Vue Basics

Prof. Emanuele Panizzi

Vue.js playground

https://sfc.vuejs.org/

Hello World

```
<template>
<h1>
  Hello World!
  </h1>
</template>
```

- The <template> in Vue.js contains what will be rendered in the html page

Stepper

- We want to create a counter that the user can increment or decrement, step by step
- composed of a "–" button, the counter value, and a "+" button

Looks like this

```
<template>
<button>
  -
  </button>
  0
<button>
  +
  </button>
</template>
```

(<button> is an html element)

The Counter

- the counter should increase or decrease when the user presses the "+" or "−" button, respectively
- we need a variable there

```
<template>
<button>
  -
  </button>
  {{ counter }}
<button>
  +
  </button>
</template>
```

- double braces {{}} can contain any js expression

Definition of the counter variable

```
<script>
export default {
  data() {       // function that returns an object
    return {     // containing all the
      counter: 0 // variable definitions
    }
  }
}
</script>
```

- variables can be used:
  - in the <template>, e.g. {{ counter }}
  - in the <script>, e.g. this.counter

Reactive state

- variables defined by data() are called 'the reactive state'
- if their values change, the template is re-rendered
- like in the observer pattern

- in our example, if counter is incremented, the template is re-rendered and the new value is shown

Button clicked

- On button click we want to increment/decrement the counter
- we listen to the DOM click event, using the @click attribute[1]

```
<template>
<button @click="counter--"> - </button>
{{ counter }}
<button @click="counter++"> + </button>
</template>
```

[1] these Vue attributes are called directives

Directives

- the value of a directive can be any js expression
- e.g. counter++
- e.g. a function call like incr()

```
<template>
<button @click="decr()"> - </button>
{{ counter }}
<button @click="incr()"> + </button>
</template>
```

Methods

- a function call refers to a method to be defined in the script

```
...
    counter: 0,
  }
},
methods: {
  incr() {
    this.counter++
  },
  decr() {
    this.counter--
  }
}
...
```

Negative numbers in red

This is static style (always red):

```
<template>
<button @click="decr()"> - </button>
<span style="color:red">{{ counter }}</style>
<button @click="incr()"> + </button>
</template>
```

Negative numbers in red

This is dynamic style (style changes if variable dCol changes):

```
<template>
<button @click="decr()"> - </button>
<span :style="dCol">{{ counter }}</style>
<button @click="incr()"> + </button>
</template>
```

- note the : notation to express the binding
- dCol variable must be defined in the reactive state
- when dCol changes, the <template> is re-rendered

Definition and update of dCol

```
data() {
  return {
    counter: 0,
    dCol: 'color:black;'
  }
},
```


```
methods: {
  incr() {
    this.counter++
    if (this.counter >= 0) {
      this.dCol = 'color:black;';
    }
  },
  decr() {
    this.counter--
    if (this.counter < 0) {
      this.dCol = 'color:red;';
    }
  }
}
```

Change font size

```
// ... in incr():
  if (this.counter >= 0) {
    this.stepperStyle =
      'color:black;font-size:'+this.counter+'em;';

// ... in decr():
  if (this.counter < 0) {
    this.stepperStyle =
      'color:red;font-size:'+-this.counter+'em;';
```

- please note: dCol renamed stepperStyle
- em is a size unit in CSS

How it is rendered

Use a computed property

- keep the incr() and decr() functions focused on their task
- add a computed property to handle the color and size changes

```
methods: {
  incr() { this.counter++; },
  decr() { this.counter--; }
},
computed: {
  stepperStyle() {
    let color = this.counter < 0 ? 'red' : 'black';
    return 'color:' + color + '; font-size:' +
           Math.abs(this.counter) + 'em';
  }
}
```

Computed property

- variable-like syntax, e.g.

```
<span :style="stepperStyle">{{ counter }}</span>
```

- reactively computed from other properties and reactive state
- automatically updates when its dependencies change

Create Vue Component

- put all the above stuff in a file, call it Stepper.vue
- import Stepper.vue in the main file App.vue
- declare the component in the app's export:

```
export default {
components: {
  Stepper
},
```

- use the newly created element <Stepper>

Stepper.vue

App.vue Stepper.vue × +

```
<!-- STEPPER COMPONENT -->
<script>
export default {
  data() {
    return {
      counter: 0,
    }
  },
  methods: {
    incr() {
      this.counter++;
    },
    decr() {
      this.counter--;
    }
  },
  computed: {
    stepperStyle() {
      let color = this.counter < 0 ? 'red' : 'black';
      return 'color:'+color+'; font-size:'+Math.abs(this.counter)+'em';
    }
  },
}

</script>

<template>
  <button @click="decr">
  -
  </button>
  <span :style="stepperStyle">{{ counter }} </span>
<button @click="incr">
  +
  </button>
<br>
</template>
```

App.vue

App.vue Stepper.vue × + Import Map PREVIEW JS

```vue
<script>
import Stepper from './Stepper.vue';
  export default {
  components: {
    Stepper
  }
}
</script>

<template>
<h1>
  My page
  </h1>
<Stepper />
  <br>
<Stepper />
</template>
```

Component attributes

- pass arguments to components, as attributes

```
<!-- in the app -->
<Stepper title="My stepper">
```

```
// ... in the <script>
 },
 props: ['title']
}
```

```
<!-- ... in the <template> -->
<span style='background-color:yellow;'>
  {{ title }}:
</span>
```

Many components

- we can use many <Stepper> components
- each one is istantiated separately, no conflicts

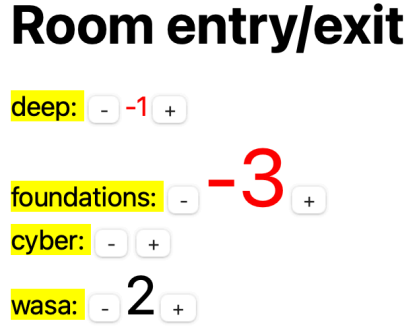
- let's use many steppers to count people that enters or exit rooms after the lessons started
- assume we have four lessons: wasa, deep learning, foundations, and cyber

Data definition in the app's <script>

```
<script>
import Stepper from './Stepper.vue';
  export default {
  components: {
    Stepper
  },
  data() {
    return {
      courses: ['deep','foundations','cyber','wasa']
    }
  }
}
</script>
```

v-for: directive in the app's <Stepper> element

```
<template>
<h1>
  Room entry/exit
  </h1>
  <Stepper v-for="course in courses" :title="course" />
</template>
```



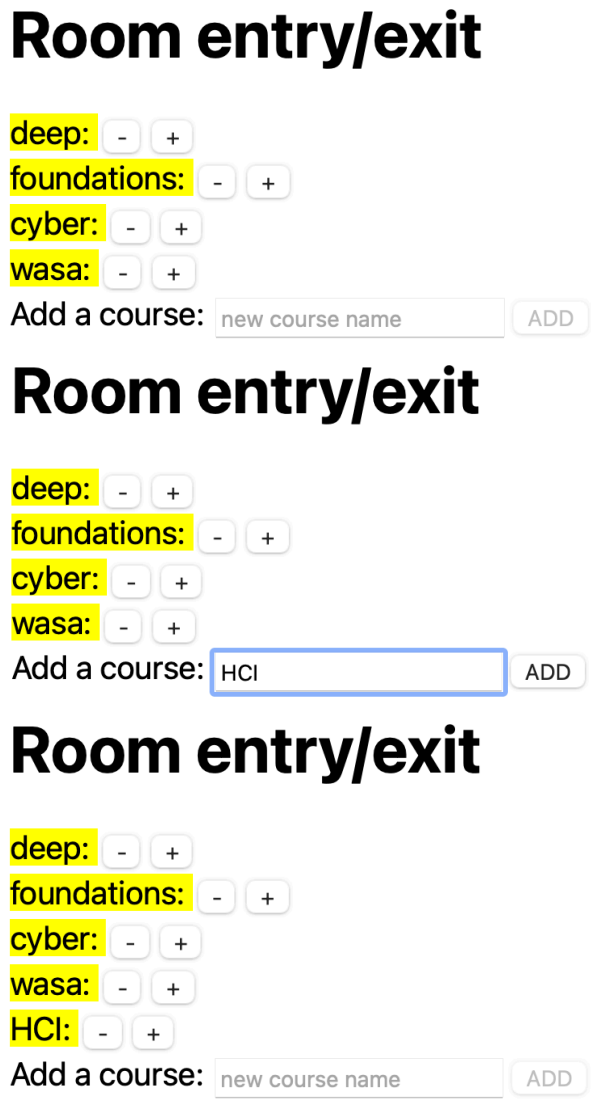
v-model

- HTML forms can collect user input
- Vue can bind input data to variables using the v-model directive
- Let's create a way to add other courses to our list

In the app's <template>

```
<!-- ... in the app's template -->

Add a course:
<input v-model="newCourse"
  placeholder="new course name">
<button
  :disabled="newCourse.length == 0"
  @click="add()">
  ADD
</button>
```



Logic to add new courses (<script>)

```
// ... in the app's script
  data() {
    return {
      courses: ['deep','foundations','cyber','wasa'],
      newCourse: ""
    }
  },
  methods: {
    add() {
      if (this.newCourse != "") {
        this.courses.push(this.newCourse);
      }
      this.newCourse = "";
    }
  }
```

Link to the playground of this lesson

Stepper playground

References

- https://vuejs.org