Vue와 웹 컴포넌트

웹 컴포넌트는 개발자가 재사용 가능한 커스텀 엘리먼트를 만들 수 있도록 해주는 웹 네이티브 API 집합을 일컫는 용어입니다.

Vue와 웹 컴포넌트는 주로 상호 보완적인 기술로 간주됩니다. Vue는 커스텀 엘리먼트의 소비와 생성 모두에 대해 뛰어난 지원을 제공합니다. 기존 Vue 애플리케이션에 커스텀 엘리먼트를 통합하든, Vue를 사용해 커스텀 엘리먼트를 빌드하고 배포하든, 좋은 선택입니다.

Vue에서 커스텀 엘리먼트 사용하기

Vue는 Custom Elements Everywhere 테스트에서 100% 만점을 기록합니다. Vue 애플리케이션 내에서 커스텀 엘리먼트를 사용하는 것은 기본 HTML 엘리먼트를 사용하는 것과 거의 동일하게 동작하지만, 몇 가지 유의할 점이 있습니다:

컴포넌트 해석 건너뛰기

기본적으로 Vue는 네이티브 HTML 태그가 아닌 태그를 등록된 Vue 컴포넌트로 해석하려 시도한 후, 실패하면 커스텀 엘리먼트로 렌더링합니다. 이로 인해 개발 중에 "컴포넌트 해석 실패" 경고가 발생할 수 있습니다. 특정 엘리먼트를 커스텀 엘리먼트로 처리하고 컴포넌트 해석을 건너뛰도록 Vue에 알리려면 `compilerOptions.isCustomElement` 옵션을 지정할 수 있습니다.

Vue를 빌드 환경에서 사용할 경우, 이 옵션은 컴파일 타임 옵션이므로 빌드 설정을 통해 전달해야 합니다.

브라우저 내 설정 예시

```js
// 브라우저 내 컴파일을 사용할 때만 동작합니다.
// 빌드 도구를 사용하는 경우 아래의 설정 예시를 참고하세요.
app.config.compilerOptions.isCustomElement = (tag) => tag.includes('-')
```

vite.config.js

```js
import vue from '@vitejs/plugin-vue'

export default {
  plugins: [
    vue({
      template: {
        compilerOptions: {
          // 대시(-)가 포함된 모든 태그를 커스텀 엘리먼트로 처리
          isCustomElement: (tag) => tag.includes('-')
        }
      }
    })
  ]
}
```

Vue CLI 설정 예시

vue.config.js

```js
module.exports = {
  chainWebpack: (config) => {
    config.module
      .rule('vue')
      .use('vue-loader')
      .tap((options) => ({
        ...options,
        compilerOptions: {
          // ion-으로 시작하는 모든 태그를 커스텀 엘리먼트로 처리
          isCustomElement: (tag) => tag.startsWith('ion-')
        }
      }))
  }
}
```

DOM 속성 전달하기

DOM 속성(attribute)은 문자열만 허용하므로, 복잡한 데이터를 커스텀 엘리먼트에 전달하려면 DOM 속성(property)으로 전달해야 합니다. 커스텀 엘리먼트에 props를 설정할 때, Vue 3는 `in` 연산자를 사용해 DOM 속성 존재 여부를 자동으로 확인하고, 키가 존재하면 DOM 속성으로 값을 설정하는 것을 우선시합니다. 즉, 커스텀 엘리먼트가 권장 모범 사례를 따르면 대부분의 경우 별도로 신경 쓸 필요가 없습니다.

하지만 드물게 데이터를 DOM 속성으로 전달해야 하지만, 커스텀 엘리먼트가 해당 속성을 제대로 정의/반영하지 않아(`in` 체크가 실패) 문제가 발생할 수 있습니다. 이 경우 `.prop` 수식어를

```template
<my-element :user.prop="{ name: 'jack' }"></my-element>

<!-- 축약형 -->
<my-element .user="{ name: 'jack' }"></my-element>
```

Vue로 커스텀 엘리먼트 빌드하기

커스텀 엘리먼트의 주요 장점은 어떤 프레임워크와도, 심지어 프레임워크 없이도 사용할 수 있다는 점입니다. 이는 최종 사용자가 동일한 프론트엔드 스택을 사용하지 않을 수 있는 컴포넌트 배포나, 최종 애플리케이션을 컴포넌트의 구현 세부사항으로부터 격리하고 싶을 때 이상적입니다.

defineCustomElement

Vue는 `defineCustomElement` 메서드를 통해 기존 Vue 컴포넌트 API와 동일하게 커스텀 엘리먼트를 생성할 수 있습니다. 이 메서드는 `defineComponent` 와 동일한 인자를 받지만, 대신 `HTMLElement` 를 확장하는 커스텀 엘리먼트 생성자를 반환합니다:

```template
<my-vue-element></my-vue-element>
```

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

const MyVueElement = defineCustomElement({
  // 일반적인 Vue 컴포넌트 옵션
  props: {},
  emits: {},
  template: `...`,

  // defineCustomElement 전용: shadow root에 주입될 CSS
  styles: [`/* 인라인된 css */`]
})

// 커스텀 엘리먼트 등록
// 등록 후, 페이지의 모든 `<my-vue-element>` 태그가
// 업그레이드됩니다.
customElements.define('my-vue-element', MyVueElement)

// 프로그래밍적으로 엘리먼트를 인스턴스화할 수도 있습니다:
// (등록 후에만 가능)
document.body.appendChild(
  new MyVueElement({
    // 초기 props (선택)
```

```
)
```

라이프사이클

Vue 커스텀 엘리먼트는 엘리먼트의 `connectedCallback` 이 처음 호출될 때, 내부적으로 Vue 컴포넌트 인스턴스를 shadow root에 마운트합니다.

엘리먼트의 `disconnectedCallback` 이 호출되면, Vue는 마이크로태스크 틱 이후 엘리먼트가 문서에서 분리되었는지 확인합니다.

엘리먼트가 여전히 문서에 있으면 이동(move)으로 간주되어 컴포넌트 인스턴스가 유지됩니다.

엘리먼트가 문서에서 분리되면 제거(removal)로 간주되어 컴포넌트 인스턴스가 언마운트됩니다.

Props

`props` 옵션으로 선언된 모든 prop은 커스텀 엘리먼트의 속성(property)으로 정의됩니다. Vue는 적절한 경우 속성(attribute)과 속성(property) 간의 반영을 자동으로 처리합니다.

속성(attribute)은 항상 해당 속성(property)으로 반영됩니다.

원시값(`string` , `boolean` , `number`)을 가진 속성(property)은 속성(attribute)으로 반영됩니다.

Vue는 또한 `Boolean` 또는 `Number` 타입으로 선언된 prop이 속성(attribute)으로 설정될 때 (항상 문자열임) 원하는 타입으로 자동 변환합니다. 예를 들어, 다음과 같이 props를 선언하면:

```js
props: {
  selected: Boolean,
  index: Number
}
```

그리고 커스텀 엘리먼트를 다음과 같이 사용하면:

```template
<my-element selected index="1"></my-element>
```

컴포넌트 내에서 `selected` 는 `true` (boolean)로, `index` 는 `1` (number)로 변환됩니다.

이벤트

CustomEvent로 디스패치됩니다. 추가 이벤트 인자(페이로드)는 CustomEvent 객체의 `detail` 속성에 배열로 노출됩니다.

슬롯

컴포넌트 내부에서는 평소처럼 `<slot/>` 엘리먼트를 사용해 슬롯을 렌더링할 수 있습니다. 하지만, 결과 엘리먼트를 사용할 때는 네이티브 슬롯 문법만 허용됩니다:

스코프 슬롯은 지원되지 않습니다.

네임드 슬롯을 전달할 때는 `v-slot` 디렉티브 대신 `slot` 속성을 사용하세요:

```template
<my-element>
  <div slot="named">hello</div>
</my-element>
```

Provide / Inject

Provide / Inject API와 Composition API 버전도 Vue로 정의된 커스텀 엘리먼트 간에 동작합니다. 단, **커스텀 엘리먼트 간에만** 동작한다는 점에 유의하세요. 즉, Vue로 정의된 커스텀 엘리먼트는 커스텀 엘리먼트가 아닌 Vue 컴포넌트가 제공한 속성을 주입받을 수 없습니다.

앱 레벨 설정 3.5+

`configureApp` 옵션을 사용해 Vue 커스텀 엘리먼트의 앱 인스턴스를 설정할 수 있습니다:

```js
defineCustomElement(MyComponent, {
  configureApp(app) {
    app.config.errorHandler = (err) => {
      /* ... */
    }
  }
})
```

SFC를 커스텀 엘리먼트로 사용하기

`defineCustomElement` 는 Vue 싱글 파일 컴포넌트(SFC)와도 함께 동작합니다. 하지만 기본 툴링 설정에서는 SFC 내부의 `<style>` 이 프로덕션 빌드 시 여전히 추출되어 하나의 CSS 파일로 병합됩니다. SFC를 커스텀 엘리먼트로 사용할 때는 `<style>` 태그를 커스텀 엘리먼트의 shadow root에 주입하는 것이 바람직할 수 있습니다.

공식 SFC 툴링은 "커스텀 엘리먼트 모드"로 SFC를 임포트하는 것을 지원합니다 (`@vitejs/plugin-vue@^1.4.0` 또는 `vue-loader@^16.5.0` 필요). 커스텀 엘리먼트 모드로 로드된 SFC는 `<style>` 태그를 CSS 문자열로 인라인 처리하고, 이를 컴포넌트의 `styles` 옵션에

에 주입됩니다.

이 모드를 사용하려면 컴포넌트 파일명을 `.ce.vue` 로 끝나게 하면 됩니다:

```js
import { defineCustomElement } from 'vue'
import Example from './Example.ce.vue'

console.log(Example.styles) // ["/* 인라인된 css */"]

// 커스텀 엘리먼트 생성자로 변환
const ExampleElement = defineCustomElement(Example)

// 등록
customElements.define('my-example', ExampleElement)
```

커스텀 엘리먼트 모드로 임포트할 파일을 커스터마이즈하고 싶다면(예: 모든 SFC를 커스텀 엘리먼트로 처리), 해당 빌드 플러그인에 `customElement` 옵션을 전달할 수 있습니다:

@vitejs/plugin-vue
vue-loader

Vue 커스텀 엘리먼트 라이브러리 제작 팁

Vue로 커스텀 엘리먼트를 빌드할 때, 엘리먼트는 Vue 런타임에 의존하게 됩니다. 사용되는 기능에 따라 약 16kb의 기본 크기 비용이 발생합니다. 즉, 단일 커스텀 엘리먼트만 배포한다면 Vue를 사용하는 것이 이상적이지 않을 수 있습니다. 이 경우 바닐라 JavaScript, petite-vue, 또는 작은 런타임 크기에 특화된 프레임워크를 고려할 수 있습니다. 하지만 복잡한 로직을 가진 여러 커스텀 엘리먼트를 함께 배포한다면, Vue의 기본 크기 비용은 충분히 정당화됩니다. 더 많은 엘리먼트를 함께 배포할수록 이점이 커집니다.

커스텀 엘리먼트가 Vue를 사용하는 애플리케이션에서 사용된다면, 빌드된 번들에서 Vue를 외부화하여 엘리먼트가 호스트 애플리케이션의 Vue 복사본을 사용하도록 할 수 있습니다.

사용자가 원하는 태그 이름으로 필요할 때마다 임포트하고 등록할 수 있도록, 개별 엘리먼트 생성자를 내보내는 것이 좋습니다. 모든 엘리먼트를 자동으로 등록하는 편의 함수도 내보낼 수 있습니다. 다음은 Vue 커스텀 엘리먼트 라이브러리의 예시 진입점입니다:

elements.js

```js
import { defineCustomElement } from 'vue'
import Foo from './MyFoo.ce.vue'
import Bar from './MyBar.ce.vue'

const MyFoo = defineCustomElement(Foo)
```

```
// 개별 엘리먼트 내보내기
export { MyFoo, MyBar }

export function register() {
  customElements.define('my-foo', MyFoo)
  customElements.define('my-bar', MyBar)
}
```

사용자는 Vue 파일에서 엘리먼트를 사용할 수 있습니다:

vue

```
<script setup>
import { register } from 'path/to/elements.js'
register()
</script>

<template>
  <my-foo ...>
    <my-bar ...></my-bar>
  </my-foo>
</template>
```

또는 JSX를 사용하는 다른 프레임워크에서, 커스텀 이름으로 사용할 수도 있습니다:

jsx

```
import { MyFoo, MyBar } from 'path/to/elements.js'

customElements.define('some-foo', MyFoo)
customElements.define('some-bar', MyBar)

export function MyComponent() {
  return <>
    <some-foo ... >
      <some-bar ... ></some-bar>
    </some-foo>
  </>
}
```

Vue 기반 웹 컴포넌트와 TypeScript

Vue SFC 템플릿을 작성할 때, 커스텀 엘리먼트로 정의된 컴포넌트를 포함해 타입 체크를 하고 싶을 수 있습니다.

커스텀 엘리먼트는 브라우저의 내장 API를 통해 전역적으로 등록되며, 기본적으로 Vue 템플릿에서 사용할 때 타입 추론이 되지 않습니다. Vue에서 커스텀 엘리먼트로 등록된 컴포넌트에 타입 지원을 제공하려면, Vue 템플릿에서 타입 체크를 위해 `GlobalComponents` 인터페이스를 보강(augment)하여 전역 컴포넌트 타입을 등록할 수 있습니다(JSX 사용자는 JSX.IntrinsicElements 타입을 보강할 수 있습니다. 여기서는 다루지 않습니다).

typescript

```ts
import { defineCustomElement } from 'vue'

// Vue 컴포넌트 임포트
import SomeComponent from './src/components/SomeComponent.ce.vue'

// Vue 컴포넌트를 커스텀 엘리먼트 클래스로 변환
export const SomeElement = defineCustomElement(SomeComponent)

// 브라우저에 엘리먼트 클래스를 등록
customElements.define('some-element', SomeElement)

// Vue의 GlobalComponents 타입에 새 엘리먼트 타입 추가
declare module 'vue' {
  interface GlobalComponents {
    // 여기에는 Vue 컴포넌트 타입을 전달해야 합니다
    // (SomeComponent, *SomeElement가 아님*).
    // 커스텀 엘리먼트는 이름에 하이픈이 필요하므로,
    // 하이픈이 포함된 엘리먼트 이름을 사용하세요.
    'some-element': typeof SomeComponent
  }
}
```

비-Vue 웹 컴포넌트와 TypeScript

Vue로 빌드되지 않은 커스텀 엘리먼트의 SFC 템플릿에서 타입 체크를 활성화하는 권장 방법은 다음과 같습니다.

> ⓘ 참고
>
> 이 방법은 가능한 한 가지 방법일 뿐이며, 커스텀 엘리먼트를 생성하는 프레임워크에 따라 다를 수 있습니다.

JS 속성과 이벤트가 정의된 커스텀 엘리먼트가 있고, `some-lib` 라는 라이브러리로 배포된다고 가정해봅시다:

some-lib/src/SomeElement.ts

ts

```ts
// 타입이 지정된 JS 속성을 가진 클래스를 정의
export class SomeElement extends HTMLElement {
  foo: number = 123
  bar: string = 'blah'

  lorem: boolean = false
```

```
  // 이 메서드는 템플릿 타입에 노출되지 않아야 합니다.
  someMethod() {
    /* ... */
  }

  // ... 구현 세부사항 생략 ...
  // ... "apple-fell"이라는 이벤트를 디스패치한다고 가정 ...
}

customElements.define('some-element', SomeElement)

// SomeElement의 속성 중 프레임워크 템플릿(예: Vue SFC 템플릿)에서
// 타입 체크에 사용할 속성 목록입니다.
// 다른 속성은 노출되지 않습니다.
export type SomeElementAttributes = 'foo' | 'bar'

// SomeElement가 디스패치하는 이벤트 타입 정의
export type SomeElementEvents = {
  'apple-fell': AppleFellEvent
}

export class AppleFellEvent extends Event {
  /* ... 세부사항 생략 ... */
}
```

구현 세부사항은 생략했지만, 중요한 점은 prop 타입과 이벤트 타입에 대한 타입 정의가 있다는 것입니다.

Vue에서 커스텀 엘리먼트 타입 정의를 쉽게 등록할 수 있는 타입 헬퍼를 만들어봅시다:

some-lib/src/DefineCustomElement.ts

ts

```
// 각 엘리먼트마다 재사용할 수 있는 타입 헬퍼입니다.
type DefineCustomElement<
  ElementType extends HTMLElement,
  Events extends EventMap = {},
  SelectedAttributes extends keyof ElementType = keyof ElementType
> = new () => ElementType & {
  // $props를 사용해 템플릿 타입 체크에 노출할 속성을 정의합니다.
  // Vue는 $props 타입에서 prop 정의를 읽습니다.
  // 엘리먼트의 prop과 전역 HTML prop,
  // Vue의 특수 prop을 조합합니다.
  /** @deprecated 커스텀 엘리먼트 ref에서 $props 속성을 사용하지 마세요.
    이 속성은 템플릿 prop 타입용입니다. */
  $props: HTMLAttributes &
    Partial<Pick<ElementType, SelectedAttributes>> &
    PublicProps

  // $emit을 사용해 이벤트 타입을 명시적으로 정의합니다.
  // Vue는 $emit 타입에서 이벤트 타입을 읽습니다.
  // $emit은 특정 포맷을 기대하므로, Events를 매핑합니다.
  /** @deprecated 커스텀 엘리먼트 ref에서 $emit 속성을 사용하지 마세요.
    이 속성은 템플릿 prop 타입용입니다. */
```

```
}

type EventMap = {
  [event: string]: Event
}

// EventMap을 Vue의 $emit 타입이 기대하는 포맷으로 매핑합니다.
type VueEmit<T extends EventMap> = EmitFn<{
  [K in keyof T]: (event: T[K]) => void
}>
```

> ⓘ 참고
>
> `$props` 와 `$emit` 에 deprecated를 표시한 이유는, 커스텀 엘리먼트의 ref를 사용할 때 이 속성을 실제로 사용하지 않도록 하기 위함입니다. 이 속성들은 커스텀 엘리먼트의 타입 체크 용도로만 존재하며, 실제 인스턴스에는 존재하지 않습니다.

이 타입 헬퍼를 사용해 Vue 템플릿에서 타입 체크에 노출할 JS 속성을 선택할 수 있습니다:

some-lib/src/SomeElement.vue.ts

```ts
import {
  SomeElement,
  SomeElementAttributes,
  SomeElementEvents
} from './SomeElement.js'
import type { Component } from 'vue'
import type { DefineCustomElement } from './DefineCustomElement'

// Vue의 GlobalComponents 타입에 새 엘리먼트 타입 추가
declare module 'vue' {
  interface GlobalComponents {
    'some-element': DefineCustomElement<
      SomeElement,
      SomeElementAttributes,
      SomeElementEvents
    >
  }
}
```

`some-lib` 가 소스 TypeScript 파일을 `dist/` 폴더로 빌드한다고 가정합시다. `some-lib` 의 사용자는 다음과 같이 `SomeElement` 를 임포트해 Vue SFC에서 사용할 수 있습니다:

SomeElementImpl.vue

```vue
<script setup lang="ts">
// 이 코드는 엘리먼트를 생성하고 브라우저에 등록합니다.
import 'some-lib/dist/SomeElement.js'
```

```
// TypeScript와 Vue를 사용하는 사용자는 추가로
// Vue 전용 타입 정의를 임포트해야 합니다(다른 프레임워크 사용자는
// 해당 프레임워크 전용 타입 정의를 임포트할 수 있습니다).
import type {} from 'some-lib/dist/SomeElement.vue.js'

import { useTemplateRef, onMounted } from 'vue'

const el = useTemplateRef('el')

onMounted(() => {
  console.log(
    el.value!.foo,
    el.value!.bar,
    el.value!.lorem,
    el.value!.someMethod()
  )

  // 이 prop들은 사용하지 마세요. undefined입니다.
  // IDE에서 취소선으로 표시됩니다.
  el.$props
  el.$emit
})
</script>

<template>
  <!-- 이제 타입 체크와 함께 엘리먼트를 사용할 수 있습니다: -->
  <some-element
    ref="el"
    :foo="456"
    :blah="'hello'"
    @apple-fell="
      (event) => {
        // `event`의 타입이 여기서 AppleFellEvent로 추론됩니다.
      }
    "
  ></some-element>
</template>
```

엘리먼트에 타입 정의가 없는 경우, 속성과 이벤트의 타입을 더 수동적으로 정의할 수 있습니다:

SomeElementImpl.vue

vue

```
<script setup lang="ts">
// `some-lib`가 타입 정의가 없는 순수 JS라서
// TypeScript가 타입을 추론할 수 없는 경우:
import { SomeElement } from 'some-lib'

// 앞서 사용한 타입 헬퍼를 그대로 사용합니다.
import { DefineCustomElement } from './DefineCustomElement'

type SomeElementProps = { foo?: number; bar?: string }
type SomeElementEvents = { 'apple-fell': AppleFellEvent }
interface AppleFellEvent extends Event {
  /* ... */
```

```
// Vue의 GlobalComponents 타입에 새 엘리먼트 타입 추가
declare module 'vue' {
  interface GlobalComponents {
    'some-element': DefineCustomElement<
      SomeElementProps,
      SomeElementEvents
    >
  }
}

// ... 이전과 동일하게 엘리먼트에 대한 참조를 사용 ...
</script>

<template>
  <!-- ... 이전과 동일하게 템플릿에서 엘리먼트를 사용 ... -->
</template>
```

커스텀 엘리먼트 작성자는 프레임워크 전용 커스텀 엘리먼트 타입 정의를 라이브러리에서 자동으로 내보내지 않아야 합니다. 예를 들어, 라이브러리의 나머지 부분과 함께 내보내는 `index.ts` 파일에서 내보내지 않아야 하며, 그렇지 않으면 사용자가 예기치 않은 모듈 보강 오류를 겪을 수 있습니다. 사용자는 필요한 프레임워크 전용 타입 정의 파일을 직접 임포트해야 합니다.

웹 컴포넌트 vs. Vue 컴포넌트

일부 개발자는 프레임워크 고유의 컴포넌트 모델을 피하고, 오직 커스텀 엘리먼트만 사용하면 애플리케이션이 "미래 지향적"이 된다고 믿습니다. 여기서는 왜 이것이 지나치게 단순화된 시각인지 설명하고자 합니다.

실제로 커스텀 엘리먼트와 Vue 컴포넌트 사이에는 일정 수준의 기능 중복이 있습니다. 둘 다 데이터 전달, 이벤트 발생, 라이프사이클 관리가 가능한 재사용 가능한 컴포넌트를 정의할 수 있습니다. 하지만 웹 컴포넌트 API는 상대적으로 저수준이고 기본적인 기능만 제공합니다. 실제 애플리케이션을 만들려면 플랫폼에서 제공하지 않는 추가 기능이 많이 필요합니다:

선언적이고 효율적인 템플릿 시스템

컴포넌트 간 로직 추출 및 재사용을 용이하게 하는 반응형 상태 관리 시스템

서버에서 컴포넌트를 효율적으로 렌더링하고 클라이언트에서 하이드레이션(SSR)하는 방법 (SEO 및 LCP와 같은 Web Vitals 지표에 중요). 네이티브 커스텀 엘리먼트 SSR은 일반적으로 Node.js에서 DOM을 시뮬레이션한 후 변경된 DOM을 직렬화하는 방식이지만, Vue SSR은 가능한 한 문자열 연결로 컴파일되어 훨씬 효율적입니다.

Vue의 컴포넌트 모델은 이러한 요구를 염두에 두고 일관된 시스템으로 설계되었습니다.

있겠지만, 이는 곧 사내 프레임워크의 장기 유지보수 부담을 떠안는 것이며, Vue와 같은 성숙한 프레임워크의 생태계 및 커뮤니티 혜택을 잃게 됩니다.

커스텀 엘리먼트를 컴포넌트 모델의 기반으로 삼는 프레임워크도 있지만, 이들 역시 위에서 언급한 문제를 해결하기 위해 고유의 솔루션을 도입할 수밖에 없습니다. 이러한 프레임워크를 사용한다는 것은, 이 문제를 어떻게 해결할지에 대한 그들의 기술적 결정을 받아들이는 것이며, 이는 광고와 달리 미래의 변화로부터 자동으로 보호받는다는 의미는 아닙니다.

또한 커스텀 엘리먼트가 한계가 있는 영역도 있습니다:

> 즉시 평가되는 슬롯은 컴포넌트 조합을 방해합니다. Vue의 스코프 슬롯은 강력한 컴포넌트 조합 메커니즘이지만, 네이티브 슬롯의 즉시 평가 특성 때문에 커스텀 엘리먼트에서는 지원할 수 없습니다. 즉시 평가 슬롯은 또한 수신 컴포넌트가 슬롯 콘텐츠를 언제, 또는 렌더링할지 제어할 수 없음을 의미합니다.
>
> 오늘날 shadow DOM 범위 CSS와 함께 커스텀 엘리먼트를 배포하려면 CSS를 JavaScript에 임베드해 런타임에 shadow root에 주입해야 합니다. SSR 시에는 마크업에 중복된 스타일이 발생합니다. 이 영역에서 플랫폼 기능이 개발 중이지만, 아직 보편적으로 지원되지 않고, 프로덕션 성능/SSR 문제도 남아 있습니다. 그동안 Vue SFC는 스타일을 일반 CSS 파일로 추출할 수 있는 CSS 범위 메커니즘을 제공합니다.

Vue는 항상 웹 플랫폼의 최신 표준을 따라가며, 플랫폼이 우리의 작업을 더 쉽게 해준다면 기꺼이 이를 활용할 것입니다. 하지만 우리의 목표는 오늘 당장 잘 동작하는 솔루션을 제공하는 것입니다. 즉, 새로운 플랫폼 기능을 비판적으로 받아들이고, 표준이 부족한 부분은 그때까지 직접 보완해야 함을 의미합니다.