



## ■ 이전 장까지의 연락처 앱 예제 리뷰

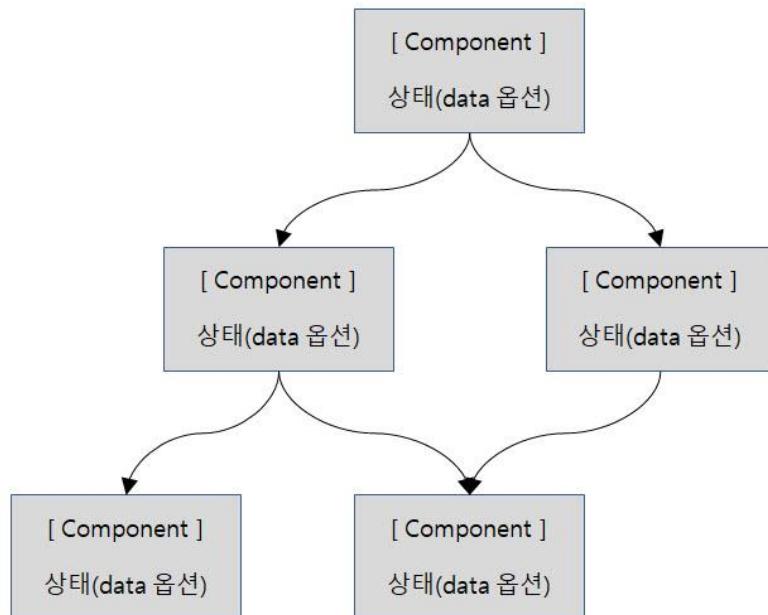
- 컴포넌트간의 정보 전달 : props, event
- 부모 컴포넌트로부터 전달받은 정보는 자식컴포넌트에서 변경할 수 없음
  - 자식컴포넌트에서 부모컴포넌트 이벤트를 발신하여 data를 보유한 컴포넌트에서 수정되도록 해야 함.
  - 컴포넌트간의 구조가 복잡해지면 어쩔 수 없이 Event Bus 객체를 사용해야 함
  - 하지만 Event Bus 객체 방식도 대규모 애플리케이션을 개발할 때는 복잡도를 증가시킬 수밖에 없음
  - 해결책 1 : 전역 객체를 생성하고 여러개의 Vue 인스턴스나 컴포넌트들이 공유하는 방법
    - 하지만 공유 객체가 여러 컴포넌트에 의해 변경될 수 있지만 추적이 힘들다
  - 해결책 2 : Vuex와 같은 상태관리 기능 이용

# Vuex를 사용하기 전 상태



## 각각의 컴포넌트의 데이터 옵션을 통해 상태를 관리?

- 각 컴포넌트들이 관리하고 있는 상태를 props로 전달하기 어려움
- 상태를 변경하기 위해 event를 emit하는 것은 더더욱 복잡해짐

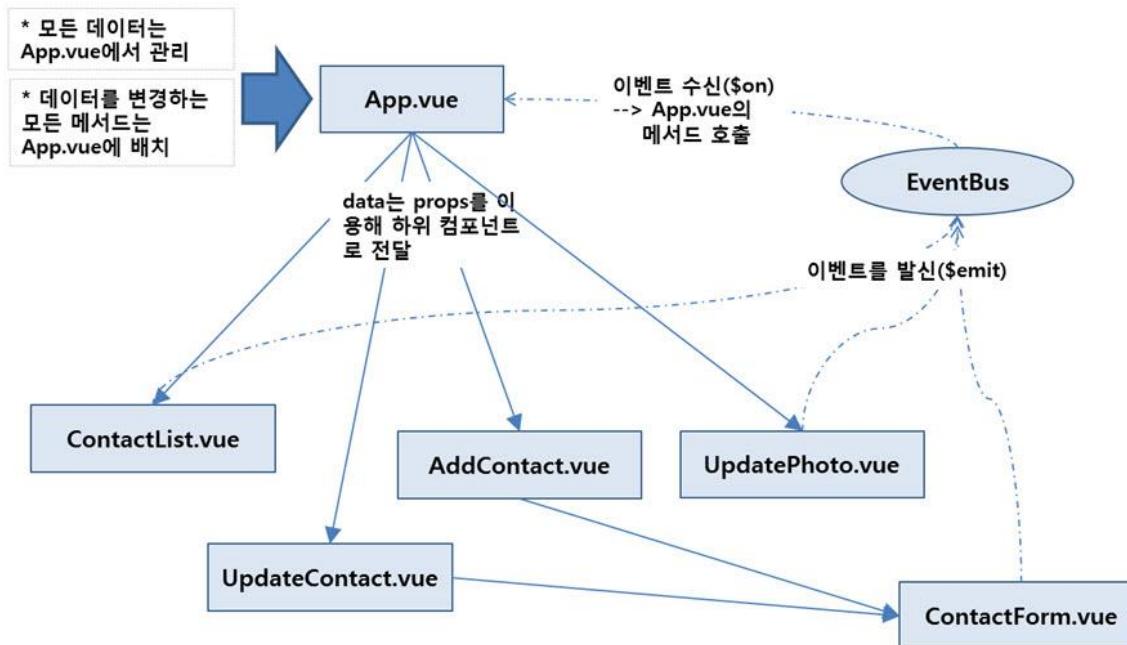


# Vuex를 사용하기 전 상태



## ■ 최상위 부모 컴포넌트에서 상태 관리

- EventBus 객체를 이용해 정보 전달
- 부모 컴포넌트의 상태를 props를 이용해 자식 컴포넌트로 전달
- 부모 -> 자식 -> 자식 -> ...

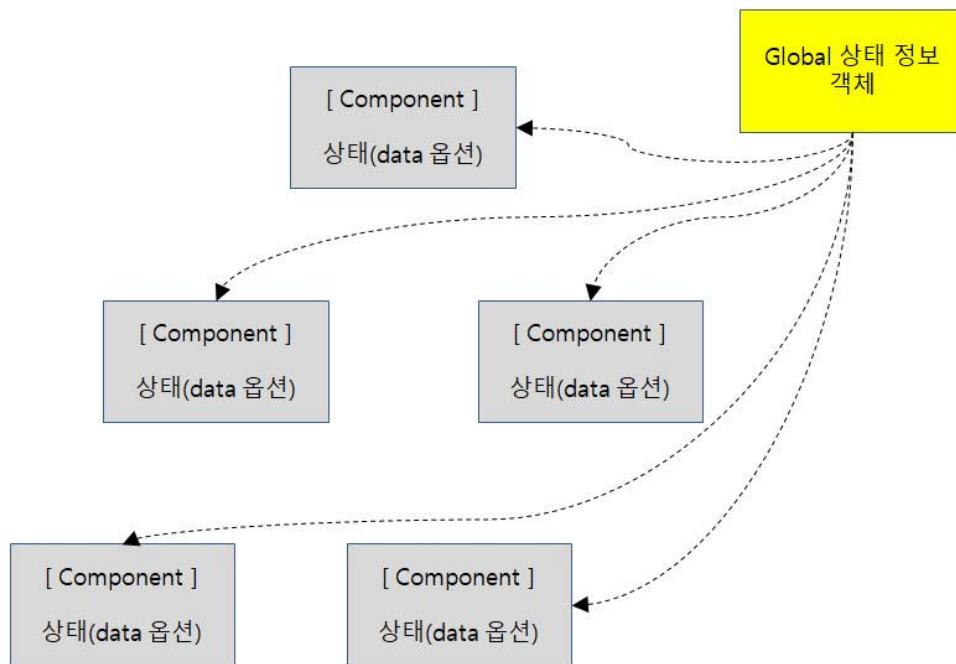


# Vuex를 사용하기 전 상태



## ■ 전역 객체를 공유해볼까?

- 객체는 참조형이니까 모든 컴포넌트가 동일한 상태를 참조할 수 있음
  - 하지만 어느 컴포넌트가 언제 어떻게 상태를 변경했는지 추적할 수 없음



```
export default {
  name : "이동률",
  scores : [
    { course : "국어", score: 100 },
    { course : "수학", score: 90 },
    ...
  ]
}
```

```
.....  
<script type="text/javascript">  
import globalState from './globalState.js'
```

```
export default {
  name : "ui-component",
  data : globalState,
  ...
}
```

# Vuex란?



## ■ Vuex?

- Vue.js 애플리케이션에서 상태관리 패턴을 지원하는 라이브러리
  - 애플리케이션 내부의 모든 컴포넌트들이 공유하는 중앙집중화된 상태 정보 저장소 역할을 수행하며 상태 변경을 투명하고 추적가능하게 함.
  - 부모에서 자식으로 props를 이용하여 속성을 계속해서 전달하지 않아도 됨.
  - 상태 데이터를 변경하기 위해 부모 컴포넌트로 이벤트를 발신하지 않아도 됨.
- Vuex와 같은 상태 관리 라이브러리가 필요한 이유
  - 중앙 집중화된 상태 정보 관리가 필요하다.
  - 상태 정보가 변경되는 상황과 시간을 추적하고 싶다.
  - 컴포넌트에서 상태 정보를 안전하게 접근하고 싶다
- 모든 애플리케이션 개발시에 Vuex를 사용해야만 하는 것은 아님
  - 간단한 구조의 애플리케이션이라면 EventBus 객체의 사용 정도로도 충분



# Vuex란?

## ■ Vuex 이해

### ■ 저장소(Store)

- 애플리케이션의 상태를 중앙집중화하여 관리하는 컨테이너
- 저장소의 상태를 직접 변경하지 않음. 반드시 변이를 통해서만 변경할 수 있도록 함.

### ■ 상태(State) : 저장소에 의해 관리되는 애플리케이션의 데이터

### ■ 변이(Mutation)

- 기존 상태를 유지하고 새로운 상태 값을 생성함.
- 상태의 변화를 추적할 수 있으며 시간 추적 디버깅이 가능하도록 함.
- 변이의 목적은 상태의 변경. 상태 변경과 관련이 없는 작업은 변이로 처리하지 않음.
- 동기적인 작업, 비동기 처리는 변이를 통해 처리하지 않음.

### ■ 액션(Action)

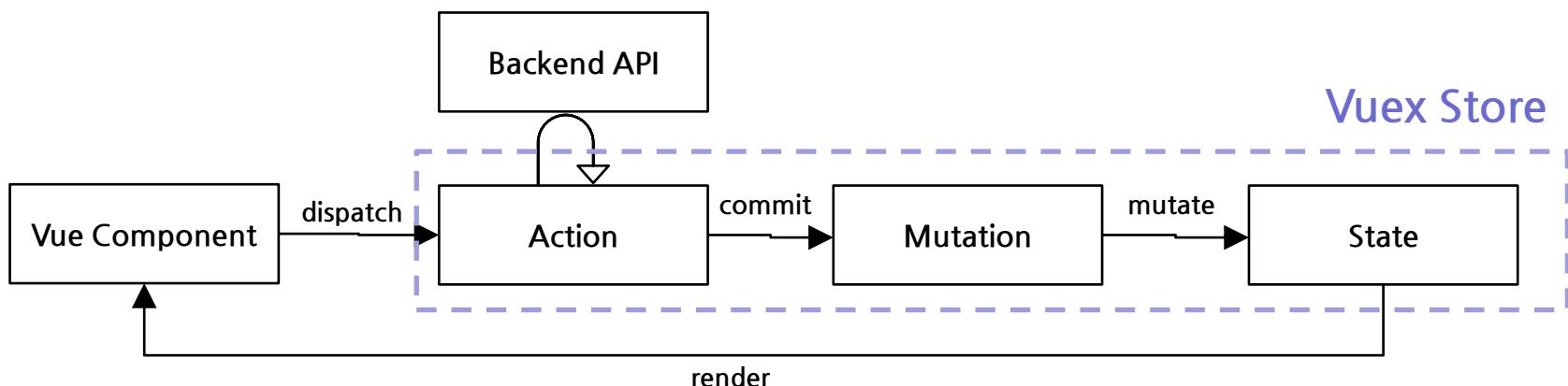
- 상태 변화와 관련이 없는 작업
- 액션을 통해 비즈니스 로직(Backend API)이 실행되게 하고 그 결과를 변이로 커밋함.
- 비동기 처리가 가능함.

# Vuex란?



## ■ Vuex 처리 흐름 : "단방향 데이터 바인딩"

- 컴포넌트가 액션(Actions)을 일으키면(예를 들면 버튼 클릭같은...)
- 액션에서는 외부 API를 호출한 뒤 그 결과를 이용해 변이(Mutations)를 일으키고(만일 외부 API가 없으면 생략)
- 변이에서는 액션의 결과를 받아 상태를 변경(Mutate)한다. 이 과정은 추적이 가능하므로 Vue DevTools와 같은 도구를 이용하면 상태 변경의 내역을 모두 확인할 수 있다.
- 변이에 의해 변경된 상태는 다시 컴포넌트에 바인딩되어 UI를 갱신(render)한다.





## ■ 저장소(Store)의 핵심 요소

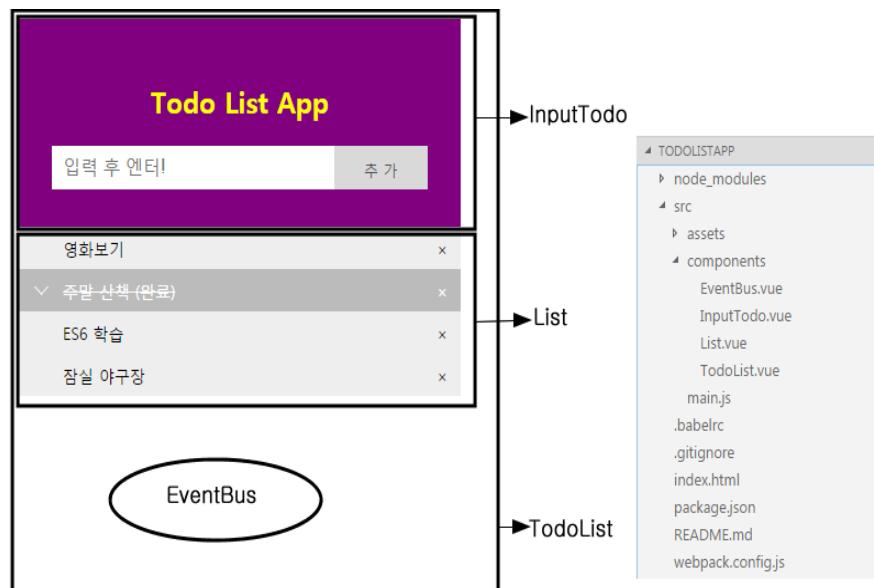
- 상태(state)는 애플리케이션의 전역 데이터
- 변이(Mutations)는 상태를 변경하는 함수들을 보유한 객체
  - 상태는 변이를 통해서 변경해야만 상태의 변경 내역을 추적할 수 있음
- 전역에서 Vue.use(Vuex) 코드의 실행으로 애플리케이션 내부의 모든 컴포넌트가 저장소의 상태, 변이 객체에 접근할 수 있음
- 상태와 변이를 적용해볼 예제
  - 9장에서 작성한 todolist 앱

# 상태와 변이



## ■ 9장의 todolist 앱 구조 분석

- 3개의 컴포넌트와 Event Bus 객체로 구성
- 컴포넌트 각각이 상태 정보를 가짐
  - 이벤트의 발신과 수신이 어렵게 배치되어 복잡함.





## ■ Vuex 다운로드

- npm install --save vuex

## ■ 상수 작성 : src/constant.js

- 변이를 일으킬 때 문자열 정보를 전달하므로 오타가 나지 않도록...

```
export default {  
    ADD_TODO : "addTodo",  
    DONE_TOGGLE : "doneToggle",  
    DELETE_TODO : "deleteTodo"  
}
```

# 상태와 변이



## 저장소 작성 : src/store/index.js

```
import Vue from 'vue';
import Vuex from 'vuex';
import Constant from '../constant';
Vue.use(Vuex);

const store = new Vuex.Store({
  state: {
    todolist: [
      { todo: "영화보기", done:false }, { todo: "주말 산책", done:true },
      { todo: "ES6 학습", done:false }, { todo: "잠실 야구장", done:false }
    ],
  },
  mutations: {
    [Constant.ADD_TODO] : (state, payload) => {
      if (payload.todo !== "") {
        state.todolist.push({ todo: payload.todo, done:false });
      }
    },
    [Constant.DONE_TOGGLE] : (state, payload) => {
      state.todolist[payload.index].done = !state.todolist[payload.index].done;
    },
    [Constant.DELETE_TODO] : (state, payload) => {
      state.todolist.splice(payload.index,1);
    }
  }
})
export default store;
```

# 상태와 변이



- 모든 애플리케이션의 상태를 Vuex로 관리할 필요는 없음
  - 하나의 컴포넌트에서만 사용되는 상태는 data 옵션을 그대로 사용할 수 있음
  - 여러 컴포넌트가 동일한 상태를 이용하는 경우에만....
- 변이 객체의 메서드
  - 첫번째 인자가 state!!
  - 두번째 인자가 payload : 변이로 전달된 데이터. 전달할 정보가 여러개라면 객체구조로 전달하면 됨.

## ■ src/main.js 변경

```
import Vue from 'vue'  
import store from './store'  
import TodoList from './components/TodoList.vue'  
  
new Vue({  
  store,  
  el: '#app',  
  render: h => h(TodoList)  
})
```

# 상태와 변이



## ■ 이벤트 버스 객체 제거 & List 컴포넌트 수정

- src/components/List.vue

```
.....(생략)
<script type="text/javascript">
import Constant from '../constant'

export default {
  computed : {
    todolist() {
      return this.$store.state.todolist;
    }
  },
  methods : {
    checked : function(done) {
      if (done) return { checked:true };
      else return { checked:false };
    },
    deleteTodo : function(index) {
      this.$store.commit(Constant.DELETE_TODO, {index: index});
    },
    doneToggle : function(index) {
      this.$store.commit(Constant.DONE_TOGGLE, {index:index });
    }
  }
}
</script>
```

# 상태와 변이



## ■ src/components/InputTodo.vue

```
<style>
.....(변경사항 없음)
</style>
<template>
  <div>
    <input class="input" type="text" id="task" v-model.trim="todo"
      placeholder="입력 후 엔터!" v-on:keyup.enter="addTodo">
    <span class="addbutton" v-on:click="addTodo">추가</span>
  </div>
</template>
<script type="text/javascript">
import Constant from '../constant'

export default {
  name : 'input-todo',
  data : function() {
    return { todo : "" }
  },
  methods : {
    addTodo : function() {
      this.$store.commit(Constant.ADD_TODO, { todo: this.todo });
      this.todo = "";
    }
  }
}
</script>
```

# 상태와 변이



## ■ 예제 11-01~05까지 실행 결과

- Vue Devtools - Vuex 탭
- 시간 추적 디버깅

The screenshot shows the Vue Devtools interface with the Vuex tab selected. On the left, there's a preview of a Todo List App with a purple header and a list of todos. The main area displays mutations in the Vuex store:

- Base State**:
  - addTodo (01:12:11)
  - addTodo (01:12:21)
  - doneToggle (01:12:24)
  - deleteTodo (01:12:28)
- state**:
  - todolist: Array[5]
    - 0: Object
    - 1: Object
    - 2: Object
    - 3: Object
    - 4: Object
- getters**: None
- mutation**:
  - type: "deleteTodo"
    - payload: Object
      - index: 0

On the right, a timeline shows the sequence of mutations:

- Base State (23:59:20)
- addTodo (00:25:02)
- addTodo (00:25:09) (highlighted with a red arrow)
- doneToggle (00:25:12)
- deleteTodo (00:25:16)

# 상태와 변이



## ■ Helper Method 적용

- mapState, mapMutations
- 예제 11-06 : src/components/List.vue
- lodash의 `_.extend()` 사용

```
<script type="text/javascript">
import Constant from '../constant'
import { mapState, mapMutations } from 'vuex'
import _ from 'lodash';

export default {
  computed : mapState([ 'todolist']),
  methods : _.extend({
    checked (done) {
      if(done) return { checked:true };
      else return { checked:false };
    }
  },
  mapMutations([
    Constant.DELETE_TODO ,
    Constant.DONE_TOGGLE
  ])
}
</script>
```

- ES6의 Rest Operator를 사용할 수 있음
- babel-preset-env를 사용하고 있다면 적용 가능

```
export default {
  computed : mapState([ 'todolist']),
  methods : {
    ...mapMutations([
      Constant.DELETE_TODO ,
      Constant.DONE_TOGGLE
    ]),
    checked : function(done) {
      if(done) return { checked:true };
      else return { checked:false };
    }
  }
}
```



## ■ 왜 계산형 속성에 상태를 바인딩할까?

- 컴포넌트 수준에서 상태를 직접 변경하지 않았으면....
- 현재는 수정이 가능하지만 권장하지 않음
  - 직접 변경하는 것을 막을 때 쓰는 옵션이 364페이지의 strict:true 옵션
  - 컴포넌트에서 변이를 거치지 않고 수정하면 오류를 발생시킴

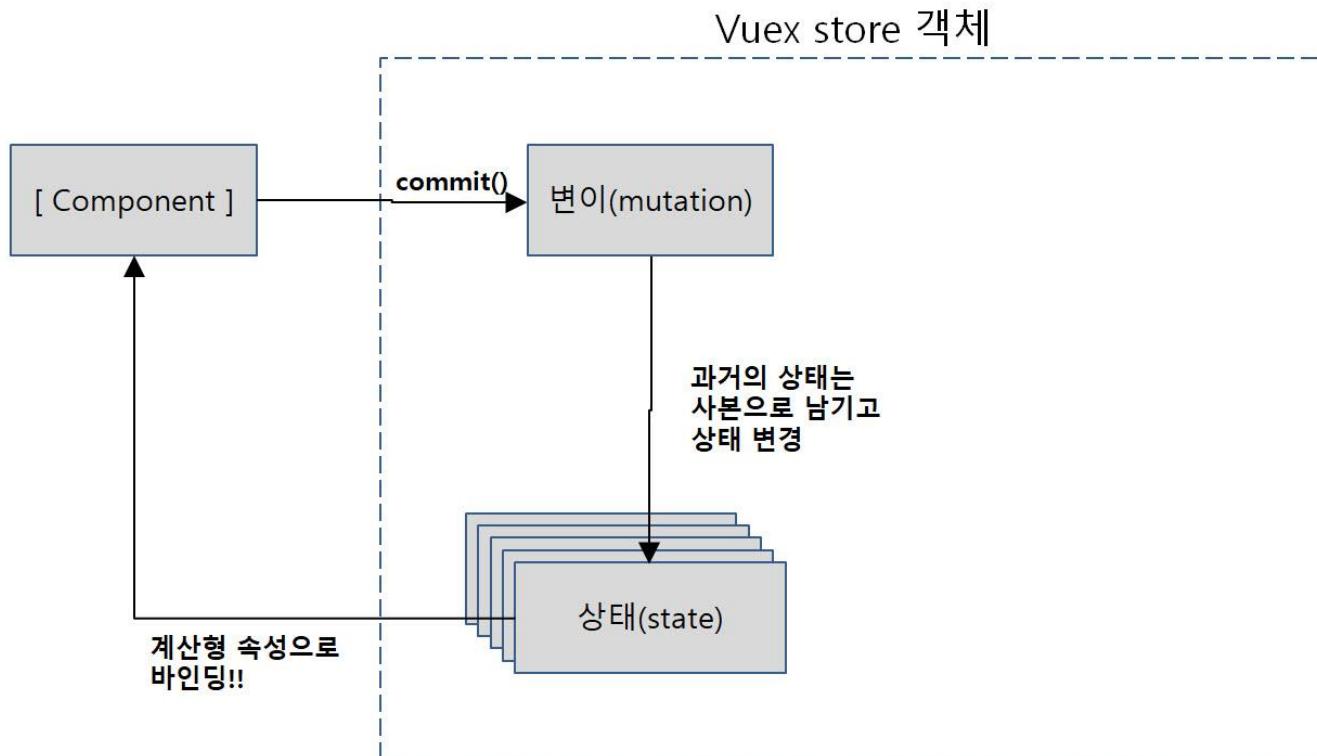
```
.....  
const store = new Vuex.Store({  
    state,  
    mutations,  
    actions,  
    strict : true  
})  
  
export default store;
```

- 이런 이유로 변이를 사용함. 변이는 변경 이력을 남김
  - 이전/이후 스냅샷을 저장

# 상태와 변이



## ■ 여기까지의 예제 앱 구조



# 상태와 변이



## ■ 변이(Mutations)에 비즈니스 로직을 집어넣으면?

- 특히 비동기 처리가 요구되는 로직을 집어넣으면 상태 변경 추적이 불가능해짐
- 변이는 동기적으로만 작동되므로 호출되고 즉시 commit됨
  - commi될 때 이전/이후 스냅샷을 기록함.
  - 비동기 처리 결과가 상태를 변경할 때는 이미 과거 내용으로 변경 이력을 저장해버린 상태
- 변이에서 비동기 처리를 했을 때의 Vue Devtools 그림

The screenshot shows the Vue Devtools interface. On the left, there's a browser window titled 'contactapp\_search' displaying a table of contacts. On the right, the Devtools panel is open, showing the state tree. A purple arrow points from the 'mutations' list to the 'state' section, highlighting the 'contacts' array. The mutations list shows a single entry: 'searchContact' at 11:27:42. The state tree shows 'state' with a child 'contacts: Array[0]'. Below the state tree, the 'mutation' section is expanded, showing a payload object with 'type: "searchContact"' and 'payload: Object' containing 'name: "ja"'.

번호	이름	전화번호	주소
1509814803154	Ambrosia Jackson	010-3456-8293	서울시
1509814803090	Drew Jackson	010-3456-8229	서울시
1509814803127	Jain Phillips	010-3456-8266	서울시
1509814803099	Kassie James	010-3456-8238	서울시
1509814803080	Koryne Jackson	010-3456-8219	서울시
1509814803082	Maima Jackson	010-3456-8221	서울시
1509814803093	Megan James	010-3456-8232	서울시
1509814803119	Nade Jackson	010-3456-8258	서울시

# 게터(Getters)



## ■ 게터란

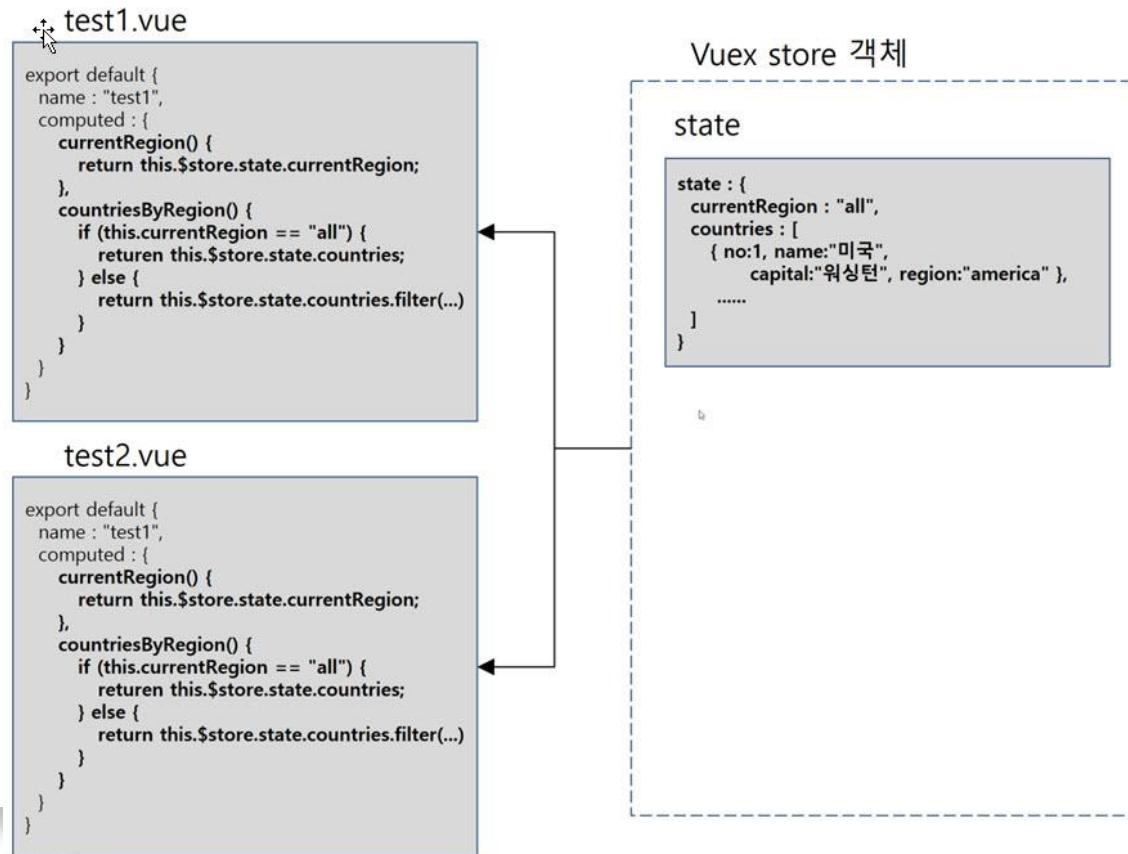
- 저장소(Store) 수준의 계산형 속성(Computed Property)
- 반드시 사용해야만 하는 것은 아님
- Vuex store의 상태를 이용해 연산이 필요한 계산형 속성이 여러 컴포넌트에서 반복적으로 사용되어지는 경우에 사용함.

# 게터(Getters)



## ■ 게터가 사용되지 않았을 때

- 저장소(store)는 상태만을 가지고 있음
- 각 컴포넌트가 계산형 속성으로 저장소 상태에 대한 연산을 직접 해야함.

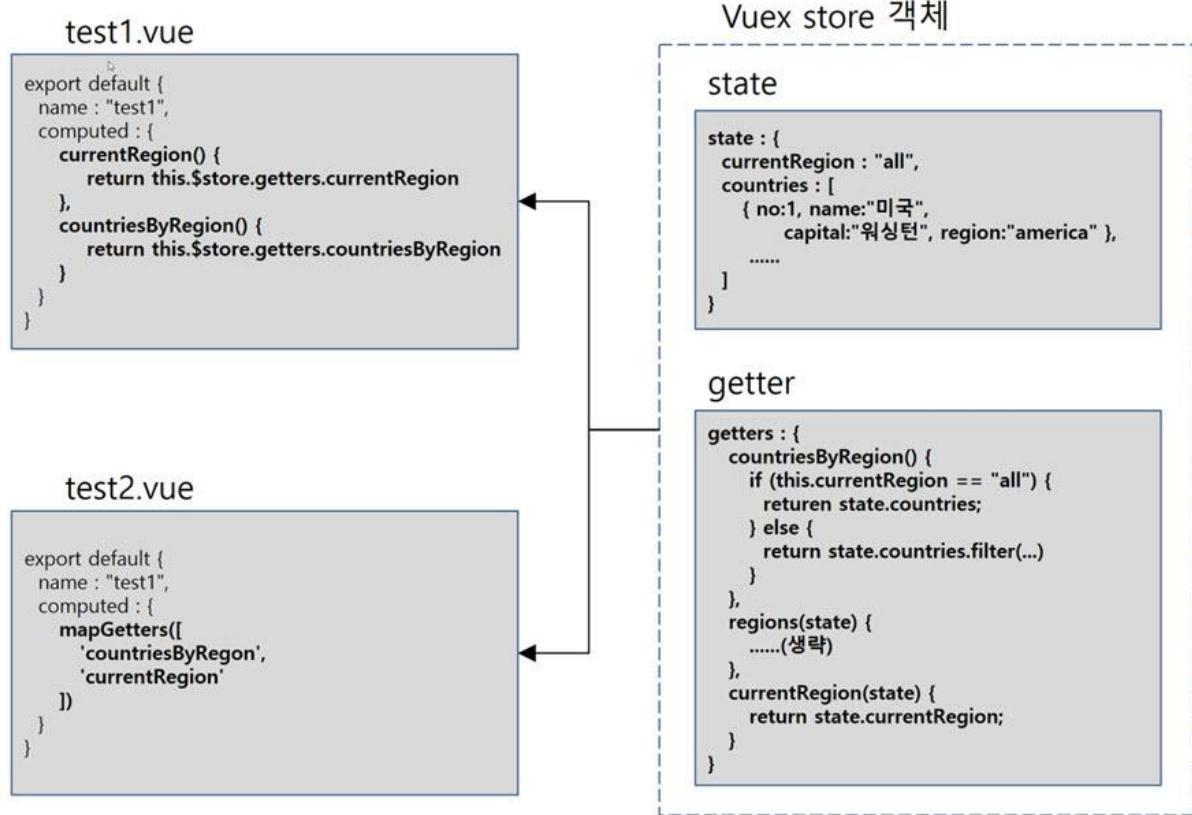


# 게터(Getters)



## ■ 게터가 사용된 경우

- 저장소의 게터를 여러 컴포넌트가 계산형 속성으로 바인딩하여 사용
- 컴포넌트에서의 코드 중복이 발생하지 않음



# 게터(Getters)



## ■ 예제 : 지역별 국가 정보 보여주기

번호	국가명	수도	지역
1	미국	워싱턴DC	america
11	자메이카	킹스턴	america
14	멕시코	멕시코시티	america
15	베네수엘라	카라카스	america

- 국가 정보에 있는 지역 정보를 이용해서 버튼을 생성해야 함.
- all을 누르면 전체 지역 정보 조회

[ 프로젝트 초기화 ]

```
vue init webpack-simple countryapp  
cd countryapp  
npm install
```

[ 필요한 라이브러리 추가 ]

```
npm install --save vuex lodash
```

# 게터(Getters)



## ■ 예제 11-07 : src/Constant.js

```
export default {
    CHANGE_REGION : "changeRegion"
}
```

## ■ 예제 11-08 : src/store/index.js

```
import Vue from 'vue';
import Vuex from 'vuex';
import Constant from '../Constant';
import _ from 'lodash';

Vue.use(Vuex);

const store = new Vuex.Store({
    state: {
        currentRegion : "all",
        countries : [
            { no:1, name : "미국", capital : "워싱턴DC", region:"america" },
            { no:2, name : "프랑스", capital : "파리", region:"europe" },
            .....
        ]
    },
    (다음 페이지에 이어집니다)
```

# 게터(Getters)



## ■ 예제 11-08 : src/store/index.js

(이전 페이지에 이어서)

```
getters : {
    countriesByRegion(state) {
        if (state.currentRegion == "all") {
            return state.countries;
        } else {
            return state.countries.filter(c => c.region==state.currentRegion);
        }
    },
    regions(state) {
        var temp = state.countries.map((c)=>c.region);
        temp = _.uniq(temp);
        temp.splice(0,0, "all");
        return temp;
    },
    currentRegion(state) {
        return state.currentRegion;
    }
},
mutations: {
    [Constant.CHANGE_REGION] : (state, payload) => {
        state.currentRegion = payload.region;
    }
}
}

export default store;
```

currentRegion을 이용해 특정 지역정보로 필터링!!

국가정보 중 region 필드를 이용해 중복을 제거하여 지역명 리스트 생성. 전체 지역 정보를 보여주기 위해 all 삽입.

상태 데이터의 이름이 변경되더라도 게터가 중간에서 완충해줌. 컴포넌트쪽에서 변경할 필요 없어짐.

# 게터(Getters)



- 예제 11-09 : src/main.js
- 예제 11-10 : src/App.vue

```
import Vue from 'vue'  
import App from './App.vue'  
import store from './store'  
new Vue({  
  store,  
  el: '#app',  
  render: h => h(App)  
})
```

```
<template>  
  <div id="app">  
    <region-buttons></region-buttons>  
    <country-list></country-list>  
  </div>  
</template>  
<script>  
  import RegionButtons from './components/RegionButtons.vue';  
  import CountryList from './components/CountryList.vue';  
  
  export default {  
    name : 'app',  
    components : { RegionButtons, CountryList }  
  }  
</script>  
<style>  
</style>
```

# 게터(Getters)



## ■ 예제 11-11: src/components/RegionButtons.vue

- mapState 헬퍼 메서드, lodash의 \_.extends 메서드 이용

```
<template>
<div>
  <button class="region" v-for="region in regions"
    v-bind:class="isSelected(region)"
    @click="changeRegion({region:region})">
    {{region}}
  </button>
</div>
</template>

<script>
import Constant from '../Constant'
import _ from 'lodash';
import { mapGetters, mapMutations } from 'vuex'

export default {
  name : "RegionButtons",
  computed : mapGetters([
    'regions', 'currentRegion'
  ]),
}
```

```
methods : _.extend(
  {
    isSelected(region) {
      if (region == this.currentRegion)
        return { selected: true };
      else
        return { selected:false }
    }
  },
  mapMutations([
    Constant.CHANGE_REGION
  ])
)
</script>

<style>
button.region { text-align: center; width:80px;
  margin:2px; border:solid 1px gray; }
button.selected { background-color:purple; color:aqua; }
</style>
```

# 게터(Getters)



## ■ 예제 11-12 : src/components/CountryList.vue

```
<template>
  <div id="exmaple">
    <table id="list">
      <thead>
        <tr>
          <th>번호</th><th>국가명</th>
          <th>수도</th><th>지역</th>
        </tr>
      </thead>
      <tbody id="contacts">
        <tr v-for="c in countries">
          <td>{{c.no}}</td>
          <td>{{c.name}}</td>
          <td>{{c.capital}}</td>
          <td>{{c.region}}</td>
        </tr>
      </tbody>
    </table>
  </div>
</template>
```

```
<script>
import Constant from '../Constant'
import { mapGetters } from 'vuex'
export default {
  name : 'CountryList',
  computed : mapGetters({
    countries : 'countriesByRegion'
  })
}
</script>
<style scoped>
  #list { width: 520px; border:1px solid black; border-collapse:collapse; }
  #list td, #list th { border:1px solid black; text-align:center; }
  #list td:nth-child(4n+1), #list th:nth-child(4n+1)
  { width:100px; }
  #list td:nth-child(4n+2), #list th:nth-child(4n+2)
  { width:150px; }
  #list td:nth-child(4n+3), #list th:nth-child(4n+3)
  { width:150px; }
  #list td:nth-child(4n), #list th:nth-child(4n+1)
  { width:120px; }
  #list > thead > tr { color:yellow; background-color:purple; }
</style>
```

# 게터(Getters)



## countryapp 실행 결과

countryapp

localhost:8080

번호	국가명	수도	지역
2	프랑스	파리	europe
3	영국	런던	europe

Ready. Detected Vue 2.3.4.

Filter mutations

Base State	changeRegion
22:23:58	00:39:39

state

```
currentRegion: "europe"
countries: Array[16]
```

getters

```
countriesByRegion: Array[2]
0: Object
1: Object
regions: Array[6]
currentRegion: "europe"
```

mutation

```
type: "changeRegion"
payload: Object
region: "europe"
```

# 액션(Actions)



## ■ 변이(mutations)의 한계

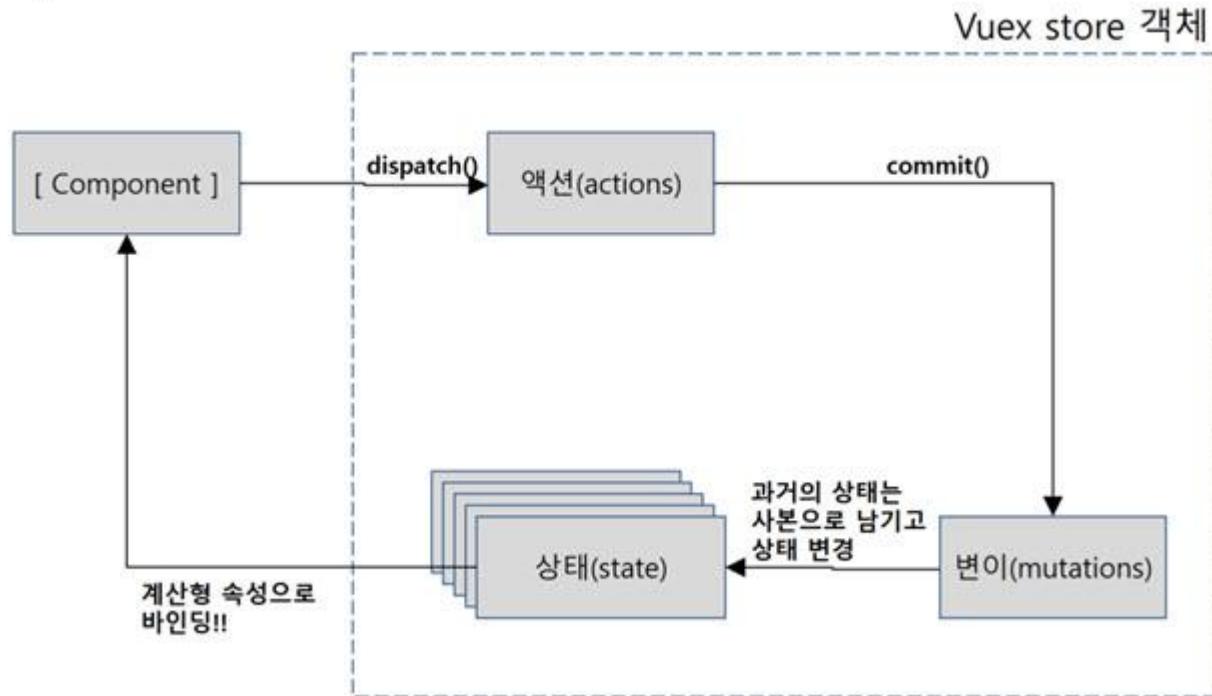
- 동기적인 작업만 수행
- 비동기적 작업을 수행하고 상태 변경을 추적가능하게 하려면 액션!!
- 단지 보기좋게 논리적 계층을 나눈 것이 아님. 분명한 필요성 존재
- 만일 외부 리소스를 이용하지 않고 앱의 메모리에서만 데이터가 관리된다면?
  - 액션이 필요없을 수도 있음. 변이만으로 충분
  - 하지만 앱이 백엔드 API, 외부 리소스를 이용하는 환경으로 변경될 수 있으므로 미리 액션을 준비해두는 것이 바람직함.

# 액션(Actions)



## 액션을 추가한 앱의 구조

- 외부 리소스, 백엔드 API에 의존하지 않을 때

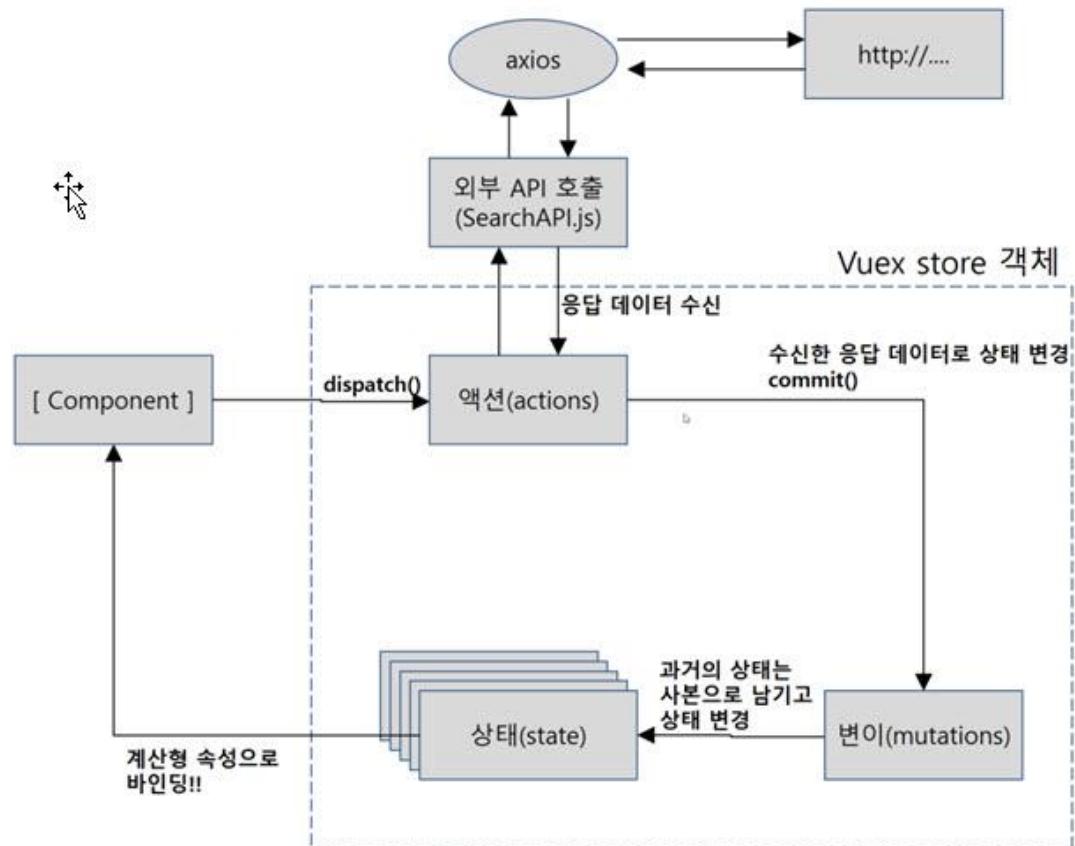


# 액션(Actions)



## 액션을 추가한 앱의 구조

- 외부 리소스, 백엔드 API를 이용하는 경우의 예



# 액션(Actions)



## ■ 기존 todolist 앱에 액션 추가

```
import Vue from 'vue';
import Vuex from 'vuex';
import Constant from '../constant';
Vue.use(Vuex);

const store = new Vuex.Store({
  state: { ..... (생략) },
  mutations: { .....(생략) },
  actions : {
    [Constant.ADD_TODO] : (store, payload) => {
      console.log("### addTodo!!!");
      store.commit(Constant.ADD_TODO, payload);
    },
    [Constant.DELETE_TODO] : (store, payload) => {
      console.log("### deleteTodo!!!");
      store.commit(Constant.DELETE_TODO, payload);
    },
    [Constant.DONE_TOGGLE] : (store, payload) => {
      console.log("### doneToggle!!!");
      store.commit(Constant.DONE_TOGGLE, payload);
    }
  }
})
export default store;
```

- src/store/index.js

- 첫번째 인자는 store 객체
- 두번째 인자는 payload

- store 객체

- Vuex Store 객체임

# 액션(Actions)



- 구조 분해 할당으로 store 객체의 필요한 속성만 이용!!

```
actions : {
  [Constant.ADD_TODO] : ({ state, commit }, payload) => {
    console.log("### addTodo!!!");
    console.log(state);
    commit(Constant.ADD_TODO, payload);
  },
  .....
}
```

## ■ 예제 11-14 : src/components/InputTodo.vue 변경

```
<script type="text/javascript">
import Constant from '../constant'

export default {
  .....
  methods : {
    addTodo : function() {
      this.$store.dispatch(Constant.ADD_TODO, { todo: this.todo });
      this.todo = "";
    }
  }
}
</script>
```

# 액션(Actions)



## ■ 예제 11-15 : src/components/List.vue 변경

```
<script type="text/javascript">
import Constant from '../constant'
import { mapState } from 'vuex'
import _ from 'lodash';

export default {
  computed: mapState(['todolist']),
  methods: {
    checked: function(done) {
      if(done) return { checked:true };
      else return { checked:false };
    },
    deleteTodo: function(payload) {
      this.$store.dispatch(Constant.DELETE_TODO, payload);
    },
    doneToggle: function(payload) {
      this.$store.dispatch(Constant.DONE_TOGGLE, payload);
    }
  }
}
</script>
```

# 액션(Actions)



## ■ mapActions 헬퍼 메서드

- 메서드 이름과 dispatch 하려는 액션의 이름이 동일하다면...

```
export default {
  computed : mapState([ 'todolist']),
  methods : _.extend({
    checked : function(done) {
      if(done) return { checked:true };
      else return { checked:false };
    }
  },
  mapActions([ Constant.DELETE_TODO, Constant.DONE_TOGGLE ])
)
}
```

## ■ 실행 결과

The screenshot shows a browser window titled "todolistapp" displaying a "Todo List App". The app has a purple header with the title "Todo List App". Below it is a search bar with placeholder text "입력 후 엔터!" and a "추가" button. The main area contains a table with two rows: "주말 산책 (완료)" and "잔실 야구장 (완료)". A sidebar on the left lists "영화보기", "저녁 약속", and "콘서트". To the right of the browser window is the browser's developer tools, specifically the "Console" tab. The console output shows several log entries:

```
###addTodo!!!
▶ object {__ob__: observer}
###doneToggle!!!
###addTodo!!!
▶ object {__ob__: observer}
###deleteTodo!!!

```

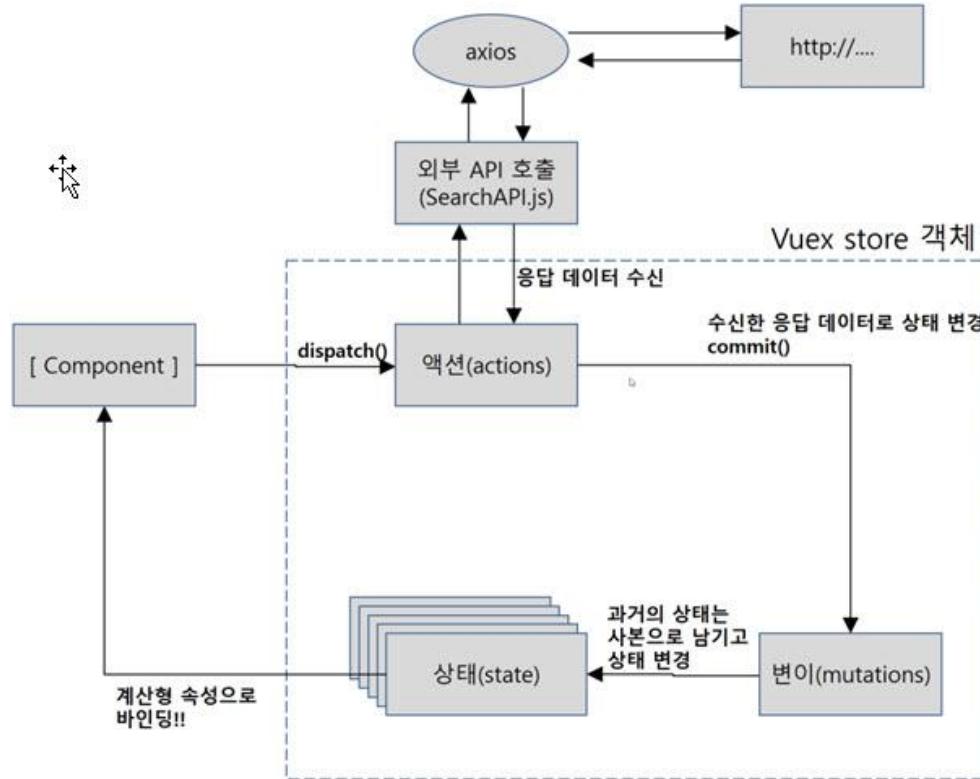
The log entries correspond to the actions triggered in the application, such as adding a new todo item, marking it as done, and deleting it.

# 액션(Actions)



## ■ 335페이지 contactsapp\_search 앱

- 액션에서 axios를 이용해 외부 API를 요청하도록 작성!!
- 자세한 코드 생략 : 335~343 페이지(예제 11-16~22)



# 액션(Actions)



## ■ 왜 액션 메서드의 첫번째 인자는 store인가?

- 변이(mutation)의 경우 state!!
  - 변이는 상태(state)의 변경이 주된 목적!!
- 액션은 상태의 변경을 제외한 다양한 작업을 수행
  - 액션은 다른 액션을 dispatch할 수 있음
    - 이것을 엮으면 꽤 복잡한 비즈니스 로직과 상태 변경을 처리할 수 있음.
  - 액션 상태를 이용한 작업을 수행할 수 있음(상태 변경은 제외)
  - 상태 변경을 위해서 변이를 호출함(commit 메서드 이용)
- 액션에서 상태를 직접 변경?
  - 할 수 있지만 권장하지 않음. → 상태 변경 내역 추적 불가

## ■ store 객체의 속성

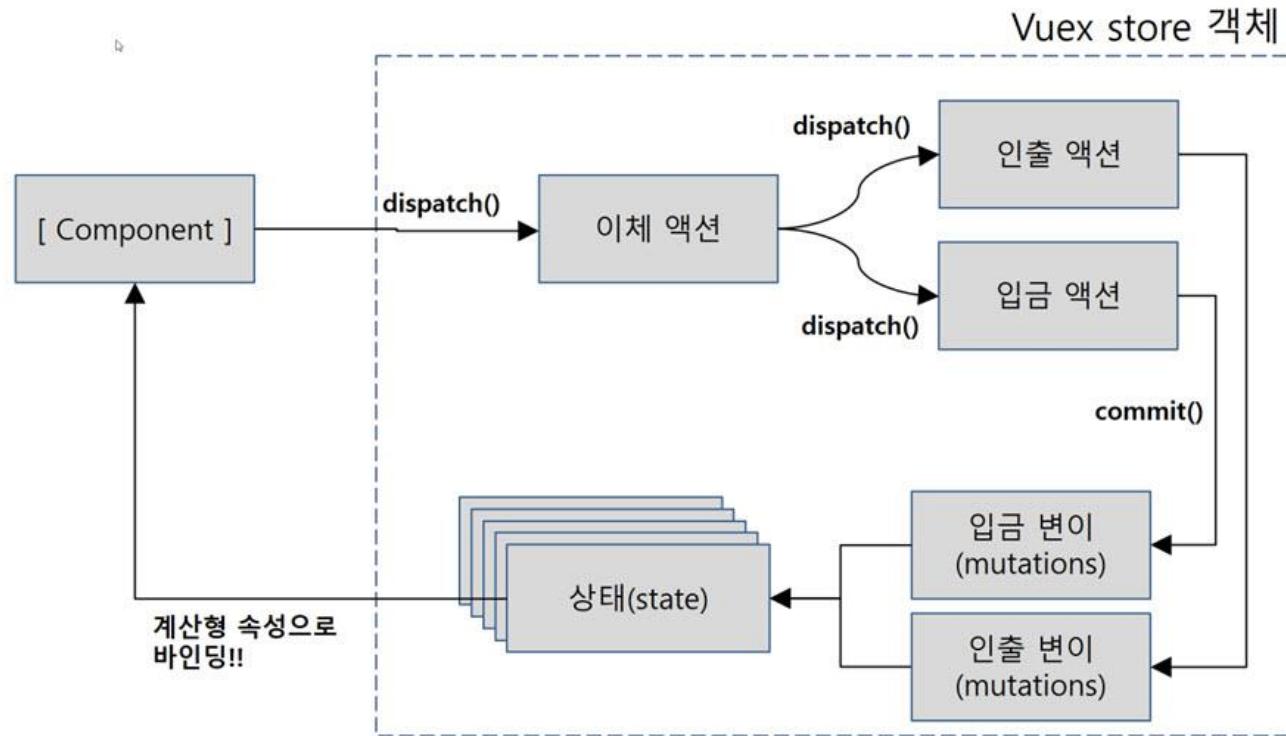
- 메서드 : dispatch(), commit()
- 속성 : getters, rootGetters, state, rootState

# 액션(Actions)



## ■ 액션이 다른 액션을 요청!!

- 이미 인출/입금 액션과 이와 관련된 변이도 작성되어 있다면?
  - 이제 액션을 추가할 때 중복 코드를 작성할 필요 없이 인출 액션, 입금 액션을 dispatch() 하는 것은 어떨까?



# 액션(Actions)



## ■ 이 책의 예제 11-34 코드

### ▪ ADD\_CONTACT 액션에서...

- Promise를 이용해 연락처 추가 API를 비동기로 호출
- 연락처 추가가 완료되면(then) 응답결과를 확인 후 연락처 조회 액션(FETCH\_CONTACTS)과 입력폼을 사라지도록 하는 액션(CANCEL\_FORM)을 dispatch 함!!

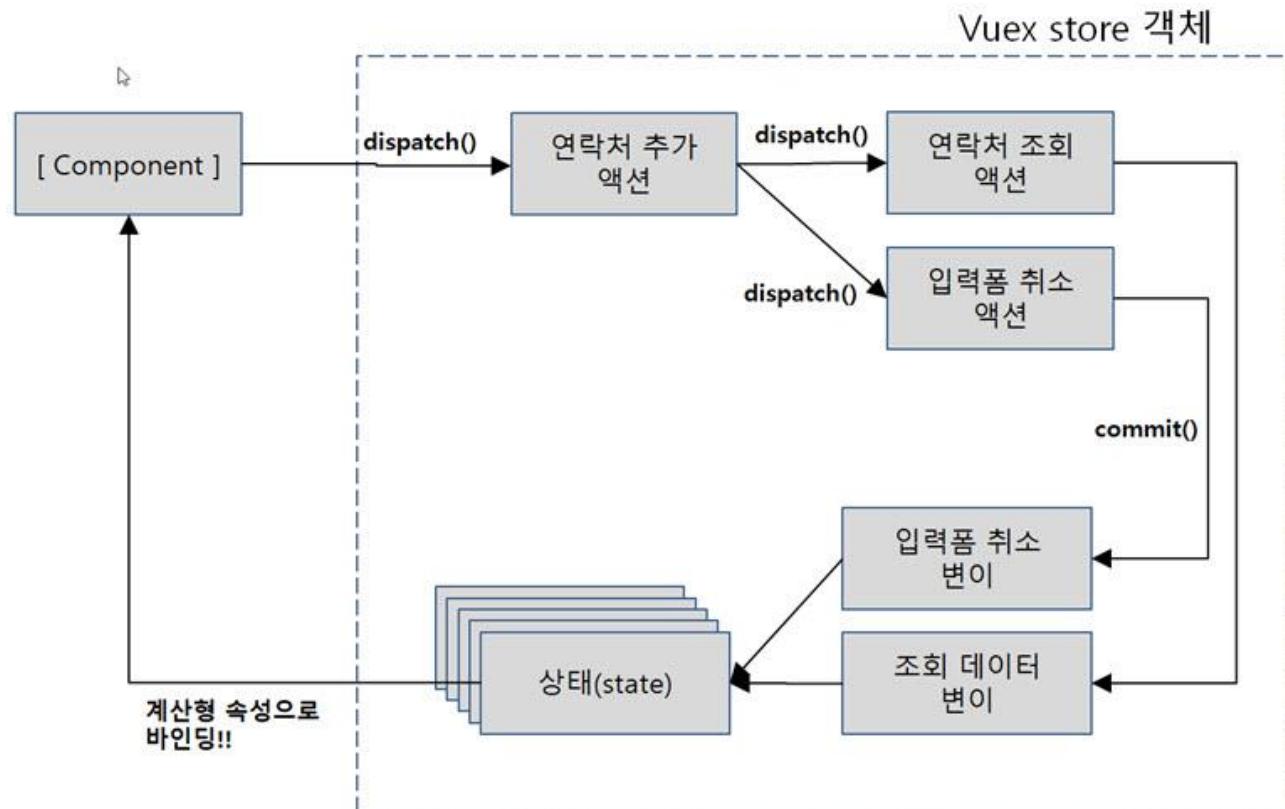
```
[Constant.ADD_CONTACT] : (store) => {
    contactAPI.addContact(store.state.contact)
    .then((response)=> {
        if (response.data.status == "success") {
            store.dispatch(Constant.CANCEL_FORM);
            store.dispatch(Constant.FETCH_CONTACTS, { pageno: 1});
        } else {
            console.log("연락처 추가 실패 : " + response.data);
        }
    })
},
```

# 액션(Actions)



## ■ 이 책의 예제 11-34 코드(이어서)

- 액션 작동 구조



# 대규모 애플리케이션의 저장소 파일



## ■ 지금까지의 저장소(Store) 객체 코드

- 하나의 .js 파일
- 간단한 애플리케이션이라면 문제없지만 대규모 앱인 경우 하나의 파일로 관리하기가 쉽지 않음
- 파일을 분리할 필요가 있음

## ■ 두가지 방법

- 역할별 분리 : 액션, 변이, 상태, 게터 등
- 모듈 사용

# 대규모 애플리케이션의 저장소 파일



## ■ 역할별 분리

- 상태, 변이, 액션, 게터의 역할 단위로 파일을 분리
- import 구문으로 store 디렉터리의 index.js를 참조(store 객체)
- store/index.js에서 import 구문으로 상태, 변이, 액션, 게터 .js 파일을 참조하여 구성
- 예제 11-23~26까지 참조

```
import Vue from 'vue';
import Vuex from 'vuex';
import state from './state.js';
import mutations from './mutations.js';
import actions from './actions.js';

Vue.use(Vuex);

const store = new Vuex.Store({
  state,
  mutations,
  actions
})

export default store;
```

# 대규모 애플리케이션의 저장소 파일

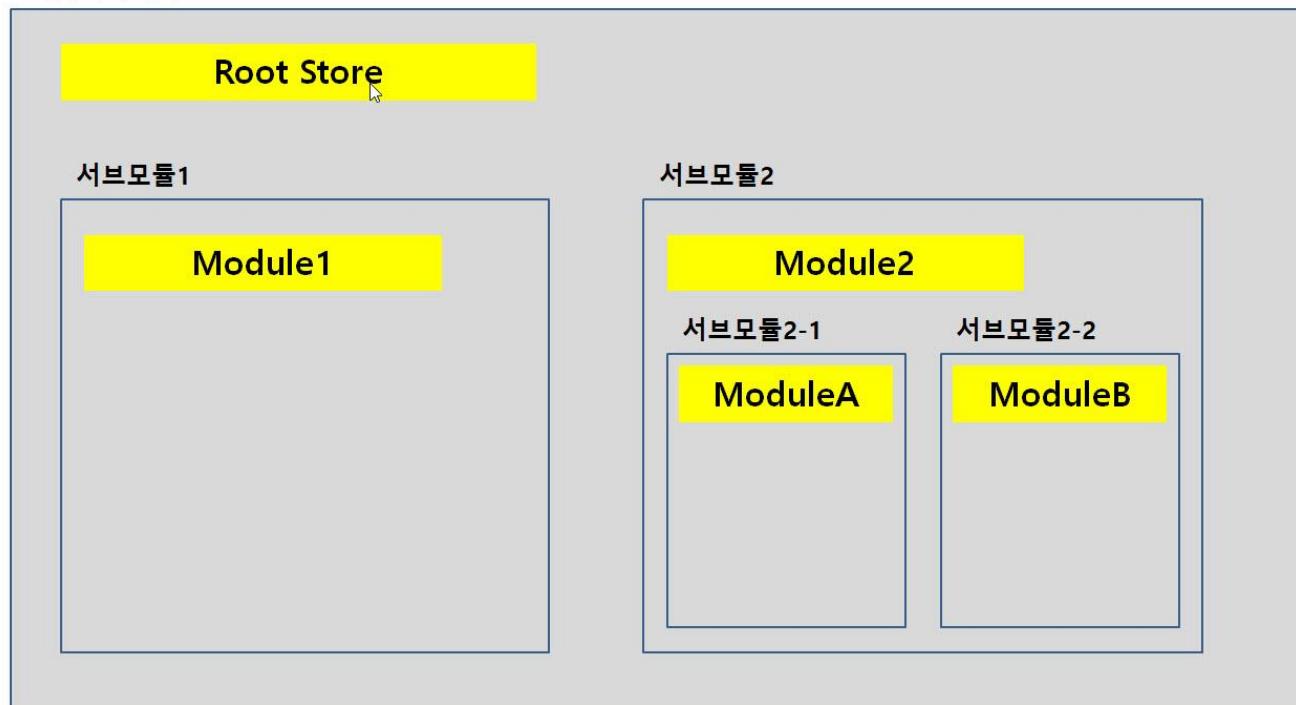


## ■ 모듈 이용

### ■ 대규모 애플리케이션에서 사용

- 대규모 앱에서는 여러개의 모듈로 나누어서 개발하는 경우가 많음.
- 서브모듈 단위로 상태(state)를 관리하고 루트 상태(rootState)는 서브모듈들이 공유

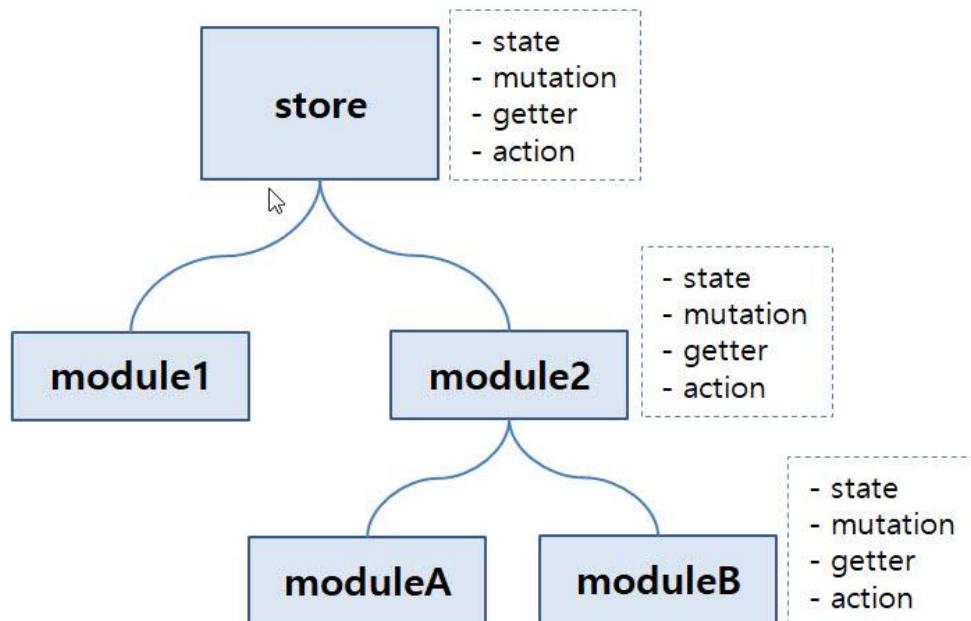
애플리케이션



# 대규모 애플리케이션의 저장소 파일



- store를 서브모듈 단위로 계층적으로 구성
  - 서브 모듈에서는 루트 저장소의 state, getter를 접근할 수 있음
  - 루트 저장소에서는 모듈의 state, getter에 직접 접근 불가(반드시 mutation, action 을 통해서만...)
  - 루트저장소 ↔ 서브모듈 간에 dispatch(), commit()은 자유롭게 호출 가능



# 대규모 애플리케이션의 저장소 파일



- store 객체의 속성(다시 한번) : 347페이지

속성명	설명
commit	변이를 일으키기 위한 메서드입니다.
dispatch	액션을 호출하기 위한 메서드입니다. 한 액션에서 다른 액션을 호출할 수 있습니다.
getters	모듈 자기 자신의 게터입니다
rootGetters	루트 저장소의 게터입니다.
state	모듈 자기 자신의 상태 데이터입니다.
rootState	루트 저장소의 상태 데이터입니다.

## ■ 예제 11-27~29 : contactsapp\_search02 프로젝트

- github.com/stepanowon/vuejs\_book을 통해서 제공
- 검색된 연락처 목록
  - Module에서 관리
- 검색어 리스트
  - 루트 저장소에서 관리

이름 :

최근 검색이름 리스트 :

- no
- an
- jo
- ia

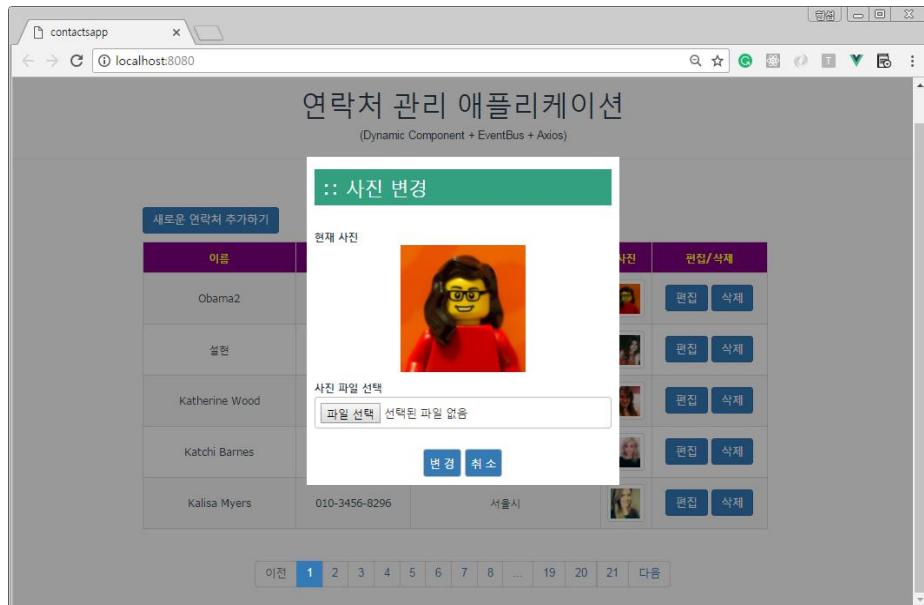
번호	이름	전화번호	주소
1509555603510	Noel Reed	010-3456-8241	서울시
1509555603550	Zenon Cooper	010-3456-8281	서울시

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 10장의 연락처 애플리케이션

- EventBus 객체를 이용해 상태 관리
- 어지러운 props, event



## ■ vuex를 이용하도록 코드 변경!!

# Vuex를 이용 연락처 애플리케이션 작성



## 필요 라이브러리 추가 다운로드

- npm install --save vuex lodash

## 예제 11-30(상수정의) : src/constant.js

```
export default {
    FETCH_CONTACTS : "fetchContacts",           //연락처 조회
    ADD_CONTACT_FORM : "addContactForm",         //입력폼 나타내기
    ADD_CONTACT : "addContact",                  //연락처 추가
    EDIT_CONTACT_FORM : "editContactForm",        //수정폼 나타내기
    UPDATE_CONTACT : "updateContact",            //연락처 수정
    CANCEL_FORM : "cancelForm",                  //입력,수정폼 닫기
    EDIT_PHOTO_FORM : "editPhotoForm",            //사진 편집폼 나타내기
    UPDATE_PHOTO : "updatePhoto",                //사진 수정
    DELETE_CONTACT : "deleteContact",           //연락처 삭제
}
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-31 : src/api/ContactsAPI.js

```
import CONF from '../Config.js';
import axios from 'axios';

export default {
    fetchContacts : function(pageno, pagesize) {
        return axios.get(CONF.FETCH, {
            params : { pageno: pageno, pagesize: pagesize }
        })
    },
    fetchContactOne : function(no) {
        return axios.get(CONF.FETCH_ONE.replace("${no}", no))
    },
    addContact : function(contact) {
        return axios.post(CONF.ADD, contact);
    },
    updateContact : function(contact) {
        return axios.put(CONF.UPDATE.replace("${no}", contact.no), contact)
    },
    deleteContact : function(no) {
        return axios.delete(CONF.DELETE.replace("${no}", no))
    },
    updatePhoto : function(no, file) {
        var data = new FormData();
        data.append('photo', file);
        return axios.post(CONF.UPDATE_PHOTO.replace("${no}", no), data)
    }
}
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-32 : src/store/state.js

- 기존 App.vue의 data 옵션을 그대로 가져옴

```
import Constant from './constant';
import CONF from './Config.js';

export default {
  currentView : null,
  mode : 'add',
  contact : { no:0, name:"", tel:"", address:"", photo:@"" },
  contactlist : {
    pageno:1,
    pagesize: CONF.PAGESIZE,
    totalcount:0,
    contacts:[]
  }
}
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-33 : src/store/mutations.js

- 상태를 변경하는 기능만을 뽑아서 변이로 작성

```
import Constant from './constant';
//상태를 변경하는 기능만을 뽑아서...
export default {
  [Constant.ADD_CONTACT_FORM] : (state) => {
    state.contact = { no:"", name:"", tel:"", address:"", photo:@"" };
    state.mode = "add";
    state.currentView = "contactForm";
  },
  [Constant.CANCEL_FORM] : (state) => {
    state.currentView = null;
  },
  [Constant.EDIT_CONTACT_FORM] : (state, payload) => {
    state.contact = payload.contact;
    state.mode = "update";
    state.currentView = "contactForm";
  },
  [Constant.EDIT_PHOTO_FORM] : (state, payload) => {
    state.contact = payload.contact;
    state.currentView = "updatePhoto";
  },
  [Constant.FETCH_CONTACTS] : (state, payload) => {
    state.contactlist = payload.contactlist;
  }
}
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-34 : src/store/actions.js

- 앞에서 작성한 src/api/ContactsAPI.js 호출하고 그 결과를 이용해 상태 변경
- 다음의 액션은 다른 액션을 호출함.
  - ADD\_CONTACT
  - UPDATE\_CONTACT
  - UPDATE\_PHOTO
  - DELETE\_CONTACT
- axios 를 이용해 API 호출 후 리턴된 값은 Promise 객체
  - API 서비스 요청 후 응답이 수신되면 then() 실행

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-35 : src/store/index.js

- Store 객체

```
import Vue from 'vue';
import Vuex from 'vuex';
import state from './state.js';
import mutations from './mutations.js';
import actions from './actions.js';
import ES6Promise from 'es6-promise';

ES6Promise.polyfill();
Vue.use(Vuex);
const store = new Vuex.Store({
  state,
  mutations,
  actions
})
export default store;
```

- 이제 EventBus 객체는 더이상 필요하지 않음
- 컴포넌트에서는 State를 계산형 속성에 바인딩하여 UI를 구성

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 이제부터 컴포넌트 작성

### ▪ 컴포넌트 측에서의 변화

- App.vue의 모든 상태(data옵션)와 상태 관련 메서드들은 모두 Vuex Store 객체로...
- AddContact.vue, UpdateContact.vue를 사용하지 않고 ContactForm.vue를 직접 사용 → mode 데이터 옵션을 Vuex의 상태 데이터로 직접 관리하기 때문에 가능함.

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-36 : src/App.vue

- data 옵션이 사라지고 계산형 속성을 추가하여 Vuex Store의 상태를 바인딩함.

```
<script>
.....
export default {
  name: 'app',
  components : { ContactList, ContactForm,
    UpdatePhoto, Paginate },
  computed : _.extend({
    totalpage : function() {
      var totalcount = this.contactlist.totalcount;
      var pagesize = this.contactlist.pagesize;
      return Math.floor((totalcount - 1) / pagesize) + 1;
    }
  }, mapState([ 'contactlist', 'currentView' ])
),
watch : {
  'contactlist.pageno' : function(page) {
    this.$refs.pagebuttons.selected= page-1;
  }
},
```

```
  mounted : function() {
    this.$store.dispatch(Constant.FETCH_CONTACTS);
  },
  methods : {
    pageChanged : function(page) {
      this.$store.dispatch(Constant.FETCH_CONTACTS,
{ pageno:page })
    }
  }
</script>
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-37 : src/components/ContactList.vue

- props를 이용해 App.vue로부터 데이터를 전달받을 필요 없음

```
<script>
import Constant from '../constant';
import { mapState } from 'vuex';

export default {
  name : 'contactList',
  computed : mapState([ 'contactlist' ]),
  methods : {
    addContact : function() {
      this.$store.dispatch(Constant.ADD_CONTACT_FORM);
    },
    editContact : function(no) {
      this.$store.dispatch(Constant.EDIT_CONTACT_FORM, {no:no});
    },
    deleteContact : function(no) {
      if (confirm("정말로 삭제하시겠습니까?") == true) {
        this.$store.dispatch(Constant.DELETE_CONTACT, {no:no});
      }
    },
    editPhoto : function(no) {
      this.$store.dispatch(Constant.EDIT_PHOTO_FORM, {no:no});
    }
  }
}
</script>
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-38 : src/components/ContactForm.vue

- mode에 따라 약간 다른 화면이 나타나야 함.

```
<script>
import Constant from '../constant';
import { mapState } from 'vuex';
import _ from 'lodash';

export default {
  name : "contactForm",
  computed : _.extend({
    btnText : function() {
      if (this.mode != 'update') return '추가';
      else return '수정';
    },
    headingText : function() {
      if (this.mode != 'update') return '새로운 연락처 추가';
      else return '연락처 변경';
    },
  },
  mapState([ 'mode', 'contact' ])
),
mounted : function() {
  this.$refs.name.focus()
},
```

```
methods : {
  submitEvent : function() {
    if (this.mode == "update") {
      this.$store.dispatch(Constant.UPDATE_CONTACT);
    } else {
      this.$store.dispatch(Constant.ADD_CONTACT);
    }
  },
  cancelEvent : function() {
    this.$store.dispatch(Constant.CANCEL_FORM);
  }
}
</script>
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 예제 11-39 : src/main.js

- store 객체 주입

```
import Vue from 'vue'  
import App from './App'  
import store from './store'  
  
Vue.config.productionTip = false  
  
new Vue({  
  store,  
  el: '#app',  
  template: '<App/>',  
  components: { App }  
})
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ src/components/UpdatePhoto.vue

- ContactForm.vue의 변경 사항과 유사함.
- 책에서는 생략되어 있음

```
<script>
import Constant from '../constant';
import { mapState } from 'vuex';

export default {
    name : "updatePhoto",
    computed : mapState([ 'contact' ]),
    methods : {
        cancelEvent : function() {
            this.$store.dispatch(Constant.CANCEL_FORM);
        },
        photoSubmit : function() {
            var file = this.$refs.photofile.files[0];
            this.$store.dispatch(Constant.UPDATE_PHOTO, { no:this.contact.no, file:file });
        }
    }
}
</script>
```

# Vuex를 이용 연락처 애플리케이션 작성



## ■ 실행

- npm run dev로 실행한 후 Vue Devtools를 열어서 Vuex 변이 정보를 확인함.

The screenshot illustrates the development environment for a contact management application. On the left, a browser window titled 'contactsapp' shows the application's interface. The main title is '연락처 관리 애플리케이션 (Dynamic Component + Vuex + Axios)'. Below it is a table listing contacts with columns for 이름 (Name), 전화번호 (Phone Number), 주소 (Address), 사진 (Photo), and 편집/삭제 (Edit/Delete). The table contains five entries. At the bottom, there is a navigation bar with page numbers from 1 to 21. On the right, the Vue Devtools extension is open in the browser's developer tools. The 'Vue' tab is selected, showing the detected version as 'Ready. Detected Vue 2.3.4.' Below this, a list of mutations is shown, each with a timestamp. To the right of the mutations, the 'state' and 'getters' sections are displayed. The state section shows variables like currentView, mode, contact, contactlist, contacts, pageno, pagesize, and totalcount. The getters section shows payload and contactlist. An arrow points from the 'Vue' tab in the Devtools to the circular icon representing the Vuex store in the Devtools sidebar.



# Vuex Strict Mode

## ■ v-model 디렉티브

- 양방향 데이터 바인딩!!
- Vuex state를 컴포넌트에서 계산형 속성으로 연결하고 v-model 디렉티브로 바인딩하면
  - 사용자 입력시 state가 바로 변경될 수 있음. 이것은 추적 불가능한 state 변경을 일으킴.
  - 이것을 막기 위해 strict mode 사용
- Strict Mode 사용시 처리 방법
  - 첫번째 방법
    - v-model 디렉티브로 state를 변경하지 않고 변이, 액션을 추가하고 changed 이벤트를 이용해 변이를 일으킴
  - 두번째 방법
    - 컴포넌트에서 로컬 상태(data 옵션)를 정의하고 created 와 같은 이벤트 흙에서 Vuex state를 로컬 data 옵션에 복사
    - 컴포넌트에 UI 이벤트가 발생하면 액션 → 변이를 거쳐 Vuex 상태가 변경되도록 함.
    - 책에서 제공되는 예제 중 contactsapp\_strict 프로젝트를 참조



## ■ Vuex 를 이용한 상태 관리 방법

- 단방향 데이터 흐름
  - 컴포넌트 → 액션 → 변이 → 상태 → 컴포넌트
- 상태가 중심이 됨.
  - 상태를 중심으로 변이를 정의하고
  - 컴포넌트에서 일어나는 액션을 정의
  - 액션으로 인해 어떤 변이를 일으킬지를 정의함.