Vue Introduction

A JavaScript framework helps us to create modern applications. Modern JavaScript applications are mostly used on the Web, but also power a lot of Desktop and Mobile applications.

Why developers love Vue:

First, Vue is called a progressive framework.

This means that it adapts to the needs of the developer. Other frameworks require a complete buy-in from a developer or team and often want you to rewrite an existing application because they require some specific set of conventions. Vue happily lands inside your app with a simple script tag to start with, and it can grow along with your needs, spreading from 3 lines to managing your entire view layer.

Vue was built by picking the best ideas of frameworks like Angular, React and Knockout, and by cherry-picking the best choices those frameworks made. And by excluding some less brilliant ones, it kind of started as a "best-of" set and grew from there.

One thing that puts Vue in a different bucket compared to React and Angular is that Vue is an **indie** project: it's not backed by a huge corporation like Facebook or Google.

Instead, it's completely backed by the community, which fosters development through donations and sponsors. This makes sure the roadmap of Vue is not driven by a single company's agenda.

Setting up Vue - CDN

First I'll go through the most basic example of using Vue.

You create an HTML file which contains:

```
<html>
  <body>
    <div id="example">
      <p>{{ hello }}</p>
    </div>
    <script src="https://unpkg.com/vue"></script>
    <script>
        new Vue({
            el: '#example',
            data: { hello: 'Hello World!' }
        })
    </script>
  </body>
</html>
```

and you open it in the browser. That's your first Vue app! The page should show a "Hello World!" message.

I put the script tags at the end of the body so that they are executed in order after the DOM is loaded.

What this code does is instantiate a new Vue app, linked to the #example element as its template. It's defined using a CSS selector usually, but you can also pass in an HTMLElement.

Then, it associates that template to the data object. That is a special object that hosts the data we want Vue to render.

In the template, the special {{ }} tag indicates that this is some part of the template that's dynamic, and its content should be looked up in the Vue app data.

Vue Dev-Tools

When you're first experimenting with Vue, if you open the Browser Developer Tools, you will find this message: "Download the Vue Devtools extension for a better development experience: https://github.com/vuejs/vue-devtools"

This is a friendly reminder to install the Vue Devtools Extension. What's that? Any popular framework has its own devtools extension, which generally adds a new panel to the browser developer tools that are much more specialized than the ones that the browser ships by default. In this case, the panel will let us inspect our Vue application and interact with it.

This tool will be an amazing help when building Vue apps. The developer tools can only inspect a Vue application when it's in development mode. This makes sure no one can use them to interact with your production app — and will make Vue more performant because it does not have to care about the Dev Tools.

Install on Chrome

Go to this page on the Google Chrome Store and click **Add to Chrome**.

The Vue.js devtools icon shows up in the toolbar. If the page does not have a Vue.js instance running, it's grayed out.

If Vue.js is detected, the icon has the Vue logo's colors.

On the top there are four buttons:

- **Components** (the current panel), which lists all the component instances running in the current page. Vue can have multiple instances running at the same time.
- **Vuex** is where you can inspect the state managed through Vuex.

- **Events** shows all the events emitted.
- **Refresh** reloads the devtools panel.

Notice the small = $vm0 text beside a component? It's a handy way to inspect a component using the Console. Pressing the "esc" key shows up the console in the bottom of the devtools, and you can type $vm0 to access the Vue component.

Filter components

Start typing a component name, and the components tree will filter out the ones that don’t match.

Setup VS Code to work with Vue

Visual Studio Code is one of the most used code editors in the world right now. The cool thing about being popular is that people dedicate a lot of time to building plugins for everything they can imagine.

One such plugin is an awesome tool that can help us Vue.js developers.

Vetur

It's called Vetur, it's hugely popular (more than 3 million downloads), and you can find it on the Visual Studio Marketplace.

What does this extension provide?

Syntax highlighting

Vetur provides syntax highlighting for all your Vue source code files.

Snippets

As with syntax highlighting, since VS Code cannot determine the kind of code contained in a part of a .vue file, it cannot provide the snippets we all love. Snippets are pieces of code we can add to the file, provided by specialized plugins.

Vetur gives VS Code the ability to use your favorite snippets in Single File Components.

IntelliSense

IntelliSense is also enabled by Vetur, for each different language, with autocomplete:

Scaffolding

In addition to enabling custom snippets, Vetur provides its own set of snippets. Each one creates a specific tag (template, script, or style) with its own language:

If you type scaffold, you'll get a starter pack for a single-file component:

```
<template>
</template>
<script>
export default {
}
</script>
<style>
</style>
```

Emmet

Emmet, the popular HTML/CSS abbreviations engine, is supported by default. You can type one of the Emmet abbreviations, and by pressing tab VS Code will automatically expand it to the HTML equivalent:

Example: ul>li*5

Linting and error checking

Vetur integrates with ESLint through the VS Code ESLint plugin.

Code Formatting

Vetur provides automatic support for code formatting to format the whole file upon save — in combination with the "editor.formatOnSave" VS Code setting.

Vue Templates

Vue.js uses a templating language that's a superset of HTML.

Any HTML is a valid Vue.js template. In addition to that, Vue.js provides two powerful things: interpolation and directives.

This can be done using interpolation. First, we add some data to our component:

and then we can add it to our template using the double brackets syntax:

One interesting thing here. See how we just used name instead of this.data.name?

This is because Vue.js does some internal binding and lets the template use the property as if it was local. Pretty handy.

In a single file component, that would be:

```
<template>

 <span>Hello {{name}}!</span>

</template>
<script>
export default {
 data() {
  return {
   name: 'Flavio'
  }
```

```
 }
}
</script>
```

You can use any JavaScript expression inside your interpolations, but you're limited to just one expression:

```
{{ name.reverse() }}
{{ name === 'Flavio' ? 'Flavio' : 'stranger' }}
```

Vue provides access to some global objects inside templates, including Math and Date, so you can use them: `{{ Math.sqrt(16) * Math.random() }}`

Directives

We saw in Vue.js templates and interpolations how you can embed data in Vue templates.

This section explains the other technique offered by Vue.js in templates: directives.

Directives are basically like HTML attributes which are added inside templates. They all start with v-, to indicate that's a Vue special attribute.

Let's see each of the Vue directives in detail.

v-text

Instead of using interpolation, you can use the v-text directive. It performs the same job:

```
<span v-text="name"></span>
```

v-once

You know how {{ name }} binds to the name property of the component data.

Any time name changes in your component data, Vue is going to update the value represented in the browser.

Unless you use the v-once directive, which is basically like an HTML attribute:

```
<span v-once>{{ name }}</span>
```

v-html

Where you want to output HTML and make the browser interpret it. You can use the v-html directive:

```
<span v-html="someHtml"></span>
```

v-bind

Interpolation only works in the tag content. You can't use it on attributes.

Attributes must use v-bind:

```
<a v-bind:href="url">{{ linkText }}</a>
```

v-bind is so common that there is a shorthand syntax for it:

```
<a :href="url">{{ linkText }}</a>
```

Two-way binding using v-model

v-model lets us bind a form input element for example, and makes it change the Vue data property when the user changes the content of the field:

```
<input v-model="message" placeholder="Enter a message">

<p>Message is: {{ message }}</p>
```

==========

```
<select v-model="selected">
 <option disabled value="">Choose a fruit</option>
 <option>Apple</option>
 <option>Banana</option>
 <option>Strawberry</option>
</select>
<span>Fruit chosen: {{ selected }}</span>
```

Using expressions

You can use any JavaScript expression inside a directive:

```
<span v-text="'Hi, ' + name + '!'"></span>
<a v-bind:href="'https://' + domain + path">{{ linkText }}</a>
```

Any variable used in a directive references the corresponding data property.

Conditionals

Inside a directive you can use the ternary operator to perform a conditional check, since that's an expression:

```
<span v-text="name == Flavio ? 'Hi Flavio!' : 'Hi' + name + '!'"></span>
```

There are dedicated directives that allow you to perform more organized conditionals: v-if, v-else and v-else-if.

```
<p v-if="shouldShowThis">Hey!</p>
```

shouldShowThis is a boolean value contained in the component's data.

Show or hide

You can choose to only show an element in the DOM if a particular property of the Vue instance evaluates to true, using v-show:

```
<p v-show="isTrue">Something</p>
```

The element is still inserted in the DOM, but set to display: none if the condition is not satisfied.

Loops

v-for allows you to render a list of items. Use it in combination with v-bind to set the properties of each item in the list.

You can iterate on a simple array of values:

```
<template>

<ul><li v-for="item in items">{{ item }}</li></ul>


</template>
<script>
export default {
 data() {
   return {
     items: ['car', 'bike', 'dog']
   }
 }
}
</script>
```

Or on an array of objects:

```
<template>

 <div>

  <!-- using interpolation -->

  <ul>

    <li v-for="todo in todos">{{ todo.title }}</li>

  </ul>

  <!-- using v-text -->

  <ul>

    <li v-for="todo in todos" v-text="todo.title"></li>

  </ul>

 </div>


</template>
<script>

export default {
 data() {
  return {
    todos: [
      { id: 1, title: 'Do something' },
      { id: 2, title: 'Do something else' }
    ]
```

```
  }
 }
}
</script>
```

v-for can give you the index using:

```
<li v-for="(todo, index) in todos"></li>
```

Styling components using CSS

The simplest option to add CSS to a Vue.js component is to use the style tag, just like in HTML:

```
<template>

 <p style="text-decoration: underline">Hi!</p>

</template>
<script>
export default {
 data() {
  return {
    decoration: 'underline'
  }
 }
}
</script>
```

This is as static as you can get. What if you want underline to be defined in the component data? Here's how you can do it:

```
<template>

 <p :style="{'text-decoration': decoration}">Hi!</p>
```

```
</template>
<script>
export default {
 data() {
  return {
    decoration: 'underline'
  }
 }
}
</script>
```

`:style` is a shorthand for `v-bind:style`. I'll use this shorthand throughout this tutorial.

Vue components generate plain HTML, so you can choose to add a class to each element, and add a corresponding CSS selector with properties that style it:

```
<template>

 <p class="underline">Hi!</p>

</template>
<style>
.underline { text-decoration: underline; }
</style>
```

You can use SCSS like this:

```
<template>
```

```
 <p class="underline">Hi!</p>


</template>
<style lang="scss">
body {
 .underline { text-decoration: underline; }
}
</style>
```

You can hard code the class like in the above example. Or you can bind the class to a component property, to make it dynamic, and only apply to that class if the data property is true:

```
<template>


 <p :class="{underline: isUnderlined}">Hi!</p>


</template>
<script>
export default {
 data() {
   return {
     isUnderlined: true
   }
 }
}
</script>
<style>
.underline { text-decoration: underline; }
</style>
```

Instead of binding an object to class, like we did with <p :class="{text: isText}">Hi!</p>, you can directly bind a string, and that will reference a data property:

```
<template>

 <p :class="paragraphClass">Hi!</p>

</template>
<script>
export default {
 data() {
   return {
     paragraphClass: 'underline'
   }
 }
}
</script>
<style>
.underline { text-decoration: underline; }
</style>
```

You can assign multiple classes, either by adding two classes to paragraphClass in this case or by using an array:

```
<template>

 <p :class="[decoration, weight]">Hi!</p>

</template>
```

```
<script>
export default {
 data() {
  return {
    decoration: 'underline',
    weight: 'weight',
  }
 }
}
</script>
<style>
.underline { text-decoration: underline; }
.weight { font-weight: bold; }
</style>
```

The same can be done using an object inlined in the class binding:

```
<template>

 <p :class="{underline: isUnderlined, weight: isBold}">Hi!</p>

</template>
<script>
export default {
 data() {
  return {
    isUnderlined: true,
    isBold: true
  }
 }
}
</script>
<style>
.underline { text-decoration: underline; }
.weight { font-weight: bold; }
```

```
</style>
```
And you can combine the two statements:
```
<template>


 <p :class="[decoration, {weight: isBold}]">Hi!</p>


</template>
<script>
export default {
 data() {
   return {
     decoration: 'underline',
     isBold: true
   }
 }
}
</script>
<style>
.underline { text-decoration: underline; }
.weight { font-weight: bold; }
</style>
```
Any CSS that's not hard coded like in the first example is going to be processed by Vue, and Vue does the nice job of automatically prefixing the CSS for us. This allows us to write clean CSS while still targeting older browsers (which still means browsers that Vue supports, so IE9+).