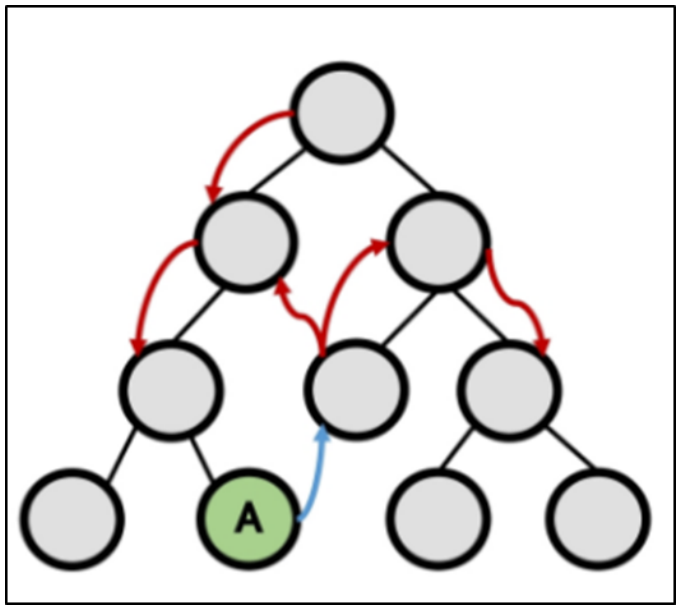
Vue day6 - Vuex

14. `Vuex`

`props` / `emit` 만으로 프로젝트를 완성할 수 있으나, 다음 경우에 문제가 된다.

- 컴포넌트 구조가 복잡해질 경우
- 컴포넌트가 부모 자식 관계가 아닐 경우

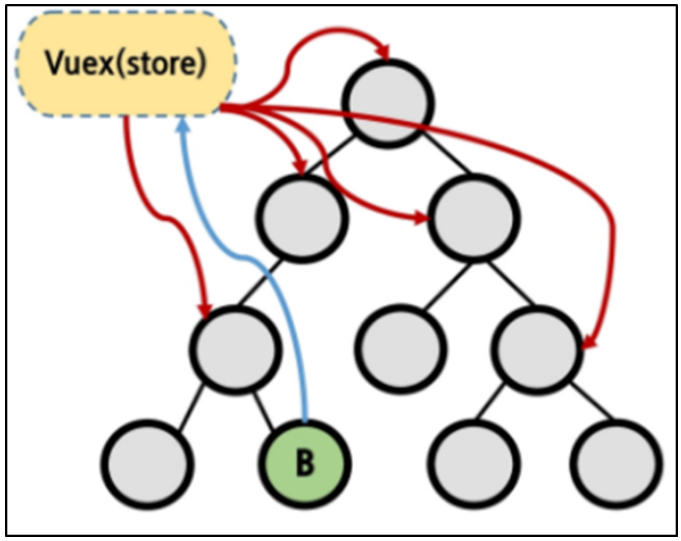
그림으로 보면 다음과 같다.



이렇게, 컴포넌트 계통의 깊이가 길어질수록 데이터를 다루기가 매우 어려워진다. 큰 프로젝트일수록 Vue.js 에서 기본 제공하는 기능 이외의 서드파티 패키지를 고려해야 한다.

Vue.js 에서 전역 데이터를 다른 말로 state (상태) 라고 하며, 여러 상태 관리 패키지 중 표준적으로 쓰이는 패키지는 현재 기준 Pinia 이지만, Vue.js 2 버전으로 개발할 경우에는 Vuex 도 추천된다.

우리가 공부할 Vuex 의 기본 아이디어는 다음과 같다.



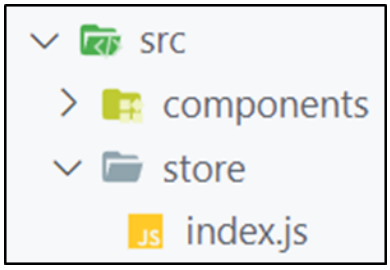
store 라는 곳에 state를 저장해서, 각 컴포넌트마다 state (전역데이터) 를 가져오거나, 변경한다. 이 경우, 위의 두 가지 문제가 해결된다.

- 컴포넌트 구조가 복잡해지더라도, store 에 접근하면 된다.
- 부모 자식 관계를 신경쓰지 않고, store 에 접근하면 된다.

Vuex 시작 전, 저번 pdf까지 했던 코드가 있다면 불러오자. `props` / `emit` 과 Vuex 모두 현업에서 자주 쓰이는 방식이다. 즉, 모든 state를 Vuex 로 작업하지는 않고 단순 애플리케이션에서는 Vuex 없이 개발을 하지만 중대형 규모의 SPA를 구축하는 경우 Vuex를 사용을 선택하게 된다.

우선 설치부터 해보자. 돌아가고 있는 서버가 있다면 서버를 잠시 종료하고, 다음 명령어를 실행하자.

```
$ vue add vuex
```



`src/` 디렉터리 안에 `store/` 디렉터리가 생겼고, 그 안에 `index.js` 가 생겼다.

열어보자.

```
import Vue from 'vue'
import Vuex from 'vuex'

Vue.use(Vuex)

export default new Vuex.Store({
  state: {
  },
  getters: {
  },
  mutations: {
  },
  actions: {
  },
  modules: {
  }
})
```

다음과 같은 형태이며, 각각의 파일을 `props` , `emit` 구문을 다 빼버리고 다음과 같이 변경한 것이다.

`App.vue` 파일이다.

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

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

export default {
  components: { AppChild },
  data() {
    return {
```

```
    };
  },
  methods: {

  },
};
</script>
```

다음, `AppChild.vue` 파일이다.

```
<template>
  <div>
    <h2>AppChild</h2>
    <div>myName: {{ }}</div>
    <div>myAge: {{ }}</div>
    <GrandChild />
  </div>
</template>

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

export default {
  components: { GrandChild },
  data() {
    return {

    };
  },
  methods: {

  },
};
</script>
```

다음, `GrandChild.vue` 파일이다.

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

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

```
    return {

    };
  },
  methods: {

  },
};
</script>
```

그리고, `store/index.js` 파일의 `state` 객체를 다음과 같이 수정한다.

```
state: {
  myName: "ssafy",
  myAge: 40
},
```

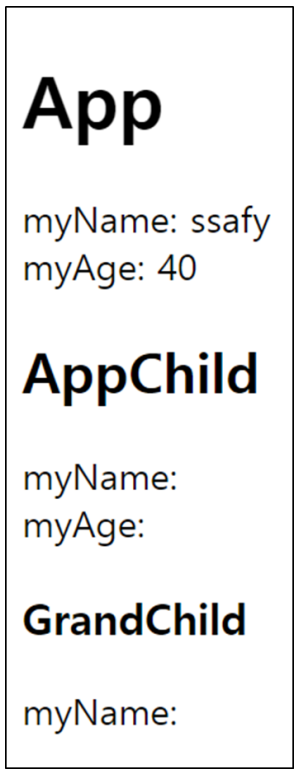
원래 어디서 관리하던 state 인가? 루트 컴포넌트인 `App` 이다.

그러나, 지금부턴 Vuex 에서 중앙 집중식으로 관리할 것이다.

`App.vue` 파일에 다음과 같이 작성해보자.

```
<div>myName: {{ this.$store.state.myName }}</div>
<div>myAge: {{ this.$store.state.myAge }}</div>
```

그리고, 서버를 작동 시키고 확인해보자.



잘 가져온 것이 확인된다.

이런 식으로, strore의 state에 전역 데이터를 정의를 하면, 각각의 컴포넌트에서 필요할 때 `this.$store.state` 에 접근해 원하는 변수를 가져올 수 있다.

그러나, 이렇게 코딩하는 방식은 추천되진 않는다. `<template>` 은 HTML 의 영역인데, 저렇게 작성할 경우 가독성에서도 좋지 않고, 화면 구성에도 지나치게 많은 연산이 들어가 성능 저하가 발생한다. 그래서 `this.$store.state`에 바로 접근하기 보다 `computed` 객체를 사용하는 것을 권장된다.

그래서 `computed` 객체를 사용하기 위해서 다음과 같이 수정한다.

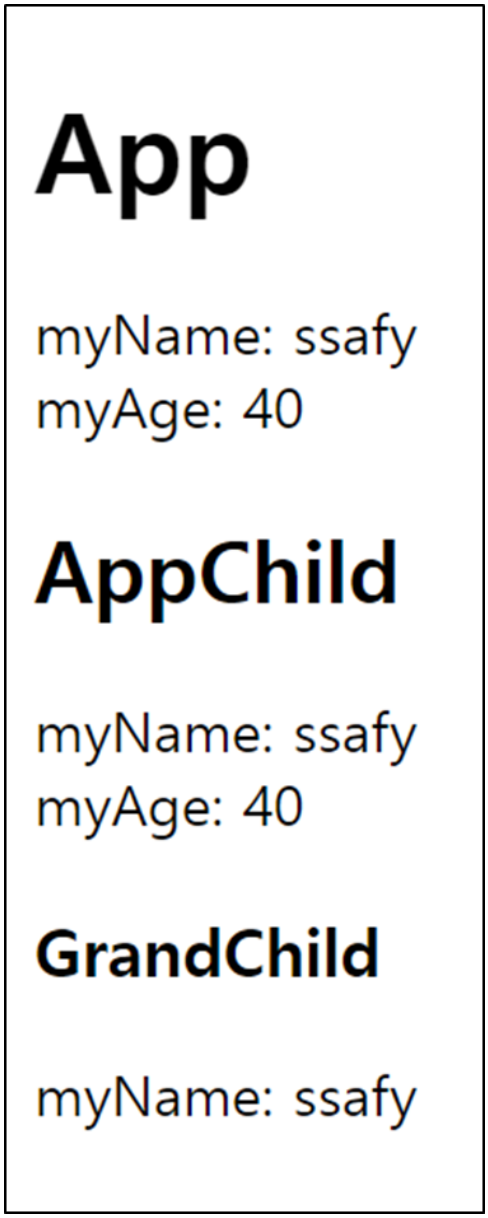
```
<template>
  <div>
    <h1>App</h1>
    <div>myName: {{ myName }}</div>
    <div>myAge: {{ myAge }}</div>
    <AppChild />
  </div>
</template>

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

export default {
```

```
  components: { AppChild },
  data() {
    return {

    };
  },
  computed: {
    myName() {
      return this.$store.state.myName;
    },
    myAge() {
      return this.$store.state.myAge;
    }
  },
  methods: {

  },
};
</script>
```

자세히보면 `<template>` 에는 `myName` , `myAge` 라고 단순하게 작성했다. 추가 설명을 조금 붙이자면 `myName` , `myAge` 이것은 변수명이 아니라 사실은 메서드다. 화면이 구성될 때 해당 이름의 함수를 실행시키는데, 메서드의 정의는 `computed` 객체 안에 있고, 실제 가져오고 싶은 값을 리턴 해준다.

- `methods` 는 트리거가 있어야만 실행된다. `computed` 는 이와 다르게, 화면의 해당 부분 변경 시 자동 실행된다. 만약 `myName` 이 변경된다면 `computed` 의 `myName` 메서드가 갱신된다. `methods` 와 메서드를 구분하기 위해 메서드 객체는 영어인 `methods` 로, 메서드는 한글인 메서드로 표현했다.

도전: 배운 것을 아이디어 삼아서, `AppChild` 와 `GrandChild` 컴포넌트에서도 `store` 에 접근해 데이터를 가져오자.



데이터를 변경할 땐 어떻게 할까? 더 이상 부모 자식 관계일 필요가 없다. 애초에 부모의 데이터가 아니라 store 의 데이터이기 때문이다.

우선, `store/index.js` 를 다음과 같이 정의 해 보자.

```
import Vue from "vue";
import Vuex from "vuex";

Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    myName: "ssafy",
    myAge: 40,
```

```
  },
  getters: {},
  mutations: {
    CHANGE_MY_NAME(state, myName) {
      state.myName = myName;
    },
    CHANGE_MY_AGE(state, myAge) {
      state.myAge = myAge;
    },
  },
  actions: {
    changeMyName(context, myName) {
      context.commit("CHANGE_MY_NAME", myName);
    },
    changeMyAge(context, myAge) {
      context.commit("CHANGE_MY_AGE", myAge);
    },
  },
});
```

`mutations` 와 `actions` 의 문법을 익혀 보도록 하자.

`mutations`

- `mutations`은 state를 변경 할 때 사용하는 메서드이다.
- 모든 글자를 대문자로, 구분되는 곳은 언더바 처리
- 첫번째 파라미터는 항상 `state` , 두 번째 이상 파라미터는 전달받은 아규먼트가 된다.
- 항상 `state.스테이트명 = 갱신할스테이트` 포맷으로 작성한다.

`actions`

- `actions`은 state를 직접 변경하는 것이 아니라 `mutations`를 호출해서 `state` 를 변경한다. `mutations` 와 비슷하지만 비동기 작업을 위해서 `actions` 을 사용한다.
- 첫번째 파라미터는 항상 `context` , 두 번째 이후의 파라미터는 전달받은 아규먼트가 된다.
- state를 변경 시 반드시 `mutations` 호출 구문인 `context.commit` 가 들어가며, 첫번째 아규먼트로 실행시킬 `mutations` 이름, 두번째 이상 아규먼트는 전달될 파라미터가 된다.

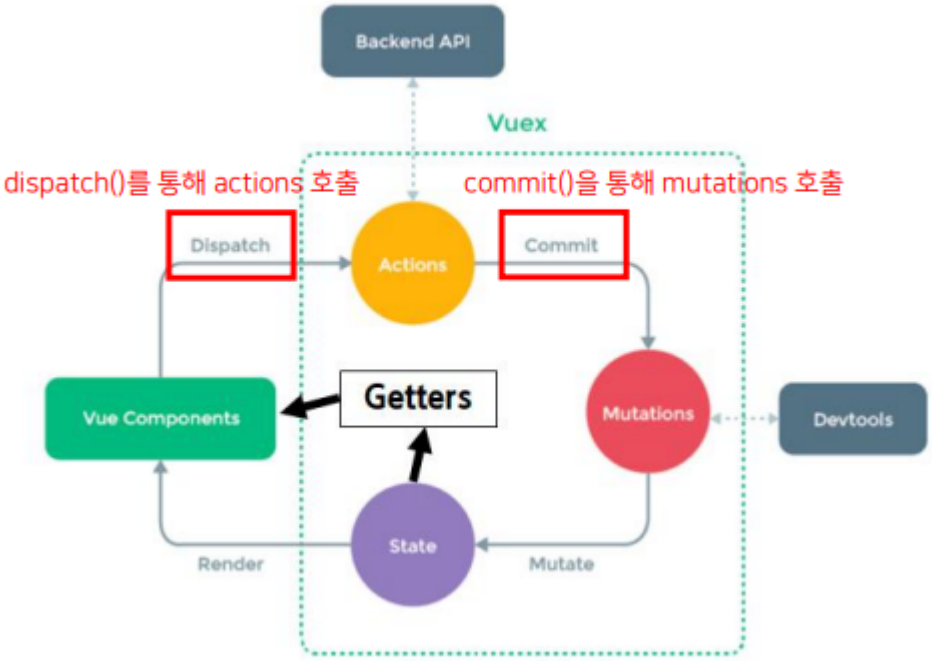
직접 사용해보자. `GrandChild` 에 버튼을 달고, 다음과 같이 작성하자.

```
<template>
  <div>
```

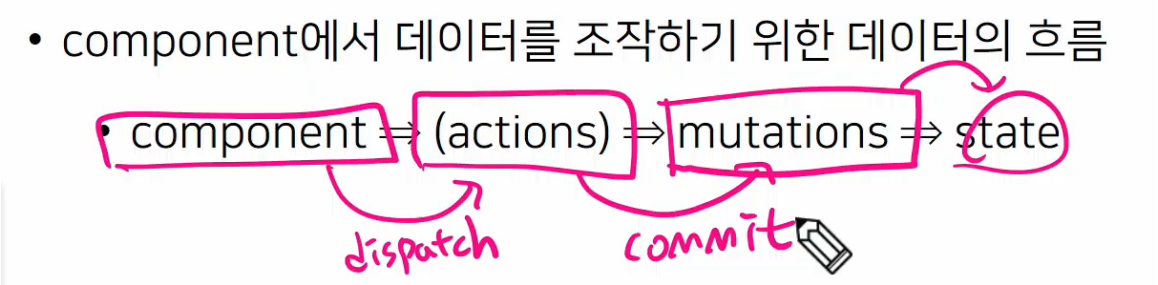
```
    <h3>GrandChild</h3>
    <div>myName: {{ myName }}</div>
    <button @click="becomingYoung">젊어지는 버튼</button>
  </div>
</template>

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

    };
  },
  computed: {
    myName() {
      return this.$store.state.myName;
    },
  },
  methods: {
    becomingYoung() {
      this.$store.dispatch("changeMyAge", 20);
    }
  },
};
</script>
```

`actions` 를 호출할 때에는 `this.$store.dispatch` 로 `actions`에 접근하며, 첫번째 아규먼트로 실행시킬 `actions` , 두번째 이상 아규먼트는 전달될 파라미터가 된다.

버튼을 클릭하면 `myAge` 가 40 에서 20 으로 바뀌는 것을 확인할 수 있다.



- 원칙이 있는데, Vue 코드에서 state를 직접 호출하는 것은 오로지 `actions` 가 되어야 하고, 절대로 `mutations` 을 직접 호출하지 않는다.



- 다시 한번 강조하지만 state를 변경하는 것은 각각의 컴포넌트에서 `this.$store.dispatch` 을 통해서 `actions`을 호출하면 `actions` 에서 `context.commit` 을 통해서 `mutations` 을 호출한 결과로 state값이 변경된다. 절대로 `mutations` 을 직접 호출하지 않는다.

`getters` 객체는 어떤 경우 사용할까? 만약 `myName` 을 그냥 가져오는 게 아니라, `"제 이름은 ssafy입니다"` 식으로, state를 활용해서 어떠한 결과값으로 계산 처리를 한 다음에 넘겨주고 싶을 때 사용한다. 이때 state의 원본 데이터를 건들이지 않고 계산된 값을 얻을 수 있는 것이다. (Vue 인스턴스의 computed 와 매우 비슷함)

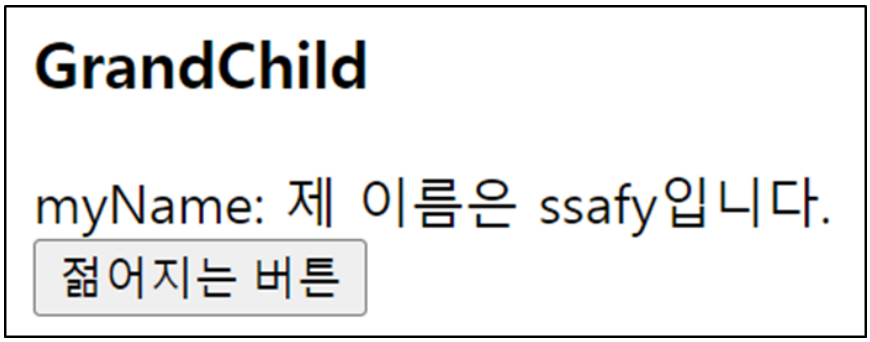
`GrandChild` 컴포넌트를 다음과 같이 수정한다.

```
computed: {
  myName() {
    return this.$store.getters.getMyName;
  },
},
```

`index.js` 의 `getters` 객체에 다음과 같이 추가한다.

```
getters: {
  getMyName(state) {
    return `제 이름은 ${state.myName}입니다.`
  }
},
```

결과는 다음과 같다.



`actions` 객체에선 `state` 변경과 관련된 axios 통신 구문이 들어간다. 그러나, 서버 통신 없이 단순히 데이터를 변경 할 것이라도, `actions` 를 거쳐서 `mutations` 를 호출해야 한다.

절대로 `mutations` 을 직접 호출하지 않는다. `actions` 를 거쳐서 `mutations` 을 호출 한다.

서버 통신 구문을 `actions` 에 추가하기 위해 axios 를 설치하겠다.

```
$ npm i axios
```

`package.json` 을 확인해서, axios 버전을 확인하고, 서버를 다시 구동한다.

`store/index.js` 를 다음과 같이 수정하자. 복붙 하지 말고, 생략된 구문을 자세히 보면서 작성해 보자.

```
import Vue from "vue";
import Vuex from "vuex";
import axios from "axios";

Vue.use(Vuex);

export default new Vuex.Store({
  state: {
    // ...
    datas: [],
  },
  getters: {
    // ...
  },
  mutations: {
    // ...
    CHANGE_DATAS(state, datas) {
      state.datas = datas;
    }
  },
  actions: {
    // ...
    getDataFromServer(context) {
      axios
        .get("https://jsonplaceholder.typicode.com/todos")
        .then((response) => {
          context.commit("CHANGE_DATAS", response.data);
        })
        .catch((error) => {
          console.log(error);
        });
    },
  },
});
```

우선, `store/index.js` 에 `axios` 패키지를 import했다.

`state` 객체에 `datas` 를 빈 배열로 정의

`mutations` 객체에 `CHANGE_DATAS` 메서드를 정의

`actions` 객체에 `getDataFromServer` 메서드를 정의 했다.

그리고 사용해보자. `App.vue` 에 버튼을 하나 달겠다.

```
<button @click="getData">데이터를 가져오자!</button>
<ul>
  <li v-for="data in datas" :key="data.id">{{ data.title }}</li>
</ul>

// ...

computed: {
  // ...
  datas() {
    return this.$store.state.datas;
  }
},
methods: {
  getData() {
    this.$store.dispatch("getDataFromServer");
  }
},
```

이 코드에서 `datas` 는 Vuex의 `state` 객체가 아니라 `computed` 에 정의된 `datas` 메서드를 의미 한다.

버튼을 누르면, `actions` 의 `getDataFromServer` 가 작동되며, `datas` 는 `computed` 에 의해 자동 갱신 되고, 화면에 전체 리스트가 출력 하게 된다.

- 그럼 모든 axios 통신은 `actions` 에 정의해야 하나요? 아니다. 오로지 Vuex State 와 관련되어 있을 경우에만 `actions` 에 정의합니다.
- Vuex 를 배웠으니 `props` / `emit` 은 쓸모가 없는가? 아니다. 이 둘은 상황에 따라 적절히 쓰는 게 좋다.
  - `props` / `emit` : 부모 자식 간에 간단한 데이터 전달을 구현할 때
  - Vuex : 여러 컴포넌트에서 특정 state 를 참조해야 할 때 사용한다.

<끝>