CHAPTER 2

Day 2: Mastering Directives in Vue.js

In this chapter, we will delve into directives in Vue.js, a key element for mastering this front-end library. Directives allow us to add new functionalities to HTML elements on the page.

We will explore several essential directives, such as `v-bind`, `v-if`, `v-else`, `v-show`, and `v-for`, providing detailed steps to understand how they function.

Next, we will focus on the `v-model` directive, which is particularly useful for two-way data binding in forms.

Finally, we will address the use of modifiers in Vue.js, which allow us to modify the behavior of directives.

Using Attributes in Vue.js Components

A component is similar to an HTML element and can have attributes (also called props, meaning properties).

Let's use two new attributes, named `init` and `end`, in the `MyCounter` component:

- The `init` attribute indicates the initialization value of the counter. If this attribute is not specified, its starting value is considered as 0.

E. Sarrion, *Master Vue.js in 6 Days*, https://doi.org/10.1007/979-8-8688-0364-2_2

- The end attribute indicates the final value of the counter. If this attribute is not specified, the counter does not stop.

As long as the counter value (incremented every second) is between the `init` and `end` values, the counter continues to increment. Once the `end` value is reached (if specified in the attributes), the counter stops.

Step 1: Using the init and end Attributes in the MyCounter Component

An example of using the `MyCounter` component with the `init` and `end` attributes could be the following:

Counter from 10 to 20

```
<MyCounter init="10" end="20" />
```

If the attribute values are written, as shown previously, within quotes (in the form of strings, "10" and "20"), they are string literals passed to the `MyCounter` component. To use the attribute value as a numeric value in the `MyCounter` component, it will employ the JavaScript function `parseInt(value)`, which returns the numeric representation of the value if it wasn't already numeric. We will demonstrate how to write the content of the `MyCounter` component in the following section.

Alternatively, one can use the `MyCounter` component in the following form, without representing the attributes as strings:

Counter from 10 to 20

```
<MyCounter init=10 end=20 />
```

In this case, the numeric values 10 and 20 are passed to the `MyCounter` component. To indicate a counter that starts at 10 but never stops, the `end` attribute is omitted in the `MyCounter` component's definition:

Counter from 10 to infinity

```
< MyCounter init=10 />
```

Finally, to indicate a counter that counts from 0 to infinity, no attributes are specified in the `MyCounter` component:

Counter from 0 to infinity

```
< MyCounter />
```

It is also possible to set the value of an attribute based on the value of a variable initialized in the program. For example, if we define the variable `init` initialized to the value 10, we can write in the `src/App.vue` file:

Counter initialized from the init variable (file src/App.vue)

```
<script setup>

import MyCounter from './components/MyCounter.vue'

const init = 10;  // The variable init is equal to 10

</script>

<template>

<MyCounter :init="init" />

</template>

<style scoped>
</style>
```

The `init` variable is defined in the `<script>` section of the component. It is accessible in the `<template>` section of the component by writing `:init="init"`. The syntax `:init="init"` signifies that the `init` attribute (indicated as `:init`) is initialized with the value of the `init` variable (indicated as `"init"`).

The ":" symbol before an attribute name indicates to interpret the following value as a JavaScript expression. One could also write `<MyCounter :init="init+3" />` to start the counter with the value 13 instead of 10, as `"init+3"` is a valid JavaScript expression.

The quotes around the value of the JavaScript expression are necessary if the JavaScript expression contains spaces. Thus, writing `="init"` or `=init` are equivalent expressions.

To initialize the counter with the numeric value 10, one can also write the following:

Initialize the init attribute to the numeric value 10

```
<MyCounter :init="10" />
```

Indeed, specifying `:init` instead of just `init` for the attribute name indicates that the following value is a JavaScript expression, specifically the numeric value 10 and not the string "10".

We have seen how to write and use the `MyCounter` component in various forms with the `init` and `end` attributes. Let's now explore how the `MyCounter` component is written to make use of these attributes.

Step 2: Writing the MyCounter Component That Utilizes the init and end Attributes

The code associated with the `MyCounter` component must consider the various possible forms for writing the attributes.

The attributes will be defined in the `<script setup>` section of the component, using Vue.js's `defineProps()` method.

MyCounter component with init and end attributes (file src/components/MyCounter.vue)

```
<script setup>
import { ref, computed, onMounted, onUnmounted, defineProps }
from 'vue';

let timer;

const props = defineProps(["init", "end"]);
// Declaration of the "init" and "end" attributes

const init = props.init || 0;    // 0: default value
const end = props.end || 0;  // 0: default value

const count = ref(parseInt(init));
const doubleCount = computed(() => count.value * 2);

const increment = () => {
  if (!end || count.value < parseInt(end)) count.value++;
  else stop();
};

const start = () => {
  timer = setInterval(() => {
    increment();
  }, 1000);
};

const stop = () => {
  clearInterval(timer);
};

onMounted(() => {
  start();
});
```

```
onUnmounted(() => {
  stop();
});

</script>


<template>
  <h3>MyCounter Component</h3>
  init : {{init}} => end : {{end}}
  <br /><br />
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
</template>
```

The `defineProps()` method is used by specifying, in an array, the name of each attribute.

When retrieving the value of the attribute by writing `const init = props.init`, it is mandatory to provide a default value for the attribute (here, 0, by writing `props.init || 0`). Otherwise, the retrieval into a variable cannot be performed. In such a case, one would be forced to use the `init` attribute in the form `props.init` throughout the program, which would not be convenient.

The values of the attributes, retrieved into the variables `init` and `end`, are then displayed in the template using {{init}} and {{end}}.

For example, suppose the `MyCounter` component is used as follows:

Counter from 10 to 20 (file src/App.vue)

```
<script setup>

import MyCounter from "./components/MyCounter.vue"

</script>
```

```
<template>
<MyCounter init=10 end=20 />
</template>
```

The counter starts at the value 10 and ends at the value 20. Let's run the program:

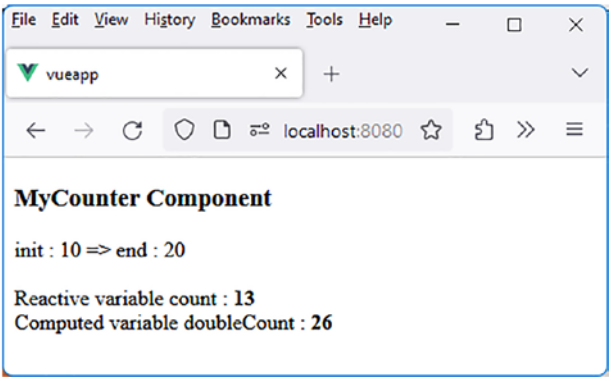

Figure 2-1. *Using the init and end attributes*

The counter starts from the value 10 and will stop when it reaches the value 20.

Passing an Object As Attributes

Instead of passing the `init` and `end` attributes individually to the `MyCounter` component, these values can also be transmitted within a JavaScript object.

Let's call the new attribute `limits`, which will replace the `init` and `end` attributes. The `limits` attribute will take the form `{init: 10, end: 20}`. Let's demonstrate how to use this attribute within the `MyCounter` component.

The App component, which uses the MyCounter component, is modified as follows:

App component using the MyCounter component and its limits attribute (file src/App.vue)

```
<script setup>

import MyCounter from "./components/MyCounter.vue"

</script>

<template>

<MyCounter :limits="{init:10, end:20}" />

</template>
```

The value of the limits attribute is specified as an object {init: 10, end: 20}. Adding a string around the object is optional if the following value does not contain spaces (here, the string is required because there is a space before the end attribute).

To ensure that Vue.js interprets the specified value as a JavaScript value (here, an object), it must be indicated by writing the attribute as :limits rather than just limits.

Let's see how the code of the MyCounter component is modified to accommodate the new limits attribute.

MyCounter component using the limits attribute (file src/components/MyCounter.vue)

```
<script setup>
import { ref, computed, onMounted, onUnmounted, defineProps }
from 'vue';

let timer;

const props = defineProps(["limits"]);
```

```
const init = props.limits.init || 0;
const end = props.limits.end || 0;

const count = ref(parseInt(init));
const doubleCount = computed(() => count.value * 2);

const increment = () => {
  if (!end || count.value < parseInt(end)) count.value++;
  else stop();
};

const start = () => {
  timer = setInterval(() => {
    increment();
  }, 1000);
};

const stop = () => {
  clearInterval(timer);
};

onMounted(() => {
  start();
});

onUnmounted(() => {
  stop();
});

</script>


<template>
  <h3>MyCounter Component</h3>
  init : {{init}} => end : {{end}}
  <br /><br />
```

```
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
</template>
```

The `limits` attribute is defined in the `definePropS()` method as `["limits"]`. The `init` value is retrieved from `props.limits.init`, and the `end` value is retrieved from `props.limits.end`. When the `end` value is reached, the counter stops:

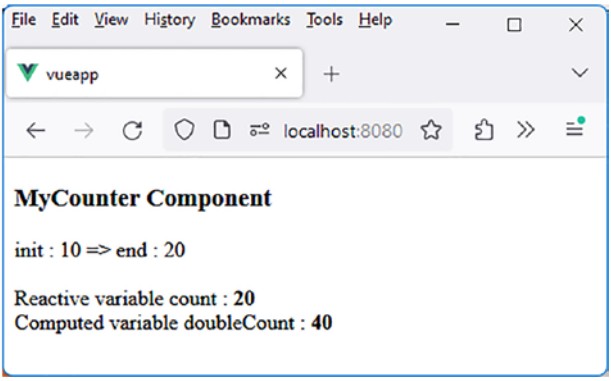

Figure 2-2. *Using the limits attribute*

We have seen how to define attributes for Vue.js components. Vue.js also allows the use of another form of attributes called directives. These are powerful tools. Let's now explore how to use the standard directives provided by Vue.js. In Chapter 5, we will learn how to create our own directives, extending Vue.js's capabilities.

Differences Between Directives and Attributes in Vue.js

A Vue.js directive is used similarly to an attribute. What then is the difference between the two?

1. An attribute represents a static value, information transmitted to the component or HTML element on which it is positioned. In the previous examples, we used the `init` and `end` attributes to statically indicate the starting and ending values of the counter. The same applies to HTML attributes associated with HTML elements. For example, the `class` attribute allows specifying a CSS class for the HTML element on which it is positioned.

2. On the other hand, Vue.js directives are used to add dynamic logic to an HTML element or Vue.js component, reacting to changes in data or performing specific actions in response to events. Directives allow binding HTML elements to the state of the Vue.js application, making the user interface responsive and interactive based on application data and logic.

A simple example of a Vue.js directive, which will be explained in the following, is the `v-show` directive. It is used in the form `v-show="condition"`. This directive shows or hides the element on which it is positioned based on the value specified in the condition. If the condition evaluates to `true`, the element is displayed; otherwise, it is hidden. This demonstrates the dynamic aspect of the directive, as opposed to static attributes.

To differentiate Vue.js directives from regular attributes (HTML attributes or those created in our components, such as the `init` and `end` attributes mentioned earlier), Vue.js directives all start with the prefix `"v-"`. Examples include `v-if`, `v-show`, `v-bind`, `v-on`, etc., which we will explain in the following.

Let's start with the `v-bind` directive.

v-bind Directive

The `v-bind` directive allows using attribute values that will be reactive, similar to reactive variables used in HTML.

For example, let's use the previous counter, where the value increments upon clicking the "count+1" button. Suppose we want to display the counter value in an input field. For this, we would like to write something like `<input type="text" value="{{count}}" />`. Indeed, it is hoped that, thanks to the reactivity of the `count` variable, the value of the input field will be updated when the counter is incremented.

Let's write this in the template of the `MyCounter` component. The `MyCounter` component becomes as follows:

Display the count variable in the value attribute of an input field (file src/components/MyCounter.vue)

```
<script setup>

import { ref, computed } from "vue"

const count = ref(0);
const doubleCount = computed(() => count.value * 2);

const increment = () => {
  count.value++;
};

</script>
```

```
<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br/>
  Input : <input type="text" value="{{count}}" />
  <br/><br/>

  <button @click="increment()">count+1</button>

</template>
```

The App component that displays the MyCounter component is as follows:

App component (file src/App.vue)

```
<script setup>

import MyCounter from "./components/MyCounter.vue"

</script>

<template>

<MyCounter />

</template>
```

After several clicks on the "count+1" button, the displayed result is as follows:

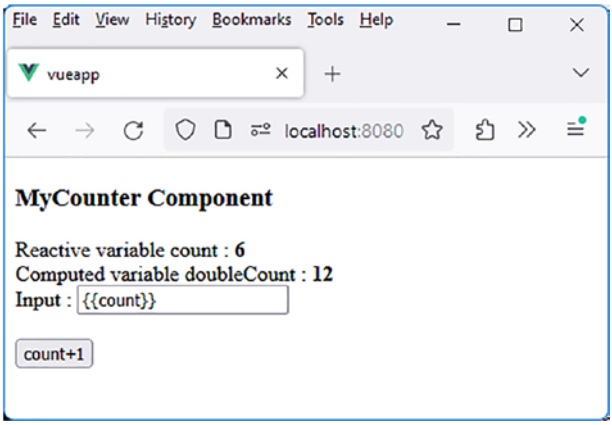

Figure 2-3. *Reactive variable in the value attribute*

The counter increments, but the value displayed in the input field does not reflect this change.

The use of `{{count}}` in the `value` attribute does not update the content of the input field, which remains fixed with the string `"{{count}}"`. To initialize and update the `value` attribute of the input field with the value of the reactive variable `count`, a directive called `v-bind` must be used. The `v-bind` directive allows binding the value of an attribute to that of a reactive variable.

Therefore, one would write `<input type="text" v-bind:value="count" />`, which binds the `value` attribute of the input field to the value of a reactive variable, in this case, the `count` variable.

The template of the `MyCounter` component becomes the following:

Display the count variable in the value attribute of an input field (file src/components/MyCounter.vue)

```
<script setup>

import { ref, computed } from "vue"

const count = ref(0);
const doubleCount = computed(() => count.value * 2);
```

```
const increment = () => {
  count.value++;
};

</script>

<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br/>
  Input : <input type="text" v-bind:value="count" />
  <br/><br/>

  <button @click="increment()">count+1</button>

</template>
```

Now, we obtain correct initialization and updates of the input field based on the changes in the reactive variable `count`.

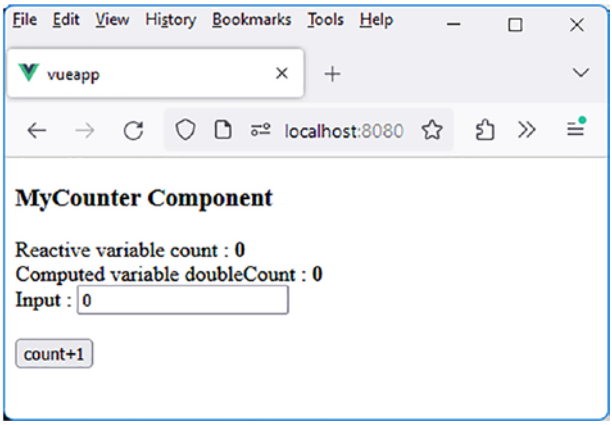

Figure 2-4. *Reactive variable in the value attribute with the v-bind directive*

The input field is now initialized with the value of the reactive variable `count`, which is 0.

As the `count` variable is reactive, incrementing it causes the update of its display wherever it is used, including in the input field.

Let's click the "count+1" button several times:

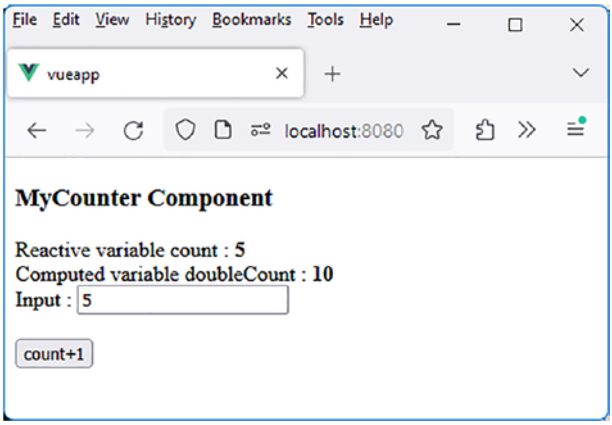

Figure 2-5. *Modification of the reactive variable count*

The value displayed in the input field is updated to reflect the value of the reactive variable `count`.

The `v-bind` directive is commonly used in templates. For this reason, Vue.js allows simplifying the syntax by writing `:value="count"` instead of `v-bind:value="count"`.

We had already used this simplified form of the `v-bind` directive by writing `:init="10"` or `:init="init"` in the previous pages.

One could also write `v-bind:value="count+3"` because the value `"count+3"` is a JavaScript expression interpreted by `v-bind`.

Additionally, one can write the shorthand form `:value="count+3"`, which is equivalent to `v-bind:value="count+3"`.

Refreshing a Component by Modifying Its Attributes

The following example demonstrates how to update a component by transmitting new values to its attributes.

In this scenario, we want the "count+1" button to be integrated into the App component rather than the MyCounter component. This means that the App component should handle the incrementation of the counter and transmit this counter value to the MyCounter component through attributes. With each increment of the counter value in the App component, the MyCounter component refreshes to display the new value.

The App component becomes as follows:

App component (file src/App.vue)

```
<script setup>

import { ref, computed } from 'vue';
import MyCounter from './components/MyCounter.vue';

const count = ref(0);
const doubleCount = computed(()=>count.value*2);
const increment = () => {
  count.value++;
};

</script>

<template>
  <MyCounter :count="count" :doubleCount="doubleCount" />
  <br /><br />
  <button @click="increment()">count+1</button>
</template>
```

The logic for incrementing the counter is implemented in the App component. The counter values (count and doubleCount) are transmitted in the attributes of the MyCounter component, which then displays them. The MyCounter component is refreshed each time one of its attributes is modified.

Step 1: Using Attributes in the <template> Section of the MyCounter Component

The MyCounter component, which utilizes the transmitted attributes, becomes as follows:

MyCounter component (file src/components/MyCounter.vue)

```
<script setup>

import { defineProps } from "vue"

// Enabling access to the attributes count and doubleCount in
the template
defineProps(["count", "doubleCount"]);

</script>

<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br/>
  Input : <input type="text" v-bind:value="count" />

</template>
```

The `defineProps(["count", "doubleCount"])` method identifies the attributes `count` and `doubleCount`, which will then be directly used by their names in the `<template>` section.

Note that the `props` variable typically returned by `defineProps()` is unnecessary here. It would be useful if you wanted to use the attribute values in the `<script setup>` section.

Let's verify that everything is working:

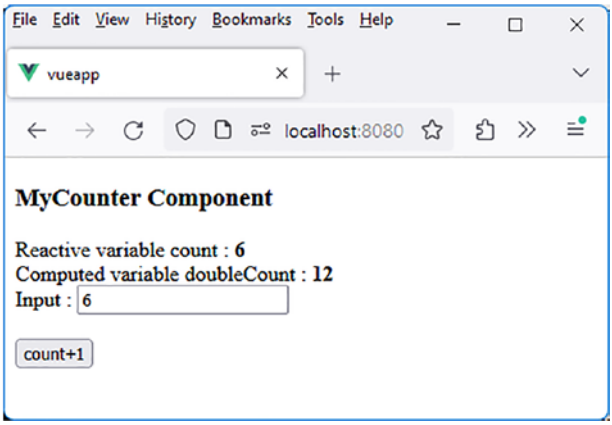

Figure 2-6. *Updating the counter through its attributes*

Step 2: Using Attributes in th <script setup> Section of the MyCounter Component

If you want to use the transmitted `count` and `doubleCount` attributes in the `<script setup>` section of the `MyCounter` component, you need to access them directly using the `props` variable, in the form of `props.count` and `props.doubleCount`. The variables `count` and `doubleCount`, corresponding to the attributes, can only be used under these names in the `<template>` section.

Let's use `props.count` in the `<script setup>` section. As this value is updated with each increment, we display its value in the `onUpdated()` lifecycle method.

Using props.count in <script setup> (file src/components/MyCounter.vue)

```
<script setup>

import { defineProps, onUpdated } from "vue"

const props = defineProps(["count", "doubleCount"]);

console.log("setup : count = ", props.count);

onUpdated(() => {
  console.log("updated : count = ", props.count);
})

</script>

<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br/>
  Input : <input type="text" v-bind:value="count" />

</template>
```

We display the value of `props.count` in the `<script setup>` section of the component (creation) and then with each update in the `onUpdated()` method.

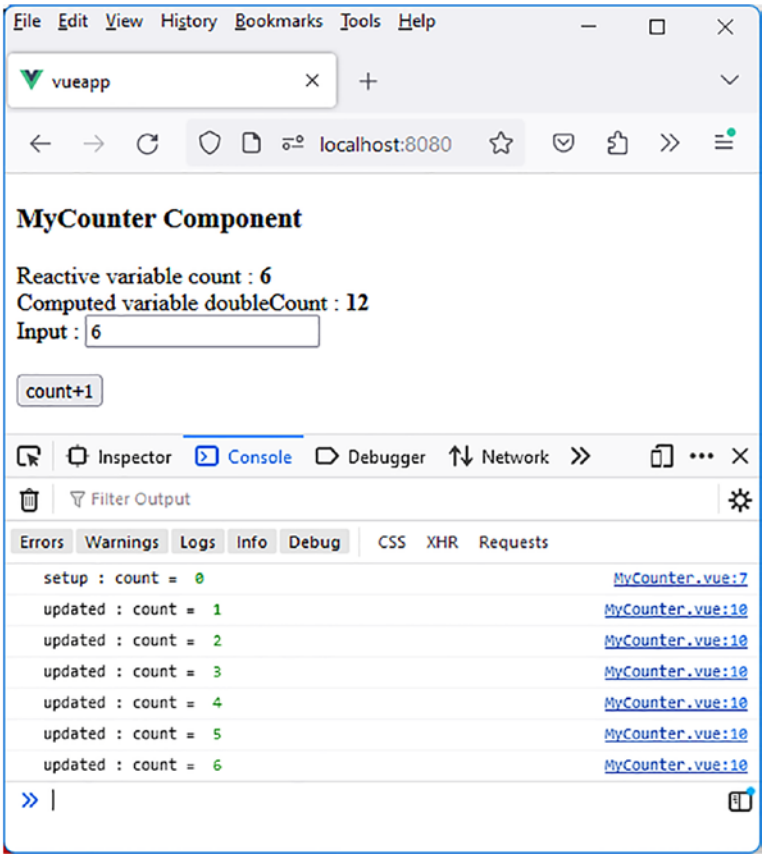

Figure 2-7. *Using attributes in <script setup>*

Through this example, we can observe that updating at least one of the attributes of a component refreshes the entire component.

v-if and v-else Directives

The `v-if` and `v-else` directives make it easy to write conditional tests in a component's template.

Let's use the example of the previous counter, adding a Start button to start the counter. Once the counter is started by clicking the Start button, the Start button is replaced by the Stop button, which allows stopping the counter. Therefore, the counter is started or stopped (depending on its state) by alternately clicking on the displayed Start or Stop button.

The `v-if` and `v-else` directives will allow us to alternately display the Start button or the Stop button:

- The `v-if` directive is used by specifying a JavaScript expression that represents a boolean value. If the value of the expression is `true`, the element using this directive is inserted into the page; otherwise, it is not.
- The `v-else` directive is used on the following element (at the same level). The element using the `v-else` directive will be inserted into the page if the one using `v-if` is not.

If the `v-else` directive is used, it must follow an element with a `v-if` directive.

In the following, the `App` component is restored to its initial state:

App component (file src/App.vue)

```
<script setup>
import MyCounter from './components/MyCounter.vue'
</script>
<template>
<MyCounter />
</template>
```

Let's now write the `MyCounter` component, which alternately displays the Start and Stop buttons. We will first code it in an intuitive way, but it won't work. Then, we'll see the modifications needed to achieve the desired result.

Step 1: Writing the MyCounter Component in an Intuitive (but Nonfunctional…) Way

Based on what we have previously explained, it would be natural to code the `MyCounter` component as follows:

MyCounter component with alternated Start and Stop buttons (file src/components/MyCounter.vue)

```
<script setup>
import { ref, computed } from "vue"

let timer = null;
const count = ref(0);
const doubleCount = computed(() => count.value * 2);

const increment = () => {
  count.value++;
};

const start= () => {
  timer = setInterval(() => increment(), 1000);
}

const stop = () => {
  clearInterval(timer);
  timer = null;
}

</script>
```

```
<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br /><br />
  <button v-if="!timer" @click="start()">Start</button>
  <button v-else @click="stop()">Stop</button>

</template>
```

The interesting part is the one written with the `v-if` and `v-else` directives:

- The `v-if` directive displays the Start button if the value of the `timer` variable is `null`, which is the case when the HTML page is initially displayed because the `timer` variable is initialized to `null`.
- The `v-else` directive displays the Stop button if the `timer` variable has a different value (i.e., not `null`).

This code looks logical and functional. Let's try to see the result obtained.

When the program is launched, the counter is at 0, and the Start button is displayed. This corresponds to the executed `v-if` directive.

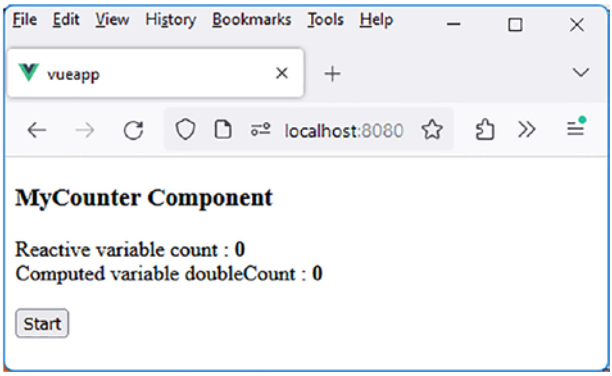

Figure 2-8. *Start button displayed*

Let's click the Start button. The counter starts, and the Stop button is displayed:

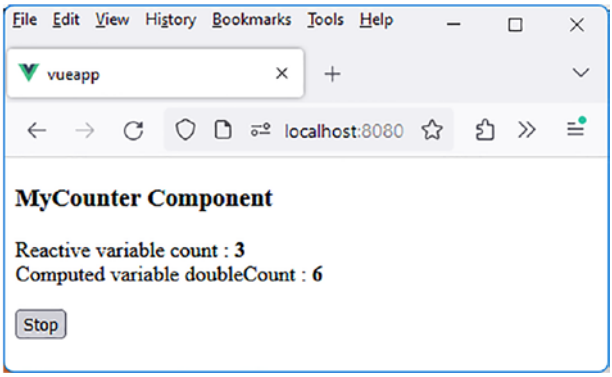

Figure 2-9. *The counter has started*

The Stop button is displayed, corresponding to the execution of the `v-else` directive. Everything seems to be working.

Let's click the Stop button to stop the counter:

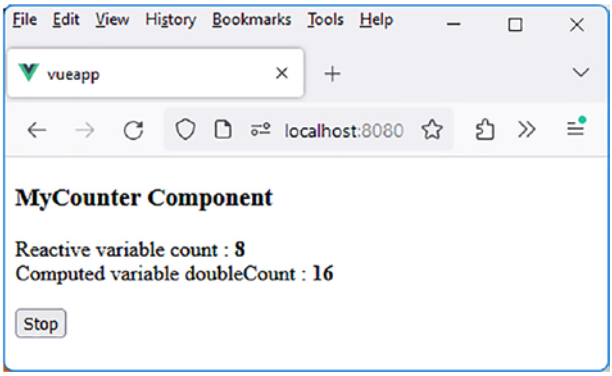

Figure 2-10. *The counter has been stopped*

Clicking the Stop button has stopped the counter. However, the button label is still "Stop" when it should be "Start." Additionally, clicking the button again does not restart the counter, which remains stuck at the stopped value.

So there is an issue. Let's explain why and resolve it now.

Step 2: Writing the MyCounter Component After Corrections (And Functional!)

The frequently made mistake is using a nonreactive variable in a directive, as seen with the `timer` variable, which is not reactive but is used in the `v-if` directive. Since the `timer` variable is not reactive, its modification is not considered by the `v-if` directive, which observes changes only on reactive variables.

We simply need to transform the `timer` variable into a reactive variable and modify the code where it is used, using `timer.value` instead of just `timer`.

The timer variable is defined as reactive (file src/components/MyCounter.vue)

```
<script setup>

import { ref, computed } from "vue"

const timer = ref(null);
const count = ref(0);
const doubleCount = computed(() => count.value * 2);

const increment = () => {
  count.value++;
};

const start= () => {
  timer.value = setInterval(() => increment(), 1000);
}

const stop = () => {
  clearInterval(timer.value);
  timer.value = null;
}

</script>

<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br /><br />
  <button v-if="!timer" @click="start()">Start</button>
  <button v-else @click="stop()">Stop</button>

</template>
```

After stopping the counter, the Start button is now visible, and the counter can restart:

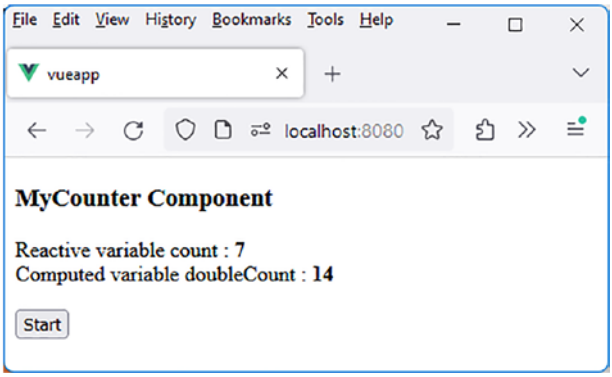

Figure 2-11. *The counter can restart*

Clicking the Start button restarts the counter:

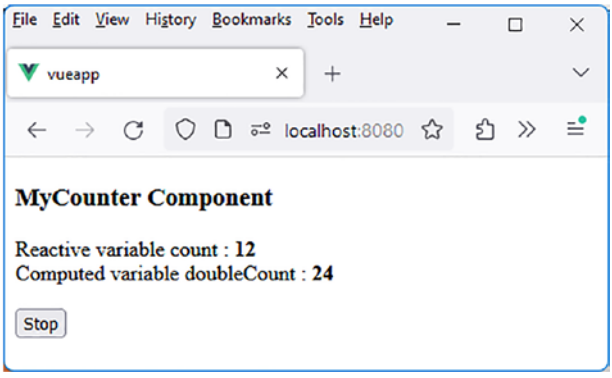

Figure 2-12. *The counter has restarted*

v-show Directive

The `v-show` directive is similar to the `v-if` directive. The difference is that `v-if` inserts the element into the page if the condition specified in the directive is `true` whereas `v-show` inserts it in all cases but only displays it if the condition is `true` (the `v-show` directive uses the element's style to hide or show it as needed).

Using the `v-show` directive in the previous template, we write the following:

Using the v-show directive (file src/components/MyCounter.vue)

```
<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br /><br />
  <button v-show="!timer" @click="start()">Start</button>
  <button v-show="timer" @click="stop()">Stop</button>

</template>
```

The `v-show` directive, being used with a condition, requires writing the negation of the first condition in the second `v-show` directive. Using `v-if` and `v-else` avoids writing two conditions (only one condition will be written in the `v-if` directive).

The result obtained is the same as the previous one.

v-for Directive

The `v-for` directive allows for looping, enabling the insertion of the directive-containing element into the HTML page multiple times.

The value of the `v-for` directive can be written in several ways, depending on the need:

1. **First Form of Writing:** `v-for="i in n"`. This form of writing allows for a loop from 1 to the value `n`.
2. **Second Form of Writing:** `v-for="item, i in items"`. This form of writing allows for traversing the `items` array and performing an operation for each element `item` in the array.

Let's start by studying the writing form `v-for="i in n"`, which allows for a loop from 1 to the value `n`.

Step 1: v-for Directive in the Form v-for="i in n"

The variable `i` corresponds to the index in the loop (starting from 1), while the variable `n` corresponds to the final value of the index in the loop.

To use this form of the `v-for` directive, suppose we want to display multiple counters like the previous one. We would then have a new `MyCounters` component that incorporates several `MyCounter` components using a `v-for` directive. For example, we would write `<MyCounters :nb="3" />` to indicate that we want to display three `MyCounter` components on the page. The `nb` attribute indicates how many `MyCounter` components we want to display in the HTML page.

The `App` component is modified to display the `MyCounters` component:

App component displaying the MyCounters component (file src/App.vue)

```
<script setup>

import MyCounters from "./components/MyCounters.vue"

</script>

<template>

<MyCounters :nb="3" />

</template>
```

Notice that if we specify `:nb="3"` instead of `nb="3"`, it allows us to transmit the numeric value 3 to the `MyCounters` component rather than the string "3". Indeed, if an attribute is preceded by the ":" sign, it means that we should interpret the following value (in quotes or not) as a JavaScript expression.

The `MyCounters` component uses the `v-for` directive to display the `MyCounter` components:

MyCounters component (file src/components/MyCounters.vue)

```
<script setup>

import MyCounter from "./MyCounter.vue";
import { defineProps } from 'vue';

defineProps(["nb"]);

</script>

<template>

<MyCounter v-for="i in nb" />

</template>
```

The v-for directive placed on an element allows displaying that element as many times as indicated in the value of the directive (here, from 1 to the value of the nb variable).

The MyCounter component is the same as before:

MyCounter component (file src/components/MyCounter.vue)

```
<script setup>

import { ref, computed } from "vue"

const timer = ref(null);
const count = ref(0);
const doubleCount = computed(() => count.value * 2);

const increment = () => {
  count.value++;
};

const start= () => {
  timer.value = setInterval(() => increment(), 1000);
}

const stop = () => {
  clearInterval(timer.value);
  timer.value = null;
}

</script>

<template>

  <h3>MyCounter Component</h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
```

```
  <br /><br />
  <button v-if="!timer" @click="start()">Start</button>
  <button v-else @click="stop()">Stop</button>

</template>
```

Let's look at the result obtained:

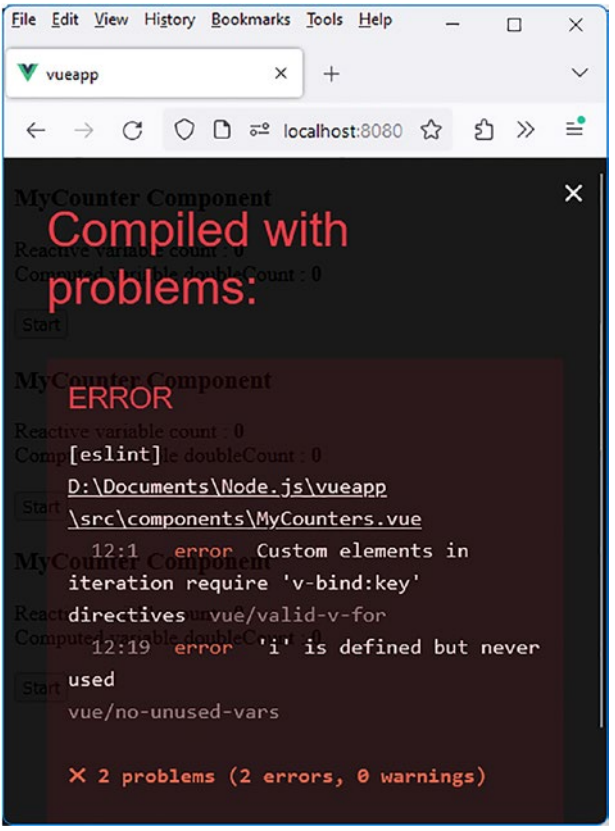

Figure 2-13. *Misuse of the v-for directive*

We encounter two errors:

1. It is necessary to use the `v-bind:key` directive when using the `v-for` directive.

2. The variable `i` used in the `v-for="i in nb"` directive is defined but not used.

These errors are quite common when starting to develop Vue.js applications. We explain how to resolve these errors in the following section. It is sufficient to use an attribute named `key` in the `v-for` directive.

Step 2: Use the Key Attribute with the v-for Directive

The previous error shows that the use of the `v-for` directive must be accompanied by the use of the `v-bind:key` directive. This `v-bind:key` directive allows defining a special attribute reserved for Vue.js (`key` attribute) that provides a unique key to each repetitively inserted element.

The `key` attribute is an attribute used internally by Vue.js and cannot be used by us in the component on which it is positioned.

To resolve the previous error, it is sufficient to specify a `key` attribute with the value of the variable `i`. As the variable `i` varies from 1 to `nb`, the `key` attribute of each inserted `MyCounter` component will thus have a different value, which is what we want.

By using the `key` attribute in this way, we solve both of the previous errors at the same time.

Use of the key attribute (file src/components/MyCounters.vue)

```
<script setup>

import MyCounter from "./MyCounter.vue";
import { defineProps } from 'vue';

defineProps(["nb"]);

</script>
```

```
<template>
<MyCounter v-for="i in nb" v-bind:key="i" />
</template>
```

The `v-bind:key="i"` directive can be simplified by simply writing `:key="i"`. We had explained this writing simplification previously when using the `v-bind` directive.

Now we have the following:

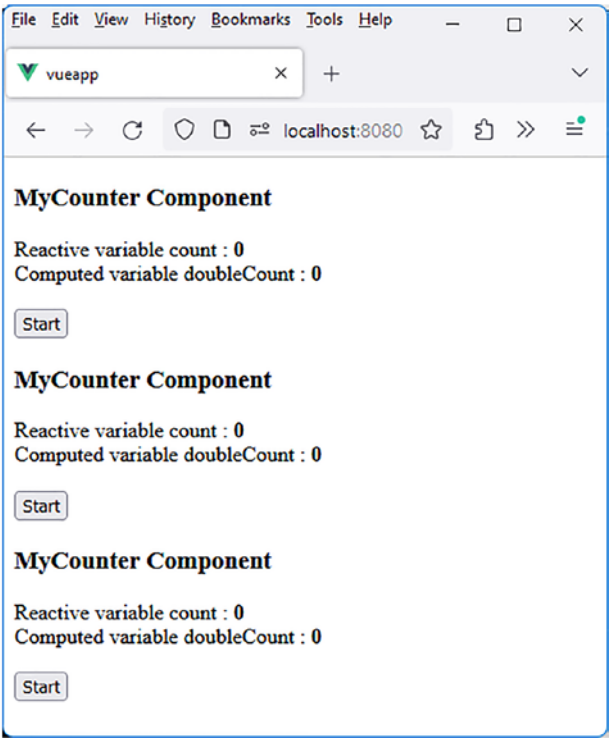

Figure 2-14. *Display of three counters in the MyCounters component*

Step 3: Rules Regarding the Key Attribute

The key attribute is mandatory when using a `v-for` directive, regardless of the form of the directive. Here are two rules regarding this attribute:

1. The value of the key attribute must be unique in the list.
2. And this value should never be modified (in the case where the list is updated).

Indeed, if a value of the key attribute is assigned to a list item, this value must always be retained for that item. Vue.js uses this value to determine if a list item is still present in the list, in order to refresh the list correctly (in the case of additions and deletions of items in the list). Therefore, if, as in our previous example, the index of the list item is used in the key attribute, it will only work if the list is static (as is the case in our example).

To build dynamic lists, it is generally recommended to use a unique ID identifier associated with each list item.

Step 4: Use an Index in the Component That Uses v-for

Suppose we want to number the previous counters, here from 1 to 3. The key attribute contains this value from 1 to 3 but cannot be used directly in the component because it is prohibited by Vue.js and produces an error. The key attribute is indeed for internal use by Vue.js.

To remedy this, simply create a new attribute, named, for example, `index`, which will have the same value. The `index` attribute can be used directly in the `MyCounter` component.

The `MyCounters` component is modified to set the `index` attribute following the `v-for` directive:

Index attribute positioned following the v-for directive (file src/components/MyCounters.vue)

```
<script setup>

import MyCounter from "./MyCounter.vue";
import { defineProps } from 'vue';

defineProps(["nb"]);

</script>

<template>

<MyCounter v-for="i in nb" :key="i" :index="i" />

</template>
```

The `index` attribute is positioned on the `MyCounter` component following the `v-for` directive. Its value will be that of the variable `i` and will thus be 1, then 2, and finally 3.

The `index` attribute is usable within the `MyCounter` component. It is sufficient to retrieve this attribute using the `defineProps(["index"])` method.

Note that if you write `defineProps(["key"])`, this will result in an error, as explained earlier.

Usage of the index attribute in the MyCounter component (file src/components/MyCounter.vue)

```
<script setup>

import { ref, computed, defineProps } from "vue"

const timer = ref(null);
const count = ref(0);
const doubleCount = computed(() => count.value * 2);
```

```
defineProps(["index"]);

const increment = () => {
  count.value++;
};

const start= () => {
  timer.value = setInterval(() => increment(), 1000);
}

const stop = () => {
  clearInterval(timer.value);
  timer.value = null;
}

</script>

<template>

  <h3> {{index}} - MyCounter Component </h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br /><br />
  <button v-if="!timer" @click="start()">Start</button>
  <button v-else @click="stop()">Stop</button>

</template>
```

The counters are now numbered:

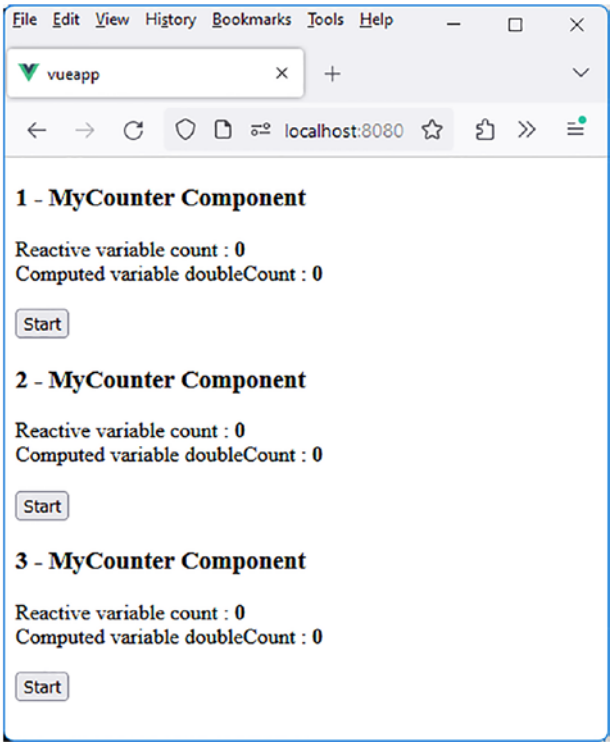

Figure 2-15. *Numbering of counters*

The counters are now numbered starting from 1. Of course, each counter starts and stops independently. For example, let's start the second counter:

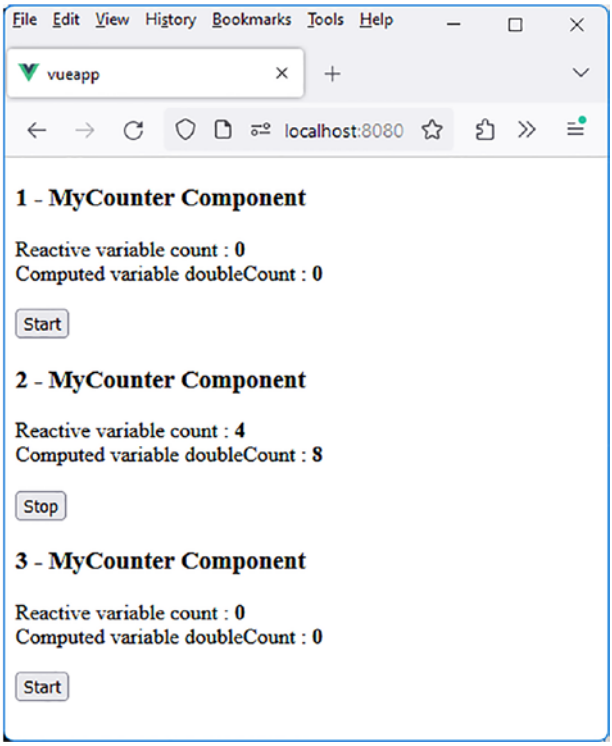

Figure 2-16. *Counter 2 started*

We have previously seen how to use the `v-for` directive in the form `v-for="i in n"`, which allows looping from 1 to `n`. Now, let's explore how to loop through an array of elements using another form of the `v-for` directive, namely, `v-for="(item, i) in items"`.

Step 5: v-for Directive in the Form v-for="(item, i) in items"

The second form of the `v-for` directive enables looping through an array of elements represented by the variable `items`. Each element in the array is represented by the variable `item`. The variable `i` corresponds to the index of the element in the array, starting from 0 (unlike the previous form of the directive where it started from 1).

It is possible to omit the parentheses and write the directive as `v-for="item, i in items"` and also as `v-for="item in items"` if the index `i` is not being used.

Let's assume the variable `items` is an array indicating, for each counter, the starting value (`init` property) and the end value of that counter (`end` property). In the following component code, the variable `items` is referred to as `limits`:

The App component is as follows:

App component (file src/App.vue)

```
<script setup>

import MyCounters from './components/MyCounters.vue'

const limits = [
  {init: 0, end: 10},
  {init: 5},
  {end: 10}
];

</script>

<template>

<MyCounters :limits="limits" />

</template>
```

The limits array is an array of objects { init, end }, indicating for each counter its initial value (init) and final value (end):

- If the initial value init is not specified, it is considered to be 0.
- If the final value end is not specified, it is considered infinite (the counter has no end limit).

The App component incorporates the MyCounters component in the form of <MyCounters :limits="limits" />. This way, the limits array is passed to the MyCounters component as the limits attribute, which will be utilized within the component.

The MyCounters component displays MyCounter components based on the content of the limits array:

MyCounters component (file src/components/MyCounters.vue)

```
<script setup>

import MyCounter from "./MyCounter.vue";
import { defineProps } from 'vue';

defineProps(["limits"]);

</script>

<template>

<MyCounter v-for="(limit, i) in limits" :key="i" :index="i"
:limit="limit" />

</template>
```

The MyCounter component receives the attributes index and limit:

- The index attribute represents the index of the element in the list, starting from 0.

- The `limit` attribute corresponds to an object `{ init, end }`, as defined in the `limits` array.

The `MyCounter` component is slightly modified to display the `init` and `end` parameters it receives through the `limit` attribute. If an initial value (`init`) is not transmitted, it is displayed as 0. If a final value (`end`) is not transmitted, "infinity" is displayed.

MyCounter component (file src/components/MyCounter.vue)

```
<script setup>
import { ref, computed, defineProps } from "vue"

const props = defineProps(["limit", "index"]);
const init = props.limit.init || 0;        // 0 if no init
                                           indicated
const end = props.limit.end || undefined;  // undefined if no
                                           end indicated
const timer = ref(null);
const count = ref(init);
const doubleCount = computed(() => count.value * 2);

const increment = () => {
  if (end == undefined || count.value < end) count.value++;
};

const start= () => {
  timer.value = setInterval(() => increment(), 1000);
}

const stop = () => {
  clearInterval(timer.value);
  timer.value = null;
}

</script>
```

```
<template>

  <h3> {{index}} - MyCounter Component </h3>
  init = {{init}}, end = {{end ? end : "infinity"}}
  <br />
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br /><br />
  <button v-if="!timer" @click="start()">Start</button>
  <button v-else @click="stop()">Stop</button>

</template>
```

The three counters are displayed here:

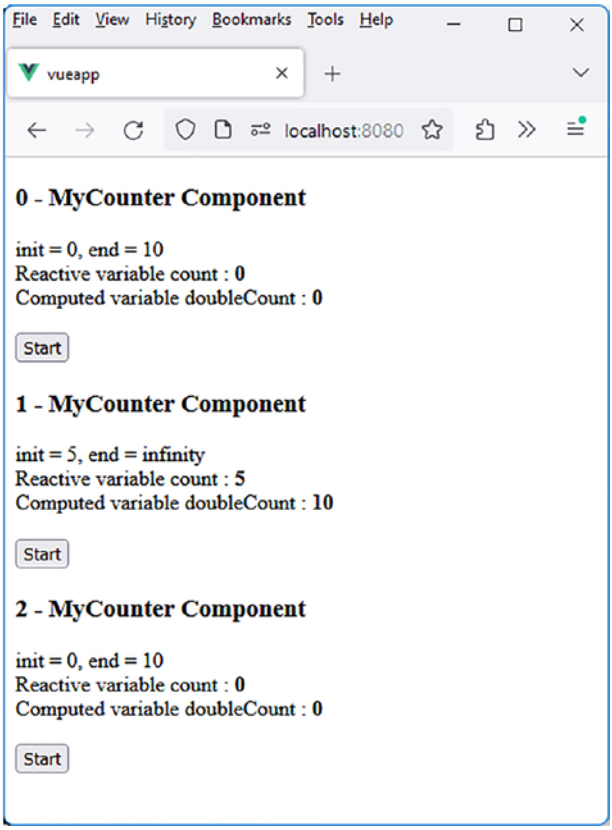

Figure 2-17. *Counters with displayed init and end attributes*

Each counter now has the `init` and `end` attributes displayed, retrieved from the `limit` attribute of the `MyCounter` component.

v-model Directive

The `v-bind` directive we studied earlier allows updating an attribute associated with a reactive variable when that variable is modified. For example, the instruction `<input v-bind:value="count" />` updates

the value attribute of the input field when the reactive variable count is changed. Therefore, the content of the input field is automatically updated.

Conversely, modifying the value attribute of the input field does not update the associated reactive variable count. To achieve this, the v-model directive is used.

The v-model directive enables two-way binding (attribute to variable and variable to attribute), while the v-bind directive allows modification in only one direction (variable to attribute).

Step 1: Difference Between v-bind and v-model Directives

To observe the behavioral difference between the v-bind and v-model directives, let's use the MyCounter component, which displays the reactive variable count and two input fields:

- The first input field is managed by v-bind.
- The second input field is managed by v-model.

The MyCounter component is directly inserted into the App component:

App component (file src/App.vue)

```
<script setup>
import MyCounter from './components/MyCounter.vue'
</script>
<template>
<MyCounter />
</template>
```

The MyCounter component becomes the following:

MyCounter component (file src/components/MyCounter.vue)

```
<script setup>
import { ref, computed } from "vue"

const count = ref(0);
const doubleCount = computed(() => count.value * 2);

</script>

<template>

  <h3> MyCounter Component </h3>
  Reactive variable count : <b>{{ count }}</b>
  <br />
  Computed variable doubleCount : <b>{{ doubleCount }}</b>
  <br /><br />
  Input for count (using v-bind): <input type="text"
  :value="count" />
  <br/><br/>
  Input for count (using v-model): <input type="text"
  v-model="count" />

</template>
```

Indeed, to use the v-bind directive, you can simplify the syntax by writing :value="count" instead of v-bind:value="count". During program execution, the value of the reactive variable (here, 0) initializes the content of both input fields. This is achieved through the functionality of the v-bind directive, and it's worth noting that the v-model directive also incorporates the behavior of v-bind.

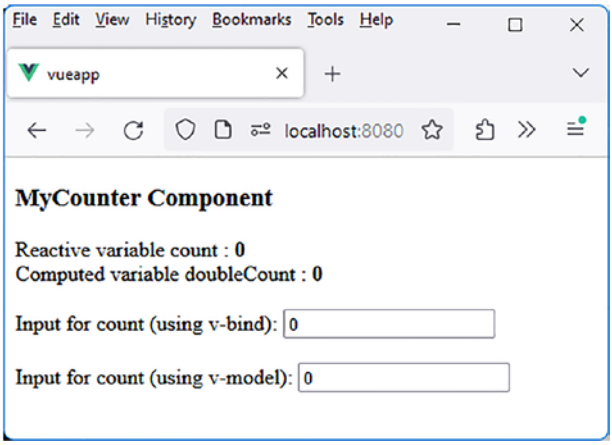

Figure 2-18. *Initialization of input fields (v-bind operation)*

The distinction between the `v-bind` and `v-model` directives becomes apparent when modifying the values in the input fields. If you modify the first input field using the `v-bind` directive, the reactive variable `count` does not get updated:

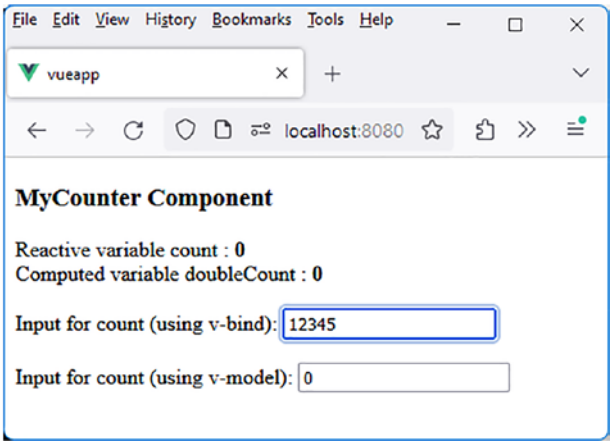

Figure 2-19. *Modification of the input field using v-bind*

Indeed, an attribute managed by the `v-bind` directive is updated if the associated reactive variable is modified, but not vice versa. Therefore, modifying the `value` attribute of the input field does not alter the associated reactive variable `count`.

On the other hand, if you modify the second input field using the `v-model` directive, the reactive variable `count` updates (thanks to `v-model`), leading to the modification of the first input field using `v-bind` (due to the behavior of the `v-bind` directive).

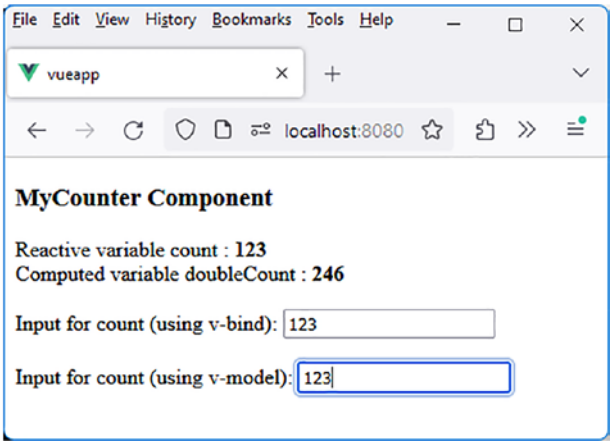

Figure 2-20. *Modification of the input field using v-model*

Step 2: Using the v-model Directive in Forms

We have seen the usefulness of the `v-model` directive in managing an input field, automatically capturing the content of the input field in a reactive variable.

The `v-model` directive is widely employed in input forms to easily retrieve the values entered/checked/selected in the form. Each form field is simply connected to a reactive variable using the `v-model` directive.

Let's use the `v-model` directive to retrieve and display information entered in a form, allowing the input of information displayed in various forms:

- An input field for the person's name
- A selection list to choose the year of birth
- Radio buttons to choose marital status
- Finally, check boxes to validate terms of use and general sales conditions

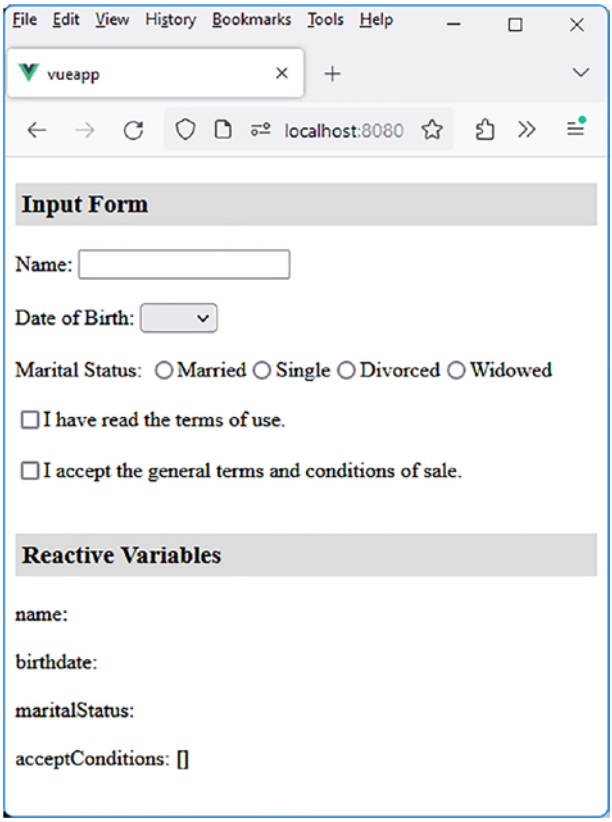

Figure 2-21. *Input form managed by v-model*

The advantage of using the `v-model` directive with the fields in this form is that modifying each field immediately updates the reactive variable associated with that field. The values of the reactive variables associated with each field are displayed below the form.

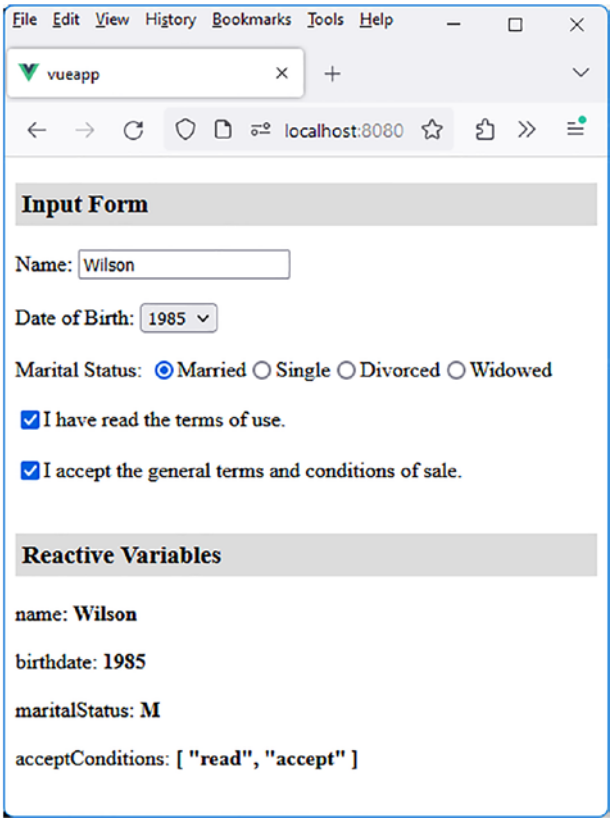

Figure 2-22. *Reactive variables updated according to input*

The fact that each form field is linked to a reactive variable through `v-model` allows for retrieving the current input or selection in the form. Let's explore how to manage each form field based on its type (input field, selection list, radio buttons, or check boxes).

A new component called `MyForm` is used, which will contain the previous display. The component file `MyForm.vue` is created in the `src/components` directory. The `MyForm` component is inserted into the `App` component:

App component using the MyForm component (file src/App.vue)

```
<script setup>
import MyForm from "./components/MyForm.vue"
</script>
<template>
<MyForm />
</template>
```

Step 3: Managing Input Fields with v-model

This case is the one we previously studied, which served as an introduction to the `v-model` directive. The `MyForm` component becomes the following:

Input field in the MyForm component (file src/components/MyForm.vue)

```
<script setup>
import { ref } from "vue"
const name = ref("");
</script>
<template>
<h3>Input Form</h3>
Name: <input type="text" v-model="name" />
```

```
<br/><br/>

<h3>Reactive Variables</h3>
name: <b>{{name}}</b>
<br/><br/>

</template>


<style scoped>
h3 {
  background-color: gainsboro;
  padding: 5px;
}
</style>
```

After entering data into the field, you get:

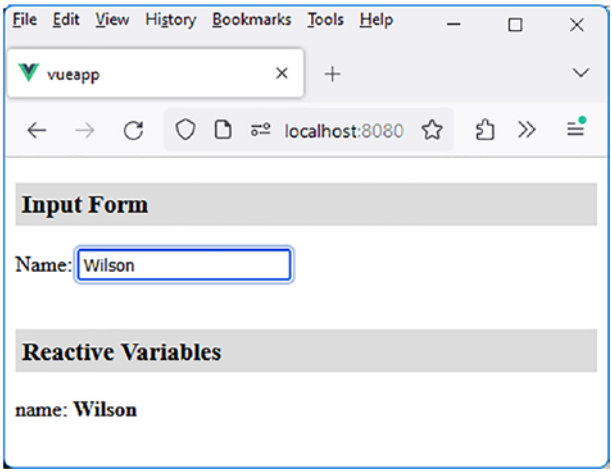

Figure 2-23. *Entering the name in the form*

Step 4: Managing Selection Lists with v-model

Now, let's see how to retrieve the selected value in a list, for example, the year of birth.

Selection list in the MyForm component (file src/components/MyForm.vue)

```
<script setup>

import { ref } from "vue"

const name = ref("");
let dates = [];
for (let year=2023; year > 1900; year--) dates.push(year);
const birthdate = ref("");

</script>

<template>

<h3>Input Form</h3>
Name: <input type="text" v-model="name" />
<br/><br/>
Date of Birth:
  <select v-model="birthdate" >
    <option v-for="date in dates" :value="date"
    :key="date">{{date}}</option>
  </select>
<br/><br/>

<h3>Reactive Variables</h3>
name: <b>{{name}}</b>
<br/><br/>
birthdate: <b>{{birthdate}}</b>
```

```
<br/><br/>

</template>

<style scoped>
h3 {
  background-color: gainsboro;
  padding: 5px;
}
</style>
```

The `v-model` directive is used on the `<select>` element. Each year in the list is displayed using a `v-for` directive, iterating over `"date in dates"`. The `dates` array has been previously populated in the `<script setup>` section of the component. If a date is chosen from the list, the selected date is displayed in the reactive variable `birthdate`:

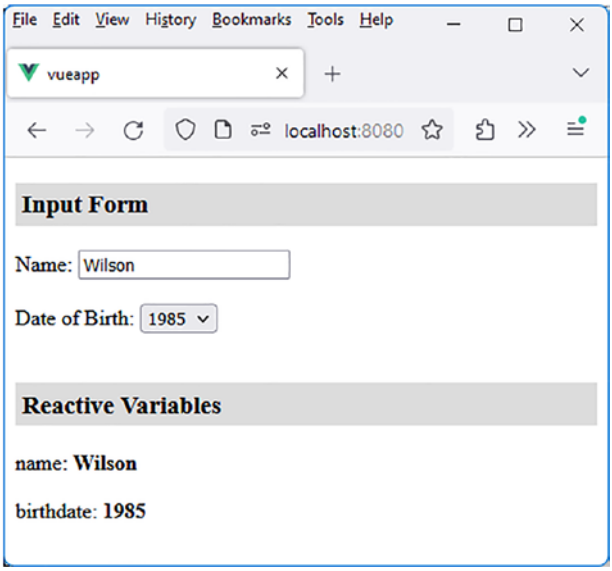

Figure 2-24. *Selecting a date in the form*

Step 5: Managing Radio Buttons with v-model

Now, let's see how to retrieve the value of the selected radio button. Here, radio buttons are used to choose marital status: Married, Single, Divorced, Widowed. Only one radio button at a time is selected in the list.

Managing radio buttons in the form (file src/components/MyForm.vue)

```
<script setup>

import { ref } from "vue"

const name = ref("");
let dates = [];
for (let year=2023; year > 1900; year--) dates.push(year);
const birthdate = ref("");
const maritalStatus = ref("");

</script>

<template>

<h3>Input Form</h3>
Name: <input type="text" v-model="name" />
<br/><br/>
Date of Birth:
  <select v-model="birthdate" >
    <option v-for="date in dates" :value="date" :key="date">
    {{date}}</option>
  </select>
<br/><br/>
Marital Status:
```

```
  <input type="radio" value="M" id="maried"
  v-model="maritalStatus">
  <label for="maried">Married</label>
  <input type="radio" value="S" id="single"
  v-model="maritalStatus">
  <label for="single">Single</label>
  <input type="radio" value="D" id="divorced"
  v-model="maritalStatus">
  <label for="divorced">Divorced</label>
  <input type="radio" value="W" id="widower"
  v-model="maritalStatus">
  <label for="widower">Widowed</label>
<br/><br/>

<h3>Reactive Variables</h3>
name: <b>{{name}}</b>
<br/><br/>
birthdate: <b>{{birthdate}}</b>
<br/><br/>
maritalStatus: <b>{{maritalStatus}}</b>
<br/><br/>

</template>

<style scoped>
h3 {
  background-color: gainsboro;
  padding: 5px;
}
</style>
```

The `v-model` directive is applied to each `<input type="radio">` element. The same reactive variable, `maritalStatus`, is associated with each element.

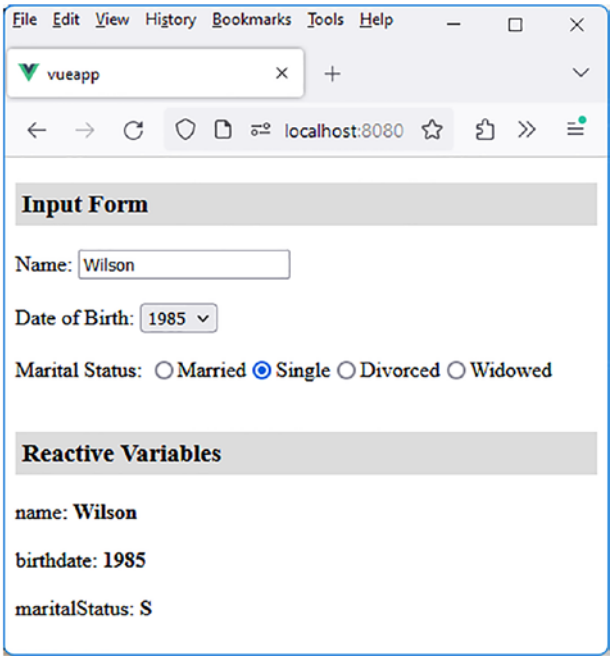

Figure 2-25. *Managing radio buttons in the form*

Step 6: Managing Check Boxes with v-model

Finally, let's see how to handle check boxes in forms. Here, two check boxes are used, which can be checked independently:

- The first one indicates that the terms of use have been read.
- The second one indicates that the general terms and conditions of sale have been accepted.

Managing check boxes in the form (file src/components/MyForm.vue)

```
<script setup>

import { ref } from "vue"

const name = ref("");
let dates = [];
for (let year=2023; year > 1900; year--) dates.push(year);
const birthdate = ref("");
const maritalStatus = ref("");
const acceptConditions = ref([]);

</script>

<template>

<h3>Input Form</h3>
Name: <input type="text" v-model="name" />
<br/><br/>
Date of Birth:
  <select v-model="birthdate" >
    <option v-for="date in dates" :value="date" :key="date">
    {{date}}</option>
  </select>
<br/><br/>
Marital Status:
  <input type="radio" value="M" id="maried"
  v-model="maritalStatus">
  <label for="maried">Married</label>
  <input type="radio" value="S" id="single"
  v-model="maritalStatus">
```

```
  <label for="single">Single</label>
  <input type="radio" value="D" id="divorced"
  v-model="maritalStatus">
  <label for="divorced">Divorced</label>
  <input type="radio" value="W" id="widower"
  v-model="maritalStatus">
  <label for="widower">Widowed</label>
<br/><br/>
<input type="checkbox" id="read" value="read"
            v-model="acceptConditions" />
<label for="read">I have read the terms of use.</label>
<br/><br/>
<input type="checkbox" id="accept" value="accept"
            v-model="acceptConditions" />
<label for="accept">I accept the general terms and conditions
of sale.</label>
<br/><br/>

<h3>Reactive Variables</h3>
name: <b>{{name}}</b>
<br/><br/>
birthdate: <b>{{birthdate}}</b>
<br/><br/>
maritalStatus: <b>{{maritalStatus}}</b>
<br/><br/>
acceptConditions: <b>{{acceptConditions}}</b>
<br/><br/>

</template>

<style scoped>
h3 {
```

```
  background-color: gainsboro;
  padding: 5px;
}
</style>
```

The reactive variable `acceptConditions`, which can contain the values of two check boxes, is initialized as an empty array [ ]. Depending on which check box is checked, its value will automatically be added to the `acceptConditions` array.

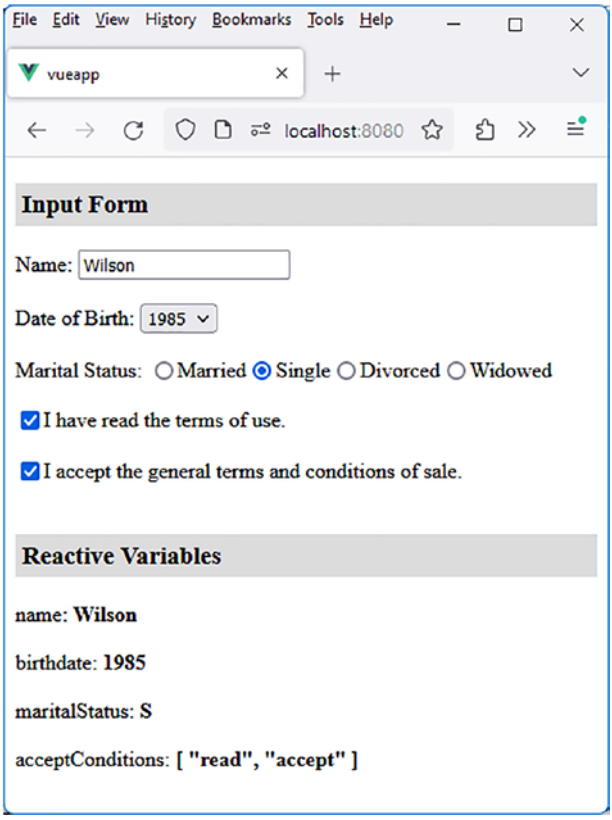

Figure 2-26. *Handling check boxes in the form*

Using Modifiers in Vue.js

Some directives in Vue.js can have what are called modifiers. These modifiers allow modifying the default behavior of the directive.

Take the example of the `v-model` directive mentioned earlier. It has the following modifiers:

- `lazy`: The `lazy` modifier changes the behavior of updating the reactive variable associated with the directive. It updates the variable only when exiting the input field, rather than updating it for every character entered. When the `lazy` modifier is used, Vue.js considers the `change` event (for updating the associated reactive variable) instead of the `input` event.
- `trim`: The `trim` modifier removes any leading or trailing spaces from the input field when updating the associated reactive variable.

A modifier is used after the name of the associated directive, prefixed with the "." character. For example, you write `v-model.trim="count"` or `v-model.lazy="count"`. Additionally, you can combine multiple modifiers by writing `v-model.trim.lazy="count"`.

Let's use the `lazy` modifier with the `v-model` directive, taking the name input field from the previous example. The `MyForm` component is modified to accommodate the `lazy` modifier in the directive.

Using the lazy modifier (file src/components/MyForm.vue)

```
<script setup>

import { ref } from "vue"

const name = ref("");

</script>
```

```
<template>

<h3>Input Form</h3>
Name: <input type="text" v-model.lazy="name" />
<br/><br/>

<h3>Reactive Variables</h3>
name: <b>{{name}}</b>
<br/><br/>

</template>

<style scoped>
h3 {
  background-color: gainsboro;
  padding: 5px;
}
</style>
```

Let's enter a name in the input field associated with `v-model`:

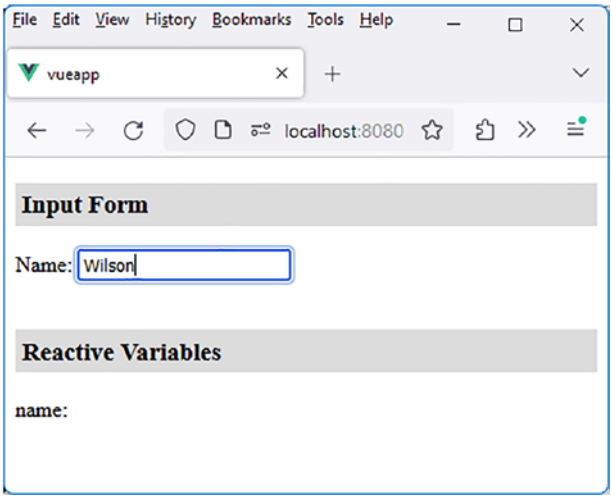

Figure 2-27. *Modifying an input field with the "lazy" modifier*

Now, we can see that the reactive variable no longer updates as we type, contrary to the usual behavior of the `v-model` directive. However, as soon as we exit the input field, the reactive variable `name` updates.

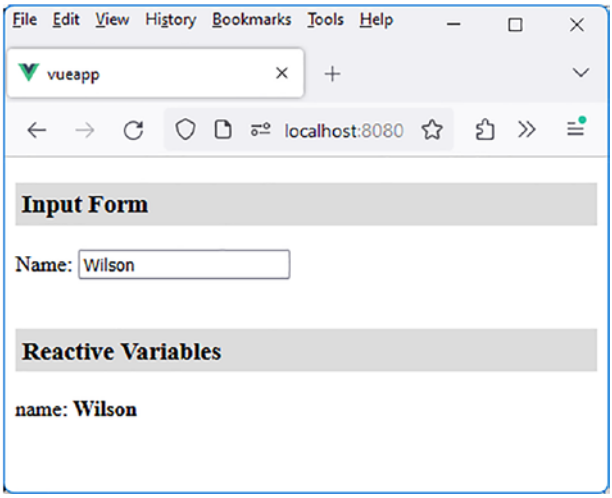

Figure 2-28. *Updating the reactive variable upon exiting the input field*

There are many modifiers available depending on the directives used. It is also possible to create new modifiers (see Chapter 5).

Conclusion

This chapter has provided us with an in-depth understanding of directives in Vue.js, ranging from using attributes to mastering directives like `v-bind`, `v-if`, and `v-model`. These skills are fundamental for building robust and maintainable Vue.js applications.

The next chapter will guide us through handling DOM events, a crucial skill to enhance interactivity in Vue.js applications.