Utility-First CSS with Tailwind

What is Utility-First CSS?

In the past few years, *utility-first CSS* has been a popular trend within the front-end landscape. You might have seen it mentioned alongside a tool called Tailwind CSS.

So what is "utility"?

What problem does this utility-first approach solve?

How does Tailwind fit into this?

And most importantly: how do you apply Tailwind in a Vue.js application?

We're going to answer all of that in this course.

What is Utility-first CSS?

Let's start off with some classic CSS/HTML code for demonstration:

CSS

```
.container {
  background-color: lightgray;
  padding: 20px;
  width: 500px;
  margin: 20px;
}

.heading {
  font-size: 20px;
  font-family: Arial;
  margin: 0;
}
```

HTML

```
<div class="container">
  <h1 class="heading">Utility-first CSS?</h1>
</div>
```

With the utility-first approach, the above could be refactored into something like this:

CSS

```
.bg-light-gray { background-color: lightgray; }
.p-20px { padding: 20px; }
.w-500px { width: 500px; }
.m-20px { margin: 20px; }
.font-20px { font-size: 20px; }
```

```
.arial { font-family: Arial; }
.m-0 { margin: 0; }
```

HTML

```
<div class="bg-light-gray p-20px w-500px m-20px">
  <h1 class="font-20px arial m-0">Utility-first CSS?</h1>
</div>
```

As you can see, it's simply replacing the *high-level classes*, such as `container` and `heading`, with the *low-level classes*, such as `bg-light-gray` and `p-20px`.

These low-level classes are called *utility classes*. You can think of them as low-level design commodities with predictable names. That's why they're also called *atomic classes.*

For example, an HTML element is using a utility class called `h-200px`.

```
<div class="h-200px"></div>
```

Without even looking at the the CSS of this class, I would have guessed it's probably something like this:

```
.h-200px {
  height: 200px
}
```

And that would be accurate.

With this kind of the naming convention, you rarely have a need to modify the CSS of a utility class. If you need to change the look and feel of your app, you simply just modify the HTML to add/remove classes. You should not be modifying the CSS of the class.

For example, if I no longer need a background color, I would just simply remove the `bg-light-gray` class like this:

```
<div class="p-20px w-500px m-20px">
  <h1 class="font-20px arial m-0">Utility-first CSS?</h1>
</div>
```

Whereas in the traditional setup, I would have to change the content of the `container` class like this:

```
.container {
  /* background-color: lightgray; */
  padding: 20px;
  width: 500px;
  margin: 20px;
}
```

At first, this doesn't seem like a huge difference. It's just a matter of modifying the CSS and modifying the HTML. But this has a huge implication in terms of maintainability.

If you change by modifying the CSS, you need extra care to make sure that the change doesn't affect anything unintended. But if you change by modifying only the HTML, you know for sure that the change is only affecting the very HTML that you modify.

The main benefit of using utility classes is simply this: *you don't have to worry about unintended consequences of adding/removing styles.*

For Vue.js developers. this might not be such a big deal if you're already using `<style scoped>`. But if you have plenty of CSS shared across different components, the certainty of knowing what your change will affect is still very helpful. The utility-first approach comes with a unique philosophy for code sharing that will enforce this certainty.

Okay, but why do we need Tailwind?

Notice that in the example above, we didn't even have to use any framework. So why do we need Tailwind, or any similar framework for that matter?

First, someone has to create all of these CSS Utility classes (hundreds of them), so either you create them yourself, or you can use an existing solution with widely-adopted naming conventions.

Second, a framework provides more useful features, such as Just-In-Time compilation to create dynamic classes, and responsive design features.

We'll talk more about these later in the course.

Lastly, a framework can be customized with your own configuration settings.

What is this course about?

This course is structured around refactoring an existing Vue.js application. Along the way, you'll learn the *basics* of using Tailwind. And we'll also get into advanced topics such as *Responsive Design*, *Conditional Style*, and *Configurations*. We'll wrap up the course with the final lessons of *Custom Class* and *Code-Reuse Patterns*.

At the end, you'll be able to apply Tailwind in a Vue.js project.

If that sounds good to you, let's move to the next lesson where we'll be *setting up Tailwind* for a Vue.js application.