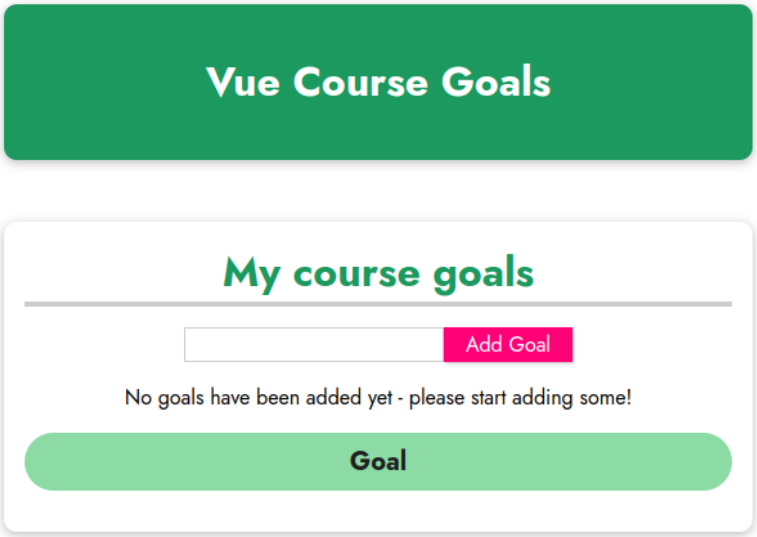
**SECTION 03 - RENDERING CONTIDIONAL CONTENT AND LISTS**

# Index

## - Rendering content conditionally




→ Initial code:

index.html > html > body > section#user-goals > p

```
19      <section id="user-goals">
20        <h2>My course goals</h2>
21        <input type="text" />
22        <button>Add Goal</button>
23        <p>No goals have been added yet - please start adding some!</p>
24        <ul>
25          <li>Goal</li>
26        </ul>
27      </section>
28    </body>
29  </html>
```

app.js

```
1  const app = Vue.createApp({
2    data() {
3      return { goals: [] };
4    },
5  });
6
7  app.mount('#user-goals');
```

```
index.html > ...
19      <section id="user-goals">
20        <h2>My course goals</h2>
21        <input type="text" v-model="goal" />
22        <button @click="addGoal()">Add Goal</button>
23        <p v-if="goals.length === 0">
24          No goals have been added yet - please start adding some!
25        </p>
26        <ul>
27          <li>Goal</li>
28        </ul>
29      </section>
30    </body>
31  </html>
```

```
app.js > ...
1   const app = Vue.createApp({
2     data() {
3       return { goals: [], goal: "" };
4     },
5     methods: {
6       addGoal() {
7         this.goals.push(this.goal);
8         this.goal = "";
9         console.log(this.goals);
10      }
11    }
12  });
```

**- v-if, v-else and v-else-if**

- The only important thing is that v-else has to be used on an element that comes directly after an element with v-if on it. You can't have another element in between. v-else needs to be used on a direct neighbor element of the element which has v-if.

**- Using v-show instead of v-if**

- There is an alternative to v-if that is v-show. v-show does not work together with v-else or v-else-if. It only works stand-alone and therefore if you have multiple alternatives, you really need to use multiple v-shows.

- In the image below we see that the paragraph is there, but so is the unordered list. And the only difference is that the unordered list is hidden because its display style is set to none and that's the difference. ***v-if*** really removes and adds elements from and to the DOM. ***v-show*** on the other hand, just hides and show items with CSS.

```
▶<ul style=
"display: none;">…
</ul> == $0
```

- Showing and hiding with CSS means that you don't have to add and remove elements all the time, which can cost performance. On the other hand, it also means that you have a bunch of elements in your DOM, which you don't really need at the moment, which also is not ideal.
- If you have an element which visibility status changes a lot, like a button that toggles an element and it's switching between visibility and being hidden all the time, then you might want to consider *v-show*. In other cases use *v-if*.

## - Rendering lists of data

```
index.html > ...
23          <p v-if="goals.length === 0">
24            No goals have been added yet - please start adding some!
25          </p>
26          <ul v-else>
27            <li v-for="goal in goals">{{ goal }}</li>
28          </ul>
29        </section>
30      </body>
31    </html>
```

- It would be bad if view would re-render the entire list with every new item that we add. This would not be good for performance and thankfully view doesn't do it.

## - Diving deeper into v-for

- To access the index:

```
<li v-for="(goal, index) in goals">{{ goal }}</li>
```

## - Removing list items

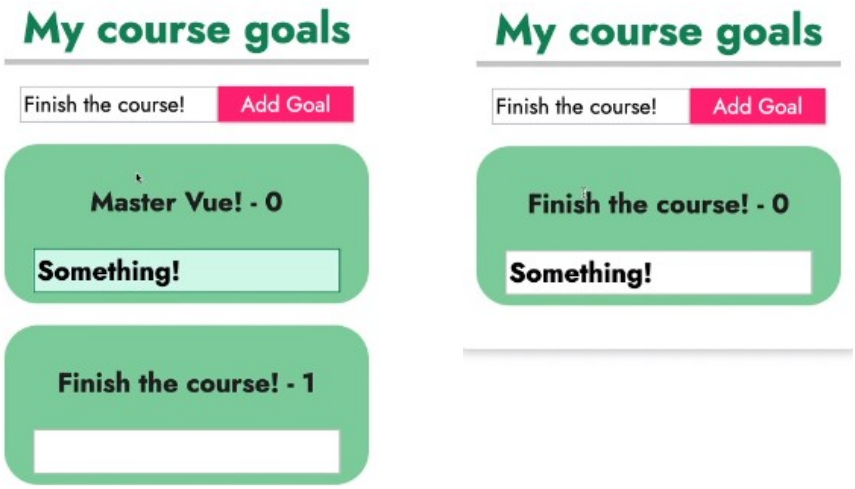
```
index.html M
index.html > ...
26          <ul v-else>
27            <li v-for="(goal, idx) in goals" @click="removeGoal(index)">{{ goal }}</li>
28          </ul>
29        </section>

app.js M
app.js > ...
1   const app = Vue.createApp({
2     data() {…
4     },
5     methods: {
6       addGoal() {…
10      },
11      removeGoal(idx) {
12        this.goals.splice(idx, 1)
13      }
14    }
15  });
```

## - Lists and keys

- I'm going to add an input besides the text and I'm gonna use @click.stop to avoid that when I click on the input, the element be removed.

```
index.html > ...
26        <ul v-else>
27          <li v-for="(goal, idx) in goals" @click="removeGoal(index)">
28            <p>{{ goal }} - {{ idx }}</p>
29            <input type="text" @click.stop>
30          </li>
31        </ul>
```

- A bug arises when we click outside the input, on the element:\




- Vue updates the list when you add and remove items. It renders the list in the real DOM and updates it as required. And it tries to do that such that it optimizes performance. That also means that it reuses DOM elements. So if we have two goals and I delete the first one, Vue will actually not re-render the entire list. It will not delete the first DOM element and move the second one around. Instead it basically takes the content of the second element and moves it into the first DOM element.

- There is no unique identification criteria for every rendered DOM element. Although the content is different, thankfully Vue is not going to compare all the content with each other, that would be super performance intensive.

- The ***key*** attribute is a non default HTML attribute, which you can add on elements on which you also use v-for.

- The index doesn't really belong to the goal content. The index is always the same. The first item always has index zero, for example. So the index isn't strictly attached to the element value in the array.

```
27          <li v-for="(goal, idx) in goals" :key="goal" @click="removeGoal(index)">
28            <p>{{ goal }} - {{ idx }}</p>
29            <input type="text" @click.stop>
30          </li>
31        </ul>
```