



렌더 함수 & JSX

Vue는 대부분의 경우 애플리케이션을 빌드할 때 템플릿 사용을 권장합니다. 하지만 JavaScript의 완전한 프로그래밍적 힘이 필요한 상황도 있습니다. 이럴 때 **렌더 함수**를 사용할 수 있습니다.

가상 DOM과 렌더 함수 개념이 처음이라면, 먼저 [렌더링 메커니즘](#) 챕터를 읽어보세요.

기본 사용법

Vnode 생성하기

Vue는 vnode를 생성하기 위한 `h()` 함수를 제공합니다:

```
import { h } from 'vue'

const vnode = h(
  'div', // 타입
  { id: 'foo', class: 'bar' }, // props
  [
    /* 자식 요소 */
  ]
)
```

js

`h()` 는 **hyperscript**의 약자입니다. 이는 "HTML(하이퍼텍스트 마크업 언어)을 생성하는 JavaScript"를 의미합니다. 이 이름은 많은 가상 DOM 구현에서 공유되는 관례에서 유래되었습니다. 더 설명적인 이름은 `createVNode()` 일 수 있지만, 렌더 함수에서 이 함수를 여러 번 호출해야 하므로 짧은 이름이 도움이 됩니다.

`h()` 함수는 매우 유연하게 설계되어 있습니다:

```
// 타입을 제외한 모든 인자는 선택 사항입니다
h('div')
h('div', { id: 'foo' })
```

js



```
// vue가 사용하는 기본 경로를 인식합니다
h('div', { class: 'bar', innerHTML: 'hello' })

// `.prop` 및 `.attr` 과 같은 props 수식자는
// 각각 `.` 및 `^` 접두사로 추가할 수 있습니다
h('div', { '.name': 'some-name', '^width': '100' })

// class와 style은 템플릿에서와 동일하게
// 객체/배열 값을 지원합니다
h('div', { class: [foo, { bar }], style: { color: 'red' } })

// 이벤트 리스너는 onXxx로 전달해야 합니다
h('div', { onClick: () => {} })

// 자식 요소는 문자열일 수 있습니다
h('div', { id: 'foo' }, 'hello')

// props가 없을 때는 props를 생략할 수 있습니다
h('div', 'hello')
h('div', [h('span', 'hello')])

// 자식 배열에는 vnode와 문자열이 혼합될 수 있습니다
h('div', ['hello', h('span', 'hello')])
```

생성된 vnode는 다음과 같은 형태를 가집니다:

```
const vnode = h('div', { id: 'foo' }, [])

vnode.type // 'div'
vnode.props // { id: 'foo' }
vnode.children // []
vnode.key // null
```

js

⚠ 참고

전체 **VNode** 인터페이스에는 이 외에도 많은 내부 속성이 있지만, 여기 나열된 속성 외에는 의존하지 않는 것이 강력히 권장됩니다. 내부 속성이 변경될 경우 예기치 않은 오류를 방지할 수 있습니다.

렌더 함수 선언하기

Composition API에서 템플릿을 사용할 때는 `setup()` 혹의 반환값이 템플릿에 데이터를 노출하는 데 사용됩니다. 하지만 렌더 함수를 사용할 때는, 렌더 함수를 직접 반환할 수 있습니다:

```
import { ref, h } from 'vue'

export default {
  props: {
```

js



```
},
setup(props) {
  const count = ref(1)

  // 렌더 함수를 반환합니다
  return () => h('div', props.msg + count.value)
}
```

렌더 함수는 `setup()` 내부에서 선언되므로, 동일한 스코프에서 선언된 `props`와 반응형 상태에 자연스럽게 접근할 수 있습니다.

단일 `vnode`를 반환하는 것 외에도, 문자열이나 배열을 반환할 수도 있습니다:

```
export default {
  setup() {
    return () => 'hello world!'
  }
}
```

js

```
import { h } from 'vue'

export default {
  setup() {
    // 배열을 사용하여 여러 루트 노드를 반환합니다
    return () => [
      h('div'),
      h('div'),
      h('div')
    ]
  }
}
```

js

TIP

값을 직접 반환하는 대신 반드시 함수를 반환해야 합니다! `setup()` 함수는 컴포넌트당 한 번만 호출되지만, 반환된 렌더 함수는 여러 번 호출됩니다.

렌더 함수 컴포넌트가 인스턴스 상태를 필요로 하지 않는 경우, 간결하게 함수로 직접 선언할 수도 있습니다:

```
function Hello() {
  return 'hello world!'
}
```

js



를 참고하세요.

Vnode는 고유해야 합니다

컴포넌트 트리의 모든 vnode는 고유해야 합니다. 즉, 다음과 같은 렌더 함수는 유효하지 않습니다:

```
function render() {  
  const p = h('p', 'hi')  
  return h('div', [  
    // 이런 - 중복된 vnode입니다!  
    p,  
    p  
  ])  
}
```

js

동일한 요소/컴포넌트를 여러 번 복제하고 싶다면, 팩토리 함수를 사용하면 됩니다. 예를 들어, 다음 렌더 함수는 20개의 동일한 단락을 렌더링하는 완전히 유효한 방법입니다:

```
function render() {  
  return h(  
    'div',  
    Array.from({ length: 20 }).map(() => {  
      return h('p', 'hi')  
    })  
  )  
}
```

js

JSX / TSX

JSX는 JavaScript에 XML과 유사한 확장 문법을 제공하여 다음과 같은 코드를 작성할 수 있게 해줍니다:

```
const vnode = <div>hello</div>
```

jsx

JSX 표현식 내부에서는 중괄호를 사용하여 동적 값을 삽입할 수 있습니다:

```
const vnode = <div id={dynamicId}>hello, {userName}</div>
```

jsx



하는 경우, [@vue/babel-plugin-jsx](#) 문서를 참고하세요.

JSX는 React에서 처음 도입되었지만, 실제로는 정의된 런타임 의미가 없으며 다양한 출력으로 컴파일될 수 있습니다. JSX를 사용해본 경험이 있다면, **Vue의 JSX 변환은 React의 JSX 변환과 다르다는 점**에 유의하세요. 따라서 React의 JSX 변환을 Vue 애플리케이션에서 사용할 수 없습니다. React JSX와의 주요 차이점은 다음과 같습니다:

`class` 와 `for` 와 같은 HTML 속성을 props로 사용할 수 있습니다. `className` 이나 `htmlFor` 를 사용할 필요가 없습니다.
컴포넌트에 자식(즉, 슬롯)을 전달하는 방식이 **다릅니다**.

Vue의 타입 정의는 TSX 사용 시 타입 추론도 제공합니다. TSX를 사용할 때는 `tsconfig.json` 에 `"jsx": "preserve"` 를 지정하여 TypeScript가 JSX 문법을 그대로 남겨두고 Vue JSX 변환이 처리할 수 있도록 해야 합니다.

JSX 타입 추론

변환과 마찬가지로, Vue의 JSX도 별도의 타입 정의가 필요합니다.

Vue 3.4부터는 더 이상 전역 `jsx` 네임스페이스를 암시적으로 등록하지 않습니다. TypeScript에 Vue의 JSX 타입 정의를 사용하도록 지시하려면, `tsconfig.json` 에 다음을 포함해야 합니다:

```

{
  "compilerOptions": {
    "jsx": "preserve",
    "jsxImportSource": "vue"
    // ...
  }
}
```

json

파일 단위로 적용하려면 파일 상단에 `/* @jsxImportSource vue */` 주석을 추가할 수도 있습니다.

전역 `jsx` 네임스페이스의 존재에 의존하는 코드가 있다면, 프로젝트에서 `vue/jsx` 를 명시적으로 import 또는 reference하여 3.4 이전의 전역 동작을 그대로 유지할 수 있습니다.

렌더 함수 레시피

아래에서는 템플릿 기능을 렌더 함수/JSX로 구현하는 일반적인 레시피를 제공합니다.



템플릿:

```
<div>
  <div v-if="ok">yes</div>
  <span v-else>no</span>
</div>
```

template

동등한 렌더 함수/JSX:

```
h('div', [ok.value ? h('div', 'yes') : h('span', 'no')])
```

js

```
<div>{ok.value ? <div>yes</div> : <span>no</span>}</div>
```

jsx

v-for

템플릿:

```
<ul>
  <li v-for="{ id, text } in items" :key="id">
    {{ text }}
  </li>
</ul>
```

template

동등한 렌더 함수/JSX:

```
h(
  'ul',
  // `items`가 배열 값을 가진 ref라고 가정
  items.value.map(({ id, text }) => {
    return h('li', { key: id }, text)
  })
)
```

js

```
<ul>
  {items.value.map(({ id, text }) => {
    return <li key={id}>{text}</li>
  })}
</ul>
```

jsx

v-on



어, `onClick` 은 템플릿의 `@click` 과 동일합니다.

```
h(  
  'button',  
  {  
    onClick(event) {  
      /* ... */  
    }  
  },  
  'Click Me'  
)
```

js

```
<button  
  onClick={ (event) => {  
    /* ... */  
  }}  
>  
  Click Me  
</button>
```

jsx

이벤트 수식자

`.passive` , `.capture` , `.once` 이벤트 수식자는 이벤트 이름 뒤에 camelCase로 연결할 수 있습니다.

예시:

```
h('input', {  
  onClickCapture() {  
    /* 캡처 모드의 리스너 */  
  },  
  onKeyUpOnce() {  
    /* 한 번만 트리거됨 */  
  },  
  onMouseoverOnceCapture() {  
    /* once + capture */  
  }  
})
```

js

```
<input  
  onClickCapture={() => {}}  
  onKeyUpOnce={() => {}}  
  onMouseoverOnceCapture={() => {}}  
>
```

jsx

기타 이벤트 및 키 수식자의 경우, `withModifiers` 헬퍼를 사용할 수 있습니다:



```
h('div', {
  onClick: withModifiers(() => {}, ['self'])
})
```

```
<div onClick={withModifiers(() => {}, ['self'])} />
```

jsx

컴포넌트

컴포넌트의 vnode를 생성하려면, `h()` 의 첫 번째 인자로 컴포넌트 정의를 전달해야 합니다. 즉, 렌더 함수를 사용할 때는 컴포넌트를 등록할 필요 없이, import한 컴포넌트를 바로 사용할 수 있습니다:

```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'

function render() {
  return h('div', [h(Foo), h(Bar)])
}
```

js

```
function render() {
  return (
    <div>
      <Foo />
      <Bar />
    </div>
  )
}
```

jsx

보시다시피, `h` 는 유효한 Vue 컴포넌트라면 어떤 파일 형식에서 import하든 사용할 수 있습니다.

동적 컴포넌트도 렌더 함수에서 간단하게 처리할 수 있습니다:

```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'

function render() {
  return ok.value ? h(Foo) : h(Bar)
}
```

js

```
function render() {
  return ok.value ? <Foo /> : <Bar />
}
```

jsx



우), `resolveComponent()` 헬퍼를 사용하여 프로그래밍적으로 해결할 수 있습니다.

슬롯 렌더링

렌더 함수에서 슬롯은 `setup()` 컨텍스트에서 접근할 수 있습니다. `slots` 객체의 각 슬롯은 **vnode 배열을 반환하는 함수**입니다:

```
export default {
  props: ['message'],
  setup(props, { slots }) {
    return () => [
      // 기본 슬롯:
      // <div><slot /></div>
      h('div', slots.default()),

      // 명명된 슬롯:
      // <div><slot name="footer" :text="message" /></div>
      h(
        'div',
        slots.footer({
          text: props.message
        })
      )
    ]
  }
}
```

js

JSX 동등 코드:

```
// 기본
<div>{slots.default()}</div>

// 명명된
<div>{slots.footer({ text: props.message })}</div>
```

jsx

슬롯 전달하기

컴포넌트에 자식을 전달하는 것은 요소에 자식을 전달하는 것과 약간 다릅니다. 배열 대신, 슬롯 함수 또는 슬롯 함수 객체를 전달해야 합니다. 슬롯 함수는 일반 렌더 함수가 반환할 수 있는 모든 것을 반환할 수 있으며, 자식 컴포넌트에서 접근할 때 항상 vnode 배열로 정규화됩니다.

```
// 단일 기본 슬롯
h(MyComponent, () => 'hello')

// 명명된 슬롯
```

js



```
// 글자 색제가 props로 처리되지 않습니다
h(MyComponent, null, {
  default: () => 'default slot',
  foo: () => h('div', 'foo'),
  bar: () => [h('span', 'one'), h('span', 'two')]
})
```

JSX 동등 코드:

```
// 기본 jsx
<MyComponent>{() => 'hello'}</MyComponent>

// 명명된
<MyComponent>{{
  default: () => 'default slot',
  foo: () => <div>foo</div>,
  bar: () => [<span>one</span>, <span>two</span>]
}}</MyComponent>
```

슬롯을 함수로 전달하면 자식 컴포넌트에서 지연 호출할 수 있습니다. 이로 인해 슬롯의 의존성이 부모가 아닌 자식에 의해 추적되어, 더 정확하고 효율적인 업데이트가 가능합니다.

스코프 슬롯

부모 컴포넌트에서 스코프 슬롯을 렌더링하려면, 슬롯을 자식에게 전달합니다. 이제 슬롯에 `text` 라는 매개변수가 있음을 주목하세요. 슬롯은 자식 컴포넌트에서 호출되며, 자식 컴포넌트의 데이터가 부모 컴포넌트로 전달됩니다.

```
// 부모 컴포넌트 js
export default {
  setup() {
    return () => h(MyComp, null, {
      default: ({ text }) => h('p', text)
    })
  }
}
```

슬롯이 props로 처리되지 않도록 `null` 을 전달하는 것을 잊지 마세요.

```
// 자식 컴포넌트 js
export default {
  setup(props, { slots }) {
    const text = ref('hi')
    return () => h('div', null, slots.default({ text: text.value }))
  }
}
```



```
}
```

JSX 동등 코드:

```
<MyComponent>{{  
  default: ({ text }) => <p>{ text }</p>  
}}</MyComponent>
```

jsx

내장 컴포넌트

내장 컴포넌트인 `<KeepAlive>` , `<Transition>` , `<TransitionGroup>` , `<Teleport>` , `<Suspense>` 등은 렌더 함수에서 사용하려면 import해야 합니다:

```
import { h, KeepAlive, Teleport, Transition, TransitionGroup } from 'vue'  
  
export default {  
  setup () {  
    return () => h(Transition, { mode: 'out-in' }, /* ... */)  
  }  
}
```

js

v-model

`v-model` 디렉티브는 템플릿 컴파일 시 `modelValue` 와 `onUpdate:modelValue` props로 확장됩니다. 따라서 이 props를 직접 제공해야 합니다:

```
export default {  
  props: ['modelValue'],  
  emits: ['update:modelValue'],  
  setup(props, { emit }) {  
    return () =>  
      h(SomeComponent, {  
        modelValue: props.modelValue,  
        'onUpdate:modelValue': (value) => emit('update:modelValue', value)  
      })  
  }  
}
```

js

커스텀 디렉티브

커스텀 디렉티브는 `withDirectives` 를 사용하여 vnode에 적용할 수 있습니다:



```
// 커스텀 디렉티브
const pin = {
  mounted() { /* ... */ },
  updated() { /* ... */ }
}

// <div v-pin:top.animate="200"></div>
const vnode = withDirectives(h('div'), [
  [pin, 200, 'top', { animate: true }]
])
```

디렉티브가 이름으로 등록되어 직접 import할 수 없는 경우, `resolveDirective` 헬퍼를 사용하여 해결할 수 있습니다.

템플릿 ref

Composition API에서 `useTemplateRef()` ^{3.5+}를 사용할 때, 템플릿 ref는 문자열 값을 vnode의 prop으로 전달하여 생성합니다:

```
import { h, useTemplateRef } from 'vue'

export default {
  setup() {
    const divEl = useTemplateRef('my-div')

    // <div ref="my-div">
    return () => h('div', { ref: 'my-div' })
  }
}
```

js

▶ 3.5 이전 버전에서의 사용법

함수형 컴포넌트

함수형 컴포넌트는 자체 상태가 없는 컴포넌트의 대안 형태입니다. 이들은 순수 함수처럼 동작합니다: props를 입력받아 vnode를 출력합니다. 컴포넌트 인스턴스를 생성하지 않고(즉, `this`가 없음), 일반적인 컴포넌트 라이프사이클 훅도 없습니다.

함수형 컴포넌트를 만들려면 옵션 객체 대신 일반 함수를 사용합니다. 이 함수는 사실상 컴포넌트의 `render` 함수입니다.



```
function MyComponent(props, { slots, emit, attrs }) {  
  // ...  
}
```

js

함수형 컴포넌트에는 대부분의 일반 컴포넌트 구성 옵션을 사용할 수 없습니다. 하지만 `props` 와 `emits` 는 속성으로 추가하여 정의할 수 있습니다:

```
MyComponent.props = ['value']  
MyComponent.emits = ['click']
```

js

`props` 옵션이 지정되지 않은 경우, 함수에 전달되는 `props` 객체에는 모든 속성이 포함되며, 이는 `attrs` 와 동일합니다. `props` 옵션이 지정되지 않으면 `prop` 이름이 camelCase로 정규화되지 않습니다.

명시적 `props` 가 있는 함수형 컴포넌트의 경우, 속성 전달은 일반 컴포넌트와 거의 동일하게 동작합니다. 하지만 `props` 를 명시적으로 지정하지 않은 함수형 컴포넌트의 경우, 기본적으로 `class` , `style` , `onXxx` 이벤트 리스너만 `attrs` 에서 상속됩니다. 두 경우 모두, `inheritAttrs` 를 `false` 로 설정하여 속성 상속을 비활성화할 수 있습니다:

```
MyComponent.inheritAttrs = false
```

js

함수형 컴포넌트는 일반 컴포넌트처럼 등록하고 사용할 수 있습니다. `h()` 의 첫 번째 인자로 함수를 전달하면, 함수형 컴포넌트로 처리됩니다.

함수형 컴포넌트 타입 지정 ^{TS}

함수형 컴포넌트는 명명된 컴포넌트인지 익명 컴포넌트인지에 따라 타입을 지정할 수 있습니다. **Vue - 공식 확장**도 SFC 템플릿에서 타입이 올바르게 지정된 함수형 컴포넌트의 타입 검사를 지원합니다.

명명된 함수형 컴포넌트

```
import type { SetupContext } from 'vue'  
type FComponentProps = {  
  message: string  
}  
  
type Events = {  
  sendMessage(message: string): void  
}
```

tsx



```
    props: FComponentProps,  
    context: SetupContext<Events>  
  ) {  
    return (  
      <button onClick={() => context.emit('sendMessage', props.message)}>  
        {props.message} {' '  
      </button>  
    )  
  }  
}  
  
FComponent.props = {  
  message: {  
    type: String,  
    required: true  
  }  
}  
  
FComponent.emits = {  
  sendMessage: (value: unknown) => typeof value === 'string'  
}
```

익명 함수형 컴포넌트

```
import type { FunctionalComponent } from 'vue'
```

tsx

```
type FComponentProps = {  
  message: string  
}
```

```
type Events = {  
  sendMessage(message: string): void  
}
```

```
const FComponent: FunctionalComponent<FComponentProps, Events> = (  
  props,  
  context  
) => {  
  return (  
    <button onClick={() => context.emit('sendMessage', props.message)}>  
      {props.message} {' '  
    </button>  
  )  
}
```

```
FComponent.props = {  
  message: {  
    type: String,  
    required: true  
  }  
}
```

```
FComponent.emits = {
```



 [GitHub에서 이 페이지 편집](#)

[< Previous](#)

렌더링 메커니즘

[Next >](#)

Vue와 웹 컴포넌트