컴포지션 API FAQ

> TIP
>
> 이 FAQ는 Vue에 대한 사전 경험, 특히 Options API를 주로 사용한 Vue 2 경험이 있다고 가정합니다.

컴포지션 API란?

Vue School의 무료 동영상 강의 보기

컴포지션 API는 옵션을 선언하는 대신 가져온 함수를 사용하여 Vue 컴포넌트를 작성할 수 있게 해주는 API 집합입니다. 다음과 같은 API를 포괄하는 상위 개념입니다:

- 반응성 API, 예: `ref()` 와 `reactive()` , 이를 통해 반응형 상태, 계산된 상태, 감시자를 직접 생성할 수 있습니다.
- 생명주기 훅, 예: `onMounted()` 와 `onUnmounted()` , 이를 통해 컴포넌트 생명주기에 프로그래밍적으로 연결할 수 있습니다.
- 의존성 주입, 즉 `provide()` 와 `inject()` , 이를 통해 Reactivity API를 사용하면서 Vue의 의존성 주입 시스템을 활용할 수 있습니다.

컴포지션 API는 Vue 3와 Vue 2.7의 내장 기능입니다. 이전 Vue 2 버전에서는 공식적으로 관리되는 `@vue/composition-api` 플러그인을 사용하세요. Vue 3에서는 주로 Single-File Component에서 `<script setup>` 문법과 함께 사용됩니다. 다음은 컴포지션 API를 사용하는 컴포넌트의 기본 예시입니다:

```vue
<script setup>
import { ref, onMounted } from 'vue'

// 반응형 상태
```

```
// 상태를 변경하고 업데이트를 트리거하는 함수
function increment() {
  count.value++
}

// 생명주기 훅
onMounted(() => {
  console.log(`초기 카운트는 ${count.value}입니다.`)
})
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
</template>
```

함수 조합에 기반한 API 스타일임에도 불구하고, **컴포지션 API는 함수형 프로그래밍이 아닙니다**. 컴포지션 API는 Vue의 변경 가능한, 세밀한 반응성 패러다임에 기반하며, 함수형 프로그래밍은 불변성을 강조합니다.

컴포지션 API로 Vue를 사용하는 방법을 배우고 싶다면, 왼쪽 사이드바 상단의 토글을 사용해 사이트 전체 API 선호도를 컴포지션 API로 설정한 후, 가이드의 처음부터 따라가 보세요.

왜 컴포지션 API인가?

더 나은 로직 재사용

컴포지션 API의 주요 장점은 컴포저블 함수 형태로 깔끔하고 효율적인 로직 재사용이 가능하다는 점입니다. 이는 Options API의 주요 로직 재사용 메커니즘인 믹스인의 모든 단점을 해결합니다.

컴포지션 API의 로직 재사용 기능 덕분에 VueUse와 같은 인상적인 커뮤니티 프로젝트가 탄생했습니다. 이는 컴포저블 유틸리티의 지속적으로 성장하는 모음집입니다. 또한 불변 데이터, 상태 머신, RxJS 등 상태를 가지는 서드파티 서비스나 라이브러리를 Vue의 반응성 시스템에 쉽게 통합할 수 있는 깔끔한 메커니즘을 제공합니다.

더 유연한 코드 구성

많은 사용자는 Options API로 기본적으로 정리된 코드를 작성할 수 있다는 점을 좋아합니다. 모든 것이 해당 옵션에 따라 제자리에 위치합니다. 하지만 Options API는 단일 컴포넌트의 로직이 일정 복잡성 임계값을 넘어서면 심각한 한계를 드러냅니다. 이 한계는 여러 **논리적 관심사**를 다

있습니다.

Vue CLI의 GUI에 있는 폴더 탐색기 컴포넌트를 예로 들어보겠습니다. 이 컴포넌트는 다음과 같은 논리적 관심사를 담당합니다:

현재 폴더 상태를 추적하고 내용을 표시
폴더 탐색 처리(열기, 닫기, 새로고침 등)
새 폴더 생성 처리
즐겨찾기 폴더만 표시 토글
숨김 폴더 표시 토글
현재 작업 디렉터리 변경 처리

이 컴포넌트의 원본 버전은 Options API로 작성되었습니다. 각 코드 줄에 해당 논리적 관심사에 따라 색을 입히면 다음과 같이 보입니다:

동일한 논리적 관심사를 다루는 코드가 서로 다른 옵션 아래로 분리되어 파일의 여러 부분에 위치해야 하는 것을 볼 수 있습니다. 수백 줄에 달하는 컴포넌트에서 단일 논리적 관심사를 이해하고 탐색하려면 파일을 계속 위아래로 스크롤해야 하므로, 생각보다 훨씬 더 어렵습니다. 또한 논리적 관심사를 재사용 가능한 유틸리티로 추출하려고 할 때, 파일의 여러 부분에서 올바른 코드를 찾아 추출하는 데 상당한 노력이 필요합니다.

다음은 컴포지션 API로 리팩터링한 후의 동일한 컴포넌트입니다:

Options API Composition API

이제 동일한 논리적 관심사와 관련된 코드를 함께 그룹화할 수 있음을 알 수 있습니다. 특정 논리적 관심사를 작업할 때 더 이상 서로 다른 옵션 블록을 오갈 필요가 없습니다. 또한 이제 코드를 그룹으로 외부 파일로 옮기는 것도 최소한의 노력으로 가능합니다. 코드를 추출하기 위해 이리저리 옮길 필요가 없기 때문입니다. 이러한 리팩터링의 마찰 감소는 대규모 코드베이스의 장기적인 유지보수성에 핵심적입니다.

더 나은 타입 추론

최근 몇 년간, 더 많은 프론트엔드 개발자들이 TypeScript를 채택하고 있습니다. 이는 더 견고한 코드를 작성하고, 더 자신 있게 변경하며, IDE 지원을 통한 훌륭한 개발 경험을 제공합니다. 하지만 Options API는 2013년에 고안될 당시 타입 추론을 염두에 두지 않았습니다. 우리는 Options API에서 타입 추론이 동작하도록 매우 복잡한 타입 체조를 구현해야 했습니다. 그럼에도 불구하고, 믹스인과 의존성 주입에서는 Options API의 타입 추론이 여전히 깨질 수 있습니다.

이로 인해 Vue를 TS와 함께 사용하고자 하는 많은 개발자들이 `vue-class-component` 기반의 Class API로 기울었습니다. 하지만 클래스 기반 API는 ES 데코레이터에 크게 의존하는데, 이는

식 API를 불안정한 제안에 기반하는 것이 너무 위험하다고 느꼈습니다. 이후 데코레이터 제안은 또 한 번 완전히 개편되었고, 2022년에야 3단계에 도달했습니다. 또한 클래스 기반 API는 Options API와 유사한 로직 재사용 및 코드 구성의 한계를 겪습니다.

이에 비해 컴포지션 API는 대부분 평범한 변수와 함수를 활용하므로 자연스럽게 타입 친화적입니다. 컴포지션 API로 작성된 코드는 거의 수동 타입 힌트 없이도 완전한 타입 추론을 누릴 수 있습니다. 대부분의 경우, 컴포지션 API 코드는 TypeScript와 일반 JavaScript에서 거의 동일하게 보입니다. 이는 일반 JavaScript 사용자도 부분적인 타입 추론의 이점을 누릴 수 있게 해줍니다.

더 작은 프로덕션 번들 및 오버헤드 감소

컴포지션 API와 `<script setup>`으로 작성된 코드는 Options API에 비해 더 효율적이고, 난독화(최소화)에 더 적합합니다. 이는 `<script setup>` 컴포넌트의 템플릿이 `<script setup>` 코드와 동일한 스코프에 인라인된 함수로 컴파일되기 때문입니다. `this`에서 프로퍼티에 접근하는 것과 달리, 컴파일된 템플릿 코드는 인스턴스 프록시 없이 `<script setup>` 내부에 선언된 변수에 직접 접근할 수 있습니다. 또한 모든 변수명이 안전하게 짧아질 수 있으므로 난독화 효과도 더 좋습니다.

Options API와의 관계

트레이드오프

Options API에서 옮겨온 일부 사용자는 컴포지션 API 코드가 덜 정돈되어 있다고 느끼고, 코드 구성 측면에서 컴포지션 API가 "더 나쁘다"고 결론짓기도 합니다. 이런 의견을 가진 사용자라면, 그 문제를 다른 관점에서 바라보길 권장합니다.

컴포지션 API는 더 이상 코드를 각 버킷에 넣도록 안내하는 "가드레일"을 제공하지 않는 것이 사실입니다. 대신, 일반 JavaScript를 작성하듯이 컴포넌트 코드를 작성할 수 있습니다. 즉, **일반 JavaScript를 작성할 때 적용하는 모든 코드 구성 모범 사례를 컴포지션 API 코드에도 적용할 수 있고, 그래야 합니다**. 잘 정돈된 JavaScript를 작성할 수 있다면, 잘 정돈된 컴포지션 API 코드도 작성할 수 있습니다.

Options API는 컴포넌트 코드를 작성할 때 "덜 생각하게" 해주므로 많은 사용자가 이를 좋아합니다. 하지만 정신적 부담을 줄이는 대신, 탈출구 없는 정해진 코드 구성 패턴에 갇히게 되어, 대규모 프로젝트에서 리팩터링이나 코드 품질 개선이 어려워질 수 있습니다. 이런 점에서 컴포지션 API는 장기적으로 더 나은 확장성을 제공합니다.

컴포지션 API가 모든 사용 사례를 포괄하나요?

`props` , `emits` , `name` , `inheritAttrs` 정도입니다.

> ⓘ TIP
>
> 3.3부터는 `<script setup>` 에서 `defineOptions` 를 직접 사용해 컴포넌트 이름이나 `inheritAttrs` 속성을 설정할 수 있습니다.

컴포지션 API(위에 나열된 옵션과 함께)만을 독점적으로 사용하려는 경우, 컴파일 타임 플래그를 통해 Vue에서 Options API 관련 코드를 제거하여 프로덕션 번들 크기를 몇 KB 줄일 수 있습니다. 이 설정은 의존성에 있는 Vue 컴포넌트에도 영향을 미칩니다.

두 API를 같은 컴포넌트에서 함께 사용할 수 있나요?

네. Options API 컴포넌트에서 `setup()` 옵션을 통해 컴포지션 API를 사용할 수 있습니다.

하지만 기존 Options API 코드베이스에 컴포지션 API로 작성된 새로운 기능/외부 라이브러리를 통합해야 할 때만 이를 권장합니다.

Options API가 폐지될 예정인가요?

아니요, 그럴 계획이 전혀 없습니다. Options API는 Vue의 핵심적인 부분이며, 많은 개발자가 이를 사랑하는 이유이기도 합니다. 또한 컴포지션 API의 많은 이점은 대규모 프로젝트에서만 두드러지므로, Options API는 여전히 저~중간 복잡도 시나리오에서 훌륭한 선택지로 남아 있습니다.

Class API와의 관계

컴포지션 API가 타입스크립트 통합, 추가적인 로직 재사용 및 코드 구성의 이점을 제공하므로, Vue 3에서는 더 이상 Class API 사용을 권장하지 않습니다.

React Hooks와의 비교

컴포지션 API는 React Hooks와 동일한 수준의 로직 조합 기능을 제공하지만, 몇 가지 중요한 차이점이 있습니다.

발자조차 혼란스러울 수 있는 여러 주의사항을 만듭니다. 또한 개발 경험에 심각한 영향을 줄 수 있는 성능 최적화 문제로 이어집니다. 예시는 다음과 같습니다:

> 훅은 호출 순서에 민감하며 조건부로 사용할 수 없습니다.
>
> React 컴포넌트에서 선언된 변수는 훅 클로저에 캡처되어, 개발자가 올바른 의존성 배열을 전달하지 않으면 "오래된(stale)" 값이 될 수 있습니다. 이로 인해 React 개발자는 올바른 의존성이 전달되었는지 확인하기 위해 ESLint 규칙에 의존하게 됩니다. 하지만 이 규칙은 종종 충분히 똑똑하지 않아, 정확성을 과도하게 보장하려다 불필요한 무효화와 예외 상황에서의 골칫거리를 유발합니다.
>
> 비용이 많이 드는 계산에는 `useMemo` 를 사용해야 하며, 이 역시 올바른 의존성 배열을 수동으로 전달해야 합니다.
>
> 자식 컴포넌트에 전달되는 이벤트 핸들러는 기본적으로 불필요한 자식 업데이트를 유발하며, 최적화를 위해 명시적으로 `useCallback` 을 사용해야 합니다. 이는 거의 항상 필요하며, 역시 올바른 의존성 배열이 필요합니다. 이를 소홀히 하면 앱이 과도하게 렌더링되어 성능 문제가 발생할 수 있습니다.
>
> 오래된 클로저 문제와 Concurrent 기능이 결합되면, 훅 코드가 언제 실행되는지 추론하기 어려워지고, 렌더 간에 유지되어야 하는 변경 가능한 상태(`useRef` 사용)가 번거로워집니다.

> 참고: 위의 메모이제이션 관련 문제 중 일부는 곧 출시될 React Compiler로 해결될 수 있습니다.

이에 비해 Vue 컴포지션 API는:

> `setup()` 또는 `<script setup>` 코드를 한 번만 호출합니다. 이로 인해 오래된 클로저를 걱정할 필요가 없으므로, 코드가 관용적 JavaScript 사용 직관과 더 잘 맞습니다. 컴포지션 API 호출은 호출 순서에 민감하지 않으며, 조건부로 사용할 수 있습니다.
>
> Vue의 런타임 반응성 시스템은 계산 속성과 감시자에서 사용된 반응형 의존성을 자동으로 수집하므로, 수동으로 의존성을 선언할 필요가 없습니다.
>
> 불필요한 자식 업데이트를 방지하기 위해 콜백 함수를 수동으로 캐시할 필요가 없습니다. 일반적으로 Vue의 세밀한 반응성 시스템은 자식 컴포넌트가 필요할 때만 업데이트되도록 보장합니다. 수동 자식 업데이트 최적화는 Vue 개발자에게 거의 걱정거리가 아닙니다.

우리는 React Hooks의 창의성을 인정하며, 이는 컴포지션 API의 주요 영감 중 하나입니다. 하지만 위에서 언급한 문제들은 실제로 존재하며, Vue의 반응성 모델이 이를 우회할 방법을 제공한다는 점을 발견했습니다.