컴포저블

TIP

이 섹션에서는 컴포지션 API 에 대한 기본 지식이 있다고 가정합니다. 옵션 API 만 배웠다면 왼쪽 사이드바 상단의 API 스타일 설정을 컴포지션 API 로 설정하고 반응형 기초 및 생명주기 훅 장을 다시 읽을 수 있습니다.

컴포저블이란?

Vue 앱의 컨텍스트에서 **컴포저블**은 Vue 컴포지션 API를 활용하여 **상태 저장 로직**를 캡슐화하고 재사용하는 함수입니다.

프론트엔드 앱을 구축할 때, 여러 곳에서 효율적으로 같은 작업을 하기 위해 로직을 재사용해야 하는 경우가 종종 있습니다. 예를 들어 여러 위치에서 날짜 형식을 지정해야 할 수 있고, 이를 위해 재사용 가능한 함수를 생성합니다. 이 함수는 상태 비저장 로직을 캡슐화합니다. 일부 입력을 받고 즉시 계산된 값을 반환합니다. 상태 비저장 로직을 재사용하기 위한 많은 라이브러리가 있습니다. 예를 들어 **lodash** 나 **date-fns**는 들어본 적이 있을 것입니다.

이에 비해 상태 저장 로직은 시간이 지남에 따라 변경되는 상태 관리가 포함됩니다. 간단한 예는 페이지에서 마우스의 현재 위치를 추적하는 것입니다. 실제 시나리오에서는 터치 제스처 또는 데이터베이스 연결 상태와 같은 더 복잡한 로직이 될 수도 있습니다.

마우스 위치 추적기 예제

컴포넌트 내에서 직접 컴포지션 API를 사용하여 마우스 추적 기능은 다음과 같이 구현할 수 있습니다:

vue

```vue
<script setup>
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>마우스 위치: {{ x }}, {{ y }}</template>
```

그러나 여러 컴포넌트에서 동일한 로직을 재사용하려면 어떻게 해야 할까요? 로직을 컴포저블 함수로 재구성 후 외부 파일로 추출할 수 있습니다:

js

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

// 관례상, 컴포저블 함수 이름은 "use"로 시작합니다.
export function useMouse() {
  // 컴포저블로 캡슐화된 내부에서 관리되는 상태
  const x = ref(0)
  const y = ref(0)
```

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

  // 컴포저블은 또한 이것을 사용하는 컴포넌트의 생명주기에 연결되어
  // 사이드 이펙트를 설정 및 해제할 수 있습니다.
  onMounted(() => window.addEventListener('mousemove', update))
  onUnmounted(() => window.removeEventListener('mousemove', update))

  // 관리 상태를 반환 값으로 노출
  return { x, y }
}
```

그리고 이것이 컴포넌트에서 사용되는 방법입니다:

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

const { x, y } = useMouse()
</script>

<template>마우스 위치: {{ x }}, {{ y }}</template>
```

마우스 위치: 455, 196

온라인 연습장으로 실행하기

보다시피, 핵심 로직은 정확히 동일합니다. 조치한 것은 그것을 외부 함수로 옮기고, 노출되어야 하는 상태를 반환하는 것뿐이었습니다. 컴포넌트 내부처럼 컴포저블에서는 컴포지션 **API** 함수의 전체를 사용할 수 있습니다. `useMouse()` 함수는 이제 모든 컴포넌트에서 사용할 수 있습니다.

하지만 컴포저블의 멋진 부분은 중첩할 수도 있다는 것입니다. 하나의 컴포저블 함수는 하나 이상의 "다른 컴포저블 함수"를 호출할 수 있습니다. 이를 통해 컴포넌트를 사용하여 전체 앱을 구성하는 방법과 유사하게 작게 독립된 단위를 사용하여 복잡한 로직을 구성할 수 있습니다. 사실 이 패턴을 가능하게 하는 API 모음을 "**컴포지션 API**"라고 부르기로 결정한 이유입니다.

예를 들어 DOM 이벤트 리스너를 자체 컴포저블에 추가하고 정리하는 로직을 추출할 수 있습니다:

```js
// event.js
import { onMounted, onUnmounted } from 'vue'

export function useEventListener(target, event, callback) {
  // if you want, you can also make this
  // support selector strings as target
  onMounted(() => target.addEventListener(event, callback))
  onUnmounted(() => target.removeEventListener(event, callback))
}
```

이제 `useMouse()` 를 다음과 같이 단순화할 수 있습니다:

```js
// mouse.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

`useMouse()` 를 호출하는 각 컴포넌트 인스턴스는 서로 간섭하지 않도록 `x` 와 `y` 상태의 자체 복사본을 생성합니다. 컴포넌트 간의 공유 상태를 관리하려면 상태 관리 문서를 읽으십시오.

비동기 상태 예제

컴포저블 함수 `useMouse()` 는 인자를 사용하지 않으므로, 인자를 사용하는 다른 예를 살펴보겠습니다. 비동기 데이터 가져오기를 수행할 때 로드, 성공 및 에러와 같은 다양한 상태를 처리해야 하는 경우가 많습니다:

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">앗! 에러 발생: {{ error.message }}</div>
  <div v-else-if="data">
    로드된 데이터:
    <pre>{{ data }}</pre>
  </div>
  <div v-else>로딩...</div>
</template>
```

다시 말하지만 데이터를 가져와야 하는 모든 컴포넌트에서 이 패턴을 반복해야 한다면 끔찍할 것입니다. 추출해서 컴포저블로 재구성 해 보겠습니다:

js

```
// fetch.js
import { ref } from 'vue'

export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)

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

  return { data, error }
}
```

이제 컴포넌트에서 다음을 수행할 수 있습니다:

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

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

리액티브 상태 수락

useFetch() 는 정적 URL 문자열을 입력으로 받아 한 번만 fetch를 수행하고 끝납니다. 그런데 URL이 변경될 때마다 다시 fetch를 수행하려면 어떻게 해야 할까요? 이를 달성하려면 리액티브 상태를 composable 함수에 전달해야 하며, composable이 전달된 상태를 사용하여 작업을 수행하는 watcher를 생성합니다.

예를 들어, useFetch() 는 ref 를 받아들일 수 있어야 합니다:

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

const { data, error } = useFetch(url)

// re-fetch를 트리거 합니다.
url.value = '/new-url'
```

또는 **getter** 함수를 받아들입니다:

```js
// props.id가 변경되면 re-fetch 됩니다.
const { data, error } = useFetch(() => `/posts/${props.id}`)
```

우리는 **watchEffect()** 와 **toValue()** API를 이용하여 기존의 구현을 리팩토링 할 수 있습니다:

```js
// fetch.js
import { ref, watchEffect, toValue } from 'vue'

export function useFetch(url) {
  const data = ref(null)
  const error = ref(null)

  const fetchData = () => {
    // reset state before fetching..
    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 3.3 버전에서 추가된 API로, ref나 getter를 값으로 정규화하는 데 사용됩니다. 인자가 ref인 경우 ref의 값을 반환하고, 인자가 함수인 경우 함수를 호출하고 그 반환값을 반환합니다. 그 외의 경우에는 인자를 그대로 반환합니다. 이는 **unref()** 와 유사하게 작동하지만, 함수에 대한 특별한 처리가 있습니다.

toValue(url) 이 watchEffect 콜백 내부에서 호출되는 것에 주목하세요. 이는 toValue() 정규화 과정 중에 접근한 모든 리액티브 의존성이 watcher에 의해 추적되도록 보장합니다.

이 버전의 `useFetch()` 는 이제 정적 URL 문자열, ref, getter를 받아들이므로 훨씬 더 유연합니다. watch 효과는 즉시 실행되며, `toValue(url)` 중에 접근한 모든 의존성을 추적합니다. 추적된 의존성이 없는 경우 (예: url이 이미 문자열인 경우) 효과는 한 번만 실행됩니다; 그렇지 않으면 추적된 의존성이 변경될 때마다 다시 실행됩니다.

다음은 인공적인 지연과 데모 목적으로 무작위 오류가 있는 **useFetch()**의 업데이트 된 버전입니다.

관례와 모범 사례

작명

"use"로 시작하는 camelCase 이름으로 컴포저블 함수의 이름을 지정하는 것이 관례입니다.

입력 인자

composable은 리액티비티에 따라 그것들을 사용하지 않아도 ref 또는 getter 인수를 받아들일 수 있습니다. 다른 개발자가 사용할 수 있는 composable을 작성하는 경우, 입력 인수가 원시 값 대신 ref나 getter인 경우를 처리하는 것이 좋습니다. 이럴 때 **toValue()** 유틸리티 함수가 유용하게 사용될 수 있습니다:

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

function useFeature(maybeRefOrGetter) {
  // MaybeRefOrGetter가 ref 또는 getter인 경우,
  // 정규화된 값이 반환됩니다.
  // 그렇지 않으면 있는 그대로 반환됩니다.
  const value = toValue(maybeRefOrGetter)
}
```

입력이 ref 또는 getter일 때 composable이 반응형 효과를 생성하면, 반드시 `watch()` 로 ref / getter를 명시적으로 감시하거나, 적절히 추적되도록 `watchEffect()` 내에서 `toValue()` 를 호출해야 합니다.

앞서 논의한 **useFetch()** 구현은 입력 인수로 ref, getter, 그리고 일반 값을 받아들이는 composable의 구체적인 예를 제공합니다.

반환 값

컴포저블에서 `reactive()` 대신 `ref()` 를 독점적으로 사용하고 있다는 것을 눈치챘을 것입니다. 컴포넌트에서 항상 ref 객체를 반환하여 반응성을 유지하면서 컴포넌트에서 구조화할 수 있도록 하는 것이 좋습니다.

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

컴포저블 함수에서 reactive 객체를 반환하는 경우, 위 코드처럼 분해 할당 문법을 사용하면 컴포저블 함수 내부 상태에 대한 반응성 연결이 끊어지지만, 분해 할당 문법을 사용하지 않는 경우에는 ref 연결상태가 유지됩니다.

컴포저블 함수로 반환된 ref 상태를 감싸는 객체를 `.value` 접두사 없이 객체 속성처럼 사용하고 싶다면, 반환된 일반 객체를 `reactive()` 로 래핑하여 ref가 래핑되지 않도록 할 수 있습니다:

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

```template
마우스 위치: {{ mouse.x }}, {{ mouse.y }}
```

사이드 이펙트

컴포저블에서 사이드 이펙트(예: DOM 이벤트 리스너 추가 또는 데이터 가져오기)를 수행하는 것은 괜찮지만 다음 규칙에 주의하십시오:

서버 사이드 렌더링(SSR)을 사용하는 앱에서 작업하는 경우, 마운트 후 생명주기 훅인 `onMounted()` 에서 DOM 관련 사이드 이펙트를 수행해야 합니다. 이 훅은 브라우저에서만 호출되므로 내부 코드가 DOM에 접근할 수 있는지 알 수 있습니다.

`onUnmounted()` 에서 사이드 이펙트를 마무리 지었는지 확인하십시오. 예를 들어, 컴포저블 코드에서 DOM 이벤트 리스너를 사용하는 경우, `onUnmounted()` 에서 해당 리스너를 제거해야 합니다(**`useMouse()`** 예제에서 본 것처럼). **`useEventListener()`** 예제와 같이 자동으로 이를 수행하는 컴포저블 코드를 구성하는 것도 좋은 아이디어일 수 있습니다.

제한사항

컴포저블(Composable)은 `<script setup>` 또는 `setup()` 훅에서만 호출해야 합니다. 이러한 컨텍스트에서는 컴포저블을 **동기적으로** 호출해야 합니다. 일부 경우에는 `onMounted()` 와 같은 라이프사이클 훅에서도 호출할 수 있습니다.

이러한 제한은 중요합니다. 왜냐하면 이러한 컨텍스트에서 Vue가 현재 활성 컴포넌트 인스턴스를 결정할 수 있기 때문입니다. 활성 컴포넌트 인스턴스에 대한 접근은 다음과 같은 이유로 필요합니다:

생명주기 훅을 등록할 수 있다.

계산된 속성과 감시자는 컴포넌트 마운트 해제 시, 관련 작업을 처리하기 위해 연결될 수 있다.

TIP

`<script setup>` 은 `await` 를 사용한 **후** 컴포저블 함수를 호출할 수 있는 유일한 곳입니다. 컴파일러는 비동기 작업 후 활성 인스턴스 컨텍스트를 자동으로 복원합니다.

체계적인 코드를 위해 컴포저블로 추출하기

컴포저블은 재사용뿐만 아니라 코드 체계화를 위해 추출할 수 있습니다. 컴포넌트의 복잡성이 증가함에 따라 탐색 및 추론하기에 너무 큰 컴포넌트가 생길 수 있습니다. 컴포지션 API는 논리적 문제를 기반으로 컴포넌트 코드를 더 작은 기능으로 구성할 수 있는 완전한 유연성을 제공합니다:

vue

```vue
<script setup>
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>
```

이렇게 추출된 컴포저블 코드는 협업을 위해 컴포넌트 범위 내에서 제공할 수 있습니다.

옵션 API에서 컴포저블 적용

옵션 API를 사용하는 경우, 컴포저블 함수는 `setup()` 내에서 호출되어야 하고 반환된 바인딩은 `this` 와 템플릿에 노출되도록 `setup()` 에서 반환되어야 합니다:

js

```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)
  }
  // ...다른 옵션 코드 로직
}
```

다른 기술과의 비교

vs. Mixins

Vue 2를 사용한 개발자는 **mixins** 옵션에 익숙할 것입니다. 이 옵션을 사용해 컴포넌트 로직을 재사용 가능한 단위로 추출할 수도 있습니다. mixins에는 세 가지 주요 단점이 있습니다:

불분명한 출처의 속성: 많은 mixins를 사용할 때, 어떤 인스턴스 속성이 어떤 mixins에 의해 주입되는지 명확하지 않아 구현을 추적하고 컴포넌트의 동작을 이해하기 어렵습니다. 이것이 컴포저블에 refs + destructure 패턴을 사용하는 것을 권장하는 이유이기도 합니다. 컴포넌트가 속성을 사용할 때 그 출처를 명확하게 만듭니다.

네임스페이스 충돌: 다른 작성자의 여러 mixins이 잠재적으로 동일한 속성 키를 등록하여 네임스페이스 충돌을 일으킬 수 있습니다. 컴포저블을 사용하면 다른 컴포저블과 충돌하는 키가 있는 경우 분해 할당 문법으로 변수의 이름을 바꿀 수 있습니다.

암시적 mixins 간 통신: 서로 상호 작용해야 하는 여러 mixins은 공유 속성 키에 의존해야 하므로 암시적으로 결합됩니다. 컴포저블을 사용하면 일반 함수와 마찬가지로 한 컴포저블에서 반환된 값을 다른 컴포저블에 인자로 전달할 수 있습니다.

위의 이유로 Vue 3에서는 더 이상 mixins를 사용하지 않는 것이 좋습니다. 이 기능은 마이그레이션 및 익숙함을 위해서만 유지됩니다.

vs. 렌더리스 컴포넌트

컴포넌트 심화의 슬롯 챕터에서는 범위가 지정된 슬롯을 기반으로 하는 렌더리스 컴포넌트 패턴에 대해 논의했습니다. 렌더리스 컴포넌트를 사용하여 동일한 마우스 추적 데모도 구현했습니다.

렌더리스 컴포넌트에 비해 컴포저블의 주요 이점은 컴포저블이 추가적인 컴포넌트 인스턴스 오버헤드를 발생시키지 않는다는 것입니다. 전체 앱에서 사용될 때 렌더리스 컴포넌트 패턴에 의해 생성된 추가 컴포넌트 인스턴스의 양이 눈에 띄게 성능 오버헤드가 될 수 있습니다.

권장 사항은 순수 로직을 재사용할 때 컴포저블을 사용하고 로직과 시각적 레이아웃을 모두 재사용할 때 컴포넌트를 사용하는 것입니다.

vs. React 훅

React에 대한 경험이 있다면 이것이 커스텀 React 훅과 매우 유사하다는 것을 알 수 있습니다. 컴포지션 API는 부분적으로 React 훅에서 영감을 얻었고 Vue 컴포저블은 실제로 로직 구성 기능 측면에서 React 훅와 유사합니다. 그러나 Vue 컴포저블은 Vue의 세분화된 반응성 시스템을 기반으로 하며, 이는 React 훅의 실행 모델과 근본적으로 다릅니다. 이는 컴포지션 **API FAQ**에서 자세히 설명합니다.

추가적인 읽을거리

반응형 심화: Vue의 반응형 시스템이 어떻게 작동하는지에 대한 심화 수준의 이해를 위해.

상태 관리: 여러 컴포넌트가 공유하는 상태를 관리하는 패턴입니다.

테스팅 컴포저블: 단위 테스트 컴포저블에 대한 팁.

VueUse: 계속 증가하는 Vue 컴포저블 컬렉션입니다. 또한 소스 코드는 훌륭한 학습 자료입니다.