native CSS. In fact, the official documentation for SASS uses SCSS syntax, instead of the classic indentation syntax, illustrated in Figure 3.

```
// CSS syntax
.class1 {
  color: red;
}

// SCSS syntax
.class1 {
  color: red;
}

// SASS syntax
.class1
  color: red
```

Figure 3. Difference between CSS, SCSS and SASS syntax

The lack of parenthesis and semi-colons in SASS often creates a sense of inconsistency when the developers switch back and forth between CSS and SASS. For instance, when using 3rd-party plugin with basic CSS, they may have an awkward experience when coming back to writing their own styling with SASS. SCSS removes this issue since it shares the same syntax with CSS, yet still retains all the important functionalities. The current project, thus, is implemented with SASS, but follows the syntax of SCSS.

2.2.5 jQuery & Litium scripting model

As HTML declares the structure of a webpage and CSS dictates its visuals, JavaScript defines the behaviors. Most of interactions available on a website, from a simple tooltip popping up to sending a registration form to the server, is the work of JavaScript. JS is a “lightweight, interpreted, object-oriented language with first-class functions”. Developed in 1995, JS has been improved yearly with its most recent draft specification being due to be released in 2019 by the ECMA International standard organization [3]. It is

used most commonly in web sites, and recently has found its way into external environments. Since the outcome of the project is needs to offer appropriate online shopping experience, it will expect multiple actions from users that return meaningful responses to them. Therefore, JS and its structuring also play a huge role in this project completion.

Although JS itself is quite powerful, the development team chose to adopt jQuery library to further its abilities and ensure a smooth development experience. jQuery is “a fast, small, and feature-rich JavaScript library”, specializing in several important cases such as Document Object Model (DOM) traversal and manipulation, event management, Asynchronous JavaScript And XML (AJAX), etc. [12]. Additionally, the use of jQuery is unavoidable since Litium integrated Zurb Foundation library into their framework, which requires jQuery to function properly.

With the help of jQuery and Zurb Foundation, Litium framework also has its own model of scripting that would ensure a logical and robust development. To be more specific, instead of combining all scripts into one big file, Litium separate them into modules based on their function [13]. Following a similar structure as Zurb Foundation, this enables concern isolation, that is, each module only does its job well and disregard other modules responsibility. In the end, there is a “master” initiator that starts the whole module system, as well as handles events between each module to ensure their communication with each other. For instance, it allows clicking on a button to add a product to cart to trigger a tooltip displaying “Product added to cart”.

While this scripting model from Litium does an excellent job at managing inter-page interactions, there is still a need for smaller actions that are also scoped to only several pages. For example, a unique menu effect which should only appear in a certain page should not require a whole new module. Instead, scripts such as those in Listing 2, are inserted directly into the static markup and later in the Razor files.

```
@section scripts {
  <script>
    $('<code>.l-catalogDetail</code>')
      .on('click', '<code>.product_wishlist a</code>', function (e) {
        // special effect for the wishlist menu
      });
  </script>
}
```

Listing 2. Function example that is applied to only certain pages.

As the script only runs in the catalog detail page (`.l-catalogDetail`) to trigger some special effects for the wishlist menu, it would be excessive to include this function in the

main JS file, which is used on all pages. Therefore, this script is rendered at the end of the catalog detail page, so it is only triggered when users visit this pag.

2.2.6 Astrum and VueJS

A decade ago, web development was much simpler with mostly HTML, CSS and JS to produce functional web sites. However, as the web technologies evolve increasingly fast today, those web sites also become more complex. Hence, raises the need for a tool to document styles and help organize intricate interfaces. With that in mind, the development and design team agreed to collaborate on a pattern library that would not only archive implemented elements, but also ensure future features will follow the same principles.

In fact, there have been many attempts to create a pattern library for a website brand. Consequently, there are also a wide variety of tools and methods to build such a system. After many trials, it was determined that Astrum — “a lightweight pattern library for any web project” — should be the pattern generator for this project [14].

Built upon VueJS framework, Astrum allows for standard HTML in the markup then uses the same CSS file as the production page without much hassle. Additionally, Astrum comes with its own Command Line Interface (CLI) to even faster the component management process. This is not to mention the core framework of Astrum is also alterable thanks to VueJS to satisfy the need of the developers should some custom features are required.

2.3 Pattern Library

As web development goes, a normal project involves all product, design and development teams. As each team exercise a different approach to task delivery and completion, it becomes hard to manage the process. Still, the common goal remains the same: to keep the user interface as consistent as possible to avoid confusion for customers. Otherwise the users will struggle to get familiarized with the brand, leading to overall reduction in sales. To keep the persistency, the engineer team may have code review sessions and unit tests to ensure the developers stay in sync. The design team, however, uses some form of style guide as a reference for future iterations. While those methods prove to be effective within a specific group, there is hardly a common voice between the entire

team. Thus, the idea of a shared component repository — a style guide — is born in the hope to help the unify the design language. [15]

A style guide covers the entirety of every design aspect of a project, for example, how the brand presents itself, how each component looks like, and how the motion should appear [16]. As useful as it may seem, the team realized it might be too ambitious a project to complete. Therefore, only one major part of the style guide — the pattern library — was chosen to be built. The pattern library should hold every repeating element such as buttons, form inputs and navigation, as well as abstract elements that can be present site-wide such as color and typography. A well-constructed library can help the designers to have an archive for their future ideas, at the same time, ensure a logical overview for the engineering team.

3 Implementation

There has been a dire need for a logical and comprehensible file structure so that the development can proceed smoothly. As the question to how such a good system can be constructed, the development team had to satisfy several requirements such as:

- The web shop spans multiple pages and multiple regions
- The final product needs to be maintainable
- The final products need to be scalable
- The project should be developer-friendly

With those needs in mind, the project was divided into three main categories, just like the three core components of a website: markup (for HTML), styling (for CSS) and scripts (for JS).

This way, new developers can start working on separate parts without worrying about residual changes. Additionally, there should be a folder for other resources such as images, fonts and icons. They are positioned at the same level as the three major component group. The folder for the pattern library is also at the top level, as it is a standalone application and should not affect the development process. Finally, the outcome files and folders are generated and put in the distribution folder called `dist`.

3.1 HTML structure

Just like a newspaper or magazine, a website also needs a clearly defined content structure. HTML is the most common tools for this purpose, being able to utilize so-called `tags` to mark different elements in one page, from a small span of text, a small paragraph or even a whole section. HTML is not considered a programming language, but a computer language only since there is little programming involved in the writing process. Nevertheless, it still needs a certain extent of structuring for Search Engine Optimization (SEO) purposes and readability.

For every page in a website, there is a respective complete HTML document to match its structure. Even though several pages may have repeated elements, notably the header and footer section, it is still essential to include those elements again in every HTML markup. Thus, this creates a strain for static frontend development as the developers have to insert those repeated elements every time a new page is created, not to mention smaller elements like a product card. Fortunately, it does not pose a threat to

markup in the backend since each recyclable element can be a separated file and just needs to be `imported` should it be required. This concept of “importing” has inspired the front-end development team to modularize the HTML markup so any shared component can be quickly included in the document.

This idea has many similarities with Web Component, a technology just recently developed to allow reusing custom elements for web applications [17]. It has proved itself as a very potent application that enables compact and recyclable modules for large-scaled development, especially without the assistance from external JavaScript frameworks. The front-end team considered adopting this, but since it is not yet supported throughout major browsers during the planning phase (for example Microsoft Edge and Safari are still not fully supported while Firefox is approaching the final implementations), it was decided to postpone this plan until later iterations [18]. Instead, a basic modularization is implemented with the help of Gulp plugins to isolate reusable components and only include them in the document when needed. On top of this, the component accepts optional parameter to alter its appearance or behavior depending on the context as well, such as in Listing 3.

```
// Inside a component markup
<div class="c-banner @if (context.isProduct) {c-banner-product} @if (con-
text.isMini) {c-banner-mini}">
  <a href=" " class="c-banner_image" style="background-image: url('@@im-
gUrl');"></a>
  <div class="c-banner_text">
    </div>
</div>

// Inside a page markup where the component is used
<section class="m-banner m-banner-triple">
  <div class="container-fluid">
    @@include('./_includes/c-banner-home.html', {"imgUrl": "img/accelera-
tor/product-1-demo.jpg", "isProduct": true})
    @@include('./_includes/c-banner-home.html', {"imgUrl": "img/accelera-
tor/banner-triple-landscape-demo.jpg", "banner_flag": "c-banner-product"})
    @@include('./_includes/c-banner-home.html', {"imgUrl": "img/accelera-
tor/banner-triple-sq-demo.jpg"})
  </div>
```

Listing 3. Simple HTML component declaration and usage

Like any international web shop, Company A also boasts a wide range of sectors featuring different purposes. On top of essential pages such as landing page, catalog, product and information page, it has multiple custom areas like group ordering, customer support, tutorial articles, etc. with their distinct appearance. Consequently, there would be a long list of markups, each representing one separate page. The markup folder, thus, is organized into three core areas: `chrome`, `includes` and `pages`. In the end, markup files are processed with Gulp to output fully completed documents.

The `chrome` folder includes elements that wrap around the site and remain the same across different pages. The two most notable of such elements are the header and footer. As a matter of principle, these two components would behave and look the same regardless of the currently viewed sector. For that reason, it is a good practice that they are separated from the rest of the page so that their functionality would not affect other components. Inside the `chrome` folder, the header and footer are further divided into smaller components. For example, the header will have a sub-module called `header_search-bar` and `header_toolbar` that can be imported into the main `header`. This helps reduce the number of code lines in a file as well as search time, not to mention the increase in modularity of the component.

The `includes` folder contains three types of files: markup files whose contents are similar but slightly different depending on the context, markup files that are repeated throughout different pages with slightly complex structure, and `component` files that are standalone elements with a clearly defined style.

```

@if (context.isRelatedProduct == true) {
  <a href="#" class="articleRelated articleRelated-product">
    <div class="item_image">
      
    </div>
  </a>
}

@if (context.isRelatedArticle == true) {
  <a href="#" class="articleRelated articleRelated-article">
    <div class="item_image">
      
    </div>
    <div class="item_title">
      Lorem ipsum dolor sit amet amet
    </div>
  </a>
}

```

Listing 4. Code snippet of an article element with two variations.

If the context of its import requires an article, the lower markup in Listing 4 will be displayed, while if there is a need for a product, the upper one is shown instead. This method is useful when there is a need for mostly shared but slightly different markups. Whereas normally a new markup file is to be created, developers can save time when searching for a specific markup. The second type consists of navigation markup with several links and lists, which is shared through many pages. A prime example of this is the navigation bar for page `profile`. Although different sub-pages of the profile section have different appearances, they all share the same navigation bar. Thus, it is logical to include it in all

variations of profile pages such as profile overview, user order history, etc. The third type is component that is complete on its own and can be included in any page if need be. This includes small essential elements such as alert, product comment, product card, etc.

```
<div class="c-alert">
  <p><a class="text-actionLink" href="#">Free shipping over € 50 Delivery</a>
time 3-4 days</p></div>
```

Listing 5. Code snippet to show the markup of the alert component.

Though simple, the alert component declared in Listing 5 is reused at all pages in times of special campaigns, or when a product is added to the cart. Having it isolated also helps the team with building the pattern library, as this is one of the most basic elements of the library.

Coming to `pages` folder, it holds the static layout of all page templates within the site. The files range from landing, catalog and product detail page to article and blog. Those are the main importing locations for the smaller `includes` and `chrome` elements. From here, they will be rendered with all shared sections such as header and footer, without bloated code lines during development.

```
gulp.task('rainbow:html', function () {
  return gulp.src(src_path)
    .pipe(plumber({
      errorHandler: function(err) {
        // handle error if needs be
      }
    }))
    .pipe(fileinclude({
      prefix: '@@',
      basepath: '@file'
    }))
    .pipe(gulp.dest(dist_path))
});
```

Listing 6. Code block of the Gulp.js task that processes HTML files.

The task to handle HTML is as written in Listing 6, with the help of `gulp-include` plugin, written by a developer named Hugo Wiledal, which enables custom markup inclusion. The plugin can accept a wide variety of files, such as JS, CSS or HTML and even those from external sources before inserting them in appropriate places. Following the same idea as other Gulp tasks, it accepts an HTML file as an input and pipes it through the plugin, which replaces certain section with an intended markup, before producing a complete HTML document.

3.2 Styling structure

Just as CSS is a crucial component in any website, its methodology also stands as a staple of any web development project. CSS methodology is a systematic process of developing CSS along with HTML [19]. Having a logical methodology is essential to a scalable and robust development project, not only because it can create a maintainable repository, but also it helps ease the efforts introduced when creating new elements. This process focuses on naming convention and file organizing.

The most basic approach to CSS structuring would be to embed CSS directly in HTML properties, as known as inline styling, although developers are advised to avoid such a procedure since the styling cannot be modularized that way. That is not to mention inline CSS has a very high specificity, which would conflict with further stylings [20]. Therefore, it is widely considered the best practice to detach CSS from the HTML markup.

In regard to styling architecture, having an organized directory is as important as establishing a good methodology. Naturally, styles should be separated into folders so that developers have an easier time understanding the structure of the project. In the long run, it will exponentially reduce the efforts to maintain the styling aspect. The development team researched into the most prevalent ways of organizing styling directory with their pros and cons. Upon learning their significance, the team would devise a custom method to suit the project.

```

stylesheets/
|
|-- modules/           # Common modules
|   |-- _all.scss      # Include to get all modules
|   |-- _utility.scss  # Module name
|   |-- _colors.scss   # Etc...
|   ...
|
|-- partials/         # Partials
|   |-- _base.sass     # imports for all mixins + global project variables
|   |-- _buttons.scss  # buttons
|   |-- _figures.scss  # figures
|   |-- _grids.scss    # grids
|   |-- _typography.scss # typography
|   |-- _reset.scss    # reset
|   ...
|
|-- vendor/           # CSS or Sass from other projects
|   |-- _colorpicker.scss
|   |-- _jquery.ui.core.scss
|   ...
|
|-- main.scss         # primary Sass file

```

Figure 4. A basic SASS project layout. Copied from John W. Long (2013) [21].

As illustrated in Figure 4, the necessary files for a standard CSS project can be divided in several different directories. It harnesses the power of SASS importing function to separate files into different places according to their purposes. The `vendor` folder holds the third-party (codes from an external source). It keeps prepackaged components written by other developers that should be exempt from direct modifications. These packages may be updated in the future unknowingly, so local changes to them can be made once an update arrives. A more flexible approach is to refer to them in other local files and modify them there, leaving the source codes intact. The `module` folder contains the non-output code from SASS, including variables, mixins and functions. This ability to have non-output code is highly useful as it allows for huge styling reusability without any extra weight to the code output. Switching to `partials`, this folder maintains the most essential construction components of a project. They include repeatable and / or site-wide elements such as buttons, grids, image figures and avatars. Finally, every file in those folders are imported into `main.scss`, which would consist multiple lines of corresponding `@import`.

```

/** IMPORT BASE LEVEL STYLING **/
@import "_shared/_all.scss";
@import "base/_all.scss";

/** IMPORT VENDOR FILES **/

```

```
@import "vendor/_all.scss";

/** IMPORT COMPONENT / MODULES / LAYOUT */
@import "components/_all.scss";
@import "chrome/_all.scss";
@import "modules/_all.scss";
@import "layout/_all.scss";
```

Listing 7. Code block to illustrate importing functions of SASS.

With the importing functions in Listing 7, the styling can be traced to a single source of control, ensuring an easier time managing dependencies. Otherwise, the compiler may have to check for an array of SCSS files, which can be tiresome for developers to add or remove. Being the most basic form of a CSS structure, this improves the modularity of component styling, as well as separates external source codes with internal development.

```

sass/
├── base/
│   ├── _reset.scss      # Reset/normalize
│   ├── _typography.scss # Typography rules
│   └── ...              # Etc...
├── components/
│   ├── _buttons.scss    # Buttons
│   ├── _carousel.scss  # Carousel
│   ├── _cover.scss     # Cover
│   ├── _dropdown.scss  # Dropdown
│   ├── _navigation.scss # Navigation
│   └── ...              # Etc...
├── helpers/
│   ├── _variables.scss  # Sass Variables
│   ├── _functions.scss  # Sass Functions
│   ├── _mixins.scss     # Sass Mixins
│   ├── _helpers.scss    # Class & placeholders helpers
│   └── ...              # Etc...
├── layout/
│   ├── _grid.scss       # Grid system
│   ├── _header.scss     # Header
│   ├── _footer.scss     # Footer
│   ├── _sidebar.scss    # Sidebar
│   ├── _forms.scss     # Forms
│   └── ...              # Etc...
├── pages/
│   ├── _home.scss       # Home specific styles
│   ├── _contact.scss    # Contact specific styles
│   └── ...              # Etc...
├── themes/
│   ├── _theme.scss      # Default theme
│   ├── _admin.scss     # Admin theme
│   └── ...              # Etc...
├── vendors/
│   ├── _bootstrap.scss  # Bootstrap
│   ├── _jquery-ui.scss  # jQuery UI
│   └── ...              # Etc...
└── main.scss           # primary Sass file

```

Figure 5. An example of a standard 7-1 project structure [22].

Still using SASS `import` feature, another CSS structure has arrived by the introduction of Front-end Developer Hugo Giraudel. The system in Figure 5 extends from the basic structure, requiring file categorization according to their purposes and significance. “Seven” in its name means seven folders with files serving different functions, while “one” dictates the only `main.scss` file that unites them. On top of `base`, `components`, `layout` and

pages, which are the building blocks of a web page, it also introduces custom folders such as `theme` for further customization and `helpers` for utilities during the development process.

Another major structure is based off the principles of Block – Element – Modifier. The idea from the core team of BEM is that there should be a standalone element that branches out different parts with little significance and different states to apply. As there are various implications from this, there presents a variety of loosely similar structures. However, they all share the `Base`, `Layout`, `Module` and another `State` folder.

```
base/  
  _b-reset.scss  
layout/  
  _l-grid.scss  
module/  
  _m-accordion.scss  
state/  
  _s-global.scss  
  _s-accordion.scss
```

Figure 6. Example of a SASS directory under SMACSS / BEM organizing method [23].

It can be seen from Figure 6 that a `base` folder is declared that carries all the important styling used across the site. This includes general styling reset, mixins and variables as well as 3rd-party styling. Inside the `module` folder, there are components at a micro level, for example, buttons or avatars. On a macro level, there is `layout` folder managing bigger components comprised of those modules. Finally, a `state` or `theme` folder houses possible thematic changes that affects the whole document (such as night mode or special campaign themes).

3.3 Methodology

As specified in Mozilla web documentation, most building blocks within a document are made of shared HTML elements such as `h1`, `h2`, `p` and `div`. Thus, it is struggling to style them individually without a way to reference these tags. One of the solutions is to use a selector such as `class name` or `id` [24]. Using class names allows intended elements to boast different appearance depending on the need of the design.

Methodology is a procedure or set of procedures to do a task [25]. Extending to CSS, methodology often means a way of writing and organizing CSS so that it is comprehensible, maintainable and quick to develop. Therefore, a desirable scheme should follow these criteria:

- Styling should be modularized for reusability.
- Styling should be resistant to styling leaks (from either third party or from internal codes).
- Styling should be logically orchestrated so different developers can follow the same idea without a steep learning curve.

The thesis will further research into popular schemes, covering their main pros and cons before introducing the final implementation in the project. The development team may need to forgo certain guidelines in each methodology to match the current project, but will try to keep the integrity and consistency of the final solution.

3.3.1 BEM

Block - Element – Modifier, or BEM is one of the most prevalent naming conventions currently, with 38.16% developers “feel comfortable using” in a recent survey for Front-End tooling in 2018, shown in Figure 7. It is “a methodology that helps you to create reusable components and code sharing in front-end development” [26].

	Never Heard of		Heard of/Read About		Used a little		Feel Comfortable Using	
CSS-in-JS	12.31% (672)	-15.67%	41.57% (2,270)	-2.93%	26.20% (1,431)	+10.21%	19.92% (1,088)	+8.39%
SMACSS	41.13% (2,246)	-0.23%	39.06% (2,133)	+4.99%	11.72% (640)	-2.65%	8.09% (442)	-2.57%
Object Oriented CSS (OOCSS)	31.99% (1,747)	+3.31%	45.07% (2,461)	+3.39%	14.03% (766)	-3.59%	8.92% (487)	-3.09%
Atomic Design	33.93% (1,853)	-8.04%	37.36% (2,040)	+4.01%	16.65% (909)	+2.17%	12.07% (659)	+1.87%
ITCSS	69.73% (3,808)	+0.72%	20.93% (1,143)	-1.00%	4.05% (221)	-0.38%	5.29% (289)	+0.66%
BEM	16.00% (874)	-9.33%	25.84% (1,411)	+2.09%	20.00% (1,092)	+1.84%	38.16% (2,084)	+5.40%
SUIT CSS	70.59% (3,855)	-0.36%	23.86% (1,303)	+0.37%	3.33% (182)	-0.46%	2.22% (121)	-0.27%

Figure 7. Most popular CSS naming schemes in 2018. Copied from the frontend tooling survey 2018 [27].

This extends beyond the scope of just naming convention, and this report will only focus the naming aspect of this methodology. Within BEM, B or “block” is a functionally independent element which can be reused and does not affect the external environment around it, that is, has no margin and positioning property on itself. “Element” is a part of the “block” and meaningless without a “block” otherwise. An element always follows a block, with a double underscore (“__”) connecting those two- For example, `form__input` would be the class name of an input element inside a form block. Finally, there can be an optional `modifier` to represent the state, appearance or behavior of the element. It is also meaningless without an element or a block before itself and is written after a double dash (“--”), for example `form__input--invalid`. If any of the block, element or modifier involves more than one word, they should be written in kebab case, that is, words connected by a dash. Combined, they form a class name and no HTML element should be styled without one. BEM introduces a wide range of benefits to a CSS system, including modularity from elements, low specificity from singular and unique class name, CSS leaking prevention from `scoped` class names. An example of BEM methodology is demonstrated in Listing 8:

```
<header class="header">
  <img class="header__logo">
  <div class="header__search">
    <input type="text" placeholder="Search">
  </div>
  <ul class="header__menu">
```

```

<li class="header__menu-item"></li>
<li class="header__menu-item"></li>
<li class="header__menu-item--active"></li>
</ul>
</header>

```

Listing 8. HTML element using BEM naming convention.

Seen from Listing 8, the structure of the `header` can be comprehended without much difficulty. This header will have 3 direct sub-elements under it, namely `logo`, `search` and a `menu`, inside which there are another set of three menu items, with one being in `active` state. With BEM, the markup will have a logical and human-friendly architecture, yet still have a high level of adaptability.

3.3.2 ITCSS

Inverted Triangle CSS, abbreviated as ITCSS, came out quite recently, posing itself as a modern approach to maintain and scale CSS with a concern for CSS specificity [28]. ITCSS instructs developers to organize their CSS code base in an upside-down triangle fashion, with the first layers being generic and turning more specific as the triangle goes deeper, as illustrated in Figure 8.

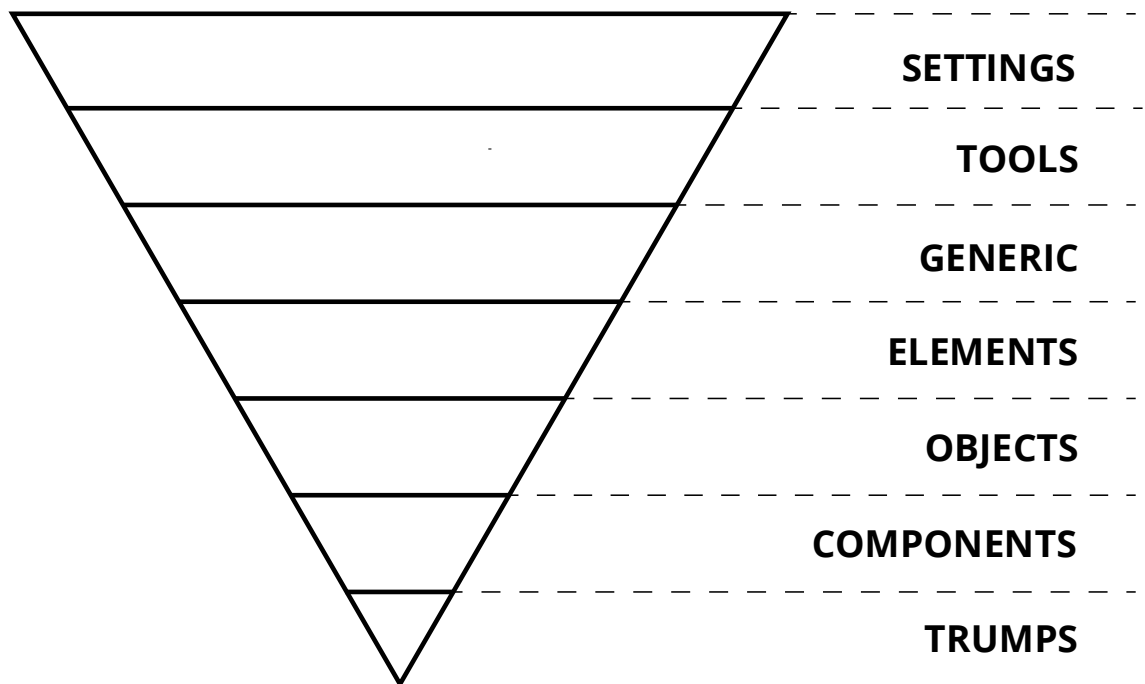


Figure 8. Diagram of ITCSS key principles. Copied from ITCSS: Scalable and maintainable CSS architecture (2016) [28].

The first layer — `settings` — define foundation elements such as color variables and fonts. Following it is the `tools` layer, which contains global mixins and functions that are shared throughout the code base. `Generic` layer represents default and reset styling that

synchronizes the styling between browsers. After this, `elements` layer introduces styling for bare HTML elements such as `h1` and `p`. `Objects` rules are for reusable and standalone UI components, and finally the `trumps` layer decides the variations of objects and components.

```
a {
  color: pink;
}

.s-purchase-page .c-modal a.linky {
  color: red;
}
```

Listing 9. Example code block of CSS declarations following ITCSS approach.

As seen from Listing 9, the first rule is contained in the `elements` layer, while the second should stay in `objects` or `components` layer. This is because `a` is a generic selector that applies to every single anchor element across the site. In contrast, `.s-purchase-page .c-modal a.linky` is very specific, only to anchor element with `linky` as its `classname` under `s-purchase-page` parent element.

3.3.3 SMACSS

Scalable and Modular Architecture for CSS (SMACSS) is also one of the prevalent naming schemes, with its core principle being categorization for patterns [29]. Though it shares some similarities with BEM, SMACSS still has some significant differences in naming and approach. To be more specific, SMACSS requires developers to separate their styling into 5 major categories: base, layout, module, state and theme. Starting is the `base` rules which are the simple reset or default rules that apply to macro elements through the pages. These rules often involve element selectors or pseudo-class selectors, such as `html`, `body`, `a` or `input`. `Modules` are the basic reusable building blocks of the design, for example callouts, alerts, product cards and so forth. `Layout` section defines how those modules connect with each other, consisting of at least two modules. `State` rules are slightly like the `Modifier` in BEM methodology in a way that they also describe the appearance or state of the module in specific situation. There can be also an optional set of `theme` rules that control the appearance of layouts should there be a need for theming. However, the need for this aspect has been on the decline for the past years, as can be seen from Figure 9.

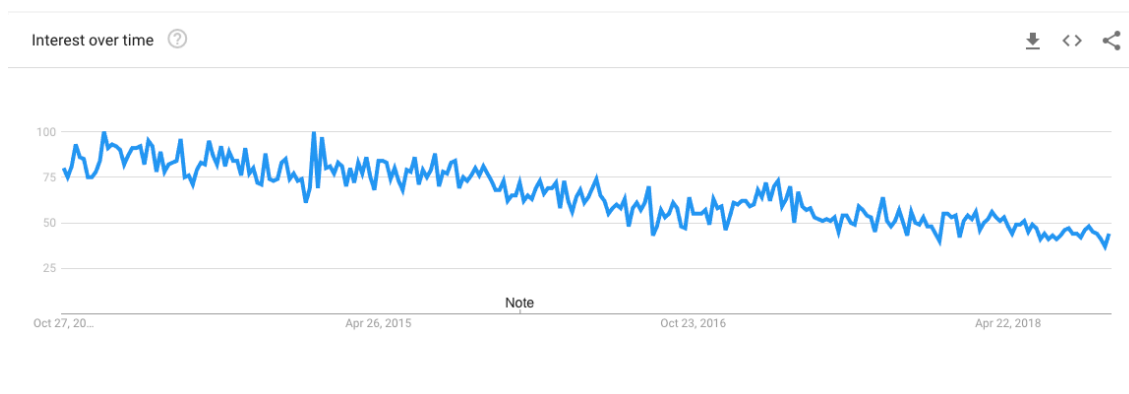


Figure 9. Google trend for website themes over the past 5 years. Copied from Google Trends [30].

Naming convention wise, SMACSS encourages the use of prefix for class names. For instance, layout `classnames` should have `.l-` at the beginning, or `.m-` for modules. This can prevent CSS leaking from 3rd-party plugins, and at the same time keep a clear context for the developers working on the markup.

Overall, SMACSS shares several similarities with BEM method, avoiding the use of element selectors such as `a`, `h1`, and `span`. while adopting the separation between states and the component's core. At the same time, it also bears some resemblances to ITCSS with its categorization by level and importance.

3.3.4 SUITCSS

Styled User Interface Tools CSS (SUITCSS) is another approach to maintainable and robust CSS development, coined around 2014 by a former Twitter tech-lead. It devises a thorough framework for UI component building, but this thesis will only mention its building principle and naming convention aspect.

The methodology also focuses on CSS modularity and cohesion, with its component being able to be reused without hassle and does not affect other components. Additionally, SUITCSS has a slightly distinguished naming approach compared to other methods, since it “relies on structured class names and meaningful hyphens” [31]. Following SUITCSS, developers will have component names in camel case (e.g. `componentName`) instead of single-dash connected words. Component's children also follow a single dash, instead of double dash as in BEM. Instead, double dash is reserved for modifier names, such as `component--default`. Utility styling is written with a `u-` in front of the rule name, for example, `u-floatLeft` for left-floated helper.

Although this naming convention differs much from the traditional method (CSS class names are connected in style of camel case instead of kebab case), it helps bring CSS closer to JS as the so-called “camel case” naming is highly honored in JS, not to mention also reduces the length of class names, which poses a huge problem in BEM.

3.4 Custom implementation

3.4.1 CSS custom implementation

After sessions of discussions and trial-error cycles, the development team finally decided to adopt a custom methodology mixed the most useful features from all the terminologies.

For file structuring, the styling directory is separated into 8 folders: `_shared`, `base`, `components`, `modules`, `layout`, `chrome`, `vendor` and `styleguide`. Out of those directories, `styleguide` is involved with the pattern library and has no impact on any page of the outcome. Therefore, only 7 folders are explained in this report section. They each have their own `_all.scss` file which import all the necessary files under their domain. Lastly, there is an outermost `site.scss` that again imports all those subsidiary `_all.scss`.

To be more detailed, the `_shared` folder is the first layer of the CSS system, holding functions, mixins and font declarations throughout the code base. The underscore in front of the folder name indicates that no rule from this folder is compiled as CSS, except for font rulesets. This directory functions in the same way as the `tools` layer from ITCSS, which proves to be essential for quick UI development.

```
background-image: url('./path/to/image');
background-repeat: no-repeat;
background-position: center;
background-size: contain;
width: 10px;
height: 10px;
```

Listing 10. A standard way to style a CSS icon.

For example, to style a 10x10 icon using a PNG or JPEG as the image source, the standard way is to have a snippet like in Listing 10. However, as the icon sizes and image source change, it quickly becomes tedious that the developers must repeat those lines of codes at every place needed and change just a few details.

```
@mixin background-div($width, $height, $url: 'placeholder') {
  @if ($url == 'placeholder') {
    background-color: #e32636;
  } @elseif ($url == 'transparent') {
```

```

    background-color: transparent;
  } @else {
    background-image: url($url);
  }
  background-repeat: no-repeat;
  background-position: center;
  background-size: contain;
  width: $width;
  height: $height;
}

```

Listing 11. A reusable function to style a CSS icon.

Instead, as seen in Listing 11, the sizes and source are parameterized and solidified into one mixin, which can be implemented as one line, for example: `@include background-div(10px, 10px, './path/to/image')`. On top of that, for development purposes, when the icon asset is not ready yet, developers can use the default argument `placeholder` to mark the icon as a highlighted square. This not only makes the process of finding the icon to fill later much faster, but also signifies for the designer and product owner about the icon positioning there.

The `base` folder consists of declarations for the color palette and typography utilized in all pages. In fact, currently those two are the only files situated in this folder, due to their significance to all the elements within the project. Each of them holds both variable declaration and utility classes so that suitable colors and typography style can be applied both through markup and CSS.

Following `base` is the `component`, which constitutes the main constructing blocks of the project. Each file in this folder is written with the prefix `c-` for easier searching and organization, within which holds the styling for a standalone component that can be reused anywhere in the page.

```

.c-review {
  text-align: left;
  display: flex;

  &_avatar {
    padding: 0 15px 0 0;
  }

  &_body {
    .c-review_byline {
      margin-bottom: 10px;
    }
  }
}

```

Listing 12. A review component styling.

Utilizing SASS mixins, variable and parent reference selector (“&”), the ruleset in Listing 12 defines the style for a normal review element, with its appearance changing on mobile

devices. Each of this component should keep a strict cohesion, meaning it should not have positioning properties, such as `margin`, `padding` or `position` that affect the component itself, unless the property persists heavily throughout all its occurrences.

Instead, such properties should be contained in the `module` folder, which defines how a combination of components should appear together. Each file in this directory also has a prefix `m-` to match its class name.

```
.m-reviewList {
  &_heading {
    @include text-heading;
  }

  .c-review {
    padding-top: 15px;
  }
}
```

Listing 13. Review component styling used in the review list module.

This module in Listing 13 is repeated in the product page and in some article page in the future, with a heading and a collection of reviews directly below it. By scoping the styling to `module` instead of `component`, the basic blocks can be used without worrying about overriding multiple properties. Any further customization will be put in the `layout` directory which decides each component and module behave on a page-wide context.

```
.l-home {
  .home_populars {
    .m-reviewList {
      display: flex;
      justify-content: space-between;
    }

    .c-productBox {
      margin: 0 15px;
    }
  }
}
```

Listing 14. Component and module styling in a certain page.

Component and module styling (`c-productBox` and `m-reviewList`) can also appear in the `layout` directory as illustrated in Listing 14, as those elements can appear differently from pages to pages. In fact, this `layout` directory includes styling that is specific to a certain page series, or even a single page. Any styling file from `layout` folder has its name starting with `l-` and so does the class name itself. Contained within `layout` are also rulesets that are not reusable, or rather, only get applied to elements from a certain page.

Folder `chrome` holds styling related only to the `chrome` components, namely the header and footer at the moment of this writing. The styling inside is unlikely to be reused, so it appeared illogical to put the rule sets in any of the `component`, `module` or `layout` folder. That is not to mention the header and footer styles are complicated enough with various

small modules such as the dropdown menus, toolbar menus and lists to have its own place.

Finally, there is a `vendor` folder to hold all the 3rd-party styling, from big framework (Foundation, jQuery) to smaller component such as `select2`, `date-picker`, etc. The styling is manually copied via either Content Delivery Network (CDN) or the author's source, then pasted as a standalone file before being imported in the `_all.scss` in the `vendor` directory. This way, the developers can select which library to include or exclude depending on their need.

As for naming convention, the team adopts a customized terminology. Apart from the require for prefixes in certain reusable classes, there are also other rules to construct a name for any element. Class names with more than one words should be written in camel case, for example, `.productBox`, with its children following after a single underscore: `.productBox_heading`. In the case of variations, a single dash with the respective modification should come after, such as `.productBox_heading-mini`. Should there be a need for state or mode, it is followed by a double dash: `.productBox_heading-mini--highlighted`. Adapted from BEM and SUITCSS, this methodology guarantees a logical naming scheme, while keeping the names short and compact.

3.4.2 JS custom implementation

JavaScript also plays an important role in a functional online shop, since purchasing online requires a variety of interactions such as adding to cart, bookmarking a product and opening a campaign modal. Having an established JS structure is helpful for a quick pace of development as well as maintainability in the future.

The script section is divided into two main directories: `accelerator` and `vendor`, with a central initialization file `startup.js`, which is similar to the CSS part. Between the two, `accelerator` folder holds the core scripts for the website, with various indispensable functionalities such as `cart`, `modal`, `search`, etc. Each of such module is separated into a JS file and will be called into action by “`startup.js`”:

```
$(document).accelerator({
  cart: { debug: true },
  checkout: { remove_confirmation: 'Do you want to remove this product?' },
  minicart: { random_setting: '' }
});
```

Listing 15. Initialization of Accelerator modules in the project.

The approach in Listing 15 helps with code organizing and the separation of concern. For example, developers can make tweaks and fixes in the module `cart` without having to worry about its impact on either `menu` or `slideshow` module. Additionally, controlling such modules is without hassle, as each can communicate easily with one another through the `events` system. When an event (e.g. a click on the buy button) is triggered, the interacted module will “broadcast” that event to the whole accelerator system with the help of jQuery's `.trigger()` function, upon which one or more corresponding modules will react depending on its logic.

```
// In the cart component
Accelerator.libs.cart = {
  addArticle: function(cartParams, source) {
    if (parseCartParams()) {
      $.ajax({
        // ajax parameters
      }).done(function() {
        if (source) {
          source.trigger('update-minicart');
        }
      });
    }
  }
}

// In the minicart component
Accelerator.libs.minicart = {
  events: function(scope) {
    $(document)
      .off('.') + this.name)
      .on('update-minicart', function (e) {
        self.update();
      })
  }

  update: function () {
    $.ajax({
      url: site.settings.minicartUrl,
    }).done(function (html) {
      // update mini cart content
    });
  }
}
}
```

Listing 16. Interactions between a cart component and minicart component in Accelerator

From Listing 16, when a product is added to cart, an event called `update-minicart` will be triggered and sent to many components, one of which is the `minicart`. From there, `minicart` will call its function `update()` to fetch the data from the server to update its content accordingly. This ensures the users will see their cart updated as soon as they purchase the product, thus, a responsive buying process.

In `vendor` directory, there are mostly 3rd-party libraries scripts that enable a smaller scope of function, such as making an interactive dropdown, or validating form inputs, etc. All of these scripts are initialized as soon as the foundation scripts are initialized (even

before any `accelerator` component) so any dependent element can function properly. At the end, files from both `vendor` and `accelerator` folder get compiled and minified by Gulp before delivering to the customer.

Rarely, there can be a need for a small code snippet that is too trivial to construct a whole new `accelerator` library. In that case, depending on how often it is used throughout the page, the snippet will be put either in the file `ClientScript`, which holds global interactions, or is written directly inside the component that utilizes the script itself. For example, the header in certain pages, which would be “fixed” onto the top when the customer scrolls past a certain point.

```
// sticky navbar when scrolling past the header
$('#navbar-stick').sticky({
  topSpacing: 0,
});
```

Listing 17. Enabling sticky header in certain pages.

This function in Listing 17, while necessary as per the design, is rather small and not streamlined enough to be an `accelerator` module. Thus, it has been left inside a separate file that will be imported into every single page of the site. However, the development team already has developing plans to transform these scripts into modules of some forms to further ensure the consistency in the code base and reduce the efforts when maintaining the features.

To sum up, the JavaScript sector is separated into 2 major parts, one of which includes the core `accelerator` library while the other contains all the necessary external scripts. Apart from these two (which eventually get compiled into a big chunk of script), there are also smaller snippets that are specific to some components without being bundled into the main script file.

3.4.3 Asset management

Even though HTML, CSS and JS are the primary components of any web page, there is also a need for media assets such as images and videos, otherwise the page will look bland to the customers. Since the outcome is a web shop, many of the images and videos will be about the products and come from the server. However, static assets such as icons should be included in the frontend since they are exclusively visual-related.

For this reason, icon sprites are housed within the `resources` folder under its own `img` directory. From there, they are optimized for web rendering through Gulp and output to `dist` folder for distribution. The same principle applies to static images, videos and even fonts.

```
const gulp = require('gulp'),
      imagemin = require('gulp-imagemin');

gulp.task('images', function (cb) {
  return gulp.src([src_images + '**/*.{png,jpg,gif,svg}'])
    .pipe(imagemin())
    .pipe(gulp.dest(dist_images));
});

gulp.task('fonts', function (cb) {
  return gulp.src([src_resources + '**/*.{ttf,otf,woff,woff2,eot}'])
    .pipe(gulp.dest(dist_resources))
    .pipe(browserSync.stream());
});
```

Listing 18. Asset processing tasks.

As shown in Listing 18, a Gulp plugin called `imagemin` was utilized for compressing images, then output it to a separate folder for images. For other assets that require no processing, they are directly transferred to their respective folders.

3.5 Pattern library

With the growing complexity of website, follows the evolving demand for a better web development workflow. Thus, design systems have become a common matter of concern in large-scale web projects lately, due to the need of a unified design language amongst the team, for example, Audi brand as in Figure 10 [32].

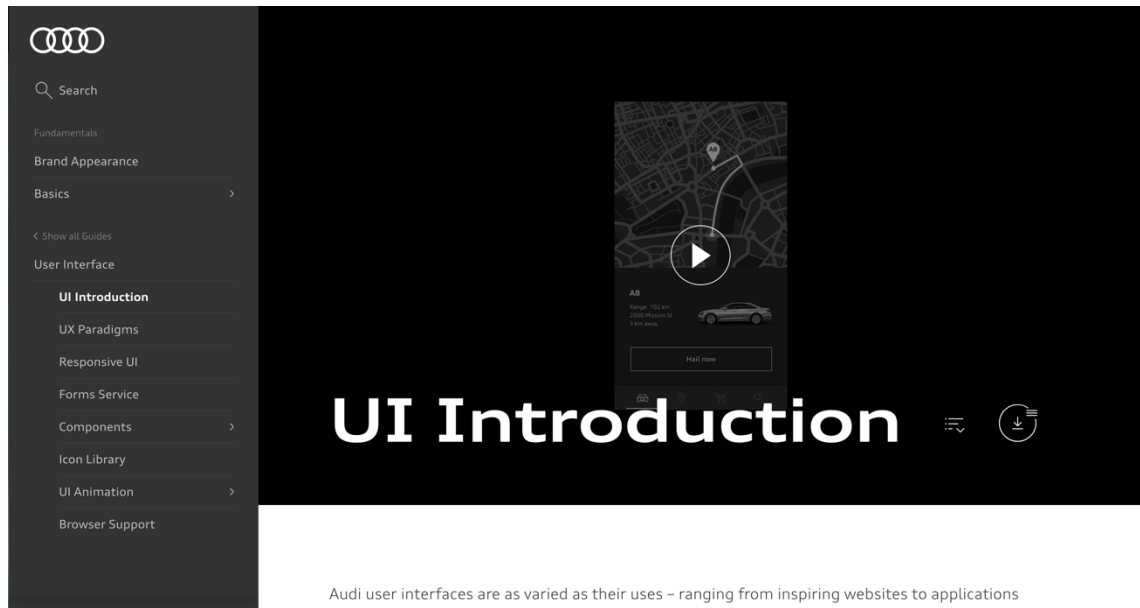


Figure 10. Audi's new design system published on the web. Copied from Audi official webpage [33].

Even without a design system, a project often already has its own language that is expressed through the colors, spacings, typography and many other elements within its concept. However, since only the designers are fluent in this language, any task of maintaining and developing such a visual system may prove to be difficult in the long run. Developers also need to understand this system, which is constructed by themselves. Therefore, our developer team decides to materialize that language, transforming it into a visual collection of elements so both developers and designers can keep track of the visual aspect.

However, design system is a large term to address, as it contains a wide variety of sub-domains such as design principles (the basic definition of the core values in the design) or writing rules (the principles of wording in the elements) [34]. After considering the work capacity and timeline of the project, the team agreed to only make a pattern library, which is a style guide subset with reusable and interactive components for building bigger sections. Such a library usually involves a collection of element markups, the codes constructing them and instructions about each of their use cases.

3.5.1 Tooling

There are a variety of tools at the moment dedicated for the building of design systems (and in particular, pattern library). In fact, new frameworks are being developed every day to smoothen this process for both designers and developers. In the current project,

the team decided to use Astrum, a compact framework from No Divide Studio, to make the foundation of the pattern library. Astrum came out first in 2016 and has been under constant development up until the beginning of 2018, with many improvements since.

To get started, this framework needs to be installed in the global environment of the machine with the command `npm install astrum -g` (this also requires the machine to have NodeJS and npm installed beforehand). Afterwards, there needs to be a folder to house all the pattern library logic, which can be created by running the command `astrum init ./[path]` where `[path]` is the `root` folder of the frontend directory. This is the basic steps required to have a functional Astrum-based pattern library initialized in the workflow.

For a new component to be created, whether a small or a big one, the command `astrum new [group]/[component]` needs to be executed, with `[component]` being the element itself to be materialized and `[group]` is its container, for example `astrum new headings/hero` was used to create the hero element of the headings group. Upon its execution, a directory of that group name is created, with the element being its child directory or if the group is already present, just the element directory will be inserted. Inside the element directory, there will be a `markup.html` file for the markup of the component and a `description.md` file to note its definition, usage and considerations. All these elements and their groups are built and loaded according to its core `data.json` file as illustrated in Listing 19.

```
{
  "project_name": "Company A Pattern Library 2018",
  "assets": {
    "css": [
      "../styles/site.min.css",
      "../styles/styleguide/styleguide.css"
    ],
    "js": [
      "../js/site.min.js"
    ]
  }
}
```

Listing 19. Astrum project configuration example.

Astrum accepts the option to include the processed CSS and JS file in its resources. This can save a considerable amount of time for development team, since the library can utilize the already-processed styling and scripts.

Not only the creating process, but also the editing and removing process in Astrum also proves to be simple. The developer needs to run `astrum edit [group]/[component]` and

`astrum delete [group]/[component]` respectively for those tasks. The framework will then present an interface in the CLI for fast and smooth interaction.

The designers of the project team also enjoy the use of Sketch, a tool specifically built for rapid website prototype building and designing. Sketch offers a wide range of tools for a better design flow, including the support of `symbols` — reusable component in the design — that is capable of exporting CSS to be used by the developers [35]. It can be said that this application plays a major part in the planning and building the library components. Together with Astrum, Sketch helps streamline our process of modularizing visual components.

3.5.2 Implementation

To cleanly implement a new component into the pattern library, both the design and development team have to work closely with each other. The process begins with the design being scanned for similarly functional and visual elements, starting from basic parts such as typography, color and going up to more complex ones, for example, icon with tooltip or product card.

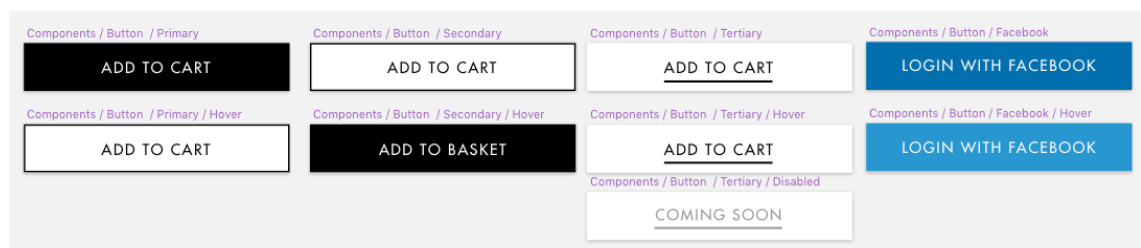


Figure 11. A collection of button components used within the project.

As the buttons in Figure 11 are confirmed, the developers simply need to right click on each of them and select “Copy CSS Attributes” to get the exact values of the element. Afterwards, its styling is defined in a separate file in either `component` or `module` styling directory. This is processed altogether with other styling and be output as one single file, which will then be included in the resource path of the pattern library. After completing the button styling, the commands `astrum init buttons/primary`, `astrum init buttons/secondary` and `astrum init buttons/tertiary` are executed so each variation can have its own markup and description container. Inside each `markup.html` file, the developers insert the proper markup as well as container if needs be, such as in Listing 20.

```
<a href="#" class="c-btn c-btn-primary">Primary Button</a>
```

Listing 20. The markup of a standard button.

Finally, a meaningful description is included in the `description.md` so that its purpose is clearly expressed to the viewers. The result will be a section in the pattern library with the component name, description, visual demonstration and code sample like in Figure 12.

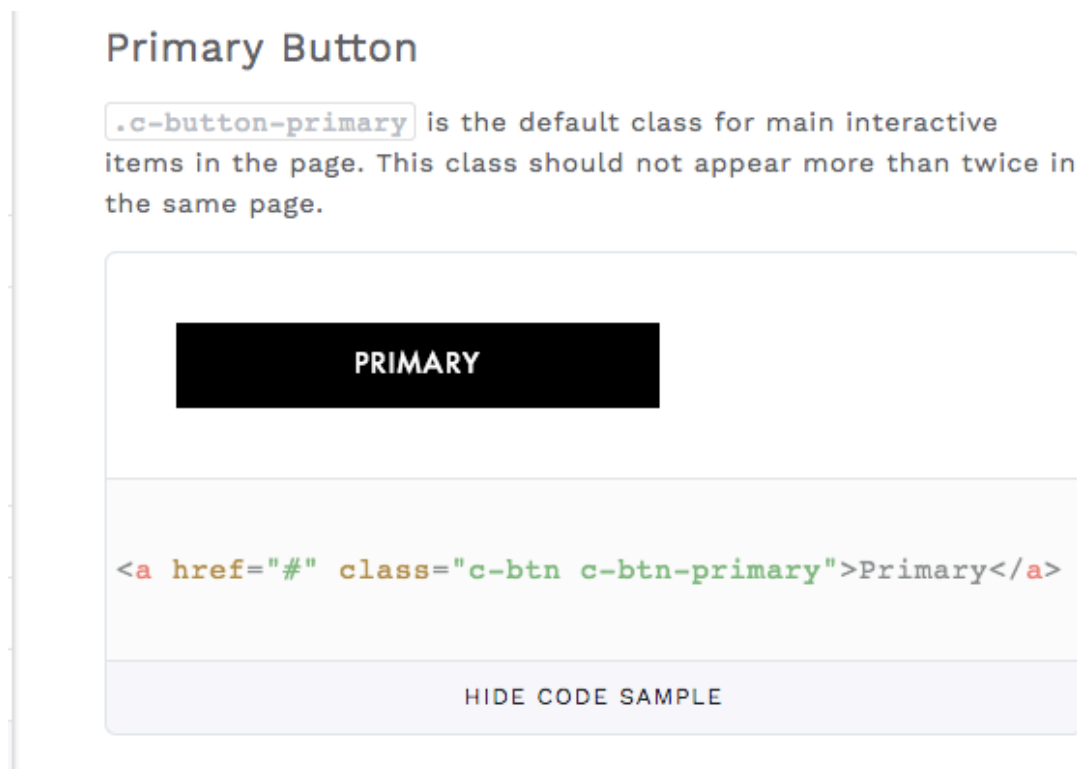


Figure 12. The outcome of a button in the library with its description, markup and code.

With the pattern library, not only can the designers see their creation in a working environment both on desktop and mobile devices, but the developers can also see how to construct a component without having to search in the code repository. For the current team, this serves as a design archive, while for future colleagues, it can introduce a smoother onboarding experience, so they spend less time trying to understand the code base. That is not mention the pattern library can also extend to the marketing team of Company A, helping to unify the appearance on the website and on their physical releases such as newspaper and magazine. At the time of this writing, the pattern library is still under development and finessing, but it should be ready by the time the migration is completed around early 2019.

4 Result & Discussion

4.1 Result

It can be said that the implementation of the new workflow was done successfully. It enables an independent front-end with seamless background processing for fast development. The outcome is served by the plugin BrowserSync through Gulp from the `dist` folder in Figure 13, from which styles and scripts will also be linked in the production site. Each `.html` file is a separate page, which can be inspected by the back-end developers so proper dynamic markup can be implemented exactly like in the design.

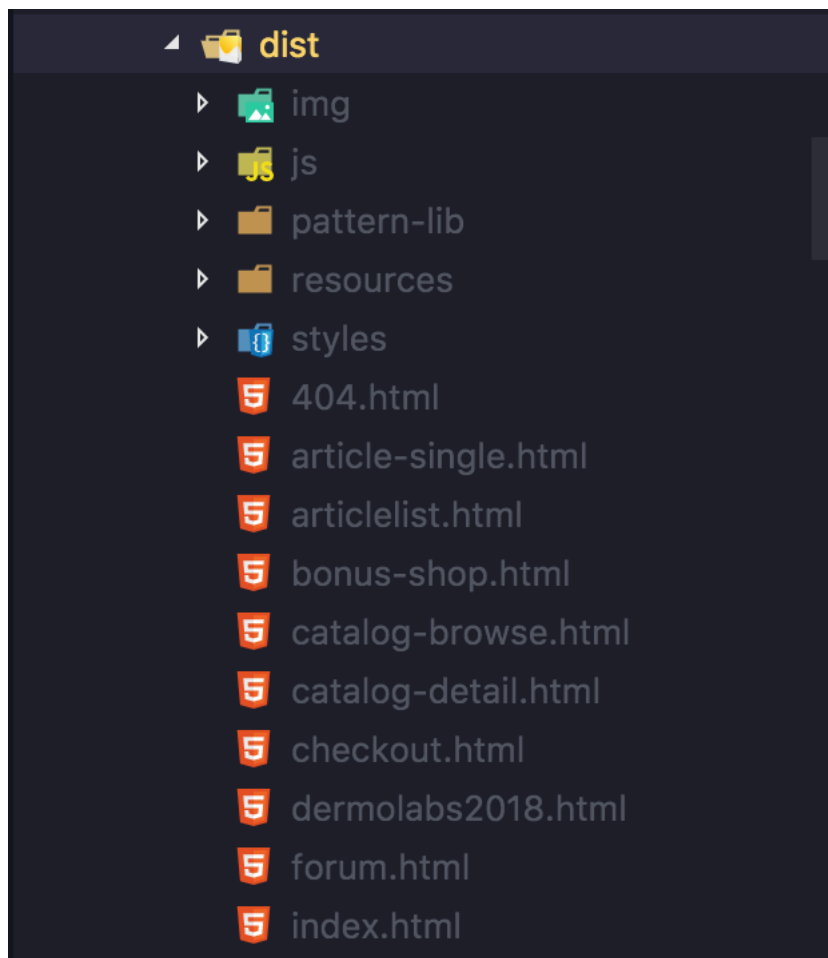


Figure 13. The final distribution folder housing all the processed frontend code.

This way, the front-end can be installed and developed without any connection with the server, ensuring the pace by each iteration. This creates a working local development environment, having all necessary components rendered with responsively proper styling and interaction, shown in Figure 14.

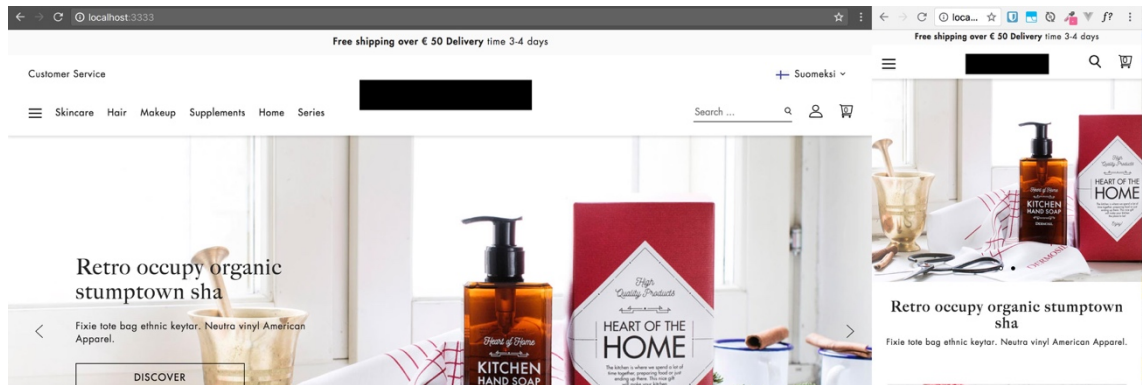


Figure 14. Illustration of the web shop landing page on desktop and mobile screen.

In addition to the main project, a pattern library is also served in this local environment and ready to be deployed on any location. Though still unfinished, the pattern library has proved its usefulness already. After a few months into the project, even the designers might have forgotten how certain element looks. After looking into the pattern library, he or she can recall again. This does not limit to designers only, but also benefits the developers whose duty is to implement and tweak dozens of components every week.

4.2 Discussion

4.2.1 Structure

To complete such a workflow, the development team had to overcome various challenges, with even some of them still being unsolved at the time of the writing. However, most crucial blocks were cleared during the process. One of the first blockages was to select which e-commerce platform to migrate with, since it also decides the frontend workflow for the team. Several options came up, such as Magento, Shopify and WooCommerce, however, Litium was the ultimate selection, due to its scalability and language handling capabilities. Each of the markets is assigned to a sector in the admin dashboard with its own content. This way, each team of corresponding market can make changes without disturbing other markets, which also leads to fewer errors and quicker translation work.

Another notable struggle was the decision to separate completely the frontend and the backend. By default, in Litium framework, there was no clear ground for static HTML and CSS, as the web pages are rendered directly using Windows IIS. This is a very direct approach, but considering the team is completely new to the framework with multiple infrastructure concerns to be handled, it was decided not to be the best way to work for

this migration. Instead, the team would create a standalone frontend sector and work independently to match the client's request. There are multiple reasons behind this argument.

First off, the current project favors the speed of the frontend team. Since the team adopts the Scrum concept of Agile process, there should be a minimally working product every week [36]. According to the process, website layout sketches are iterated every certain week and urged to be implemented so that the client can see its preview in a real browser. This is to allow Company A to have a good understanding about the outcome gradually. If the frontend must rely on the data from the backend, it is certain this weekly iteration, or so-called "sprint" will be delayed. While there can be various uncertainties in the data migration process, thus delaying the backend integration, the design process hardly suffers the same thing. Therefore, the frontend should prepare the fully styled markup first, then transfer to the backend developers to inject proper data later.

Secondly, this separation also helps improving the project's modularity. As backend and frontend technologies often advance at different paces, it is often logical to update just parts of the solution instead of the entire project. Having those two sides coupled together can obstruct this process. Instead, the team can change technology in either side and keep the other one intact. In addition to the ease of solution upgrading, the approach also helps with failure prevention and error detection. Different strategies to compile, test and deploy can be utilized to ensure the team can locate errors and ensure as little interruption for clients as possible.

Finally, the current approach encourages the development of API, which enables future standalone projects and even help save time in migrations if needs be. API dictates how the components interact between each other, allowing separation of concern during development and guarantee the data can be used even outside the scope of Company A's website. For example, the client can issue another web application on a different domain just for a specialized product ordering. Normally this would be a huge effort without the use of API, since there are limited ways an external application can receive data from the database. It opens many opportunities in the future if the client decides to develop a native mobile application or a small campaign using just specific data from the backend.

Albeit, it should be noted that this method does not go without drawbacks. One notable issue is the effort to make both ends work separately, including building, testing and

pushing to production. While this only requires time in the starting period, it still involves a great amount of thinking and planning, which might be overwhelming for the team. Also, since many functionalities in the frontend side still relies on the data output from the Litiium framework, the separation is not complete and there will still be some coupling points during the development process. It was not the easiest decision to be made and cost the team a considerable time to finalize.

4.2.2 Task automation

Task automation selection was also a matter of concerns at the beginning. Before Gulp.js was picked, the adopted task runner was Grunt.js. Grunt.js was, in fact, the default tool that the Litiium framework included in their package. Grunt.js appeared first in early 2012, and got the first major update in February 2013, making it one of the first JavaScript task automation tools [37]. It puts heavy focus on configuration, with each task being defined clearly and separately from one another. Grunt.js comes with a set of built-ins for common functions such as compressing images, compiling CSS, composing JavaScript etc., each requiring a source file (or files) and destination for output in a loosely JavaScript Object Notation (or JSON) format. Gulp.js materialized later in 2014. While the end goal is the same — reliable task automation — Gulp.js handles the process differently by utilizing Node.js' stream [38]. It also changes the format for writing tasks, which looks and feels more JavaScript than JSON. Along with default functions, Gulp.js also introduces various opportunities for 3rd-party plugins that developers can write and use themselves without much hassle.

Both Grunt.js and Gulp.js aim to be the most dependable task automation tool. However, the core principles differ between the two. While Grunt.js is focused on configuring every single task, Gulp.js harnesses the power of Node.js stream to do tasks so they are connected to each other seamlessly. The benefit from this is not visible for small projects, but quickly become appreciated in larger ones with dozens of tasks. In this case, Grunt.js configuration file can grow to hundred lines of repetition due to each task being declared separately. Gulp.js removes this problem by `pipiing` tasks, thus reducing developer's time to set up. On top of that, Gulp.js proves to be faster than Grunt.js as demonstrated in Figure 15, with its ability to build files without writing intermediary files onto disk first [39]. That is not to mention Gulp enjoys a broader community, offering a wider range of custom plugins than its counterpart. For those reasons, the development team decided to opt for Gulp.js as the main build tool instead of Grunt.js.



Figure 15. Compiling speed comparison between Grunt.js and Gulp.js. Copied from Gulp vs Grunt – Comparing Both Automation Tools (2017) [39].

Gulp.js was picked to automate a handful of tasks in the development phase, including markup, styling and script processing, assets optimization and live-reloading. As Gulp use Node.js streams, a Gulp task is written as a bundle of smaller tasks. Each of those minor tasks takes the input and outputs a `stream` of objects representing the result. For example, with the task in Listing 21.

```
const gulp = require('gulp'),
    sass = require('gulp-sass'),
    newer = require('gulp-newer'),
    prefixer = require('gulp-autoprefixer'),
    concat = require('gulp-concat'),
    path = require('path'),
    browserSync = require('browser-sync').create();

gulp.task('scss', function (cb) {
  return gulp.src(path.join(src_styles, 'site.scss'))
    .pipe(newer({
      dest: path.join(dist_styles, 'site.css')
    }))
    .pipe(sass({ outputStyle: 'compressed' }))
    .pipe(concat('site.css'))
    .pipe(prefixer({ browsers: ['> 1%', 'ie 10'] }))
    .pipe(gulp.dest(dist_styles))
    .pipe(browserSync.stream());
});
```

Listing 21. Anatomy of a Gulp.js task.

The target files are read from the file `site.scss`, then piped (or sent through) to be checked for changes by the plugin `newer` before getting processed by `sass`, `concat` and `prefixer`. The output then is written by `gulp.dest` to the desired folder, and eventually

the browser will be refreshed by another plugin called `browserSync`. This process can be demonstrated as Figure 16.

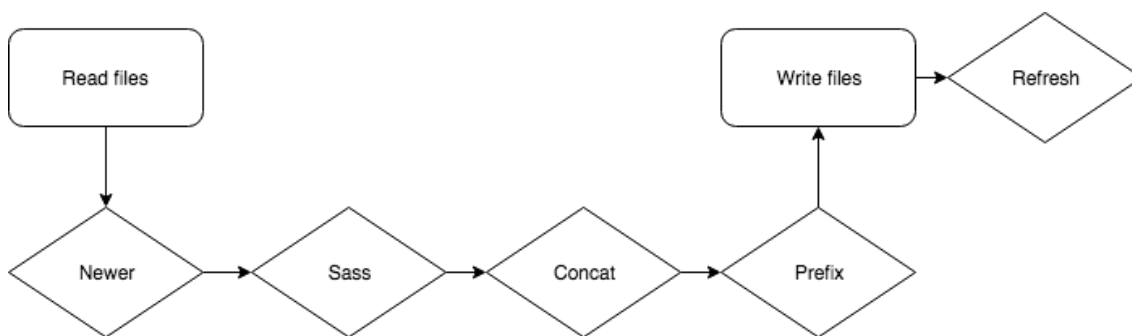


Figure 16. Anatomy of a Gulp.js task flow.

Another difficulty was to establish a well-organized automation flow, so that any implementation would be done seamlessly in the background and just display the relevant outcome to the developers. This included processing all the necessary assets (notably CSS, JS, images and fonts), building the pattern library and separating the development and production build. Initially, the automation was done in Grunt that came along with the Litium Accelerator framework. Even though it has several limitations, the team was reluctant to switch to a new tool since little was known about the Litium framework. For example, it would be possible that some components within the framework could malfunction after such a big change.

However, as the project demanded it, all Grunt tasks were rewritten into Gulp. Soon, this proved to be a correct decision, since many other possibilities were unlocked, such as recursive file inclusion, webpack processing and faster building time. The ability to import file recursively also helps when implementing dynamic markup in the backend, as HTML files can be separated into components, reducing the lookup time should a new view for such elements be needed. This used to be impossible with Grunt, so backend developers would have to inspect a complete HTML page to find just a certain element to isolate into a new Razor file.

4.2.3 Asset optimization

All assets are also optimized for production so that the local development imitates the production environment as much as it can. In Figure 17, the minified version of the scripts, which is served to clients, is only around 40% the size of the full script. Once compressed by the server for delivery, it will become even lighter.



 site.js	Today at 12.32	973 KB	J
 site.min.js	Today at 12.32	402 KB	J

Figure 17. Illustration of different script outputs — for development and for production.

In fact, its size could have been further reduced if only core functionalities are included, while leaving the 3rd-party scripts to be loaded externally as the site is rendered. However, this approach was rejected due to multiple drawbacks. With a wide array of external scripts, the client must wait for a long time due to each styling request is processed separately in order. On top of that, in case a request fails, it can affect other components. Therefore, it was decided that the important 3rd-party styling or scripts should be included in the build process so that the team has full control over the output.

4.2.4 Styling methodology

The development team also spent a great amount of time devising a custom naming scheme for variables and files. Naming is important at all stages of development, as a logical scheme can ensure a faster building time and an reduce the effort to maintain the project in the future. From researching popular naming approaches, the team drew out their pros and cons. BEM is arguably the most prevalent method, with its clear instruction to construct a name, from block to element and modifier at the end. While it is clear and logical, BEM methodology also slows down the team due to its strict naming scheme. Every element needs to be given a `classname`, which can be quite long to read,, for example, `main-header__navigation-links-no-js`. This can cause some stressful moments to developer when only some simple element needs to be styled. ITCSS and SMACSS also offers some good principles such as categorization based on purpose and the according prefixes.

Finally, SUITCSS presents a fresh approach to naming classes and variables in general, that is to go with `camelCase` instead of traditional `kebab-case`, which is similar to JS variable naming and reduces the length of each `classname`. The ultimate scheme was a mix of all those four methodologies, adopting the best possible effects while removing rigid propositions. Compared to the same class name in BEM for example, `.productBox_heading-mini--highlighted` is just as concise as `.product-box__heading--mini--highlighted`, with even more clarity. Although it differs slightly from the prevalent BEM naming method, once picked up, this approach allows for more versatility and rapid development. At the time of this writing, this implementation has yet to reach its maturity, with

still discussions being held for edge cases and code consistency, but it has been showing promising results within the team so far.

4.2.5 Pattern library

To select the approach for the pattern library was not a rushed decision as well. The developer team considered several options, including the traditional Knyle Style Sheets (KSS), SC5 Style Guide Generator, Living Style Guide or even modern libraries like Storybook. However, after trials and errors, Astrum was selected due to multiple reasons:

- Astrum is light-weighted and can be included without a complicated setup phase
- Astrum is un-opinionated, that is, it does not require any specific syntax or environment to function
- Astrum offers CLI to interact, which makes the onboarding and development process much smoother
- Astrum is configurable by default, and also can be extended since its Vue framework is quite familiar with the frontend team

Astrum has a definite advantage over many other libraries that it is un-opinionated. Developers are not expected to follow any set syntax from the library. Whereas, Knyle Style Sheets, another popular choice to construct a pattern library, requires a specific markup in the styling files to function [40]. While this approach can save some space, it also bloats the CSS files and steepens the learning curve for new developers. However, Astrum is not without its lacking. For example, it does not have a searching function built-in, which may prove useful in a complex pattern library. The frontend team was considering building a custom search functionality inside the Astrum framework but has met with several technical difficulties considering developments from the No Divide Studio were halted for months. However, just with its core functionalities, Astrum still proves to work sufficiently well in the building of the pattern library.

It is also a great endeavor to make a good markup both for frontend and backend according to the design. Without knowing about the restrictions and requirements from Lithium, it is possible that the markup has to be remade either partially or completely. This would not have happened if the development team chose to only work from the backend side, since there would be only one type of markup to implement. Nevertheless, the approach is not suitable for creating a pattern library and would not be able to output weekly iteration for the client. Therefore, while continuing with the separation method, the development team had to devise some method to reduce the conflicts between backend /

frontend markup and logic. The situation has been constantly improved, with our increasing insight into Litium framework and an established visual component library.

5 Conclusion

The primary goal of the thesis was to describe the frontend structure of an overhaul project for a major web shop in Finland. This required a migration from an old ASP.NET platform to a new one called Litium, thus re-writing the entire frontend with a new automation flow, new styling structure and pattern library for reference purposes. After several discussions, it was decided that the frontend and backend would be separated to provide a flexible development environment. The team also discussed the principles for all major frontend components, including HTML, CSS and JS along with the best structure to establish.

Through a great amount of efforts, a standalone frontend was constructed, giving smooth development experience, comprehensible workflow and optimal output to distribute to customers. More specifically, the setup can handle markup files in a modularized way for a more organized and intuitive component system. Additionally, SASS files are written in a logical methodology inspired by established CSS approaches to ensure a robust and maintainable styling system. The setup will also process scripts to produce a single compressed file, saving downloading time for the clients. Lastly, other static assets such as fonts, images and videos can also be optimized and ready to deploy on the production server.

On top of that solution, the same setup was also utilized to build a pattern library, which would help immensely with the effort to have a consistent visual system. This library is a completely standalone application, with the goal to unify developers and designers' intention, and also that of the visual department at the client company as well.

There were multiple issues the development team had to solve, most of which were complete. There remain still some struggles, one notable example being the discrepancy between the static HTML markup in the frontend and the dynamic Razor markup in the backend. As the static pages are scaffolded and decorated at the same pace as designer's work, they might not follow the requirements of the Litium framework markup. This leads to a recurring slow-down later as the frontend developers have to return and fix the markup accordingly. Nevertheless, the current structure has proved to accomplish the initial goals effectively and opens opportunities for further upgrades. As the front-end technology grows, it will keep evolving to adapt to the need of future projects should they be required.

References

1. World Wide Web Consortium. HTML 5.2 [Internet]. 2017 December 14 [cited 2018 August 16]. Available from: <https://www.w3.org/TR/html52/introduction.html#background>.
2. World Wide Web Consortium. HTML & CSS [Internet]. 2016 [cited 2018 August 16]. Available from: <https://www.w3.org/standards/webdesign/htmlcss>.
3. Mozilla Developer Network. JavaScript [Internet]. 2018 March 26 [cited 2018 August 16]. Available from: <https://developer.mozilla.org/bm/docs/Web/JavaScript>.
4. DocForge. Web application framework [Internet]. 2014 June 20 [cited 2018 August 16]. Available from: http://docforge.com/wiki/Web_application_framework.
5. Litiium Team. Architecture. Litiium Documentation Portal [Internet]. 2018 [cited 2018 August 16]. Available from: <https://docs.litiium.com/documentation/architecture>.
6. Node.js Foundation. Node.js [Internet]. 2018 [cited 2018 August 16]. Available from: <https://nodejs.org/en/>
7. Schoffstall, E. & contributors. What is Gulp?. Gulp.js GitHub repository [Internet]. 2018 [cited 2019 February 16]. Available from: <https://github.com/gulpjs/gulp>.
8. World Wide Web Consortium. What is CSS [Internet]. 2016 [cited 2019 February 16]. Available from: <https://www.w3.org/standards/webdesign/htmlcss#whatcss>.
9. Hidayat, A. Capturing Web Page without Stylesheets [Internet]. 2013 June [cited 2018 November 18]. Available from: <https://ariya.io/2013/06/capturing-web-page-without-stylesheets>.
10. Weizenbaum, N. & contributors. About SASS. 2013 [cited 2019 February 16]. Available from: <https://web.archive.org/web/20130901145805/http://sass-lang.com/about.html>.
11. Weizenbaum N. & contributors. Intro to SCSS for SASS Users [Internet]. 2019 [cited 2019 February 16]. Available from: https://sass-lang.com/documentation/file.SCSS_FOR_SASS_USERS.html.
12. jQuery Team. What is jQuery?. jQuery official website [Internet]. 2018 [cited 2018 August 16]. Available from: <https://jquery.com/>.
13. Litiium Team. JavaScript. Litiium Documentation Portal [Internet]. 2018 [cited 2018 August 16]. Available from: <https://docs.litiium.com/documentation/litiium-accelerators/develop/front-end/javascript>.
14. No Divide Studio Ltd. What is it?. Astrum [Internet]. 2018 [cited 2018 August 16]. Available from: <http://astrum.nodividestudio.com/>.
15. Debenham, A. FRONT-END STYLE GUIDES. 2017.

16. Laubheimer, P. What Is a Front-End Style Guide?. NNGroup [Internet]. 2016 March 27 [cited 2019 February 17]. Available from: <https://www.nngroup.com/articles/front-end-style-guides/>.
17. Mozilla Developer Consortium. Web Components [Internet]. 12 October 2018 [cited 2019 February 17]. Available from: https://developer.mozilla.org/en-US/docs/Web/Web_Components.
18. Web Components. Browser support [Internet]. 2018 [cited 2018 August 16]. Available from: <https://www.webcomponents.org/>.
19. Graham, D. An introduction to CSS Methodologies: Naming Conventions and File Structures. Codepen [Internet]. 2017 March 14 [cited 2018 August 16]. Available from: <https://codepen.io/hidanielle/post/css-methodologies-naming-conventions-and-file-structures>.
20. Bos, B. & contributors. Assigning property values, Cascading and Inheritance [Internet]. 2018 [cited 2018 August 16]. Available from: <https://www.w3.org/TR/CSS21/cascade.html#specificity>.
21. Long, J. W. How to Structure a Sass Project [Internet]. 2013 April 13 [cited 2017 November 11]. Available from: <http://thesassway.com/beginner/how-to-structure-a-sass-project>.
22. Giraudel, H. The 7-1 Pattern [Internet]. 2018 [cited 2019 February 16]. Available from: <https://sass-guidelin.es/#architecture>.
23. Graham, D. SMACSS. Codepen [Internet]. 2017 March 14 [cited 2018 August 16]. Available from: <https://codepen.io/hidanielle/post/css-methodologies-naming-conventions-and-file-structures#smacss-23>.
24. Goodman D., Dynamic HTML: The Definitive Reference, 2nd Edition. 2002.
25. Merriam-Webster. Methodology definition [Internet]. 2018 [cited 2018 August 16]. Available from: <https://www.merriam-webster.com/dictionary/methodology>.
26. Starkov, V. & Stepanova, V. Get BEM Introduction [Internet]. 2017 [cited 2017 November 14]. Available from: <http://getbem.com/introduction/>.
27. Nolan, A. The Front-End Tooling Survey 2018 – Results [Internet]. 2018 July 25 [cited 2018 November 10]. Available from: <https://ashleynolan.co.uk/blog/frontend-tooling-survey-2018-results#css-naming>.
28. Kmetko, L. ITCSS: Scalable and maintainable CSS architecture [Internet]. 2016 February 10 [cited 2018 August 16]. Available from: <https://www.xfive.co/blog/itcss-scalable-maintainable-css-architecture/>.
29. Snook, J. Categorizing CSS Rules [Internet]. 2018 [cited 16 August 2018]. Available from: <https://smacss.com/book/categorizing>.

30. Google Trends. Interest over time for “website themes” [Internet]. 2018 [cited August 16]. Available from: <https://trends.google.com/trends/explore?date=to-day%205-y&q=website%20themes>.
31. Smith, S. & contributors. SUIT CSS naming conventions [Internet]. 2018 March 26 [cited 2018 August 16]. Available from: <https://github.com/suitcss/suit/blob/master/doc/naming-conventions.md>.
32. ADCI Solutions. Top Design Trends for 2017 – 2018. Muzli. 2018 March 19 [cited 2018 November 11]. Available from: <https://medium.muz.li/top-design-trends-for-2017-2018-b9fd3025bc88>.
33. Audi Group. User Interface / UI Introduction [Internet]. 2018 [cited 2018 August 16]. Available from: <https://www.audi.com/ci/en/guides/user-interface/introduction.html>.
34. Sylvee L., What is a design system. Muzli. 2018 February 21 [cited 2018 August 16]. Available from: <https://medium.muz.li/what-is-a-design-system-1e43d19e7696>.
35. Sketch. Landing site [Internet]. 2018 [cited 2018 August 16]. Available from: <https://www.sketchapp.com/>.
36. Mountain Goat Software Team. What is Scrum [Internet]. 2018 [cited 2018 August 16]. Available from: <http://www.mountaingoatsoftware.com/agile/scrum>.
37. Alman, B. & contributors. Releases. Grunt.js GitHub repository [Internet]. 2018 June 4 [cited 2018 August 16]. Available from: <https://github.com/gruntjs/grunt/releases>.
38. Schoffstal, E. & contributors. Efficient Builds. Gulp.js Homepage [Internet]. 2018 [cited 2018 August 16]. Available from: <https://gulpjs.com/>.
39. Arsenault, C. Gulp vs Grunt – Comparing Both Automation Tools [Internet]. 2017 March 10. Available from: <https://www.keycdn.com/blog/gulp-vs-grunt>.
40. Neath, K. & contributors. Knyle Style Sheets [Internet]. 2018 [cited 2018 August 16]. Available from: <https://github.com/kneath/kss>.