Revisiting design systems

Design systems are an interesting because of how they scale design to multiple teams and multiple sites/products while still providing a consitent and recognizable brand.

For a full introduction to design systems, see Introducing Design Systems

The work I do is not always conductive to creating a design system but it's always good to know about them so we can jump in without having to start completely from scratch.

That said we'll look at three areas:

- Building a basic component using just HTML, and CSS
- Building the same component using existing design systems
    - Material Design from Google
    - Atomic Design from Brad Frost
- Building the component using Web Components
    - Plain Web Components
    - Polymer

No Design Systems

I want to design a card component where we can control the direction (vertical or horizontal), the dimensions (height and width) and the border. The component should also provide ways to layout its children using Flex.

The CSS looks something like this:

```
:root {
  --card-border-color: #000;
  --card-border-width: 1px;
  --card-border-style: solid;
  --card-display: flex;
  --card-direction: column;
  --card-width: 400px;
  --card-height: 300px;
```

```
  --card-padding: 1em;
}

.card {
  display: var(--card-display);
  flex-flow: var(--card-direction);
  border: var(--card-border-width) var(--card-border-style) var(--card-bo
  height: var(--card-height);
  width: var(--card-width);
}

.card-title {
  color: #fff;
  background-color: #000;
}

.card-content {
  padding: var(--card-padding);
}
```

The idea is that by changing the value of the variable declared in the `:root` element we change the way that all card elements will look on the page. I choose the `:root` element rather than `html` because `:root` has higher specificity and it would override any declarations in HTML rather than having to depend on the way properties were declared every time.

Let's saay that we want to create a horizontal card. All we have to do is redefine the variables for the elements we need to change, something like the following example:

In this example we make the card layout its children in a row, rather than a column, we also change the dimensions of the card but we keep all other values the same as our `:root` declaration.

```
:root {
  --card-border-color: #000;
  --card-border-width: 1px;
  --card-border-style: solid;
```

```
  --card-display: flex;
  --card-direction: column;
  --card-width: 400px;
  --card-height: 300px;
  --card-padding: 1em;
}

.card2 {
  --card-border-color: #000;
  --card-direction: row;
  --card-height: 300px;
  --card-width: 800px;

  border: var(--card-border-width) var(--card-border-style) var(--card-bo
  display: var(--card-display);
  flex-flow: var(--card-direction);
  height: var(--card-height);
  width: var(--card-width);

  .card-title {
    width: 10em;
  }
}
```

This is just an initial step, we could further refine the card component by providing deefaults in case we want to add an image instead of text on the title, but it'll do as the basic example of what we want to accomplish.

One other thing that need to happen for this card to become an element is to document the ways in which you can customize the component... ideally with full examples that can be copy and pasted to be used.

Design System

Now that we have a basic idea of what we want to do, let's look at different ways in which design systems can make the process easier, less painful or both.

Material Design

The new iteration of material design has its good and bad aspects.

The good is that the new version is much easier to customize and make look “less like a Google product” than the original

The bad is that Material Design now reuires SCSS and ES6 for development. You can’t just drop the Material Design components into an existing project if the project doesn’t use a build system or SASS to process the CSS.

Since I already use both ES6 and SCSS for my projects, I think the drawbacks, if you can call them that, are worth the investment.

As you can see the code depends hevily on classes starting with `mdc-`. The reason why will become clearer when we look at the SCSS code that styles the element.

```
<div class="mdc-card mdc-card--outlined my-card ">
  <div class="mdc-card__media mdc-card__media--square">
    <div class="mdc-card__media-content">Title</div>
  </div>
  <p>Content goes here</p>

  <div class="mdc-card__actions">
    <div class="mdc-card__action-buttons">
      <button class="
        mdc-button
        mdc-card__action
        mdc-card__action--button">Action 1</button>
      <button class="mdc-button
        mdc-card__action
        mdc-card__action--button">Action 2</button>
    </div>
  </div>
</div>
```

Material design is based on SCSS so to add the components’ styles we just import them using SASS `@import` syntax… with a twist.

```
@import '@mater'@material/card/mdc-card'@import '@material/button/mdc-butt

body {
  font-family: Roboto, sans-serif;
}

.mdc-card {
  margin: 2em auto;
  height: 350px;
  width: 350px;
}

.mdc-card__media ackground-image: url('../images/sunrise.jpg');

  img {'../images/sunrise.jpg'img {
    width: 350px;
  }
}
```

Because we are going into the `node_modules` directory we have to call the sass command with a special flag to include the modules directory into the path SASS searches. The command is:

```
sass -I node_modules/ sass/:css/
```

This works in version 1.13.0, the current Dart version of SASS.

Atomic Design

Atomic Design, conceived by Brad Frost, and explained in the book of the same name is a way to break a page into its components down to the single HTML element. The introduction of the book explains it like so:

> And then Brad Frost the front-end developer started talking like Brad Frost the chemist. He talked about atoms and molecules and organisms—about how large pieces of design can be broken down into smaller ones and even recombined

> into different large pieces. Instead of visualizing the finished recipe for the design, in other words, he was showing us the ingredients. And we lit up: this was a shift in perspective, a way to move away from conceiving a website design as a collection of static page templates, and instead as a dynamic system of adaptable components. It was an inspired way to approach this responsive website—and all responsive projects for that matter.
>
> Atomic Design — Introduction.

Atomic design uses three structures to build pages: Atoms, molecules and organisms.

Think of individual HTML elements as **atoms**, the basic building block of a web page. You can't break down an atom into smaller parts without loosing functionality and the effect the element has on the page.

Look at the table below to get an idea of what the possible atoms of a web page are.

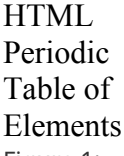

Figure 1: HTML Periodic Table of Elements

These atoms may be the plain elements or styles for specific theme or molecule designs.

The next step up is a **molecules**. We create a molecule when we combine two or more atoms to perform a given task.

Organisms are groups of molecules functioning together as a unit. These structures can be as small or as large as you need them to be. We can use these organisms to bbuild pages and applications.

For more infomation about Atomic Design, check the the book.

For the card organism we'll take the same card declaration that we used when

we created the card component.

> We have two molecules making the card organism:

- A `card-title` molecule that contains title and, optionally, an image inside
- A `card-content` molecule that contains the text content

Each of these molecules can have any number of atoms inside of them. We can choose to document individual atoms or to just to document the styles for colors, links and typography, among others.

```
<div class="card">

  <div class="card-title">
    <h1>Card Title</h1>
  </div>

  <div class="card-content">
    <p>I want to design a card component</p>
  </div>

</div>
```

The styling is the same as for the no-design-system using CSS variables to handle different alternatives for the content and some effects on the text and background colors of the molecules.

```
:root {
  --card-border-color: #000;
  --card-border-width: 1px;
  --card-border-style: solid;
  --card-display: flex;
  --card-direction: column;
  --card-width: 400px;
  --card-height: 300px;
  --card-padding: 1em;
  --card-title-color: #fff;
  --card-title-background-color: #000;
```

```
}

body {
  font-family: Roboto, sans-serif;
  width: 80%;
  margin: 0 auto;
}

.card {
  display: var(--card-display);
  flex-flow: var(--card-direction);
  border: var(--card-border-width) var(--card-border-style) var(--card-bo
  height: var(--card-height);
  width: var(--card-width);
}

.card-title {
  color: var(--card-title-color);
  background-color: var(--card-title-background-color);
}

.card-content {
  padding: var(--card-padding);
}
```

Web Components As Design Systems

Web components, to me, address some of the shortcomings of design systems.

They encapsulate styles so there's no bleeding between the containing page and the components, this means that styles accross different components can have the same name as they are exclusive to the element... We'll explore advantages and disadvantages of the model.

Vanilla JS

The vanilla Javascript version consists of two parts: The CSS and HTML that we'll stamp every time we create an element of this kind.

We use CSS variables that we'll define in the HTML container for the element.

```
let template = document.createElement('template');
template.innerHTML = `<style>
  .card {
    width: 80% ;
    margin: 0 auto;
    display: var(--card-display);
    flex-flow: var(--card-direction);
    border: var(--card-border-width)
            var(--card-border-style)
            var(--card-border-color);
    height: var(--card-height);
    width: var(--card-width);
  }

  .card-title {
    color: var(--card-title-color);
    background-color: var(--card-title-background-color);
  }

  .card-content {
    padding: var(--card-padding);
  }</style>
```

The HTML portion of the template uses slots to reflect elements of the 'light side' declaration to places in the template.

The slots also provide additional ways to style the content.

We'll look at slots in more detail on the slotted content when we look at the implementation of the card element.

```
  <div class="card">
    <div class="card-title">
      <slot id="title" name="title"><h1>Title</h1></slot>
    </div>

    <div class="card-content">
      <slot id="content" name="content"><p>Content</p></slot>
    </div>
  </div>
`;
```

The second part of the `card-element` element is the definition of the custom element itself. We do it by creating a class that extends `HTMLElement`, the base class for all HTML elements.

We then build the constructor where we dwothe following:

1. Call the parent's class constructor using `super()`
2. create a shadow root for the element
3. Append the content of the template to the shadow root we just created

Finally we use the `define` method of the customElements registry to associate a tag name with the component we just defined.

```
class cardComponent extends HTMLElement {
  constructor() {
    // Allways call super() first
    super();
    let root = this.attachShadow({
      mode: 'open',
    });
    root.appendChild(template.content.cloneNode(true));
  }
}

window.customE'open's.define('card-element', cardComponent);
```

With that we're ready to implement the element.

In the head of the element I'll place a link to Google Fonts and a set of variable declarations for the `:root` element.

```
<link rel="stylesheet"
href="https://fonts.googleapis.com/css?family=Roboto:400,400i,700,700i">
<style><style>
:root {
  font-family: Roboto, sans-serif;
  --card-border-color: #000;
  --card-border-width: 1px;
  --card-border-style: solid;
  --card-display: flex;
  --card-direction: column;
  --card-width: 400px;
  --card-height: 300px;
  --card-padding: 1em;
  --card-title-color: #fff;
  --card-title-background-color: #000;
  }
</style>
```

We then import the card element as a module using the `type=module` attribute of the script tag.

```
<script type="module" src="wc-card.js"></script>
```

In the body of the page we stamp out multiple versions of our `card-element` element. The first one is just the element. This will create a card with only fallback content.

```
<card-element></card-element>
```

The second element uses the slots we defined in the element's template. We can add as much content to each slot as we want... as far as I know there is no limit.

```
<card-element>
```

```
  <div slot="title">
    <h1>Card Demo #2</h1>
  </div>
  <div slot="content">
    <p>Some content that is different from our default</p>
  </div>
</card-element>
```

Polymer

Polymer adds syntactic sugar on top of web components with additional functionality and syntactic sugar. The idea is to drop these components.

Building a Polymer element is very similar to building a vanilla Javascript web component. Instead of extending from `HTMLElement` we extend from `PolymerElement`, the base class for Polymer.

Since we're not using any binding in the element, we can use use a template like the one we created for the vanilla components.

The main difference is that we must import the `html` tagged template literal and `PolymerElement` base class before we can use them anywhere.

```
import { html, PolymerElement } from '@polymer/polymer/polymer-element.js

class MyCard extends PolymerElement {
  static get template() {
    return html`
  <style>
  :host {
    width: 80% ;
    margin: 0 auto;
    font-family: Roboto, sans-serif;
    --card-border-color: #000;
    --card-border-width: 1px;
    --card-border-style: solid;
    --card-display: flex;
    --card-direction: column;
```

```
    --card-width: 400px;
    --card-height: 300px;
    --card-padding: 1em;
    --card-title-color: #fff;
    --card-title-background-color: #000;
  }

  /* Rest of the CSS Removed for brevity */
  </style>

  <div class="card">
    <div class="card-title">
      <slot id="title" name="title">
        <h1>Title</h1>
      </slot>
    </div>

    <div class="card-content">
      <slot id="content" name="content">
        <p>Content</p>
      </slot>
    </div>
  </div>
`;
  }
}

window.customElements.define('my-card', MyCard);
```

The host document is almost identical to the plain vanilla web component. We add the following elements to the head of the element.

We include the wecomponents polifill to make sure that the component works in browsers without native support.

```
<link rel="stylesheet" href="https://fonts.googleapis.com/css?
family=Roboto:400,400i,700,700i">
```

```
<style><style>>
<script src="../node_modules/@webcomponents/webcomponentsjs/webcomponents-
<script type="mo<script src="../node_modules/@webcomponents/webcomponents
```

The Polymer version of the element is identical to the vanilla component. We insert the following into the body of the page.

```
<my-card>
  <div slot="title">
    <h1>Card Demo #2</h1>
  </div>
  <div slot="content">
    <p>Some content that is different from our default
       so we can be sure that the slots are working as
       intended and we are not pulling the default content</p>
  </div>
</my-card>
```

Documenting Your Design System

Now that we've decided to use a design system and we've chosen the components that we want to share. Now we need to document it.

Fractal provides tools and structure for design systems. We will not address the specifics of Fractal or any other tools to document.

I chose Fractal because it gives you the flexibility to use components with copy/paste or incorporating into a build system.

Fractal's downside is the documentation. It leaves a lot to be desired when it comes to documenting the process to incorporate the tool into other applications, the API and mount points for the application.

There is ongoing work to improve and update Fractal. See issue 449 in the Fractal Github repository for more details as to current status.

Looking forward: AI Design

Systems?

An interesting take on design systems is what would it look like if we can get working code from a hand-drawn wireframe?

There are projects in early prototype stages that will let you do this.

Pix2Code has published both a paper and the corresponding Github repo explaining their work.

According to the paper:

> Transforming a graphical user interface screenshot created by a designer into computer code is a typical task conducted by a developer in order to build customized software, websites, and mobile applications. In this paper, we show that deep learning methods can be leveraged to train a model end-to-end to automatically generate code from a single input image with over 77% of accuracy for three different platforms (i.e. iOS, Android and web-based technologies).
>
> pix2code: Generating Code from a Graphical User Interface Screenshot

This would mean that the aproximation we craft can be translated to code that can be further iterated on or, if it meets the team's quality, can be put into production.

In an interesting twist AirBnB is alos looking at using Machine Learning to translate mockups into code that can be further moved up and down the design and development chain.

There is no actual code prototype but the idea is intriguing enough to merit further attention and research.

Links and Resources

- AirBnB
  - The Way We Build
  - Building a Visual Language
- Atomic Design

        - Book
- Google
    - Google Design Library
- Design Systems Repo
    - Repo
    - Gallery
    - Articles
- Mailchimp
    - Content Style Guide
- Web Components
    - Google Developers
        - Custom Elements v1: Reusable Web Components
        - Shadow DOM v1: Self-Contained Web Components
    - The Holy Grail Of Reusable Components: Custom Elements, Shadow DOM, And NPM
    - Polymer
    - Wired Elements
- Sites Using Fractal
    - City of Boston
    - City of Ghent
    - 24Ways.org
    - Clearleft
    - Perch CMS
    - US Web Design System
    - Kanbasu
    - Liip
    - Eurostar
- Machine Learning and Design Systems
    - Pix2code
        - pix2code: Generating Code from a Graphical User Interface Screenshot
        - Github Repo
    - Turning Design Mockups Into Code With Deep Learning
        - Sketching Interfaces