Chapter 1: Web Components: First Steps

Before we dive into Vue, I wanted to quickly touch on Web Components. You might be wondering what they are and how they relate to Vue. Web Components are a set of APIs that allow you to create reusable and modular components for the web.

They're made up of three main technologies:

- Custom Elements
- Shadow DOM
- HTML Templates

Don't worry if you haven't heard of these before – you don't need to know about them to use Vue. However, it's always good to have a basic understanding of the technologies you're working with, and learning about Web Components can give you some insight into why certain features in Vue work the way they do. Plus, it's always good to have a few extra tools in your toolbelt, right?

1.1 New Era

Have you ever wished to grab a pre-made component and drop it into your app to add some much-needed functionality for your users?

Well, that's exactly what Components are all about!

In the past, we've had components in web development, but they often required dependencies like jQuery, Prototype or AngularJS, etc.

Web Components aim to solve this problem by providing reusable and encapsulated components that work across different frameworks.
They rely on a set of emerging standards that are fully supported by all browsers, it's still an exciting topic to explore – even if it might be a few years before we can use them to their full potential.

The emerging standard is made up of three main technologies that we have mentioned above: Custom Elements, Shadow DOM, and HTML Templates. Let’s explore these standards together.

1.2 Custom Elements

Custom elements allow developers to create their unique HTML tags, like <my-raccoon></my-raccoon>. This new web standard provides guidelines on how to declare these elements and how to extend or modify existing elements. It also outlines how to design the API for these custom elements.

To create a custom element, you can use the customElements.define function:

```
class Raccoon extends HTMLElement {
      constructor() {
            super();
            console.log("Hey, from Raccoon!");
      }
}
customElements.define('my-raccoon', Raccoon);
```

Once you have defined a custom element, you can use it in your HTML code just like any other element:

```
<my-raccoon></my-raccoon>
```

It's important to include a dash in the name of your custom element so that the browser knows it's a special, custom-defined tag. Your custom element can have its properties, methods, and even a template. Additionally, there are lifecycle callbacks available, which allow you to run certain code at specific times, such as when the element is added or removed from the page, or when one of its attributes is changed. For example, the custom-raccoon element could display an adorable image or just the name of a raccoon.

If you inspect the DOM, you'll see your custom element and any information you provided for it. This means that your app's CSS and JavaScript logic could potentially have unintended effects on your component. To prevent this, it's common to encapsulate the template in something called a Shadow DOM, which hides it from the rest of the page. If you inspect the DOM, you'll only see the custom element's tag, like <custom-raccoon></custom-raccoon>, and not its internal content or structure.

1.3 Shadow DOM

Shadow DOM is a way to keep the elements of a component separate from the rest of the code on your webpage. It allows you to create a hidden DOM tree that is attached to an element in the regular DOM tree and manipulate it just like the regular DOM. This separation can prevent your component's styles and JavaScript from affecting the rest of your app, and vice versa.

Shadow DOM has some specific terms to be aware of:

- **Shadow host**: The regular DOM node that the shadow DOM is attached to
- **Shadow tree:** The DOM tree inside the shadow DOM
- **Shadow boundary:** The point where the shadow DOM ends and the regular DOM begins
- **Shadow root**: The root node of the shadow tree

Shadow DOM has been used by browsers for a while to isolate the internal elements of certain elements, like the controls on an <video> element.
The shadow DOM specification now allows you to manipulate the shadow DOM of your own custom elements.

Let's now modify our Raccoon class that we have previously created

```
class Raccoon extends HTMLElement {
      constructor() {
            super();
            const shadow = this.attachShadow({ mode: 'open' });
            const title = document.createElement('p');
            title.textContent = 'I'm the Raccoon!';
            shadow.appendChild(title);
      }
}
```

If you take a look now, you'll see...

```
<my-raccoon>
 #shadow-root (open)
 <p>I'm the Raccoon!</p>
</my-raccoon>
```

The Shadow DOM acts like a barrier and protects the visual aspect of the component from being changed, even if you try to add style to the <p> elements.

Instead of using a string as a template for your web component, it's generally considered a best practice to use the <template> element. So let's dive into the next section where we will talk about the Template.

1.4 Template

When you use an <template> element in your HTML, the contents of the template are not immediately visible in your browser. Instead, the template is intended to be used as a blueprint for creating new elements on your page. Think of it like a mold for making copies.

The things you put inside a template, such as scripts or images, will not be activated when the template is first loaded. They are essentially "inert" or dormant until the template is used to create a new element. And, because the contents of a template are not part of the main document, you can't interact with them using normal JavaScript methods like getElementById().

The good thing is, you can place a template anywhere on your page, and it won't affect the rest of your HTML. This allows you to have a reusable piece of code that you can use over and over again, without having to worry about it conflicting with other elements on your page.

When working with templates in HTML, it's important to understand the difference between moving the content and cloning it. If you simply move the content out of the template without cloning it, the template will become empty and you won't be able to use it again.

On the other hand, if you clone the content before moving it, you'll be creating a copy of the template's content. This allows you to keep the original template intact, so you can use it to create more elements in the future. Think of it like copying a file on your computer, where the original file remains unchanged, and you're working with the copy.

It's important to note that when you clone an element, it's a new instance and changes to the clone will not affect the original template. Also when you clone an element any event handlers are not copied, so if you want them to work with the cloned elements you need to attach the event handlers again.

In summary, if you want to be able to continue using a template, it's a good idea to clone its content before moving it out of the template. This way, you can keep the original template for future use, and work with the clone without affecting the original.

```
<template id="raccoon">
 <style>
  p{ font-weight: bold; }
 </style>
 <p>I'm the Raccoon!</p>
</template>
```

Also let's update the Raccoon class to clone the template:

```
class Raccoon extends HTMLElement {
      constructor() {
            super();
            const template = document.querySelector('#raccoon');
            const cloned = document.importNode(template.content, true);
            const shadow = this.attachShadow({ mode: 'open' });
            shadow.appendChild(cloned);
      }
}
```

Conclusion

It's important to keep in mind that this introduction is just the tip of the iceberg when it comes to Web Components. As you continue to read and learn, you'll likely encounter more concepts and features that are part of the technology.
Web Components are a powerful and versatile way to build reusable and interoperable UI components on the web. Although Web Components are already supported in modern browsers, it's always a good idea to include a polyfill library to ensure compatibility with older browsers.

If you're using Vue, it's possible to use Web Components in conjunction with Vue components and export Vue components as Web Components.

If you're looking for resources to help you get started with Web Components, the website https://www.webcomponents.org is a great place to find polyfills, components, and information about browser support.

Overall, Web Components are a great way to build scalable, reusable, and interoperable code, and well worth exploring as a developer.