Vue.js

.Vue file => responsible for the portion of the UI that uses HTML

1. Hello world .Vue – contains links
2. App.vue - logo and contains Helloworld.vue

Consists of three top level language blocks

Template => HTML of the UI

Script=> JS

Styles=> css block

Browser don’t understands the .vue file so vue loader parses and extract all three blocks and pipe through other loaders then assessable back together for browser to understand . vue Cli takes care of all those above

Components :

.vue file is called the single file component

Moreover a component is a .vue file

APP.vue :

The work on the .vue file is that wiring up the logic to markup (script—>html)

Declarative programming approach is that specification of the connection (binding ) of the script to template.

**Binding:**

**1.Binding text from script to template :**

1. Mustache syntax => {{ }} Binding text using mustache syntax is text interpolation ( similar to template strings).

<template>

  <div id="app">

   <h1> Hii !! This is {{name}} {{ lastname}}'s Practice Page </h1>

  </div>

</template>

export default {

  name: 'App',

  data(){return{

   name: "rama" ,

   lastname:"prabha"

  };

  } ,

  };

**2. properties can also be bound. Faster than directive**

1. Directive v- to bind text is v-text

*template: <div> v-text=”name” </div> =>output displayed is rama*

adding another property to the partial static text directive is not good

it can only bind full text content.

Writing 2 words only using properties cannot be bound like in mustache syntax(above example where **name** and **initial** is used ).

<template>

  <div id="app">

   <h1> Hii !! This is {{  name}}{{lastname}}'s Practice Page</h1>

   <h2 v-text="directive"></h2>

  </div>

</template>

export default {

  name: 'App',

  data(){return{

   name: " Rama" ,

   lastname:"prabha",

   directive:" Directive v- text is implemented here...which can only be used for full text content",

};

  },

  };

**2.Binding Html** for rendering the html (as like in editors bold italics etc.,)

v-html=” native”

<template>

  <div id="app">

   <h3 v-html =" bindingHtml">   </h3>

  </div>

</template>

export default {

  name: 'App',

  data(){return{

   bindingHtml: '<u> Binding HTML Here  </u> ' };

  },

  };

**3.Binding attributes** id class style disabled (Boolean)

Directive: v-bind

In template **: <h1 v-bind:id=” headed”>Heading</h1>**

Which is similar to **<h1 id=”headed”>Heading</h1>**

At script : headed : heading

Idname/placeholder id

<template>

  <div id="app">

   <h4 v-bind:id="placeholderforID"> The Binding of ID attributes</h4>

  </div>

</template>

export default {

  name: 'App',

  data(){return{

   placeholderforID:'realIdName',

   };

  },

  };

4.**Binding class:**

Defining class in styles

can have both static and dynamic class on same element

<template>

  <div id="app">

   <button class="buttonHolder"> Static class HTML </button><br>

   <button v-bind:class="buttonHolderVue"> VUE  </button>

   </div>

</template>

export default {

  name: 'App',

  data(){return{

   buttonHolderVue:'ButtonClassVue'

   };

  },

  };

**5conditional binding**

can use ternary operator for making use of dynamic classes.

One class for true and another for false

If lots of classes to be added dynamically can use array and objects

Conditions can also be specified in the array and objects

<template>

  <div id="app">

   <button type="radio" v-bind:class=" 'enabled' ? 'trueClass' : 'falseClass ' "> VUE  </button>

  </div>

</template>

export default {

  name: 'App',

  data(){return{

   enabled:true,

   buttonHolderVue:'ButtonClassVue'

   };   },  };

**5.Binding styles:**

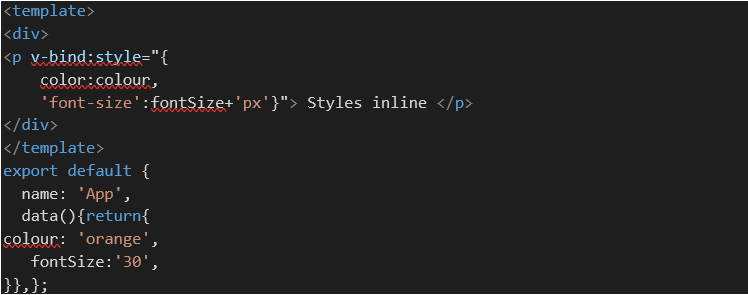
1. Object syntax :

*<h1 v-bind:style=”{Color:colorValueInScript,*

*‘font-size’ : fontSizeInScript }”>Styles</h1>*

//wrapped in quotes because font-size is not a single word

And also can use camelCase **fontSize**

****

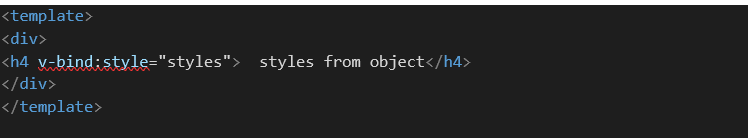
1. Bind using js object

Specifying the styles in an object ar script and then bindingit to the template . *……..*

*<h1 v-bind:style=”headingStyle”> Object styles</h1>*

*headingStyle:{*

*Color:yellow,*

*‘font-size’ : 50px;}*

**

1. Array syntax:

Style attributes are given in the form of array

Last style given priority if conflicts occurs

< *h1 v-bind:style=”[headingStyle,anotherstyle]”> Object styles</h1>*

**V-bind shorthand:**

Can remove the v-bind alone and **:** with there can be excuted flawlessly.

**Rendering in Vue**

* Conditional rendering
* List rendering

Conditional rendering :

To show / hide some html depending on conditions.

1. v-if
2. v-else
3. v-else-if

<template>

  <div id="app">

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

    <h2 v-else-if="num<0"> the number is negative</h2>

    <h2 v-else-if="num>0"> the number is positive</h2>

    <h2 v-else> not a number </h2>

</div>

</template>

export default {

  name: 'App',

  data(){return{

   num: '0'

   };

  },

  };

1. v-show – as same as v-if

but if condition evaluates to false , the element v-if will be completely removed from the DOM but v-show continues to stay in the DOM without displaying it. V-show is best for toggling b/w displaying and hiding. Unlike using v-if for mounting and unmounting in the DOM.

List rendering :

v-for –

array of strings

array of objects

array of array

objects

</template>

   </div>

 <h4 v-for=" (name,index) in names" :key="name"> {{index}}{{name}} </h4>

    <div v-for="edu in education" :key=" edu.school">

      <h4> {{edu.school}}</h4>

      <h3 v-for="n in edu.percent" :key="n"> {{n}}

      </h3>

    </div>

    <div v-for="fam in family" :key="fam.names">

      <h1> {{fam.names}}</h1>

      <h2 v-for="w in fam.whereabouts" :key="w"> {{w}}</h2>

    </div>

    <h4 v-for="place in karur" :key="place">{{place}}</h4>

   </div>

</template>

export default {

  name: 'App',

  data(){return{

names:['ramaprabha','21','karur'], //array

   education:[

    {

       school:"star",percent:["10th","98%"]},

       { school:"star",percent:["12th","96%"]}    //array of objects

   ],

   family:[

        { names:"dhivya",

           whereabouts:["2nd","studying", "chennai"] },

        { names :"rama",

          whereabouts:["1 st","working","karur"] },   ],//array of array

   karur:{

    temp:"38 c",

    constituency:"4",

    day:"sept 30th"   } //objects

   };

  },

  };

Key attribute :

Should have unique value

The keys help the vue identify which items in the list has changed / added / placed.

* A hint for vues virtual DOM ALGORITHM to identify nodes

Absence of keys suits when the list render output does not rely on the temporary DOM state or child component state.

Conditonal list rendering :

<template>

  <div id="app">

<template v-for="f in familymine" >

      <h1 v-if="name==='Rajendran'" :key="f.names">{{f}}</h1>

    </template>

    </template>

</div>

 export default {

  name: 'App',

  data(){return{

familymine:{

      names:["Rajendran","Gunavathi","Ramaprabha","Dhivya"]

    }

}}

};

**Methods:**

Using arrow functions leads to loss the vue binding based on key

 </template>

<div>

    <h2>Methods  {{ add()}}</h2>

     <h2>Methods  {{ add1(5,6)}}</h2>

      <h2>Methods  {{ mult(6)}}</h2>

       <h2>Methods  {{ mult1(value)}}</h2>

   </div>

</template>

    export default {

  name: 'App',

  data(){

return{

emultiargs:"5",

    value:"4"

   }

     },

 methods:{

add(){

return 5+2; //adding an returning values

},

add1(a,b){

return a+b; //by parameters

},

mult(num){

  return num\*this.multiargs; //to use a variable for calculation this keyword is //muust

},

mult1(num){

  return num\*this.multiargs; //vlue is passed as parameters doesn’t require the this

}

    }

  };

</script>

**Event Handing :**

v-on for event listening

* Inline event handlers

<template>

<div>

<h1> Event handlers </h1>

   <h4> {{name}}</h4>

   <button v-on:click = " name ='prabha'  "> click me</button>

</div>

</template>

* Method to execute at method property

<template>

<div>

<h1> Event handlers </h1>

<h4> {{name1}}</h4>

   <button v-on:click = change()  > click me</button>

</div>

</template>

export default {

  name: 'App',

  data(){return{  name1:"rama"}},

methods:{

change(){

  return this.name1="last name prabha";

}

    }

  };

**Shorthand for event handling:**

v-on:click === @click

   <button @click = change()  > click me</button>

Vue automatically passes event object as parameter to event handler method

The above statement is true in case , when the method(without parameters) is called on the occasion of an event.

   <button v-on:click = change()  > click me</button>

If the method contains parameter while calling it can’t be accessed as specified above . it requires $

 <button v-on:click = change(name,$event)  > click me</button>

methods:{

change(name,event){

  return this.name1="last name prabha";

}

**Multiple event handlers :**

Multiple event can be done on the same element with same event (click) by adding the methods using comma(,)

<button v-on:click = change(event) ,increment(1,$event) > click me</button>

**Form Handling :**

Form controls in template

Form data in script => have to be in sync since the

To propagate the data in the script

Form controls---------------------------------> form data

To load the data back to template incase of changes

Form controls---------------------------------> form data

**V-model** is used to bind this as a two-way process

<template>

<div>

<h2> Form Handling </h2>

<form @submit="submitfn()">

<label for="name"> NAME </label>

<input type="text" id="name"  v-model="formvalues.formname">

<br>

<label for="gender"> Gender </label>

  <input type="radio"   value=" male " v-model="formvalues.gender" > MAle

  <input type="radio"   value=" feale "  v-model="formvalues.gender"> feMAle

  <br>

<label for=" qualification">Select your qualification </label>

<br>

<select id=" qualification" multiple v-model="formvalues.qualification">

<option value=" 10th"> 10th</option>

<option value=" 12th"> 12th</option>

<option value=" ug"> UG</option>

<option value=" pg"> PG</option>

</select>

<input type="checkbox" id=" submit" >

<label for ="submit"> i agree</label>

<button > Submit form </button>

</form>

   </div>

</template>

export default {

  name: 'App',

  data(){return{ formvalues:{

         formname:"",

         gender:"",

        qualification:[],

        ug:"",

    },

}

},

methods:{

submitfn(Event){

  Event.preventDefault();

  console.log( " form submitted " + this.formvalues)

}

    }

  };

**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 method 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
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,

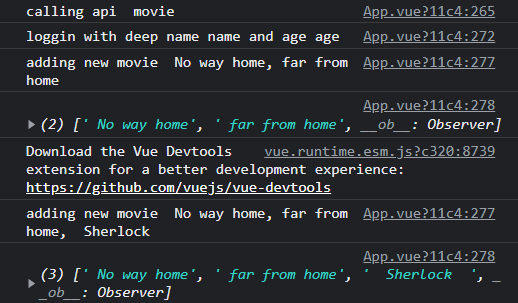
    immediate:true,

   }

       }

  };

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

    },

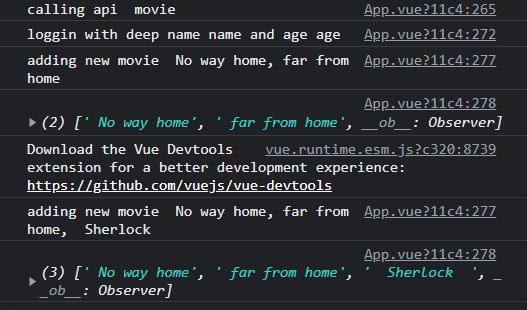
    immediate:true,

   }

       }

  };

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,

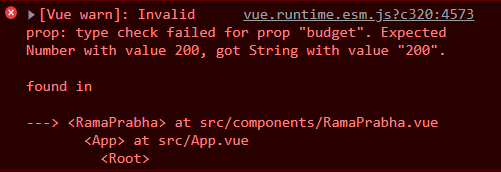
            budget:Number,

            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,

            },

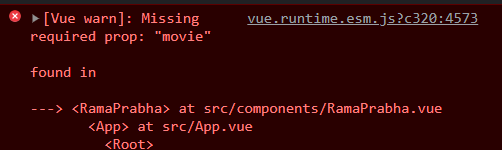
            budget:Number,

            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>

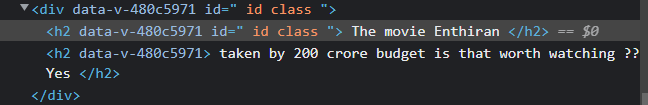
    <div>

    <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,

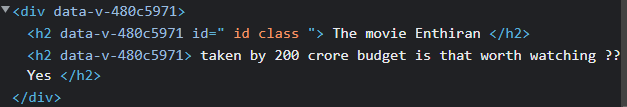
            Status:Boolean

        },

        inheritAttrs:false,

    };

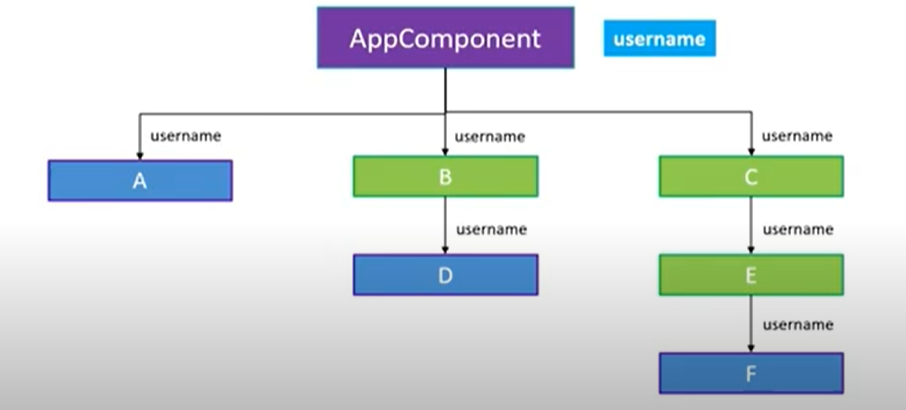
Output:



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 defau

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:{},

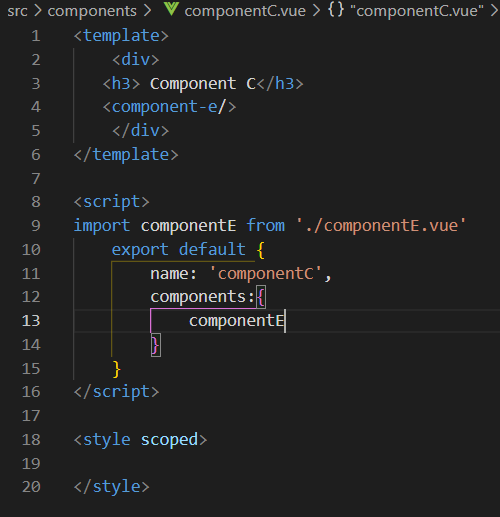
 provide:{

      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"]

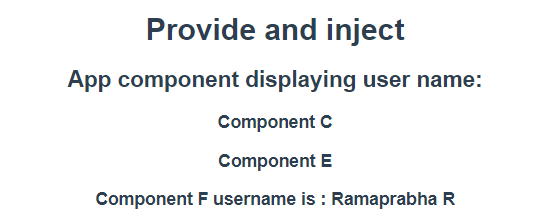
    }

</script>

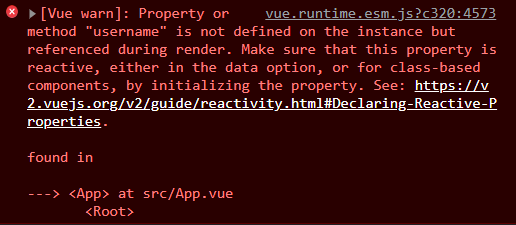
<style scoped>

</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(){

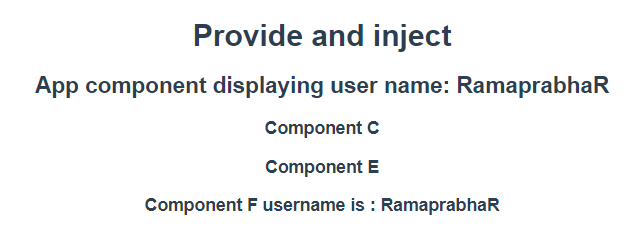
return{

      username:this.compoentName}

    }

  };

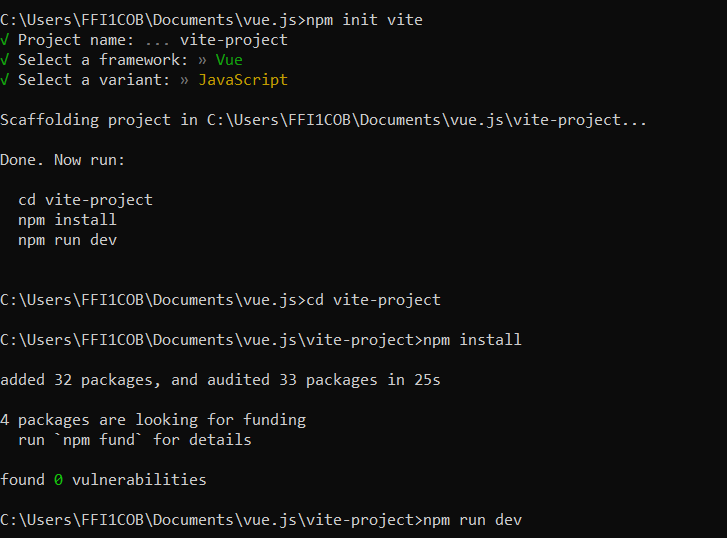
Output:



Vite :

* build tool that provides scaffolding and development servers like CLI.
* Has own dev server and not based on webpack like CLI.
* Faster since no web packs

Installation :



**Component Events:**

Props 🡪 communicate from parent component to child component

Custom events🡪 child to parent.

Frpm App.vue :

<h1> Pop-up</h1>

<button @click=" showPopup=true"> Open pop-up</button>

<pop-up v-show="showPopup" @close=" showPopup =false" />

   </div>

</template>

Initially the pop-up shouldn’t be shown so it is set to false in its tag itself. The showpopup is bind with pop-up tag for toggling its display.

 Data():{return{

showPopup:false,

     }

     },

It has be displayed only when the close button is clicked. When close button is clicked the poop-up has to be hidded /closed . So for that custom event an option called emits in included in chid component with the event close. Which is then binded to the button for it to happen, using like $emit(‘event’). This is how an event is specified in the component. To make it listen in the parent, the event close here is a custom event (@close) now, so it is listened like the DOM event @click)

Creating a custom event: " $emit('close')"

From PopUp :

<template>

    <div>

<h3> this a pop-up from the PopUp component</h3>

<button @click=" $emit('close')"> Close</button>

    </div>

</template>

<script>

    export default {

        name:"PopUp",

        emits:['close']

    }

</script>

<style scoped>

</style>

Emits is an array of events that can emit to the parent.

Passing data along with custom event:

On closing the pop-up we are sending a name to display it on te console. can be done using the second parameter from the $emit('close’) . At the pop-up tag , on listening to the event change it to function and specify the showpuop to false and the second argument is passed in the form of argument.

<pop-up v-show="showPopup" @close=" closePopup " />

Method:{

closePopup(namefromevent){

this.showPopup=false;

console.log("Name passed :", namefromevent)

}

  }

Output:



Using the emit instance variable , to emit an event in the child and handle the event in the parent component .

**Validation of events:**

Validating custom events that are emitted from the child components.

Emits changed to object.

Key🡪 custom event name ,

value 🡪 validation function receives the args specified when emitting the event

can return true/ false for the result of validation.

* Warns the user in the console.

**Component Events and V-model:**

Here an input area in a component can be inherited with styles and functions to all of the application by embedding the component into other components. But the name typed is not updated in the data property since we haven’t mentioned about the custom event to the v-model directive. (ie typed value getting stored is a custom event)

in the child , when we use the v-model in custom component it automatically receives props called **modelValue** which we need to specify in the props option of child to bind the props in input in the child using :value=”modelValue”.

For handling the input from the user , when we use the v-model in custom component it automatically listens to the event called **update:modelValue** that should be emitted in the tags.

**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.

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.

The slot tag in the child is substituted with the content within the card tag in the parent component while executing which helps the parent component in having the control over rendering of the content.

App.vue

<hr>

<h1> cards</h1>

<card-comp/>

<br>

<card-comp> <h1> card rendered by heading </h1></card-comp>

<br>

<card-comp ><h6> hello</h6></card-comp>

   </div>

</template>

CardComp.vue:

<template>

    <div class="cards">

 <h2> hiii</h2>

 <slot></slot>

    </div>

</template>

<script>

    export default {

        name:'CardComp',

    }

</script>

<style scoped>

.cards{

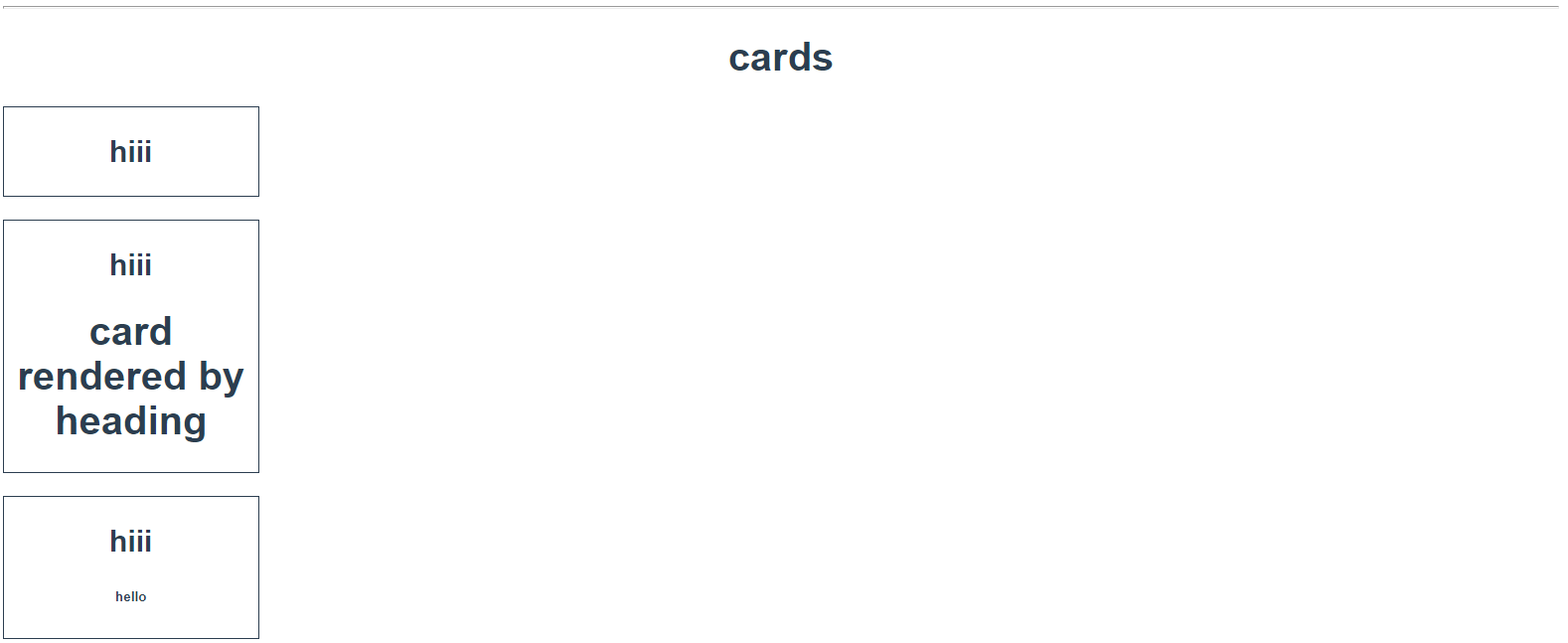
     border-width: 3px;

     border-style: double ;

     width: 200px;}

</style>

Output:



**Named slots:**

Named slots must use '<template>' on a custom element.

<card-comp>

  <template v-slot:heading >

    <h3 > this is heading </h3>

  </template>

  <template >

    <h6 v-slot:foot> Footer line</h6>

  </template>

</card-comp>

<template>

    <div class="cards">

 <h2> hiii</h2>

 <slot></slot>

    <div>

       <slot name="heading"></slot>

    </div>

    <div>

      <slot name="foot" > </slot>

    </div>

    </div>

</template>

<script>

    export default {

        name:'CardComp',

    }

</script>

<style scoped>

.cards{

     border-width: 3px;

     border-style: double ;

     width: 200px;}

</style>

Output:



To access the data property of the child component in parent component we use **Slop-props** which is obtained with template tag.