

# API Specification

## Silhouette

Group 12:  
Mats Engelen  
Lars Erik Faber  
Håkon Marthinsen

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Group Description . . . . .	3
1.2	Delegation of Work . . . . .	3
<b>2</b>	<b>Background</b>	<b>3</b>
<b>3</b>	<b>Method</b>	<b>4</b>
3.1	API Design Specification . . . . .	4
3.2	API Design . . . . .	5
3.3	Implementation . . . . .	5
3.4	User Testing . . . . .	6
<b>4</b>	<b>Design Process/Results</b>	<b>6</b>
4.1	Design Specification . . . . .	6
4.2	Project Structure (File Structure) . . . . .	7
4.3	Client Code . . . . .	7
4.4	User Testing . . . . .	9
4.5	Revised API . . . . .	9
<b>5</b>	<b>Resulting Framework</b>	<b>9</b>
<b>6</b>	<b>Discussion</b>	<b>9</b>
<b>7</b>	<b>References</b>	<b>9</b>

# 1 Introduction

Silhouette is a web framework that let's users create websites with Java. But why Java? Although websites can be made using raw HTML and CSS, this approach comes with a few caviats that can lead to issues during development. In-short our solution aims to avoid these issues altogether, making the development process overall more enjoyable and creative, but let's go a little more into detail.

By design, HTML is a fixed markup language where the structure of tags determine not only the presentation but also the layout in the document object model (DOM). Content is grouped together in containers and follow a tree-like structure, which is nice for the end-user, but it can restrict the programmer. The problem here is that the programmer must follow the strict formatting guidelines of HTML, leaving no space for personalization. Another problem is that browsers will try to interpret the markup regardless of its validity, meaning that typos and other seemingly minor issues can easily go unnoticed.

An article titled "The Problem with HTML" by Dr. Maxime Chevalier-Boisvert argues how HTML, CSS and JavaScript is growing fast to become more than originally intended, blurring the lines between the three[1]. For instance, it is possible to do animations in both JavaScript and CSS. The problem with blurred distinctions is that web development becomes less organized. A second argument can be made that due to the fast growth of these languages, browsers fail to support the newest and greatest features. This forces developers to take precautions when designing web pages.

With the Silhouette framework, it keeps function areas more organized by clearly distinguishing between markup and styling through different APIs. It tries to promote best practices, such as not using in-line styling, by making these processes easier, and hiding away the bad practices. In addition, the methods will make sure to correct minor mistakes to prevent nasty surprises. Silhouette also implements solutions for tags that are less supported among browsers.

In addition to all this, the user will have access to all the OOP benefits that lets anyone have more control, automate, and format the code after any given standard. We think this alone is a good enough reason to work with markup in OOP. This also means that if the code is not valid, it will simply not compile, and the compiler will inform of the problem.

So to sum it up short, Silhouette acts like a gateway to make websites with OOP, while imposing best practices and removing the caveats found in standard HTML. It should make the overall process more enjoyable and creative, rather than a chore. People also don't need prior knowledge of HTML and CSS to be able to make websites through Silhouette.

## 1.1 Group Description

## 1.2 Delegation of Work

### Description of Work

Each students writes about scenarios they have contributed with... Workload...

# 2 Background

There is no hiding that there are plenty of existing web frameworks out on the internet already, each designed to cover certain needs. With regards to this, Silhouette also does not try to reinvent the wheel, but it uses a lot of the same tools found in these well established frameworks, along with some quality of life improvements.

People generally use web frameworks to gain access to additional features and to reuse existing code, as opposed to redoing it from scratch. It is important that the web framework increases the velocity of web development compared to doing it in plain HTML and CSS. A good example is the *JSF framework* (Java Server Faces). It has a unique feature where users design UI-components through high level abstractions that are converted into the appropriate representation of HTML depending on the device and version of the internet. This greatly reduces repeated code and means that users can worry less about the technical aspects.

Another important side of Frameworks is to establish a structure in the application. The *Spring framework* creates such a structure by abstracting POJOs (Plain Old Java Objects) into special components called beans. These components are designed to be flexible and adjustable and can be wired together with other beans. This way, users can manipulate each component by interacting with the beans. However, with Silhouette, we took a different approach using the builder pattern to account for all the different options when writing HTML and CSS. The pattern creates a structure in the code that consists of building components and appending them to each other. With Spring, the user builds a component through the factory pattern that initializes the beans by following given instructions, but we thought the builder pattern would be more flexible on the go:

Creating a paragraph and adding it to a div

```
Paragraph p = new Paragraph.Builder()
    .setText("Hello World")
    .build();
Container div = new Container.Builder()
    .addElements(p)
    .build();
```

**Vaadin** is probably the framework closest to Silhouette. It enables users to make web applications completely in Java, which means it does not require any prior knowledge to HTML, CSS or JavaScript altogether. Everything made with Vaadin is a component that are presented in layouts on the page. To add functionality, the user adds event listeners to the desired components. And lastly, to stylize the web app, Vaadin uses a theme roller called *Vaalo* that is similar to CSS but comes with some additional features, like auto-contrasting text color depending on the background color. It does the majority of what Silhouette wants to achieve, which is why it is the closest to Silhouette.

What separates Silhouette is the user of the builder pattern. In Silhouette, builder pattern is used for just about everything to account for all the different options when writing HTML or CSS. Therefore, POJOs won't hold.

## 3 Method

"A section to describe the method under which the framework has been created (can be shared). This section should contain a description for each of the stages given in the "Framework Design" lecture (API Design Specification, API Design, Implementation and User Testing). You should describe how you have worked, not what you produced, in this section."

### 3.1 API Design Specification

Forklar hvordan vi kom fram til Tankesett av hvordan API-et skal gå fram, høy-nivå og lav-nivå prinsipper

From the get-go we knew our framework needed to generate HTML somehow, and decided that it would be best to

From the get-go we knew that our framework needed to support both the declarative and stylistic sides of webdesign, so we started forming two bas

## 3.2 API Design

Forklar hvordan vi kom fram til Builder pattern... Hvordan API-et faktisk ser ut

When designing the APIs in Silhouette, we started writing client-code to see what would work the best. We knew it was important to preserve the advantages of OOP, giving the user control by making objects and manipulating them in various ways. This meant that users have the ability to generate blocks of HTML quickly with the tools we already know from OOP. We decided to write some client-code to see what would satisfy our requirements, and this would be one of the earliest iterations.

SHOW CLIENT-CODE HERE

The idea of x was important, and you can see that with the way it's set up and all. But another thing that we couldn't dismiss was the fact that we would get way too many classes if we had kept going the route of making one-to-one relations between classes and HTML tags, following the rule of 30 (Insert Source Here!), so we came up with a secondary solution. Say, for example, the user wants to make a video on their page. The video tag in HTML requires some smaller tags inside it that specifies file and filetypes and controls etc.. an we saw it as an opportunity to hide away these smaller tags within methods. This proved to be a good solution as it both cut down on the total amount of classes, and the user had less things to worry about but still achieve the same things. Eventually we settled on some one-to-one classes, and some methods to generate simpler HTML code...

### Design Decisions

Following best-practices for making APIs, for example, from

Later on once we got some feedback from other groups we came across another problem. Users expected to be able to chain multiple methods after eachother. But that was just not possible with how we originally structured our methods and classes, that's where builder pattern comes in. Builder pattern aided the framework in two areas: One, it served as a tool to hide the backend-implementation from the user, and two, it allowed for method-chaining which was exactly what we needed. This mean that previously, the user had to type:

SHOW CLIENT-CODE HERE

But now, with the builder-pattern in place, the user can do this:

SHOW CLIENT-CODE HERE

### Personal Decisions

## 3.3 Implementation

Implementation was done in various stages as we had many ideas for how we wanted the resulting framework to behave. After a lot of discussion with the group, we began writing client-code to help us visualize the API to get better ideas on how we got incorporate the implementation. After several supervisions with the lectruer, we were advised to us a builder-pattern, which works wonders for the type of tasks we want to do, however it makes implementing a bit of a chore, as this means usually our source-code will double. Getting to understand how this pattern works was not easy, since as we were writing our API-code we did not have a full understanding of the structure of method-calls/chaining, nor how to really instantiate these objects. This led to some early errors like giving each builder-class a specific name, and not implementing it correctly.

After getting to understand these things there are other issues that appear throughout. We have some global attributes that all objects in the API share, so it's logical to just write the method once and be done with it, however this does mean that casting between classes becomes necessary. This is possibly because we opted for internal static Builder classes. We felt that was the most logical and direct connection we could make between an objects and its builder.

Lastly, we needed to form a system to generate the actual. The big idea was to make some sort of compiler that puts strings together to make the necessary web files. It later became apparent that we needed one compiler for each API, as the logic and syntax are vastly different. As it stands, the compilers will either read html and css in the form of Strings, or plain java old objects (POJOs). Then once the user is done defining their pages and stylings, they will inform the respective compiler to perform a compile and generate the web files. If these files already exist, the compiler should just overwrite them to apply the new changes.

### **3.4 User Testing**

User testing started in the middle of the development of Silhouette, right after all the classes that was need to builde a HTML document (the logical structure part) and the same for the CSS part. At this time the code was just object oriented programming and it flet easy to use, but it was alot of unnecessary writing and to a degree hard to follow.

After the first user testing with another group, we didn't get alot of crtical feedback on whats "good" and "bad" excepted for the things that we knew about, but we got feedback on the logical structure part and that is one of the things we needed.

After some time we knew what we wanted the code ot looke like and how to simplify it more, and that is when we started to use "Builder pattern" instead of the heavy object coding we had previously. This made it more readable for the eyes and work with.

#### **Description of setup**

We sent a jar file containing the framework to group and started with some simple instructions. Firstly, they needed to start a new project and add our framework/library to their project from the project structure menu. Once this was done, they were ready to start coding inside the main method.

At first, we let them figure out the process themselves, but once they met issues and got stuck, we gave explanations to what each code block represented and guided them through the process. The next section will cover the resulting code and feedback from our testing.

#### **Frist user testing session**

Right after they had tested our framework and completed the setup, we asked them to give their initial reaction and opinion. They said, "It was pretty good once we got the hang of the class names" which was a relief to hear. It seemed to us that they found the APIs to be intuitive, as they quickly grew familiar with the main classes. We then followed up with a few questions regarding their experience, and they told us they could easily expand further on their website with enough practice.

#### **Second user testing session**

## **4 Design Process/Results**

### **4.1 Design Specification**

Forklar hva High Level Design Principles...

#### **Design Patterns**

**Builder Pattern** We were very quickly guided towards using the Builder pattern when we first had decided on the Framework-type. The Builder pattern, unlike the Abstract Factory pattern, is a lot more like a "Fluent interface", meaning it relies heavily on method cascading. The builder pattern is useful as it attempts to seperate the construction of an object from the way it is represented, this way we can use the same process to create many different looking objects that are fundementally similar.

We need to use the builder pattern when we want immutable objects. We want immutable classes because they have a wide range of benefits when we are developing an API. They are simple to use, they are synchronization safe, we only need one constructor, and they are great for maps(which we're using) to name a few.

The main problem with implementing the builder pattern is the sheer amount of extra code required. The lines will most likely more than double. However, those extra lines of code give us much more readable code and design flexibility. We exchange parameters in the constructor for much prettier method calls.

## 4.2 Project Structure (File Structure)

### Type Reference Documentation

[Link to type doc...](#)

## 4.3 Client Code

### Scenarios and Solutions

**Scenario 1** Make two rulesets, one that is a regular ruleset and one that is a grid ruleset. Give each ruleset a unique selector. For the regular ruleset, add a blue background and change the text color to #32a852. For the grid, define three columns and two rows of varied size. Lastly, apply both of the rulesets to a Container of type "header".

#### Scenario 1 - Proposed Solution

```
RuleSet color = new RuleSet(".color");
color.addRule("background-color", "blue");
color.addRule("color", Color.Hex(#32a852));

Grid grid = new Grid("#grid");
grid.setColumns("1fr", "100px", "2em");
grid.setRows("50%", "120px");

Container myHeader = new Container("header");
myHeader.applySelector(".color");
myHeader.applySelector("#grid");
```

**Scenario 2** Make a table whose size changes dynamically, add values to the header row and add values to the rest of the rows as they are generated. Apply a class to the table and set a header color for the table.

#### Scenario 2 - Proposed Solution

```
Table table = new Table();
int col = headerArray.length;
int row = array.length;
RuleSet color = new RuleSet(".tableClass");
color.addRule("background-color", Color.RGB(255,255,255));
ColGroup headerCol = new ColGroup();

table.setSize(row, col);
table.applySelector(".tableClass");

for(int i= 0; i<row; i++){
    for(int j= 0; j<col; j++){
        while(i<1){
            table.insertHead(i,j, headerArray[j]);
        }
    }
}
```

```

    }
    table.insertValue(i,j, array[j])
    if(i<1){
        headerCol.addCol(col, color.toString());
    }
}
}

```

**Scenario 3** Make a frontpage and two article pages and share a common stylesheet between them. Add a simple ruleset that targets the body element and give it a lightgreen background to see that the changes have applied to all the pages. Lastly, generate the code.

#### Scenario 3 - Proposed Solution

```

HTML myWebsite = new HTML("example website");

Page frontPage = new Page("frontpage", "My Front Page");
Page articlePage1 = new Page("article", "My First Article");
Page articlePage2 = new Page("article", "My Second Article");

StyleSheet style = new StyleSheet("style.css");

RuleSet bg = new RuleSet("body");
bg.addRule("background-color", "lightgreen");

style.append(bg);

website.link("style.css");
website.addPage(frontPage);
website.addPage(articlePage1);
website.addPage(articlePage2);

website.initialize();

```

**Scenario 4** Make a simple form consisting of three input fields for username, password and submit. Label each of the fields accordingly and give the fieldset a legend of "Login Credentials".

#### Scenario 4 - Proposed Solution

```

FieldSet myFields = new FieldSet();

Input username = new Input("text");
username.addLabel("Type Username");

Input password = new Input("password");
password.addLabel("Type Password");

Input submit = new Input("submit");

myFields.addFields(username, password, submit);
myFields.addLegend("Login Credentials");

```

**Scenario 5** Make a Container of type header and give it a navigation bar with links to all pages previously created. Make the navigation bar into a CSS flexbox so that all anchors be laid horizontally and justify the content to be "space-around".

#### Scenario 5 - Proposed Solution

```

Container header = new Container("header");

```



```

Container nav = new Container("nav");

Anchor link1 = new AnchorBuilder("Front Page",
    "frontpage.html").build();
Anchor link2 = new AnchorBuilder("First Article",
    "article1.html").build();
Anchor link3 = new AnchorBuilder("Second Article",
    "article1.html").build();

nav.addElements(link1, link2, link3);

FlexBox flexbox = new FlexBox("header nav");
flexbox.setFlexDirection("row");
flexbox.setJustifyContent("space-around");

```

**Scenario 6** Make a simple article page about yourself. Write your name in a level 2 heading at the top of the article, then add an unordered list containing key points about you. Lastly, add your email in bold text below the text.

**Scenario 7** Display a local video on your website and make sure that it supports at least three different file formats.

**Scenario 8** Make a questionnaire consisting of two questions. Make it so the first question has two possibilities and one correct answer, and so the second question has three possibilities with multiple correct answers. (Radio and Checkbox)

**Scenario 9** Remove underline for all anchors inside nav Containers and change the text color to red on mouse hover for each anchor.

**Scenario 10** Give the header a fixed gradient background from blue to red (right).

**Scenario 11** Define a grid that serves as a page layout, and give it three columns and one row with equally distributed length. Make sure to wrap them under each other once the screen is narrower than 1000px, such that there are one column and three rows.

## 4.4 User Testing

### The Code

### Feedback

## 4.5 Revised API

# 5 Resulting Framework

Framvisning av det ferdie API-et

## 6 Discussion

## 7 References

[1] - <https://pointersgonewild.com/2015/04/16/the-problem-with-html/>