Vue.js is an open-source model–view–viewmodel front end JavaScript framework for building user interfaces and single-page applications

****

```
<script setup>
 import { ref } from 'vue';

 let count = ref(0);
</script>
```

****

```
<p>
    {{ count }}
  </p>
  <p>
    <button @click="count++">+</button>
    <button @click="count--">-</button>
  </p>
```

****

All Vue Files That Return HTML Must Contain ***<template>...</template>***

And We Called That HTML: Vue Template File

****

We Can Add Styling To Our Vue Template Files:

- Adding New Style-Element **After/Before** The **template-element**
- Define New **Style.css File**

****

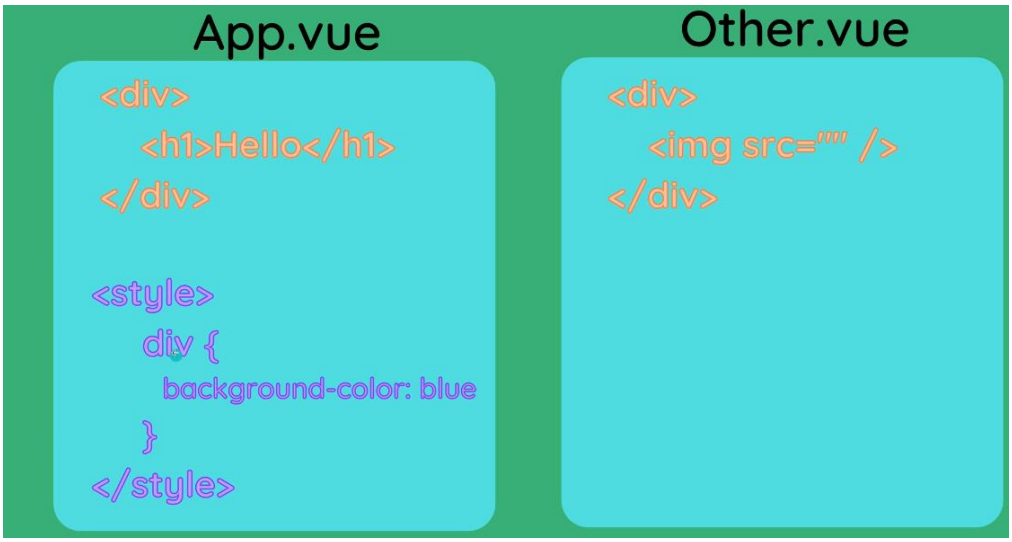
The Best Practice Is To Use **scoped** with **style-element**, so we can not fail with other **Vue-files**.

****

```
<style scoped>
div {
  height: 100vh;
  width: 100vw;
  background: linear-gradient(250deg, lightblue 75%, blue);

  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
}

</style>
```



****

For The First **script-tag**, It Should Contain **setup-attribute**

```
<script setup>
 import { ref } from 'vue';

 let count = ref(0);
</script>
```

Any Vue Template Files Can Have At Most One Script-Tag With setup Attribute Only.

****

If We Define Our State Variables Like This, And Want To Change Their Values In This Way, Nothing Will Changed:

```
<script setup>
 let count = 0;
 let add = () => {
  console.log("The Count is Inc");
  count = ++count;
 }
</script>
```

```
<template>
 <div>
  <p>
   {{ count }}
  </p>
  <p>
   <button @click="add()">+</button>
  </p>

 </div>
</template>
```

****

Vue Will Update The State Of The Application By Re-Rendering The HTML-Page.

****

To Define A State Variable, We Need To Impot ref From 'vue':

```
import { ref } from 'vue';
```

****

Then The Updated Code is, And That Will Work:

```
import { ref } from 'vue';

 let count = ref(0);

 console.log("The count State Variable is: ", count);

 let add = () => {
  console.log("The Count is Inc");
  count.value = ++count.value;
 }
```

```
<p>
    <button @click="add()">+</button>
</p>
```

****

****

Composition API: Is The New Way That Vue Build The Data It In Future.

****

The Options API, Depending On Export default that must contain data()-function, That Return The State Variables As Key-Pair Object.

And To Define The Methods That Handle And Change The State Variables, we Need To Declare Another Object Called methods:

```
export default({
  data(){
    return {
      count: 0
    }
  },
  methods: {
    addToCount(){
      console.log("hellloooooooo")
    }
  }
})
```

****

Note (To Remember): When We Have Container Above Container And Nothing Happened For The Main One, Then Change The z-index Of The Target One.

****

To Use **Conditional Rendering** Using Vue We Need To Use: v-if

****

Note (From Me): Don't Use <> with **ref** in Vue.

****

****

Example Of v-if And Props Function:

```
<ModalComponent :close-function="close" v-if="showModal" />
```

Where We Set Inside The script-setup Of ModalComponent:

```
<script setup>

defineProps({
  closeFunction: Function,
});

</script>
```

****

The Same Effect We Can Use With v-show:

Note (From Me): Worked Only Once, With Strange Behavior:

```
<ModalComponent :close-function='close' v-show='showModal' />
```

****

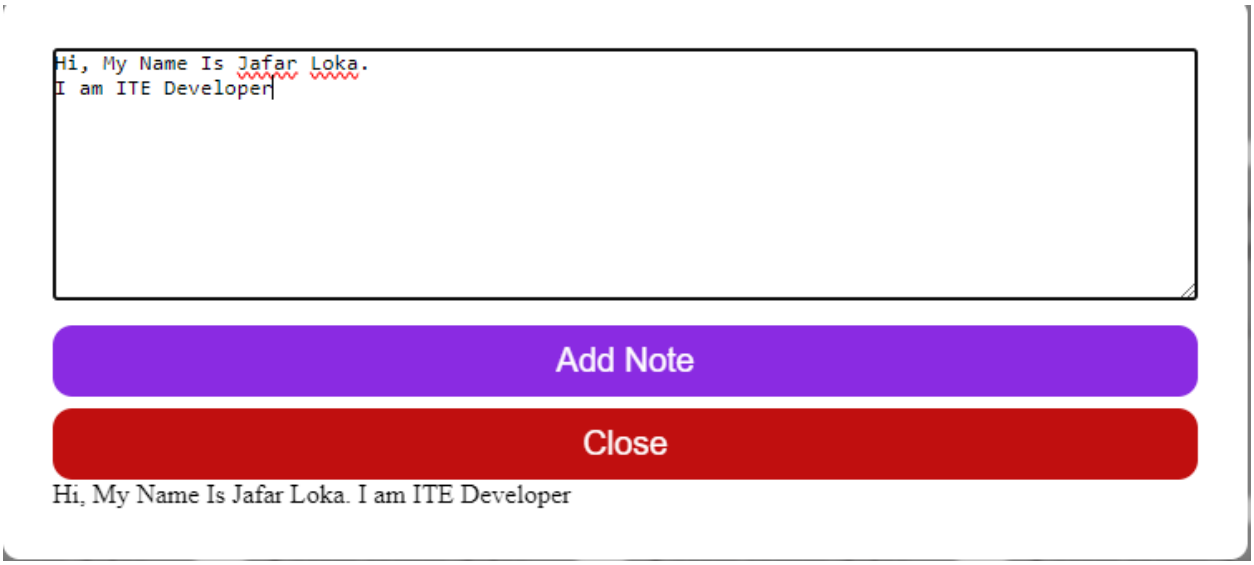
Two Way Binding: The Way When We Bind The State With The UI Of User, So When We Change The State The UI Updated With The Current Data, And When We Change The UI (Ex: textarea, input, ...etc.) The State Will Change.

****

Note (From Me): If The v-model Not Worked With ref(...) Then Check If We Import The ref From Vue.

```
const newNote = ref("Hi, MyName Is Jafar Loka");
```

****



src > components > ModalComponent.vue > { } template > div.overlay > div.modal

```vue
<script setup>

import { ref } from 'vue';

defineProps({
    closeFunction: Function,
});

const newNote = ref("Hi, My Name Is Jafar Loka");

</script>

<template>

    <div class='overlay'>
      <div class='modal'>
        <textarea name='note' id='note' cols='30' rows='10' v-model="newNote"></textarea>

        <button>
          Add Note
        </button>

        <button class='close' @click="closeFunction">
          Close
        </button>

        {{ newNote }}
      </div>


    </div>

</template>
```

****

Th Right Way To Use inline-style As JS in Vue is By Using:

Note 1: Here We Use :style For Inline-Style.

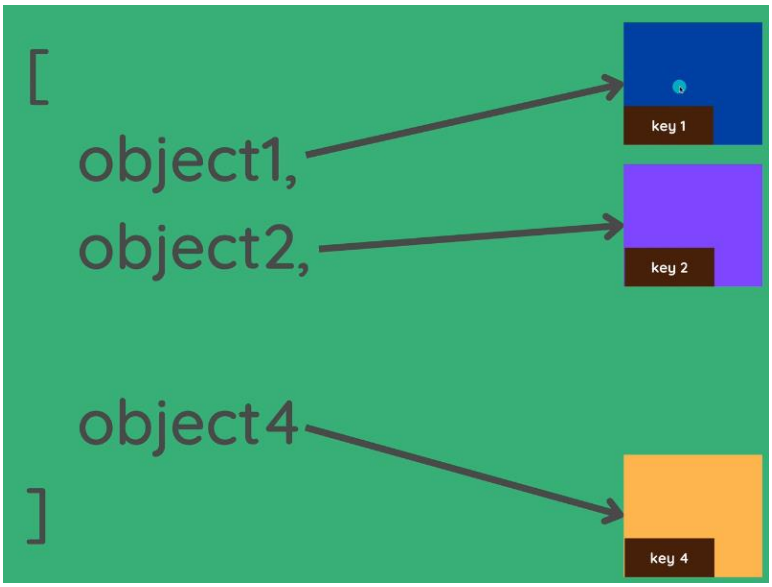
Note 2: The Value Of :style Is Object Because It is JS.

Note 3: The v-for Directive Used To Render A List Of Data.

Note 4: The Important Of :key For Rendering Performance, Without Key; If We Update/Remove Any Item In The Array, Vue Will Remove Every Thing And Re-Render It Again.

```
<div
        class='card'
        v-for="note in notes"
        :style="{ backgroundColor: note.backgroundColor }"
        :key="note.id">
                <p class='card-main-text'>
                        {{ note.text }}
                </p>

                <p class='card-date'>{{ note.date }}</p>
        </div>
</div>
```



****

In This Way We Can Use Props That Defined Using defineProps()-Function:

```
<script setup>
import { ref } from 'vue';
const props = defineProps({
   closeFunction: Function,
   addNote: Function,
});
const newNote = ref("");
const errMsg = ref('');
const addNote2 = (note) => {
 if(note.length < 10) {
   errMsg.value = 'Note Must Be At Least 10-characters';
 } else {
   props.addNote(note);
 }
}
</script>
```

****

To Use trim with two-way-bindings, We Can use v-model.trim='ref-data-here':

```
<textarea name='note' id='note' cols='30' rows='10' v-
model.trim="newNote"></textarea>
```

****