Vue.js is a JavaScript framework that involves the use of using multiple smaller components to create applications.
Vuetify is a CSS/HTML Material Design framework that integrates with Vue.js and provides custom Vue components to take care of styling as we need.

Vue.js files are suffixed with ‘.vue’ and usually consist of 3 major areas:
<template></template><script></script><style></style>

For HTML/Components, JavaScript, and CSS respectively.

To get started with Vue, the CLI is a pretty easy and common way to do it, especially if you’re familiar with using other Node packages.

From the command line, enter “`npm install -g vue-cli”` to install the Vue CLI.
With the Vue CLI, you can create a sample Vue project by typing “vue create {app_name}” and it will take you through a setup process to get you acquainted with Vue.

Once the Vue project is set up, you can add Vuetify simply by typing “vue add vuetify” and it should start to install the package.

With both Vue and Vuetify installed, you’re ready to start experimenting with both by creating your first Vue file.

Inside of your project folder that Vue created, you should have some files and folders. There should be a file called “App.vue” inside of the “src” folder, this will be the main directory that will serve your Vue app, and from here you can create more pages just like any other HTML website.

This “App.vue” file should be split into multiple sections like I listed above. The <template> part of the code is essentially just some raw HTML with some CSS baked in, but it also includes some of the custom components included with Vuetify. You can tell which components are part of Vuetify’s library by seeing any that are prefixed with “v-” like the “<v-img>” component. A full list of the components can be found on Vueitfy’s website under “UI Components.”

If you look at the very top, you’ll see a tag called “<v-app>” which is Vue’s standard tag that they use to bind this Vue instance to the DOM (Document Object Model), or our whole HTML project. The <v-app> tag doesn’t have anything special with it aside from some preset CSS and an id of “#app”. Inside of the “main.js” file you’ll notice that after we create our Vue instance, in the “new Vue” section, we mount this Vue instance to “#app” which essentially means that everything encompassed by the HTML element with an id of “#app” is going to be part of our Vue instance.

Further down the file should have a “<v-main>” tag which serves as the main wrapper component for the application, and you can see that there is a custom component placed inside of it called “<HelloWorld>.” This is a custom component that was created by Vue and can be found inside the “components” folder, and it gets imported into this file right below the “<script>” tag. Components are essentially just pieces of HTML/CSS/JS that are put together and can be used anywhere in the app. Everything that is inside of the “HelloWorld.vue” component file is what will get placed into the “App.vue” file where we have the “<HelloWorld>” tag, with these components essentially being just reusable portable code.

The main thing about Vue that makes it really useful is that it allows you to bind Javascript to HTML in a simple and easy to use way. For example, if you have a page that displays a certain image and text when you first load the page, but you want it to dynamically change to display a different image and text after the user does something, you can have the image and text be Javascript variables, monitor for that change, and once it occurs, change the value of those variables, and the page will update to match these changes. I’ll go more in detail with more specific examples after explaining the main Javascript components of Vue.

If you look at the <script> part of the file, you’ll notice some functions and objects that Vue uses, like “data.” Vue has a variety of different functions/objects that it uses to define the behavior of different code you want to use and have binded in your application. The main ones are: “data()”, “mounted()”, “computed:”, “methods:”, and “props:”.

“data()” serves as a location to place any sort of variables you want to be available in this component for use in either functions, or binding it to some HTML. Variables defined in “data()” can be changed at any time just like normal variables in Javascript, but they can’t contain any sort of dynamic data. By this, I mean you can’t have a variable defined with a value that isn’t readily known, so you can’t have any sort of logic be used to define a variable. If you wanted to have a variable get a certain value depending on a certain state of your app when the user gets to this component, you’d have to use the mounted().

“mounted()” is a code block that contains any sort of logic you want to occur when the component is first loaded up. This means for example if you have an application with two user types, and you have a variable called “msg” in your data() section that contains a certain message to be displayed on screen, you can call a function to check for the user’s type in mounted(), and depending on the result, you can modify “msg” to display a different message.

“Computed:” is similar to a data(), except the variables defined here are values that can be ‘computed’ in real time. This means if you need a variable to change its value based on a certain condition, once that condition is reached the value will change and be reflected in the component. Computed properties normally cannot be set using an assignment operator “=” after they’ve been defined, but there is special behavior where they can, but it most likely isn’t needed most times. An example of something that can be computed is the example from above, where instead of having to modify a variable in “data()” based on something that happened in “mounted()”, we can instead have that condition be in our “computed:” and return a different value based on that condition.

“Methods:” is the section to define all functions that will be used in the application. Any sort of function gets placed here and can be called later on from other functions, in the mounted(), or even bind it to components so that if you want a certain action, like clicking a button, to trigger the function, it can easily do that.

“Props” are properties that you can pass from one component to another. This allows components to become really abstract and almost serve as templates in your project. You could in theory create a component that is just a message box styled in a certain way, and have the text that is inside of that message box be a prop received. You can then place this component in any other component you want, and pass in the message as a prop on that component to have it be given that value.
The syntax for props is the same as defining properties or attributes on normal HTML elements, so if you have a prop called “myProp” you would just do:
<MyComponent myProp=”Hello World!”>
You can also put any Javascript data like variables or functions you defined to be passed as props as well by prefixing the prop with a colon “:”. If you have a variable called “myVar” you can pass it in like:
<MyComponent :myProp=”myVar”>

To access any of the data defined in any of the sections above, just prefix the name with “this.”. For example, if you had a variable “myVar”, you would call it with “this.myVar”, and a function called “myFunc()”, you would call it with “this.myFunc()”. The only exception to this would be when calling these in the <template> section, you don’t need to add the “this.” prefix.

data() example:
```
data() {
    return {
        str: “Hello World!”,
        num: 2002
    }
}
```

mounted() example:
```
mounted() {
    if (this.num == 2002) {
        this.str = “Goodbye World!”
    } else {
        console.log(“It isn’t 2002”)
    }
}
```

computed: example:

```
computed: {
    //You can use them just like normal variables
    simpleComputed() {
        return 2003
    },
    //Or make them contain some logic
    complexComputed() {
        if (this.num == 2002) {
            return 2003
        }
        return 2001
    }
}
```

Computed variables can also be defined like this if you want custom behavior for “getting” and “setting” the value, similar to how you would use “getters” and “setters” for private variables in Java:

```
computed: {
    myComputed: {
        get() {
            return this.num
        },
        set(val) {
            this.num = val
        }
    }
}
```

methods: example:

```
methods: {
    myFunc(str) {
        let temp = str
        if (this.num == 2002) {
            temp = temp + “ from inside myFunc()”
        }
        return temp
    }
}
```

props: example:
Parent Component (passing prop into child)

```
<HelloWorld myProp=”Hello World” :alsoProp=”num”>
```

Child Component (receiving prop from parent)

```
props: [‘myProp’, ‘alsoProp]
```

Props can also be defined like this to be more explicit, and if you want to type check the value of the prop, as well as provide a default value if that prop doesn’t get provided:

```
props: {
    myProp: {
        type: String,
        default: null
    },
    alsoProp: {
        type: Number,
        default: 0
    }
}
```

Now that we’ve covered all of the different fields that you can insert into the <script> tags, I’ll go over how you can use these in the rest of the program file. In your HTML part of the file, you can directly pass in these javascript variables and functions if you’d like, allowing you to have a text field contain just a variable instead of having to write out all of the information inline, which can be incredibly useful if you ever want to change what it says or swap it out for another piece of text. The way to use Javascript variables in HTML in place of text would be to use “mustache syntax” which is essentially wrapping any Javascript in “{{ }}” tags. For example:
If we have a variable defined in data called “str” and we want this to be used in our code, we would do

<h1> {{ str }} </h1>

You can also use a ternary operator here if you need to like:
<h1> {{ condition ? ‘Hello’ : ‘Goodbye’ }} </h1>