**VUE JS**

Vue.js is an **open-source progressive JavaScript framework** used to develop interactive **web user interfaces and single-page applications (SPAs)**. Vue.js is commonly referred to as Vue and pronounced as "view.js" or "view.

Vue does not focus on the other aspects of application like routing or HTTP requests

Vue has a rich ecosystem of other powerful libraries that can be integrated.

Vuex npm package for complex state management, vue router for routing, Vuetify for UI elements and lot more are there.

**Single page application**

A single page application or SPA is a web application or a website that provides users a very fluid, reactive, and fast experience similar to a desktop application. A single page application contains a menu, buttons, and blocks on a single page. When a user clicks on any of them, it dynamically rewrites the current page rather than loading entire new pages from a server. That's the reason behind its reactive fast speed.

**Features of vue.js**

**Components**

Vue.js Components are one of the important features of this framework. They are used to extend basic HTML elements to encapsulate reusable code. We can create reusable custom elements in Vue.js applications that can be later reused in HTML.

**Templates**

Vue.js provides HTML-based templates that can be used to bind the rendered DOM with the Vue instance data. All Vue templates are valid HTML that can be parsed by specification-compliant browsers and HTML parsers. Vue.js compiles the templates into Virtual DOM render functions.

### Reactivity

Vue provides a reactivity system that uses plain JavaScript objects and optimizes re-rendering. In this process, each component keeps track of its reactive dependencies, so the system knows precisely when, and which components to re-render.

### Routing

Navigation between pages is performed with the help of vue-router. You can use the officially-supported vue-router library for your Single Page Application.

**Vue installation**

There are three ways

**By using cdn package**

 <script *src*="https://unpkg.com/vue@3/dist/vue.global.js"></script>

**By using npm**

Preferred approach over cdn when building large scale applications with Vue**.**

npm install vue@next

**By using CLI command line**

Vue provides an official CLI for quickly scaffolding single page applications

npm install -g @vue/cli

or

yarn global add @vue/cli

**Create a first project with vue-cli**

Open command prompt in the folder where you want to create the project

vue create <project-name>

Choose the version and give yes for rest of the things

cd <project-name> // to move to project folder

npm run serve // run the project

**Files and folder structure of created app.**

load your project in any of the IDE(VS code). After loading your project, click on the **public** folder as well as the **src** folder. These two folders contain the most important files of your project.

**Package.json:** The Package.json folder contains all configuration of your app. Don't need to touch that folder.

**Index.html:** The most important file of your app is index.html file. It contains all the data that is appeared on the UI of your app. This is the only one div element which Vue will use to attach the DOM.  <div id="app"></div>

**Src/main.js:** main.js is the main JavaScript file which is used to drive your app.

*import* { createApp } *from* 'vue'

*import* App *from* './App.vue'

createApp(App).mount('#app')

In the above code, the first line specifies the import of the Vue library, and the second line is used to import the App Component from app.vue. After that, Vue instance is created by assigning it to DOM element #app, which has already been defined in index.html file and specified to use the App component.

**Src/App.vue:** The App.vue is single file component that contains three parts: **HTML, CSS and JAVASCRIPT**. It is used to manage the single file components of your app. Here, whatever additional component we use we have to import it, register it and use it. Eg: helloworld component in app.vue.

**Src/component/HelloWorldComponent:** This file is included in HelloWorld.vue file. This file contains all the style and HTML which you see when you open your app in the browser.

The **<style scoped>** attribute is used here to limit the below CSS to this component only. After using the <style scoped> attribute, your CSS would not be leaked to another part.

<style *scoped*>

h3 {

  margin: 40px 0 0;

}

ul {

  list-style-type: none;

  padding: 0;

}

li {

  display: inline-block;

  margin: 0 10px;

}

a {

  color: #42b983;

}

</style>

# **Vue.js Declarative Rendering**

It enables us to declaratively render data to the DOM using simple, straightforward template syntax.

**Syntax**

Index.html

<div *id*="app">

        {{message}}

 </div>

App.js

var app = new Vue({

    el: '#app',   // el parameter is used to carry id of DOM eleemnt

    data: {

      message: 'This is a Vue.js Declarative Rendering example!'

    }

  })

There are two types of directives in Vue.js: built-in directives and user-defined directives. Vue.js uses double braces {{ }} as place-holders for data, and the Vue.js directives are HTML attributes that use a **v-** prefix.

Example-1

Index.html

<!DOCTYPE *html*>

<html *lang*="en">

<head>

    <meta *charset*="UTF-8">

    <meta *name*="viewport" *content*="width=device-width, initial-scale=1.0">

    <title>Document</title>

    <script *src*="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

</head>

<body>

    <div *id*="app">

        {{message}} //message is rendered

    </div>

   <script *src*="app.js"></script>

</body>

</html>

App.js

const app = new Vue({

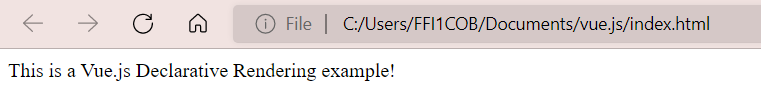
  el: '#app',

  data: {

    message: 'This is a Vue.js Declarative Rendering example!'

  }

})



Now, the data and the DOM are linked, and everything is now reactive. we no longer have to interact with the HTML directly. A Vue app attaches itself to a single DOM element and fully controls it. In the above example case, it is #app. Now, HTML is only the entry point, and everything else happens within the newly created Vue instance.

# **Vue.js Conditional Rendering**

Conditions and Loops are used in all programming languages to provide repetitive control structures. They can repeat one or more various steps depending on the conditions.

There are 4 elements

* v-if
* v-else
* v-else-if
* v-show

## **v-if**

The directive v-if is used to conditionally render a block. The block will only be rendered if the directive's expression returns a truthy value

<div *id*="app">

        <h2 *v-if*="num===0">This number is zero </h2>

         </div>

   <script>

    var app = new Vue({

  el: '#app',

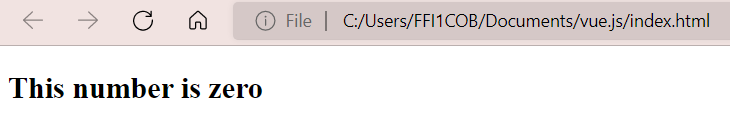
  data: {

    num: 0

  }

})

   </script>



## **v-else**

You can use the v-else directive to indicate an "else block" for v-if.

<div *id*="app">

        <button @*click*="show = !show">Toggle</button>

        <h1 *v-if*="show">Vue is interesting</h1>

        <h1 *v-else*>Oh no!</h1>

         </div>

   <script>

    var app = new Vue({

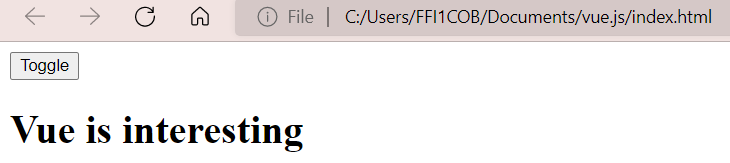
  el: '#app',

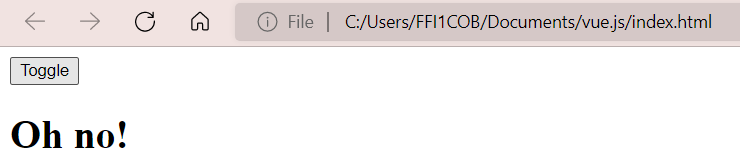
  data: {

    show: true

  }

})



****

## **v-else-if**

The v-else-if, as the name suggests, serves as an "else if block" for v-if. It can also be chained multiple times

<div *id*="app">

        <h1 *v-if*="num==0">The number is zero</h1>

        <h1 *v-else-if*="num < 0">The number is negative</h1>

        <h1 *v-else-if*="num > 0">The number is positive</h1>

        <h1 *v-else*>Not a number</h1>

         </div>

   <script>

    var app = new Vue({

  el: '#app',

  data: {

    num: -5

  }

})

   </script>

****

## **v-show**

Another option for conditionally displaying an element is the v-show directive. The usage is largely the same.

<h1 *v-show*="ok">Hello!</h1>

The difference is that an element with v-show will always be rendered and remain in the DOM; v-show only toggles the display CSS property of the element. v-if **conditionally renders** elements and v-show \*\*conditionally displays \*\*elements.

This means that v-if will actually destroy and recreate elements when the conditional is toggled. Meanwhile, v-show will always keep the element in the DOM and will only toggle its display by changing its CSS. (display:none)

**Vue.js Template**

Vue uses an HTML-based template syntax that allows you to declaratively bind the rendered DOM to the underlying component instance's data.

If you choose the simple interpolation method i.e., with double curly brackets i.e {{html content}} to display the HTML content on the web browser, it will show the wrong result. To resolve this use **v-html** directive. When we assign v-html directive to the html element, Vue.js knows that it has to output it as HTML content.

**Example**

 <div *id*="app">

            <h2>Firstname : {{firstname}}</h2>

            <h2>Lastname : {{lastname}}</h2>

            <div *v-html* = "htmlcontent"></div>

         </div>

   <script>

    var app = new Vue({

  el: '#app',

  data: {

      firstname : "Isac",

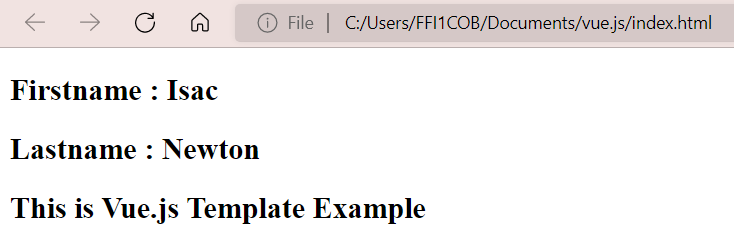
      lastname : "Newton",

      htmlcontent : "<div><h2>This is Vue.js Template Example</h2></div>"

   }

})

   </script>



### Add attributes to HTML element

### Suppose , we have an image tag in the HTML file and we want to assign ****src****, which is a part of Vue.js. we have to use ****v-bind directive**** to assign any attribute to the HTML tag.

 <div *id*="app">

            <h2>Firstname : {{firstname}}</h2>

          <h2>Lastname : {{lastname}}</h2>

            <div *v-html* = "htmlcontent"></div>

            <img *v-bind*:*src* = "imgsrc" *width* = "400" *height* = "300" />

         </div>

   <script>

    var app = new Vue({

  el: '#app',

  data: {

      firstname : "Isac",

      lastname : "Newton",

      htmlcontent : "<div><h2>This is Vue.js Template Example</h2></div>" ,

        imgsrc:"https://cdn.pixabay.com/photo/2015/12/01/20/28/road- 1072821\_\_480.jpg"

   }

})

### 

**VUE JS COMPONENTS**

Vue follows a component based architecture.

This lets you break the application into small encapsulated parts which can then be composed to make more complex user interfaces. Eg: A traditional website can be broken down into Header, SideNav, Main content & Footer i.e total 5 components. Fifth component is to contain every other component (Root (App)component).

**MyHome.vue**

<template>

    <div>

        <h1>Hello from home component</h1>

    </div>

</template>

<script>

    export *default*{

        name:'MyHome',

    }

</script>

**App.vue**

<template>

  <div *id* = "app">

    <img *alt*="Vue logo" *src*="./assets/logo.png">

  <MyHome */>* //use component

  </div>

</template>

<script>

*import* MyHome *from* './components/MyHome.vue' // import componnet

*export* *default* {

  name: 'App',

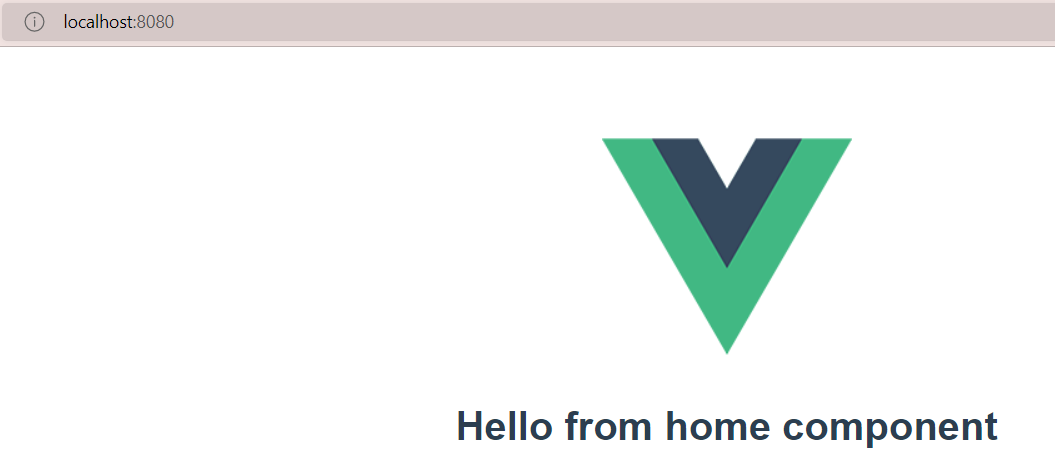
  components: {

    MyHome //register component

  }

}

</script>

****

**List Rendering**

**v-for**

We can use the v-for directive to render a list of items based on an array. The v-for directive requires a special syntax in the form of item in items, where items is the source data array and item is an **alias** for the array element being iterated on.

<template>

        <h2 *v-for*="item in technology" :*key*="item">{{item}}</h2>

        <h3 *v-for*="name in fullName" :*key*="name">{{name.first}} {{name.last}}</h3>

        <div *v-for*="actor in actors" :*key*="actor.name">

            <h2>{{actor.name}}</h2>

            <h3 *v-for*="movie in actor.movies" :*key*="movie">{{ movie }}</h3>

        </div>

        <h3 *v-for*="(value,key,index) in myInfo" :*key*="value">{{index}} {{key}} {{value}}</h3>

        </template>

<script>

    export *default*{

        name:'MyHome',

        data(){

*return*{

                technology: ['Java','Springboot','dotnet'], *//array*

                fullName:[

                    {first:'Albert', last:'Einstein'}, *//array of objects*

                    {first:'Nikola', last:'Tesla'},

                    {first:'Michael', last:'Faraday'},

                ],

                actors:[                     *//array of arrays*

                    {

                        name: 'Jayam Ravi',

                        movies: ['Ponniyin selvan','Thani Oruvan']

                    },

                    {

                        name: 'Surya',

                        movies: ['Jhai Bhim','Soorarai Potru']

                    },

                ],

             myInfo:{ //object

                name:'Vaishnavi',

                education:'B.E.',

                company:'Robert Bosch'

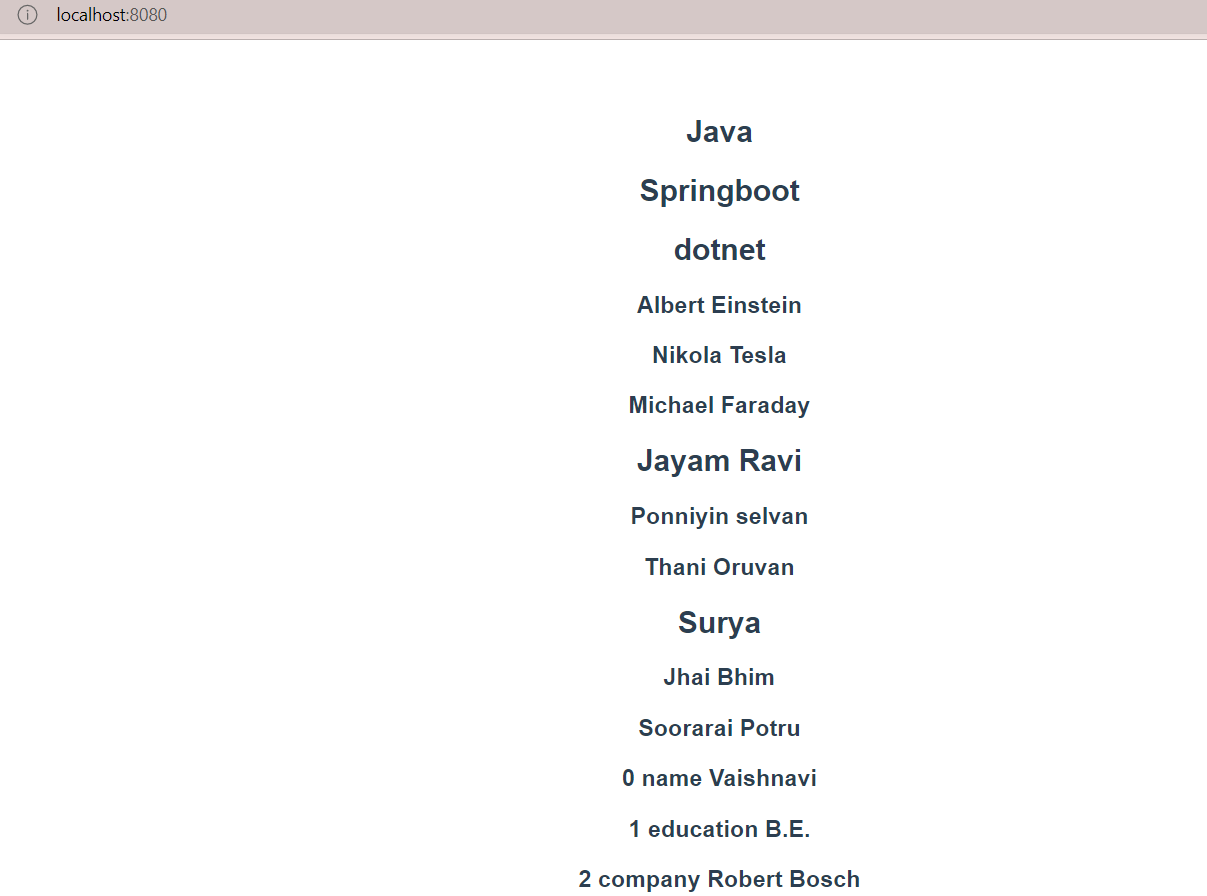
             }

            }

        },

    }

</script>

****

**Event handling**

The most common events that are generally used to be handled:

* submit
* keyup
* drag
* scroll
* error
* abort
* mouseover
* mouseout
* load etc

We can use the v-on directive, which we typically shorten to the @ symbol, to listen to DOM events and run some JavaScript when they're triggered. The usage would be v -on:click="handler" or with the shortcut, @click="handler".

The handler value can be one of the following:

1. **Inline handlers:** Inline JavaScript to be executed when the event is triggered (similar to the native onclick attribute).
2. **Method handlers:** A property name or path that points to a method defined on the component.

<template>

<h2>{{fruit}}</h2>

<div>

    <button *v-on*:*mouseover*="fruit= 'Strawberry'">Change Fruit</button> //inline </div>

<h2>{{count}}</h2>

<div>

    <button *v-on*:*click*="increment(1)">Increment 1</button> //method handlers

    <button *v-on*:*click*="decrement(1)">Decrement 1</button>

    <button *v-on*:*click*="increment(5)">Increment 5</button>

    <button *v-on*:*click*="decrement(5)">Decrement 5</button>

</div>

</template>

<script>

    export *default*{

        name:'MyHome',

        data(){

*return*{

                fruit:'Apple',

                count:0

            }

        },

        methods:{

            increment(num){

*this*.count+=num

            },

            decrement(num){

*this*.count-=num

            }

        },

    }

</script>

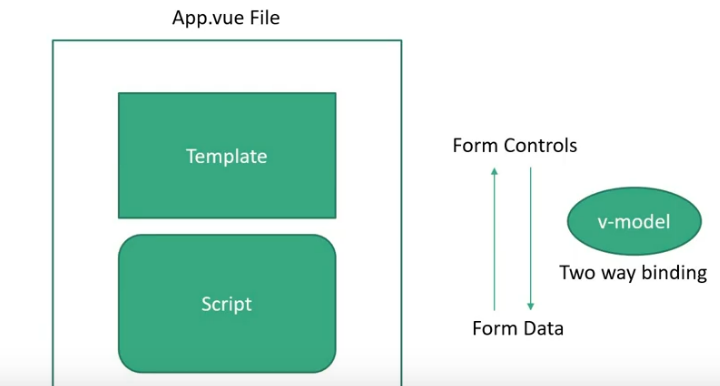


**Event Modifiers**

Vue.js provides some event modifiers available on v-on attribute. We can very easily call event.preventDefault() or event.stopPropagation() inside event handlers. Here, .prevent and .stop are event modifiers. These modifiers are directive postfixes denoted by a dot. Following is the list of most common modifiers available on v-on attribute:

* .once
* .prevent
* .stop
* .capture
* .self
* .passive

**Form Handling**

****

The **v-model** directive provides two way binding which is binding from the template to the data and data back to the template. It basically ensures that the model and vue are always in sink.

Example.

<template>

     <div>

    <pre>

      {{ JSON.stringify(formValues, null, 2) }}

    </pre>

  </div>

  <form @*submit*.*prevent*="submitForm">

    <div>

      <label *for*="name">Name</label>

      <input *type*="text" *id*="name" *v-model*.*trim*="formValues.name" />

    </div>

    <div>

      <label *for*="profile">Profile Summary</label>

      <textarea *id*="profile" *v-model*="formValues.profileSummary" />

    </div>

    <div>

      <label *for*="country">Country</label>

      <select *id*="country" *v-model*="formValues.country">

        <option *value*="">Select a country</option>

        <option *value*="india">India</option>

        <option *value*="vietnam">Vietnam</option>

        <option *value*="singapore">Singapore</option>

      </select>

    </div>

    <div>

      <label *for*="job-location">Job location</label>

      <select *id*="job-location" *v-model*="formValues.jobLocation" *multiple*>

        <option *value*="india">India</option>

        <option *value*="vietnam">Vietnam</option>

        <option *value*="singapore">Singapore</option>

      </select>

    </div>

    <div>

      <input *id*="remoteWork" *type*="checkbox" *v-model*="formValues.remoteWork"

*true-value*="yes"*false-value*="no"/>

      <label *for*="remoteWork">Open to remote work?</label>

    </div>

    <div>

      <label>Skill set</label>

      <input *type*="checkbox" *id*="html" *value*="html"

*v-model*="formValues.skillSet” />

      <label *for*="html">HTML</label>

      <input *type*="checkbox" *id*="css" *value*="css"

*v-model*="formValues.skillSet"/>

      <label *for*="css">CSS</label>

      <input *type*="checkbox" *id*="javascript" *value*="javascript"

*v-model*="formValues.skillSet"/>

      <label *for*="javascript">JavaScript</label>

    </div>

    <div>

      <label>Years of Experience</label>

      <input *type*="radio" *id*="0-2" *value*="0-2"

*v-model*="formValues.yearsOfExperience" />

      <label *for*="0-2">0-2</label>

      <input *type*="radio" *id*="3-5" *value*="3-5"

*v-model*="formValues.yearsOfExperience"/>

      <label *for*="3-5">3-5</label>

      <input

*type*="radio" *id*="6-10" *value*="6-10"

*v-model*="formValues.yearsOfExperience" />

      <label *for*="6-10">5-10</label>

      <input *type*="radio" *id*="10+" *value*="10+"

*v-model*="formValues.yearsOfExperience" />

      <label *for*="10+">10+</label>

    </div>

    <div>

      <label *for*="age">Age</label>

      <input *type*="number" *id*="age" *v-model*.*number*="formValues.age" />

    </div>

    <div>

      <button>Submit</button>

    </div>

  </form>

</template>

<script>

     export *default*{

        name:'MyHome',

        data(){

*return*{

                formValues:{

                    name:'',

                    ProfileSummary:'',

                    Country:'',

                    JobLocation:[],

                    remoteWork: 'no',

                    skillSet: [],

                    yearsOfExperience:'',

                },

            }

        },

        methods:{

          submitForm(event){

            event.preventDefault() // to prevent page refresh on submission

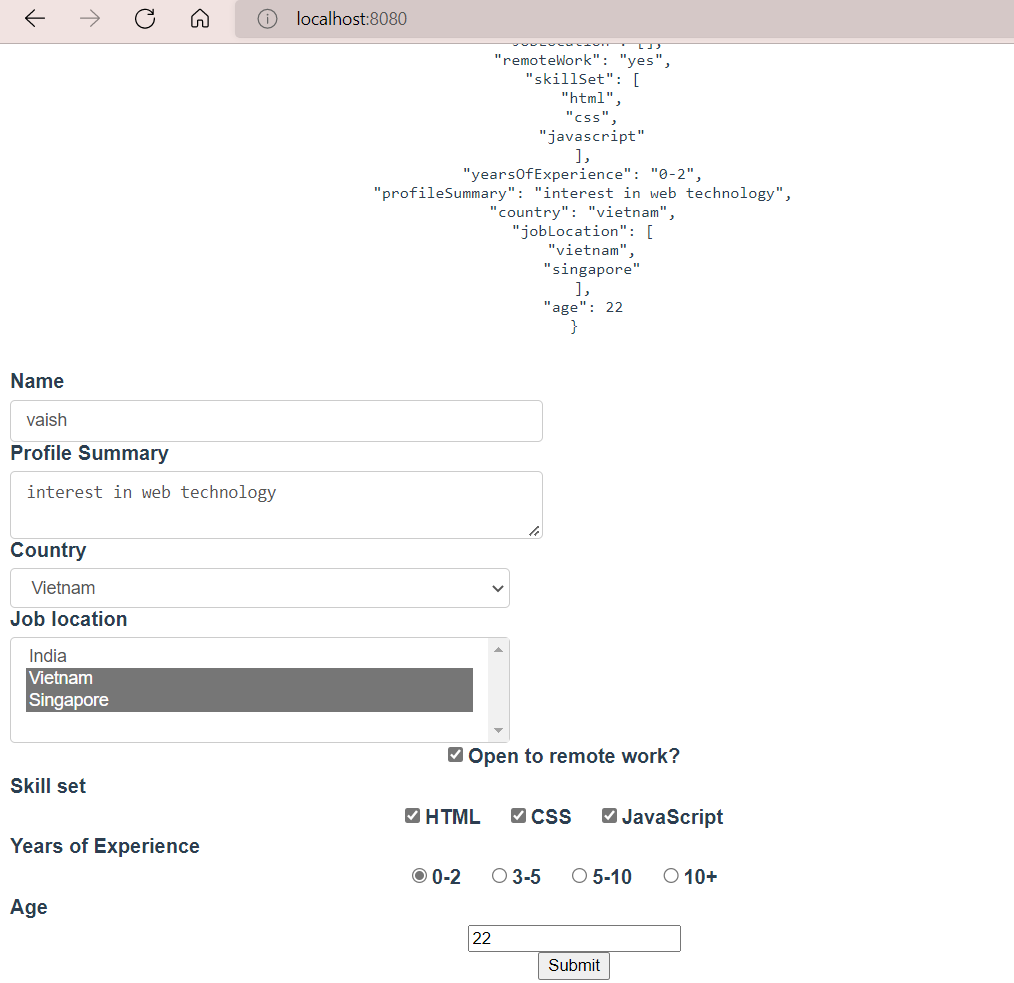
            console.log('Form values',*this*.formValues)

          }

        },

     }

</script>





**Modifiers :**

v-on or v-model to add some functionality inline within template.

**trim** – to trim the white spaces entered by user.

Add as a suffix to v-model like **v-model.trim=”formvalues.name”**

**number** – to store the number as number rather than strings from the input fields

**lazy** – to bind the data only when the event occurs

because generally when typing in a field the data get bound into the script as we type each letter. To avoid this lazy is used.

**prevent** – can be used as an alternative for preventDefault()

**.enter** - to submit the form when enter button is pressed.

<input @keyup.enter=”submitfn()”type="text" id="name" v-model="formvalues.formname">

v-once : - to render only once . if done subsequently it will be skipped

v-pre: doesn’t compile thee given element

**Computed properties**

Displaying data in UI

* Static HTML
* Text interpolation
* Expression in mustache syntax
* Methods
* **Computed properties**
* A property Can be bound to template like data properties
* Used for composing new data from existing sources
* Highly performing as they are cached calculations that update only when the dependencies change

Advantages : reusability of code

Using methods also computed properties can be done in method property but the difference is that whenever any changes occurs in the UI , the ethod will be called and it re-computes. Unlike methods, computed properties don’t execute on any change in UI until the browser reloading.

<template>

  <div id="app">

<h2> computed properties</h2>

<h3> {{fullfam }}</h3>

<button @click="family.push({names :' rajendran',whereabouts:[' ','working','karur']  })"> add another name</button>

</div>

</template

export default {

  name: 'App',

data(){return{ family:[

        {  names:"dhivya ",whereabouts:["2nd","studying", "chennai"]   },

        {  names :"rama",whereabouts:["1 st","working","karur"]  },   ],

}},

method:{

},

computed:  {

 fullfam(){

  return this.family.reduce((all,curr)=> (all=all+curr.names ),"");

},};

**Conditional list rendering using computed properties :**

<template>

  <div id="app">

<h2 v-for="obj in filterfam" :key="obj.name"> {{ obj.name}} {{obj.age}}</h2>

</div>

</template>

export default {

  name: 'App',

  data(){return{

newobj:[

      {name:"rajendran",

      age:"59"},

      {name:"gunavathi",

      age:"43"},

      {

        name:"Dhivya",

        age:"18"

      },

      {

        name:"rama",

        age:"21"

      }

     ]

     }

     },

 methods:{

filterfam(){

  return this.newobj.filter(item => item.age>50)

}    }  };

**Computed Setter :**

All the above computed properties are read only, whereas the computed setter is used to store the value received. To achieve this , a separate function for the property.

Defaultly, computed properties are getters.

In the abode code , we can only get the **name** of person aged above 50 and can’t set it.

get() – called whenever wanted to read the vale same as before code

set()- when a new value is assigned to computed property

initially whose name (rama )’ s lastname -prabha is changed to R using computed setters.

A function **full** is called upon a button click, which passes a new last name value to the set method which calls the set () and it sets the new value to the lastname variable which will be reflected after the button is clicked.

<template>

  <div id="app">

<hr> <h1> computed setters</h1>

<button @click="newEntry()"> Add new entry to Newobj </button>

   </div>

</template>

export default {

  name: 'App',

  data(){   return{

 lastname:"prabha",

} },

    methods:{

newEntry(){

    this.full="R "

  }

  },

 computed:  {

full:{

  get(){

  return this.newobj.filter(item => item.age>50)

  },

  set(value){

     this. lastname=value;

  }

}

    }

  };

**Usage of computed Properties :**

1. When new data is to be composed from existing data sources
2. Reducing length of the variable

**Watchers:**

To observe the data or computed property and performs actions in return to the changes in values.

**Usage:**

1. When an action is to performed as a result of a change
2. When have to call an API in response to change in application data
3. To apply transitions

**Note:**

* The watch property here is an object… the keys in here corresponds to the data property or computed property that we wanted to watch for a change in value.
* Here we watch the volume in the data property.
* Each key is assigned a function that will be executed whenever the property value changes.
* The function automatically receives the updated value as an argument.
* The function basically receives two arguments one is the updated value and another is old value .it can be read/ manipulated only when the argument is specified in watcher property

 <template>

<div id="app">

<hr> <h1>Watchers </h1>

<h3> volume tracker </h3>

<h3> volume {{volume}} </h3>

<button @click=" volume = volume+2"> increase volume </button>

<button @click=" volume -=1"> decrease volume </button>

   </div>

</template>

export default {

  name: 'App',

  data(){   return{

volume:0,

     }

     },

},

computed:  {

 },

    watch:{

   volume(valuenew,valueold){

    if(valuenew>valueold && valuenew==6){

      return alert("Listening to higher volume will damage your ears")

    }

   }

    }

  };

Can call API to fetch data while the page is loading.

<template>

<div>

 <input type="text" v-model="movie">

   </div>

</template>

export default {

  name: 'App',

  data(){   return{

movie:"new"

     }

     },

    methods:{

},

computed:  {

},

    watch:{

movie(val){

console.log(`calling api  ${val} `)

   }

    }

  };

In above only when changes is done logged into the console.

If an API have to be called immediately after the page is loaded ,even without any changes done **Immediate property** with TRUE value will make it happen.

And also have to change the function to object

<template>

<div>

 <input type="text" v-model="movie">

   </div>

</template>

export default {

  name: 'App',

  data(){   return{

movie:"new"

     }

     },

    methods:{

},

computed:  {

},

    watch:{

movie:{

    handler(val){

console.log(`calling api  ${val} `)

},

immediate:true,

   }}};

The movie function in above code is now changed into object with handler function to log immediately after page loaded.

Watchers by default does not watch for deeply nested properties of objects and also for mutating arrays. To make it happen set **deep** property to TRUE.

<template>

<div>

 <br> <br><input type="text" v-model="me.name">

<br> <br><input type="text" v-model="me.age">

   </div>

</template>

export default {

  name: 'App',

  data(){   return{

me:{

      name:"name",

      age:"age",

     },

     }

     },

    methods:{

},

computed:  {

},

    watch:{

me:{

    handler(Val){

      console.log(`loggin with deep name ${Val.name} and age ${Val.age}`  )

    },deep:true, immediate:true,

   }

       }

  };

**deep** property to mutate the array or object instead of returning a new reference

to return reference deep is not needed

<template>

<div>

 <button @click='moviearr.push( "  Sherlock  " )' > Add movie  </button>

   </div>

</template>

export default {

  name: 'App',

  data(){   return{

moviearr:[ " No way home"," far from home"]

     },

     }

     },

    methods:{  },

computed:  {

},

watch:{

    moviearr:{

    handler(Val){

      console.log(`adding new movie ${Val}`  )

      console.log(this.moviearr) //mutated array which not return the reference

    },

    deep:true,

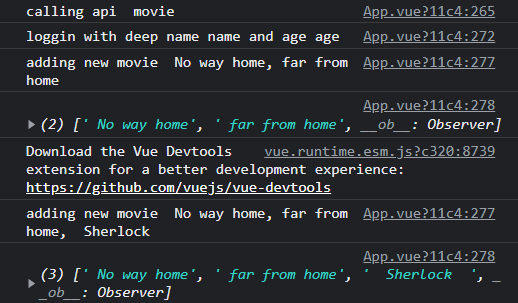
    immediate:true,

   }

       }

  };

Output:



<template>

<div>

 <button @click='moviearr=moviearr.concat( ["  Sherlock  " ])' > Add movie  </button>

   </div>

</template>

export default {

  name: 'App',

  data(){   return{

moviearr:[ " No way home"," far from home"]

     },

     }

     },

    methods:{

},

computed:  {

},

    watch:{

    moviearr:{

    handler(Val){

      console.log(`adding new movie ${Val}`  )

      console.log(this.moviearr) //mutated array which not return the reference

    },

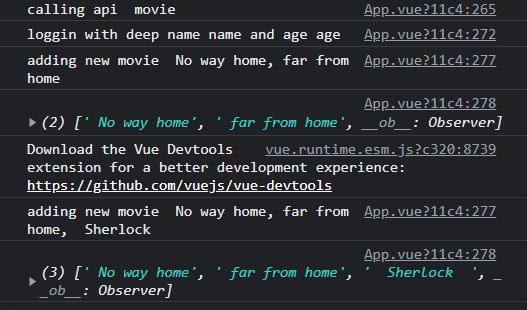
    immediate:true,

   }

       }

  };

Output:



**Components:**

* Vue follows **component based architecture.**

Helps to break down the application into small encapsulated parts , which then can be composed to make more complex UI.

* Every small components are finally contained in a container component called APP component which is the root written in App.vue
* Reusable with different properties only with data changes.
* Registering a component -

1. export the component (rama.vue ) from rama.vue

<script>

    export default {

        // eslint-disable-next-line vue/multi-word-component-names

        name: 'rama',

    }

</script>

1. and import it in App.vue

import rama from ' ./components/rama.vue'

1. then include it in the APP component template. (including a **property** called **component** in the export segment of APP as object and the file name(rama) as key )

export default {

  name: 'App',

  components: {

  // eslint-disable-next-line vue/no-unused-components

  rama,

  }

};

vbase-css – shorthand property that provides the basic structure (skeleton code )including template, scripts, styles etc.,

* Can create a component (here we say rama.vue) that can return a hmtl that we wanted and can include it in any part of the application.

**Component props :**

Custom attributes that we can register on component and allows the component content to be dynamic.

<h1> Components</h1>

<RamaPrabha name =" Rama "/>

<RamaPrabha name = "vaish"/>

   </div>

</template>

Values to be displayed is added in the template syntax of the App.vue which are then should be accepted by the child component Ramaprabha.vue. for being accepted we specify the **props.**

two steps are there ;

1. Specify a props property on the default of the newly created component
2. Props is an array of all data properties and custom attributes that component will accept from parent component. Here **name** is the data property that should be accepted, so it is specified in the array

**App.vue is parent;**

**RamaPrabha.vue is child**

export default {

        name: 'RamaPrabha',

        props: ["name"]

    };

1. This array now contains the value to be displayed dynamically.

Then bind the data property to the template syntax

<h2> Hii from RamaPrabha.vue component for {{name}} </h2>

In the above, in our app component we are passing the static values as props but we can also pass dynamic values by v-bind directive.

<h1> Components</h1>

<RamaPrabha name =" Rama " place =" karur"/>

<RamaPrabha name = "vaish" place="somewhere in KA "/>

<RamaPrabha :name = " course1" :place="course2 "/>

export default {

  name: 'App',

  components:{

             // eslint-disable-next-line vue/no-unused-components

             RamaPrabha,

  },

  data(){   return{

 course1:" web development",

 course2:" cloud tech",

}},};

To specify what type the props is the props array can be replaced by props object in which the **value** can be **props name** and the **keys** can be **props type**.

<new-com movie="Enthiran"   budget="200" :Status="true" > </new-com >

   </div>   //camelCase and kebab case both are valid

</template>

//in child component

<template>

<div>

<h2> The movie {{movie}}  taken by {{budget}} crore budget is that worth watching ?? {{Status ? ‘ Yes‘ : ‘ no’}}</h2>

</div>

</template>

export default {

        name: 'NewCom',

            movie:String,

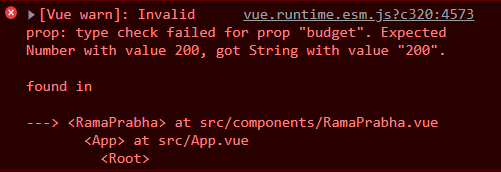
            budget:Number,

            Status:Boolean

        }

    };

For the movie property correct type of value is given but for Budget instead of number string is specified.



<new-com movie="Enthiran"   :budget="200" :Status="true" > </new-com >

Replacing the budget property with v-bind it will become a number.

* The value of the props object can also be given as objects like the movie can be object with its keys as type, default value, and whether it is required or not.
* If the value is not there for the movie props then default value will be displayed. Incase of the value specification , the value overrides the default value

<new-com :budget="200" :Status="true" > </new-com >

//at child component

export default {

        name: 'NewCom',

             props:{

            movie:{

                type: String,

                required: true,

            },

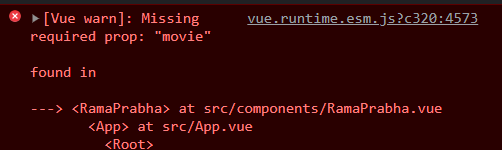
            budget:Number,

            Status:Boolean

        }

    };

Since the movie is not mentioned in the parent component and also as it is required it throws an error. If the default specified, error will be error thrown as same the value will be overridden by the default value



<new-com :budget="200" :Status="true" > </new-com >

//at child component

export default {

        name: 'NewCom',

             props:{

            movie:{

                type: String,

                required: true,

                default:" default movie name "

            },

            budget:Number,

            Status:Boolean

        }

output:

**Non props attribute** :

It is passed to a component but does not have a corresponding property defined in the props option (ex., id, class, style)

As they can be specified to a component similar to props

<new-com id=" id class "  :budget="200" :Status="true" > </new-com >

In the child component there is only one root node called <div>. so after specifying this non props to the parent , we can see it added to the root node of the child component. Incase , if there is no root node , then this id will not be added to the elements



If we want to apply the attribute id to the movie alone in parent with special component properties as its value , we can specify it in the child component itself

$attrs to bind the non props for our desired nodes

<template>

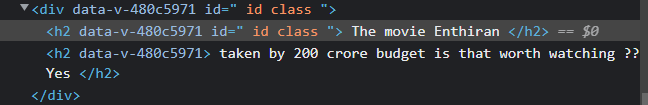
    <div>

    <h2 v-bind="$attrs"> The movie {{movie}} </h2>

    <h2> taken by {{budget}} crore budget is that worth watching ?? {{ Status ? 'Yes ':'No' }}</h2>

    </div>

</template>



As specified here , the id is present in div also but if we don’t require it at the root node we can remove it by adding InheritAttrs property to false.

    export default {

        name: 'RamaPrabha',

       // props: ["name","place"],

        props:{

            movie:{

                type: String,

                required: true,

                default:" default movie name "

            },

            budget:Number,

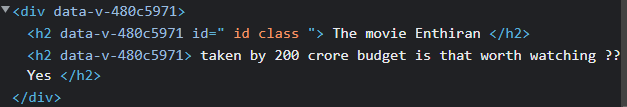
            Status:Boolean

        },

        inheritAttrs:false,

    };

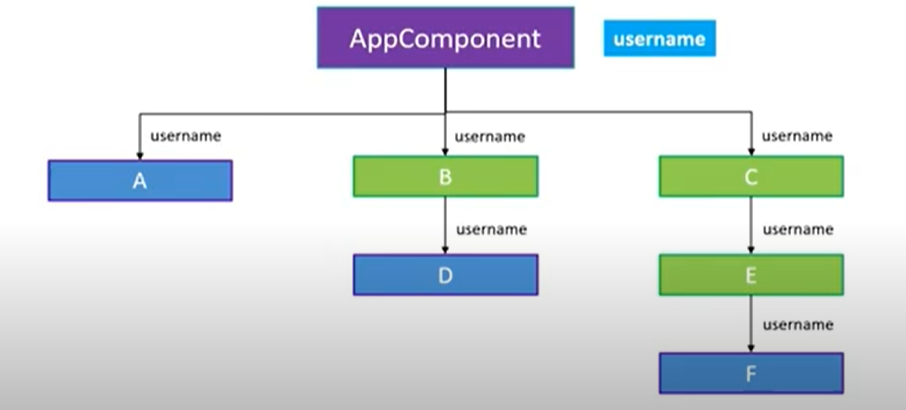
Output:



**Provide and Inject** :

For a scenario in which we have to print a user name in Component A , D, E which is maintained as a props in app component. So we need to pass down the username as a **prop** down the components manually. For A straightly from App, but for D, the props passage is from App 🡪 B and then B 🡪 D. Similarly for C, App🡪C, C🡪 E and E🡪F. Even though, B C E don’t need the username props but for the sake of D, F it has to be passed through them.

To simplify this, by directing the required props to required component without manually Provide and Inject API introduced.



**Components Nesting :**

Individually create three components, make them nested by importing the child in its parent. Like import F in E and then E in C an then C in App component. And also include the imported components under components property in export default.

App component.vue :

<template>

  <div id="app">

<h1>Provide and inject</h1>

<h2>App component displaying user name: {{username}}</h2>

<component-c/>

   </div>

</template>

<script>

import componentC from './components/componentC.vue'

export default {

  name: 'App',

  components:{ componentC,  },

data(){

return{} },

methods:{},

computed:{},

watch:{},

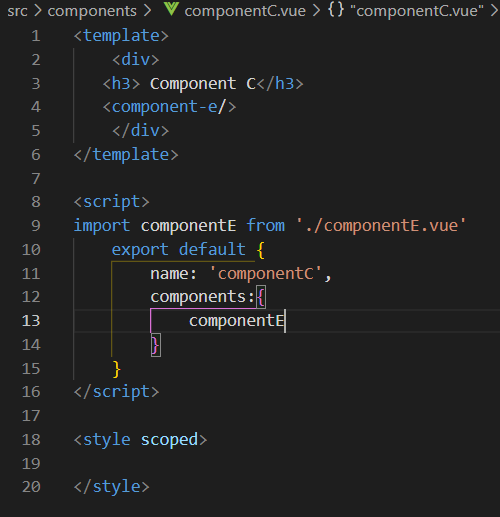
 provide:{

      username:"Ramaprabha R",

    },

  };

componentC.vue componentE.vue



componentF.vue

Two steps :

1. Provide the value in App component
2. Inject the value in required component

In the App component , a property called Provide is defined as object in the export default.

<template>

<div>

<h1>Provide and inject</h1>

<h2>App component displaying user name: {{username}}</h2>

<component-c/>

   </div>

</template>

export default {

  name: 'App',

  components:{

             // eslint-disable-next-line vue/no-unused-components

             RamaPrabha,NewCom,componentC,

  },

provide:{

      username:"Ramaprabha R",

    }

  };

In required child component, Use the property Inject as array within export default

Then bind in template with mustache syntax.

<template> //componentF

    <div>

<h3> Component F username is : {{ username}}</h3>

    </div>

</template>

<script>

    export default {

         name: 'componentF',

         inject:["username"]

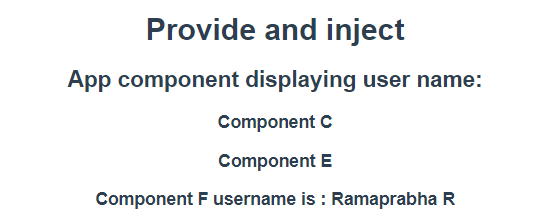
    }

</script>

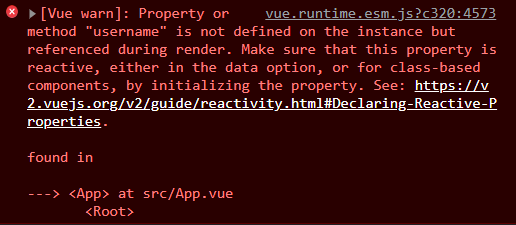
<style scoped>

</style>

This will display the username in the Component F , but not in the App component.



Because,



To make it display , define a name in data property, then change the **provide** into a function returning the name (username:this.name).

<template>

<div>

<h1>Provide and inject</h1>

<h2>App component displaying user name: {{componentName}}</h2>

<component-c/>

   </div>

</template>

export default {

  name: 'App',

  components:{

             // eslint-disable-next-line vue/no-unused-components

             RamaPrabha,NewCom,componentC,

  },

data(){

return{

componentName:”RamaprabhaR”

}

provide(){

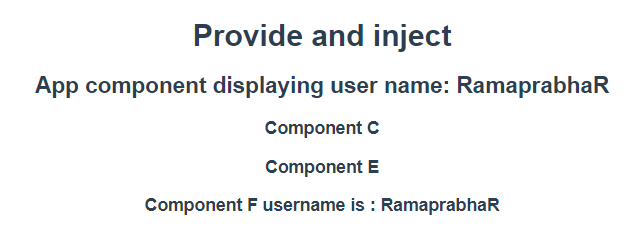
return{

      username:this.componentName}

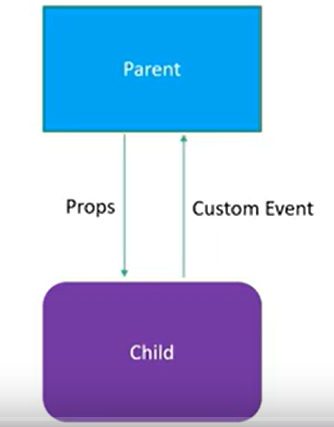
    }

  };

Output:

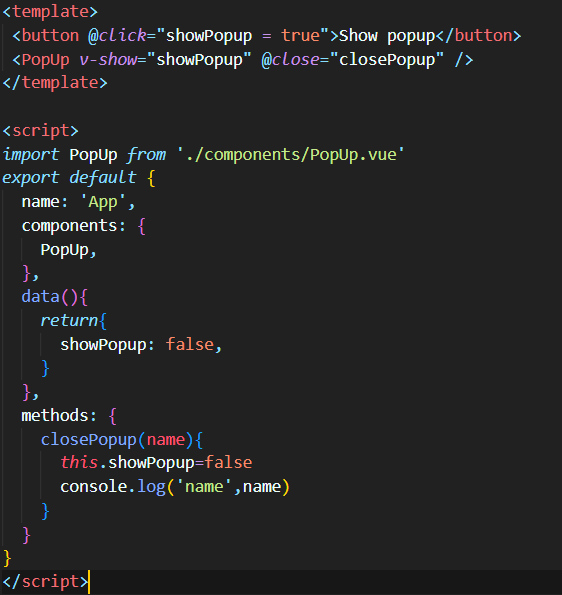


**Custom Component Events**

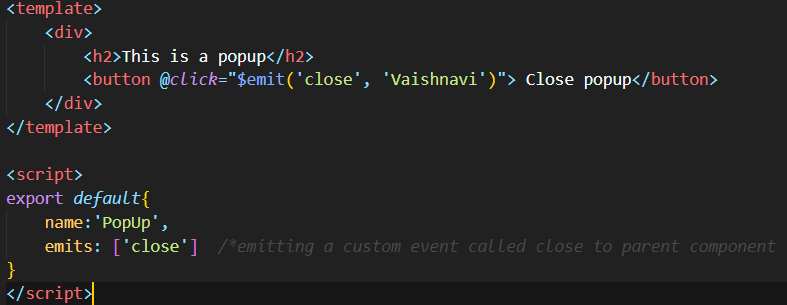
****

**Custom event:** To communicate from child component to parent component.

**App.vue**

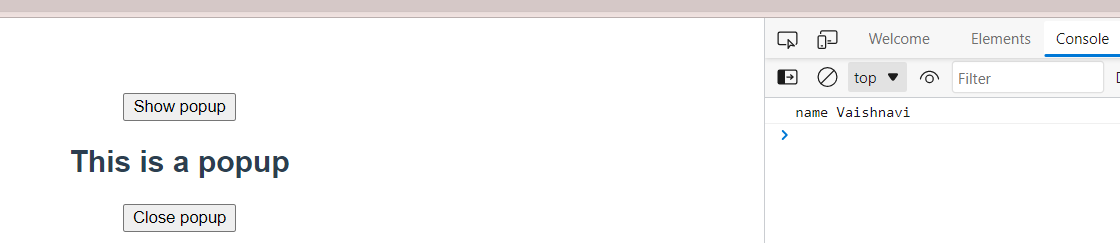
****

**PopUp.vue**

****

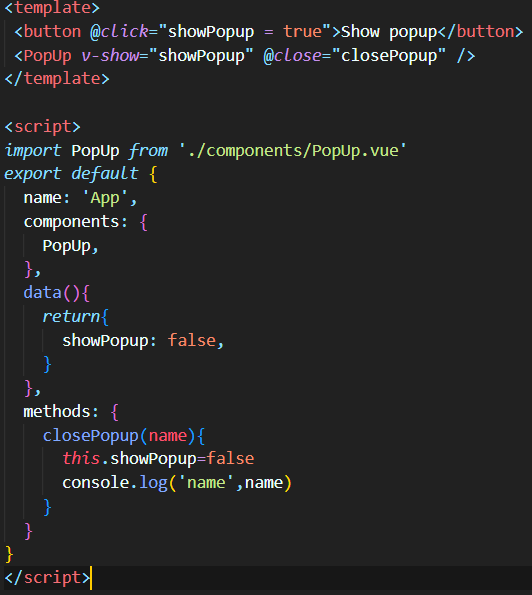
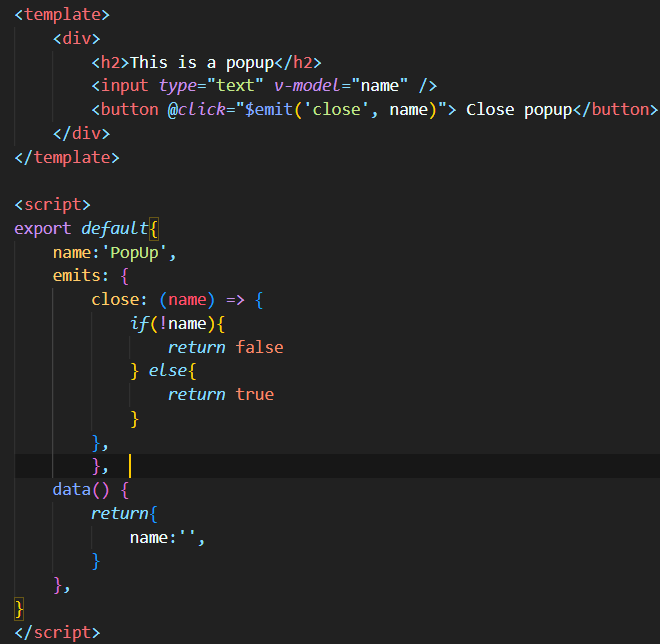
Specify the emits option in the child component and emit an event using the dollar instance property, bind to the custom event using the event binding in the parent component and assign the appropriate handler, any data that has to be sent from child component to parent can be included as the second argument to dollar emit.

**Output**

****

**Validating Custom events emitted from child component**

**App.vue PopUp.Vue**

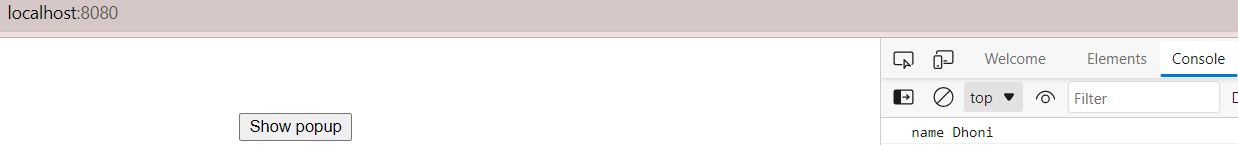
** **

**Output:**

Warning occurs in console while clicking close popup with an empty field.

****

****

****

**SLOTS**

Props allow you to re-use components by passing in different data

Although props are great for re-usability, we do have a strict parent-child relationship

The child will always be in the control of the HTML content and the parent can only pass in different data values.

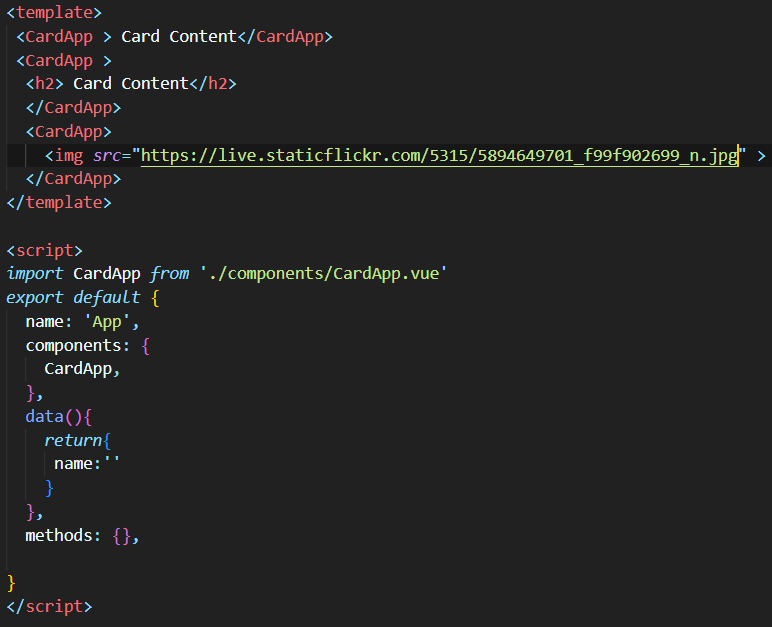
Slots on other hand are more powerful.They allow you to re-use a component

They allow the parent component to control the content inside the child component

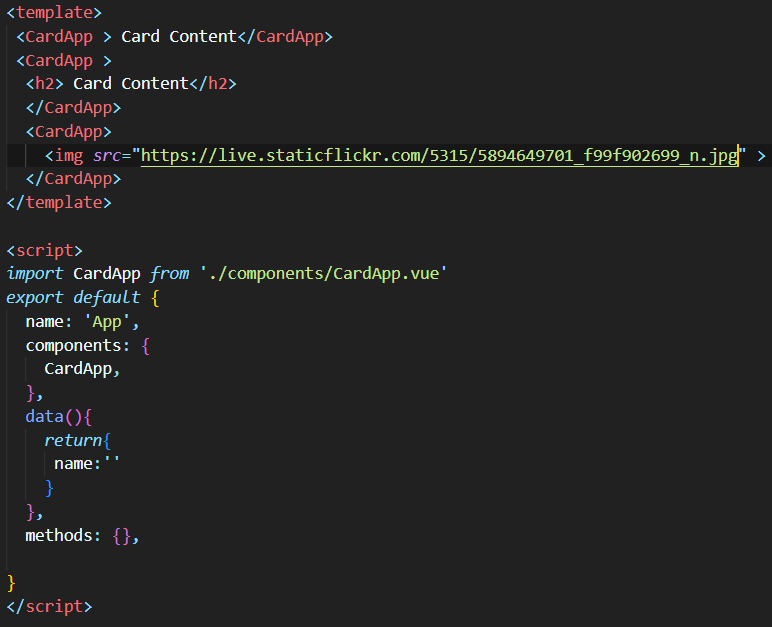
Slots allow a parent component to embed any content in a child component including HTML elements.

In simple terms, slots are openings within a component similar to props. Unlike props, they not only take data, but can also take other markups or even other components. Their major difference is that with props, the parent can only pass the data down to the child without any control over how it will be rendered. But with slots, the parent can determine exactly how the data should be rendered, or even pass down another component.

App.vue

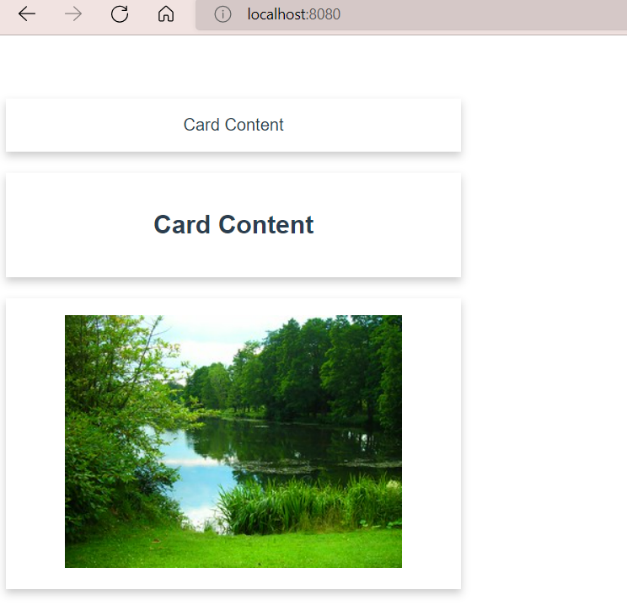


CardApp.vue



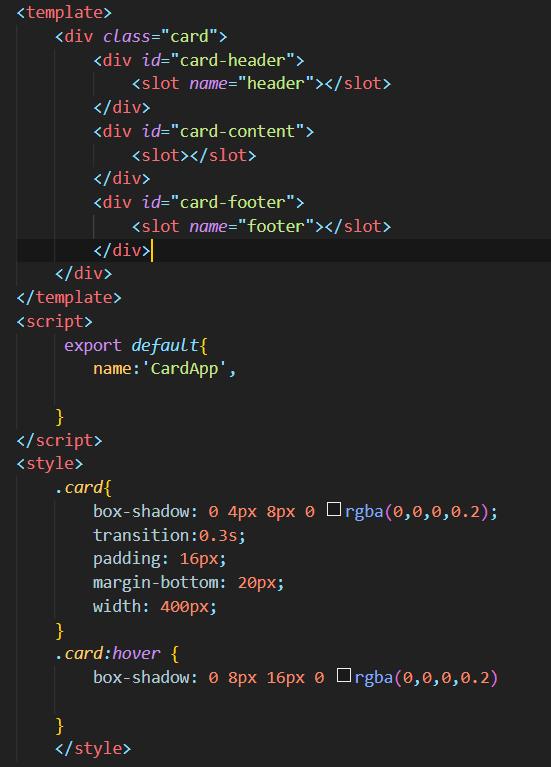
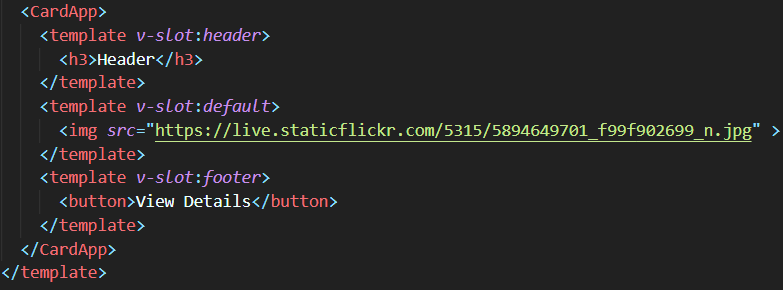
Slots give greater control in terms of re-using and composing new components in Vue.

Output

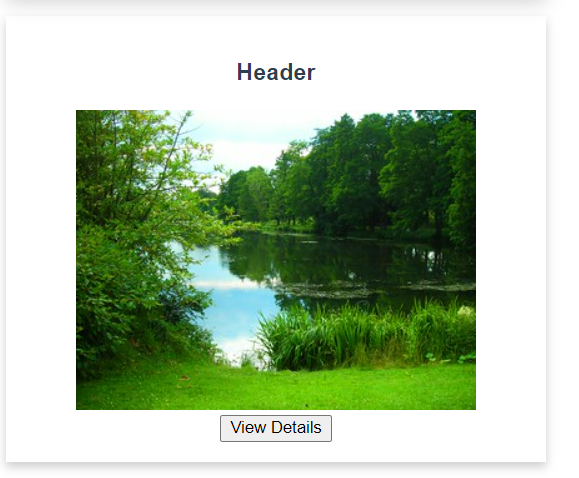


**Named Slots**

**CardApp.vue App.vue**

 ****

**Output**



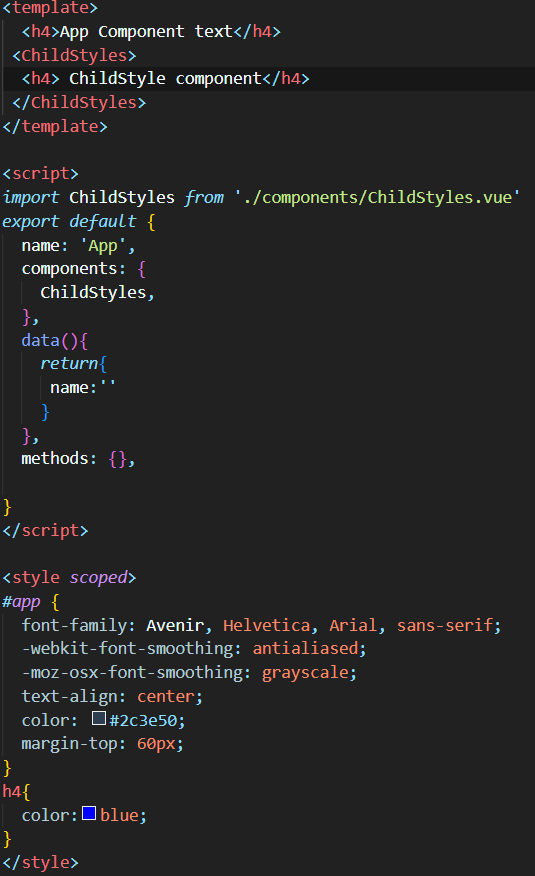
**Component styles**

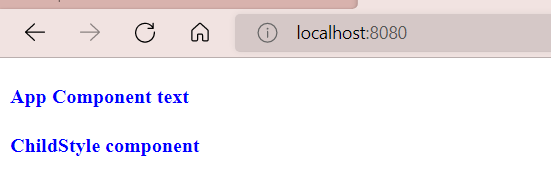
1. By default, Component styles are applied globally in the component tree.
2. Scoped attribute ensures a component’s CSS will apply only to its own HTML elements

“*With scoped, the parent component’s styles will not leak into child components. Incase, if the child component style is not scoped but only parent styles is scoped means, in that case parent component style will be applied to child component.”*

1. When using slots, the parent components styles are applied and not the child component styles, even though the content is embedded inside the child component.

App.vue ChildStyles.vue



**Dynamic components**

Vue dynamic components enable users to switch between two or more components without routing, and even retain the state of data when switching back to the initial component.

The central idea is to let users dynamically mount and unmount components in the user interface without using routers

Why dynamic component?

* When designing your user interface, you’ll want some form of flexibility to show or hide nested components based on the application state. Dynamic components provide that platform in an efficient and simple way.
* The feature saves you from a lot of code since you can easily achieve dynamic components with Vue conditional structures such as v-if and v-else. You can use conditional structures to achieve dynamic components by using a placeholder approach to easily bind logic to the component.

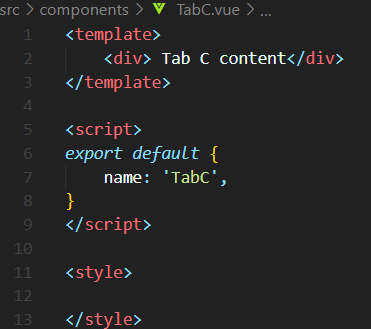
**Syntax**

<component *v-bind*:*is*=”currentComponent”></component>

**App.vue TAB A**

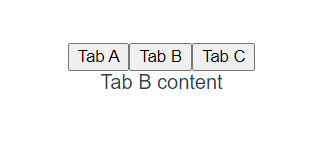
** **

**TAB B TAB C**

** **

To switch between components we can make use of component HTML tag with the is attribute to inform vue about the component that has to be rendered. Dynamic component also reduce the amount of code to be written.

**OUTPUT**

****

**Keep-alive**

To store the data, Vue provides a template element called **keep-alive**. Using keep-alive, you can ensure your component state stays exactly as you left it after you switch back from one component to the other.

For example, if you click on a link or enter a value in a text box and then switch components, keep-alive brings you back to the same link or text box you were using when you switch back.

To enable keep-alive, go to the template section of your app.vue file and wrap the component element with the keep-alive element.

<keep-alive>

    <component :*is*="activeTab"> </component>

  </keep-alive>

**TELEPORT COMPONENT**

Teleport can be used while working with components like modals, notification etc where their position in the DOM is important. If you use a modal inside a deeply nested element, for instance, the CSS properties on the parent will affect the styles on the modal.

Teleport specifies exactly where in the DOM we want to render the modal or any other piece of HTML. All without worrying about managing global state or creating a fresh new component for the modal.

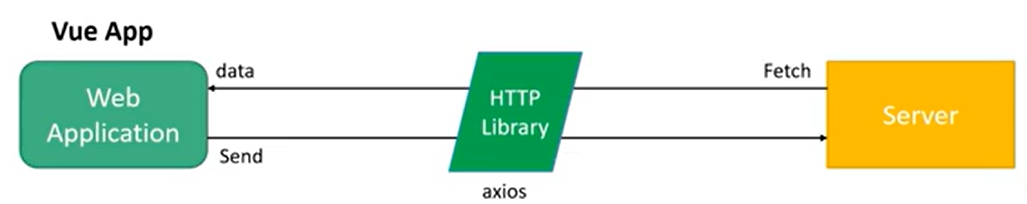
**Working**

The **<teleport>** tag takes a **to** attribute that specifies where in the DOM you want to teleport an element to.

It is recommended putting it below the body tag in the index.html file of public directory.

Refer: <https://blog.logrocket.com/positioning-elements-with-vue-3-teleport/>

**VUE and HTTP**

****

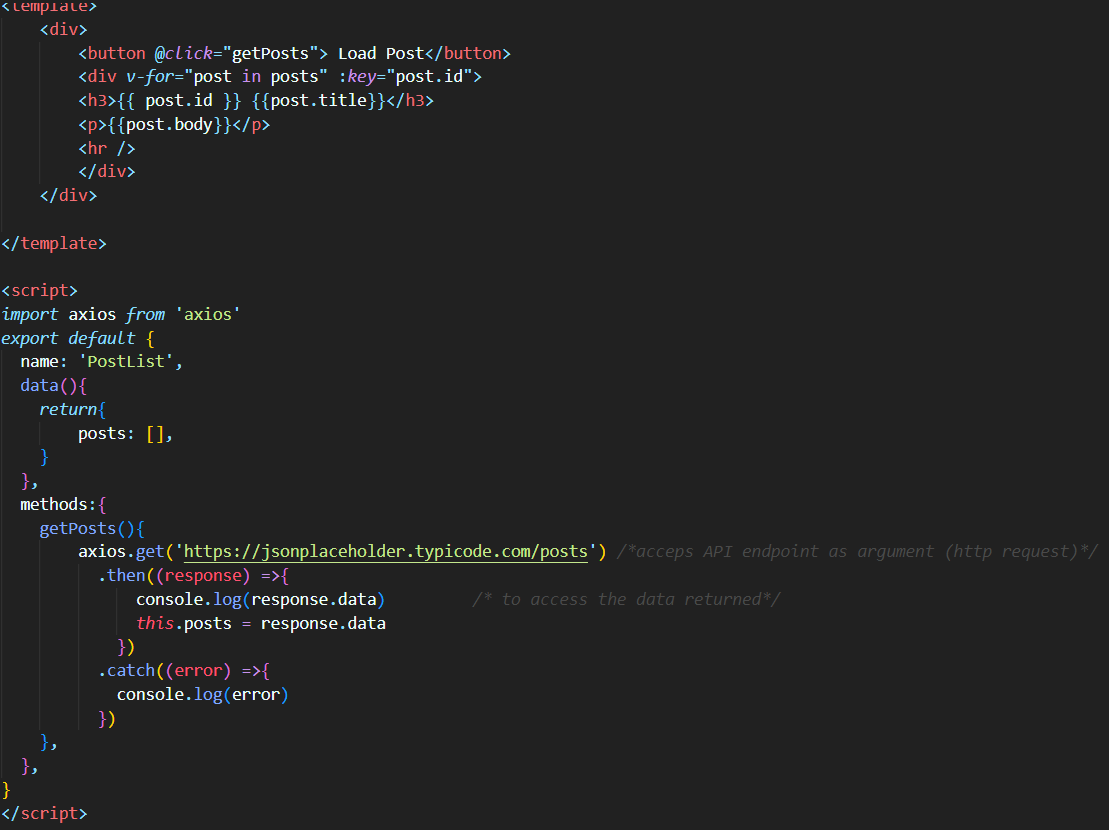
* Vue does not have a particular way to fetch or send data to the server.
* Vue Components simply read data properties and bind them to UI.
* Therefore, to use some data from the server, you just have to get data into your component’s data object.
* As Vue does not handle such things, additional HTTP library like axios comes into picture.

**Commands to add axios**

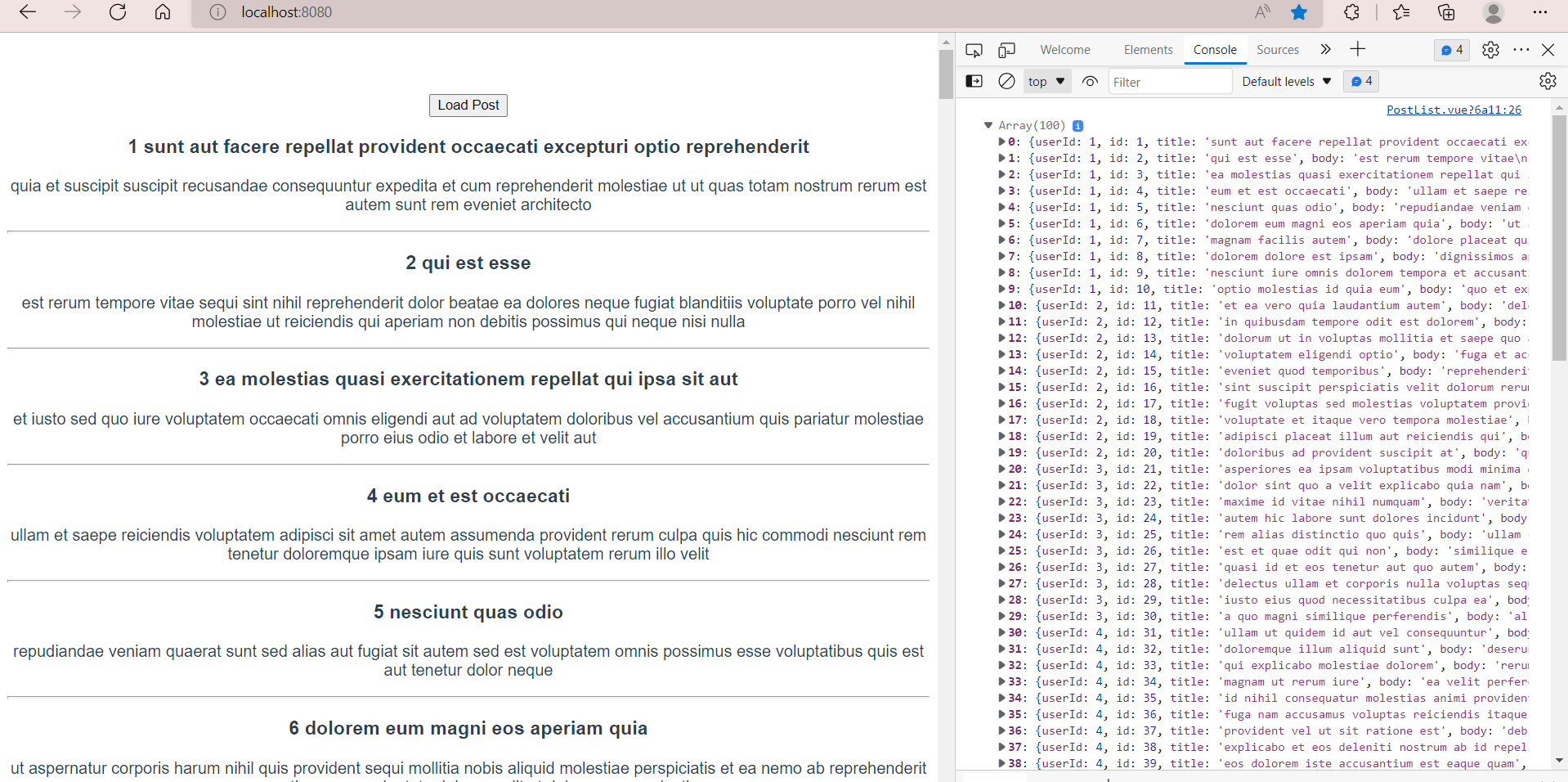
* **yarn add axios (or)**
* **npm i axios**

**To fetch data:** [**https://jsonplaceholder.typicode.com/**](https://jsonplaceholder.typicode.com/)

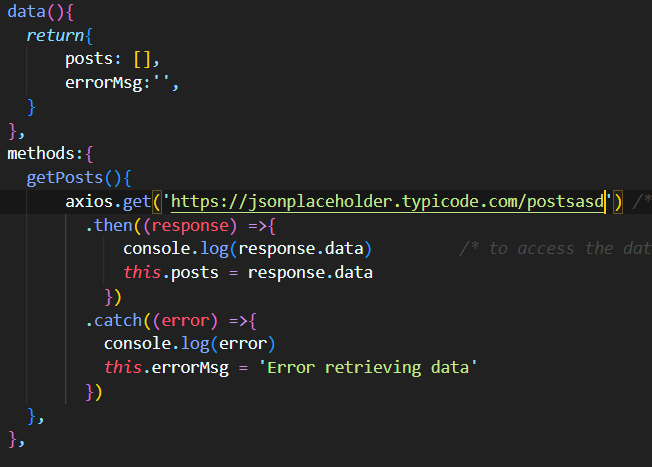
**Get method**

****

**Output**

****

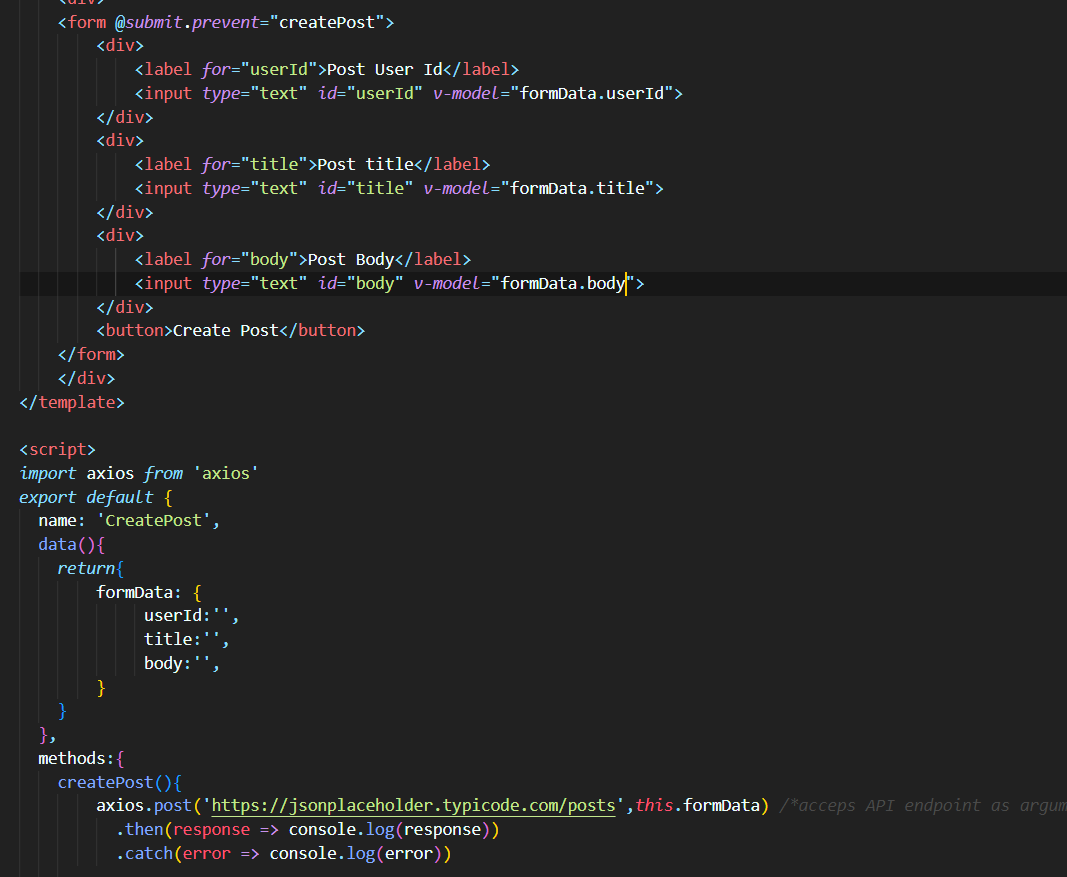
**If the argument passed in get method is invalid means, error message will be displayed as below.**

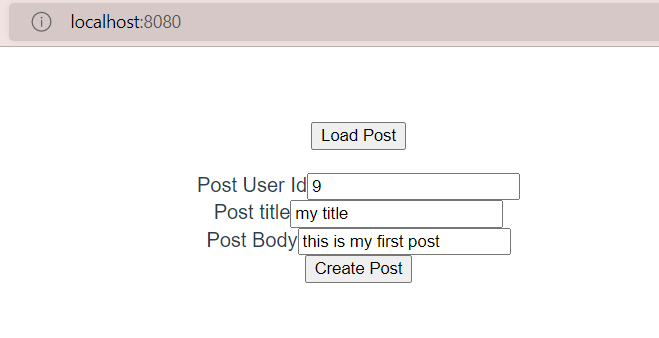
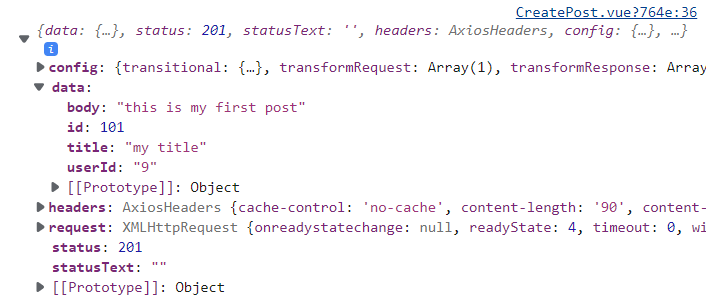
****

**The errorMsg should be rendered in template part.**

****

**POST Method**

****

** **