CHAPTER 1

Day 1: Mastering Components in Vue.js

Welcome to Chapter 1 of our journey to master Vue.js in just six days. During this first day, we will delve into the world of Vue.js, starting by understanding why it is so powerful and exploring its key concept, the Virtual DOM. We will then follow a step-by-step process to create our first Vue.js application, guiding you through the installation of Node.js and Vue CLI, as well as examining the default files generated in a Vue.js application.

We will also explore the structure of a Vue.js application, examining configuration files, directories, and essential components. Additionally, we will demonstrate how to break down an application into Vue.js components, adhering to naming conventions and creating our first component.

Finally, we will delve into fundamental aspects such as reactivity usage, defining methods and computed properties in a Vue.js component, and managing the lifecycle.

E. Sarrion, *Master Vue.js in 6 Days*, https://doi.org/10.1007/979-8-8688-0364-2_1

Why Use Vue.js

When it comes to choosing a JavaScript framework for modern and interactive web development, Vue.js stands out among the most popular and compelling choices. With its simple and reactive approach, Vue.js offers a smooth and enjoyable development experience, suitable for both beginners and experienced developers.

In this section, we will explore five essential reasons why Vue.js is a wise choice for your development projects. From its ease of learning to optimal performance, innovative Composition API, and dynamic ecosystem, let's quickly discover why Vue.js rightfully deserves its place among modern development frameworks.

We will, of course, have the opportunity to explore these different facets in depth later in the book.

1. **Ease of Learning and Implementation**

   Vue.js is renowned for its gentle learning curve. Its simple and declarative syntax, along with comprehensive and user-friendly documentation, makes it an excellent choice for both novice web developers and experienced professionals. Concepts such as components, directives, and composables are intuitive and easy to grasp, speeding up the learning and development process.

2. **Reactivity and Efficient Rendering**

   Reactivity is at the core of Vue.js. With its bidirectional data binding system and the ability to track changes in the application's state, Vue.js ensures reactive and efficient rendering. User interface updates are handled optimally, resulting in a smooth and performant user experience, even for complex applications.

3. **Composition API for Better Code Organization**

   With the introduction of the Composition API (starting from Vue.js version 3, as used in this book), Vue.js provides a more structured and modular way to organize code. Instead of dividing logic by options in components (as proposed in Vue.js version 2), the Composition API allows grouping logic by functionality, making the code more readable, maintainable, and scalable. It also facilitates logic sharing between components.

4. **Extensive Ecosystem of Libraries and Tools**

   Vue.js benefits from a dynamic ecosystem composed of many third-party libraries and complementary tools. Whether it's state management with Vuex, routing with Vue Router, or integration with other libraries and frameworks, Vue.js offers flexible options to meet various development needs.

5. **Optimal Performance and Lightweight Size**

   Vue.js is designed to be lightweight and fast. With its compact size and optimized performance, Vue.js applications load quickly in the browser, enhancing the overall user experience. Additionally, Vue.js allows server-side rendering (SSR) for even better performance in terms of SEO and initial loading time.
   Vue.js distinguishes itself through its simplicity, reactivity, flexibility, performance, and ecosystem, making it a wise choice for developing modern and dynamic web applications.

Virtual DOM

The Virtual DOM (Document Object Model) is a key concept in many modern JavaScript frameworks, including Vue.js. It is a lightweight and efficient abstraction of the real DOM, which represents the structure of a web page in the browser's memory. The goal of the Virtual DOM is to improve performance and the efficiency of DOM updates by minimizing direct manipulations, which can be time-consuming.

Let's now explore how the binding between the browser's DOM and Vue.js's Virtual DOM works.

Step 1: Virtual DOM Operation

Here's how the Virtual DOM operates in Vue.js:

1. **Virtual DOM Creation**: When a Vue.js component is created, it generates an internal Virtual DOM that reflects the current structure of the DOM. This Virtual DOM is a virtual and lightweight copy of the actual DOM tree.

2. **Initial Rendering**: At startup, the component generates the Virtual DOM using the data and templates defined in the Vue.js code.

3. **Change Detection**: When a component's data changes (due to user interaction, such as clicking a button), Vue.js uses a process called "reactivity" to detect data changes. This triggers a Virtual DOM update process.

4. **Comparison and Update**: Once data changes are detected, Vue.js compares the new state of the Virtual DOM with the previous state, identifying differences between the two versions.

5. **Patch Generation**: Vue.js generates a set of instructions (reconciliation process) describing the modifications to be made to the real DOM to reflect the changes. These instructions are created efficiently, minimizing the number of direct DOM manipulations.

6. **DOM Update**: Finally, Vue.js applies the reconciliation process to the real DOM in an optimized manner. Only the parts of the DOM that have changed are updated, significantly reducing performance compared to a full DOM update.

The major contribution of the Virtual DOM lies in the efficiency and speed of updates. Instead of directly manipulating the DOM with each data change, Vue.js uses the Virtual DOM to minimize actual DOM modifications, resulting in improved performance and a better user experience. This allows developers to focus on application logic rather than complex DOM manipulations. Vue.js's Virtual DOM is therefore a crucial technique that optimizes browser performance by intelligently detecting changes and efficiently updating the DOM, enhancing reactivity and the user experience.

Step 2: Concrete Example

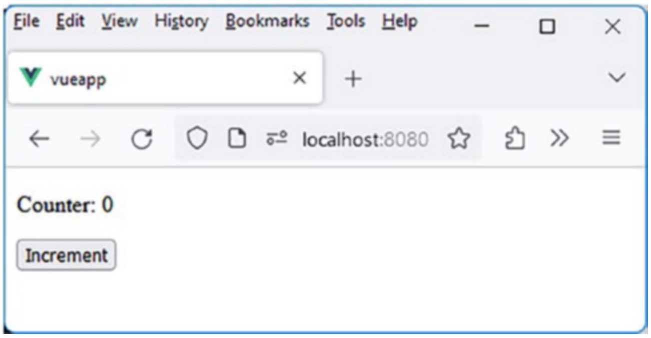
Here is a concrete example to illustrate how the Virtual DOM works in Vue.js through a simple case: incrementing a counter by clicking a button.

Suppose we have a Vue.js component called `MyCounter` that displays a counter and a button to increment it. Here is how it could be implemented (this code example will be explained later in this chapter; the key here is to explain how the DOM is updated with each click on the "Increment" button). The `MyCounter` component is associated with a `MyCounter.vue` file described as follows:

File MyCounter.vue

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

const count = ref(0);
const increment = () => count.value++

</script>

<template>
  <div>
    <p>Counter: {{ count }}</p>
    <button @click="increment()">Increment</button>
  </div>
</template>
```

This component is displayed in a browser as follows:

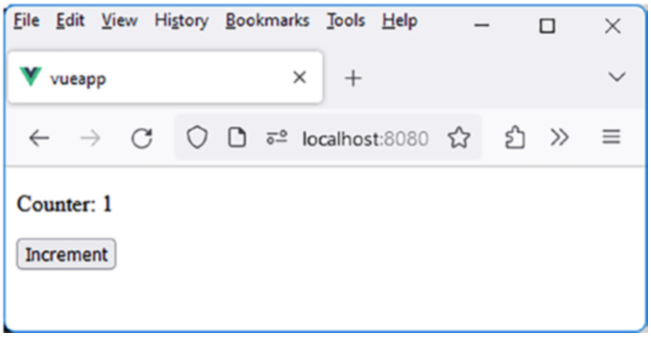


Counter: 0

Increment

Figure 1-1. *First display of the Vue.js application*

Upon clicking the "Increment" button once, the counter value increments from 0 to 1:



Counter: 1

Figure 1-2. *Incrementing the counter from 0 to 1*

Let's examine how Vue.js's Virtual DOM handles updates when clicking the button to increment the counter:

1. Upon startup, the `MyCounter` component is mounted, and the Virtual DOM is created based on the template and initial data.
2. When clicking the "Increment" button, the `increment()` method is called. This changes the value of the `count` variable in the component.
3. Vue.js detects the data change through its reactivity system.
4. The Virtual DOM is updated to reflect the new value of `count`. Vue.js compares the old and new Virtual DOM to determine differences.
5. Vue.js generates a set of instructions describing the modification to be made to the real DOM. In this case, the instructions simply indicate that the counter text should be updated.

6. The real DOM is updated with the executed instructions. Only the part of the DOM corresponding to the counter is modified, minimizing direct DOM manipulations.

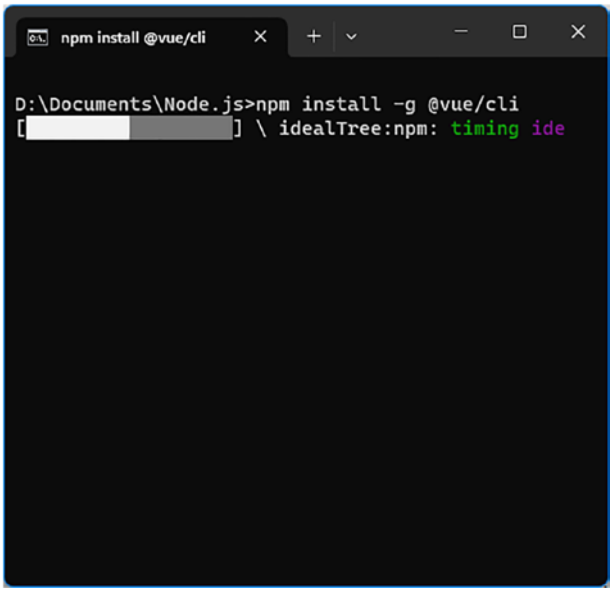
This example illustrates how Vue.js's Virtual DOM optimizes DOM updates by detecting changes, then generating efficient instructions to selectively update the real DOM. This provides a reactive user experience while optimizing performance.

Creating a First Vue.js Application

Creating a Vue.js application requires installing the Vue CLI utility, downloadable after installing the npm utility from the Node.js server.

Step 1: Installing Node.js and Vue CLI

Ensure that you have Node.js installed on your system, which you can download from the official Node.js website (`https://nodejs.org/`). Next, install Vue CLI using the `npm install -g @vue/cli` command in your terminal:



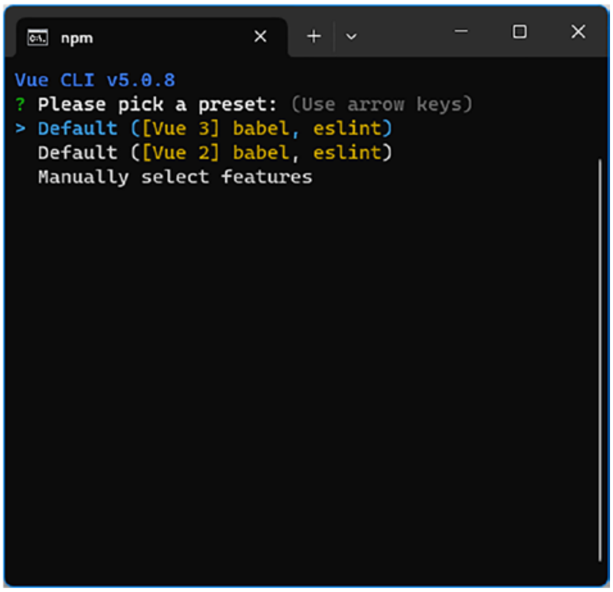
```
D:\Documents\Node.js>npm install -g @vue/cli
[                  ] \ idealTree:npm: timing ide
```

Figure 1-3. *Vue CLI installation*

Once Vue CLI is installed, you can use the `vue create` command to create the Vue.js application.

Step 2: Creating the Vue.js Application

After installing Vue CLI, you can create a new Vue.js project by running the command `vue create vueapp`. This will create the `vueapp` application in the newly created `vueapp` directory:

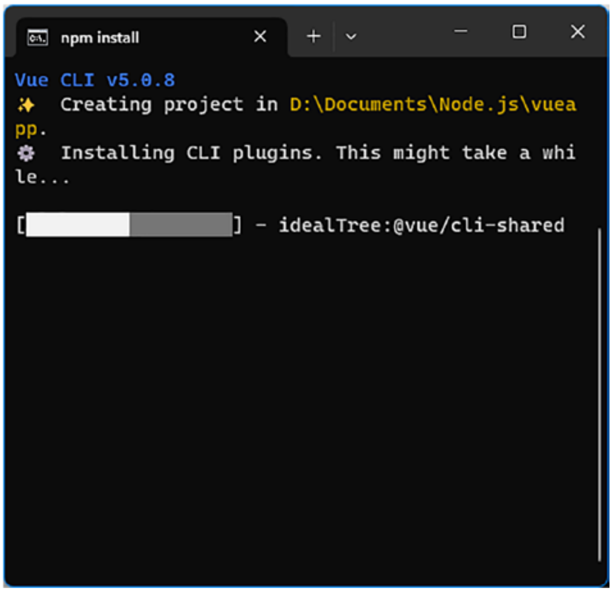


```
Vue CLI v5.0.8
? Please pick a preset: (Use arrow keys)
> Default ([Vue 3] babel, eslint)
  Default ([Vue 2] babel, eslint)
  Manually select features
```

Figure 1-4. *Creating the Vue.js application with Vue CLI*

The system prompts for the desired version of Vue.js. We retain the default selection of the current version 3 by pressing the Enter key on the keyboard.

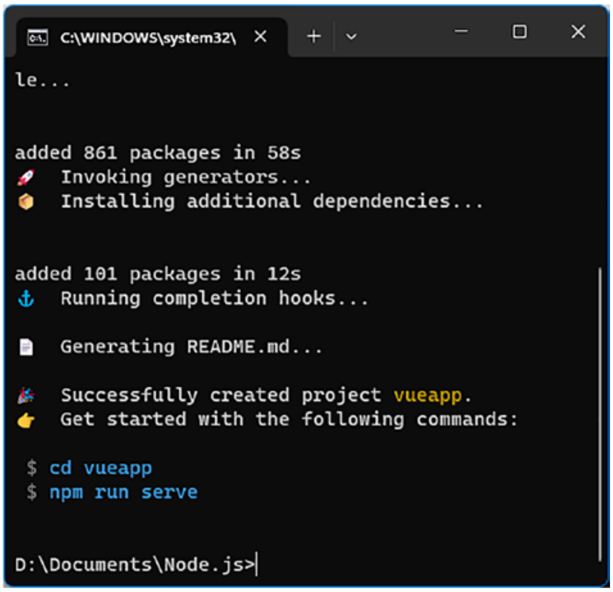
The Vue.js application named vueapp begins to be created:



```
Vue CLI v5.0.8
✨  Creating project in D:\Documents\Node.js\vuea
pp.
⚙  Installing CLI plugins. This might take a whi
le...

[          ] - idealTree:@vue/cli-shared
```

Figure 1-5. *Creating the Vue.js application in progress*

Once the application creation process is complete, it will be displayed on the screen:



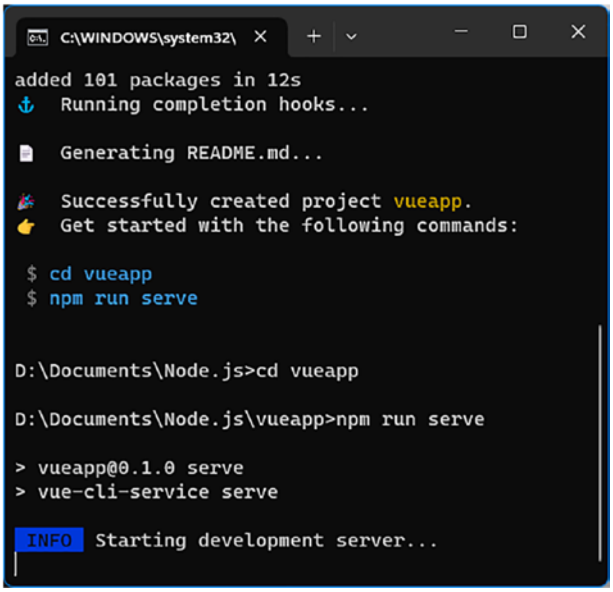
```
le...


added 861 packages in 58s
🚀  Invoking generators...
📦  Installing additional dependencies...


added 101 packages in 12s
⚓  Running completion hooks...

📄  Generating README.md...

🎉  Successfully created project vueapp.
👉  Get started with the following commands:

 $ cd vueapp
 $ npm run serve


D:\Documents\Node.js>
```

Figure 1-6. *Completion of Vue.js application creation*

Once the Vue.js application is created, the next step is to start it.

Step 3: Launching the Vue.js Application

To start the previously created Vue.js application, simply type the two commands in the terminal window as indicated: `cd vueapp`, then `npm run serve`.

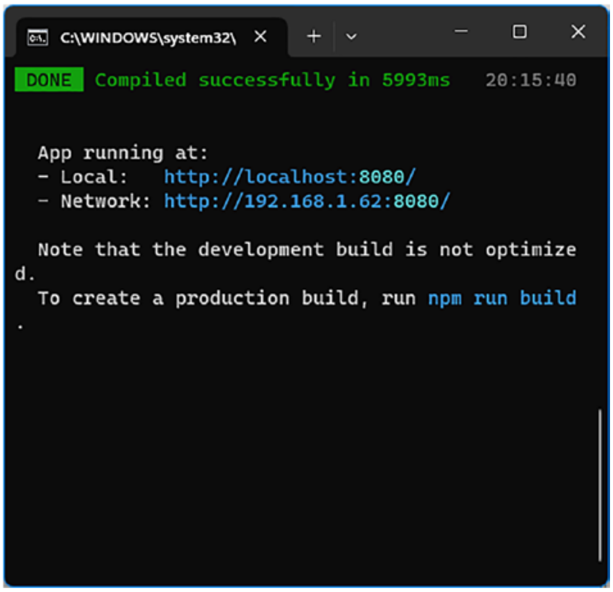


```
added 101 packages in 12s
⚓  Running completion hooks...

📄  Generating README.md...

🎉  Successfully created project vueapp.
👉  Get started with the following commands:

 $ cd vueapp
 $ npm run serve


D:\Documents\Node.js>cd vueapp

D:\Documents\Node.js\vueapp>npm run serve

> vueapp@0.1.0 serve
> vue-cli-service serve

 INFO  Starting development server...
```

Figure 1-7. *Launching the Vue.js application*

The `npm run serve` command starts a Node.js server on which the Vue.js application will run.

Once the Node.js server is launched, the terminal screen becomes the following:

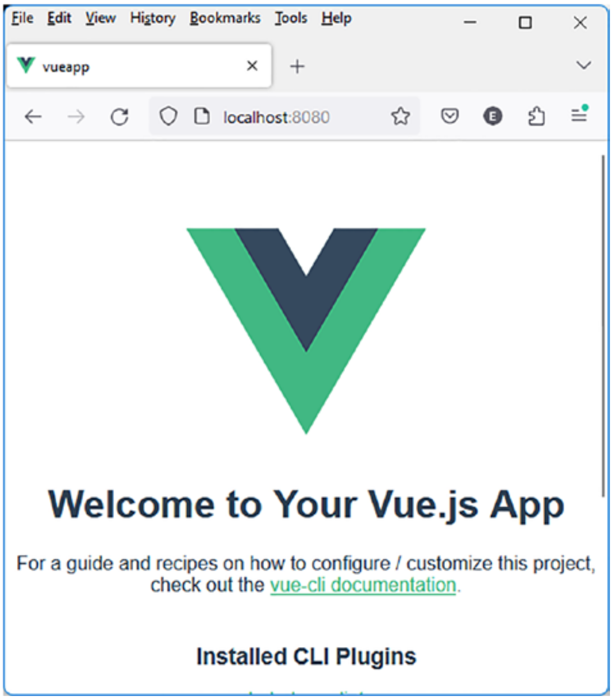


```
 DONE  Compiled successfully in 5993ms    20:15:40


  App running at:
  - Local:   http://localhost:8080/
  - Network: http://192.168.1.62:8080/

  Note that the development build is not optimize
d.
  To create a production build, run npm run build
.
```

Figure 1-8. *Completion of the server launch process with the Vue.js application*

The terminal window indicates that the Vue.js application is accessible at the URL `http://localhost:8080`.

Let's enter this URL in a browser:



Welcome to Your Vue.js App

For a guide and recipes on how to configure / customize this project, check out the vue-cli documentation.

Installed CLI Plugins

Figure 1-9. *Default Vue.js application created*

We now have access to the previously created Vue.js application.

Analyzing the Files Created by Default in the Vue.js Application

The `vue create vueapp` command created a vueapp directory containing configuration files and directories containing the source code for our Vue.js application.

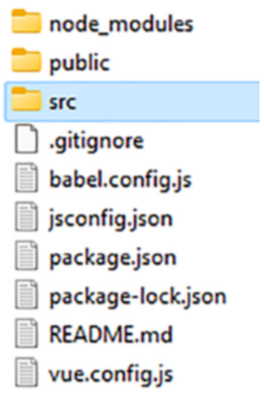

node_modules
public
src
.gitignore
babel.config.js
jsconfig.json
package.json
package-lock.json
README.md
vue.config.js

Figure 1-10. *Contents of the "vueapp" directory in the Vue.js application*

We can see that the Vue.js application directory primarily contains configuration files and three main directories (`node_modules`, `public`, and `src`). Let's now explain their role and contents.

Step 1: Configuration Files (with .js and .json Extensions)

The configuration files are directly attached to the root of the application. They serve, among other things, to enable the execution of the Vue.js application on a Node.js server. For example, here is the content of the `package.json` file, which is traditionally used to configure an application to run on a Node.js server:

File package.json

```
{
  "name": "vueapp",
  "version": "0.1.0",
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": {
    "core-js": "^3.8.3",
    "vue": "^3.2.13"
  },
  "devDependencies": {
    "@babel/core": "^7.12.16",
    "@babel/eslint-parser": "^7.12.16",
    "@vue/cli-plugin-babel": "~5.0.0",
    "@vue/cli-plugin-eslint": "~5.0.0",
    "@vue/cli-service": "~5.0.0",
    "eslint": "^7.32.0",
    "eslint-plugin-vue": "^8.0.3"
  },
  "eslintConfig": {
    "root": true,
    "env": {
      "node": true
    },
    "extends": [
      "plugin:vue/vue3-essential",
      "eslint:recommended"
    ],
    "parserOptions": {
      "parser": "@babel/eslint-parser"
    },
    "rules": {}
  },
  "browserslist": [
    "> 1%",
    "last 2 versions",
    "not dead",
    "not ie 11"
  ]
}
```

In this file, you'll find the list of dependencies that our application needs to run, along with their respective versions.

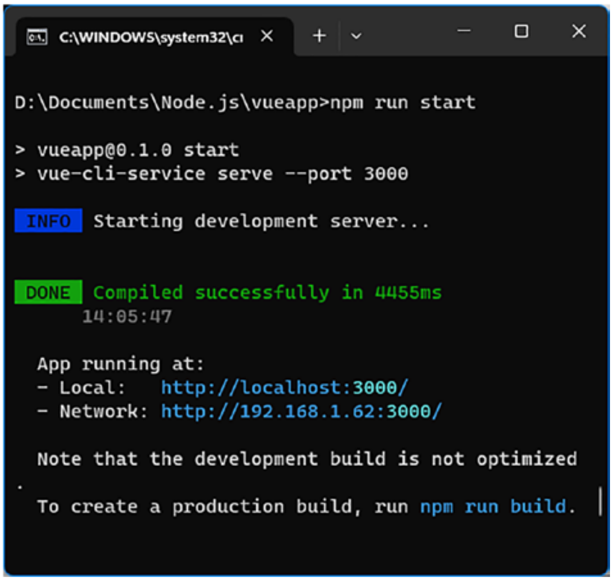
The "scripts" key allows the execution of commands such as `npm run serve`, which launches the development server on port 8080.

Additional scripts can be added. For example, let's insert a new script "start" in the "scripts" section that starts the Vue.js application on port 3000 instead of the default port 8080.

Adding the "start" script (package.json file)

```
"scripts": {
    "serve": "vue-cli-service serve",
    "start": "vue-cli-service serve --port 3000",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
},
```

The command `npm run start` starts the server on port 3000.



```
D:\Documents\Node.js\vueapp>npm run start

> vueapp@0.1.0 start
> vue-cli-service serve --port 3000

 INFO  Starting development server...


 DONE  Compiled successfully in 4455ms
      14:05:47

  App running at:
  - Local:   http://localhost:3000/
  - Network: http://192.168.1.62:3000/

  Note that the development build is not optimized
.
  To create a production build, run npm run build.
```

Figure 1-11. *Execution of the "start" script*

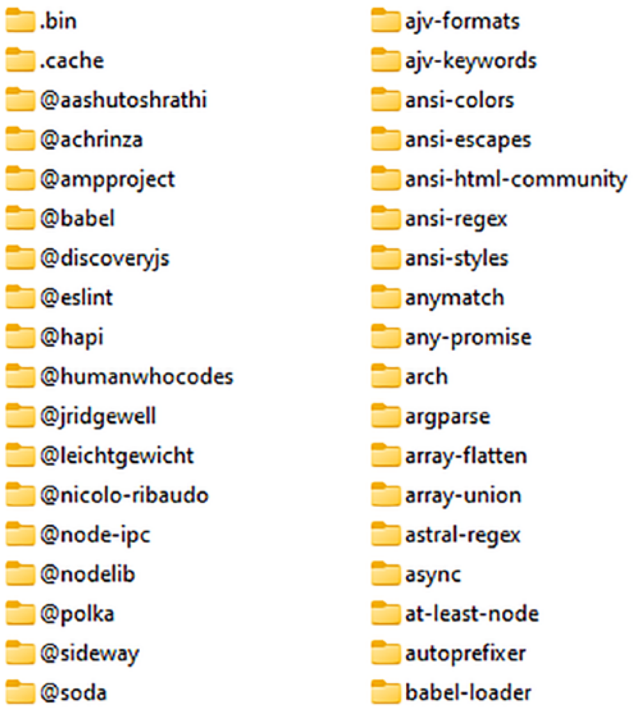
The Vue.js application is now accessible at the URL `http://localhost:3000`, thanks to the server launched on port 3000.

We briefly examined the configuration files of the Vue.js application. Let's now look at the application directories, starting with the `node_modules` directory.

Step 2: node_modules Directory

The `node_modules` directory contains external dependencies necessary for the proper functioning of the Vue.js application (those specified in the `package.json` file), as well as other libraries and modules that may be added to the project later.

Here is a partial content of this directory shortly after creating the application.



```
.bin                    ajv-formats
.cache                  ajv-keywords
@aashutoshrathi         ansi-colors
@achrinza               ansi-escapes
@ampproject             ansi-html-community
@babel                  ansi-regex
@discoveryjs            ansi-styles
@eslint                 anymatch
@hapi                   any-promise
@humanwhocodes          arch
@jridgewell             argparse
@leichtgewicht          array-flatten
@nicolo-ribaudo         array-union
@node-ipc               astral-regex
@nodelib                async
@polka                  at-least-node
@sideway                autoprefixer
@soda                   babel-loader
```

Figure 1-12. *Partial content of the node_modules directory*

Step 3: public Directory

The `public` directory contains the static files of the application. This `public` directory typically includes the following two files:

- The `favicon.png` file, which specifies the application's icon to be displayed in the browser tabs.
- The `index.html` file, which is the starting file of the Vue.js application.

Let's see the content of the `index.html` file. This content is rarely modified as it incorporates a crucial `<div>` HTML element for the functioning of the Vue.js application.

File public/index.html

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-
    scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.
      title %> doesn't work properly without JavaScript
      enabled. Please enable it to continue.</strong>
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

The `index.html` file contains a single `<div>` element, which has been assigned the default identifier `"app"`. This convention allows us to insert the Vue.js components of our application into this `<div>` element, visualizing the Vue.js application in the browser. We will see here how this is achieved.

Now let's examine the content of the `src` directory, which will help us understand the purpose of the previous `<div>` element.

Step 4: src Directory

The `src` directory of the application is the most widely used and modified when building our Vue.js applications. It contains the source code for our Vue.js components as well as static files such as images or CSS style files.

It consists of the following files and directories:

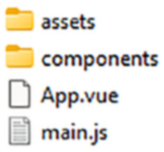


Figure 1-13. *Content of the src directory*

Let's examine in detail each file and directory listed in the `src` directory. We'll start with the `main.js` file.

Step 5: src/main.js

The `main.js` file is crucial for starting a Vue.js application. Let's see its content:

File src/main.js

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

Here's an explanation of the previous code:

1. The first statement `import { createApp } from 'vue'` imports the `createApp()` function from the "vue" package. The "vue" package is located in the `node_modules` directory of the application. The `createApp()` function will be used later to create a Vue.js application instance.

2. The second statement `import App from './App.vue'` imports the `App` component from the `App.vue` file. The `App` component will be the root component of the application.

3. Finally, the statement `createApp(App).mount('#app')` first calls the `createApp(App)` function to create a Vue.js application instance using the previously imported `App` component. Then the `mount('#app')` method is called on this instance. This mounts the application onto the DOM element with the id `"app"`. This `"app"` element was present in the `index.html` file seen earlier. This element serves as the main container in which the Vue.js application will be displayed.

In summary, the code in the main.js file creates an instance of the Vue.js application using the `App` component as the root component and then mounts this instance on an element with the id `"app"` in the DOM. This effectively initializes the Vue.js application and makes it ready to be displayed and used in the browser.

Step 6: src/App.vue

We explained previously that the App.vue file is associated with the App component, which will be displayed in the HTML page. The App component describes the structure of our Vue.js application using the syntax provided by the Vue.js framework. Here is the content of the App.vue file:

App component (src/App.vue file)

```
<template>
  <img alt="Vue logo" src="./assets/logo.png">
  <HelloWorld msg="Welcome to Your Vue.js App"/>
</template>

<script>
import HelloWorld from './components/HelloWorld.vue'

export default {
  name: 'App',
  components: {
    HelloWorld
  }
}
</script>

<style>
#app {
  font-family: Avenir, Helvetica, Arial, sans-serif;
  -webkit-font-smoothing: antialiased;
  -moz-osx-font-smoothing: grayscale;
  text-align: center;
  color: #2c3e50;
  margin-top: 60px;
}
</style>
```

The `App.vue` file is a Vue.js component that defines the structure (`<template>` section), logic (`<script>` section), and style (`<style>` section) of the main part of the application. Here is a concise explanation of the content:

1. **<template> section**
   - The content between the `<template>` tags defines the HTML structure of the component.
   - It includes an `<img>` tag displaying a Vue.js logo and a `<HelloWorld>` component with a "`msg`" property.

2. **<script> section**
   - The `<script>` section contains the JavaScript logic of the component.
   - It imports the `HelloWorld` component from the `HelloWorld.vue` file.
   - The exported object (via the `export default` statement) defines the name of the component (here, "`App`") and the components it uses (in this case, "`HelloWorld`").

3. **<style> section**
   - The `<style>` section contains CSS style rules for the component.
   - The element with the id "`app`" is styled with various CSS properties for formatting the application. Recall that the element with the id "`app`" is the one on which the Vue.js application is "mounted."

In summary, the `App.vue` file combines HTML structure, JavaScript logic, and CSS styles to define the main component of the application. It imports the `HelloWorld` component and displays a Vue.js logo and a message inside this component. The style is applied to the main container with the id `"app"`. This file forms the visual and functional basis of the application.

This `App.vue` file (associated with the `App` component) will need to be modified later to display our own Vue.js application.

We have described the two main files in the `src` directory, namely, the `main.js` file and the `App.vue` file. We have seen that the `App` component uses another component named `HelloWorld`. This new component is located in the `components` directory of the application. Let's describe the contents of the `components` directory, specifically the `HelloWorld.vue` file associated with the `HelloWorld` component.

Step 7: src/components Directory

The `src/components` directory contains files describing the internal components of our Vue.js application. Subdirectories can be added if a more structured organization is desired.

Let's examine the file currently present in this directory. It is the `HelloWorld.vue` file associated with the `HelloWorld` component.

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

```
<template>
  <div class="hello">
    <h1>{{ msg }}</h1>
    <p>
      For a guide and recipes on how to configure / customize
      this project,<br>
      check out the
      <a href="https://cli.vuejs.org" target="_blank"
      rel="noopener">vue-cli documentation</a>.
```

```
</p>
<h3>Installed CLI Plugins</h3>
<ul>
  li><a href="https://github.com/vuejs/vue-cli/tree/
  dev/packages/%40vue/cli-plugin-babel" target="_blank"
  rel="noopener">babel</a></li>
  <li><a href="https://github.com/vuejs/vue-cli/tree/
  dev/packages/%40vue/cli-plugin-eslint" target="_blank"
  rel="noopener">eslint</a></li>
</ul>
<h3>Essential Links</h3>
<ul>
  <li><a href="https://vuejs.org" target="_blank"
  rel="noopener">Core Docs</a></li>
  <li><a href="https://forum.vuejs.org" target="_blank"
  rel="noopener">Forum</a></li>
  <li><a href="https://chat.vuejs.org" target="_blank"
  rel="noopener">CommunityChat</a></li>
  <li><a href="https://twitter.com/vuejs" target="_blank"
  rel="noopener">Twitter</a></li>
  <li><a href="https://news.vuejs.org" target="_blank"
  rel="noopener">News</a></li>
</ul>
<h3>Ecosystem</h3>
<ul>
  <li><a href="https://router.vuejs.org" target="_blank"
  rel="noopener">vue-router</a></li>
  <li><a href="https://vuex.vuejs.org" target="_blank"
  rel="noopener">vuex</a></li>
  <li><a href="https://github.com/vuejs/vue-devtools#vue-
  devtools" target="_blank" rel="noopener">
  vue-devtools</a></li>
```

```
      <li><a href="https://vue-loader.vuejs.org"
      target="_blank" rel="noopener">vue-loader</a></li>
      <li><a href="https://github.com/vuejs/awesome-vue"
      target="_blank" rel="noopener">awesome-vue</a></li>
    </ul>
  </div>
</template>

<script>
export default {
  name: 'HelloWorld',
  props: {
    msg: String
  }
}
</script>

<!-- Add "scoped" attribute to limit CSS to this component
only -->
<style scoped>
h3 {
  margin: 40px 0 0;
}
ul {
  list-style-type: none;
  padding: 0;
}
li {
  display: inline-block;
  margin: 0 10px;
}
```

```
a {
  color: #42b983;
}
</style>
```

We find the three sections of a Vue.js component, as we discussed in the previous section when describing the `App` component:

- The `<template>` section defines the HTML structure of the component.
- The `<script>` section describes the JavaScript logic of the component.
- The `<style>` section describes the styles used in the component. The `scoped` attribute indicated here localizes the styles defined only for the `HelloWorld` component.

We will have the opportunity in the following pages to explain in detail the possible content of a Vue.js component. We already know that it contains the three sections mentioned earlier.

Step 8: src/assets Directory

The `src/assets` directory is used to store static files of the application, such as images or globally applied CSS files for the entire Vue.js application. In our case, it contains only the `logo.png` file corresponding to the image displayed in the Vue.js application.

We have now completed the quick description of each of the files and directories created in the default Vue.js application `"vueapp"` using the `vue create vueapp` command. We have seen that the files describing our application are mainly located in the `src` directory and its subdirectories.

We will now explain how to design a Vue.js application by breaking it down into different components according to what we want to achieve.

Decomposition of a Vue.js Application into Components

Breaking down an application into components, as Vue.js suggests, is a common approach in software development. This methodology promotes the creation of a modular structure that simplifies code maintenance, reuse, and clarity.

Here is a simple example of breaking down an application into components.

Let's imagine that we want to design an application that displays a list of tasks to be done, each accompanied by a check box to indicate its status (completed or not). To do this, we would start by segmenting the application into two main components: a parent component called "TaskList" and a child component named "TaskItem."

The parent component "TaskList" would be responsible for managing the complete list of tasks to be done. Its responsibility would involve retrieving the list data from a data source (such as an API or a database), performing sorting and filtering operations, and then passing this data to the child component "TaskItem."

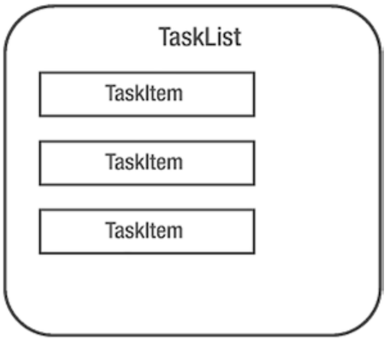


Figure 1-14. *TaskList and TaskItem components*

The child component “TaskItem,” on the other hand, would be responsible for displaying an individual task. It would show the task’s name and a check box to indicate its status (completed or not). This component would be versatile and display each task whenever it is called by the parent component “TaskList.”

Next, we could further fragment the “TaskList” component into subcomponents. For example, “TaskListHeader” could handle the header of the list, “TaskListFooter” the bottom of the list, and “TaskListFilter” the management of filters within the list. Each subcomponent would be responsible for a specific part of the task list, contributing to code clarity and reusability.

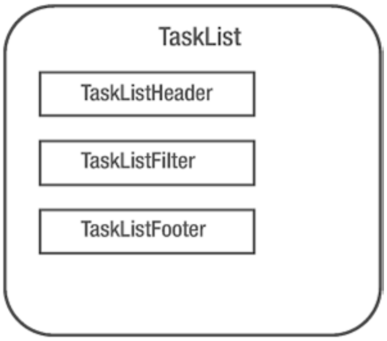


Figure 1-15. *TaskList, TaskListHeader, TaskListFilter, and TaskListFooter components*

Finally, we could also subdivide the “TaskItem” component into additional subcomponents. For example, “TaskName” could focus on displaying the task’s name, “TaskCheckbox” on showing the check box, and “TaskDueDate” on displaying the task’s due date. Each of these subcomponents would play a specific role in displaying an individual task, bringing more modularity to the design and simplifying code management.

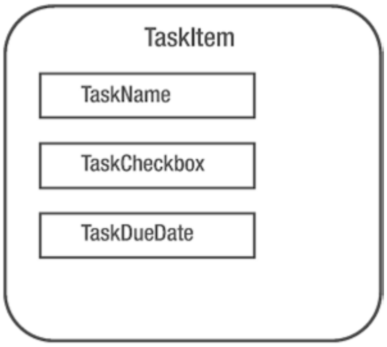


Figure 1-16. *TaskItem, TaskName, TaskCheckbox, and TaskDueDate components*

By decomposing our application into components, we've established a modular architecture, simplifying code readability, update management, and the potential for reuse. Moreover, since each component is responsible for a well-defined task, the debugging process is greatly facilitated, as each component can be addressed independently.

Naming Conventions for Vue.js Components

A Vue.js component name is written in PascalCase, with an initial capital letter for each word in the component's name. To avoid assigning a component name that might also be associated with an HTML element, the Vue.js component name must be defined with a minimum of two words. HTML elements are all defined with a single word, such as `<img>`, `<p>`, `<span>`, etc. Therefore, if Vue.js component names are defined with a minimum of two words, there is no risk of confusion with HTML tag names.

Thus, you can create the `MyCounter` component in Vue.js, but you cannot create the `Counter` component because it consists of a single word, while `MyCounter` consists of two words.

The only exception to this rule is the `App` component, which is the only one written with a single word.

Once the `MyCounter` component is created, it can be used in the `<template>` sections as `<MyCounter>` or `<my-counter>`. Both forms of writing are equivalent in Vue.js, although the `<MyCounter>` syntax is recommended.

Creating a First Component with Vue.js

A component is defined using a `.vue` extension file and must be in the form of components like `App` and `HelloWorld`, defined in the `App.vue` and `HelloWorld.vue` files described earlier. It will thus include the `<script>`, `<template>`, and `<style>` sections, which will be filled in according to the component's needs. If one of the sections has no elements, it can be omitted. This is often the case for the `<style>` section.

Here is an example of the `MyCounter.vue` component, which simply displays the text "MyCounter Component" in an `<h1>` tag. The `MyCounter.vue` file will be located in the `components` directory of the `src` directory in the application.

File src/components/MyCounter.vue

```
<script>

</script>

<template>

<h1> MyCounter Component </h1>

</template>

<style scoped>

</style>
```

The `<script>` and `<style>` sections are not necessary but are present as they may contain added instructions later. The `MyCounter` component needs to be inserted into the `App` component to be displayed within it. Indeed, the `App` component currently uses the `HelloWorld` component seen previously. Let's modify the `App` component to incorporate the `MyCounter` component.

The `App` component can be written in two ways, depending on the Vue.js syntax one wishes to use:

- Options API syntax (available from Vue.js version 2)
- Composition API syntax (available from Vue.js version 3)

Using the Options API syntax, the `App.vue` file becomes the following:

Using the Options API syntax (file src/App.vue)

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

export default {
  components : {
    MyCounter : MyCounter
    // As the key and the value are identical, we can also
    write more simply:
    // MyCounter
  }
}

</script>

<template>
  <MyCounter />
</template>

<style scoped>
</style>
```

The MyCounter.vue file is imported into the App component using the import statement in the <script> section of the component. Then, the MyCounter component is displayed in the <template> section using the <MyCounter /> tag.

In the <script> section, we added the export default statement of an object with the components option describing the components used in the App component. This is the traditional way of using the syntax with options, proposed in version 2 of Vue.js. It is called the Options API syntax.

Another syntax is possible, introduced from version 3 of Vue.js. It is called the Composition API syntax.

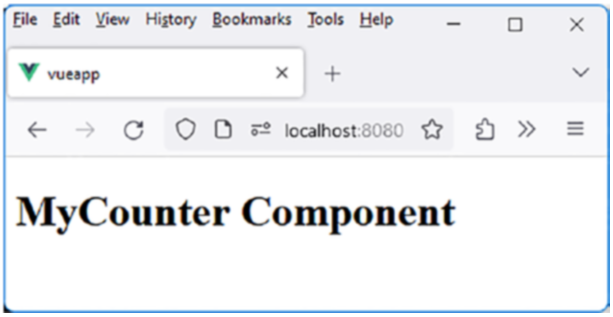
Let's use the Composition API syntax to write the App component in the App.vue file:

Using the Composition API syntax (file src/App.vue)

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

<template>
  <MyCounter />
</template>

<style scoped>
</style>
```

Here, note the setup attribute added to the <script> element. This attribute in the <script> tag allows importing the MyCounter.vue file without having to explicitly declare the MyCounter component within the App component. This is a convenience added in Vue.js version 3, known as the Composition API syntax.

Let's verify that both versions of the App component work by displaying the URL http://localhost:8080.



MyCounter Component

Figure 1-17. *MyCounter Component*

Which syntax, Options API or Composition API, should we use to write a Vue.js component? We will use the Composition API syntax here, as it is newer and simpler to use for writing components.

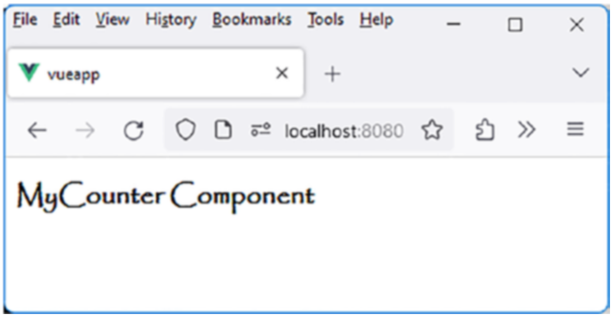
Defining Styles in a Vue.js Component

Let's modify the style of the `<h1>` element in the `MyCounter` component. To do this, simply add a CSS style in the `<style>` section of the `MyCounter` component.

Style of the <h1> element (file src/components/MyCounter.vue)

```
<script setup>

</script>

<template>
<h1> MyCounter Component </h1>

</template>

<style scoped>

h1 {
  font-family:papyrus;
  font-size:20px;
}

</style>
```

The MyCounter component now displays with a new font style.



MyCounter Component

Figure 1-18. *Component MyCounter with a new font style*

Let's now explain the use of the scoped attribute in the <style> tag.

Using the Scoped Attribute in the <style> Tag

The scoped attribute in a <style> tag in Vue.js is used to restrict the scope of CSS styles to a specific component. This means that styles defined within a <style scoped> tag will only apply to elements within the current Vue component and will not propagate to elements in other components.

Here is an example to illustrate the difference between using scoped and not using it:

Without the scoped attribute (file src/components/MyCounter.vue)

```
<template>
  <div>
    <p class="red-text">MyCounter Component</p>
  </div>
```

```
</template>

<style>
.red-text {
  color: red;
}
</style>
```

In this example, the CSS class `.red-text` is defined in the global style of the component. It could potentially impact other components if the class is used elsewhere in the application.

With the scoped attribute (file src/components/MyCounter.vue)

```
<template>
  <div>
    <p class="red-text">MyCounter Component</p>
  </div>
</template>

<style scoped>
.red-text {
  color: red;
}
</style>
```

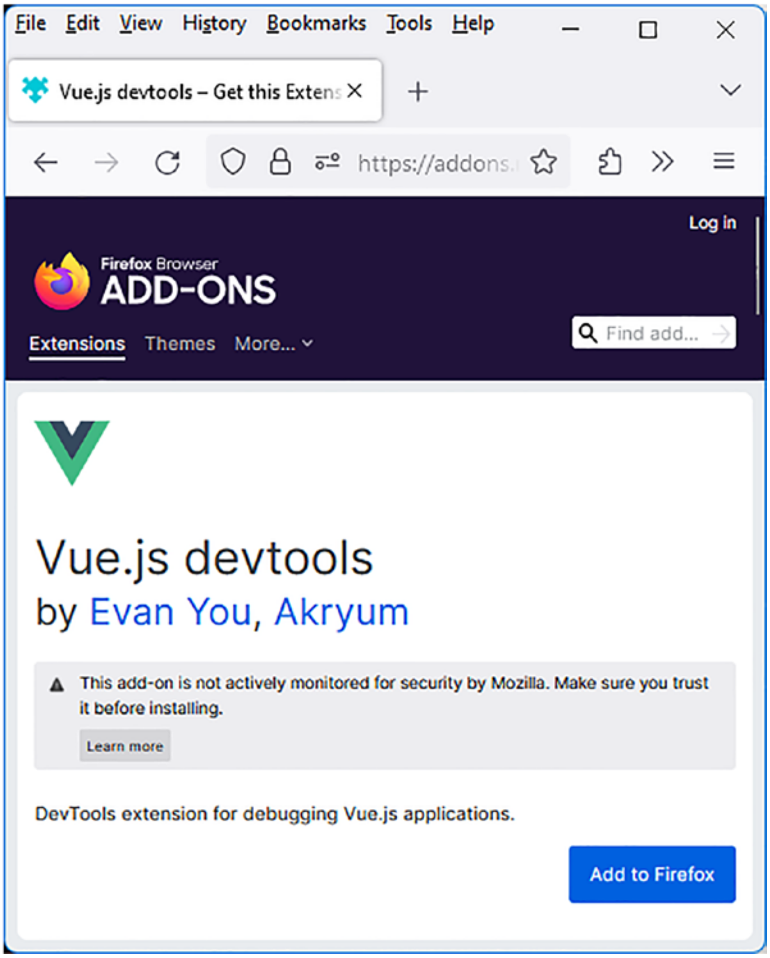
In this example, the `.red-text` class is defined within a `<style scoped>` tag, indicating that it will only apply to elements within the current component. The styles defined here will have no impact on other components.

The `scoped` attribute is particularly useful for avoiding style conflicts between components and maintaining style isolation. It facilitates the creation of reusable components and simplifies style management in a Vue.js application. Each component can define its styles without concern for styles defined in other components, contributing to better code organization and maintainability.

Using Vue.js DevTools

Vue.js DevTools is a browser extension designed for debugging applications built with Vue.js. Simply search for "Vue.js DevTools" on Google to download it for our browser.

For instance, here is the link for Firefox:



This add-on is not actively monitored for security by Mozilla. Make sure you trust it before installing.

DevTools extension for debugging Vue.js applications.

Figure 1-19. *Installation of the Vue DevTools extension on Firefox*

Click the "Add to Firefox" button. Next, display the console by pressing the F12 key on the keyboard. Then, select the Vue menu by clicking >>.

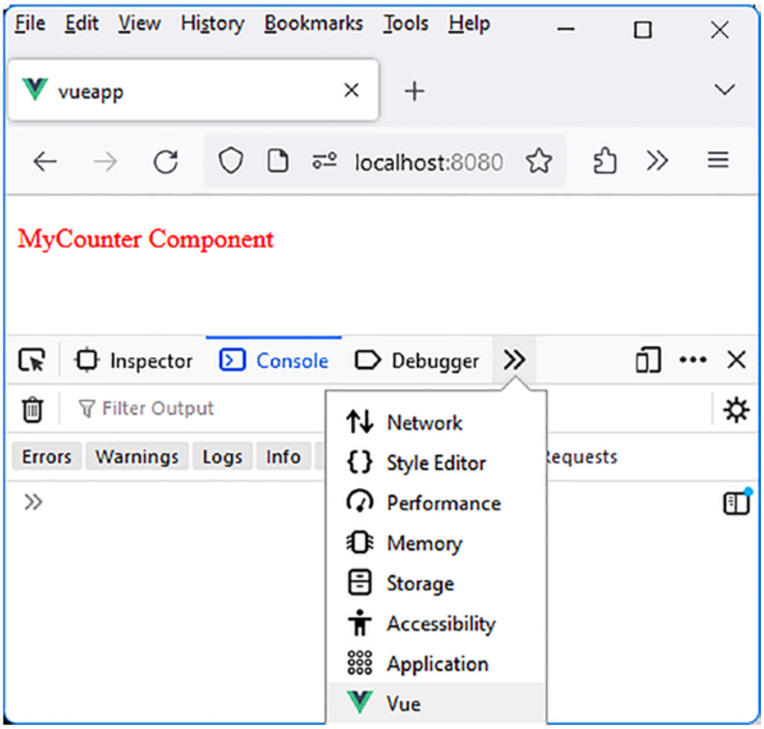
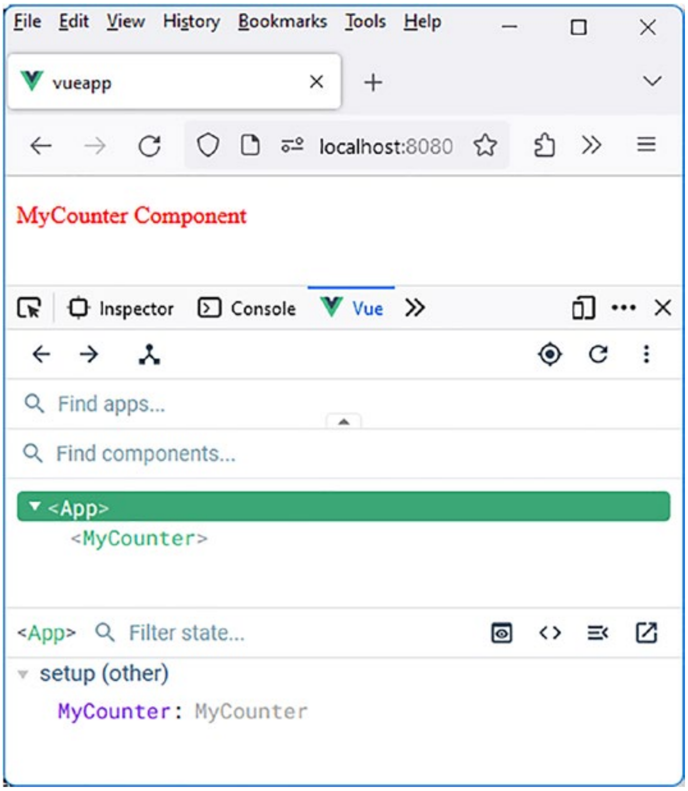


Figure 1-20. *Selection of the Vue DevTools extension*

The console window refreshes, displaying the components used in the application:



MyCounter Component

```
▼ <App>
    <MyCounter>

<App>
▼ setup (other)
    MyCounter: MyCounter
```

Figure 1-21. *Components of the application displayed in Vue DevTools*

Using Vue.js Reactivity with the ref( ) Method

Reactivity is a crucial concept when developing applications with Vue.js. This concept allows for automatic updating of the HTML page display when a so-called reactive variable is modified in the program.

If a reactive variable is modified by the program, this modification will be visible wherever the reactive variable is used in the component's display.

This concept enables the separation of program variables and where they are displayed, as Vue.js takes care of managing the display modification.

A reactive variable is defined within a component and is associated with the component in which it is defined. We create a reactive variable using the `ref()` method defined in Vue.js. Let's demonstrate how to define and modify a reactive variable with the `ref()` method. We will use a component called `MyCounter`, which displays a reactive variable `count` that increments when a button is clicked.

In all the following examples, the `App` component, which incorporates the `MyCounter` component, is as described here:

App component (file src/App.vue)

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

<template>
  <MyCounter />
</template>
```

We use the Composition API syntax to define and use a reactive variable `count` that increments upon clicking a button labeled "count+1". The Composition API of Vue.js defines the `ref()` method, which we will employ.

The `MyCounter` component is written as follows:

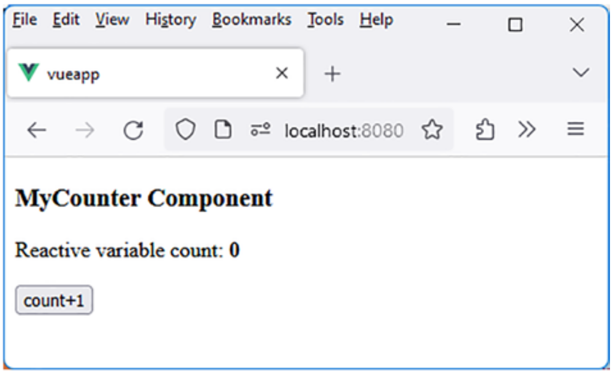
MyCounter component (file src/components/MyCounter.vue)

```
<script setup>

import { ref } from "vue";

// Creating a reactive variable count with an initial
value of 0
```

```
const count = ref(0);

</script>

<template>

<h3>MyCounter Component</h3>

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

</template>
```

The Composition API is used here within the `<script setup>` syntax. The `ref()` method, defined in Vue.js, is usable because it is imported with the statement `import { ref } from "vue"`. All Vue.js methods used in our programs must be imported in this manner before they can be utilized. A slight variation of this syntax is explained in the following section.

The reactive variable `count` is created and initialized with the statement `count = ref(value)`. Subsequently, the `count` variable can be used in the `<template>` section, as mentioned earlier. Let's display the URL `http://localhost:8080` in the browser:



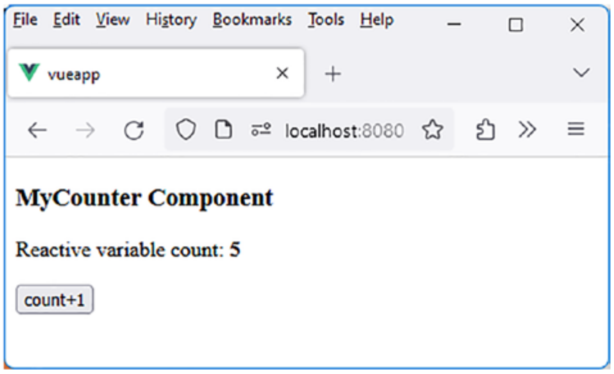
MyCounter Component

Reactive variable count: **0**

count+1

Figure 1-22. *Usage of a reactive variable "count"*

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



MyCounter Component

Reactive variable count: 5

Figure 1-23. *Incrementing the reactive variable "count"*

The variable `count` is indeed reactive as it is modified with each click on the "count+1" button.

Notice that the incrementation of the `count` variable is performed directly in the `<template>` section by writing the `<button>` element as follows:

Incrementation of the reactive variable count (file src/components/MyCounter.vue)

```
<button @click="count++">count+1</button>
```

The operation performed here is simple and corresponds to the `"count++"` instruction. However, in more complex scenarios, it would be more appropriate to use a function call to perform the corresponding processing.

Importing Vue.js Methods into Our Programs

The use of the `ref()` method requires writing the statement `import { ref } from "vue"`, where the curly braces contain the list of methods used in the JavaScript code.

Alternatively, one can perform a global import of all Vue.js methods without having to enumerate them in the list. This can be achieved using the statement `import * as vue from "vue"`. The variable name used after the `"as"` keyword is arbitrary but must be used in the subsequent JavaScript code. Let's rewrite the previous program following this principle:

Using import * as vue from "vue" (file src/components/MyCounter.vue)

```
<script setup>

import * as vue from "vue";

// Creating a reactive variable count with an initial
value of 0
const count = vue.ref(0);

</script>

<template>

<h3>MyCounter Component</h3>

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

</template>
```

We observe that the statement `ref(0)` needs to be prefixed with the variable `"vue"`, so it is written as `vue.ref(0)`. In this instance, we chose to use the variable name `"vue"`, but it can be any variable name.

Defining Methods in a Vue.js Component

We've seen how to create reactive variables in the component using the `ref(value)` method. It's also possible to create methods in a component that can be used in the `<template>` section of the component.

Previously, we wrote the processing to be executed after clicking the button directly in the value of the `@click` attribute. Instead of specifying the `count++` instruction, we can replace it with a method call, for example, `increment()`. Thus, we would write `@click="increment()"` instead of `@click="count++"`.

Let's explore how to define and use methods using the Composition API syntax.

Definition of the increment() method (file src/components/MyCounter.vue)

```
<script setup>

import { ref } from "vue";

// Creating a reactive variable count with an initial
value of 0
const count = ref(0);

function increment() {
  count.value++;  // One accesses the value of count using
                  count.value
}

</script>
```

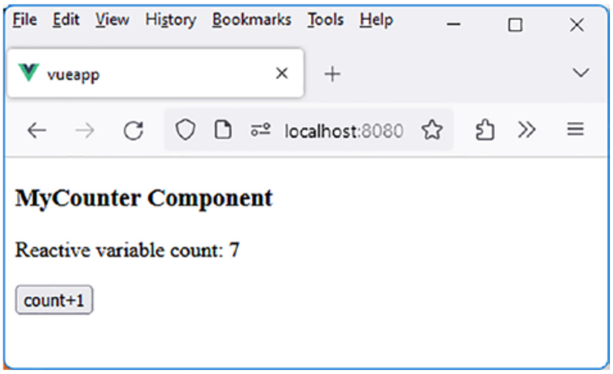
```
<template>

<h3>MyCounter Component</h3>

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

</template>
```

Functions are defined in the `<script setup>` section of the component. Functions defined in this section will then be accessible in the `<template>` section of the component.

Accessing the reactive variable `count` defined by `count = ref(0)` is done using the `value` property of the reactive variable `count`, namely, `count.value`.

Note that the variable `count` can be defined using the `const` keyword because it remains constant (its value, representing a reference to an object in memory, does not change). It is the `value` property of this object that is modified.

Let's verify that everything works in the same manner:



MyCounter Component

Reactive variable count: 7

Figure 1-24. *Usage of methods in a component*

Note that the `increment()` function can also be defined using the ES6 syntax. Instead of:

Traditional definition of the increment() function

```
function increment() {
  count.value++;  // One accesses the value of count using
                  count.value
}
```

one can also write the following:

Function increment() defined using ES6 syntax

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

Now that we know how to define methods in a component, let's explore how to define and use new reactive variables that depend on those already created. These new reactive variables are called computed properties.

Defining Computed Properties in a Vue.js Component

Computed properties allow the creation of new reactive variables based on those already defined. Since computed properties are also reactive, they will be updated if any of the reactive variables they depend on is modified.

The advantage of computed properties is that their result is calculated only if one of the reactive variables associated with them is modified. This saves processing time compared to calling a traditional method (as a method performs its processing each time it is used).

To illustrate this, let's create a new reactive variable named `doubleCount`, which is used to calculate double the value of the reactive variable `count`. The variable `doubleCount` is a computed property because it depends on one or more reactive variables. Its value will be automatically updated whenever any of the reactive variables it depends on is updated.

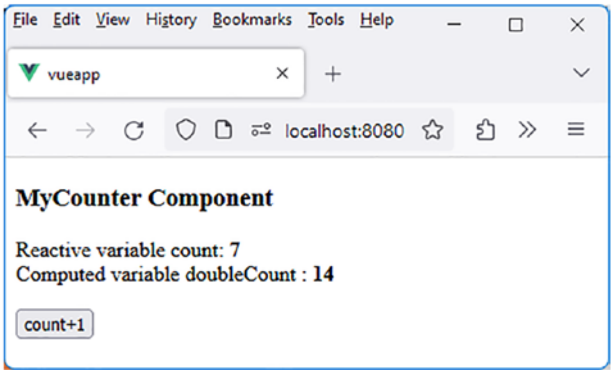
To define and use a computed property with the Composition API, we use the `computed(callback)` method defined in Vue.js. The `callback()` function should return the value of the computed property. The return value of the `computed()` method is associated with the name of the computed property, in the form of `const doubleCount = computed(callback)` to create the computed property `doubleCount`.

Creation of the computed property doubleCount (file src/components/MyCounter.vue)

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

// Usage of ref() to create a reactive variable
const count = ref(0);

// Usage of computed() to create a computed variable
const doubleCount = computed(function() {
  return 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 /><br />
<button @click="increment()">count+1</button>

</template>
```

After several clicks on the "count+1" button:



MyCounter Component

Reactive variable count: 7
Computed variable doubleCount : 14

Figure 1-25. *Creation of the computed property doubleCount*

We have seen how to use the click on a button to increment the reactive variable `count`. But suppose we want to increment the variable automatically, every second, as soon as the component is displayed. We will need to use a new concept, which is to use the lifecycle methods of a Vue.js component.

Lifecycle in a Vue.js Component

When a Vue.js component is created in memory, it follows an internal lifecycle that corresponds to the different phases of its evolution:

1. It is created in memory and then attached to the DOM tree of the current HTML page.
2. Next, it is possibly updated, if necessary (in the Virtual DOM and then in the real DOM).
3. Finally, it is possibly removed, if necessary (from the Virtual DOM and the real DOM).

Each of the preceding steps is associated with an internal method in the Vue.js component. These internal methods are called lifecycle methods, and it is possible to use them in each Vue.js component. You can then write specific processing in each method.

Here are the different lifecycle methods that can be used in a Vue.js component. They all use a `callback` parameter, which is a function that describes the processing to be performed when the lifecycle method is triggered. These methods are executed by Vue.js in the order they are written here:

1. `setup`: The setup is not associated with a method but corresponds to the code written in the `<script setup>` section. It is executed only once during the creation of the component in memory.
2. `onMounted(callback)`: This method is executed only once when the component has been inserted into the HTML page. The HTML elements associated with the component are in the DOM.
3. `onUpdated(callback)`: This method is executed when the component has been updated (in memory and in the DOM). It is executed with each update.
4. `onUnmounted(callback)`: This method is executed only once when the component is removed from memory.

The aforementioned methods allow processing after the insertion, update, or destruction of the component. Other methods exist to perform processing before it is realized. These are the following methods:

1. `onBeforeMount(callback)`: This method is executed only once before the component is inserted into the HTML page.

2. `onBeforeUpdate(callback)`: This method is executed before the component is updated, with each update.

3. `onBeforeUnmount(callback)`: This method is executed only once before the component is removed.

Let's write these methods in the `MyCounter` component and use them to display traces in the console.

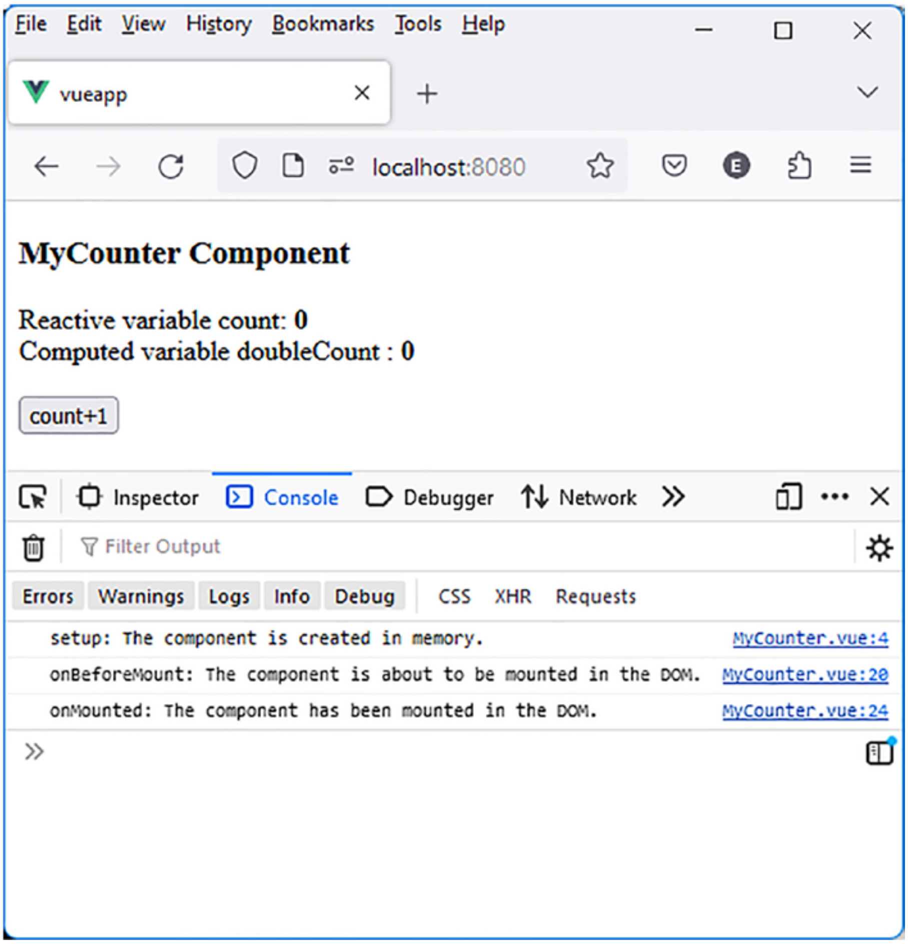
Using lifecycle methods (file src/components/MyCounter.vue)

```
<script setup>
import { ref, computed, onBeforeMount, onMounted,
onBeforeUpdate, onUpdated, onBeforeUnmount, onUnmounted }
from "vue";

console.log("setup: The component is created in memory.");

// Usage of ref() to create a reactive variable
const count = ref(0);

// Usage of computed() to create a computed variable
const doubleCount = computed(function() {
  return count.value * 2;
});
```

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

// Lifecycle Methods
onBeforeMount(() => {
  console.log("onBeforeMount: The component is about to be
  mounted in the DOM.");
});

onMounted(() => {
  console.log("onMounted: The component has been mounted in
  the DOM.");
});

onBeforeUpdate(() => {
  console.log("onBeforeUpdate: The component is about to be
  updated.");
});

onUpdated(() => {
  console.log("onUpdated: The component has been updated.");
});

onBeforeUnmount(() => {
  console.log("onBeforeUnmount: The component is about to be
  unmounted.");
});

onUnmounted(() => {
  console.log("onUnmounted: The component has been
  unmounted.");
});
```

```
</script>

<template>

<h3>MyCounter Component</h3>
Reactive variable count: <b>{{ count }}</b>
<br />
Computed variable doubleCount : <b>{{ doubleCount }}</b>
<br /><br />
<button @click="increment()">count+1</button>

</template>
```

Let’s run the previous program and observe the traces displayed in the console:



MyCounter Component

Reactive variable count: 0
Computed variable doubleCount : 0

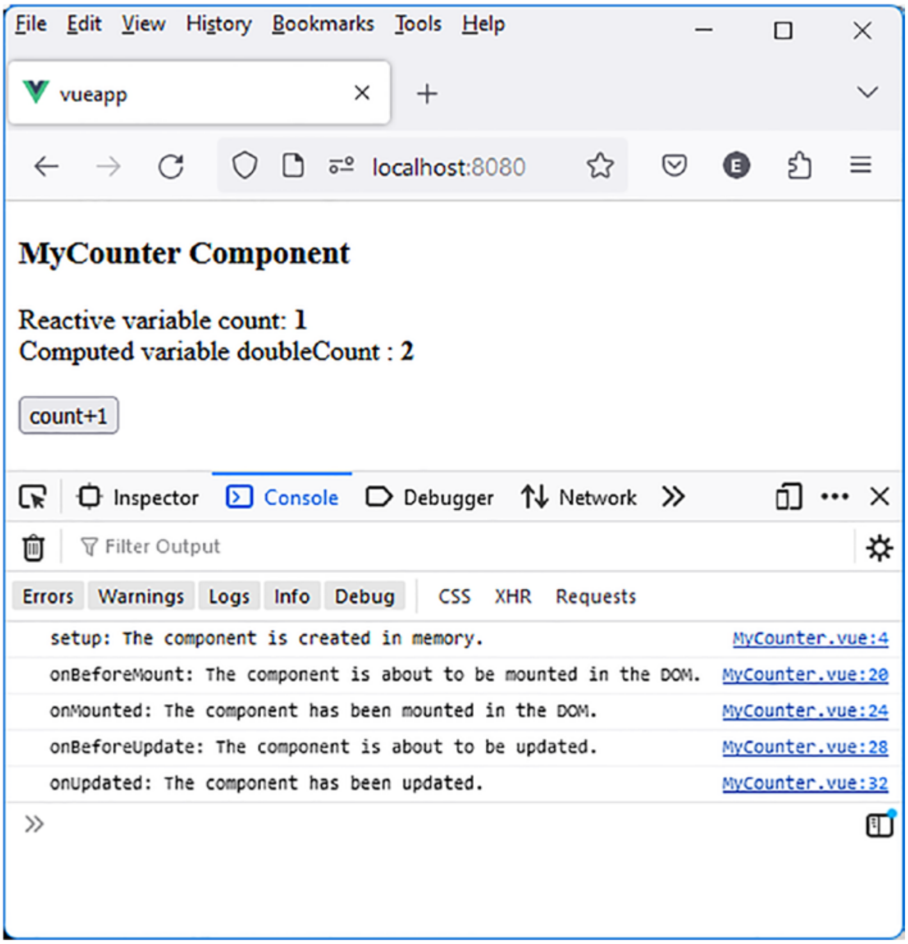
count+1

```
setup: The component is created in memory.                                MyCounter.vue:4
onBeforeMount: The component is about to be mounted in the DOM.         MyCounter.vue:20
onMounted: The component has been mounted in the DOM.                   MyCounter.vue:24
```

Figure 1-26. *Lifecycle methods*

We can see that the component is first created in memory and then mounted in the DOM.

Let's click the "count+1" button. This will update the component by incrementing the reactive variable `count`:


File Edit View History Bookmarks Tools Help
vueapp
localhost:8080
Inspector Console Debugger Network
Filter Output
Errors Warnings Logs Info Debug CSS XHR Requests

MyCounter Component

Reactive variable count: 1
Computed variable doubleCount : 2

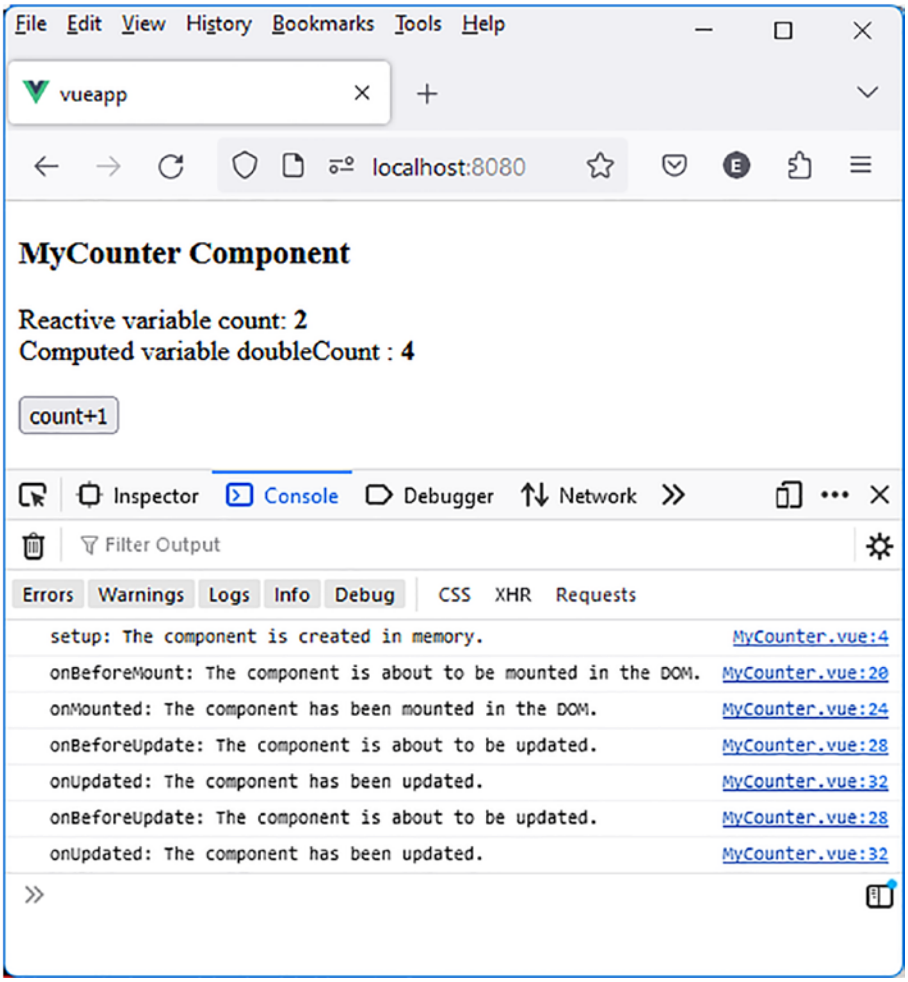
count+1

```
setup: The component is created in memory.                           MyCounter.vue:4
onBeforeMount: The component is about to be mounted in the DOM.      MyCounter.vue:20
onMounted: The component has been mounted in the DOM.                MyCounter.vue:24
onBeforeUpdate: The component is about to be updated.                MyCounter.vue:28
onUpdated: The component has been updated.                           MyCounter.vue:32
```

Figure 1-27. *Component updated*

The `onBeforeUpdate()` and `onUpdated()` methods were executed during the update of the reactive variable.

If we click again the "count+1" button, these methods are executed again:



MyCounter Component

Reactive variable count: 2
Computed variable doubleCount : 4

count+1

```
setup: The component is created in memory.                        MyCounter.vue:4
onBeforeMount: The component is about to be mounted in the DOM.   MyCounter.vue:20
onMounted: The component has been mounted in the DOM.             MyCounter.vue:24
onBeforeUpdate: The component is about to be updated.             MyCounter.vue:28
onUpdated: The component has been updated.                        MyCounter.vue:32
onBeforeUpdate: The component is about to be updated.             MyCounter.vue:28
onUpdated: The component has been updated.                        MyCounter.vue:32
```

Figure 1-28. *New update of the component*

Example of Using Lifecycle Methods in a Vue.js Component

We've seen when the lifecycle methods are triggered.

Let's see how to use these lifecycle methods so that the MyCounter component displays a counter that increments every second (instead of using the button click as before). To achieve this, we need to use a timer with the setInterval() method of JavaScript. The timer will start in one of the methods indicating the creation of the component (e.g., the onMounted() method), and it will be stopped in one of the methods indicating the removal of the component (e.g., the onUnmounted() method).

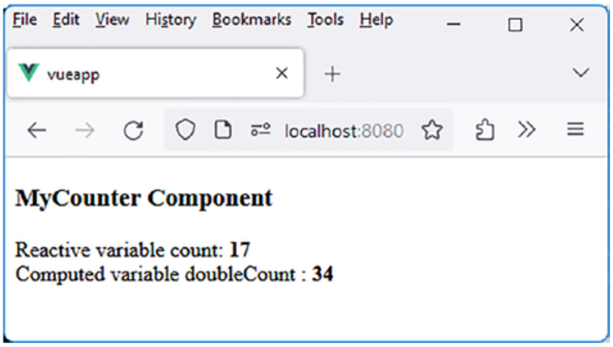
We define two new methods in the MyCounter component, which are the start() and stop() methods:

1. The start() method starts the timer and will be called in the onMounted() method.
2. The stop() method stops the timer and will be called in the onUnmounted() method.

Start the counter automatically (file src/components/MyCounter.vue)

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

let timer;

const count = ref(0);

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

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

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

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

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

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

</script>


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

</template>
```

Since the counter is started automatically, the "count+1" button has been removed from the component. The `start()` method is called in the `onMounted()` method but could have been called directly from the `<script setup>` section. Let's verify that the counter is now automatically started without having to click the button:



MyCounter Component

Reactive variable count: 17
Computed variable doubleCount : 34

Figure 1-29. *Automatic start of the counter*

The counter is now automatically started when the component is displayed.

Managing Reactivity Ourselves with the customRef( ) Method

In everything we have previously explained, we have seen that whenever a reactive variable is updated, its new value is automatically reflected in the component where it is displayed. This operation is due to the internal functioning of reactivity in Vue.js, which performs it for us. Most of the time, this is the desired behavior.

However, it is possible to create our own reactivity. This would allow us to perform processing before a reactive variable is displayed or before a reactive variable is modified.

Here are some examples of using this mechanism:

- **Date Formatting:** Before displaying a date, we format it according to the format of the country using it. The date would be stored internally in the YYYYMMDD format but would be displayed, for example, in the DD/MM/YYYY format for better display comfort or according to the MM-DD-YYYY format.

- **Email Validation:** Once the email is entered, we check that it is in the correct format before storing it.
- **Processing During Field Entry:** Instead of performing processing for each character entered, we wait for a minimum number of characters to be entered before performing it. This prevents making requests to a server too quickly.
- **Checking the Format of a Password:** It is sometimes useful to check that the chosen password has characteristics such as at least ten characters, at least one digit, at least one uppercase letter, at least one lowercase letter, etc.

This operation is possible, thanks to the `customRef()` method, which allows creating reactive variables with a specific behavior. Let's now examine how the `customRef()` method works to define new reactive variables.

Step 1: Operation and Use of customRef( )

A reactive variable defined with `customRef()` works in the same way as one defined with `ref()`. Thus, the `value` property of the variable allows accessing it in both reading and writing.

However, we define ourselves what the reading of the variable should return and also what the variable should finally contain when its value is initialized or modified. This is the advantage of using this type of reactive variables defined by `customRef()` rather than `ref()`, as we can modify the result of reading and/or the result of writing the variable, and this in a centralized manner (in the `customRef()` method).

The difference with the `ref()` method is that reading a variable defined with `ref()` returns the exact value of the content of the variable, and its writing writes exactly into the variable what is indicated in the value attribute.

The questions to ask when creating a variable with `customRef()` are as follows:

- What do we want to get back when reading the variable?
- What do we want to get in the content of the variable when it is modified?

If the answer to these two questions is to keep the current value, just use the `ref()` method to create the reactive variable. The `customRef()` method would not add anything more in this case.

The `customRef(callback)` method uses the `callback(track, trigger)` function, which allows returning an object with the properties `{ get, set }`, which are functions defining the reading of the variable (with the `get()` function) and the modification of the variable (with the `set()` function).

- The `get()` function must return the value we want to read when accessing the variable (in reading).
- The `set(newValue)` function must set the new value of the variable (in writing). The `newValue` parameter is the value we want to write, but it can be modified in the `set()` method.

The `track()` and `trigger()` parameters of the callback function are functions to call when you want to trigger reading (by `track()`) or update (by `trigger()`).

To understand these explanations, the simplest way is to write a minimal program that does this work. The traces written in the program will help understand how it works.

We create a count variable defined by customRef(), which increments by 1 with each click on a "count+1" button.

Incrementing a count variable defined by customRef() (file src/components/MyCounter.vue)

```
<script setup>
import { customRef } from 'vue';

// Create a custom reference (customRef)
const count = customRef((track, trigger) => {
  let value = 0;  // value will be the variable being tracked,
  initialized here to 0.
  return {
    get() {
      // Track the dependency when the value is read.
      track();
      console.log("get value =", value);
      return value;
    },
    set(newValue) {
      // Update the value and trigger reactivity.
      value = newValue;
      trigger();
      console.log("set value =", value);
    }
  };
});

function increment() {
  console.log("before increment value");
  count.value += 1;  // Increment the value of the variable.
  console.log("after increment value");
}
```

```
</script>

<template>

<h3> MyCounter Component </H3>

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

</template>
```

The `count` variable is created when calling the `customRef(callback)` method. Once the `count` variable is created, it can be used by writing instructions like `count.value += 1`, which here increments the value by 1. The usage principle is the same as if the variable had been created by `ref()`.

The callback function used in `customRef(callback)` has the form `callback(track, trigger)`, where `track()` will be used in the `get()` method and `trigger()` in the `set()` method.

The callback function starts by creating a variable (here `value`) that will be the internally manipulated variable and corresponds to the value of the `count` variable. The variable is initialized to 0 here.

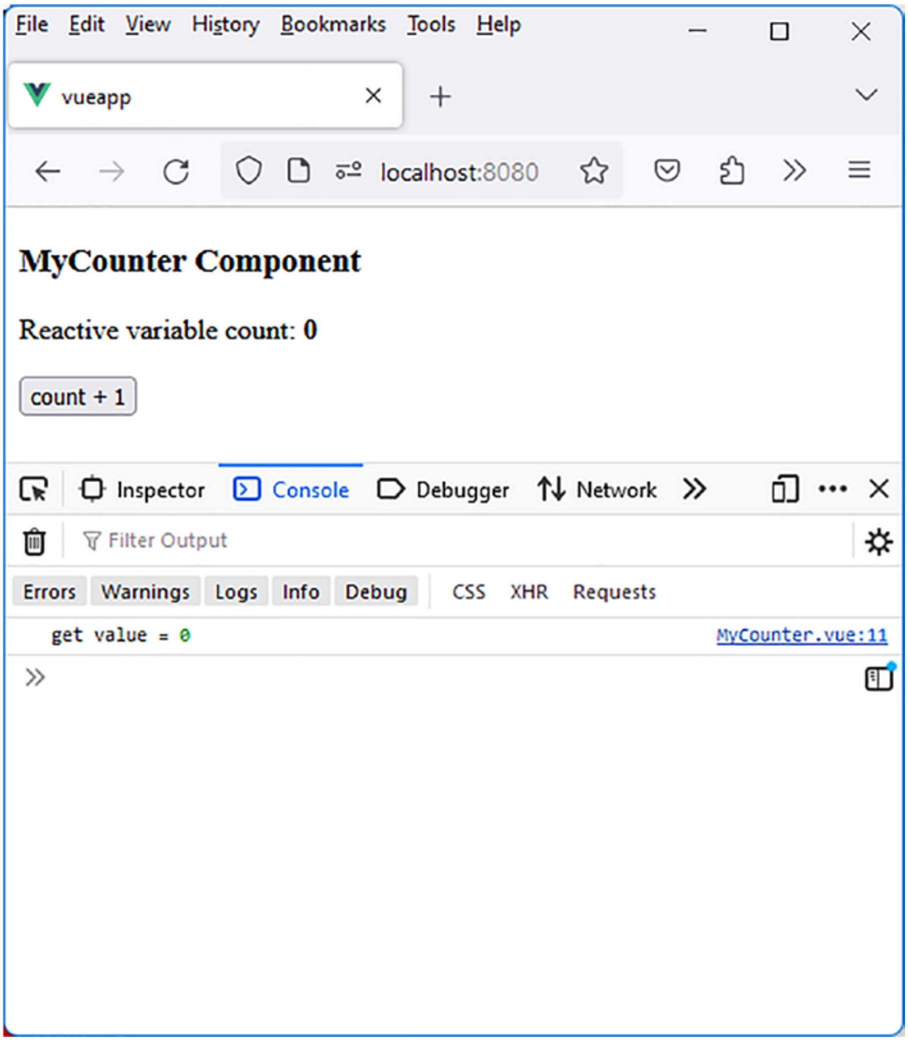
The callback function then returns an object `{ get, set }` that defines the `get()` and `set()` methods:

- The `get()` method calls the `track()` method, indicating that we want to track the variable mentioned next, and its value is returned by `get()`.
- Here, we want to return the exact value of the variable that will be used when reading the `count` variable, but we could return something else.

- The `set(newValue)` method calls the `trigger()` method to indicate that we are going to modify the variable's value, and this new value must be taken into account. Here, we assign the `newValue` to `value`, but we could assign another value.

The implemented functionality here corresponds to the functionality used with the `ref()` method. However, it allows, with the traces performed in the methods, to see the internal functioning provided by the `customRef()` method.

Let's execute the previous program, displaying the traces in the console (F12 key):


File Edit View History Bookmarks Tools Help
vueapp
localhost:8080
Inspector Console Debugger Network
Filter Output
Errors Warnings Logs Info Debug CSS XHR Requests
count + 1

MyCounter Component

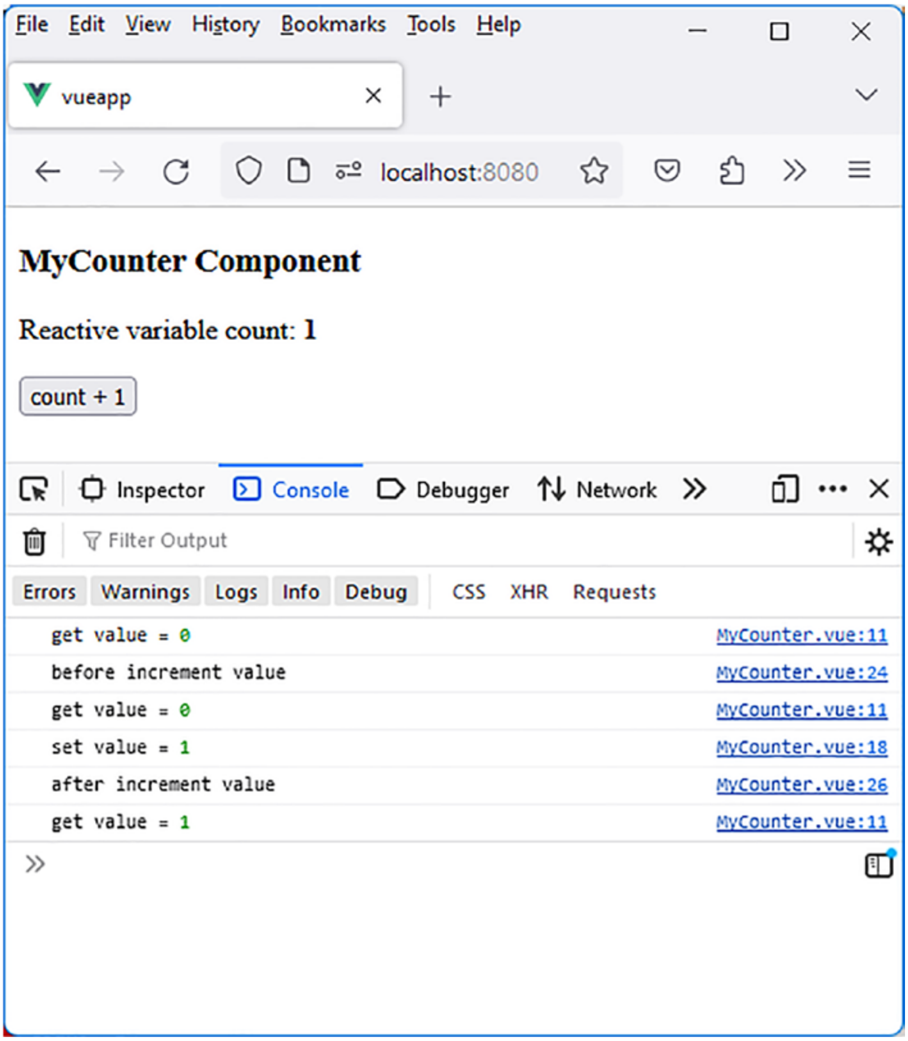
Reactive variable count: 0

```
get value = 0                                    MyCounter.vue:11
```

Figure 1-30. *Using a variable defined by customRef()*

We can see that at the program's launch, the `get()` method is called to retrieve the variable's value and display it in the component.

Next, let's click the "count+1" button.


File Edit View History Bookmarks Tools Help
vueapp
localhost:8080
MyCounter Component
Reactive variable count: 1
count + 1
Inspector Console Debugger Network
Filter Output
Errors Warnings Logs Info Debug CSS XHR Requests

```
get value = 0                                   MyCounter.vue:11
before increment value                          MyCounter.vue:24
get value = 0                                   MyCounter.vue:11
set value = 1                                   MyCounter.vue:18
after increment value                           MyCounter.vue:26
get value = 1                                   MyCounter.vue:11
```

Figure 1-31. *Incrementing a variable defined by customRef()*

When clicking the "count+1" button, the click handling function `increment()` is executed. We can see in the traces that to execute the `count.value += 1;` instruction, we first read the value (via `get()`) and then update the value (via `set()`). Afterward, a final read (via `get()`) is performed, which corresponds to displaying the new value in the component.

Now that we have seen the internal functioning of `customRef()`, let's use it to produce more interesting results.

Step 2: Limiting the Increment of a Variable

During the variable update (in the `set()` method), it is possible to decide the value that will finally be assigned to the variable.

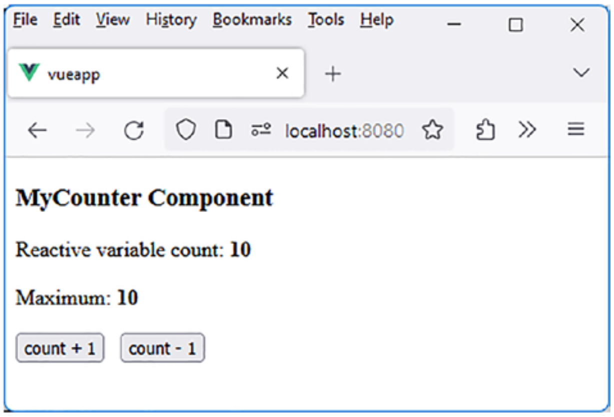
For example, we can create a counter that stops at the value 10 without being able to exceed it.

For this example, we use a "count+1" button that increments the counter and a "count - 1" button that decrements it. Once the maximum value is reached, we can only decrement the counter, which can then be incremented again.

Using a maximum value (file src/components/MyCounter.vue)

```
<script setup>
import { customRef } from 'vue';

// Maximum value not to be exceeded
const maximum = 10;

// Create a custom reference (customRef)
const count = customRef((track, trigger) => {
  let value = 0;  // value will be the variable being tracked,
                  initialized here to 0.
  return {
    get() {
      // Track the dependency when the value is read.
      track();
      return value;
    },
    set(newValue) {
      // Update the value and trigger reactivity.
      if (newValue <= maximum) value = newValue;
      trigger();
    }
  };
});

function increment() {
  count.value += 1;  // Increment the value of the variable.
}

function decrement() {
  count.value -= 1;  // Decrement the value of the variable.
}

</script>

<template>

<h3> MyCounter Component </H3>

Reactive variable count: <b>{{ count }}</b>
<br><br>
Maximum: <b>{{ maximum }}</b>
<br><br>
<button @click="increment">count + 1</button>
&nbsp;
<button @click="decrement">count - 1</button>

</template>
```

We use a variable named `maximum` with a value of 10. Since this variable will not be modified, there is no need to make it reactive. The `count` variable is managed by `customRef()`. The value of the variable managed by `customRef()` is modified only if it is less than the maximum value. When the maximum value (here, 10) is reached, the counter is locked:



MyCounter Component

Reactive variable count: **10**

Maximum: **10**

Figure 1-32. *Blocking the counter at the maximum value*

We have prevented the possibility of incrementing the value of the `count` variable. Let's delve further into program structuring by creating a function known as a "composable" that easily creates reactive variables with the characteristic of limiting their value to a maximum.

Step 3: Create a Composable useMaximum( ) to Limit the Value

Instead of creating the `count` variable by defining it in the component's code, we create a function that generates the desired variable. This allows us to reuse the function in various parts of the program if necessary.

This approach will be explored in Chapter 6 and corresponds to the creation of composables. Composables are utility functions that can be used elsewhere in the program or even in other projects.

Here, we will create the `useMaximum()` composable, which limits the value of the reactive variable to the specified `max` value.

Create and use the useMaximum(max) composable (file src/components/MyCounter.vue)

```
<script setup>
import { customRef } from 'vue';

// Maximum value not to be exceeded
const maximum = 10;

const useMaximum = (max) => {
  // Create a custom reference (customRef)
  return customRef((track, trigger) => {
    let value = 0;  // value will be the variable being
    tracked, initialized here to 0.
    return {
      get() {
        // Track the dependency when the value is read.
        track();
        return value;
      },
      set(newValue) {
        // Update the value and trigger reactivity.
        if (newValue <= max) value = newValue;
        trigger();
      }
    };
  });
}
```

```
const count = useMaximum(maximum);

function increment() {
  count.value += 1;  // Increment the value of the variable.
}

function decrement() {
  count.value -= 1;  // Decrement the value of the variable.
}

</script>

<template>

<h3> MyCounter Component </H3>

Reactive variable count: <b>{{ count }}</b>
<br><br>
Maximum: <b>{{ maximum }}</b>
<br><br>
<button @click="increment">count + 1</button>
&nbsp;
<button @click="decrement">count - 1</button>

</template>
```

The `count` variable is now created when calling the `useMaximum()` method. This approach allows us to easily create multiple variables of this type.

It is customary for the composable to be placed in a separate file. Therefore, we create the file `useMaximum.js`, which corresponds to the source code of the `useMaximum()` composable. This file is placed in the `src/composables` directory, which is created if necessary.

Composable useMaximum() (file src/composables/useMaximum.js)

```
import { customRef } from "vue";

const useMaximum = (max) => {
  // Create a custom reference (customRef)
  return customRef((track, trigger) => {
    let value = 0;  // value will be the variable being
                    tracked, initialized here to 0.
    return {
      get() {
        // Track the dependency when the value is read.
        track();
        return value;
      },
      set(newValue) {
        // Update the value and trigger reactivity.
        if (newValue <= max) value = newValue;
        trigger();
      }
    };
  });
}

export default useMaximum;
```

The code of the `MyCounter` component can then be simplified. It is enough to import the composable `useMaximum()` file into the component:

MyCounter component using the useMaximum() composable (file src/components/MyCounter.vue)

```
<script setup>

import useMaximum from "../composables/useMaximum.js"
```

```
// Maximum value not to be exceeded
const maximum = 10;

const count = useMaximum(maximum);

function increment() {
  count.value += 1;  // Increment the value of the variable.
}

function decrement() {
  count.value -= 1;  // Decrement the value of the variable.
}

</script>

<template>

<h3> MyCounter Component </H3>

Reactive variable count: <b>{{ count }}</b>
<br><br>
Maximum: <b>{{ maximum }}</b>
<br><br>
<button @click="increment">count + 1</button>
&nbsp;
<button @click="decrement">count - 1</button>

</template>
```

The component's code is simplified. The functionality remains the same.

Step 4: Create a Composable useFormatDate( ) to Format Dates

A final example of using `customRef()` would be to create a composable that displays dates according to the desired format:

- MM-DD-YYYY (internal format "en-US")
- DD-MM-YYYY (internal format "en-GB")
- MM/DD/YYYY (internal format "en-US")
- DD/MM/YYYY (internal format "en-GB")

The `useFormatDate()` composable returns a reactive variable with the desired format.

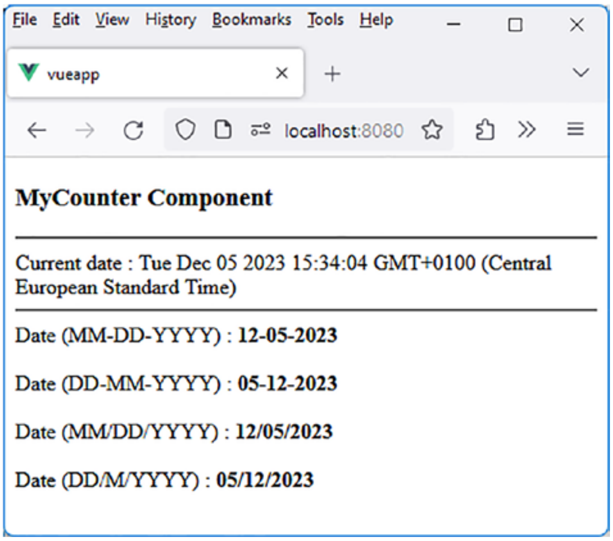
Composable useFormatDate(date, format) (file src/components/MyCounter.vue)

```
<script setup>
import { customRef } from "vue";

const formatDate = (date, format) => {
  const options = { year: 'numeric', month: '2-digit',
  day: '2-digit' };

  if (format == "MM-DD-YYYY")
    return date.toLocaleDateString('en-US', options).
    replace(/\//g, '-');
  else if (format == "DD-MM-YYYY")
    return date.toLocaleDateString('en-GB', options).
    replace(/\//g, '-');
  else if (format == "MM/DD/YYYY")
    return date.toLocaleDateString('en-US', options);
  else if (format == "DD/MM/YYYY")
```

```
    return date.toLocaleDateString('en-GB', options);
}

const useFormatDate = (date, format) => {
  return customRef((track, trigger) => {
    let value = date;  // value will be the tracked variable
    return {
      get() {
        // track the dependency when the value is read
        track();
        return formatDate(value, format);
      },
      set(newValue) {
        // update the value and trigger reactivity
        value = newValue;
        trigger();
      }
    };
  });
};

const date= new Date();  // Current date

const dateMMDDYYYY = useFormatDate(date, "MM-DD-YYYY");
const dateDDMMYYYY = useFormatDate(date, "DD-MM-YYYY");
const dateMMDDYYYY_slash = useFormatDate(date, "MM/DD/YYYY");
const dateDDMMYYYY_slash = useFormatDate(date, "DD/MM/YYYY");

</script>

<template>

<h3> MyCounter Component </H3>
```

```
<hr>
Current date : {{date}}
<hr>
Date (MM-DD-YYYY) : <b>{{ dateMMDDYYYY }}</b>
<br><br>
Date (DD-MM-YYYY) : <b>{{ dateDDMMYYYY }}</b>
<br><br>
Date (MM/DD/YYYY) : <b>{{ dateMMDDYYYY_slash }}</b>
<br><br>
Date (DD/M/YYYY) : <b>{{ dateDDMMYYYY_slash }}</b>
<br><br>

</template>
```

We display today's date in various formats. The obtained result is displayed as follows:



MyCounter Component

Current date : Tue Dec 05 2023 15:34:04 GMT+0100 (Central European Standard Time)

Date (MM-DD-YYYY) : **12-05-2023**

Date (DD-MM-YYYY) : **05-12-2023**

Date (MM/DD/YYYY) : **12/05/2023**

Date (DD/M/YYYY) : **05/12/2023**

Figure 1-33. *Displaying dates in various formats*

As in the previous example, it is desirable to create a separate file that contains the code for the composable. This simplifies the component's writing and enables the use of the composable in other components.

The file for the `useFormatDate()` composable will be registered in a file named `useFormatDate.js`, located in the `src/composables` directory.

Composable useFormatDate() (file src/composables/useFormatDate.js)

```
import { customRef } from "vue";

const formatDate = (date, format) => {
  const options = { year: 'numeric', month: '2-digit', day:
  '2-digit' };

  if (format == "MM-DD-YYYY")
    return date.toLocaleDateString('en-US', options).
    replace(/\//g, '-');
  else if (format == "DD-MM-YYYY")
    return date.toLocaleDateString('en-GB', options).
    replace(/\//g, '-');
  else if (format == "MM/DD/YYYY")
    return date.toLocaleDateString('en-US', options);
  else if (format == "DD/MM/YYYY")
    return date.toLocaleDateString('en-GB', options);
}

const useFormatDate = (date, format) => {
  return customRef((track, trigger) => {
    let value = date;  // value will be the tracked variable
    return {
      get() {
        // track the dependency when the value is read
        track();
        return formatDate(value, format);
      },
      set(newValue) {
        // update the value and trigger reactivity
        value = newValue;
        trigger();
      }
    };
  });
};

export default useFormatDate;
```

The code for the `MyCounter` component becomes the following:

MyCounter component using the useFormatDate() composable (file src/components/MyCounter.vue)

```
<script setup>

import useFormatDate from "../composables/useFormatDate.js"

const date= new Date();  // Current date

const dateMMDDYYYY = useFormatDate(date, "MM-DD-YYYY");
const dateDDMMYYYY = useFormatDate(date, "DD-MM-YYYY");
const dateMMDDYYYY_slash = useFormatDate(date, "MM/DD/YYYY");
const dateDDMMYYYY_slash = useFormatDate(date, "DD/MM/YYYY");

</script>

<template>

<h3> MyCounter Component </H3>

<hr>
Current date : {{date}}
```

```
<hr>
Date (MM-DD-YYYY) : <b>{{ dateMMDDYYYY }}</b>
<br><br>
Date (DD-MM-YYYY) : <b>{{ dateDDMMYYYY }}</b>
<br><br>
Date (MM/DD/YYYY) : <b>{{ dateMMDDYYYY_slash }}</b>
<br><br>
Date (DD/M/YYYY) : <b>{{ dateDDMMYYYY_slash }}</b>
<br><br>

</template>
```

Step 5: Entering Text in Uppercase

Another use of this mechanism is, for example, to enter text in uppercase, which standardizes the entry of names in an application. The field is transformed into uppercase as characters are entered into the field.

We still use the `MyCounter` component in which a variable named `name` corresponding to an input field is displayed. To achieve this, we use the `v-model` directive, which will be explained in the next chapter.

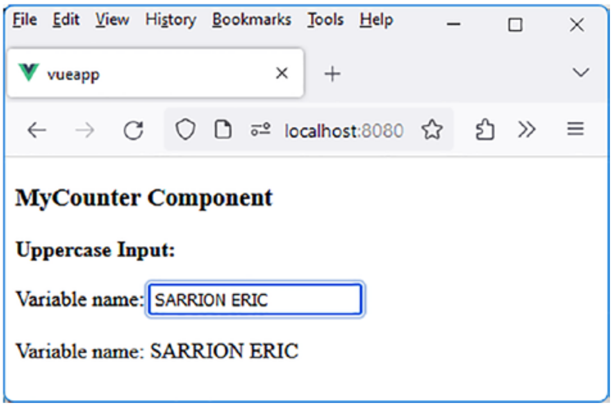
The input in the field is displayed in uppercase letters, regardless of the entered letters.

Entering text in uppercase (file src/components/MyCounter.vue)

```
<script setup>
import { customRef } from 'vue';

const name = customRef((track, trigger) => {
  let value = "";  // value will be the tracked variable
  return {
    get() {
      // track the dependency when the value is read
```

```
        track();
        return value.toUpperCase();
      },
      set(newValue) {
        // update the value and trigger reactivity
        value = newValue;
        trigger();
      }
    };
  });

</script>

<template>

<h3> MyCounter Component </H3>

<b>Uppercase Input:</b>
<br><br>
Variable name: <input type="text" v-model="name" />
<br><br>
Variable name: {{ name }}

</template>
```

As explained earlier, it is sufficient to replace the value to be read from the variable with its uppercase representation. This is done in the `get()` method.



MyCounter Component

Uppercase Input:

Variable name: SARRION ERIC

Variable name: SARRION ERIC

Figure 1-34. *Uppercase name input*

It is also possible to create a separate file that defines the `useUpperCase(initialValue)` composable with an `initialValue` parameter. The file `useUpperCase.js` will be placed in the `src/composables` directory.

Composable useUpperCase(initialValue) (file src/composables/useUpperCase.js)

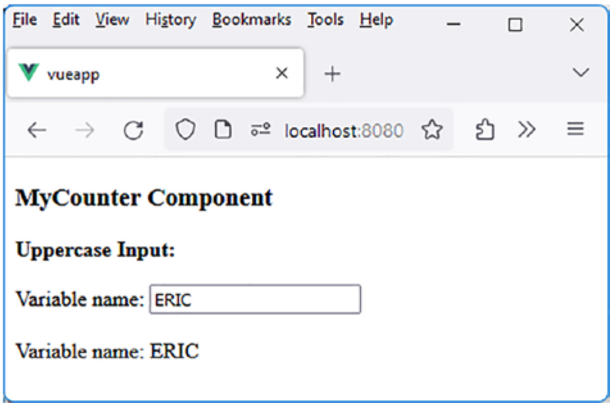
```
import { customRef } from 'vue';

const useUpperCase = (initValue) => {
  return customRef((track, trigger) => {
    let value = initValue;  // value will be the tracked
                            variable
    return {
      get() {
        // track the dependency when the value is read
        track();
        return value.toUpperCase();
      },
      set(newValue) {
        // update the value and trigger reactivity
        value = newValue;
        trigger();
      }
    };
  });
};

export default useUpperCase;
```

The use of the `useUpperCase()` composable in the `MyCounter` component is as follows:

MyCounter component using the useUpperCase() composable (file src/components/MyCounter.vue)

```
<script setup>

import useUpperCase from "../composables/useUpperCase.js"

const name = useUpperCase("eric");

</script>

<template>

<h3> MyCounter Component </H3>

<b>Uppercase Input:</b>
<br><br>
Variable name: <input type="text" v-model="name" />
<br><br>
Variable name: {{ name }}

</template>
```

The component is initialized with the variable `name` set to the value "eric", which will be immediately displayed in uppercase in both the input field and the following text.



MyCounter Component

Uppercase Input:

Variable name: ERIC

Variable name: ERIC

Figure 1-35. *Utilization of the useUpperCase() composable*

Conclusion

We have come a long way during this first day dedicated to mastering Vue.js! We have covered the necessary steps for creating the first Vue.js application, from installing Vue.js to analyzing the generated files.

We have also delved into key concepts such as component structuring, reactivity, defining methods and computed properties, as well as managing the lifecycle of components.

In the upcoming chapters, we will further deepen our Vue.js skills and explore more advanced concepts to create high-performance web applications. So get ready for the exciting continuation of this learning journey!