

Vue.js 실용 문법

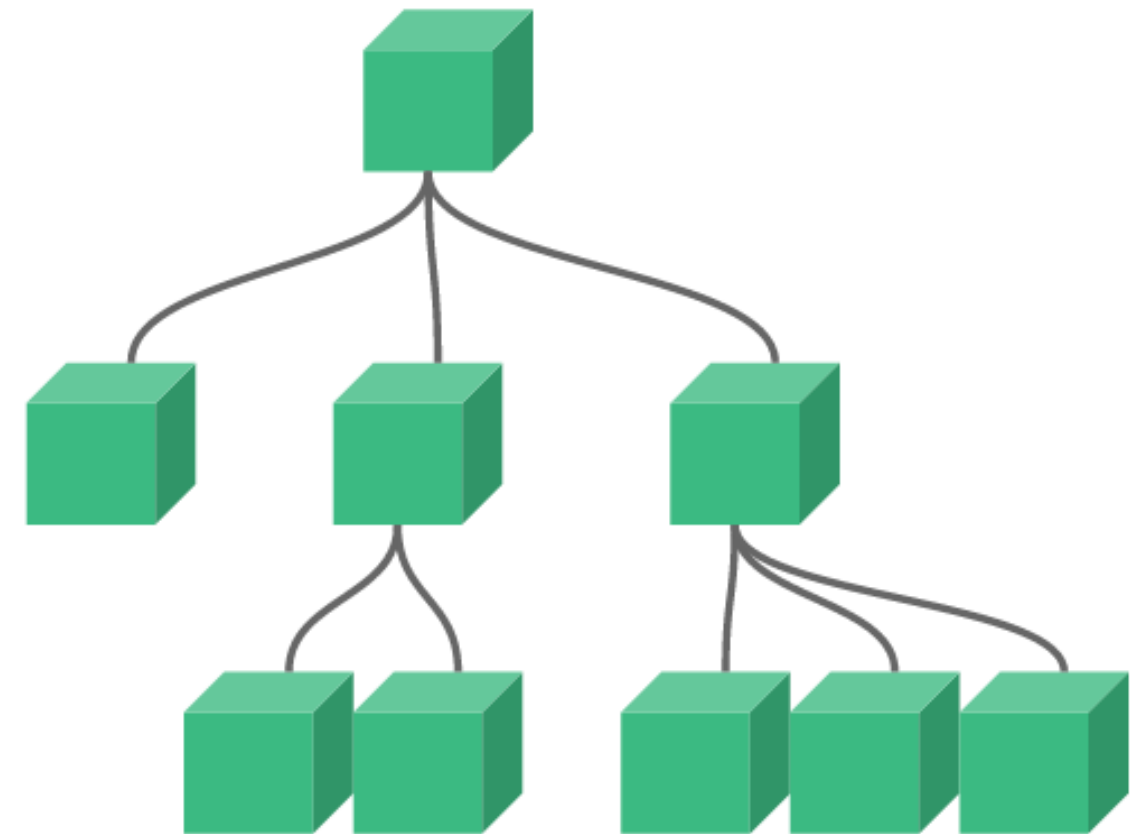
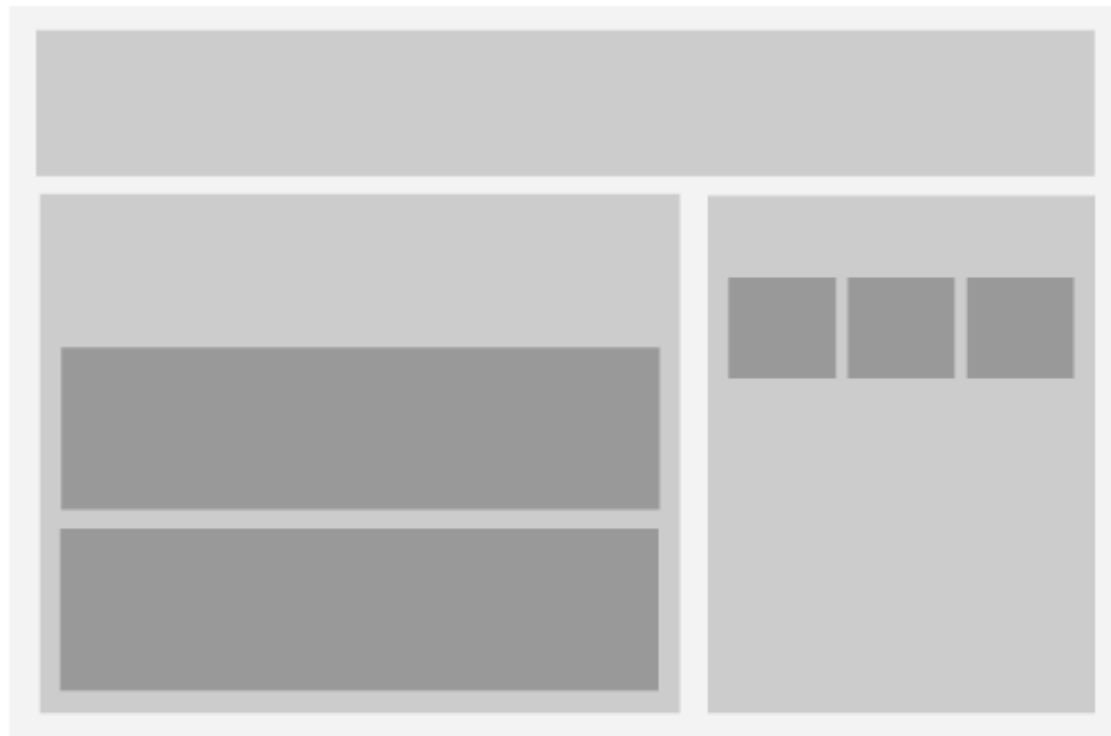
목차

- ▶ 컴포넌트 통신과 이벤트 버스
- ▶ Smart Watch
- ▶ Intuitive Computed
- ▶ Render Function
- ▶ 컴포넌트 스타일링 팁
- ▶ Form 제작

컴포넌트 통신과 이벤트 버스

컴포넌트란?

화면의 특정 영역을 **재사용** 가능한 형태로 구성한 코드



지역, 전역 컴포넌트

// 전역 컴포넌트

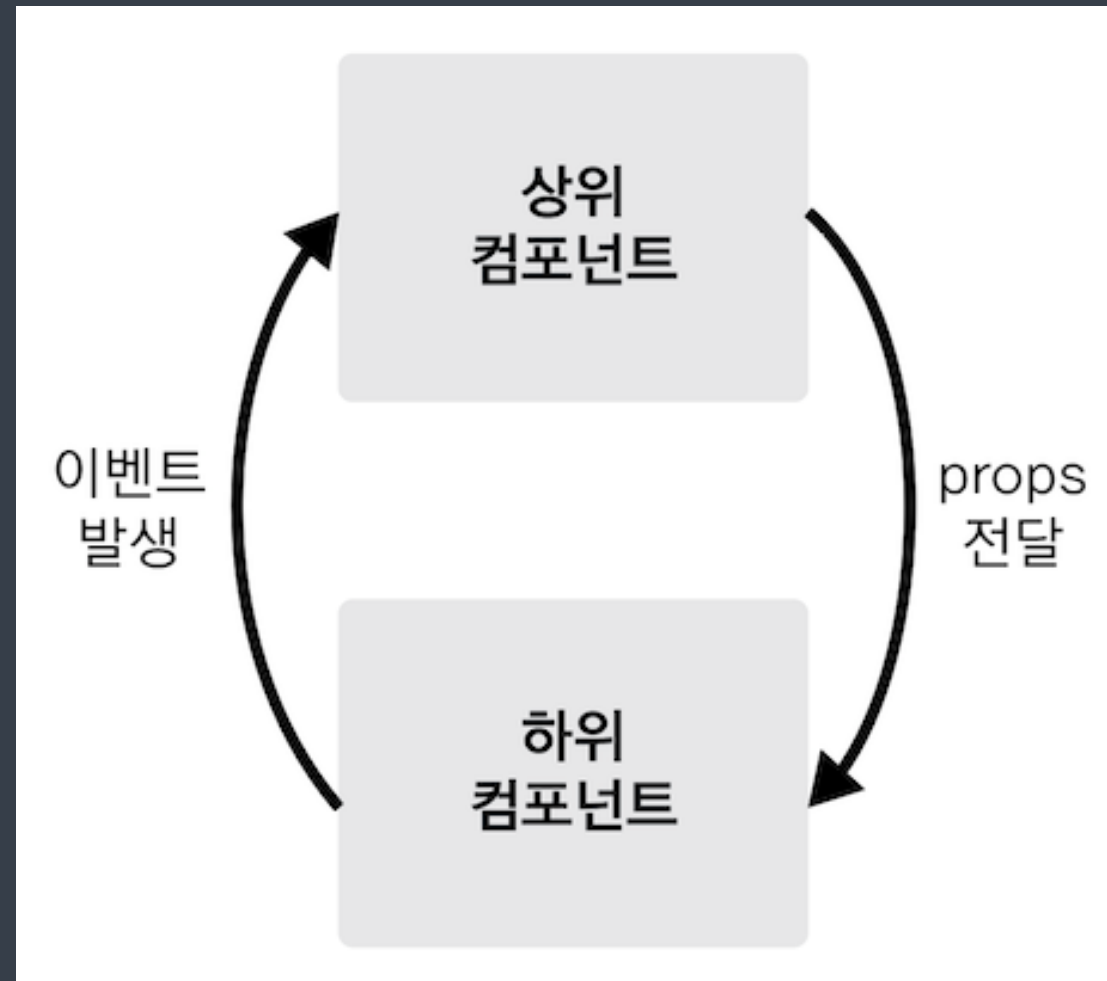
```
Vue.component('컴포넌트 이름', 컴포넌트 내용);
```

// 지역 컴포넌트

```
new Vue({  
  components: {  
    '컴포넌트 이름': 컴포넌트 내용  
  }  
});
```

컴포넌트 통신 방법

위에서 아래는 **Props**, 아래에서 위는 **Event Emit**



Props

```
<div id="app">  
  <child-component just-string="hi"></child-component>  
</div>
```

```
components: {  
  'child-component': {  
    props: ['justString'],  
    template: '<p>{{ justString }}</p>'  
  }  
}
```

Props

```
<div id="app">  
  <child-component just-string="hi"></child-component>  
</div>
```

```
components: {  
  'child-component': {  
    props: ['justString'],  
    template: '<p>{{ justString }}</p>'  
  }  
}
```


Props

```
<div id="app">  
  <child-component just-string="hi"></child-component>  
</div>
```

```
components: {  
  'child-component': {  
    props: ['justString'],  
    template: '<p>{{ justString }}</p>'  
  }  
}
```

Props Validation

```
// 최소한 타입이라도  
props: {  
  justString: String  
}
```

```
// 그리고 더 자세하게  
props: {  
  justString: {  
    type: String,  
    required: true,  
    ...  
  }  
}
```

Event Emit

```
<div id="app">  
  <child-component v-on:add="addCount"></child-component>  
</div>
```

```
// ChildComponent - 하위 컴포넌트  
this.$emit('add');
```

```
// RootComponent - 상위 컴포넌트  
methods: {  
  addCount: function() { ... }  
}
```

Event Emit

```
<div id="app">  
  <child-component v-on:add="addCount"></child-component>  
</div>
```

```
// ChildComponent - 하위 컴포넌트  
this.$emit('add');
```

```
// RootComponent - 상위 컴포넌트  
methods: {  
  addCount: function() { ... }  
}
```

Event Emit

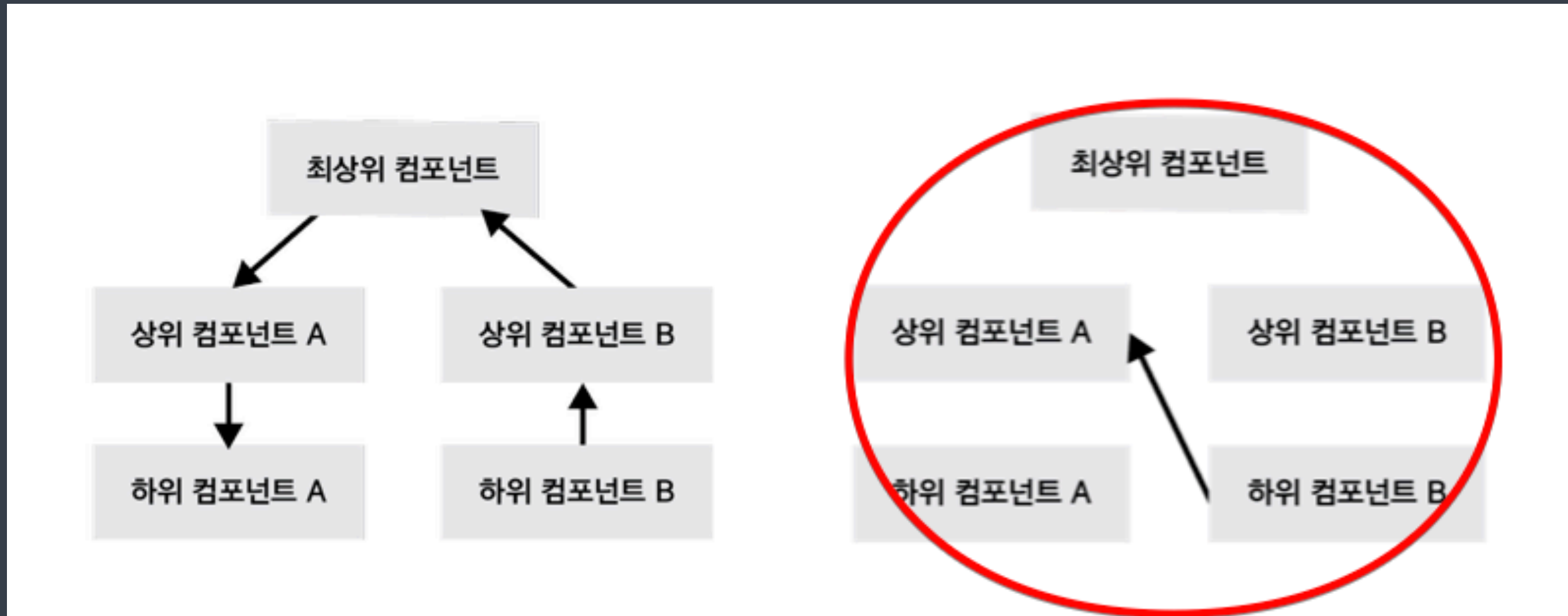
```
<div id="app">  
  <child-component v-on:add="addCount"></child-component>  
</div>
```

```
// ChildComponent - 하위 컴포넌트  
this.$emit('add');
```

```
// RootComponent - 상위 컴포넌트  
methods: {  
  addCount: function() { ... }  
}
```

Event Bus

컴포넌트 통신의 기본 규칙을 따르지 않고 **특정 컴포넌트** 간에 통신하는 방법



Event Bus

```
var bus = new Vue();
```

```
// 이벤트를 보낼 컴포넌트의 메서드  
bus.$emit('show');
```

```
// 이벤트를 받는 컴포넌트의 라이프 사이클 훅  
bus.$on('show', callbackFunction);
```

Event Bus 사용시 주의할 점

쓰고 나면 반드시 `$off()`로 **해제**. 안그러면 아래와 같이 적체가..

```
▼ _events:
  ▼ show: Array(3)
    ► 0: f ()
    ► 1: f ()
    ► 2: f ()
```


컴포넌트 통신 방법 vs Event Bus

- ▶ Event Bus가 없어도 서비스는 구현할 수 있습니다.

컴포넌트 통신 방법 vs Event Bus

- ▶ Event Bus가 없어도 서비스는 구현할 수 있습니다.
- ▶ 컴포넌트 레벨이 2단계 이하면 props, event emit을 쓰는 것이 좋아요.

컴포넌트 통신 방법 vs Event Bus

- ▶ Event Bus가 없어도 서비스는 구현할 수 있습니다.
- ▶ 컴포넌트 레벨이 2단계 이하면 props, event emit을 쓰는 것이 좋아요.
- ▶ 토스트 팝업, 프로그레스 바와 같이 on, off 성격의 UI 컴포넌트에 활용

컴포넌트 통신 방법 vs Event Bus

- ▶ Event Bus가 없어도 서비스는 구현할 수 있습니다.
- ▶ 컴포넌트 레벨이 2단계 이하면 props, event emit을 쓰는 것이 좋아요.
- ▶ 토스트 팝업, 프로그레스 바와 같이 on, off 성격의 UI 컴포넌트에 활용

프로젝트 여건과 리소스에 맞춰 최대한 클린 코드를 지향!

Smart Watch

Watch

공식 사이트에 안내된 기본 문법

```
watch: {  
  // whenever question changes, this function will run  
  question: function (newQuestion, oldQuestion) {  
    this.answer = 'Waiting for you to stop typing...'  
    this.debouncedGetAnswer()  
  }  
},
```

Watch

실제 기본 문법은 이렇습니다

```
data: {  
  num: 0  
}  
  
watch: {  
  num: function(newValue, oldValue) {  
    // ...  
  }  
}
```

Watch의 숨겨진 속성들

```
data: {  
  num: 0  
}  
  
watch: {  
  num: {  
    handler: function(newValue, oldValue) {},  
    immediate: true, // created 라이프 사이클 후 대신  
    deep: true, // 객체의 속성 레벨이 깊어도 반응  
  }  
}
```


Intuitive Computed

Computed

data 속성의 변화에 따라 함께 변하는 속성

```
<!-- html -->  
<p>{{ doubleNum }}</p>
```

```
// js  
data: {  
  num: 10  
}
```

```
computed: {  
  doubleNum: function() {  
    return this.num * 2;  
  }  
}
```

Computed

템플릿 코드를 **깔끔하게** 해주는 핵심 속성

```
<!-- X -->
```

```
<div>
```

```
  {{ this.alertTitle === 'Delete' ? `Delete ${this.itemForDelete.name}` :  
    this.alertTitle }}
```

```
</div>
```

```
<!-- O -->
```

```
<div>
```

```
  {{ deleteAlertTitle }}
```

```
</div>
```

Computed

템플릿 코드를 **깔끔하게** 해주는 핵심 속성

```
<!-- X -->
```

```
<div>
```

```
  {{ this.alertTitle === 'Delete' ? `Delete ${this.itemForDelete.name}` :  
    this.alertTitle }}
```

```
</div>
```

```
<!-- O -->
```

```
<div>
```

```
  {{ deleteAlertTitle }}
```

```
</div>
```

Computed

직관적인 템플릿 코드 작성 가능

```
<li v-bind:class="{ 'disabled': isLastPage }"></li>
```

```
computed: {  
  isLastPage() {  
    const lastPageCondition =  
      this.paginationInfo.current_page >= this.paginationInfo.last_page;  
    const nothingFetched = Object.keys(this.paginationInfo).length === 0;  
    return lastPageCondition || nothingFetched;  
  },  
}
```

Computed

직관적인 템플릿 코드 작성 가능

```
<li v-bind:class="{ 'disabled': isLastPage }"></li>
```

```
computed: {  
  isLastPage() {  
    const lastPageCondition =  
      this.paginationInfo.current_page >= this.paginationInfo.last_page;  
    const nothingFetched = Object.keys(this.paginationInfo).length === 0;  
    return lastPageCondition || nothingFetched;  
  },  
}
```

Computed

여러 개의 클래스를 다룰 때도 편리

```
<!-- X -->
```

```
<li  
  v-bind:class="{  
    'disabled': isLastPage,  
    'fixed': isUserLoggedIn,  
    'flexible': totalElements,  
  }"  
></li>
```

```
<!-- 0 -->
```

```
<li v-bind:class="listItemClass"></li>
```

Computed

여러 개의 클래스를 다룰 때도 편리

```
<!-- X -->
```

```
<li  
  v-bind:class="{  
    'disabled': isLastPage,  
    'fixed': isUserLoggedIn,  
    'flexible': totalElements,  
  }"  
></li>
```

```
<!-- 0 -->
```

```
<li v-bind:class="listItemClass"></li>
```


Watch vs Computed

"While **computed properties** are more appropriate in most cases, there are times when a custom watcher is necessary" - Vue.js Guide-

- ▶ **watch**를 사용하기 전에 **computed**로 해결할 수 있는지 확인
- ▶ **computed**로 해결되지 않으면 **methods**로 해결할 수 있는지 확인
- ▶ **watch**는 데이터 호출 로직과 연관된 동작에만 사용하면서 최소한으로 사용

클린 코드를 위한 속성 우선 순위 **Computed > Methods > Watch**

Render Function

먼저 간단한 템플릿 문법 부터..!

뷰 템플릿 문법 - 뷰 디렉티브

화면의 **DOM** 요소를 쉽게 **조작**할 수 있게 돕는 문법

```
<p v-bind:id="uuid">hello</p>
<span v-if="isAdmin">admin</span>
<button v-on:click="showAlert">click me</button>
```

```
data: {
  uuid: 100,
  isAdmin: true,
},
methods: {
  showAlert: function() { ... }
}
```

뷰 템플릿 문법 - 데이터 바인딩

뷰 인스턴스의 데이터를 화면에 연결해주는 문법

```
<div id="app">  
  <p>{{ message }}</p>  
</div>
```

```
new Vue({  
  el: '#app',  
  data: {  
    message: 'Naver Tech Share',  
  },  
});
```

데이터 바인딩

개발자가 직관적인 템플릿 코드를 작성할 수 있도록 돕는 문법

```
<!-- HTML -->
```

```
<div>{{ message }}</div>
```

```
// Javascript
```

```
data: {  
  message: 'Naver Tech Share',  
},
```

데이터 바인딩의 내부 동작 과정

템플릿 변환(before)

```
<!-- HTML -->  
<div id="app">  
  <p>{{ message }}</p>  
</div>
```

```
// Javascript  
new Vue({  
  el: '#app',  
  data: {  
    message: 'Naver Tech Share',  
  },  
});
```


템플릿 변환(after)

```
<!-- HTML -->
<div id="app">
  <!-- Render Function 결과가 들어갈 곳 -->
</div>

// Javascript
new Vue({
  el: '#app',
  data: { message: 'Naver Tech Share' },
  render: function(createElement) {
    return createElement('p', this.message);
  },
});
```

템플릿 변환(after)

```
<!-- HTML -->
<div id="app">
  <!-- Render Function 결과가 들어갈 곳 -->
</div>

// Javascript
new Vue({
  el: '#app',
  data: { message: 'Naver Tech Share' },
  render: function(createElement) {
    return createElement('p', this.message);
  },
});
```

템플릿 변환(after)

```
<!-- HTML -->
<div id="app">
  <!-- Render Function 결과가 들어갈 곳 -->
</div>

// Javascript
new Vue({
  el: '#app',
  data: { message: 'Naver Tech Share' },
  render: function(createElement) {
    return createElement('p', this.message);
  },
});
```

render() ?

Render Function

개발자가 직관적인 템플릿 코드를 작성하도록 뷰 내부적으로 실행하는 함수

```
render: function(createElement) {  
  return createElement('태그 이름', '태그 속성', '하위 태그 정보');  
}
```

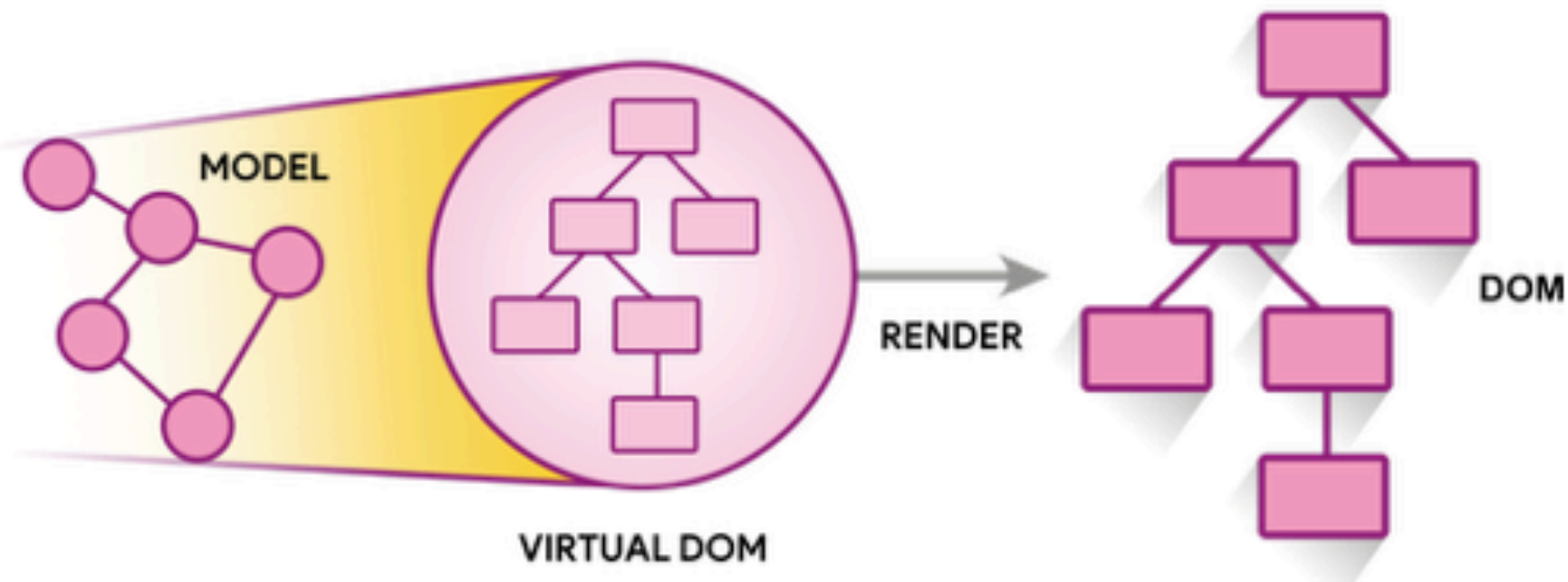
Template Compilation

템플릿 변환 순서

1. Render Function으로 변환

2. Virtual DOM 반환

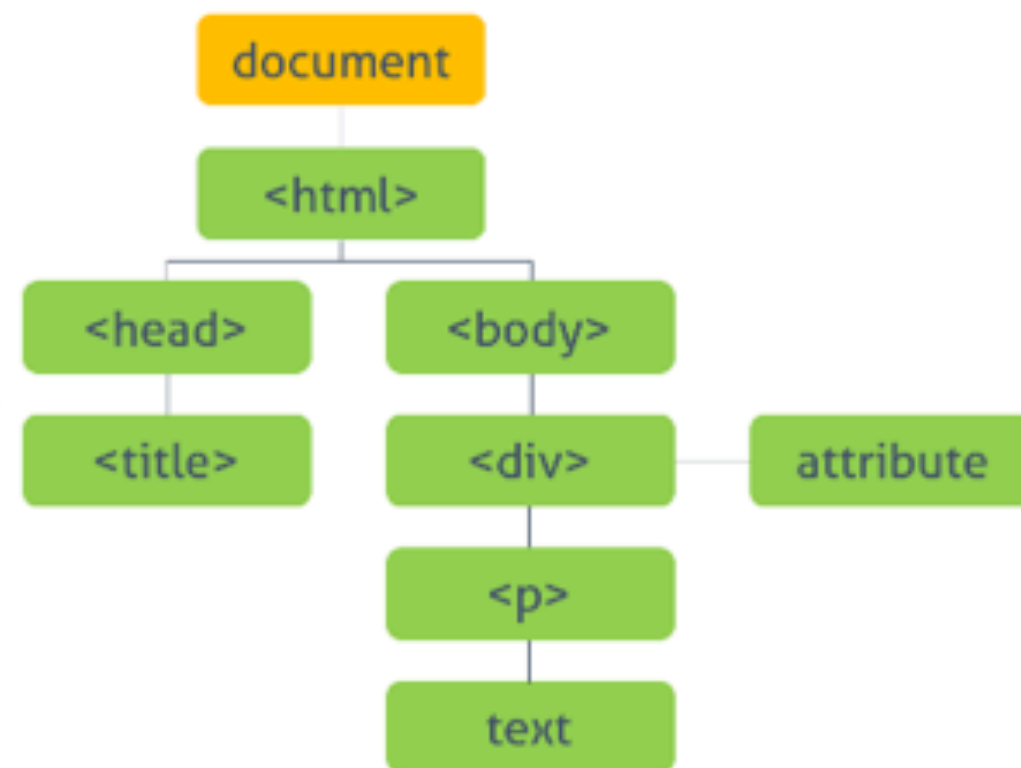
3. 실제 DOM 부착



```
<template>
  <div>
    <section>
      Do it! Vue.js 입문
    </section>
    <p>
      출간 기념 공개 세미나
    </p>
  </div>
</template>
```



```
render: function (createElement)
{
  return createElement('div', [
    createElement(
      'section',
      'Do it! Vue.js 입문',
      createElement(
        'p',
        '출간 기념 공개 세미나'
      )
    )
  ]);
}
```



Vue.js 템플릿 코드

render() 함수

DOM 트리에 요소 추가

뷰 컴포넌트 스타일링 팁

뷰 애플리케이션 스타일링 방법

1. index.html 헤더에 <link> 태그 삽입
2. 싱글 파일 컴포넌트의 scoped 속성
3. 싱글 파일 컴포넌트에서 외부 CSS 로딩
4. 싱글 파일 컴포넌트에서 여러 개의 외부 CSS 로딩

뷰 애플리케이션 스타일링 I

index.html 헤더에 <link> 태그 삽입

```
<html>  
  <head>  
    <link rel="stylesheet" href="./asset/css/app.css" />  
  </head>  
</html>
```

뷰 애플리케이션 스타일링 II

싱글 파일 컴포넌트의 scoped 속성 활용

```
<template>  
  <!-- HTML -->  
</template>
```

```
<script>  
  // Javascript  
</script>
```

```
<style scoped>  
  /* CSS */  
</style>
```

뷰 애플리케이션 스타일링 III

싱글 파일 컴포넌트에서 외부 CSS 파일 로딩

```
<template>  
  <!-- HTML -->  
</template>
```

```
<script>  
  // Javascript  
</script>
```

```
<style src="./assets/css/common.css"></style>
```

뷰 애플리케이션 스타일링 III

싱글 파일 컴포넌트에서 여러 개의 외부 CSS 파일 로딩

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

```
<script></script>
```

```
<style scoped>
```

```
  @import './assets/css/common.css';
```

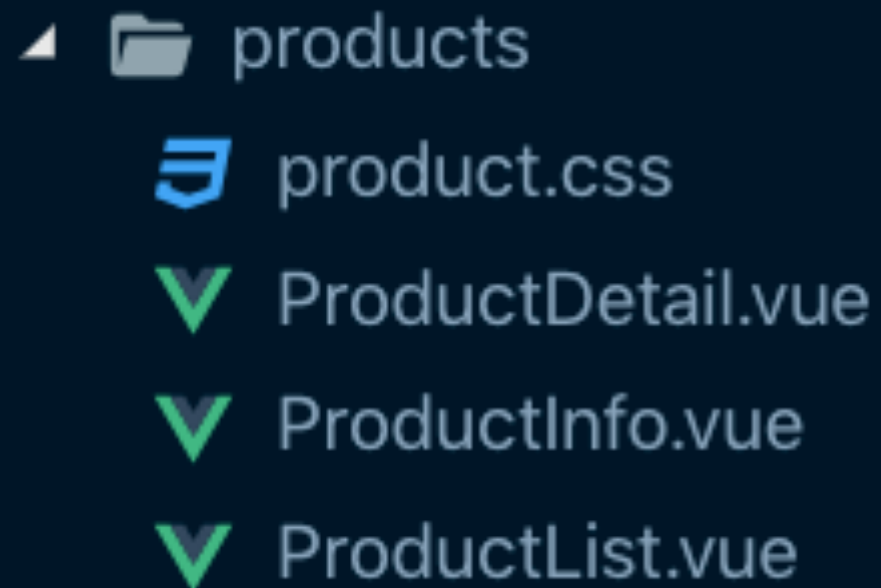
```
  @import './assets/css/reset.css';
```

```
  .container { /* ... */ }
```

```
</style>
```

싱글 페이지 애플리케이션 기준 스타일링 팁

- ▶ 디자인 공통 CSS : App.vue 파일에 @import로 적용
- ▶ 페이지 전용 CSS : 페이지 기준 최상위 컴포넌트에 페이지 전용 공통 CSS 적용. @import 및 scoped 사용



```
products
├── product.css
├── ProductDetail.vue
├── ProductInfo.vue
└── ProductList.vue
```

Form 제작 실습

함께하는 실습

아래의 조건을 만족하는 Form을 뷰로 제작해보세요.

- ▶ (필수) 폼에 입력되어야 할 입력 값은 **이메일, 비밀번호**
- ▶ (필수) 입력된 값을 제출하는 **버튼 1개**
- ▶ (선택) `computed` 속성을 이용한 **이메일 형식 유효성 검사**
- ▶ (선택) 이메일 형식 유효성 검사 **실시간 피드백** 출력 및 스타일링

끝