SECTION 02 - BASIC AND CORE CONCEPTS
DOM INTERACTION WITH VUE

Index

- Creating and connecting Vue app instances

```html
index.html > html > head
<script src="https://unpkg.com/vue@3.4.9/dist/vue.global.js" defer></script>
<script src="app.js" defer></script>
</head>
<body>
<header>
<h1>Vue Course Goals</h1>
</header>
<section id="user-goal">
<h2>My Course Goal</h2>
<p></p>
</section>
</body>
</html>
```

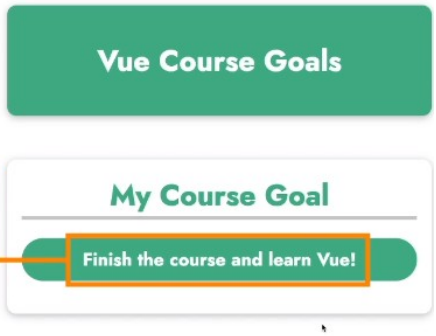
```js
app.js
app.js > ...
const app = Vue.createApp({
  // data() {..} -> shortcut
  data: function() {
    return {
      courseGoal: 'Finish the course and learn Vue.'
    };
  }
});

app.mount('#user-goal')
```

- Anything that's part of the object returned in *data* can be used in the Vue controlled HTML part.

- Interpolation and data binding

```html
index.html > html > body > section#user-goal
<h1>Vue Course Goals</h1>
</header>
<section id="user-goal">
<h2>My Course Goal</h2>
<p>{{ courseGoal }}</p>
</section>
</body>
```

```js
app.js
app.js > app > data > courseGoal
const app = Vue.createApp({
  data() {
    return {
      courseGoal: 'Finish the course and learn Vue!'
    };
  }
});
```



- Binding attributes with the *v-bind* directive

- You don't always want to use interpolation for outputting data.
- v-bind is used to bind or set the values of attributes.

→ Wrong way:

```
index.html > html > body > section#user-goal > p > a
18      </header>
19      <section id="user-goal">
20        <h2>My Course Goal</h2>
21        <p>{{ courseGoal }}</p>
22        <p>Learn more <a href="{{ vueLink }}">about Vue</
23      </section>
24    </body>
```

```
JS app.js
JS app.js > app > data > vueLink
1   const app = Vue.createApp({
2     data() {
3       return {
4         courseGoal: 'Finish the course and learn Vue!',
5         vueLink: 'https://vuejs.org/'
6       };
7     }
8   });
```

→ Right way:

```
index.html > html > body > section#user-goal > p > a
19      <section id="user-goal">
20        <h2>My Course Goal</h2>
21        <p>{{ courseGoal }}</p>
22        <p>Learn more <a v-bind:href="vueLink">about Vue</a></p>
23      </section>
24    </body>
25  </html>
```

```
JS app.js
JS app.js > ...
1   const app = Vue.createApp({
2     // data() {..} -> shortcut
3     data: function() {
4       return {
5         courseGoal: 'Finish the course and learn Vue.',
6         vueLink: 'https://vuejs.org/'
7       };
8     }
9   });
10
11  app.mount('#user-goal')
```

- Understanding *methods* in Vue apps

- ***methods*** allows you to define functions which should execute when something happens. When you call them for example, or when a user event like a button click occurs.

- *methods* takes an object, which will be full of methods or functions.

```
index.html > ...
19      <section id="user-goal">
20        <h2>My Course Goal</h2>
21        <p>{{ outputGoal() }}</p>
22        <p>Learn more <a v-bind:href="vueLink">about Vue</a></p>
23      </section>
24    </body>
25  </html>
```

```
app.js
app.js > ...
3     data: function() {
4       return {
5         courseGoal: 'Finish the course and learn Vue.',
6         vueLink: 'https://vuejs.org/'
7       };
8     },
9     methods: {
10      outputGoal() {
11        const randomNumber = Math.random();
12        if (randomNumber < 0.5) {
13          return 'Learn Vue!';
14        } else {
15          return 'Master Vue!';
16        }
17      }
18    }
19  });
```

- Working with data inside of a Vue app

- Vue takes all the data returned in the data object (from the data function), and it merges it into a global Vue instance object.

- Your *methods* are also available there. And they do have access to anything stored in that global object through the *this* keyword.

```
app.js > ...
3     data: function() {
4       return {
5         courseGoalA: 'Finish the course and learn Vue.',
6         courseGoalB: 'Maste Vue and build amazing apps.',
7         vueLink: 'https://vuejs.org/'
8       };
9     },
10    methods: {
11      outputGoal() {
12        const randomNumber = Math.random();
13        if (randomNumber < 0.5) {
14          return this.courseGoalA;
15        } else {
16          return this.courseGoalB;
17        }
18      }
19    }
20  });
```

- Outputting raw HTML content with v-html

- You, for example, fetched a data and it should be output from a database and from there you're getting some structured HTML code. If you use HTML elements in this string, and you output this with interpolation, so with the double curly braces {{ outPutGoal }} , you'll see that the HTML elements are just output as text, so they are not interpreted as HTML. That is a useful security feature because it protects you against cross site scripting attacks.

- Sometimes you want to output this as HTML and not as text that looks like HTML. For that you've got another directive that is the ***v-html*** directive.

```
index.html > ...
19      <section id="user-goal">
20        <h2>My Course Goal</h2>
21        <p v-html="outputGoal()"></p>
22        <p>Learn more <a v-bind:href="vueLink">about Vue</a></p>
23      </section>
24    </body>
25  </html>
```

```
app.js
app.js > ...
1   const app = Vue.createApp({
2     // data() {..} -> shortcut
3     data: function() {
4       return {
5         courseGoalA: 'Finish the course and learn Vue.',
6         courseGoalB: '<h2>Maste Vue and build amazing apps.</h2>',
7         vueLink: 'https://vuejs.org/'
8       };
9     },
```

- Understanding *event binding*

index.html > html > body > section#events > button

```html
<body>
  <header>
    <h1>Vue Events</h1>
  </header>
  <section id="events">
    <h2>Events in Action</h2>
    <button v-on:click="counter++">Add</button>
    <button v-on:click="counter--">Remove</button>
    <p>Result: {{ counter }}</p>
  </section>
</body>
</html>
```

app.js

app.js > ...

```js
const app = Vue.createApp({
  data() {
    return {
      counter: 0,
    };
  },
});

app.mount('#events');
```

- Events and methods

- Currently, I'm updating the counter in the HTML code and it's generally is considered a bad or not optimal practice. You should't put too much logic into your HTML code. Instead, the HTML code should really just be about outputting stuff.

- We don't just use *methods* to manually call them, we can also use methods to connect them to event listeners and let Vue call them for us when a certain event occurs.

```
index.html > html > body > section#events > button
19        <section id="events">
20          <h2>Events in Action</h2>
21          <button v-on:click="add()">Add</button>
22          <button v-on:click="counter--">Remove</button>
23          <p>Result: {{ counter }}</p>
24        </section>
25      </body>
26    </html>
27
```

```
JS app.js  M ×
JS app.js > ...
1   const app = Vue.createApp({
2     data() {
3       return {
4         counter: 0,
5       };
6     },
7     methods: {
8       add() {
9         this.counter++;
10      }
11    }
12  });
13
14  app.mount('#events');
```

- Besides calling add explicitly with parentheses, you can also just point at it so that you pass to Vue the name of the function so that it executes it when a click occurs, Vue accepts both. Typically, you just wanna point at it since that is closer to vanilla JavaScript and how you would set up an event listener but it is worth noting that both syntaxes are allowed.

index.html > html > body > section#events > button

```html
    <section id="events">
      <h2>Events in Action</h2>
      <button v-on:click="add">Add</button>
      <button v-on:click="reduce">Remove</button>
      <p>Result: {{ counter }}</p>
    </section>
  </body>
</html>
```

app.js

app.js > ...

```js
const app = Vue.createApp({
  data() {
    return {
      counter: 0,
    };
  },
  methods: {
    add() {
      this.counter++;
    },
    reduce() {
      this.counter--;
    }
  }
});

app.mount('#events');
```

- Working with event arguments

index.html > html > body > section#events > button

```html
      <section id="events">
        <h2>Events in Action</h2>
        <button v-on:click="add(10)">Add 10</button>
        <button v-on:click="reduce(10)">Remove 10</button>
        <p>Result: {{ counter }}</p>
      </section>
    </body>
  </html>
```

app.js M

app.js > ...

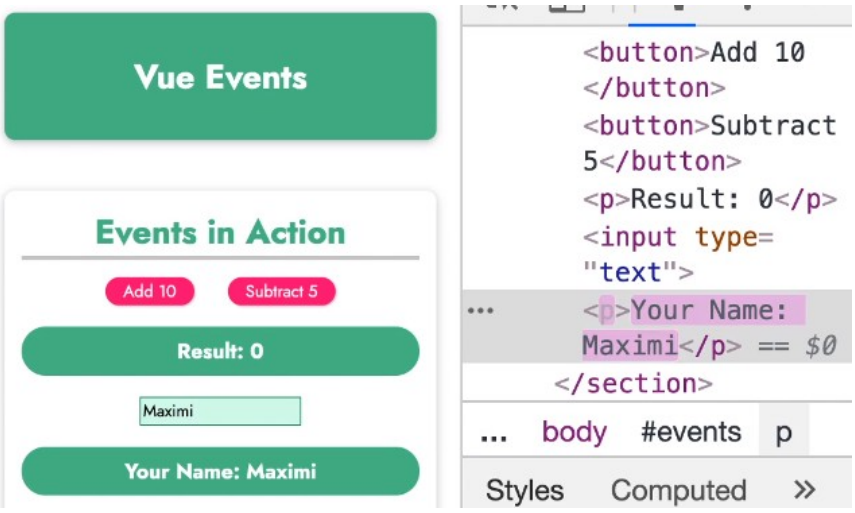
```js
const app = Vue.createApp({
  data() {
    return {
      counter: 0,
    };
  },
  methods: {
    add(num) {
      this.counter = this.counter + num;
    },
    reduce(num) {
      this.counter = this.counter - num;
    }
  }
});
```

- Using the native event object

```
index.html > html > body > section#events > input
19      <section id="events">
20        <h2>Events in Action</h2>
21        <button v-on:click="add(10)">Add 10</button>
22        <button v-on:click="reduce(10)">Remove 10</button>
23        <p>Result: {{ counter }}</p>
24        <input type="text" v-on:input="setName">
25        <p>Your name: {{ name }}</p>
26      </section>
27    </body>
```

```
JS app.js  M ×
JS app.js > ...
8     methods: {
9       add(num) {
10        this.counter = this.counter + num;
11      },
12      reduce(num) {
13        this.counter = this.counter - num;
14      },
15      setName(event) {
16        this.name = event.target.value;
17      }
18    }
19  });
```

- The parts that are rerendered the browser flash, in this case the paragraph with the name flashes and nothing else on the screen flashed. If Vue would update the entire screen, whenever something changed somewhere, this would be bad for performance. Instead Vue is really smart and it sees that with every keystroke, only the name changes.



- Use $event to pass a parameter and not lose the event property:

```
index.html > html > body > section#events > input
19      <section id="events">
20        <h2>Events in Action</h2>
21        <button v-on:click="add(10)">Add 10</button>
22        <button v-on:click="reduce(10)">Remove 10</button>
23        <p>Result: {{ counter }}</p>
24        <input type="text" v-on:input="setName($event, 'Mr')">
25        <p>Your name: {{ name }}</p>
26      </section>
27    </body>
```

```
app.js
app.js > ...
8     methods: {
9       add(num) {
10        this.counter = this.counter + num;
11      },
12      reduce(num) {
13        this.counter = this.counter - num;
14      },
15      setName(event, someText) {
16        this.name = someText + ' ' + event.target.value;
17      }
18    }
```

- Exploring event modifiers

- If I add a button and click it, the page will reload. The reason for that is that the browser default is to submit that form and send an HTTP request to the server serving this app.

```
<form>
  <input type="text">
  <button>Sign Up</button>
</form>
```

- Typically with frameworks like Vue, you wanna prevent this browser default and instead you wanna handle this manually in JavaScript with help of Vue.

- One way to prevent this default behavior is by calling *preventDefault* method, that's a javascript method.

```
    <form v-on:submit="submitForm">
      <input type="text">
      <button>Sign Up</button>
    </form>
  </section>
```

app > methods > submitForm

```
methods: {
  submitForm(event) {
    event.preventDefault();
    alert('Submitted!');
  },
```

- This works and there's nothing wrong with it, but Vue has a nicer way.

index.html > ...

```
26        <form v-on:submit.prevent="submitForm">
27          <input type="text">
28          <button>Sign up</button>
29        </form>
30      </section>
31    </body>
32  </html>
```

app.js M

app.js > ...

```
8     methods: {
9       add(num) {
10        this.counter = this.counter + num;
11      },
12      reduce(num) {
13        this.counter = this.counter - num;
14      },
15      setName(event, someText) {
16        this.name = someText + ' ' + event.target.value;
17      },
18      submitForm() {
19        alert('Submitted');
20      }
21    }
```

- Another modifiers are for instance *v-on:click.right* which will make that it only reacts for right clicks.

- The following modifier is for reacting to the *enter* keystroke.

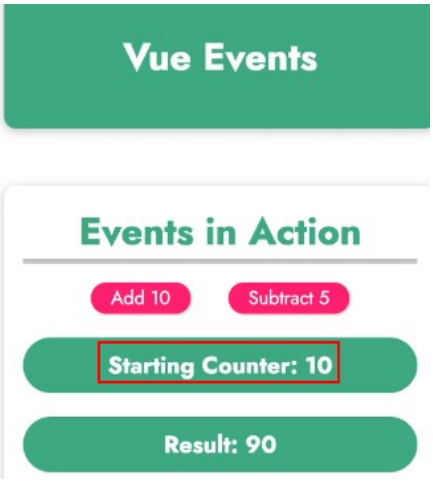
```
index.html > ...
24        <input
25          type="text"
26          v-on:input="setName($event, 'Mr')"
27          v-on:keyup.enter="confirmInput"
28        />
29        <p>Your name: {{ confirmedName }}</p>
```

```
app.js  M ×
app.js > ...
16      setName(event, someText) {
17        this.name = someText + " " + event.target.value;
18      },
19      confirmInput() {
20        this.confirmedName = this.name;
21      },
22      submitForm() {
23        alert("Submitted");
24      },
25    },
26  });
```

- Locking content with v-once

- If you have such a scenario that you have some data that changes and you want to preserve the initial state and not reflect any other changes, there is a special directive you can put on to the element where you are using that dynamic value in. And that's the v-once directive. This tells Vue that any dynamic data bindings should only be evaluated once.

```
<section id="events">
  <h2>Events in Action</h2>
  <button v-on:click="add(10)">Add 10</button>
  <button v-on:click.right="reduce(5)">Subtract 5</but
  <p v-once>Starting Counter: {{ counter }}</p>
  <p>Result: {{ counter }}</p>
  <input
    type="text"
    v-on:input="setName($event, 'Schwarzmüller')"
app > data > counter
```



- Data binding + event binding = Two-way binding

- Use v-bind:value on the input to bind the input's value to a variable so that when the variable's value is cleared, the input will be cleared as well.

```html
index.html > ...
<input
  type="text"
  v-on:input="setName($event, 'Mr')"
  v-on:keyup.enter="confirmInput"
  v-bind:value="name"
/>
<button v-on:click="resetInput">Reset input</button>
<p>Your name: {{ confirmedName }}</p>
<form v-on:submit.prevent="submitForm">…
</form>
</section>
</body>
</html>
```

```js
app.js M ×
app.js > ...
setName(event, someText) {
  this.name = event.target.value;
},
confirmInput() {
  this.confirmedName = this.name;
},
submitForm() {
  alert("Submitted");
},
resetInput() {
  this.name = '';
  this.confirmedName = '';
}
},
```

- Vue has a shortcut for this. It has a special built in directive which simplifies this so that if you bind the value and listen to input changes on input, you can get rid of all of that.

- ***v-model*** is a shortcut for ***v-bind:value v-on:input***. This is a concept called ***two-way binding*** because we're communicating in both ways. We are listening to an event coming out of the input element and at the same time, we're writing the value back to the input element through its value attribute.

index.html > html > body > section#events > input

```html
22        <button v-on:click="reduce(10)">Remove 10</button>
23        <p>Result: {{ counter }}</p>
24        <input
25          type="text"
26          v-on:keyup.enter="confirmInput"
27          v-model="name"
28        />
29        <button v-on:click="resetInput">Reset input</button>
30        <p>Your name: {{ confirmedName }}</p>
31 >      <form v-on:submit.prevent="submitForm">…
34        </form>
35      </section>
36    </body>
```

app.js M ×

app.js > ...

```js
16      // setName(event, someText) {
17      //   this.name = event.target.value;
18      // },
19      confirmInput() {
20        this.confirmedName = this.name;
21      },
22      submitForm() {
23        alert("Submitted");
24      },
25      resetInput() {
26        this.name = '';
27        this.confirmedName = '';
28      }
```

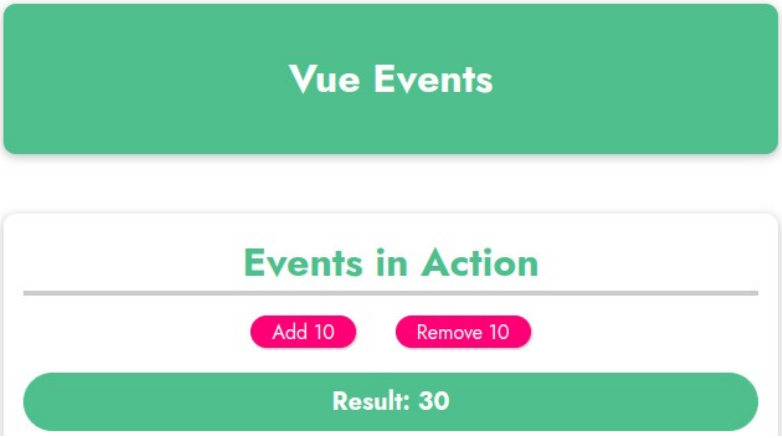
- Methods used for data binding: how it works

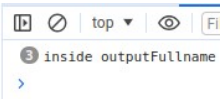
- If we have something like this:

```
index.html > ...
21        <button v-on:click="add(10)">Add 10</button>
22        <button v-on:click="reduce(10)">Remove 10</button>
23        <p>Result: {{ counter }}</p>
24        <input
25          type="text"
26          v-on:keyup.enter="confirmInput"
27          v-model="name"
28        />
29        <button v-on:click="resetInput">Reset input</button>
30        <p>Your name: {{ outputFullname() }}</p>
```

```
app.js  M ×
app.js > ...
29      outputFullname() {
30        if (this.name === '') {
31          return '';
32        }
33        return this.name + ' ' + 'perez'
34      }
35    },
36  });
```

- If the counter changes, Vue sees that in that paragraph we're using the *counter* and hence it needs to update that part. That's exactly why we use Vue, it updates the page for us automatically. The problem is if we call a method (like *outputFullname*). This method will also be re-executed by Vue whenever anything on the page changes because Vue can't know what this method does, maybe the *counter* gets used in there.





- That's why *methods* are not the best solution for outputting some dynamically calculated value.

- Introducing *computed properties*

```
    <p>Your Name: {{ name + ' ' + 'Schwarzmüller' }}</p>
  </section>
</body>
```

app > methods > outputFullname

```
methods: {
  outputFullname() {
    console.log('Running again...');
    if (this.name === '') {
      return '';
    }
    return this.name + ' ' + 'Schwarzmüller';
  },
```

- Although that works, we have more logic in the HTML code and that's not good.
- There is a third feature we can use, ***computed properties***.
- With ***Computed properties*** Vue will be aware of their dependencies and only reexecute them if one of the dependencies changed.
- *Computed* is the third big configuration option. The first one was data and the second one was methods.
- You should name your *computed properties* just as you would name your data properties because we're going to use this like a data property, not like a method, even though technically it is a method.
- For performance it's better to use *computed properties* than methods for outputting values.

```html
index.html > html > body > section#events > p
23        <p>Result: {{ counter }}</p>
24        <input
25          type="text"
26          v-on:keyup.enter="confirmInput"
27          v-model="name"
28        />
29        <button v-on:click="resetInput">Reset input</button>
30        <p>Your name: {{ fullname }}</p>
31 >      <form v-on:submit.prevent="submitForm">…
34        </form>
```

```js
app.js  M ×
app.js > ...
30    computed: {
31      fullname() {
32        console.log("inside fullname");
33        if (this.name === "") {
34          return "";
35        }
36        return this.name + " " + "perez";
37      },
38    },
39  });
```

- Only use *methods* if you know that you want to recalculate a value whenever anything on the page changed.

- Working with *Watchers*

- A ***watcher*** is a function you can tell Vue to execute when one of its dependencies changed.

```
index.html > html > body > section#events > input
23        <p>Result: {{ counter }}</p>
24        <input
25          type="text"
26          v-on:keyup.enter="confirmInput"
27          v-model="name"
28        />
29        <button v-on:click="resetInput">Reset input</button>
30        <p>Your name: {{ fullname }}</p>
```

```
app.js  M ×
app.js > ...
3       return {
4         counter: 0,
5         name: "",
6         confirmedName: "",
7         fullname: "",
8       };
9     },
10 >  methods: {…
30    },
31 >  computed: {…
39    },
40    watch: {
41      name(value) {
42        if (value === "") {
43          this.fullname = "";
44        } else {
45          this.fullname = "Mr" + " " + value;
46        }
47      },
48    },
49  });
```

- This tells Vue that whenever *name* changes, that *watcher* method will re execute.

- We also could accept a second argument, besides the *value* and that would then be the previous value.

- You could use *watch* as an alternative to a *computed property*. We have a couple of problems if we use it for that though.

- I would argue that the concept of a *computed property* is maybe a bit easier to understand, but the bigger problem arises If we have a *computed property* that would use more than one dependency.

→ ***Computed properties* vs *watchers***

- We want to update the *fullname* whenever the *name* or the *lastName* changed. To achieve this with a *watcher*, we need to add a second *watcher* because the first one simply watches the name. Now we have two *watchers* to reflect the *fullname*, which is managed with two inputs. This works, but it's a lot of code, especially if we compare it to the *computed property* alternative.

```
<input type="text" v-model="name">
<input type="text" v-model="lastName">
<button v-on:click="resetInput">Reset Input</button>
<p>Your Name: {{ fullname }}</p>
```

```
watch: {
  name(value) {
    if (value === '') {
      this.fullname = '';
    } else {
      this.fullname = value + ' ' + this.lastName;
    }
  },
  lastName(value) {
    if (value === '') {
      this.fullname = '';
    } else {
      this.fullname = this.name + ' ' + value;
    }
  }
```

→ Alternative with *computed property*:

```
index.html > html > body > section#events > p
24        <input type="text" v-model="name" />
25        <input type="text" v-model="lastName" />
26        <button v-on:click="resetInput">Reset input</button>
27        <p>Your name: {{ fullname }}</p>
```

```
app.js  M ×
app.js > ...
2     data() {
3       return {
4         counter: 0,
5         name: "",
6         lastName: "",
7         confirmedName: "",
8         // fullname: "",
9       };
10    },
11 >  methods: {…
31    },
32    computed: {
33      fullname() {
34        console.log("inside fullname");
35        if (this.name === "" || this.lastName === "") {
36          return "";
37        }
38        return this.name + " " + this.lastName;
39      },
40    },
```

- Why do we have *watchers* then if we can use them, but they're worse? Because this works but it's not the main scenario for using *watchers*. Watchers are powerful, if you have a different kind of intent in mind. Let's say when the counter exceeds 50 we wanna reset it.

- That's the kind of thing where watchers can be helpful. If you wanna run logic, that maybe also updates a data property, but which shouldn't always do that.

```
index.html > html > body > section#events > input
18      </header>
19      <section id="events">
20        <h2>Events in Action</h2>
21        <button v-on:click="add(10)">Add 10</button>
22        <button v-on:click="reduce(10)">Remove 10</button>
23        <p>Result: {{ counter }}</p>
```

```
app.js  M ×
app.js > ...
38    watch: {
39      counter(value) {
40        if (value > 50) {
41          this.counter = 0;
42        }
43      }
44    },
45  });
```

- Another example, would be HTTP requests, which you wanna send if certain data changes, or timers which you wanna set, if certain values change.

```
watch: {
  counter(value) {
    if (value > 50) {
      const that = this;
      setTimeout(function () {
        that.counter = 0;
      }, 2000);
    }
  }
```

- Methods vs computed properties vs watchers

Methods	Computed	Watch
Use with event binding OR data binding	Use with data binding	Not used directly in template
Data binding: Method is executed for every "re-render" cycle of the component	Computed properties are only re-evaluated if one of their "used values" changed	Allows you to run any code in reaction to some changed data (e.g. send Http request etc.)
Use for events or data that really needs to be re-evaluated all the time	Use for data that depends on other data	Use for any non-data update you want to make

- *v-bind* and *v-on* shorthands

- You can replace ***v-on:****click* with the *@click*.
- ***v-bind:****value* can be replaced for ***:****value*.

- Dynamic styling with inline styles

- With dynamic styling refers to change the style dynamically in reaction to something, for example, in reaction to a click or to the user entering something.

index.html > ...

```html
<section id="styling">
  <div
    class="demo"
    :style="{borderColor: boxASelected ? 'red' : '#ccc'}"
    @click="boxSelected('A')"
  ></div>
  <div class="demo" @click="boxSelected('B')"></div>
  <div class="demo" @click="boxSelected('C')"></div>
</section>
```

app.js

app.js > ...

```js
data() {
  return {
    boxASelected: false,
    boxBSelected: false,
    boxCSelected: false,
  };
},
methods: {
  boxSelected(box) {
    if (box === 'A') {
      this.boxASelected = true;
    } else if (box === 'B') {
      this.boxBSelected = true;
    } else if (box === 'C') {
      this.boxCSelected = true;
    }
  },
},
```

- Adding CSS classes dynamically

- Inline styles have a couple of disadvantages. They overrule all other styles, which sometimes is what you'll need, but often leads to problems. Therefore in modern web development and CSS, we typically don't use inline styles too often.

index.html > html > body > section#styling > div.demo

```html
<div
  class="demo"
  :class="{active: boxASelected}"
  @click="boxSelected('A')"
></div>
<div
  class="demo"
  :class="{active: boxBSelected}"
  @click="boxSelected('B')"
></div>
<div
  class="demo"
  :class="{active: boxCSelected}"
  @click="boxSelected('C')"
></div>
```

styles.css > ...

```css
.active {
  border-color: red;
  background-color: salmon;
}
```

app.js

app.js > ...

```js
const app = Vue.createApp({
  data() {
    return {
      boxASelected: false,
      boxBSelected: false,
      boxCSelected: false,
    };
  },
  methods: {
    boxSelected(box) {
      if (box === 'A') {
        this.boxASelected = !this.boxASelected;
      } else if (box === 'B') {
        this.boxBSelected = !this.boxBSelected;
      } else if (box === 'C') {
        this.boxCSelected = !this.boxCSelected;
      }
    },
  },
});
```

- This way:
  :class="{active: boxASelected}"
  is more readable than:
  :class="boxASelected ? 'demo active' : 'demo'"

- Classes and computed properties

- We can move the HTML classes related code to the javascript class.

```
index.html
index.html > html > body > section#styling >
<section id="styling">
  <div
    class="demo"
    :class="boxAClasses"
    @click="boxSelected('A')"
```

```
app.js
app.js > app > computed > boxAClasses >
    boxCSelected: false,
  };
},
computed: {
  boxAClasses() {
    return { active: this.boxASelected };
  },
},
```

- This can especially be helpful if you have more complex dynamic class code, if you're not just referring to a true or false boolean, but if you had a more complex check or different if statements where you want to return different objects. In such cases as always computed properties can shine a lot, but you can also use it to simplify your HTML code a bit more and move that logic into JavaScript.

- Dynamic classes: array syntax

```
<div
  :class="['demo', {active: boxBSelected}]"
  @click="boxSelected('B')"
></div>
```