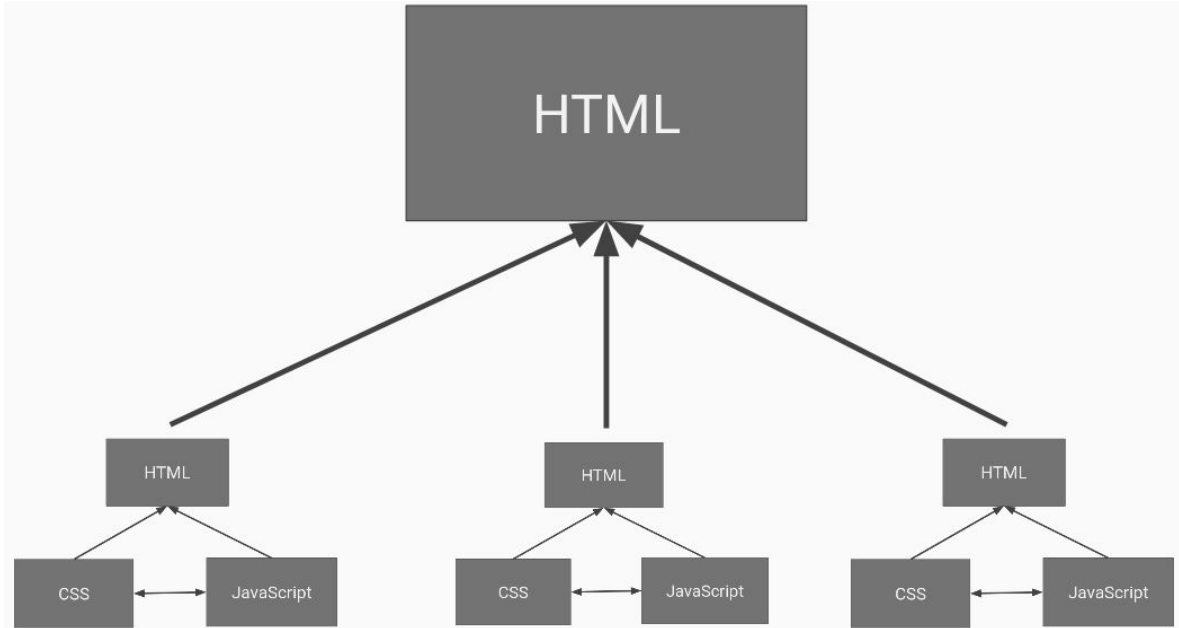


MODULE 3

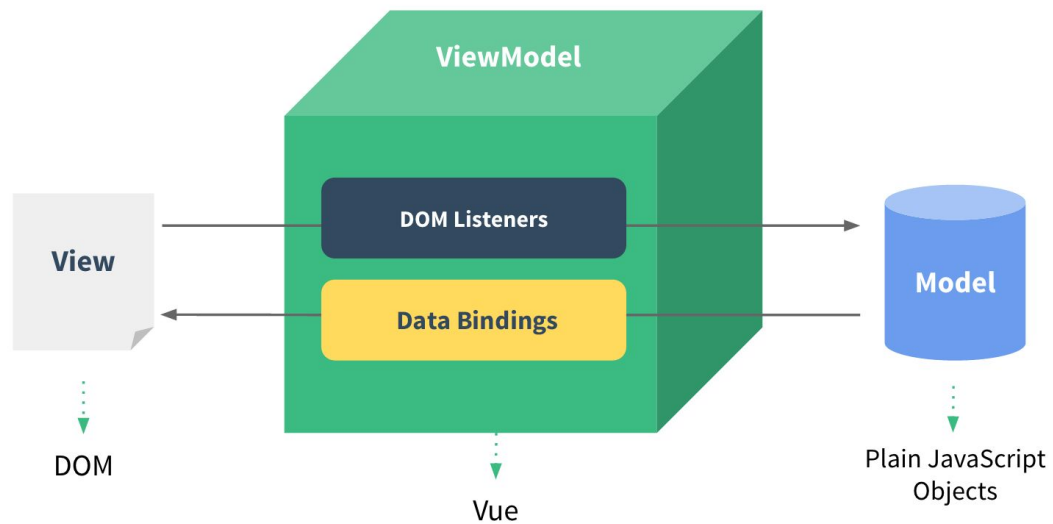
WEEK 3 - REVIEW

# JAVASCRIPT USING COMPONENTS

- HTML page
  - Main content
  - Structure
- Components
  - HTML
  - CSS
  - JavaScript
  - Act as single "plug-and-play" front-end element.
  - Simplify creation and maintenance



# VUE.JS DESIGN



# ANATOMY OF A VUE.JS COMPONENT

```
<template>
  <div class="main">
    <h2>Product Reviews for {{ name }}</h2>
    <p class="description">{{ description }}</p>
  </div>
</template>

<script>
export default {
  name: 'product-review',
  data() {
    return {
      name: 'Cigar Parties for Dummies',
      description: 'Host and plan the perfect cigar party for all of y
    }
  }
}
</script>

<style scoped>
div.main {
  margin: 1rem 0;
}
</style>
```

A VUE component is made up of three parts:

- The **<template>** section which contains HTML elements.
- The **<script>** section contains the JavaScript code.
- The **<style>** section which contains CSS styling.

What the user sees on the screen is defined mostly by the HTML elements created in the template section and any CSS styles applied in the style section.

The behavior of the component is defined by what's in the script section.

# APP.VUE: THE MAIN PAGE

```
<template>
  <div id="app">
    <product-review></product-review>
    
    <HelloWorld msg="Welcome to Your Vue.js App"/>
  </div>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'
import ProductReview from './components/ProductReview.vue'

export default {
  name: 'App',
  components: {
    HelloWorld,
    ProductReview
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

Here we have integrated both components into the main App.vue file. Note the following checklist:

- You are rendering the components in the `<template>`
- In `<script>` the components have been imported
- In `<script>` ,the components object is populated with the two component names.

# VUE METHODS

- Vue methods are similar to a function or method in other languages - they are called when needed, optionally taking in parameters and possibly providing some kind of output.
- Just like the **computed** section, the **methods** section is JavaScript and goes inside the **<script>** section.

# VUE METHODS VS. COMPUTED PROPERTIES

- Computed properties:
  - They return values based on your component's internal data (they are derived).
  - They are cached. If the value isn't different every time computed properties are a better choice.
- Methods are executed only when called and can execute code based on input parameters

# THE V-ON DIRECTIVE

- The **v-on:** directive acts similarly to an eventListener in vanilla JavaScript
- You specify the event that you want an element to respond to using the **v-on:** syntax:

```
<input type="button" value="Do Something" v-on:click="handleButtonClick" />
```

**v-on:** for click event

**handleButtonClick()** method will be called when a click event occurs.



# PREVENT AND STOP

- The `v-on:` directive can be modified to add the `preventDefault` and `stopPropagation` behaviors:
  - `v-on:submit.prevent="doFormSubmit"`
  - `v-on:submit.stop="doFormSubmit"`
  - Can be chained:
    - `v-on:submit.stop.prevent="doFormSubmit"`

# V-SHOW VS. V-IF

- **v-show** can be used to show/hide element
  - **v-show="isSelected"** will show element if **isSelected** is **true**, hide otherwise
- **v-if** can be used to include/remove element in/from DOM
  - **v-if="isSelected"** will include element in DOM if **isSelected** is **true**, will remove otherwise.

# SINGLE RESPONSIBILITY PRINCIPLE

- The Single Responsibility Principle (SRP) states that each component in the application should only handle one job.
- If a component isn't trying to handle an entire dashboard but is instead focused on just one graph on that dashboard, that component is easier to test, easier to maintain, and easier to reuse in another context..

# PASSING DATA TO CHILD COMPONENTS

- Components can use other Components.
- A Component can expect a property to be passed to it from another component by specifying the property in the **props** property.
  - **prop: ['posts']**
- Properties passed via **props** can be used like any properties defined in the Component itself.
- Other Components can pass data to a Component with a **props** property by binding an attribute with the name of the expected property:
  - **<blog-posts v-bind:posts="posts"></blog-posts>**

# PROP NAMES

- Multi-word **props** should be defined using camelCase
  - **props: ['blogPosts']**
- When passing an attribute via **v-bind**, multi-word props should be specified in kebab case:
  - **<blog-posts v-bind:blog-posts="posts"></blog-posts>**

# COMPONENT COMMUNICATION USING VUEX

- Vuex is a **state management pattern** and library for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the **state** (data) can only be **mutated** (changed) in a predictable fashion.
- state and state management refer to data within an application and how it is managed.
- Vuex data store is contained in a `/src/store/index.js` file.
- If more than one Component needs access to the same data, this is it would be stored.

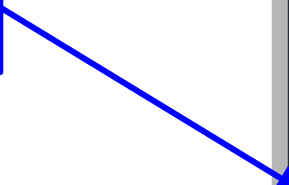
# ACCESSING DATA FROM THE STORE

- Data in the store can be accessed via the `$store` object in a Component.
- Since the data portion of the store is in the state property, properties can be accessed via `$store.state`
  - `$store.state.posts`
- As with properties, to access the store in your methods and computed properties you have to use this:
  - `this.$store.state.posts`

# VUEX MUTATIONS

- The only way to change state in a Vuex store is by committing a mutation.
- Mutations are defined in the mutations object of the store.

Mutations will be defined here.



```
export default new Vuex.Store({
  state: {
    posts: [
      {
        id: 1,
        title: 'My First Post',
        content: '<p>This is my first post</p>'
      },
      {
        id: 2,
        title: 'My Second Post',
        content: '<p>This is my second post</p>'
      }
    ]
  },
  mutations: {}, // MUTATIONS GO HERE
  actions: {},
  modules: {}
});
```



# DEFINING MUTATIONS

- To define a mutation that adds a new post to your list of posts, you'd start by creating a function called `ADD_POST()`. The function is where you perform state modifications, and it receives the state as the first argument.

```
mutations: {  
  ADD_POST(state) {  
    state.posts.push({  
      id: 3,  
      title: 'My Third Post',  
      content: '<p>This is my third post</p>'  
    });  
  }  
},
```

# USING MUTATIONS

- Mutation handlers can't be called directly.
- To call a mutation we use `$store.commit` with the mutation type
  - `$store.commit('ADD_POSTS')`
- An addition object (known as the payload) can be passed
  - `$store.commit('ADD_POSTS', posts)`
- When Vuex store is created with the `strict` property set to `true`, an error is thrown if code attempts to modify store state data directly rather than through mutations.