[CSR 과 SSR]

	CSR	SSR
장점	- 화면 깜빡임이 없음 - 초기 로딩 이후 구동 속도가 빠름 - TTV와 TTI 사이 간극이 없음 - 서버 부하 분산	- 초기 구동 속도가 빠름 - SEO에 유리함
단점	- 초기 로딩 속도가 느림 - SEO에 불리함	- 화면 깜빡임이 있음 - TTV와 TTI 사이 간극이 있음 - 서버 부하가 있음

만약 서비스가 사용자와의 상호작용이 많고, 대부분 페이지가 고객의 개인정보 기반이라면 SEO보다 데이터 보호측면이 중요. (모든 서비스에 SEO가 필요하지 않다) → CSR이 적합

만약 회사 홈페이지와 같은 노출에 신경써야 하는 페이지나 누구에게나 같은 페이지를 공유해야 하는 서비스라면 (+ 업데이트도 자주 필요한)→ SSR이 적합

구동방식, 렌더링 방식 선택기준

그렇다면, 어떤 방식을 쓰는게 좋을까?

결론: 서비스, 프로젝트, 콘텐츠의 성격에 따라 달라진다

상황 1

1) 상위 노출이 필요하거나
2) 누구에게나 동일한 내용을 노출하거나
3) 페이지마다 데이터가 자주 바뀐다면?

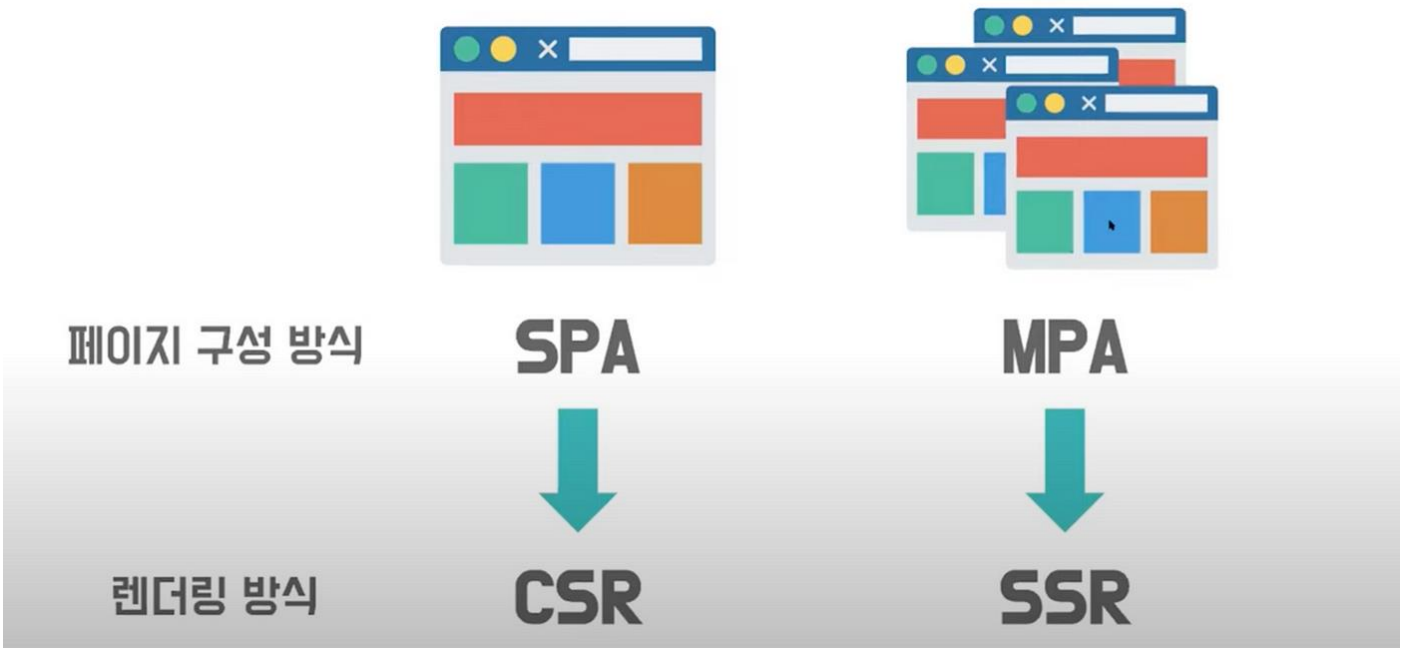
→ **Server Side Rendering**

상황 2

1) 개인정보 데이터를 기준으로 구성되거나
2) 보다 나은 사용자 경험을 제공하고 싶거나
3) 상위노출보다 고객의 데이터 보호가 더 중요하다면

→ **Client Side Rendering**


빠른 인터랙션이 중요해요.
화면 깜빡임 안돼요.
상위 노출되면 좋겠어요.
사용자에 따라서 페이지
내용이 달라져요.
CSR + SSR !


페이지 구성 방식
SPA
MPA
렌더링 방식
CSR
SSR

[SPA 를 구성하는 이론]

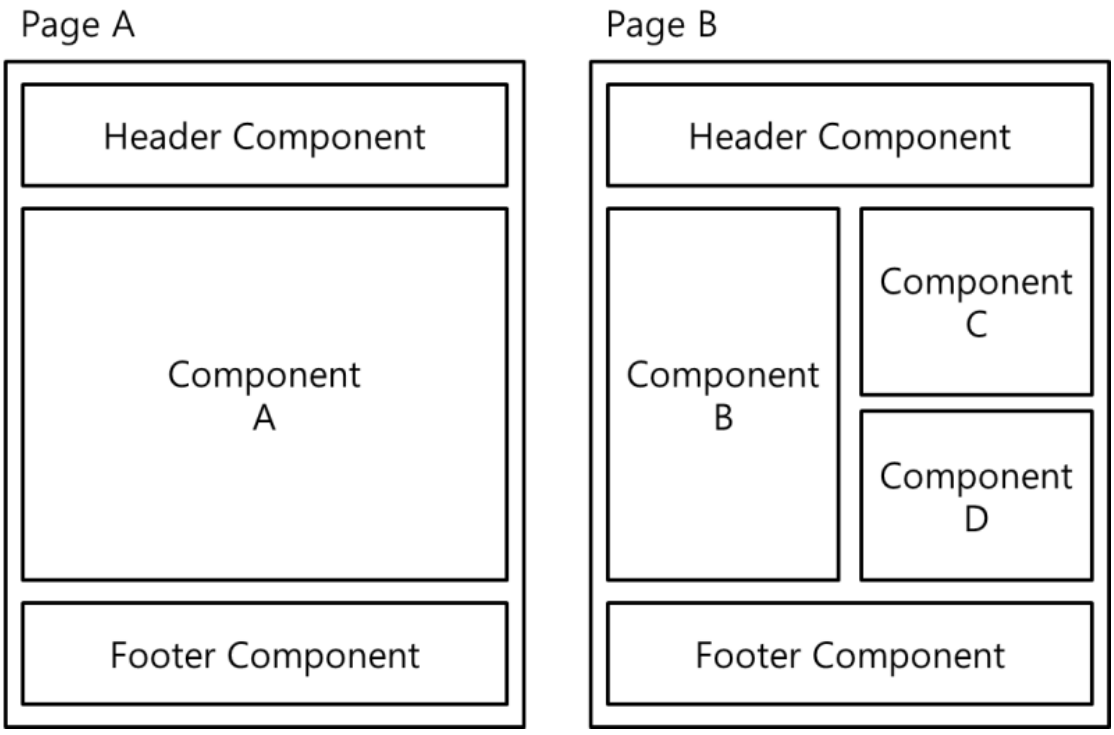
- 라우팅(Routing)

라우팅은 출발지에서 목적지까지의 경로를 결정하는 기능입니다. 애플리케이션의 라우팅은 사용자가 태스크를 수행하기 위해 어떤 화면(view)에서 다른 화면으로 화면을 전환하는 내비게이션을 관리하기 위한 기능을 의미합니다. 일반적으로 사용자가 요청한 URL 또는 이벤트를 해석하고 새로운 페이지로 전환하기 위한 데이터를 취득하기 위해 서버에 필요 데이터를 요청하고 화면을 전환하는 위한 일련의 행위를 말합니다.

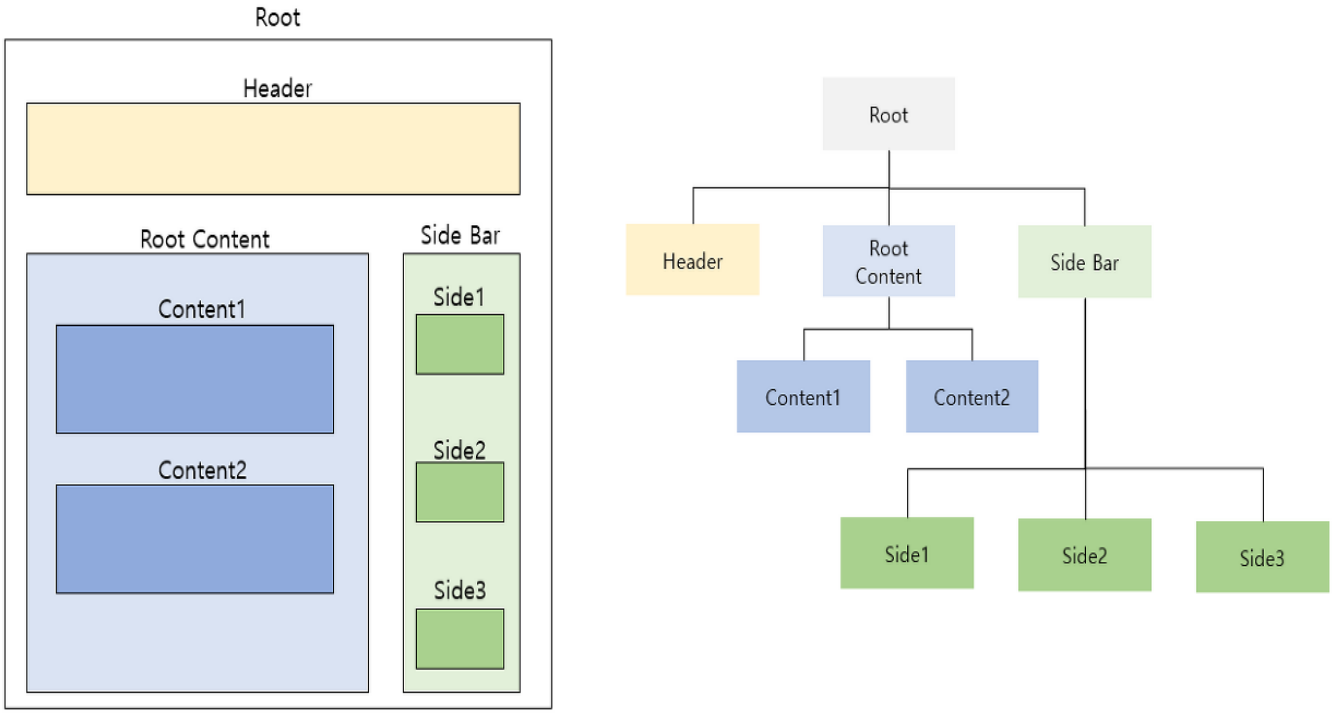
이러한 라우팅을 서버가 아닌 브라우저 단에서 구현해야 하는것이 SPA 개발의 핵심입니다. 이 방식을 쉽게 말하자면, 요청 URI에 따라 브라우저에서 돔(DOM)을 변경하는 방식입니다. 따라서 요청 경로에 따라 동적으로 렌더링 되도록 만들면 라우팅에 따라 다른 화면을 구현할 수 있는 것입니다.

- 컴포넌트(Component)

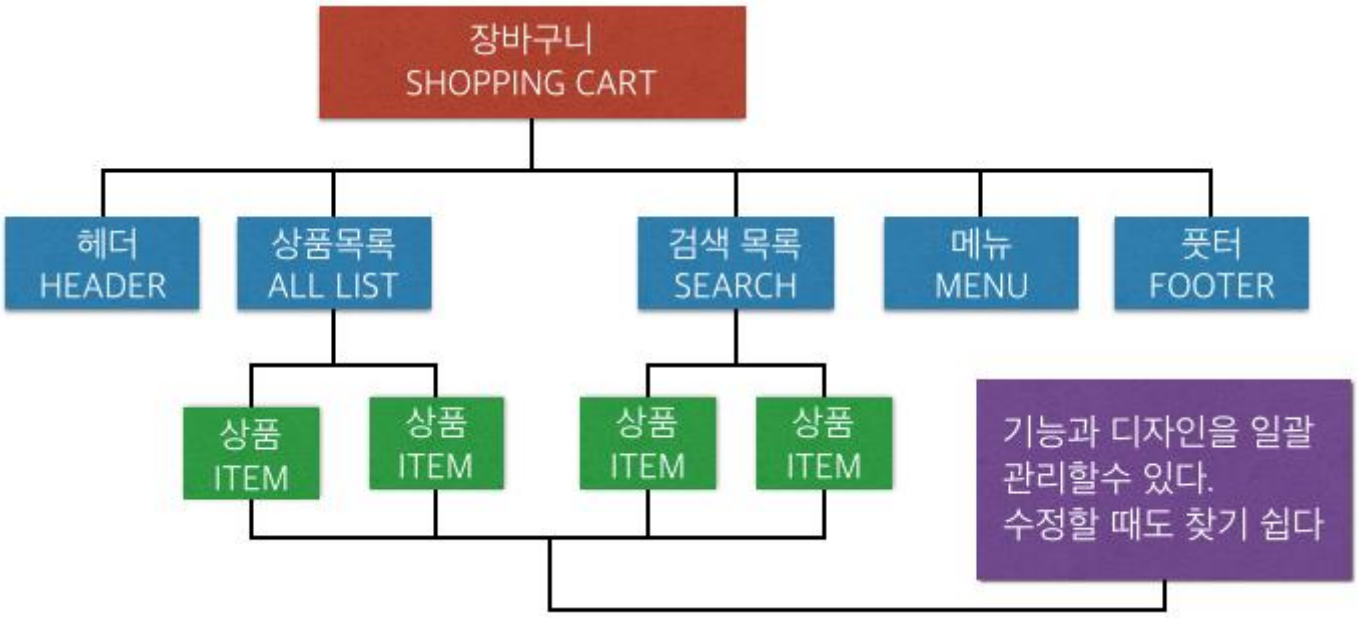
컴포넌트들이 모여 한 페이지를 구성하고, 특정 부분만 데이터를 바인딩하는 개념입니다. 쉽게 말하면 index.html 파일 하나에서 js, css등 리소스 파일들과 모듈들을 로드해서 페이지 이동 없이 특정 영역만 새로 모듈을 호출하고 데이터를 바인딩하는 개념입니다.



[컴포넌트로 구현하는 웹 페이지의 구성과 계층구조]



[구현하려는 화면 구성과 관련하여 정의할 수 있는 컴포넌트 구조의 예]



반응형 변수

Vue 3의 Composition API 를 사용해서 뷰 컴포넌트를 개발할 때 ~~~
Vue 3의 setup() 함수에서 만들어진 에서 변수들은 값이 변경되었을 때 화면이 자동으로 갱신되지 않는다. 데이터 변경에 반응형으로 자동 갱신을 하기 위해서는 ref 또는 reactive를 사용해야 한다.

ref 값을 변경할 때는 변수명.value 값을 변경하면 되며, reactive는 객체의 속성 값을 변경하면 된다. reactive는 primitive 값에 대해서는 반응형을 갖지 않는다. 다만 구조체 형태로 선언하면 반응형으로 동작한다.

```
<template>
  <div>
    {{refCount}}
    <br/>
    {{reactiveState.count}}
  </div>
</template>

<script>
  import { reactive, ref } from 'vue';

  export default {
    setup() {
        const refCount = ref(0);
        const increaseRefCount = () => {
                refCount.value++; // .value 속성을 사용하여 값을 가져오고 설정
        };
        const reactiveState = reactive({
          count: 0
        });
        const increaseReactiveStateCount = () => {
                reactiveState.count++;
        };
        return {
                refCount, reactiveState,
        }
    },
}
```

<script setup>

SFC(Single-File Components) 내에서 Composition API를 사용하기 위한 컴파일 타임 구문 설탕

compile-time syntactic sugar

```
<script setup>
    // 컴파일 될 때 Setup() 함수에 들어가는 내용이 된다..
</script>
```

- 변수 선언, 함수 정의, import 처리 등 모두 템플릿 영역에서 사용되도록 자동 바인딩 된다.

```
<script setup>
    const msg = 'Hello!'
    function log() {
        console.log(msg)
    }
</script>

<template>
    <button @click="log">{{ msg }}</button>
</template>
```

```
<script setup>
    import { capitalize } from './helpers'
</script>

<template>
    <div>{{ capitalize('hello') }}</div>
</template>
```

- 반응성 변수들을 사용해야 템플릿과 바인딩 된다.

```
<script setup>
    import { ref } from 'vue'
    const count = ref(0)
</script>

<template>
    <button @click="count++">{{ count }} </button>
</template>
```

- import 된 컴포넌트도 컴포넌트 등록을 하지 않아도 템플릿에서 태그로 사용 가능하다.

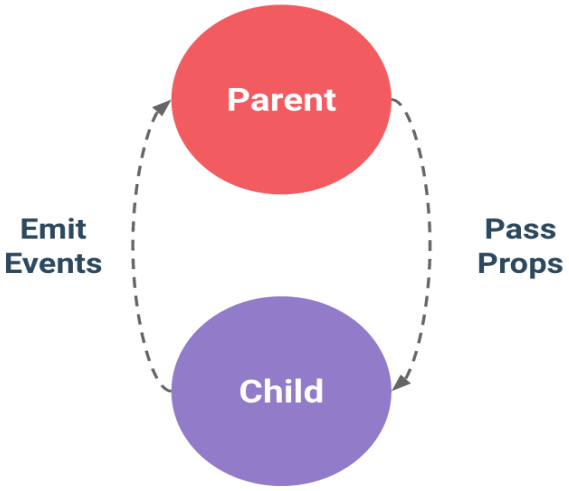
```
<script setup>
    import MyComponent from './MyComponent.vue'
</script>

<template>
    <MyComponent />
</template>
```

- defineProps() & defineEmits() 을 사용해서 부모 컴포넌트와 데이터를 송수신 할 수 있다.

```
<script setup>
    const props = defineProps({
        foo: String
    })

    const emit = defineEmits(['change', 'delete'])
    // setup code
</script>
```

[Vue 3 의 컴포넌트간의 통신 방법]

(1) props
(2) emit
(3) v-model
(4) refs
(5) provide/inject
(6) eventBus
(7) vuex/pinia



- Vue 3.0 의 Emit 구현 방법
emit는 컴포넌트에서 발생한 이벤트를 상위 부모에게 전달하는 키워드이다. 다음과 같이 script 코드에서 이벤트를 발생하는 방법과 template 코드에서 이벤트를 발생시키는 2가지 방법이 있다.

[emit 예제(1)]

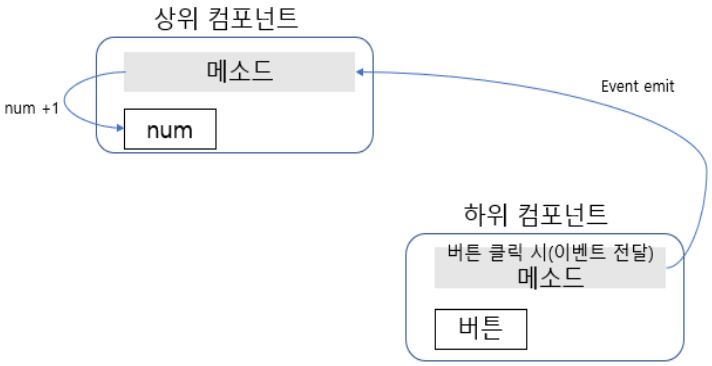
```
<template>
  <h2>Child Component</h2>

  <button @click="onClicked">Emit from script</button>
  <button @click="$emit('event2', 2)">Emit from template</button>
</template>

<script>
export default {
  emits: ["event1", "event2"],
  setup(props, context) {
```

```
    const { emit } = context;
    const onClicked = () => {
      emit("event1", 10);
    };
    return {
      onClicked,
    };
  },
};
</script>
```

[emit 예제(2) : getCurrentInstance 사용]

emit는 getCurrentInstance()를 이용해서도 획득할 수 있다. 기존에는 useContext를 이용해서 획득할 수 있었는데, Vue 3.2.0 버전부터 useContext는 deprecated 되었다.

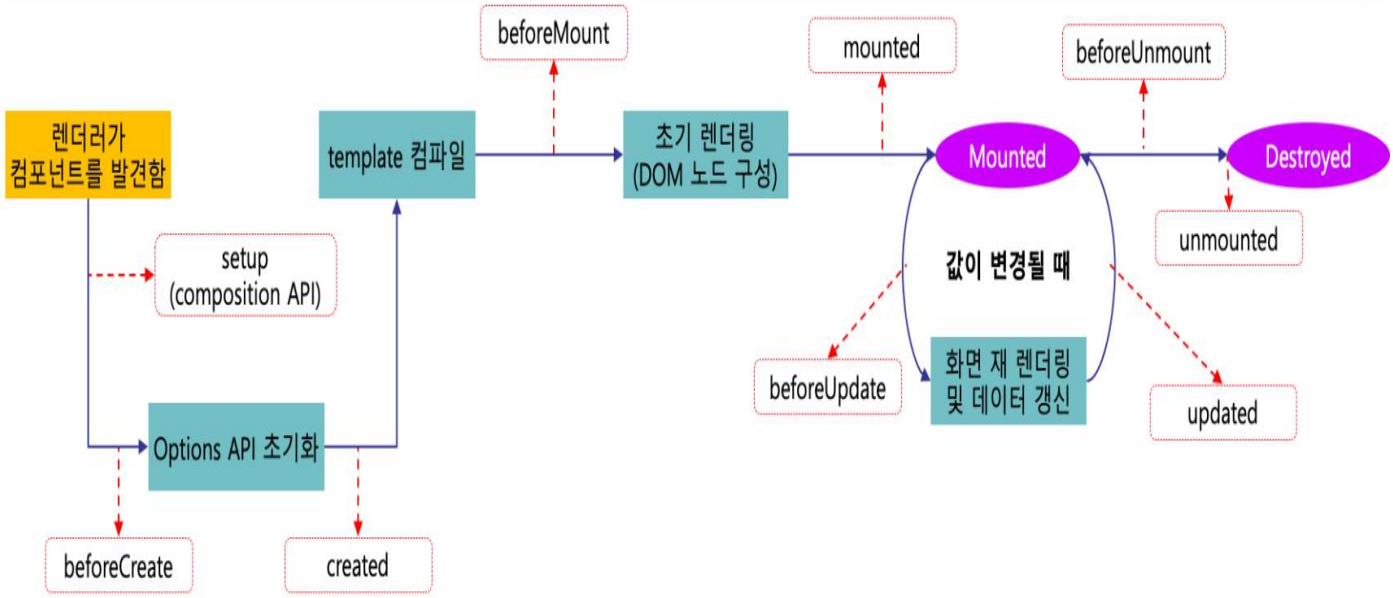
```
import { getCurrentInstance } from "vue";

export default {
  emits: ["event1", "event2"],
  setup() {
    const { emit } = getCurrentInstance();
    const onClicked = () => {
      emit("event1", 10);
    };
    return {
      onClicked,
    };
  },
};
```



[emit 예제(3) : <script setup> 내에서 사용하는 방법]

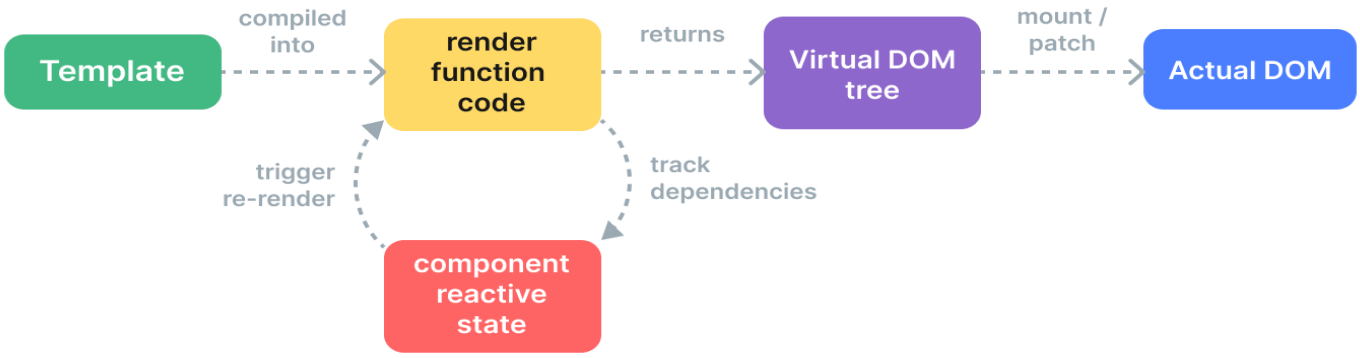
```
<script setup>
import { defineEmits } from "vue";

const emit = defineEmits(["커스텀이벤트명"]);
const toggleDrawer = () => {
  emit("커스텀이벤트명");
};
</script>
```

[Vue 컴포넌트 랜더링의 전체 흐름]



[Vue 컴포넌트가 마운트 되면 내부적으로 처리되는 과정]

Vue 컴포넌트가 마운트 되면 다음과 같은 일이 발생한다.

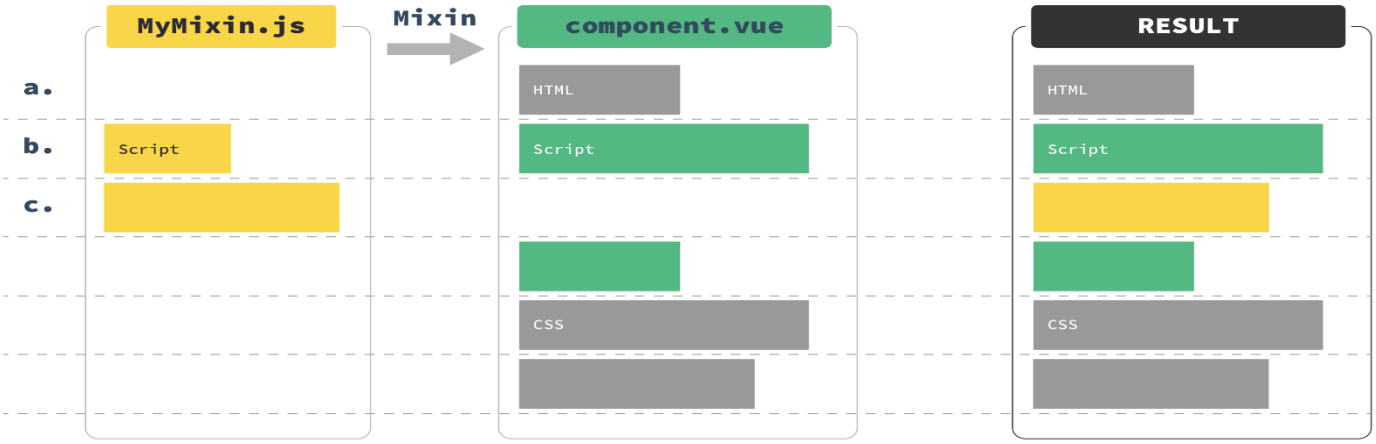


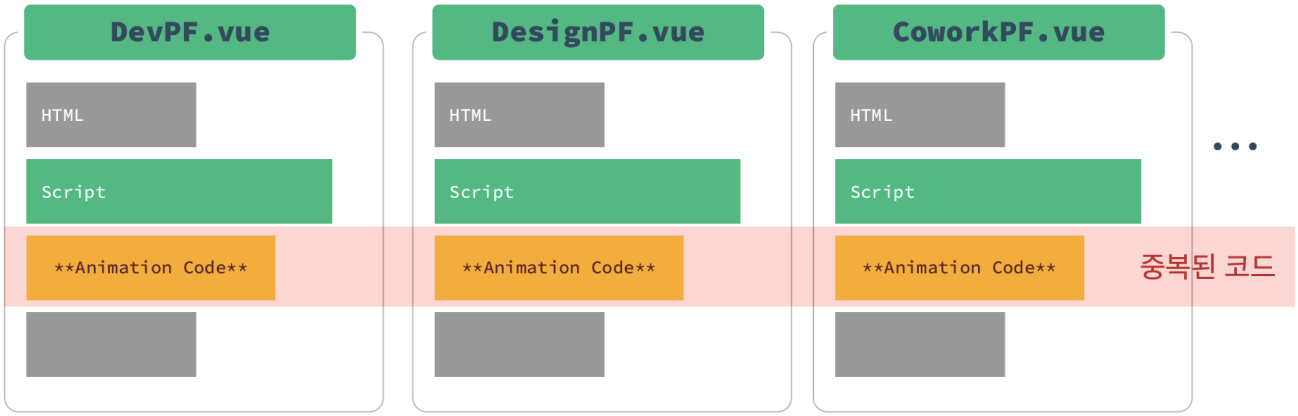
컴파일 : Vue 템플릿은 렌더링 함수 로 컴파일된다 . 즉, 가상 DOM 트리를 반환하는 함수이다.
이 단계는 빌드 단계를 통해 미리 수행하거나 런타임 컴파일러를 사용하여 즉석에서 수행할 수 있다.

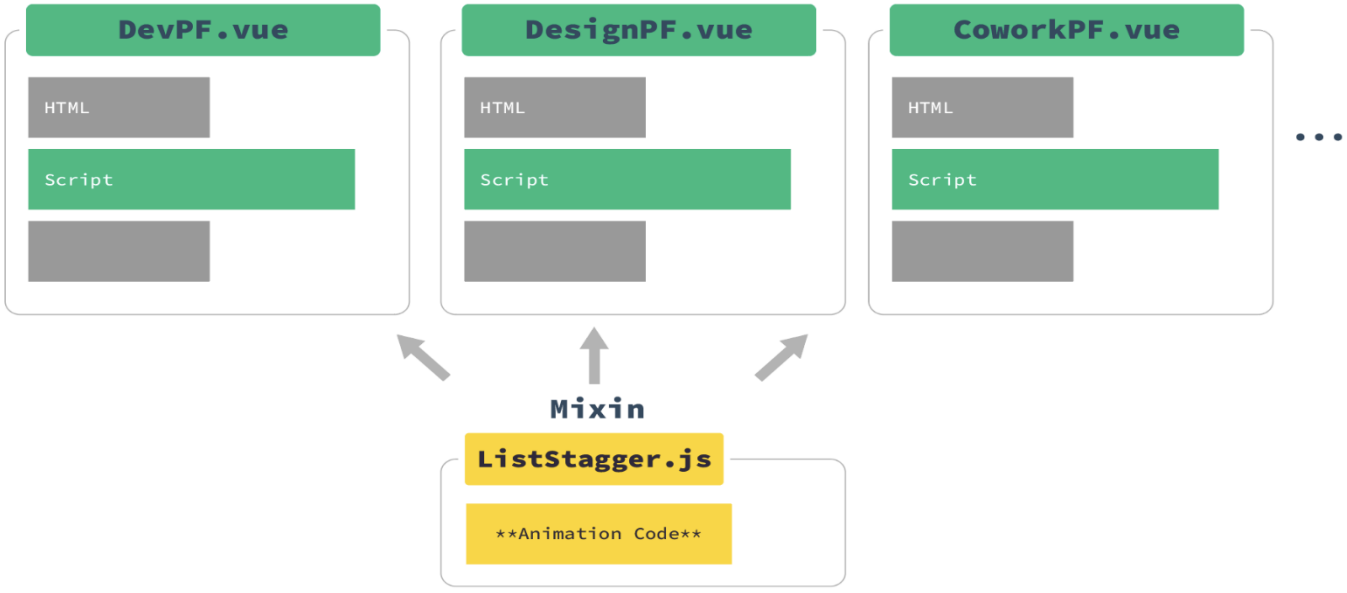
마운트 : 런타임 렌더러는 렌더링 기능을 호출하고 반환된 가상 DOM 트리를 탐색하며 이를 기반으로 실제 DOM 노드를 생성한다. 이 단계는 반응 효과 로 수행되므로 사용된 모든 반응 종속성을 추적한다.

패치 : 마운트 시 사용된 종속성(반응형 변수)이 변경되면 관련된 요소들이 다시 실행된다. 이번에는 업데이트된 새로운 Virtual DOM 트리가 생성된다. 런타임 렌더러는 새 트리를 탐색하고 이전 트리와 비교하고 필요한 업데이트를 실제 DOM에 적용한다.

[믹스인(mixin)]


MyMixin.js
Mixin
component.vue
RESULT
a.
b.
c.
Script
HTML
Script
CSS
HTML
Script
CSS


DevPF.vue
DesignPF.vue
CoworkPF.vue
HTML
Script
Animation Code
중복된 코드


DevPF.vue
DesignPF.vue
CoworkPF.vue
HTML
Script
Mixin
ListStagger.js
Animation Code