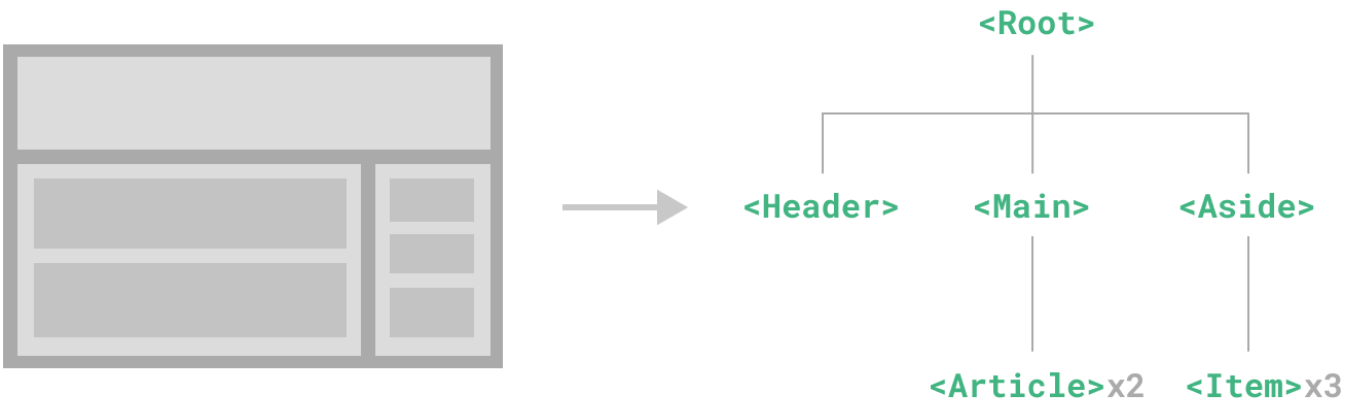
컴포넌트 기초

컴포넌트를 사용하면 UI를 독립적이고 재사용 가능한 일부분으로 분할하고 각 부분을 개별적으로 다룰 수 있습니다. 따라서 앱이 중첩된 컴포넌트의 트리로 구성되는 것은 일반적입니다:



이것은 기본 HTML 엘리먼트를 중첩하는 방법과 매우 유사하지만, Vue는 각 컴포넌트에 사용자 정의 컨텐츠와 논리를 캡슐화할 수 있는 자체 컴포넌트 모델을 구현합니다. Vue는 기본 웹 컴포넌트와도 잘 작동합니다. Vue 컴포넌트와 기본 웹 컴포넌트 간의 관계가 궁금하시다면 여기에서 자세히 읽어보세요.

컴포넌트 정의하기

빌드 방식을 사용할 때 일반적으로 싱글 파일 컴포넌트(줄여서 SFC)라고 하는 `.vue` 확장자를 사용하는 전용 파일에 각 Vue 컴포넌트를 정의합니다:

vue

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

const count = ref(0)
</script>

<template>
  <button @click="count++">당신은 {{ count }} 번 클릭했습니다.</button>
</template>
```

빌드 방식을 사용하지 않을 때, Vue 컴포넌트는 Vue 관련 옵션을 포함하는 일반 JavaScript 객체로 정의할 수 있습니다:

js

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

export default {
  setup() {
    const count = ref(0)
    return { count }
  },
  template: `
    <button @click="count++">
      당신은 {{ count }} 번 클릭했습니다.
    </button>`
  // DOM 내의 템플릿을 대상으로 할 수도 있습니다:
```

```
  // template: '#my-template-element'
}
```

JavaScript 문자열로 정의한 템플릿은 Vue가 즉석에서 컴파일합니다. 엘리먼트(보통 기본 `<template>` 엘리먼트)를 가리키는 ID 셀렉터를 사용할 수도 있습니다. Vue는 해당 컨텐츠를 템플릿 소스로 사용합니다.

위의 예는 단일 컴포넌트를 정의하고 이를 `.js` 파일의 내보내기 기본 값으로 내보냅니다. 그러나 명명된 내보내기를 사용하여 한 파일에서 여러 개의 컴포넌트로 내보낼 수 있습니다.

컴포넌트 사용하기

> **TIP**
>
> 이 가이드의 나머지 부분에서는 SFC 문법을 사용할 것입니다. 컴포넌트에 대한 개념은 빌드 방식를 사용하는지 여부에 관계없이 동일합니다. 예제 섹션은 두 시나리오 모두에서 컴포넌트 사용을 보여줍니다.

자식 컴포넌트를 사용하려면 부모 컴포넌트에서 가져와야(import) 합니다. 파일 안에 `ButtonCounter.vue` 라는 카운터 컴포넌트를 배치했다고 가정하면, 해당 컴포넌트 파일의 기본 내보내기가 노출됩니다:

vue

```vue
<script setup>
import ButtonCounter from './ButtonCounter.vue'
</script>

<template>
  <h1>아래에 자식 컴포넌트가 있습니다.</h1>
  <ButtonCounter />
</template>
```

`<script setup>` 을 사용하면 가져온 컴포넌트를 템플릿에서 자동으로 사용할 수 있습니다.

컴포넌트를 전역으로 등록하면, 가져오기(import) 없이 지정된 앱의 모든 곳에서 컴포넌트를 사용할 수 있습니다. 전역 및 로컬 등록의 장단점은 컴포넌트 등록 섹션에서 설명합니다.

컴포넌트는 원하는 만큼 재사용할 수 있습니다:

template

```vue
<h1>여기에 많은 자식 컴포넌트가 있습니다!</h1>
<ButtonCounter />
<ButtonCounter />
<ButtonCounter />
```

온라인 연습장으로 실행하기

버튼을 클릭할 때 각 버튼은 독립적인 `count` 를 유지합니다. 컴포넌트를 사용할 때마다 해당 컴포넌트의 새 **인스턴스**가 생성되기 때문입니다.

SFC에서는 네이티브 HTML 엘리먼트와 구별하기 위해 자식 컴포넌트에 `PascalCase` 태그 이름을 사용하는 것이 좋습니다. 기본 HTML 태그 이름은 대소문자를 구분하지 않지만, Vue의 SFC는 컴파일된 포맷으로 대소문자를 구분하여 태그 이름을 사용할 수 있습니다. 또한 `/>` 를 사용하여 태그를 닫을 수 있습니다.

템플릿을 DOM에서 직접 작성하는 경우(예: 기본 `<template>` 엘리먼트의 컨텐츠로), 템플릿은 브라우저의 기본 HTML 구문 분석 동작을 따릅니다. 이러한 경우 컴포넌트는 `kebab-case` 및 명시적 닫는 태그를 사용해야 합니다:

template

```vue
<!-- 이 템플릿이 DOM에 작성된 경우 -->
<button-counter></button-counter>
<button-counter></button-counter>
<button-counter></button-counter>
```

자세한 내용은 **in-DOM 템플릿 파싱 주의 사항**을 참조하세요.

Props 전달하기

블로그를 구축하는 경우 블로그 게시물을 나타내는 컴포넌트가 필요할 수 있습니다. 우리는 모든 블로그 게시물이 동일한 시각적 레이아웃을 공유하기를 원하지만 컨텐츠는 다릅니다. 이러한 곳에 사용할 컴포넌트는 표시하려는 특정 게시물의 제목 및 컨텐츠와 같은 데이터를 전달할 수 없으면 유용하지 않습니다. `props` 가 필요한 건 바로 이때입니다.

`props` 은 컴포넌트에 등록할 수 있는 사용자 정의 속성입니다. 블로그 게시물 제목을 컴포넌트에 전달하려면, **`defineProps`** 매크로를 사용해야 합니다:

vue
```
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
</script>

<template>
  <h4>{{ title }}</h4>
</template>
```

`defineProps` 는 `<script setup>` 내에서만 사용할 수 있는 컴파일 타임 매크로이며, 템플릿에 선언된 `props` 는 자동으로 노출됩니다. (역자주: 컴파일러 매크로이기 때문에 개발 환경 설정에 따라 lint 에러나 경고가 나올수 있습니다) 그리고 `defineProps` 는 컴포넌트에 전달된 모든 `props` 를 객체로 반환하므로, 필요한 경우 JavaScript에서 접근할 수 있습니다:

js
```
const props = defineProps(['title'])
console.log(props.title)
```

참고: 컴포넌트 **Props** 타입 지정하기

`<script setup>` 을 사용하지 않는 경우, `props` 옵션을 선언해서 사용해야 하며, `props` 객체는 `setup()` 에 첫 번째 인자로 전달됩니다:

js
```
export default {
  props: ['title'],
  setup(props) {
    console.log(props.title)
  }
}
```

컴포넌트는 원하는 만큼 `props` 를 가질 수 있으며, 기본적으로 모든 값을 모두 `props` 에 전달할 수 있습니다.

`props` 가 등록되면, 다음과 같이 데이터를 사용자 정의 속성으로 전달할 수 있습니다:

template
```
<BlogPost title="Vue와 함께한 나의 여행" />
<BlogPost title="Vue로 블로깅하기" />
<BlogPost title="Vue가 재미있는 이유" />
```

그러나 일반적인 앱에서는 부모 컴포넌트에 다음과 같은 게시물 배열이 있을 수 있습니다:

js
```
const posts = ref([
  { id: 1, title: 'Vue와 함께한 나의 여행' },
  { id: 2, title: 'Vue로 블로깅하기' },
  { id: 3, title: 'Vue가 재미있는 이유' }
])
```

그런 다음 v-for 를 사용하여 각각을 컴포넌트로 렌더링하려고 합니다:

template

```
<BlogPost
  v-for="post in posts"
  :key="post.id"
  :title="post.title"
/>
```

온라인 연습장으로 실행하기

v-bind 구문 (:title="post.title")이 동적 prop 값을 전달하는 데 사용되는 것에 주목하세요. 이는 미리 정확히 렌더링할 내용을 모를 때 특히 유용합니다.

지금은 이것이 props 에 대해 알아야 할 전부입니다. 하지만 이 페이지를 다 읽고 내용에 익숙해지면 나중에 다시 돌아와 **Props**의 전체 가이드를 읽는 것이 좋습니다.

이벤트 청취하기

<BlogPost> 컴포넌트를 개발할 때 일부 기능은 상위 항목과 다시 통신해야 할 수 있습니다. 예를 들어, 페이지의 나머지 부분은 기본 크기로 유지하면서, 블로그 게시물의 텍스트를 확대하는 접근성 기능을 포함하기로 결정할 수 있습니다.

부모 컴포넌트에서 postFontSize ref를 추가하여 이 기능을 지원할 수 있습니다:

js

```
const posts = ref([
  /* ... */
])

const postFontSize = ref(1)
```

템플릿에서 모든 블로그 게시물의 글꼴 크기를 제어하는 데 사용할 수 있습니다:

template

```
<div :style="{ fontSize: postFontSize + 'em' }">
  <BlogPost
    v-for="post in posts"
    :key="post.id"
    :title="post.title"
  />
</div>
```

이제 <BlogPost> 컴포넌트의 템플릿에 버튼을 추가해 보겠습니다:

vue

```
<!-- BlogPost.vue의 <script> 생략 -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button>텍스트 확대</button>
  </div>
</template>
```

버튼은 아직 아무것도 하지 않습니다. 버튼을 클릭하여 모든 게시물의 텍스트를 확대해야 한다고 부모에게 알리고 싶습니다. 이 문제를 해결하기 위해 컴포넌트는 커스텀 이벤트 시스템을 제공합니다. 부모 컴포넌트는 네이티브 DOM 이벤트와 마찬가지로 v-on 또는 @ 를 사용하여 자식 컴포넌트 인스턴스의 모든 이벤트를 수신하도록 선택할 수 있습니다:

template

```
<BlogPost
  ...
```

```
  @enlarge-text="postFontSize += 0.1"
/>
```

그런 다음 자식 컴포넌트는 빌트인 **$emit** 메서드를 호출하고 이벤트 이름을 전달하여 자체적으로 이벤트를 생성할 수 있습니다:

```vue
<!-- BlogPost.vue의 <script> 생략 -->
<template>
  <div class="blog-post">
    <h4>{{ title }}</h4>
    <button @click="$emit('enlarge-text')">텍스트 확대</button>
  </div>
</template>
```

@enlarge-text="postFontSize += 0.1" 리스너 덕분에 부모 컴포넌트는 이벤트를 수신하고 postFontSize 값을 업데이트합니다.

온라인 연습장으로 실행하기

defineEmits 매크로를 사용하여 원하는 이벤트를 선언할 수 있습니다.:

```vue
<!-- BlogPost.vue -->
<script setup>
defineProps(['title'])
defineEmits(['enlarge-text'])
</script>
```

이것은 컴포넌트가 내보내는 모든 이벤트를 문서화하고 선택적으로 유효성 검사를 합니다. 또한 Vue가 자식 컴포넌트의 루트 엘리먼트에 암시적으로 네이티브 리스너가 적용되는 것을 방지할 수 있습니다.

defineProps 와 마찬가지로 defineEmits 도 <script setup> 에서만 사용할 수 있으며 import 할 필요가 없습니다. $emit 메서드와 동일한 emit 함수를 반환하므로, 컴포넌트의 <script setup> 섹션에서 이벤트를 내보내는 데 사용할 수 있습니다.

```vue
<script setup>
const emit = defineEmits(['enlarge-text'])

emit('enlarge-text')
</script>
```

참고: 컴포넌트 **emit** 타입 지정하기

<script setup> 을 사용하지 않는 경우, emits 옵션을 사용하여 내보낼 이벤트를 선언할 수 있습니다. setup 컨텍스트의 속성으로 emit 함수에 접근할 수 있습니다(setup() 의 두 번째 인자로 전달됨):

```js
export default {
  emits: ['enlarge-text'],
  setup(props, ctx) {
    ctx.emit('enlarge-text')
  }
}
```

지금은 이것이 사용자 정의 컴포넌트 이벤트에 대해 알아야 할 전부입니다. 그러나 이 페이지를 다 읽고 내용에 익숙해지면 나중에 다시 돌아와 사용자 정의 이벤트의 전체 가이드를 읽는 것이 좋습니다.

슬롯이 있는 컨텐츠 배포

HTML 엘리먼트와 마찬가지로 다음과 같이 컴포넌트에 컨텐츠를 전달할 수 있으면 종종 유용합니다:

template

```
<AlertBox>
 나쁜 일이 일어났습니다.
</AlertBox>
```

다음과 같이 렌더링할 수 있습니다:

이것은 데모용 에러입니다.

나쁜 일이 일어났습니다.

이것은 Vue의 사용자 정의 `<slot>` 엘리먼트를 사용하여 달성할 수 있습니다.

vue

```
<!-- AlertBox.vue -->
<template>
  <div class="alert-box">
    <strong>이것은 데모용 에러입니다.</strong>
    <slot />
  </div>
</template>

<style scoped>
.alert-box {
  /* ... */
}
</style>
```

위에서 볼 수 있듯이 컨텐츠를 이동하려는 자리 표시자로 `<slot>` 을 사용합니다. 우리가 할 일은 이게 끝입니다!

온라인 연습장으로 실행하기

지금은 이것이 슬롯에 대해 알아야 할 전부입니다. 그러나 이 페이지를 다 읽고 내용에 익숙해지면 나중에 다시 돌아와서 **Slots**의 전체 가이드를 읽는 것이 좋습니다.

동적 컴포넌트

때로는 탭 인터페이스와 같이 컴포넌트 간에 동적으로 전환하는 것이 유용할 수 있습니다:

온라인 연습장으로 예제보기

위의 내용은 Vue의 `<component>` 엘리먼트에 특별한 `is` 속성이 있어 가능합니다:

template

```
<!-- currentTab이 변경되면 컴포넌트가 변경됩니다 -->
<component :is="tabs[currentTab]"></component>
```

위의 예에서 `:is` 에 전달된 값은 다음 중 하나를 포함할 수 있습니다:

- 등록된 컴포넌트의 이름 문자열
- 실제 가져온 컴포넌트 객체

`is` 속성을 사용하여 일반 HTML 엘리먼트를 만들 수도 있습니다.

`<component :is="...">` 를 사용하여 여러 컴포넌트 간에 전환할 때, 다른 컴포넌트로 전환되면 컴포넌트가 마운트 해제됩니다. 내장된 **`<KeepAlive>`** 컴포넌트를 사용하여 비활성 컴포넌트를 "활성" 상태로 유지하도록 강제할 수 있습니다.

in-DOM 템플릿 파싱 주의 사항

Vue 템플릿을 DOM에서 직접 작성하는 경우, Vue는 DOM에서 템플릿 문자열을 검색해야 합니다. 이것은 브라우저의 기본 HTML 파싱 동작으로 인해 몇 가지 주의 사항으로 이어집니다.

> **TIP**
>
> 아래에 설명된 제한 사항은 템플릿을 DOM에서 직접 작성하는 경우에만 적용된다는 점에 유의해야 합니다. 다음 소스의 문자열 템플릿을 사용하는 경우에는 적용되지 않습니다:
>
> - 싱글 파일 컴포넌트(SFC)
> - 인라인 템플릿 문자열(예: `template: '...'`)
> - `<script type="text/x-template">`

대소문자를 구분하지 않음

HTML 태그와 속성의 이름은 대소문자를 구분하지 않으므로 브라우저는 대문자를 소문자로 해석합니다. 즉, DOM 내 템플릿을 사용할 때 PascalCase 컴포넌트 이름과 `props`의 camelCased 이름 또는 `v-on` 이벤트 이름은 모두 kebab-case(하이픈으로 구분된) 기반으로 사용해야 합니다:

```js
// JavaScript에서 camelCase
const BlogPost = {
  props: ['postTitle'],
  emits: ['updatePost'],
  template: `
    <h3>{{ postTitle }}</h3>
  `
}
```

```template
<!-- HTML에서 kebab-case -->
<blog-post post-title="안녕!" @update-post="onUpdatePost"></blog-post>
```

셀프 테그 닫기

컴포넌트에 자동 닫기 태그를 사용했습니다:

```template
<MyComponent />
```

Vue의 템플릿 파서는 유형에 관계없이 모든 태그를 닫으라는 표시로 `/>`를 허용하기 때문입니다.

그러나 in-DOM 템플릿에서는 항상 명시적인 닫는 태그를 포함해야 합니다.

```template
<my-component></my-component>
```

이는 HTML 사양에서 몇 가지 특정 엘리먼트가 닫는 태그를 생략할 수 있도록 허용하기 때문입니다. 가장 일반적인 것은 `<input>` 및 `<img>` 입니다. 다른 모든 엘리먼트의 경우, 닫는 태그를 생략하면 기본 HTML 파서는 사용자가 여는 태그를 종료하지 않은 것으로 간주합니다. 예를 들어, 스니펫은 다음과 같습니다:

```template
<my-component /> <!-- 우리는 여기서 태그를 닫으려 했습니다... -->
<span>hello</span>
```

하지만 아래와 같이 파싱됩니다:

template

```
<my-component>
  <span>안녕</span>
</my-component> <!-- 그러나 브라우저는 여기에서 닫을 것입니다. -->
```

엘리먼트 배치 제한

`<ul>`, `<ol>`, `<table>` 및 `<select>` 와 같은 일부 HTML 엘리먼트에는 내부에 표시할 수 있는 엘리먼트에 대한 제한이 있습니다. 또한 `<li>`, `<tr>` 및 `<option>` 와 같은 일부 엘리먼트는 특정 다른 엘리먼트 내부에만 사용할 수 있습니다.

이러한 제한이 있는 엘리먼트가 있는 컴포넌트를 사용할 때 문제가 발생합니다. 예를 들어:

template

```
<table>
  <blog-post-row></blog-post-row>
</table>
```

사용자 정의 컴포넌트 `<blog-post-row>` 는 잘못된 컨텐츠로 호이스트(hoisted)되어 최종적으로 렌더링된 출력에서 에러를 발생시킵니다. 특별한 `is` 속성을 해결 방법으로 사용할 수 있습니다:

template

```
<table>
  <tr is="vue:blog-post-row"></tr>
</table>
```

> **TIP**
>
> 기본 HTML 엘리먼트에 사용되는 경우, `is` 값은 Vue 컴포넌트로 해석되기 위해 `vue:` 접두사를 사용해야 합니다. 이는 기본 맞춤형 내장 엘리먼트와의 혼동을 피하기 위해 필요합니다.

이것이 현재로서는 Vue를 사용할 때 in-DOM 템플릿 파싱 경고에 대해 필수적으로 알아야 할 전부입니다. 축하합니다! 아직 배울 것이 더 있지만 먼저 잠시 휴식을 취하며 Vue를 직접 다뤄보는 것이 좋습니다. 재미있는 것을 빌드하거나 아직 확인하지 않았다면 몇 가지 예제를 확인해보세요.

방금 익힌 지식에 익숙해지면 가이드를 계속 진행하여 컴포넌트에 대해 자세히 알아보세요.