Props

> 이 페이지는 이미 컴포넌트 기본을 읽었다고 가정합니다. 컴포넌트가 처음이라면 먼저 해당 내용을 읽어보세요.

Props 선언

Vue 컴포넌트는 명시적인 props 선언이 필요합니다. 그래야 Vue가 컴포넌트에 전달된 외부 props 중 어떤 것을 통과 속성(fallthrough attributes)으로 처리해야 하는지 알 수 있습니다(이 내용은 별도의 섹션에서 다룹니다).

SFC에서 `<script setup>` 을 사용할 때는 `defineProps()` 매크로를 사용하여 props를 선언할 수 있습니다:

```vue
<script setup>
const props = defineProps(['foo'])

console.log(props.foo)
</script>
```

`<script setup>` 이 아닌 컴포넌트에서는 `props` 옵션을 사용하여 props를 선언합니다:

```js
export default {
  props: ['foo'],
  setup(props) {
    // setup()은 props를 첫 번째 인자로 받습니다.
    console.log(props.foo)
  }
}
```

`defineProps()` 에 전달된 인자는 `props` 옵션에 제공된 값과 동일합니다. 두 선언 방식 모두 동일한 props 옵션 API를 공유합니다.

```js
// <script setup>에서
definePropsㅣ({
  title: String,
  likes: Number
})
```

```js
// <script setup>이 아닌 경우
export default {
  props: {
    title: String,
    likes: Number
  }
}
```

객체 선언 문법에서 각 속성의 키는 prop의 이름이고, 값은 기대하는 타입의 생성자 함수여야 합니다.

이렇게 하면 컴포넌트의 문서화뿐만 아니라, 브라우저 콘솔에서 잘못된 타입이 전달될 경우 다른 개발자에게 경고도 해줍니다. prop 검증에 대한 자세한 내용은 이 페이지 아래에서 다룹니다.

TypeScript와 `<script setup>` 을 함께 사용하는 경우, 순수 타입 주석만으로도 props를 선언할 수 있습니다:

```vue
<script setup lang="ts">
definePropsㅣ<{
  title?: string
  likes?: number
}>()
</script>
```

자세한 내용: 컴포넌트 Props 타입 지정 TS

반응형 Props 구조 분해 3.5+

Vue의 반응성 시스템은 속성 접근을 기반으로 상태 사용을 추적합니다. 예를 들어, 계산된 getter나 watcher에서 `props.foo` 에 접근하면, `foo` prop이 의존성으로 추적됩니다.

따라서 다음과 같은 코드가 있다고 가정해봅시다:

```
watchEffect(() => {
  // 3.5 이전에는 한 번만 실행됨
  // 3.5+에서는 "foo" prop이 변경될 때마다 다시 실행됨
  console.log(foo)
})
```

3.4 버전 이하에서는 `foo` 가 실제 상수이기 때문에 절대 변경되지 않습니다. 3.5 버전 이상에서는, Vue의 컴파일러가 동일한 `<script setup>` 블록 내에서 `defineProps` 에서 구조 분해된 변수를 접근할 때 자동으로 `props.` 를 앞에 붙입니다. 따라서 위 코드는 다음과 동일하게 변환됩니다:

```js
const props = defineProps(['foo'])

watchEffect(() => {
  // 컴파일러가 `foo`를 `props.foo`로 변환
  console.log(props.foo)
})
```

또한, JavaScript의 기본값 문법을 사용하여 props의 기본값을 선언할 수 있습니다. 이는 타입 기반 props 선언을 사용할 때 특히 유용합니다:

```ts
const { foo = 'hello' } = defineProps<{ foo?: string }>()
```

IDE에서 구조 분해된 props와 일반 변수를 시각적으로 구분하고 싶다면, Vue의 VSCode 확장 프로그램에서 구조 분해된 props에 대한 인레이 힌트(inlay-hints)를 활성화하는 설정이 있습니다.

구조 분해된 Props를 함수에 전달하기

구조 분해된 prop을 함수에 전달할 때, 예를 들어:

```js
const { foo } = defineProps(['foo'])

watch(foo, /* ... */)
```

이 코드는 기대한 대로 동작하지 않습니다. 왜냐하면 이는 `watch(props.foo, ...)` 와 동일하기 때문입니다. 즉, 반응형 데이터 소스가 아닌 값을 전달하게 됩니다. 실제로 Vue의 컴파일러는 이러한 경우를 감지하여 경고를 발생시킵니다.

getter로 감싸서 감시할 수 있습니다:

```js
watch(() => foo, /* ... */)
```

또한, 구조 분해된 prop을 외부 함수에 전달하면서 반응성을 유지해야 할 때도 이 방법이 권장됩니다:

```js
useComposable(() => foo)
```

외부 함수는 getter를 호출하거나(toValue로 정규화)하여, 예를 들어 계산된 값이나 watcher getter에서 전달된 prop의 변화를 추적할 수 있습니다.

Prop 전달 세부사항

Prop 이름 표기법

긴 prop 이름은 camelCase로 선언합니다. 이렇게 하면 속성 키로 사용할 때 따옴표를 사용할 필요가 없고, 템플릿 표현식에서 직접 참조할 수 있습니다. camelCase는 유효한 JavaScript 식별자이기 때문입니다:

```js
defineProps({
  greetingMessage: String
})
```

```template
<span>{{ greetingMessage }}</span>
```

기술적으로는 자식 컴포넌트에 props를 전달할 때도 camelCase를 사용할 수 있습니다(in-DOM 템플릿에서는 예외). 하지만 HTML 속성과 일치시키기 위해 모든 경우에 kebab-case를 사용하는 것이 관례입니다:

```template
<MyComponent greeting-message="hello" />
```

컴포넌트 태그에는 PascalCase를 사용하는 것이 가능하다면 권장됩니다. 이렇게 하면 Vue 컴포넌트와 네이티브 요소를 구분하여 템플릿 가독성이 향상됩니다. 하지만 props를 전달할 때 camelCase를 사용하는 실질적인 이점은 크지 않으므로, 각 언어의 관례를 따릅니다.

지금까지는 다음과 같이 props를 정적 값으로 전달하는 예시를 보았습니다:

template
```
<BlogPost title="My journey with Vue" />
```

또한 `v-bind` 또는 `:` 단축키를 사용하여 props를 동적으로 할당하는 예시도 보았습니다:

template
```
<!-- 변수의 값을 동적으로 할당 -->
<BlogPost :title="post.title" />

<!-- 복잡한 표현식의 값을 동적으로 할당 -->
<BlogPost :title="post.title + ' by ' + post.author.name" />
```

다양한 값 타입 전달하기

위 두 예시에서는 문자열 값을 전달했지만, 어떤 타입의 값도 prop으로 전달할 수 있습니다.

숫자

template
```
<!-- `42`가 정적이지만, v-bind를 사용해 -->
<!-- 이것이 문자열이 아닌 JavaScript 표현식임을 Vue에 알려야 합니다. -->
<BlogPost :likes="42" />

<!-- 변수의 값을 동적으로 할당 -->
<BlogPost :likes="post.likes" />
```

불리언

template
```
<!-- 값을 지정하지 않고 prop만 포함하면 `true`로 간주됩니다. -->
<BlogPost is-published />

<!-- `false`가 정적이지만, v-bind를 사용해 -->
<!-- 이것이 문자열이 아닌 JavaScript 표현식임을 Vue에 알려야 합니다. -->
<BlogPost :is-published="false" />

<!-- 변수의 값을 동적으로 할당 -->
<BlogPost :is-published="post.isPublished" />
```

배열

template
```
<!-- 배열이 정적이지만, v-bind를 사용해 -->
<!-- 이것이 문자열이 아닌 JavaScript 표현식임을 Vue에 알려야 합니다. -->
```

```
<!-- 변수의 값을 동적으로 할당 -->
<BlogPost :comment-ids="post.commentIds" />
```

객체

template

```
<!-- 객체가 정적이지만, v-bind를 사용해 -->
<!-- 이것이 문자열이 아닌 JavaScript 표현식임을 Vue에 알려야 합니다. -->
<BlogPost
  :author="{
    name: 'Veronica',
    company: 'Veridian Dynamics'
  }"
 />

<!-- 변수의 값을 동적으로 할당 -->
<BlogPost :author="post.author" />
```

객체를 사용하여 여러 속성 바인딩하기

객체의 모든 속성을 props로 전달하고 싶다면, `v-bind` 에 인자 없이 사용할 수 있습니다 (`:prop-name` 대신 `v-bind`). 예를 들어, `post` 객체가 있다고 가정해봅시다:

js

```
const post = {
  id: 1,
  title: 'My Journey with Vue'
}
```

다음 템플릿은:

template

```
<BlogPost v-bind="post" />
```

다음과 동일하게 동작합니다:

template

```
<BlogPost :id="post.id" :title="post.title" />
```

단방향 데이터 흐름

업데이트되면 자식에게 전달되지만, 반대 방향은 아닙니다. 이렇게 하면 자식 컴포넌트가 실수로 부모의 상태를 변경하는 것을 방지하여 앱의 데이터 흐름을 더 쉽게 이해할 수 있습니다.

또한, 부모 컴포넌트가 업데이트될 때마다 자식 컴포넌트의 모든 prop이 최신 값으로 갱신됩니다. 즉, 자식 컴포넌트 내부에서 prop을 **변경하려고 시도해서는 안 됩니다**. 만약 그렇게 하면, Vue는 콘솔에 경고를 표시합니다:

```js
const props = defineProps(['foo'])

// ❌ 경고, props는 읽기 전용입니다!
props.foo = 'bar'
```

props를 변경하고 싶은 유혹을 느끼는 경우는 보통 두 가지입니다:

1. **prop이 초기값을 전달하는 용도로 사용되고, 자식 컴포넌트가 이후에 이를 로컬 데이터 속성으로 사용하고 싶을 때.** 이 경우, prop을 초기값으로 사용하는 로컬 데이터 속성을 정의하는 것이 가장 좋습니다:

   ```js
   const props = defineProps(['initialCounter'])

   // counter는 props.initialCounter를 초기값으로만 사용;
   // 이후 prop 업데이트와는 연결되지 않습니다.
   const counter = ref(props.initialCounter)
   ```

2. **prop이 변환이 필요한 원시 값으로 전달될 때.** 이 경우, prop 값을 사용하는 계산된 속성을 정의하는 것이 가장 좋습니다:

   ```js
   const props = defineProps(['size'])

   // prop이 변경될 때 자동으로 업데이트되는 계산된 속성
   const normalizedSize = computed(() => props.size.trim().toLowerCase())
   ```

객체/배열 Props 변경하기

객체와 배열이 prop으로 전달될 때, 자식 컴포넌트는 prop 바인딩 자체를 변경할 수는 없지만, 객체나 배열의 중첩 속성은 **변경할 수 있습니다**. 이는 JavaScript에서 객체와 배열이 참조로 전달되기 때문이며, Vue가 이러한 변경을 막는 것은 비효율적이기 때문입니다.

이러한 변경의 주요 단점은 자식 컴포넌트가 부모 상태에 영향을 줄 수 있다는 점입니다. 이는 부모 컴포넌트 입장에서는 명확하지 않아, 향후 데이터 흐름을 이해하기 어렵게 만들 수 있습니다. 모범 사례로, 부모와 자식이 설계상 밀접하게 결합되어 있지 않다면 이러한 변경을 피해야 합니다. 대부분의 경우, 자식이 이벤트를 발생시켜 부모가 변경을 수행하도록 하는 것이 좋습니다.

Prop 검증

컴포넌트는 props에 대한 요구사항(이미 본 타입 등)을 지정할 수 있습니다. 요구사항이 충족되지 않으면, Vue는 브라우저의 JavaScript 콘솔에 경고를 표시합니다. 이는 다른 사람이 사용할 컴포넌트를 개발할 때 특히 유용합니다.

prop 검증을 지정하려면, `defineProps()` 매크로에 문자열 배열 대신 검증 요구사항이 담긴 객체를 제공하면 됩니다. 예를 들어:

```js
defineProps({
  // 기본 타입 체크
  //  (`null` 및 `undefined` 값은 모든 타입 허용)
  propA: Number,
  // 여러 타입 허용
  propB: [String, Number],
  // 필수 문자열
  propC: {
    type: String,
    required: true
  },
  // 필수이지만 null 허용 문자열
  propD: {
    type: [String, null],
    required: true
  },
  // 기본값이 있는 숫자
  propE: {
    type: Number,
    default: 100
  },
  // 기본값이 있는 객체
  propF: {
    type: Object,
    // 객체나 배열의 기본값은 반드시
    // 팩토리 함수에서 반환해야 합니다. 이 함수는
    // 컴포넌트가 받은 원시 props를 인자로 받습니다.
    default(rawProps) {
      return { message: 'hello' }
    }
  },
  // 커스텀 검증 함수
  // 3.4+에서는 전체 props가 두 번째 인자로 전달됨
  propG: {
    validator(value, props) {
      // 값이 아래 문자열 중 하나와 일치해야 함
      return ['success', 'warning', 'danger'].includes(value)
    }
  },
  // 기본값이 있는 함수
  propH: {
    type: Function,
```

```
    // 함수가 아니라 기본값으로 사용할 함수입니다
    default() {
      return 'Default function'
    }
  }
})
```

> ⓘ TIP
>
> `defineProps()` 인자 내부의 코드는 **`<script setup>` 에서 선언된 다른 변수에 접근할 수 없습니다**. 전체 표현식이 컴파일 시 외부 함수 스코프로 이동되기 때문입니다.

추가 세부사항:

- 모든 prop은 기본적으로 선택 사항이며, `required: true` 가 지정된 경우에만 필수입니다.
- `Boolean` 이 아닌 선택적 prop이 없으면 값은 `undefined` 가 됩니다.
- `Boolean` prop이 없으면 `false` 로 변환됩니다. 이를 변경하려면 `default` 를 설정할 수 있습니다. 예: `default: undefined` 로 설정하면 Boolean이 아닌 prop처럼 동작합니다.
- `default` 값이 지정된 경우, prop 값이 `undefined` 로 해석되면(즉, prop이 없거나 명시적으로 `undefined` 가 전달된 경우) 해당 값이 사용됩니다.

prop 검증에 실패하면, Vue는 콘솔에 경고를 출력합니다(개발 빌드 사용 시).

타입 기반 props 선언 TS 을 사용하는 경우, Vue는 타입 주석을 동등한 런타임 prop 선언으로 컴파일하려고 시도합니다. 예를 들어, `defineProps<{ msg: string }>` 는 `{ msg: { type: String, required: true }}` 로 컴파일됩니다.

런타임 타입 체크

`type` 은 다음과 같은 네이티브 생성자 중 하나일 수 있습니다:

- `String`
- `Number`
- `Boolean`
- `Array`
- `Object`
- `Date`
- `Function`
- `Symbol`
- `Error`

합니다. 예를 들어, 다음과 같은 클래스가 있다고 가정해봅시다:

```js
class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }
}
```

이를 prop의 타입으로 사용할 수 있습니다:

```js
defineProps({
  author: Person
})
```

Vue는 `author` prop의 값이 실제로 `Person` 클래스의 인스턴스인지 확인하기 위해 `instanceof Person` 을 사용합니다.

Nullable 타입

타입이 필수이지만 null을 허용해야 한다면, `null` 을 포함한 배열 문법을 사용할 수 있습니다:

```js
defineProps({
  id: {
    type: [String, null],
    required: true
  }
})
```

`type` 이 배열 문법 없이 단순히 `null` 인 경우, 모든 타입을 허용합니다.

Boolean 변환

`Boolean` 타입의 prop은 네이티브 불리언 속성의 동작을 모방하기 위해 특별한 변환 규칙을 가집니다. 다음과 같이 선언된 `<MyComponent>` 가 있다고 가정해봅시다:

```js
defineProps({
  disabled: Boolean
```

컴포넌트는 다음과 같이 사용할 수 있습니다:

```template
<!-- :disabled="true"를 전달한 것과 동일 -->
<MyComponent disabled />

<!-- :disabled="false"를 전달한 것과 동일 -->
<MyComponent />
```

prop이 여러 타입을 허용하도록 선언된 경우에도, `Boolean` 에 대한 변환 규칙이 적용됩니다. 하지만 `String` 과 `Boolean` 이 모두 허용되는 경우에는 예외가 있습니다. 이때 Boolean 변환 규칙은 Boolean이 String보다 먼저 나올 때만 적용됩니다:

```js
// disabled는 true로 변환됨
defineProps({
  disabled: [Boolean, Number]
})

// disabled는 true로 변환됨
defineProps({
  disabled: [Boolean, String]
})

// disabled는 true로 변환됨
defineProps({
  disabled: [Number, Boolean]
})

// disabled는 빈 문자열로 파싱됨 (disabled="")
defineProps({
  disabled: [String, Boolean]
})
```

GitHub에서 이 페이지 편집

Previous
등록

Next
이벤트