컴포저블(Composables)

> TIP
>
> 이 섹션은 Composition API에 대한 기본 지식을 전제로 합니다. 만약 Options API만으로 Vue를 학습해왔다면, 왼쪽 사이드바 상단의 토글을 사용해 API Preference를 Composition API로 설정하고 반응성의 기초 및 생명주기 훅 챕터를 다시 읽어보세요.

"컴포저블"이란?

Vue 애플리케이션의 맥락에서 "컴포저블"이란 Vue의 Composition API를 활용하여 **상태를 가진 로직**을 캡슐화하고 재사용하는 함수입니다.

프론트엔드 애플리케이션을 개발할 때, 우리는 종종 공통 작업을 위한 로직을 재사용해야 합니다. 예를 들어, 여러 곳에서 날짜를 포맷해야 할 때, 이를 위한 재사용 가능한 함수를 추출할 수 있습니다. 이 포매터 함수는 **상태가 없는 로직**을 캡슐화합니다: 입력을 받아 즉시 예상되는 출력을 반환합니다. 상태가 없는 로직을 재사용하기 위한 라이브러리는 lodash나 date-fns처럼 많이 존재합니다.

반면, 상태를 가진 로직은 시간이 지남에 따라 변하는 상태를 관리하는 것을 포함합니다. 간단한 예로는 페이지에서 마우스의 현재 위치를 추적하는 것이 있습니다. 실제 시나리오에서는 터치 제스처나 데이터베이스 연결 상태와 같은 더 복잡한 로직일 수도 있습니다.

마우스 트래커 예제

만약 Composition API를 사용하여 마우스 추적 기능을 컴포넌트 내부에 직접 구현한다면, 다음과 같이 작성할 수 있습니다:

MouseComponent.vue

```
import { ref, onMounted, onUnmounted } from 'vue'

const x = ref(0)
const y = ref(0)

function update(event) {
  x.value = event.pageX
  y.value = event.pageY
}

onMounted(() => window.addEventListener('mousemove', update))
onUnmounted(() => window.removeEventListener('mousemove', update))
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

하지만 동일한 로직을 여러 컴포넌트에서 재사용하고 싶다면, 로직을 외부 파일의 컴포저블 함수로 추출할 수 있습니다:

mouse.js

```js
import { ref, onMounted, onUnmounted } from 'vue'

// 관례상, 컴포저블 함수 이름은 "use"로 시작합니다.
export function useMouse() {
  // 컴포저블이 캡슐화하고 관리하는 상태
  const x = ref(0)
  const y = ref(0)

  // 컴포저블은 시간이 지남에 따라 관리하는 상태를 업데이트할 수 있습니다.
  function update(event) {
    x.value = event.pageX
    y.value = event.pageY
  }

  // 컴포저블은 소유 컴포넌트의
  // 생명주기에 훅을 걸어 부수 효과를 설정 및 해제할 수 있습니다.
  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))

  // 관리하는 상태를 반환값으로 노출
  return { x, y }
}
```

그리고 컴포넌트에서는 다음과 같이 사용할 수 있습니다:

MouseComponent.vue

```vue
<script setup>
import { useMouse } from './mouse.js'
```

```
</script>

<template>Mouse position is at: {{ x }}, {{ y }}</template>
```

Mouse position is at: 0, 0

Playground에서 직접 실행해보기

보시다시피, 핵심 로직은 동일하게 유지됩니다. 단지 외부 함수로 옮기고 노출할 상태를 반환하기만 하면 됩니다. 컴포넌트 내부에서와 마찬가지로, 컴포저블에서도 Composition API 함수를 모두 사용할 수 있습니다. 이제 동일한 `useMouse()` 기능을 어떤 컴포넌트에서도 사용할 수 있습니다.

컴포저블의 더 멋진 점은, 컴포저블을 중첩할 수도 있다는 것입니다: 하나의 컴포저블 함수가 하나 이상의 다른 컴포저블 함수를 호출할 수 있습니다. 이를 통해 작은, 독립적인 단위로 복잡한 로직을 조합할 수 있으며, 이는 전체 애플리케이션을 컴포넌트로 조합하는 방식과 유사합니다. 사실, 이러한 패턴을 가능하게 하는 API 모음을 Composition API라고 부르게 된 이유이기도 합니다.

예를 들어, DOM 이벤트 리스너를 추가하고 제거하는 로직을 별도의 컴포저블로 추출할 수 있습니다:

event.js

```js
import { onMounted, onUnmounted } from 'vue'

export function useEventListener(target, event, callback) {
  // 원한다면, target을
  // 셀렉터 문자열로도 지원할 수 있습니다.
  onMounted(() => target.addEventListener(event, callback))
  onUnmounted(() => target.removeEventListener(event, callback))
}
```

이제 우리의 `useMouse()` 컴포저블은 다음과 같이 더 간단해질 수 있습니다:

mouse.js

```js
import { ref } from 'vue'
import { useEventListener } from './event'

export function useMouse() {
  const x = ref(0)
  const y = ref(0)

  useEventListener(window, 'mousemove', (event) => {
    x.value = event.pageX
    y.value = event.pageY
```

```
  return { x, y }
}
```

> ⓘ TIP
>
> 각 컴포넌트 인스턴스가 `useMouse()` 를 호출하면 `x` 와 `y` 상태의 자체 복사본이 생성되므로 서로 간섭하지 않습니다. 컴포넌트 간에 상태를 공유하고 싶다면 상태 관리 챕터를 읽어보세요.

비동기 상태 예제

`useMouse()` 컴포저블은 인자를 받지 않으므로, 인자를 사용하는 또 다른 예제를 살펴보겠습니다. 비동기 데이터 패칭을 할 때는 로딩, 성공, 에러 등 다양한 상태를 처리해야 합니다:

vue

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

const data = ref(null)
const error = ref(null)

fetch('...')
  .then((res) => res.json())
  .then((json) => (data.value = json))
  .catch((err) => (error.value = err))
</script>

<template>
  <div v-if="error">Oops! Error encountered: {{ error.message }}</div>
  <div v-else-if="data">
    Data loaded:
    <pre>{{ data }}</pre>
  </div>
  <div v-else>Loading...</div>
</template>
```

이 패턴을 데이터를 패칭해야 하는 모든 컴포넌트에서 반복하는 것은 번거롭습니다. 이를 컴포저블로 추출해봅시다:

fetch.js

js

```
import { ref } from 'vue'

export function useFetch(url) {
```

```
  const error = ref(null)

  fetch(url)
    .then((res) => res.json())
    .then((json) => (data.value = json))
    .catch((err) => (error.value = err))

  return { data, error }
}
```

이제 컴포넌트에서는 다음과 같이 간단히 사용할 수 있습니다:

vue

```
<script setup>
import { useFetch } from './fetch.js'

const { data, error } = useFetch('...')
</script>
```

반응형 상태 받기

`useFetch()` 는 정적인 URL 문자열을 입력으로 받으므로, 한 번만 패칭을 수행하고 끝납니다. 만약 URL이 변경될 때마다 다시 패칭하고 싶다면 어떻게 해야 할까요? 이를 위해서는 반응형 상태를 컴포저블 함수에 전달하고, 컴포저블이 전달받은 상태를 사용해 동작을 수행하는 watcher를 생성해야 합니다.

예를 들어, `useFetch()` 는 ref를 받을 수 있어야 합니다:

js

```
const url = ref('/initial-url')

const { data, error } = useFetch(url)

// 이 코드는 다시 패칭을 트리거해야 합니다.
url.value = '/new-url'
```

또는, getter 함수를 받을 수도 있습니다:

js

```
// props.id가 변경될 때마다 다시 패칭
const { data, error } = useFetch(() => `/posts/${props.id}`)
```

기존 구현을 `watchEffect()` 와 `toValue()` API로 리팩터링할 수 있습니다:

fetch.js

js

```
import { ref, watchEffect, toValue } from 'vue'
```

```
  const data = ref(null)
  const error = ref(null)

  const fetchData = () => {
    // 패칭 전에 상태를 초기화합니다..
    data.value = null
    error.value = null

    fetch(toValue(url))
      .then((res) => res.json())
      .then((json) => (data.value = json))
      .catch((err) => (error.value = err))
  }

  watchEffect(() => {
    fetchData()
  })

  return { data, error }
}
```

`toValue()` 는 3.3에 추가된 API입니다. ref나 getter를 값으로 정규화하기 위해 설계되었습니다. 인자가 ref라면 ref의 값을 반환하고, 함수라면 함수를 호출해 반환값을 반환합니다. 그렇지 않으면 인자를 그대로 반환합니다. `unref()` 와 유사하지만, 함수에 대해 특별히 처리합니다.

`toValue(url)` 이 **watchEffect 콜백 내부**에서 호출된다는 점에 주목하세요. 이렇게 하면 `toValue()` 정규화 과정에서 접근한 모든 반응형 의존성이 watcher에 의해 추적됩니다.

이 버전의 `useFetch()` 는 이제 정적 URL 문자열, ref, getter를 모두 받아들일 수 있어 훨씬 유연해졌습니다. watch effect는 즉시 실행되며, `toValue(url)` 에서 접근한 모든 의존성을 추적합니다. 만약 추적된 의존성이 없다면(예: url이 이미 문자열인 경우), effect는 한 번만 실행되고, 그렇지 않으면 추적된 의존성이 변경될 때마다 다시 실행됩니다.

▶ 업데이트된 `useFetch()` 버전는 데모를 위해 인위적인 지연과 무작위 에러가 추가되어 있습니다.

관례와 모범 사례

네이밍

컴포저블 함수는 camelCase로 작성하며 "use"로 시작하는 것이 관례입니다.

입력 인자

다. 다른 개발자가 사용할 수 있는 컴포저블을 작성할 때는 입력 인자가 원시 값이 아닌 ref나 getter일 수도 있음을 처리하는 것이 좋습니다. 이를 위해 `toValue()` 유틸리티 함수를 사용할 수 있습니다:

```js
import { toValue } from 'vue'

function useFeature(maybeRefOrGetter) {
  // maybeRefOrGetter가 ref나 getter라면,
  // 정규화된 값이 반환됩니다.
  // 그렇지 않으면 그대로 반환됩니다.
  const value = toValue(maybeRefOrGetter)
}
```

입력값이 ref나 getter일 때 컴포저블이 반응형 효과를 생성한다면, 반드시 `watch()` 로 ref/getter를 명시적으로 감시하거나, `watchEffect()` 내부에서 `toValue()` 를 호출하여 제대로 추적되도록 하세요.

앞서 논의한 useFetch() 구현은 ref, getter, 일반 값을 모두 입력 인자로 받는 컴포저블의 구체적인 예시입니다.

반환값

지금까지 컴포저블에서 `reactive()` 대신 `ref()` 만을 사용해왔다는 점을 눈치채셨을 것입니다. 권장되는 관례는 컴포저블이 여러 ref를 포함하는 평범한, 비반응형 객체를 항상 반환하는 것입니다. 이렇게 하면 컴포넌트에서 구조 분해 할당을 하더라도 반응성이 유지됩니다:

```js
// x와 y는 ref입니다.
const { x, y } = useMouse()
```

컴포저블에서 반응형 객체를 반환하면 구조 분해 할당 시 컴포저블 내부 상태와의 반응성 연결이 끊기지만, ref는 그 연결을 유지합니다.

컴포저블에서 반환된 상태를 객체 속성으로 사용하고 싶다면, 반환 객체를 `reactive()` 로 감싸 ref가 언랩되도록 할 수 있습니다. 예를 들어:

```js
const mouse = reactive(useMouse())
// mouse.x는 원래 ref와 연결되어 있습니다.
console.log(mouse.x)
```

```template
Mouse position is at: {{ mouse.x }}, {{ mouse.y }}
```

컴포저블에서 부수 효과(예: DOM 이벤트 리스너 추가, 데이터 패칭)를 수행해도 괜찮지만, 다음 규칙을 유의하세요:

서버 사이드 렌더링 (SSR)을 사용하는 애플리케이션이라면, DOM 관련 부수 효과는 반드시 post-mount 생명주기 훅(예: `onMounted()`)에서 수행하세요. 이 훅들은 브라우저에서만 호출되므로, 내부 코드가 DOM에 접근할 수 있음을 보장합니다.

`onUnmounted()` 에서 부수 효과를 반드시 정리하세요. 예를 들어, 컴포저블이 DOM 이벤트 리스너를 설정했다면, `useMouse()` 예제에서처럼 `onUnmounted()` 에서 해당 리스너를 제거해야 합니다. `useEventListener()` 예제처럼 자동으로 정리해주는 컴포저블을 사용하는 것도 좋은 방법입니다.

사용 제한

컴포저블은 `<script setup>` 또는 `setup()` 훅에서만 호출해야 합니다. 또한 이들 컨텍스트에서 **동기적으로** 호출해야 합니다. 경우에 따라 `onMounted()` 와 같은 생명주기 훅에서도 호출할 수 있습니다.

이러한 제한은 Vue가 현재 활성 컴포넌트 인스턴스를 결정할 수 있는 컨텍스트이기 때문에 중요합니다. 활성 컴포넌트 인스턴스에 접근해야 하는 이유는 다음과 같습니다:

1. 생명주기 훅을 등록할 수 있습니다.
2. 계산 속성과 watcher를 인스턴스에 연결하여, 인스턴스가 언마운트될 때 이들이 해제되어 메모리 누수를 방지할 수 있습니다.

> ⓘ TIP
>
> `<script setup>` 은 `await` 사용 **이후**에도 컴포저블을 호출할 수 있는 유일한 곳입니다. 컴파일러가 비동기 작업 이후에도 활성 인스턴스 컨텍스트를 자동으로 복원해줍니다.

코드 조직화를 위한 컴포저블 추출

컴포저블은 재사용뿐만 아니라 코드 조직화를 위해서도 추출할 수 있습니다. 컴포넌트의 복잡성이 커지면, 너무 커져서 탐색하거나 이해하기 어려운 컴포넌트가 생길 수 있습니다.
Composition API는 논리적 관심사에 따라 컴포넌트 코드를 더 작은 함수로 자유롭게 조직할 수 있게 해줍니다:

```
import { useFeatureA } from './featureA.js'
import { useFeatureB } from './featureB.js'
import { useFeatureC } from './featureC.js'

const { foo, bar } = useFeatureA()
const { baz } = useFeatureB(foo)
const { qux } = useFeatureC(baz)
</script>
```

어느 정도까지는, 이렇게 추출된 컴포저블을 서로 통신할 수 있는 컴포넌트 범위의 서비스로 생각할 수 있습니다.

Options API에서 컴포저블 사용하기

Options API를 사용하는 경우, 컴포저블은 반드시 `setup()` 내부에서 호출해야 하며, 반환된 바인딩은 `setup()` 에서 반환되어야 `this` 와 템플릿에서 노출됩니다:

```js
import { useMouse } from './mouse.js'
import { useFetch } from './fetch.js'

export default {
  setup() {
    const { x, y } = useMouse()
    const { data, error } = useFetch('...')
    return { x, y, data, error }
  },
  mounted() {
    // setup()에서 노출된 속성은 `this`에서 접근할 수 있습니다.
    console.log(this.x)
  }
  // ...기타 옵션
}
```

다른 기법과의 비교

믹스인과의 비교

Vue 2에서 온 사용자라면 믹스인 옵션에 익숙할 수 있습니다. 믹스인 역시 컴포넌트 로직을 재사용 가능한 단위로 추출할 수 있게 해줍니다. 하지만 믹스인에는 세 가지 주요 단점이 있습니다:

주입되었는지 불분명해져 구현을 추적하고 컴포넌트의 동작을 이해하기 어려워집니다. 이것이 컴포저블에서 ref + 구조 분해 패턴을 권장하는 이유이기도 합니다: 소비하는 컴포넌트에서 속성의 출처가 명확해집니다.

2. **네임스페이스 충돌**: 서로 다른 작성자의 여러 믹스인이 동일한 속성 키를 등록할 수 있어 네임스페이스 충돌이 발생할 수 있습니다. 컴포저블에서는 서로 다른 컴포저블에서 충돌하는 키가 있을 경우 구조 분해 변수명을 변경할 수 있습니다.

3. **암묵적 믹스인 간 통신**: 서로 상호작용해야 하는 여러 믹스인은 공유 속성 키에 의존해야 하므로 암묵적으로 결합됩니다. 컴포저블에서는 한 컴포저블에서 반환된 값을 인자로 다른 컴포저블에 전달할 수 있으므로, 일반 함수처럼 명시적으로 연결할 수 있습니다.

이러한 이유로, Vue 3에서는 믹스인 사용을 더 이상 권장하지 않습니다. 이 기능은 마이그레이션 및 익숙함을 위해서만 유지됩니다.

렌더리스 컴포넌트와의 비교

컴포넌트 슬롯 챕터에서, 스코프 슬롯을 기반으로 한 렌더리스 컴포넌트 패턴을 논의했습니다. 마우스 추적 데모도 렌더리스 컴포넌트로 구현한 바 있습니다.

컴포저블이 렌더리스 컴포넌트보다 가지는 주요 이점은, 컴포저블은 추가적인 컴포넌트 인스턴스 오버헤드가 없다는 점입니다. 전체 애플리케이션에서 렌더리스 컴포넌트 패턴을 사용할 경우, 생성되는 추가 컴포넌트 인스턴스의 양이 성능 오버헤드로 이어질 수 있습니다.

순수 로직을 재사용할 때는 컴포저블을, 로직과 시각적 레이아웃을 모두 재사용할 때는 컴포넌트를 사용하는 것이 권장됩니다.

React 훅과의 비교

React 경험이 있다면, 이 패턴이 커스텀 React 훅과 매우 유사하다는 것을 알 수 있습니다. Composition API는 부분적으로 React 훅에서 영감을 받았으며, Vue 컴포저블은 로직 조합 능력 측면에서 React 훅과 매우 유사합니다. 하지만 Vue 컴포저블은 Vue의 세밀한 반응성 시스템을 기반으로 하며, 이는 React 훅의 실행 모델과 근본적으로 다릅니다. 이에 대한 자세한 내용은 Composition API FAQ에서 다룹니다.

추가 읽을거리

반응성 심층 분석: Vue의 반응성 시스템이 어떻게 동작하는지 저수준에서 이해할 수 있습니다.

컴포저블 테스트하기: 컴포저블의 단위 테스트 팁을 제공합니다.

VueUse: 계속 성장하는 Vue 컴포저블 모음집입니다. 소스 코드도 훌륭한 학습 자료입니다.

GitHub에서 이 페이지 편집

Previous
비동기 컴포넌트

Next
커스텀 디렉티브