**LIFECYCLE HOOKS IN VUE 3**

In the modern web application development, the web pages are made up of number of components. Each component is responsible for its own business. The components ease the development, performance and readability of web page creation.

**Lifecycle hooks**

While we create components, we will go step by step procedure to fulfill its requirements. In Vue.js we can derive this as follows,

* Data observation
* Template compilation
* Mount the component instance to DOM
* Update the DOM based on data

Between the steps, Vue.js provides a predefined function called Life Cycle Hooks. We can add our own logics by calling these life cycle hooks in between the component creations.

**Options API vs Composition API**

Above all, we can access lifecycle hooks in both options API and composition API. We have to invoke life cycle methods inside the setup function in the Composition API.

**Note:**

Option API is available in both Vue.js 2 and 3 whereas composition API is available from Vue.js 3.

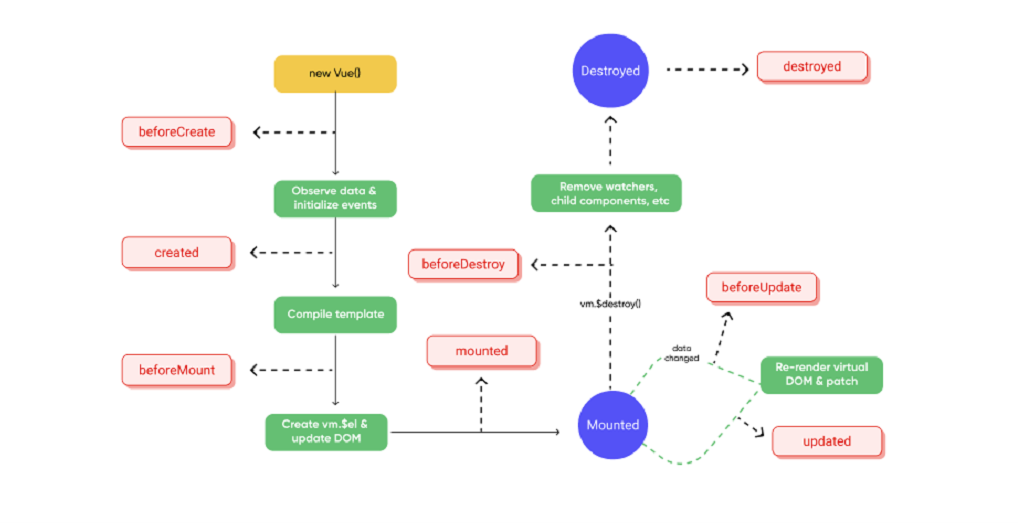
|  |  |
| --- | --- |
| **Options API** | **Composition API** |
| beforeCreate | Not needed\* (directly write in setup) |
| created | Not needed\*(directly write in setup) |
| beforeMount | onBeforeMount |
| mounted | onMounted |
| beforeUpdate | onBeforeUpdate |
| updated | onUpdated |
| beforeUnmount | onBeforeUnmount |
| unmounted | onUnmounted |
| errorCaptured | onErrorCaptured |
| renderTracked | onRenderTracked |
| renderTriggered | onRenderTriggered |

Most importantly, beforeCreate and created are not available in composition API. However, we can write it’s logic directly in the setup function since setup is invoked around the created hook.

**Note**

In Optional API, all lifecycle hooks have this context bound to the instance. So we should not go for arrow functions (e.g. created: () => this.createdCallBack()) since arrow functions bind the parent context to this.

**Vue --- Life cycle diagram**



**Lifecycle hooks – Options API**

|  |  |
| --- | --- |
| beforeCreate | Called after the component instance has been initialized but before data observation. |
| created | Called after the component instance has been created. Since instance is created we can set up data observation, computed properties, methods, watch/event callbacks. But we can not do DOM operation since the component is not mounted. |
| beforeMount | Called just before the component mounting process. |
| mounted | This is an important hook after created hook in the options API. mounted called after the component instance has been mounted. If the root component instance is mounted, then the variables in data() will be there in the DOM. But it does not guarantee that all child components have also been mounted. |
| beforeUpdate | Called at run time when data changes before the DOM is updated. It is a good place to remove manually added event listeners. |
| updated | Called at runtime after the data changes and DOM patched. |
| beforeUnmount | Called before a component instance is unmounted. |
| unmounted | Called after a component instance has been unmounted. |
| errorCaptured | Called when an error occurs from any child component. |
| renderTracked | Called when virtual DOM rerender tracked. |
| renderTriggered | Called when virtual DOM rerender triggered. |

**Coding example for Options API Life cycle hooks**

<template>

<div>

<h3> Life Cycle Hooks Demo - Options API </h3>

Count Value : {{count}}

</div>

<div>

<button @click.prevent="count++"> Click Me </button>

</div>

</template>

<script>

export default{

data(){

return{

count : 0

} },

beforeCreate(){

console.log('Before create called');

},

created(){

console.log('Created');

},

beforeMount(){

console.log('Before Mounted');

} ,

mounted(){

console.log('Mounted');

},

beforeUpdate(){

console.log('Before updated')

},

updated(){

console.log('updated')

},

beforeUnmount(){

console.log('Before Unmount')

},

unmounted(){

console.log('unmounted')

}

}

</script>

**Lifecycle hooks Vue 3 (Composition API )**

Life cycle hooks provide the ways to add their logic at specific stages while creating the components. Moreover in Vue.js 3, they are tree-shakable modules. You can import and use them in your composable logics.

**Coding example for Composition API Life cycle hooks**

<template>

<div> <h2> Life Cycle Hooks - Composition API </h2></div>

<div> Count Value : {{ state.count }}

<br />

<button @click.prevent="state.count++">Click Me </button>

</div>

<div v-if="showComp">

<AboutView />

</div>

<div>

<button @click="showComp=!showComp" > Toggle Show Component </button>

</div>

</template>

<script>

import { onBeforeMount,

onBeforeUnmount, onBeforeUpdate,

onMounted, onRenderTracked, onRenderTriggered, onUnmounted,

onUpdated , reactive, ref } from 'vue'

import AboutView from '@/views/AboutView.vue';

export default{

setup() {

let showComp = ref(true);

const state = reactive({

count: 0

});

onBeforeMount(() => {

console.log("Before Mounted"); });

onMounted(() => {

console.log("On mounted");

});

onBeforeUpdate(() => {

console.log("On Before Updated");

});

onUpdated(() => {

console.log("On Updated");

});

onBeforeUnmount(() => {

console.log("On Before Unmounted");

});

onUnmounted(() => {

console.log("On Unmounted");

});

onRenderTracked((e) => {

console.log(e);

});

onRenderTriggered((e) => {

console.log(e);

});

return { state, showComp };

},

components: { AboutView }

}

</script>

<style scoped></style>

It is not compulsory that you have to use Composition API in order to use Vue 3. It still supports Options API at its core. It is also possible to use both methods together in a single Vue component.

**What is The Composition API?**

This is a new way to write Vue components. With this method, you can make use of different hooks provided by Vue to add features to your applications.

We can use **ref()** and **reactive()** to create the reactive state in your component. There are **computed(), watch()** and **watchEffect()** to catch the changes in reactive states and perform some actions according to your requirements.

We have many lifecycle hooks for your component like **onMounted(), onBeforeMount(), onUpdated()** etc. You can run functions in different stages of your component lifecycle using these hooks.

Within Composition API, you have two types of syntax available to write your component. One is using **setup() function** and another one is using **<script setup>** (which is recommended by Vue).The **<script setup>** syntax provides some additional advantages over setup() function syntax. It takes less boilerplate to start, better runtime performance, you can use typescript more efficiently, better IDE type-interface support, etc.

**// Composition API with <script setup>**

<script setup>

import { onMounted, ref } from 'vue';

const amount = ref(0);

const balance = ref(0);

const balanceString = computed(() => `Account Balance: ${balance.value}`);

const addBalance = () => {

balance.value += amount.value;

amount.value = 0;

}

const subtractBalance = () => {

balance.value -= amount.value;

amount.value = 0;

}

onMounted(() => {

console.log('Application mounted');

});

</script>

**What is The Options API?**

The Options API is the old way to write a Vue component. In this method, you have to export an object with all the options.

By options, I mean the properties and methods in that exported object. Through those properties and methods, you can add functionalities to your application.

You can declare your state inside the data() method. You state will be an object returned from the **data()** method. **They are reactive by default**.

You can use a **computed property to declare getter functions**. The value returned from the computed function will be readonly.

There are **methods, props, watch** etc properties to use inside your exported object.

You can add some specific methods to work with your component lifecycle. For example, **mounted(), beforeMount(), created(), beforeCreate(), updated()** etc.

In the case of Options API, you have to do everything inside a primary object. Your logic gets scattered into different properties and methods. When your application becomes bigger, it makes the application hard to manage.

// Options API

<script>

export default {

data() {

return {

balance: 0,

amount: 0

}

},

computed: {

balanceString() {

return `Account Balance: ${this.balance}`;

}

},

methods: {

addBalance() {

this.balance += this.amount

this.amount = 0;

},

subtractBalance() {

this.balance -= this.amount

this.amount = 0;

}

},

mounted() {

console.log('Application mounted'); }, } </script>

**Composition API VS Options API in Vue 3**

Both APIs are alternatives to each other. You can also use them together if you want. But they are different in terms of their appearance.

In Options API, we were limited to an object to configure a component with properties and methods. But in Compositions API, we use different hooks to do the same things.

This feature gives us complete control over our code. You can organize your code, however you want.

**Compositions API solves 2 major limitations that Options API had:**

* Group relevant pieces of codes together using hooks.
* Helps to reuse code throughout your application very easily using composable.

1. **Difference in Handling Reactive Data**

To create reactive data we use state in Vue. These APIs use different approaches to create states.

**Options API**

In Options API, you can **declare a method called data() in the main object**. This method always returns an object with the state you need in that particular component.

This is the only place in your component where you can add any types of data you might need.

<script>

export default {

data() {

return {

name: 'Arockia Seelan',

age: 30,

users: ['Eric', 'Erin', 'Seelan',’Churchill’],

}

},

computed: {

details() {

return `${this.name} is ${this.age} years old`;

},

usersList() {

return this.users.join(', ');

} } } </script>

In this example, data() method returns an object with a name, age, and friends list. When you use these values in your template, they will always update automatically whenever their values change.

To add computed properties, you have to add a property with the name computed which will be an object. Inside this computed object, you can add your computed properties.

**Compositions API**

In Composition API, you can **add reactive data using ref() and reactive() functions**. When you call these functions, you just have to provide the initial values.

**Way – I ----Using <script setup> (No need for return)**

<script setup>

import { computed, ref } from 'vue';

const name = ref('Arockia Seelan');

const age = ref(30);

const details = computed(() => `${name.value} is ${age.value} years old`);

const users = ref(['Eric', 'Erin', 'Seelan',’Churchill’]);

const usersList = computed(() => users.value.join(', ')); </script>

**Way - II**

Instead of using <script setup> we can also use <script> tag. If so, we must use write all the data declarations and functions within setup() function and return the variables and functions to use them.

Example:

<script>

import { computed, ref } from 'vue';

export default{

setup(){

let gender = ‘Male’

const name = ref('Arockia Seelan');

const age = ref(30);

const users = ref(['Eric', 'Erin', 'Seelan',’Churchill’]);

const details = computed(() => `${name.value} is ${age.value} years old`);

const usersList = computed(() => users.value.join(', '));

return {

name, age, details, users, userList }

} }

</script>

In this example, we are doing the same thing just using **ref()** and **computed()** functions. The values returned from these functions will be reactive.

**Difference between ref() and reactive()**

* We use **ref()** for any primitive values like string, number, boolean, etc.
* The **reactive()** is used only for objects.

To add computed properties, we have **computed()** function provided by Vue. This computed() function accepts another function as its parameter. This inner function must return a value.

1. **Props in Composition API and Options API**

The prop is the way to pass data from one component to another. It is required if you want to pass external data to a component. With this Vue what values you should pass when you try to use that component.

**Options API**

To declare **props** in the Options API, you need to use the **props property.** This property accepts two types of values, one is an array and another one is an object.

<script>

export default {

**props**: ['color'],

}

</script>

Here, we are telling Vue that this component accepts a prop called "color".

**Composition API**

To declare props in Composition API, we have **defineProps()** function. This function will take an object as a parameter. In this object, you will declare your props.

<script setup>

const props = **defineProps**({

color: String

})

</script>

1. **Emit Custom Events in Composition API and Options API**

There are many browser events like click, change, focus, blur, input, etc. You can make your custom event in a component. When you use that component, you will be able to listen to that event.

**Options API**

You have to use a **property known as emits which will be an array**. This array will contain the list of names that will be emitted from that component.

<script>

export default {

**emits: ['update:name'],**

data() {

return {

name: 'Seeli',

}

},

methods: {

updateName(newName) {

this.name = newName;

**this.$emit('update:name', newName);**

} } } </script>

After declaring **emits property** will the event name, you can create that event anywhere you want. In this case, I am generating the **update:name** event inside **updateName()** method and also passing the new name with the event.

**When this method(updateName()) runs, the event will be emitted as well.**

**Composition API**

You have to use the function called **defineEmits()** to declare any custom event in Composition API. This function will accept the same array that you have seen in the Options API.

<script setup>

const emits = **defineEmits(['update:name']);**

const name = ref('Arulseeli');

const updateName = (newName) => {

name.value = newName;

**emits('update:name', newName);**

}

<script>

Here, I am using **defineEmits()** function with the array of event names. This function will return another function that you will use to emit your custom event.

**Note:** The **defineEmits(**) is available in every component so you don't have to import it from Vue.

You can see it looks a lot simple and clean compared to the Options API.

1. **Methods in Composition API and Options API**

In Vue, we use methods when we want to perform some actions. Methods are mostly used with **v-on** directive on an element to handle browser events.

**Options API**

In Options API, we **add methods using a property called methods** which will be an object. In this property, you can add as many methods as you want.

<script>

export default {

**methods**: {

updateName(newName) {

this.name = newName;

},

updateAge(newAge) {

this.age = newAge;

},

addUser(user) {

this.users.push(user);

},

removeUser(username) {

this.users = this.users.filter(user => user !== username);

}

}

}

</script>

**Composition API**

To declare methods in Composition API, you just have to add plain JavaScript functions. You can use these functions inside your template as methods

<script setup>

const updateName = (newName) => {

name.value = newName;

}

const updateAge = (newAge) => {

age.value = newAge;

}

const addUser = (user) => {

users.value.push(user);

}

const removeUser = (username) => {

users.value = users.value.filter(user => user !== username); } </script>

1. **Watch in Composition API and Options API**

In Vue, watches help us to watch reactive data properties and perform some actions whenever they change.

**Options API**

We can **watch any type of reactive data by adding a property called watch** which will be an object.

<script>

export default {

props: ['color'],

data() {

return {

name: 'Arockia Seelan',

age: 30,

users: ['Eric', 'Erin', 'Churchill'],

}

},

watch: {

name(newName, oldName) {

// Do something when name changes

}

}

}

</script>

Inside the watch object, you have to add methods to watch any reactive data.

**Important Note :**

* The method name will be the same as the reactive data that you want to watch.
* This method automatically gets new and old values of that reactive data.

**Compositions API**

To watch reactive data in Composition API, you have to use **watch() function** imported from Vue.

<script setup>

import { computed, ref, watch } from 'vue';

const name = ref('John Doe');

const age = ref(30);

const details = computed(() => `${name.value} is ${age.value} years old`);

watch(name, (newName, oldName) => {

// Do something when name changes

});

<script>

Here, we are watching the name using watch() function. This function accepts two parameters. The first one is the data you want to watch and the second one is a function that will be called when the data changes.

1. **Lifecycle in Composition API and Options API**

Vue component lifecycle allows us to run code at different stages. For example, you can execute a function right before a component is created, mounted, updated, etc.

**Options API**

In Options API, you have to add some specific methods in order to use component lifecycle in Vue. Vue will execute those methods automatically at different points in time.

<script>

export default {

beforeCreate() {

// Do something before component is created

},

created() {

// Do something when component is created

},

beforeMount() {

// Do something before component is mounted

},

mounted() {

// Do something when component is mounted

},

beforeUpdate() {

// Do something before component is updated

},

updated() {

// Do something when component is updated

},

beforeUnmounted() {

// Do something before component is destroyed

},

unmounted() {

// Do something when component is destroyed

},

}

</script>

Here, you will find all the lifecycle methods available in Options API.

**Composition API**

To add lifecycle in Composition API, we use different hooks provided by Vue. You need to import those hooks before using them.

<script setup>

import { onBeforeMount, onBeforeUnmount, onBeforeUpdate, onMounted, onUnmounted, onUpdated } from 'vue';

onBeforeMount(() => {

// Do something before component is mounted

});

onMounted(() => {

// Do something when component is mounted

});

onBeforeUpdate(() => {

// Do something before component is updated

});

onUpdated(() => {

// Do something when component is updated

});

onBeforeUnmount(() => {

// Do something before component is destroyed

});

onUnmounted(() => {

// Do something when component is destroyed

});

</script>

When you call a lifecycle hook it will accept a function. Vue will automatically execute this function.

**Grouping Codes in Composition API and Options API**

I mentioned earlier that Composition API solves 2 major limitations that Options API has. This is one of them. It is very difficult to organize related codes together in Options API. Because we have to put our code in a specific section to make them work.

For example, if you have two methods one for users another for products, you have to put both of them together inside the methods property ever though they are not related to each other.

If you have two computed properties for users and products, you have to put them together inside the computed property.

So, you can see what is happening. Computed property and method for users are in different sections even though they are related.

**Composition API VS Options API in Vue 3 For Beginners**

In Options API everything is scattered in data(), computed, methods, watch, and lifecycle methods.

It might look simple but imagine you have 100 components like this in your application. It becomes very painful and difficult to handle when you try to modify or add new features.

On the other side, you can see in Compositions API I have grouped all related codes together into 3 sections. The first section works with personal details like name and age.

The second section handles everything about the users. And the third section contains all the lifecycle hooks.

Now If you want to modify or add something to this component you just know where to look. Because everything is so organized and clean.

**Reuse Code in Composition API and Options API**

Sharing code among multiple components was very difficult in Options API. This is another major limitation that Composition API solves.

Now you can easily reuse your code in multiple components using hooks. So you don't have to write the same code in multiple components.

// src/composables/user-list.js

import { computed, ref } from 'vue';

export const useUserList = () => {

const users = ref(['Jane', 'Mark', 'Bob']);

const usersList = computed(() => users.value.join(', '));

const addUser = (user) => {

users.value.push(user);

};

const removeUser = (username) => {

users.value = users.value.filter((user) => user !== username);

};

return {

users,

usersList,

addUser,

removeUser,

};

};

<script setup>

import { useUserList } from '@/composables/user-list';

const { users, usersList, addUser, removeUser } = useUserList();

</script>

You can share your code among multiple components by creating composable. It is a simple function that will return all the necessary data, computed or methods.

Here, I have created useUserList() function that returns user, usersList, addUser(), and removeUser(). Now all I have to do is import this function in any component.

As I am using ref() and computed() in my function, all this data will be reactive. When I add or remove any user all the components that are using this composable function, will update automatically.

Now I can reuse this function again and again without repeating the same code in different places.

**What Makes the Composition API Better than the Options API?**

After you know the differences between Composition API and Options API, you can identify the places where Composition API provides better value.

**Advantages of Composition API**

* It helps to write clean code.
* It makes our code look very simple and easily readable.
* You can effortlessly update or add new features to your application.
* You can organize your component into different sections with the related codes.
* You can create a composable function to reuse your code among different components.