CHAPTER 3

Day 3: Mastering Events in Vue.js

In this third day of our Vue.js study, we will focus on event handling, a crucial element for creating interactive applications with Vue.js.

You will learn how to intercept events related to user actions in the HTML page and use these events to enable components to exchange information. This skill is crucial for increasing the reactivity and interactivity of your Vue.js applications.

Intercepting Events

Vue.js allows you to perform a process when an event occurs using the `v-on` directive. To specify the event you want to handle, write its name after `v-on`, preceded by ":". For example, you write `v-on:click` to indicate that you want to handle the `click` event.

To handle a click on a button, you write the following:

Activate the increment() method when the button is clicked

```
<button v-on:click="increment()">Increment</button>
```

The value of the `v-on` directive corresponds to the action to be performed when the event occurs, in this case, calling the `increment()` function. Since the use of the `v-on` directive is so common, Vue.js allows

E. Sarrion, *Master Vue.js in 6 Days*, https://doi.org/10.1007/979-8-8688-0364-2_3

simplifying its syntax by using `@click` instead of `v-on:click`. Therefore, you can also write the following:

Activate the increment() method when the button is clicked

```
<button @click="increment()">Increment</button>
```

An example handling a click on a button in the `MyCounter` component to increment a reactive variable would be as follows:

App Component (file src/App.vue)

```
<script setup>

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

</script>

<template>

<MyCounter />

</template>
```

The file for the `MyCounter` component is as follows:

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

```
<script setup>

import { ref } from "vue"

const count = ref(0);

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

</script>

<template>
```

```
<h3>MyCounter Component</h3>
Reactive variable count : <b>{{count}}</b>
<br/><br/>
<button v-on:click="increment()">count+1</button>

</template>
```

After several clicks on the button, the reactive variable `count` has been incremented:

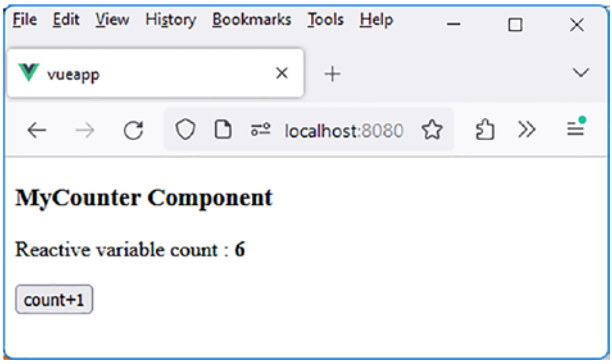

Figure 3-1. *Handling click on the button*

Writing in the Form "increment( )" or "increment"?

When writing the event handling function, you can use either `@click="increment()"` or simply `@click="increment"`.

Writing in the form `"increment()"` allows for specifying potential parameters, such as the increment value. You would then write `@click="increment(2)"` to increment by 2 with each click. The increment function with a parameter would look like this:

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

```
<script setup>
```

```
import { ref } from "vue"

const count = ref(0);

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

</script>

<template>

<h3>MyCounter Component</h3>
Reactive variable count : <b>{{count}}</b>
<br/><br/>
<button v-on:click="increment(2)">count+1</button>

</template>
```

Each click increments the counter by 2 instead of 1.

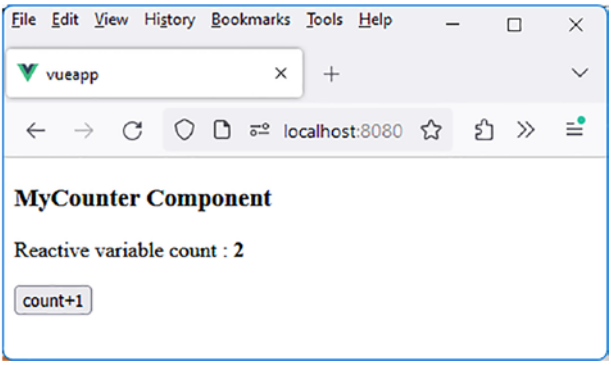

Figure 3-2. *Incrementing by 2 instead of 1*

Therefore, in cases where you want to indicate a parameter, you write it as `"increment(2)"`. In other cases, you can write `"increment()"` or `"increment"` interchangeably.

Using the Event Object

Sometimes it's useful for the event-handling function to know the parameters of that event. For example:

- For a mouse click, which mouse button was used, and the coordinates of the mouse pointer at the click
- For a keyboard press, which key was pressed

This information is available by using the `event` object (handled by Vue.js) in the event-handling function. The `event` object is available in all Vue.js event-handling functions.

Let's see a few examples of how to use the `event` object in event-handling functions.

Step 1: Filtering Pressed Keys

Here's a first example. Let's use the `event` object to filter allowed keys in an input field, allowing only digits from 0 to 9 and keys for moving or modifying the field.

We will use the `keydown` event for this purpose. The `keydown` event is generated with each keystroke before the key is processed and displayed in the input field. The `keyup` event is triggered after the key is processed, preventing the filtering of allowed keys if used here.

Therefore, we write the `MyCounter` component as follows:

Filtering keyboard keys (file src/components/MyCounter.vue)

```
<script setup>

import { ref } from "vue"

const count = ref();
```

```
const verifyKey = () => {
  const numbers = ["0", "1", "2", "3", "4", "5", "6", "7",
  "8", "9"];
  const moves = ["Backspace", "ArrowLeft", "ArrowRight",
                 "Delete", "Tab", "Home", "End"];

  let authorized;  // Allowed keys in the input field
                   authorized = [...numbers, ...moves];

  // If the key is not allowed, do not take it into account.
  // The event object is available here.
  if (!authorized.includes(event.key)) event.preventDefault();
}

</script>

<template>

<h3>MyCounter Component</h3>

Reactive variable count: <input type="text"
@keydown="verifyKey()" v-model="count" />
<br/><br/>
Entered value: <b>{{count}}</b>

</template>
```

The event-handling function `verifyKey()` uses the `event` object to determine the pressed key. The `key` property of the `event` object retrieves the code of the pressed key as a string:

- "Backspace": Represents the code for the backspace key
- "ArrowLeft" and "ArrowRight": Represent the codes for the left and right arrow keys

- "Delete": Represents the code for the Delete key
- "Tab": Represents the code for the Tab key (moves to the next field if it exists)
- "Home" and "End": Represent the codes for keys that go to the beginning or end of the input field

The numbers 0 to 9 are represented by the codes "0" to "9".

The main processing involves rejecting all keys that are not the allowed ones mentioned previously. For disallowed keys, the `event.preventDefault()` method of JavaScript is called, preventing the default event processing and therefore not taking the event into account. For all other allowed keys, the default processing is carried out (so the corresponding key is processed).

You can verify that only the keys from 0 to 9 are taken into account, as well as the arrow keys and field modification keys.

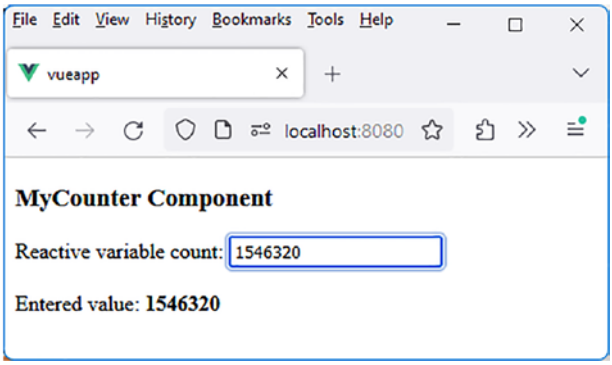

Figure 3-3. *Taking only digits into account in the field*

Step 2: Handling the Content of a Field

Let's enhance this program by displaying an error message if the value entered in the field is greater than 100.

We use the input event for this purpose, which is triggered when the key has been processed. If the value entered in the field is greater than 100, we display an error message using a reactive variable message initialized to "".

Do not exceed the value 100 in the input field (file src/components/MyCounter.vue)

```
<script setup>

import { ref } from "vue"

const count = ref();
const message = ref("");

const verifyKey = () => {
  const numbers = ["0", "1", "2", "3", "4", "5", "6", "7",
  "8", "9"];
  const moves = ["Backspace", "ArrowLeft", "ArrowRight",
                 "Delete", "Tab", "Home", "End"];

  let authorized;  // Allowed keys in the input field
  authorized = [...numbers, ...moves];

  // If the key is not allowed, do not take it into account.
  // The event object is available here.
  if (!authorized.includes(event.key)) event.preventDefault();
}

const verifyMax100 = () => {
  message.value = "";
  // The event object is available here.
  if (parseInt(event.target.value) > 100) message.value = "Do
  not exceed 100!";
}
```

```
</script>

<template>

<h3>MyCounter Component</h3>

Reactive variable count: <input type="text" v-model="count"
    @keydown="verifyKey()"
    @input="verifyMax100()" />
<br/><br/>
Entered value: <b>{{count}}</b>
<br/><br/>
Message : <b>{{message}}</b>

</template>
```

During the `input` event, the value entered in the input field is available in the variable `event.target.value`. If this value is greater than 100, the reactive variable `message` is initialized with the text of the error message to be displayed; otherwise, the message is cleared.

For example, if you enter the value 101 in the input field, you can see the error message as shown in Figure 3-4.

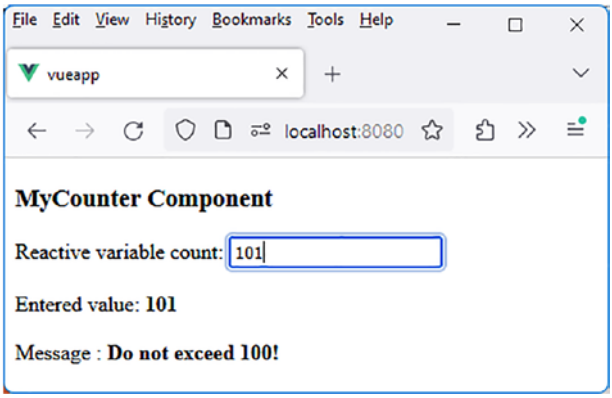

Figure 3-4. *Error message displayed if exceeding 100*

Step 3: Clearing the Field Content on Click

During a click in the input field, the current value is currently retained. Now, we want the field to be cleared upon clicking it so that a new value can be entered directly.

To achieve this, we need to handle the `focus` event on the input field.

Clear the field content on focus (file src/components/MyCounter.vue)

```
<script setup>

import { ref } from "vue"

const count = ref();
const message = ref("");

const verifyKey = () => {
  const numbers = ["0", "1", "2", "3", "4", "5", "6", "7",
  "8", "9"];
  const moves = ["Backspace", "ArrowLeft", "ArrowRight",
                 "Delete", "Tab", "Home", "End"];

  let authorized;  // Allowed keys in the input field
  authorized = [...numbers, ...moves];

  // If the key is not allowed, do not take it into account.
  // The event object is available here.
  if (!authorized.includes(event.key)) event.preventDefault();
}

const verifyMax100 = () => {
  message.value = "";
  // The event object is available here.
```

```
  if (parseInt(event.target.value) > 100) message.value = "Do
  not exceed 100!";
}

const eraseField = () => {
  event.target.value = "";
  count.value = "";
  message.value = "";
}

</script>

<template>

<h3>MyCounter Component</h3>

Reactive variable count: <input type="text" v-model="count"
    @keydown="verifyKey()"
    @input="verifyMax100()"
    @focus="eraseField()" />
<br/><br/>
Entered value: <b>{{count}}</b>
<br/><br/>
Message : <b>{{message}}</b>

</template>
```

The field is now cleared upon clicking it:

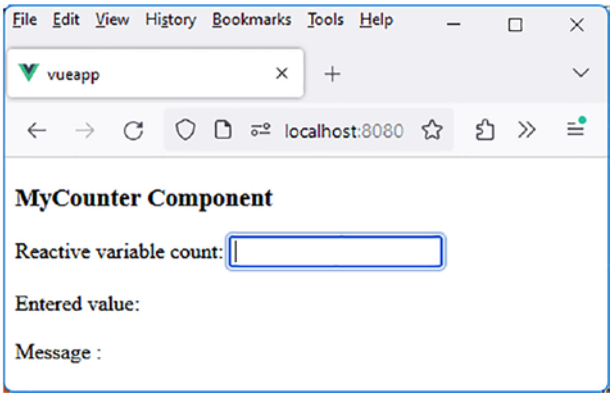

Figure 3-5. *The content of the field is cleared on click*

Communication Between a Child Component and a Parent Component

When creating an application, it is common to need to transmit information to a child component or send information back to its parent component.

Vue.js enables this through the following:

- Props (or attributes) for transmitting information from the parent component to the child component
- Events for sending information from the child component to the parent component

Let's explore these two types of communication through an example.

Step 1: Communicating with a Child Component

Communication from a parent component to a child component is done using attributes that are passed to it.

We have already used this type of communication, for example, with the `MyCounters` component in the form `<MyCounters :nb="3" />`.

App Component (file src/App.vue)

```
<script setup>

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

</script>

<template>

<MyCounters :nb="3" />

</template>
```

The `App` component (parent of the `MyCounters` component) here transmits the desired number of counters when displaying it.

Notice the use of `:nb` instead of just `nb`. Without the colon, the string "3" would be transmitted to the `MyCounters` component, rather than the JavaScript value 3 written within quotes.

The `MyCounters` component receives this new attribute and processes it using a `v-for` directive.

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

```
<script setup>
import MyCounter from "./MyCounter.vue";
import { defineProps } from "vue";

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

```
const nb = props.nb || 1;   // If the nb attribute is not
                            specified, it defaults to 1.

</script>

<template>

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

</template>
```

If you transmit `nb="3"` instead of `:nb="3"` in the App component, you need to write the `v-for` directive as `v-for="i in parseInt(nb)"` to retrieve the value of the `nb` attribute as an integer.

The `MyCounter` component increments a reactive variable `count` upon clicking the "count+1" button.

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

```
<script setup>

import { ref, defineProps } from "vue"

const count = ref(0);
const props = defineProps(["index"]);
const index = props.index || 1;

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

</script>

<template>

<h3>{{index}} - MyCounter Component</h3>

Reactive variable count : <b>{{count}}</b>
```

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

</template>
```

In Figure 3-6, you can see the display of the three counters.

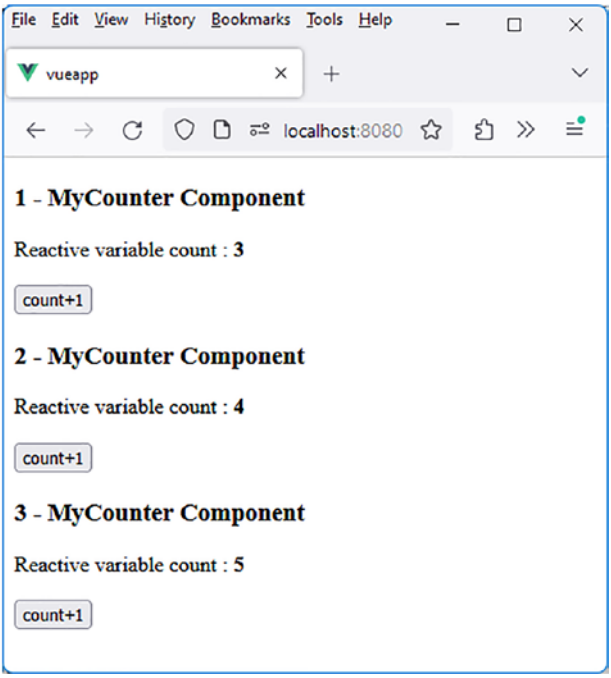

Figure 3-6. *Display of the three counters*

Each counter is independent of the others. Therefore, it is currently challenging to calculate the sum of the counters. Let's see how to proceed using events to communicate with the parent component.

Step 2: Communicating with the Parent Component

The sum of the counters is a variable that will be stored in the `MyCounters` component, which gathers all the counters. Each `MyCounter` component must notify the parent component, `MyCounters`, whenever an incrementation occurs on a counter. The `MyCounter` child component uses an event for this purpose, which the parent component, `MyCounters`, receives and processes.

First, let's see how the `MyCounters` parent component can receive and handle an event indicating that its overall total should be incremented.

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

```
<script setup>
import MyCounter from "./MyCounter.vue";
import { ref, defineProps } from "vue";

const props = defineProps(["nb"]);
const nb = props.nb || 1;

const total = ref(0);

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

</script>

<template>

<MyCounter v-for="i in nb" :key="i" :index="i" @
incr="increment"/>
<br/><hr/><br/>
Overall Total : <b>{{total}}</b>

</template>
```

We added a reactive variable called `total` in the `MyCounters` component, which will be incremented upon receiving the `incr` event. Note the syntax `@incr="increment"` instead of `@incr="increment()"`. Using `"increment()"` would assume that the `increment()` function doesn't take any parameters. However, in this case, we want to pass the `value` parameter, which represents the increment value. Now, let's see how the child component `MyCounter` can transmit a value when triggering an event (in this case, the `incr` event) to the parent component.

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

```
<script setup>

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

const count = ref(0);
const props = defineProps(["index"]);
const index = props.index || 1;

// We define the "incr" event to be used towards the parent
const emit = defineEmits(["incr"]);

const increment = () => {
  count.value++;
  emit("incr", 1);   // Sending the "incr" event with the value 1
}

</script>

<template>

<h3>{{index}} - MyCounter Component</h3>

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

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

</template>
```

The `defineEmits()` method allows us to define one or more events that can be used in the component. The events are described as strings in an array, in this case, `["incr"]`.

Since the child component is transmitting a value during the `incr` event, it is necessary to receive the `incr` event in the parent using the syntax `@incr="increment"` and not `@incr="increment()"`, which would not allow us to retrieve the transmitted value.

Let's verify that the total is updated correctly by clicking on each button:

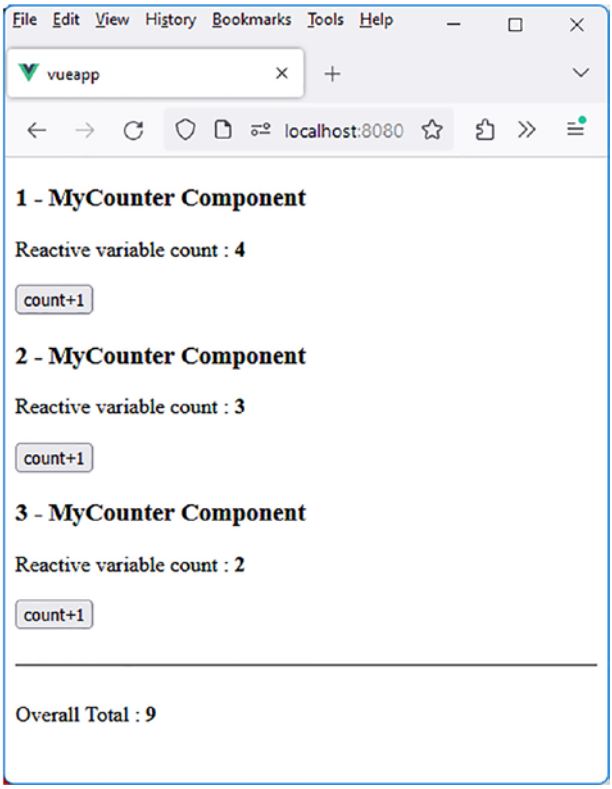

Figure 3-7. *Displaying the total of counters*

Using provide( ) and inject( ) for Communication Between Components

The mechanism we studied earlier, using events to communicate with the parent or props to communicate with children, is the basic mechanism offered by Vue.js. However, sometimes a different approach is desired, especially when a component needs to communicate not with its direct parent but with a parent further up in the hierarchy. This is also the case when you want to transmit props to components that are more distant than direct children.

Of course, this could be achieved by propagating events or passing attributes from child to child. However, intermediate components would then be involved in actions that don't concern them.

Vue.js provides an alternative system using the `provide()` and `inject()` methods:

- `provide(name, value)` is used to describe what a component makes available, under the name `name`, for its child components (direct or further down the hierarchy). This can include reactive variables, attributes (props), methods, etc.
- `inject(name)` allows a component in the descendant tree to specify what it wants to use, provided by a parent component through `provide(name)`.

Elements made available by `provide()` can be accessed by child components that request them using `inject()`.

To illustrate this, let's revisit the previous example. We modify the `App` component by integrating the reactive variable `total` and removing it from the `MyCounters` component. Now, each click on the "count+1" button in the `MyCounter` component needs to update the `total` variable located not in the `MyCounters` parent component but in the root `App` component.

Step 1: Using the provide( ) Method

We use the `provide(name, value)` method to make the `total` variable available for the descendants. This value becomes accessible to all components in the descendant tree.

The App component can be written as follows:

App Component (src/App.vue file)

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

import { ref, provide } from "vue";

const total = ref(0);

// The total variable is made available for the descendants
under the name "total"
provide("total", total);

</script>

<template>

<MyCounters :nb="3" />
<br/><hr/><br/>
Overall Total: <b>{{total}}</b>

</template>
```

The `"total"` functionality made available can now be used in the descendants' components of the App component, including the `MyCounter` component. The `inject()` method is used to retrieve these functionalities in the component that will use them.

Step 2: Use the inject( ) Method

Now let's see how the functionalities provided by `provide()` are used. The `inject()` method allows us to retrieve them in the component that will use them. In this case, we use them in the `MyCounter` component. The `MyCounters` component is not affected and remains as simple as possible, just displaying the requested `MyCounter` components.

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

```
<script setup>

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

const props = defineProps(["nb"]);
const nb = props.nb || 1;

</script>

<template>

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

</template>
```

The `MyCounter` component is modified to use the `"total"` functionality provided by the `App` component. The `"total"` functionality is a reactive variable made available in the `App` component to be updated in the `MyCounter` component.

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

```
<script setup>

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

const count = ref(0);
```

```
const props = defineProps(["index"]);
const index = props.index || 1;

// Access to the "total" functionality (which is a reactive
variable)
const total = inject("total");

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

</script>

<template>

<h3>{{index}} - MyCounter Component</h3>

Reactive variable count: <b>{{count}}</b>
<br/><br/>
<button @click="increment()">count+1</button>
<br/>

</template>
```

Let's verify that it works:

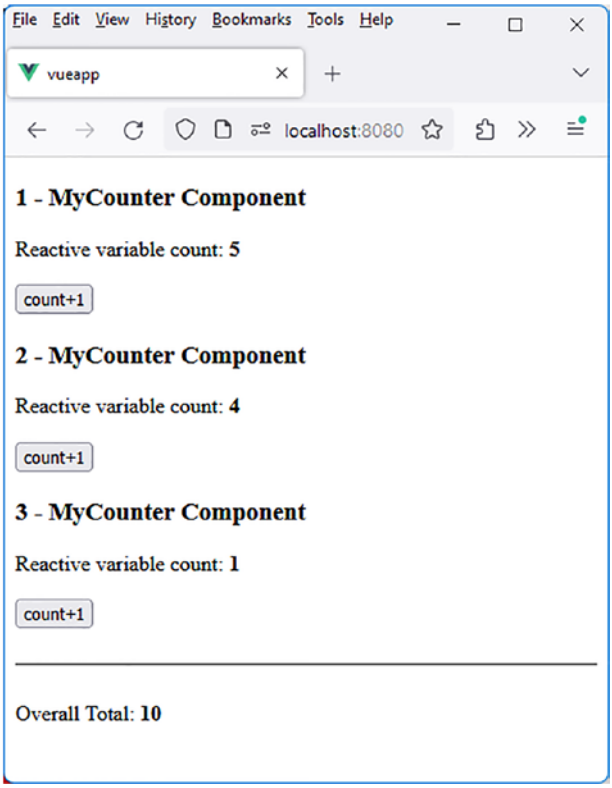

Figure 3-8. *Using provide() and inject() for component communication*

Conclusion

At this point, you've gained a solid understanding of the basics of Vue.js, including state and event management, directives, and components. However, modern web applications often don't operate in isolation; they frequently interact with remote servers to retrieve or send data.

In the next chapter, we'll take a crucial step forward by exploring how to make HTTP requests to communicate with a remote server. We'll put this knowledge into practice by building an application that retrieves and displays a list of all countries worldwide using a REST API. You'll also learn how to filter this data based on user input.