Vue day3 -Vue Component, Props, Emit

8. `@vue/cli`

오늘 수업에 앞서 먼저, Node.js 가 필요하다. 우리가 앞에서 학습한 자바스크립트는 프론트엔드 분야에서 주로 활용됐다. 그런데 Node.js 가 등장하면서 그 용도가 전혀 달라졌다. Node.js 라는 프레임워크 덕분에 자바스크립트로 서버단 기술까지 제어할 수 있게 된 것이다. Node.js 는 자바스크립트 엔진 'V8' 위에서 동작하는 이벤트 처리 I/O 프레임워크 이며 Node.js 로 비동기 프로그래밍을 비교적 쉽게 할 수 있게 되었다. 프론트엔드와 백엔드를 자바스크립트라는 같은 언어로 다같이 관리할 수 있는 것도 Node.js 의 큰 장점이다. Node.js 개발환경을 구축 해 보자.

- https://nodejs.org/ko/ 접속 후 좌측 `LTS` 버전을 다운로드 및 설치를 한다.

그 다음, Git Bash 창을 통해서 `@vue/cli` npm 패키지를 전역 설치한다.

```
$ npm i -g @vue/cli
```

- npm 은 전 세계 JavaScript 패키지를 모아 놓은 저장소이자 패키지 관리자 이다. Python에 pip와 같은 느낌이다.

그리고, 디렉터리를 원하는 곳에 만든다.

- 주의! 프로젝트 경로엔 절대로 점 `.` 등의 특수문자가 들어가선 안된다. 여러분들이 깃 레포지토리 한번에 관리한다고 `1. vue` 이런 식으로 디렉토리 이름 짓는데, 전체 경로에 특수문자, 이 경우엔 점 `.` 이 들어가서 에러를 일으킨다. 차라리 `1_vue` 식으로, 언더바로 이름지은 프로젝트를 만들도록 하자. 다시한번 강조하지만 파일명 또는 경로에 .이 들어가지 않도록 주의하자.

디렉터리를 VSCode 로 실행한 이후 다음 명령을 실행시키자. 그리고 기다리자.

```
$ vue create .
```

맨 끝에 점 `.` 이 들어갔다. 이는 현재 위치한 경로에 프로젝트를 만들겠다는 뜻이다.

```
Vue CLI v5.0.8
? Generate project in current directory? (Y/n) y
```

현재 디렉터리에 프로젝트를 만들 것인지 묻는다. `y` 를 입력 엔터

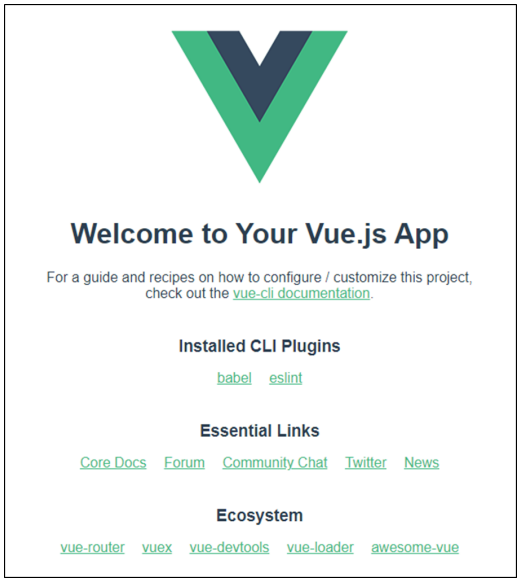
```
Vue CLI v5.0.8
? Please pick a preset:
  Default ([Vue 3] babel, eslint)
> Default ([Vue 2] babel, eslint)
  Manually select features
```

우리는 Vue.js 2 버전을 사용할 것이기 때문에, `Default ([Vue 2] babel, eslint)` 를 방향키를 이용해서 선택하고 엔터를 입력한다.

그럼 잠시 후, [#####.....] 가 모두 채워지며 프로젝트가 생성되면 run serve를 해보자. (윈도우 디펜더가 뜬다면 허용해주자)

```
$ npm run serve

DONE  Compiled successfully in 3688ms

  App running at:
  - Local:   http://localhost:8080/
  - Network: http://0.0.0.0:8080/

  Note that the development build is not optimized.
  To create a production build, run npm run build.
```

현재 `http://localhost:8080/` 에 동작중이므로, 열어보자.



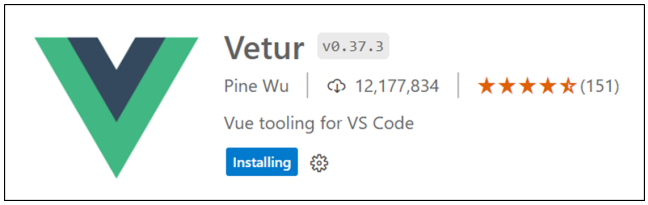
일단, Vue.js 개발을 위해 크롬 확장 프로그램 하나와, VSCode 확장 하나를 설치해주도록 하겠다.

크롬 확장 프로그램 이름은 Vue.js devtools 이다. (이미 설치 되신 분은 pass)

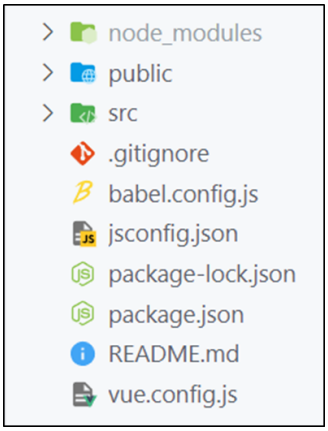
설치 이후, 다음과 같이 추가적인 설정을 해준다.

사이트 엑세스 → 모든 사이트에서
시크릿 모드에서 → 허용
파일 URL에 대한 엑세스 → 허용

다음, VSCode 확장 이름은 Vetur 이다.



이후, VSCode 에서 Vue 3 전용 추가적인 확장을 설치 할 것이냐고 묻는데, 설치 안하고 Vetur 만 사용하길 권장한다.



전체 디렉터리 구조는 다음과 같고, 하나 하나 무슨 뜻인지 살펴 보자면,

`node_modules/` : 프로젝트에 필요한 패키지가 실제 설치된 곳

`public/` : 파비콘, `index.html` 파일이 담기는 곳으로 Vue 앱의 뼈대가 되는 것이 index.html

`src/` : 가장 중요한 곳. 실제 Vue.js 코딩의 대부분 하는 곳

`.gitignore` : 깃 커밋때 제외할 리스트. 대표적으로 `node_modules/` 가 포함된다.

`package-lock.json` : `package.json` 의 상세 내역으로 사용 할 패키지 버전을 고정(python에서의 requirements.txt 역할)

`package.json` : 프로젝트에 관한 각종 정보 그리고 지원되는 브라우저에 대한 구성 옵션을 요약해 놓은 명세서

`README.md` : 리드미 마크다운 파일

그리고 디렉토리에 가보면 `.git/` 숨김 디렉터리가 존재 할 것이다. 만약에 여러 프로젝트를 모아서 하나의 깃 레포를 관리하는 사람이라면 삭제해주자.

이 중, `package.json` 설정 파일을 열어보자.

```
{
  "name": "test",
  "version": "0.1.0",
```

```
  "private": true,
  "scripts": {
    "serve": "vue-cli-service serve",
    "build": "vue-cli-service build",
    "lint": "vue-cli-service lint"
  },
  "dependencies": {
    "core-js": "^3.8.3",
    "vue": "^2.6.14"
  },
  "devDependencies": {
    "@babel/core": "^7.12.16",
    "@babel/eslint-parser": "^7.12.16",
    "@vue/cli-plugin-babel": "~5.0.0",
    "@vue/cli-plugin-eslint": "~5.0.0",
    "@vue/cli-service": "~5.0.0",
    "eslint": "^7.32.0",
    "eslint-plugin-vue": "^8.0.3",
    "vue-template-compiler": "^2.6.14"
  },
  "eslintConfig": {
    "root": true,
    "env": {
      "node": true
    },
    "extends": [
      "plugin:vue/essential",
      "eslint:recommended"
    ],
    "parserOptions": {
      "parser": "@babel/eslint-parser"
    },
    "rules": {}
  },

...
```

`name` : 패키지 이름

`version` : 패키지 버전

`private` : 패키지의 private 여부. 명시적인 것이라 `true` 라도 공개된다.

`dependencies` : 이 패키지를 작동 시키기 위해 필요한 다른 패키지 목록을 말함

`devDependencies` : 이 패키지를 개발할 때 필요한 패키지 목록

`package-lock.json` 이라는 파일이 있다. 락파일이라고 부르는데, `package.json` 의 세부 명세라고 알아두면 되고, 협업 시 반드시 같이 넘겨 줘야 한다. `package-lock.json` 에 적힌 `dependencies`, `devDependencies` 의 목록은 실제론 `node_modules/` 에 다 저장되어 있다. `node_modules/` 은 python 의 venv와 비슷한 역할을 함으로써 git에 올릴 때는 보통 .gitignore에 node_modules를 등록한다. 그리고 프로젝트를 clone 받으면 npm install 하면 되기 때문이다.

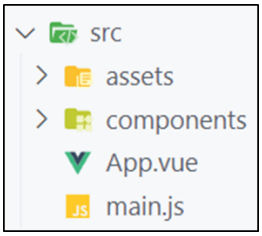
다음은 `public/` 디렉터리다. 파비콘과 `index.html` 파일이 존재한다. 열어보자.

```
<!DOCTYPE html>
<html lang="">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width,initial-scale=1.0">
    <link rel="icon" href="<%= BASE_URL %>favicon.ico">
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <noscript>
      <strong>We're sorry but <%= htmlWebpackPlugin.options.title %> doesn't work properly without JavaScript enabled. Please enable i
    </noscript>
    <div id="app"></div>
    <!-- built files will be auto injected -->
  </body>
</html>
```

우리가 작성할 수천줄의 Vue 코드는 실제로 이 파일의 `<div id="app"></div>` 한 줄 안에 들어간다. 이 말은 즉, `index.html` 이 있고, 그 안에 가상의 영역이 있어서 그 영역에서 Vue.js 가 동작한다고 생각하자.

`index.html` 파일은 타이틀을 변경하거나, 각종 CDN 을 추가할 때 방문할 것이다. (대표적으로 Bootstrap, Font …)

다음은 `src/` 디렉터리다.



`assets` : 정적 파일을 저장하는 디렉토리인데 지금은 필요 없어서 삭제할 예정이다

`components` : 하위 컴포넌트 들이 위치하는 곳인데 조금 더 이따가 사용 할 예정이라서 삭제 할 예정이다.

`App.vue` : 루트 컴포넌트(최상위 컴포넌트)를 의미하며 public/index.html과 연결된다.

`main.js` : src/App.vue 와 public/index.html를 연결 시키는 작업이 이루어지는 곳이다.

우선, `assets/` 와 `components/` 를 삭제하고, `main.js` 부터 열어보자.

```
import Vue from 'vue'
import App from './App.vue'

Vue.config.productionTip = false

new Vue({
  render: h => h(App),
}).$mount('#app')
```

여기 맨 마지막줄에 보면 `#app` 이라는 게 보일 것이다. `index.html` 의 `<div id="app">` 의 그 `app` 이 맞다. 그 한 줄에, 우리의 Vue.js 프로젝트를 밀어 넣기 위한 루트 JS 파일이다.

이 파일은 Vue 전역에서 활용 할 모듈을 등록할 수 있는 파일이다. 우리들이 각종 npm 패키지를 설치하고 세팅을 할 때 들릴 일이 많다.

다음은 `App.vue` 파일인데, 모든 것을 다 지우고, 다음과 같이 작성하자.

```
<template>
  <div>
    <h1>App</h1>
  </div>
</template>
```

이 상태에서 `http://localhost:8080/` 을 확인해보자. 참고로 이미 8080 포트가 이미 사용 중인 경우, vue cli에서 port를 바꿔서 server를 띄운다. 포트가 이미 점유 되어 있으면 해당 포트에 +1 해서 다시 사용 가능한지 확인을 반복하는 형식이다. 그래서 8080 포트가 아닌 8081 포트가 띄어 질 수 있으니 크게 신경 쓸 필요 없다.

Vue.js 를 시작하기 완벽하게 좋은 상태가 되었다.

중요: 만약, 이 템플릿을 남에게 넘겨주고 싶다면 어떻게 넘겨주는 게 좋을까? `node_modules` 디렉터리 만큼은 삭제하고 넘겨 줘야 한다. `node_modules/` 는 100메가바이트가 넘어가는 초대형 디렉터리기 때문이다.

- 그럼 패키지를 못 넘겨받지 않나요?

  아니다. 해당 리스트의 요약은 `package.json` 과 `package-lock.json` 에 기록되어있으니 `node_modules/` 는 불필요하다.

그럼 우리가 해당 파일을 넘겨 받았다고 가정해보겠다. `ctrl + c` 로 터미널을 꺼서 서버를 종료하고, `node_modules/` 를 삭제해보자. (삭제되는데 시간이 조금 걸린다.)

이 상태에서 다음 명령어를 실행해보자.

```
$ npm run serve
```

당연히 작동 안된다. 이 경우, 다음 두 명령어 중 마음에 드는 명령어 하나를 입력해주면 된다.

```
$ npm install

// 또는 install 의 약어
$ npm i
```

이후 install이 다 된 후에 `$ npm run serve` 실행 시 잘 작동된다.

9. component

컴포넌트는 Vue.js, React.js 등의 프론트엔드에서 프레임워크를 쓰는 가장 큰 이유다.

위 사진은 인스타그램 웹 페이지다. 만약에 프론트 웹 개발에 있어서 프레임워크 없이 HTML, CSS, JavaScript 만으로 개발한다고 생각해 보자. 하나의 HTML페이지에 위 사진과 같은 페이지 혹은 더 복잡한 앱의 모든 페이지 내용들을 담으려면 코드가 몇 천줄이 이상 되는 것도 충분히 가능할 것이다. 이것은, 유지 보수 관점에서 보면 최악이다. 두 가지 상황을 가정하겠다.

첫번째 상황.

인스타그램 팀에서, 오른쪽 위의 하트 아이콘을 별모양 아이콘으로 바꾼다고 가정하자.

1. `index.html` 파일을 연다.
2. 몇 천줄의 코드 중에서 하트를 찾는다.
3. 다른 아이콘으로 교체한다.

문제 없어 보이지만, 하트 하나 교체하려고 몇천줄을 뒤적이는 것이 과연 안전하고 효율적인 일 인가?

두번째 상황.

당신은 인스타그램 팀에 입사한 신입이다. 디자이너는 당신에게 `<footer>` 태그에 들어갈 새 아이콘 리스트를 주며 `<footer>` 작업을 부탁했고, 다른 선배 개발자들은 메인 피드 영역을 작업 중이다.

1. `index.html` 을 가지고 나는 `<footer>` 를 작업을 한다.
2. `index.html` 을 가지고 다른 선배 개발자 들은 피드를 작업하고, 서버에 업로드 했다.
3. 그러나, `<footer>` 를 작업하는 나는 이를 모르고 내 작업 종료 후 서버에 업로드를 했다.
4. 즉, `<footer>` 는 변경 적용되었지만, 피드 변경은 적용되지 않았다!

이런 문제를 방지하기 위해서는 내가 수정한 코드만 파트장에게 건내 준다던가, 조심스럽게 git을 사용 한다던가 등의 다양한 협업 방법이 연구 및 진행되고 있다.

그러나, 어느 똑똑한 사람이 이렇게 생각한 것이다.

하나의 페이지를 여러 개의 파일로 나눠버리면 되지 않을까?

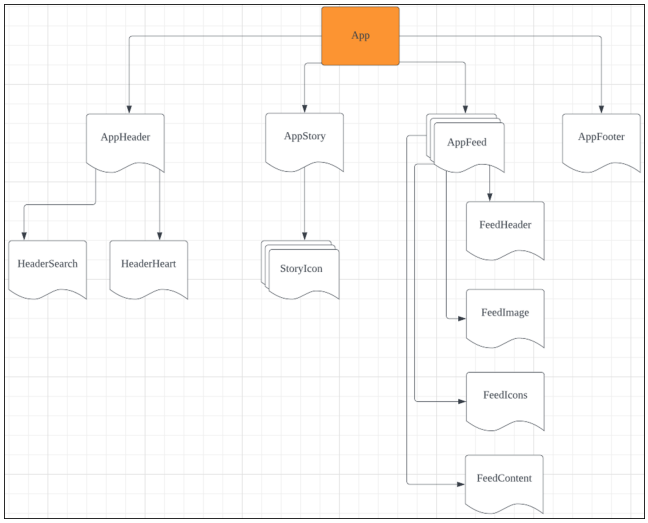
그게 바로 컴포넌트다. 레고를 생각하면 쉬운데, 각자 파트 별로 부품을 따로 만들고, 싹 다 모아서 조립하면 하나의 웹페이지가 된다.

방금 전의 화면은 다음과 같이, 총 네 개의 컴포넌트로 나뉠 수 있다.

각각의 컴포넌트는 각각의 `.vue` 파일이 되는데, 각각 `AppHeader.vue` , `AppStory.vue` , `AppFeed.vue` , `Footer.vue` 로 이름을 지어봤다.

그리고 각각의 컴포넌트는(빨강색네모) 필요에 따라서 각각의 자식 컴포넌트(연두색 네모)를 둘 수 있다.

즉, 다음과 같은 트리 형태를 띠게 된다.



각각의 트리 요소는 하나의 파일, 즉 하나의 컴포넌트다.

- 자세히 보면, `AppFeed` 나 `StoryIcon` 처럼 여러개가 겹쳐져 있는 컴포넌트가 보이는데, 이는 `v-for` 를 통한 반복을 의미한다.
- 컴포넌트 이름은 루트 컴포넌트 `App` 을 제외하고 다음의 원칙에 따라 컴포넌트 이름을 지어 보았다.
  - 두 개의 명사 결합
  - 각 명사는 대문자로 시작
  - 컴포넌트 이름으로 두 개의 명사를 생각 해 내기 어렵다면, 첫번째 명사는 부모를 나타낼 수 있도록 할 것. ex) `StoryIcon` 의 부모 컴포넌트는 `AppStory` 일 것이다.

이런 컴포넌트 방식을 따를 경우, 처음 언급했던 두 가지 문제가 해결된다.

첫번째 상황.

인스타그램 팀에서, 오른쪽 위의 하트 아이콘을 별모양 아이콘으로 바꾼다고 가정하자.

1. `FeedHeart.vue` 파일을 연다.

2. 하트를 바꾼다.
3. 다른 컴포넌트는 건드리지 않는다.

두번째 상황.

당신은 인스타그램 팀에 입사한 신입이다. 디자이너는 당신에게 `<footer>` 태그에 들어갈 새 아이콘 리스트를 주며 `<footer>` 작업을 부탁했고, 다른 선배 개발자들은 메인 피드 영역을 작업 중이다.

1. 한쪽에선 `AppFooter.vue` 를 작업한다.
2. 다른 한쪽에선 `AppFeed.vue` 를 작업한다.
3. 별개의 파일을 각각 작업하므로 안전하게 원하는 시기에 서버에 업로드가 가능하다. 각각 작업한 두 개의 컴포넌트 파일을 변경하고 업로드한다.

즉, 컴포넌트 방식으로 개발할 때, 유지보수와 협업에 있어서 엄청 유리해진다.

자 이제 직접 자식 컴포넌트를 만들어보자.

`App.vue` 는 다음과 같다.

```
<template>
  <div>
    <h1>App</h1>
  </div>
</template>
```

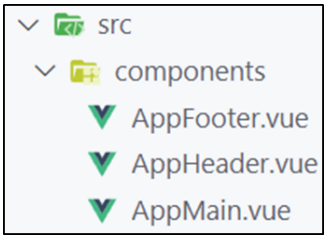
`App.vue` 를 이제부터 루트 컴포넌트라고 하겠다.

자식 컴포넌트 만들기 1단계: 파일을 만든다.

루트 컴포넌트는 총 세 개의 자식 컴포넌트를 가질 것이다.

`src/` 디렉터리에 `components/` 디렉터리를 생성하고, `AppHeader.vue` , `AppMain.vue` , `AppFooter.vue` 를 아래 양식으로 만든다.

```
<template>
  <div>
    <h2>현재 컴포넌트 이름</h2>
  </div>
</template>
```



- `<h1>` 을 쓰던 `<h2>` 를 쓰던 아무 상관 없지만, 반드시 `<template>` 아래에 `<div>` 태그 하나를 넣고 시작하자. 없으면 에러고, 두 개가 있어도 에러다.

자식 컴포넌트 만들기 2단계: 자식을 `import` 한다.

`App.vue` 루트 컴포넌트 코드 아래에, `<script>` 태그를 하나 만들어서, 다음과 같이 작성한다.

```
import AppHeader from "@/components/AppHeader.vue";
import AppMain from "@/components/AppMain.vue";
import AppFooter from "@/components/AppFooter.vue";
```

- `@` 는 `src/` 디렉터리를 뜻하는 약어다.

자식 컴포넌트 만들기 3단계: 컴포넌트를 등록한다.

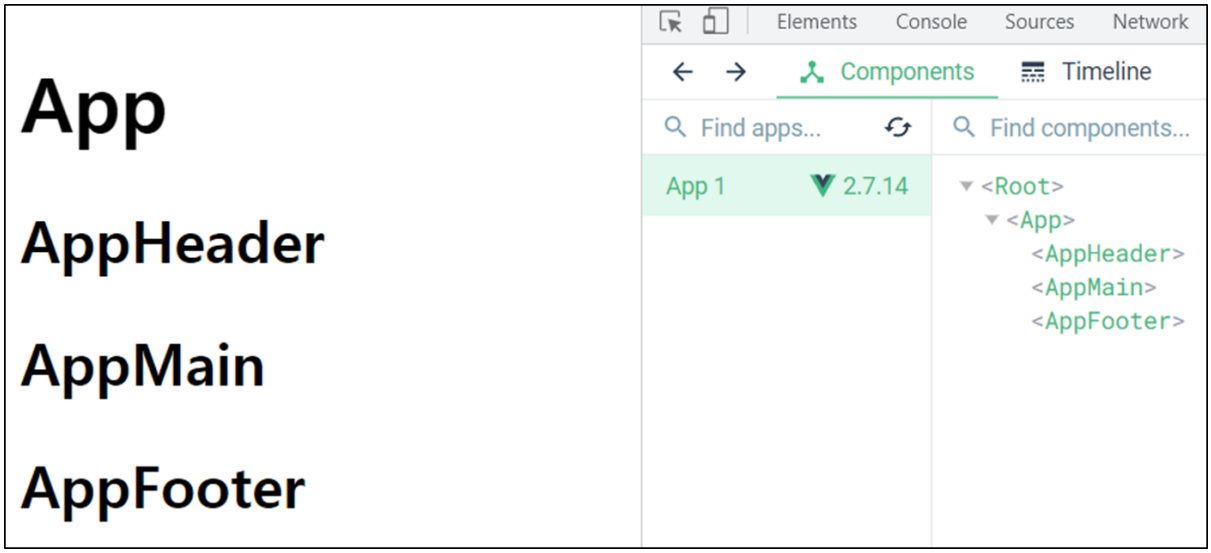
`<script>` 부분을 다음과 같이 수정한다.

```
import AppHeader from "@/components/AppHeader.vue";
import AppMain from "@/components/AppMain.vue";
import AppFooter from "@/components/AppFooter.vue";

export default {
  components: { AppHeader, AppMain, AppFooter },
};
```

자식 컴포넌트 만들기 4단계: 자식 컴포넌트를 부모에 붙인다.

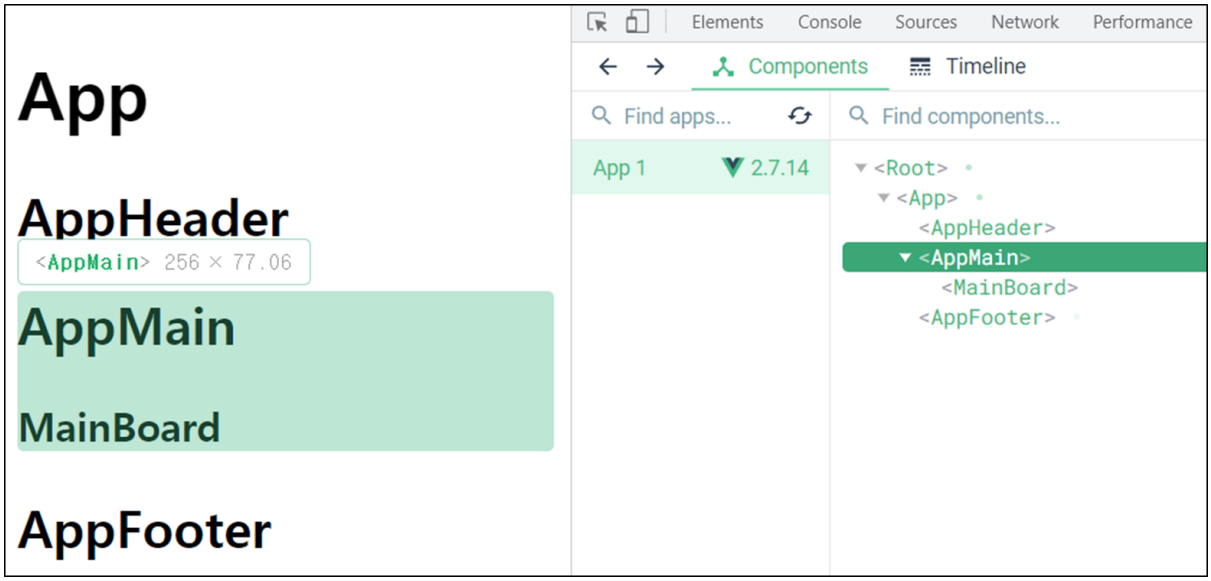
완성된 코드는 다음과 같다.

```
<template>
  <div>
    <h1>App</h1>
    <AppHeader />
    <AppMain />
    <AppFooter />
  </div>
</template>

<script>
import AppHeader from "@/components/AppHeader.vue"; //.vue 와 ; 는 생략 가능
import AppMain from "@/components/AppMain.vue";
import AppFooter from "@/components/AppFooter.vue";

export default {
  components: { AppHeader, AppMain, AppFooter },
};
</script>
```



Vue 개발자 도구를 열어서 확인해보면, 부모 자식 관계가 잘 설정된 것을 확인할 수 있다.

도전미션 !! : `AppMain` 컴포넌트에, `MainBoard` 라는 자식 컴포넌트를 만들어보자.



지금까지 위에서 거시적으로 봐 오던 것과 같이 컴포넌트 들이 tree구조를 이루어 하나의 페이지를 만든다. root에 해당하는 최상단 컴포넌트가 App.vue 이며 App.vue는 main.js를 통해서 index.html과 연결 된다. 즉, 하나의 index.html 파일을 랜더링 함으로써 하나의 웹페이지가 완성되는 SPA(Single Page Application) 가 완성된다. SPA는 단일 페이지 어플리케이션으로 하나의 페이지 안에서 변화가 필요한 영역 부분만 새로고침(동적) 하는 형태로 웹 개발이 가능해 지는 것이다.

10. scoped

하나의 컴포넌트는 기본적으로 다음과 같은 구조를 지닌다. (예시)

```
<template>
  <div>

  </div>
</template>

<script>
export default {
  data() {
    return {

    };
  },
  methods: {

  },
};
</script>

<style scoped>

</style>
```

이 기본적인 구조는 반드시 알아둬야 한다.

`<template>` : HTML

`<script>` : JavaScript

`<style>` : CSS

즉, 하나의 컴포넌트에서 별도의 파일 분리를 하지 않고, HTML, CSS, JavaScript 를 같이 사용한다.

그런데, `<style>` 태그 안에 `scoped` 애트리뷰트는 대체 무슨 뜻일까?

`scoped` 가 없다면 전역 스타일링 ⇒ 루트 컴포넌트에서 `scoped` 없이 `<style>`사용

`scoped` 가 있다면 지역 스타일링 ⇒ 일반 컴포넌트에서 `scoped` 붙여서 `<style>` 사용

차이를 알아보자.

먼저, `App.vue` 루트 컴포넌트에서 `<style>` 태그를 `scoped` 없이 작성하겠다. 아래 코드를 참고하자.

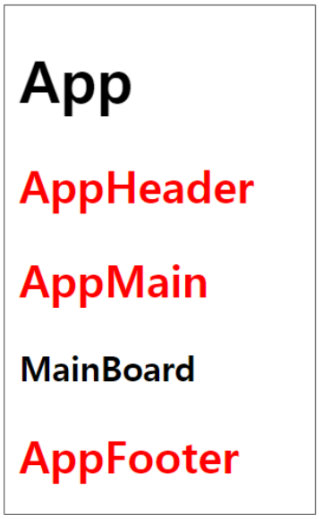
```
# App.vue 컴포넌트
```

```
<template>
  <div>
    <h1>App</h1>
    <AppHeader />
    <AppMain />
    <AppFooter />
  </div>
</template>

<script>
import AppHeader from "@/components/AppHeader.vue";
import AppMain from "@/components/AppMain.vue";
import AppFooter from "@/components/AppFooter.vue";

export default {
  components: { AppHeader, AppMain, AppFooter },
};
</script>

<style>
h2 {
  color: red;
}
</style>
```

무슨 뜻인가? 모든 `<h2>` 태그의 글자 색은 빨간색으로 하라는 뜻이다.

그런데 자세히 보면, `App.vue` 엔 `<h2>` 태그가 존재하지 않는다!

그러나, 결과는 다음과 같다.



자세히 보면, 자식 컴포넌트 중 h2 태그들의 모든 색깔은 빨간색으로 처리되었다.

왜 이런 현상이 생기는것일까? 수천 줄 수만 줄의 코드라도, Vue.js 프로젝트는 결국 하나의 `index.html` 로 이루어져있다. 페이지가 단 하나라는 뜻이다. 이는 어떤 컴포넌트 파일이든 상관없이, 모든 컴포넌트 파일의 태그에 접근해서 스타일링할 수 있다는 것을 뜻한다. 현재 `<h2>` 태그는 각각 `AppHeader` , `App Main` , `AppFooter` 컴포넌트에 있으므로, 해당 태그는 모두 빨간색이 된다.

그렇다면, `scoped` 애트리뷰트를 추가할 시 어떻게 될까? 이번엔 `AppMain` 에서 `<style>` 태그를 만들되, `scoped` 를 붙여서 다음과 같이 만들어주자.

```
# AppMain.vue 컴포넌트
```

```
<template>
  <div>
    <h2>AppMain</h2>
    <MainBoard />
  </div>
</template>

<script>
import MainBoard from "./MainBoard.vue";
export default {
  components: { MainBoard },
};
</script>

<style scoped>
h2 {
  color: blue;
}
</style>
```

`scoped` 를 붙이면, 스타일의 영역을 해당 컴포넌트로 한정짓는다. 다른 컴포넌트에 `<h2>` 가 있든 말든, 오로지 현재 컴포넌트의 `<h2>` 에만 스타일링을 적용한다.

그럼 둘은 언제 쓰는 게 좋을까? 일반적인 사용법은 다음과 같다.

`<style>` : 루트 컴포넌트에서 전역 스타일링할 때 사용

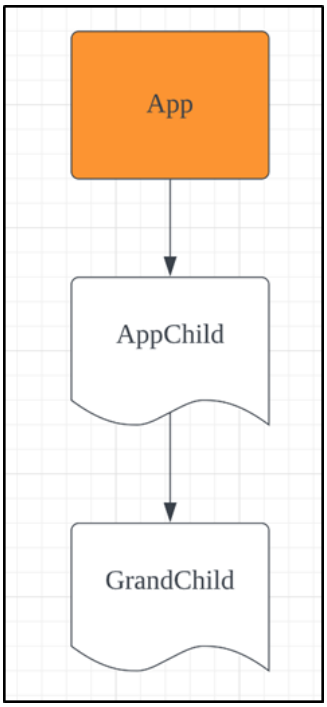
`<style scoped>` : 일반 컴포넌트에서 사용한다. <끝>

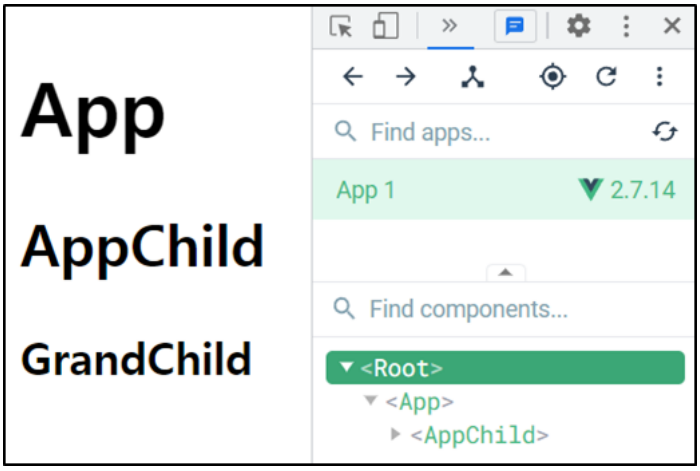
11. `props`

`components/` 디렉터리 안에 있는 모든 컴포넌트를 지우고, `App.vue` 를 다음과 같이 초기 상태로 작성하자.

```
<template>
  <div>
    <h1>App</h1>
  </div>
</template>
```

도전: 다음 구조로 컴포넌트를 만들자.





그리고 `App.vue` 를 다음과 같이 변경한다.

```
<template>
  <div>
    <h1>App</h1>
    <div>myName: {{ myName }}</div>
    <div>myAge: {{ myAge }}</div>
    <AppChild />
  </div>
</template>

<script>
import AppChild from "@/components/AppChild.vue";
```

```
export default {
  components: { AppChild },
  data() {
    return {
      myName: "ssafy",
      myAge: 40
    }
  },
  methods: {

  }
}
</script>
```

현재 데이터는 어디에 있는가? App 에 있다. 지금 하고 싶은 건, AppChild 로, `myName` 과 `myAge` 를 자식 컴포넌트로 내려보내는 것이다.

왜 이런 일을 하고싶을까? 인스타그램이나 페이스북에 접속하면, 한 화면에 프로필사진이나 이름이 중복되어서 표시되는 영역이 확인된다.

지금 보면 sujeongsshi 라는 아이디가 보이는 곳은 총 두 곳이다. 하나는 피드의 헤더 부분, 다른 하나는 피드의 컨텐트 부분이다. 이 둘은 엄연히 다른 컴포넌트이므로, 아무 생각 없이 개발한다면 별도의 `data` 안에서 관리하게 될 것이다.

문제는 아이디 sujeongsshi 의 이름이 변경 되었을 때다. 그럼 각 컴포넌트별로, 서버로 이름을 바꿔 달라고 두 번 요청을 하게 될 것이다.

그럼 아래 딜레마가 생기게 된다.

- 만약 두 개 정도가 아니라, 열 개의 서로 다른 컴포넌트에서 동일한 아이디를 사용한다면?
- 만약 아이디같은 `string` 타입이 아니라, 프로필 사진 URL 과 같은 용량이 큰 데이터라면?
- 이러한 상황이 수만명의 유저에게 동일하게 적용된다면? 서버가 힘들어 질 것이다.

이 문제를 해결하기 위해, 데이터를 부모에서 통합해서 관리 하고자 한다. 실제 데이터는 부모 컴포넌트에 있고, 자식은 부모의 데이터를 그저 보여주는 역할만 하는 것이다.

`App.vue` 의 `<template>` 을 다음과 같이 변경한다.

```
<template>
  <div>
    <h1>App</h1>
    <div>myName: {{ myName }}</div>
    <div>myAge: {{ myAge }}</div>
    <AppChild :myName="myName" :myAge="myAge" />
  </div>
</template>
```

무엇이 사용되었는지 보이는가? 느껴지는가? 바로 `v-bind` 가 사용되었다.

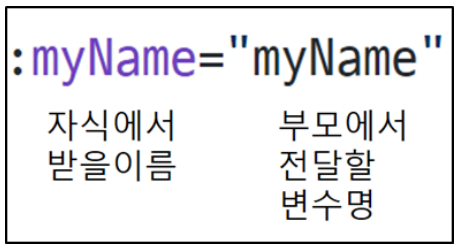
`v-bind` 는 두 가지 사용 목적이 있다.

- 태그의 애트리뷰트를 변수화 ex) `<a :href="URL">`

  그리고

- 자식에게 데이터 전달

  하는 용도가 있다.



- 변수명과 값이 다르면 혼동이 올 수 있으니 이름을 통일 하는 것을 추천한다.

자, 그럼 자식인 `AppChild` 에선 다음과 같이 받는다.

```
export default {
  components: { GrandChild },
  props: {
    myName: String,
    myAge: Number,
  },
};
```

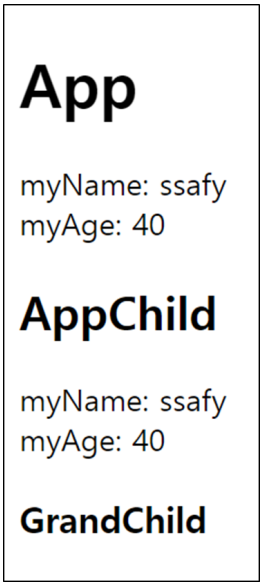
`props` 객체를 하나 선언하고, 받은 데이터의 타입을 대문자로 시작해 작성해준다.

- `AppChild` 의 `props` 객체 안에 `myName` 은, `App` 의 `myName` 이라는 이름과 동일하지만, 서로 다르게 지정 해도 괜찮다. 그러나, 이해의 편의를 위해 보통은 동일한 이름으로 지정할 뿐이다.

그리고 화면에 붙여보자. `AppChild` 컴포넌트의 `<template>` 을 다음과 같이 작성한다.

```
<template>
  <div>
    <h2>AppChild</h2>
```

```
    <div>myName: {{ myName }}</div>
    <div>myAge: {{ myAge }}</div>
    <GrandChild />
  </div>
</template>
```



부모로부터 데이터를 잘 받은 것을 확인할 수 있다.

- 이 데이터는 복사된 것인가? 아니다. 실제 데이터는 부모인 `App` 에 있을 뿐이다. `AppChild` 는 그저 부모의 데이터를 보여주기만 할 뿐이지, 데이터는 존재하지 않는다.

만약, `AppChild` 에 버튼을 달아서, `myAge` 를 변경하려고 하면 어떤 일이 발생할까?

```
<template>
  <div>
    <h2>AppChild</h2>
    <div>myName: {{ myName }}</div>
    <div>myAge: {{ myAge }}</div>
    <button @click="to_be_younger">나이를 줄이자!</button>
    <GrandChild />
  </div>
</template>

<script>
import GrandChild from "./GrandChild.vue";

export default {
  components: { GrandChild },
  props: {
    myName: String,
    myAge: Number,
  },
  methods: {
    to_be_younger(){
      this.myAge=20;
    }
  }
};
</script>
```

```
ERROR  Failed to compile with 1 error                                                    오전 10:36:04

[eslint]
C:\Users\SSAFY\Desktop\test\src\components\AppChild.vue
  22:7  error  Unexpected mutation of "myAge" prop  vue/no-mutating-props
```

예상치 못한 `"myAge"` prop 변경!

즉, 자식은 부모의 데이터를 함부로 수정할 수 없다. 하극상이다.

할 수 없다는 것을 알았으니까, 다시 원래 코드로 돌려놓자.

(직접 경험을 해봐야 기억에 남는다.)

만약, 부모로부터 받은 값을, 나의 자식에게 돌려주고 싶다면?

즉, 중계 역할을 하고 싶다면 어떻게 하면 될까?

그냥 주면 된다. 이번엔 `GrandChild` 로 `myName` 만 넘겨줄 것이다.

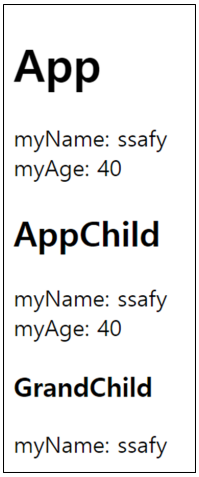
먼저, 부모인 `AppChild` 다.

```
<GrandChild :myName="myName" />
```

다음, 자식인 `GrandChild` 다.

```
<template>
  <div>
    <h3>GrandChild</h3>
    <div>myName: {{ myName }}</div>
  </div>
</template>

<script>
export default {
  props: {
    myName: String
  }
}
</script>
```

결과는 다음과 같다. (화면에 띄우고 새로고침을 해서 확인해 보자)



12. `emit`

`emit` 은 부모의 함수를 실행하는 것이다. 이 것을 이용하여 부모의 데이터를 변경할 수 있다. `GrandChild` 컴포넌트를 다음과 같이 작성하자.

```
<template>
  <div>
    <h3>GrandChild</h3>
    <div>myName: {{ myName }}</div>
```

```
    <button @click="changeMyName">부모의 함수 실행</button>
  </div>
</template>

<script>
export default {
  props: {
    myName: String
  },
  methods: {
    changeMyName() {
      this.$emit("changeMyName", "싸피");
    }
  }
}
</script>
```

코드를 보면,

클릭 시 `changeMyName` 이라는 메서드가 실행되고,

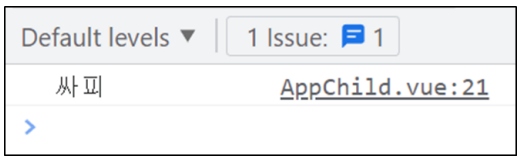
해당 메서드에선 `"changeMyName"` 이라는 이벤트를 발생시키고 있다.

그리고, 메서드에 들어갈 아규먼트로 `"싸피"` 를 보내도록 하겠다.

다음은 부모인 `AppChild` 의 코드를 다음과 같이 변경한다.

```
<GrandChild :myName="myName" @changeMyName="changeMyName" />

...

methods: {
  changeMyName(myName) {
    console.log(myName);
  }
}
```

버튼을 눌렀을 때 결과는 다음과 같다.



`AppChild` 의 21번째 라인의 `console.log` 가 작동되었다는 뜻이다.

자, 다시 정리해보자. 이 장을 시작할 때 맨 처음 무슨 문장으로 시작했는지 기억나는가?

`emit` 은 부모의 함수 실행이다.

그럼 처음부터 다시 정리를 한번 해보자.

먼저, 자식인 `GrandChild` 에선 다음과 같은 코드를 작성했다.

```
methods: {
  changeMyName() {
    this.$emit("changeMyName", "싸피");
  }
}
```

여기서, `$emit` 의 첫번째 아규먼트 `"changeMyName"` 는 부모인 `AppChild` 로 보낼 이벤트 이름이다. 현재 작동하고 있는 `changeMyName` 과는 이름이 우연히 일치할 뿐이다.

두번째 아규먼트 `"싸피"` 는 부모에서 받을 메서드의 첫번째 아규먼트가 된다. 물론 여러 개 아규먼트를 사용할수도 있다. 뒤에 콤마로 구분지어서 연달아 써주면 된다.

다음, 부모인 `AppChild` 에선 다음과 같은 코드를 작성했다.

```
<GrandChild :myName="myName" @changeMyName="changeMyName" />
```

무엇이 사용되었는지 보이는가? 느껴지는가? `v-on` 이 사용되었다.

`v-on` 은 두 가지 사용 목적이 있다.

- 이벤트 발생 시 실행시킬 메서드를 지정

  ex) click, change, keyup …
- 자식컴포넌트에서 `$emit` 을 통해 지정한 이벤트 발생 시 실행시킬 메서드를 지정

앞에 위치한 `changeMyName` 은 자식인 `GrandChild` 에서 작동시킨 `$emit` 의 첫번째 아규먼트와 일치해야한다.

뒤에 위치한 `"changeMyName"` 은 `changeMyName` 이벤트 발생 시 실행할 메서드 이름이다.

개발의 편의를 위해, 모든 이름을 `changeMyName` 으로 통일해서 사용해서 그렇지, 각각의 `changeMyName` 이 의미하는 바는 완전히 다르다. 이름은 우연히 일치 되었을 뿐이다.

결국 자식에서 `changeMyName` 이벤트가 발생하면, 아래 코드의, 부모에 있는 `changeMyName` 메서드가 실행된다.

```
methods: {
  changeMyName(myName) {
    console.log(myName);
  }
}
```

즉, `emit` 은 부모의 함수 실행이다.

그런데 대체 왜 `changeMyName` 이라는 이름을 달았을까? 자식에서 부모의 데이터를 함부로 바꿀 수 없다는 것을 우린 직접 에러를 일으켜보면서 알게 되었다.

그러면 자식에서 부모 함수에 아규먼트를 전달해서, 부모가 직접 데이터를 바꾸게 하면 될 것이다.

즉, `emit` 은 이 원리를 사용해 부모의 데이터를 변경할 때 사용된다.

그러나 `myName` 은 실제론 `App` 의 데이터이기 때문에 한번 더 올려 보내야 한다. 즉, `AppChild` 는 중계기 역할을 하게 된다.

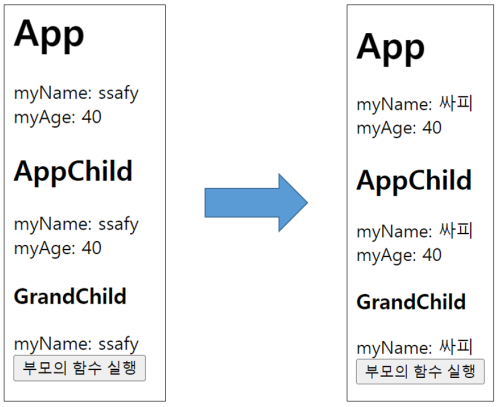
다음과 같이 `AppChild` 코드를 수정하자.

```
methods: {
  changeMyName(myName) {
    this.$emit("changeMyName", myName);
  }
}
```

자식으로부터 `myName` 을 받아서, 부모로 `myName` 을 보낼 것이다.

부모인 `App` 의 코드는 다음과 같다.

```
<AppChild :myName="myName" :myAge="myAge" @changeMyName="changeMyName" />

...

methods: {
  changeMyName(myName) {
    this.myName = myName;
  }
},
```



부모의 데이터가 잘 변경된 것을 확인할 수 있다.

- 도전mission : `AppChild` 에서 버튼을 하나 달고, `emit` 을 사용해, `myAge` 를 20 으로 변경해보자.

<끝>