**Differences between Options API and Composition API**

**Options API**

  The Options API is the traditional way of building Vue components, and it has been the primary method for building components in Vue since its inception. It involves using a set of options, such as data, methods, and computed properties, to define the behavior and state of a component.

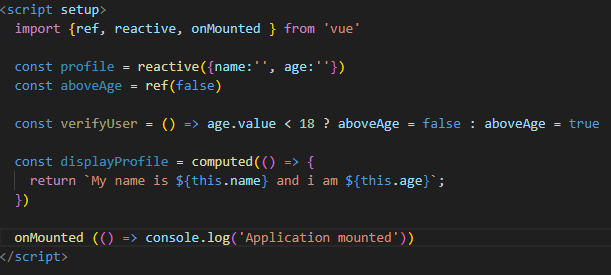


Three of the limitations of Options API which the Composition API tries to solve:

1. Segmented Logical concerns (features) via component options.
2. Flawed reusable component logic with Mixins
3. Limited Typescript support

**Composition API**

On the other hand, the Composition API is a new way of building components in Vue 3.0 that was introduced to address above mentioned limitations of the Options API. It allows developers to use a functional, reactive programming style to build components, and it offers a more flexible and expressive way of defining component behavior.



**Structure**

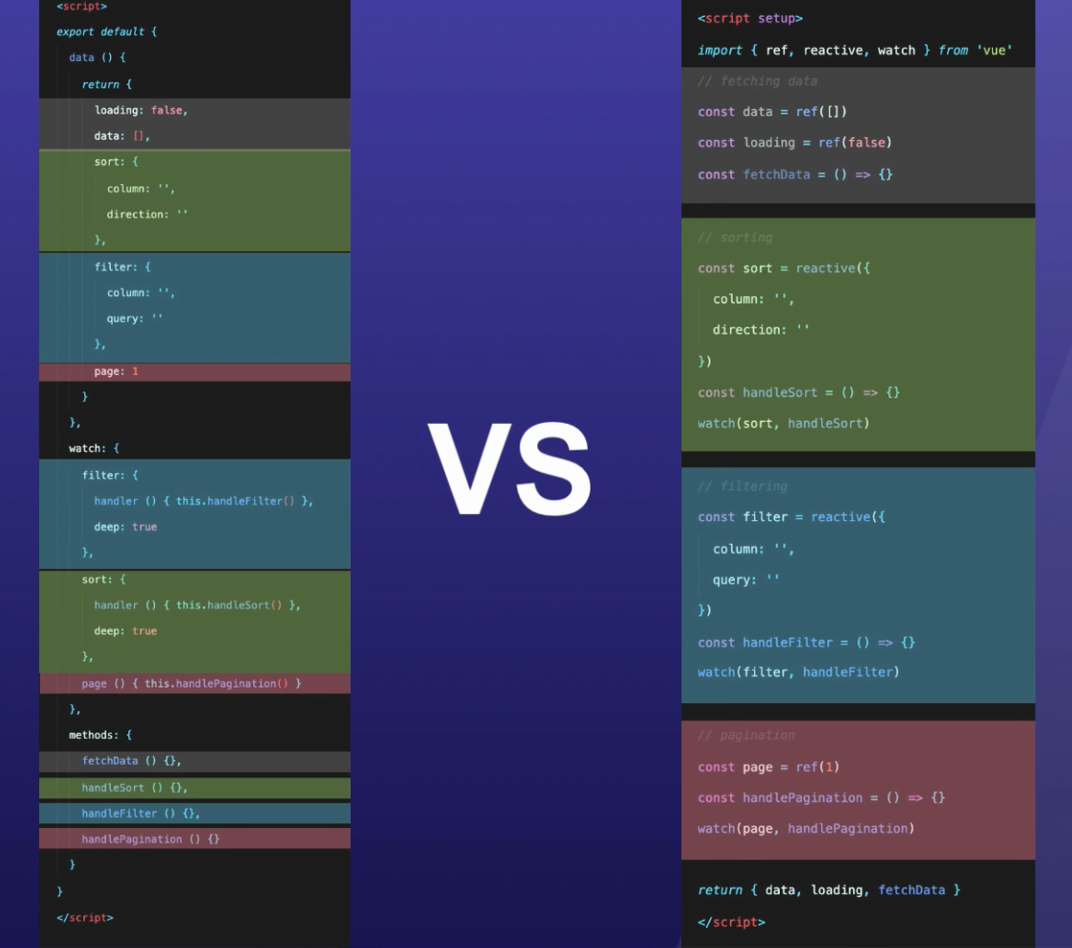
One of the main benefits of the Options API is that it is simple and easy to understand. It follows a clear, declarative pattern that is familiar to many developers, and it is well-documented in the Vue documentation. This makes it a good choice for beginners who are just starting out with Vue.

**Segmented Logical Concerns in Options API:**

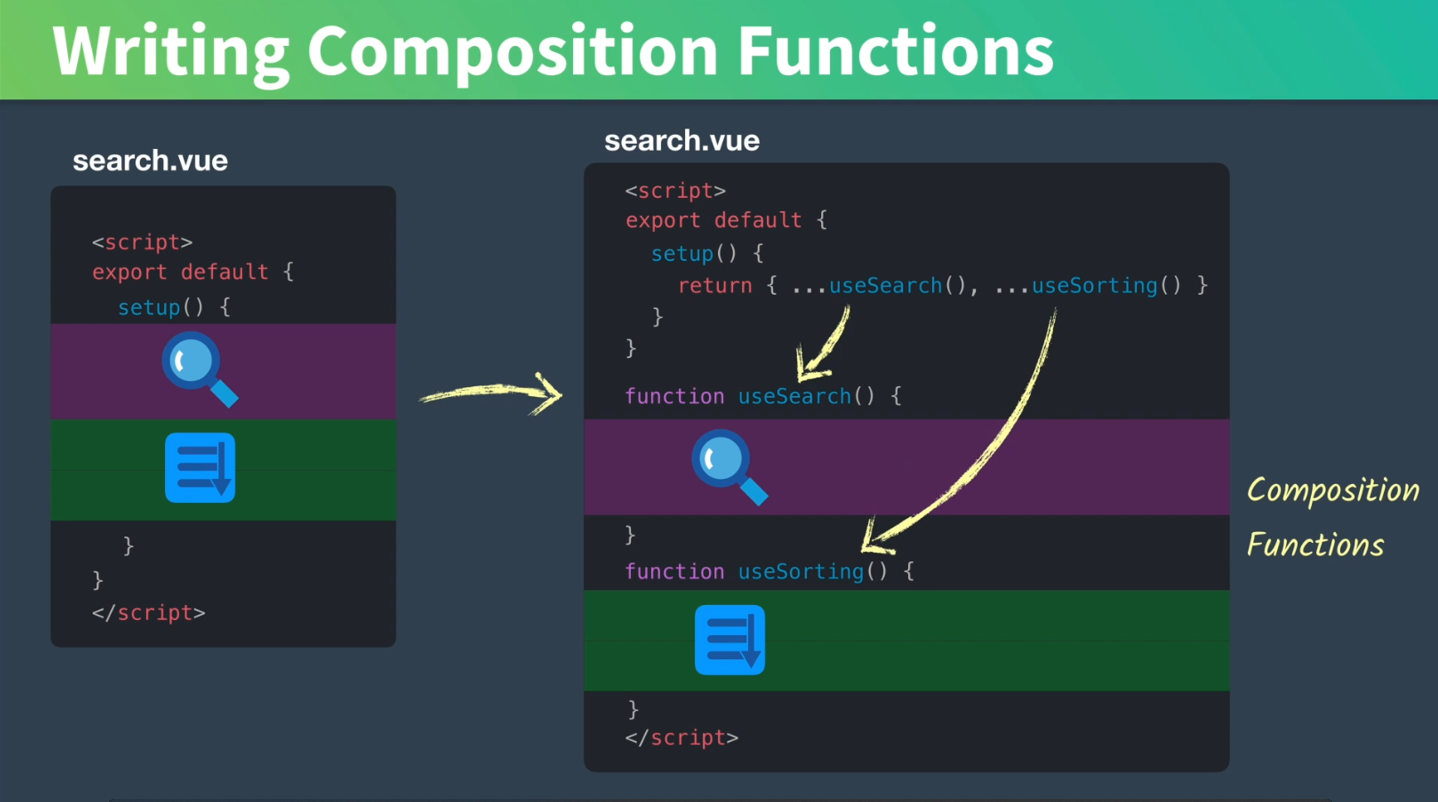
* 1. However, the Options API has some limitations that can make it difficult to use for more complex projects. One issue is that it can become cumbersome to manage a large number of options as the complexity of a component increases. This can lead to a phenomenon known as "option explosion," where a component becomes so large and unwieldy that it is difficult to maintain.

**Grouped Logical Concerns in Compositions API**





Another way of encapsulate the logical concerns together at one place is by using “Composables” or “Composition Functions”



**Reusability**

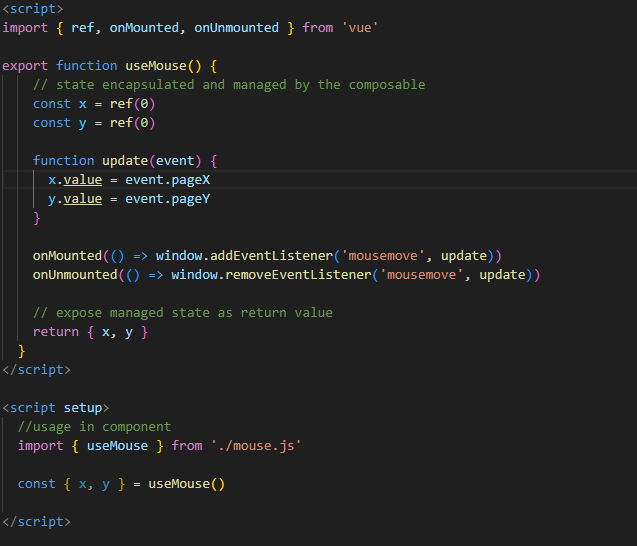
**Options API**

Another limitation of the Options API is that it can be inflexible when it comes to sharing logic between components. If you want to reuse a piece of logic across multiple components, you need to either copy and paste the code or abstract it into a mixin. This can make it difficult to manage code dependencies and maintain code quality as the size of a project grows.



**Compositions API**

The Composition API addresses these issues by providing a more flexible and expressive way of defining component behavior through composables. It allows developers to use a functional, reactive programming style to build components, which makes it easier to reuse logic and manage code dependencies.

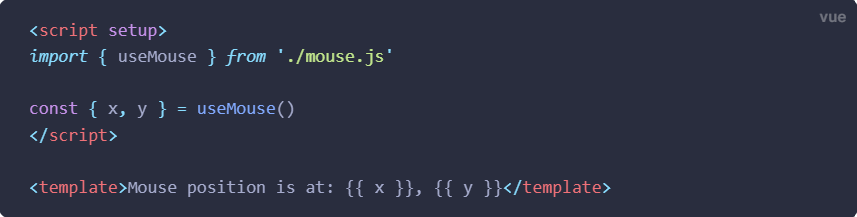


This makes it easier to share logic between components and manage code dependencies as the size of a project grows. It also makes it easier to test and debug code, as the reactive declarative compositions are isolated and easier to reason about.

But what if we want to reuse the same logic in multiple components? We can extract the logic into an external file, as a composable function:

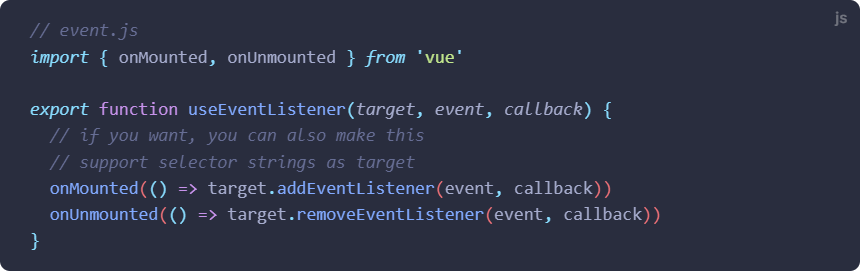


And this is how it can be used in components:



The cooler part about composables though, is that you can also nest them: one composable function can call one or more other composable functions. This enables us to compose complex logic using small, isolated units, similar to how we compose an entire application using components. In fact, the Composition API derives its names from the collection of APIs called composables.

For example, we can extract the logic of adding and removing a DOM event listener into its own composable:



And now our useMouse() composable can be simplified to:



**Usage**

Another superior quality of the Composition API is that it allows developers to use the full power of JavaScript to define component behavior. This makes it possible to use advanced features such as async/await, Promises, and even third-party libraries like RxJS to build components. This can make it easier to build complex, interactive components that would be difficult or impossible to build with the Options API.

**Lifecycle Hooks**

|  |  |
| --- | --- |
| **Options API** | **Composition API**  **(Hooks inside setup)** |
| beforeCreate() | Not needed |
| created() | Not needed |
| beforeMount() | onBeforeMount() |
| mounted() | onMounted() |
| beforeUpate() | onBeforeUpdate() |
| updated() | onUpdated() |
| beforeUnmount() | onBeforeUnmount() |
| unmounted() | onUnmounted() |
| errorCaptured() | onErrorCaptured() |
| renderTracked() | onRenderTracked() |
| renderTriggered() | onRenderTriggered() |
| activated() | onActivated() |
| deactivated() | onDeactivated() |

The hooks’ names have mostly been borrowed from Options API only with the addition of “on-” prefix to them.

**New features**

* setup() function
* Reactivity API
* watchEffect()
* composables

**Learning**

The Composition API may seem like the best options to use. However, the Composition API is not without its drawbacks. One issue is that it can be more difficult to learn for developers who are not familiar with functional, reactive programming. It also has a steeper learning curve than the Options API, which can make it a less appealing choice for beginners.

**Compatibility**

Another issue is that the Composition API is not backwards compatible with Vue 2.6 and under by default. This means that if you are using Vue.js 2.6 and under in an existing project, you will need to either upgrade to Vue 3.0 or import the Composition API via a plugin.

**Bundle Size**

**Smaller Production Bundle and Less Overhead**

Code written in Composition API and <script setup> is also more efficient and minification-friendly than Options API equivalent. This is because the template in a <script setup> component is compiled as a function inlined in the same scope of the <script setup> code. Unlike property access from this, the compiled template code can directly access variables declared inside <script setup>, without an instance proxy in between. This also leads to better minification because all the variable names can be safely shortened.

**Summary:**

|  |  |
| --- | --- |
| **Options API** | **Composition API** |
| Simple and easy to understand | Has a steeper learning curve and demands developers to be familiar with functional, reactive programing |
| Uses a set of options, such as data, methods, and computed properties | Allows developers to use a functional, reactive programming style to build components |
| Inflexible when it comes to sharing logic between components.Generally, mixins is used for this. | More flexible and expressive way of defining component behavior through composables |
| Backwards compatible for any version of Vue, since its present since the  inception of Vue and still continuing | Backwards compatible until 2.7. Older verisons are not compatible |
| Lesser scope of efficient and minifiable code as compared to Composition API | Code written in Composition API is also more efficient and minifiable |

|  |  |  |
| --- | --- | --- |
|  | **Vue Options API** | **Vue Composition API** |
| **Best for** | **Building small and simple, single-feature components requiring low reusability** | **Building medium to complex, multi-featured components requiring higher reusability** |
| **Use in** | **Simple apps or in web sites for progressive enhancement** | **Medium to complex apps with a lot of logic and features** |
| **Learning curve** | **Shallow** | **Steep** |
| **Reactivity** | **Automatic, Hidden, Implicit** | **Manual, Exposed, Explicit** |
| **Reusability approach** | **Mixins** | **Composables** |
| **Flexibility** | **Less** | **More** |
| **Code logic** | **Group by options** | **Group by features** |
| **Bundle size** | **Bigger** | **Smaller** |
| **Supported in** | **All Vue versions** | **Vue 2.7 and Vue 3.0+** |
| **Typescript Support** | **Yes** | **Yes(better)** |