

Vuex

Vuex is a **state management pattern + library** for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion.

<https://vuex.vuejs.org/>

Для чого потрібе Vue.js?

Розглянемо меджер задач

Додати задачу

Задача

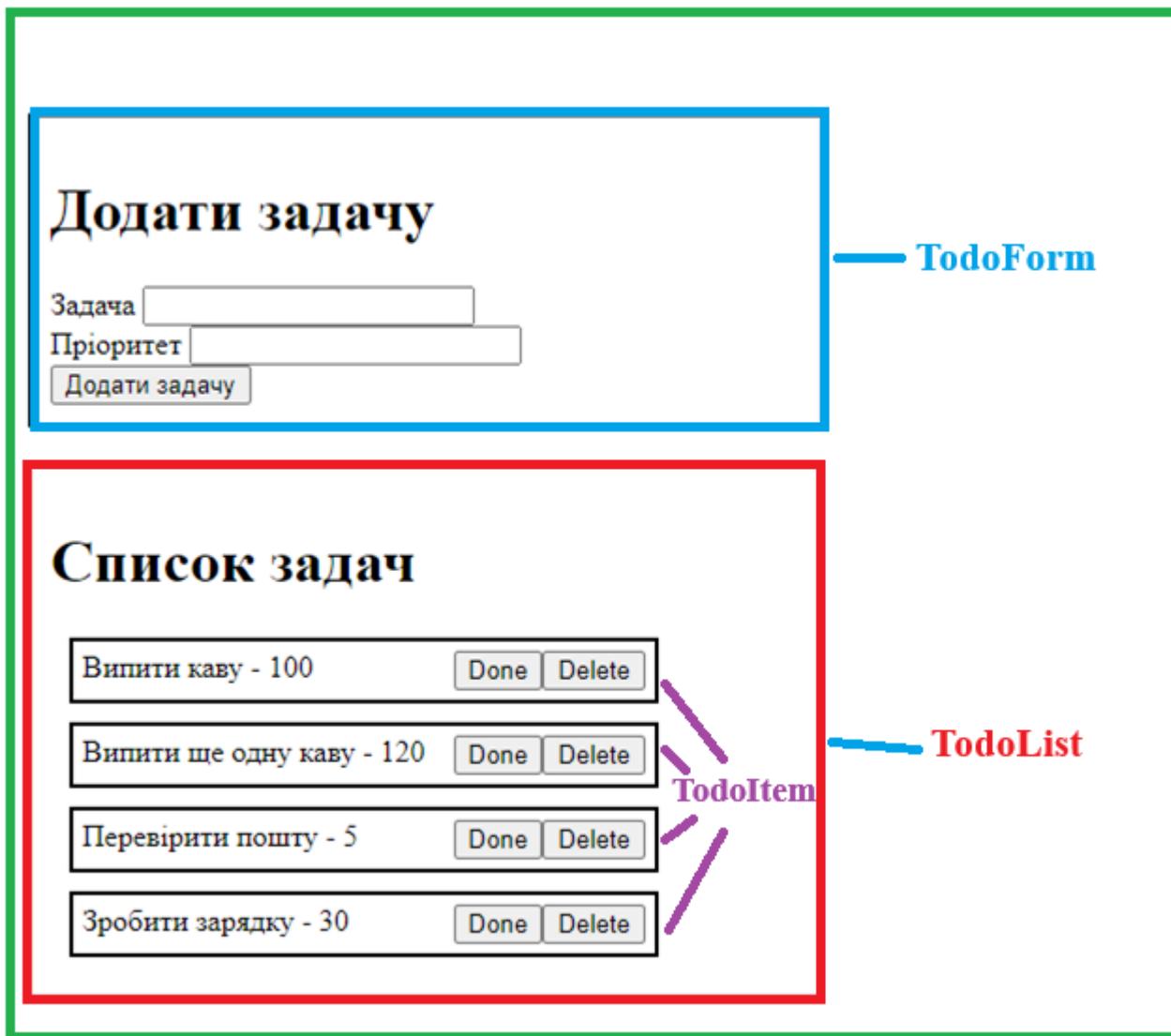
Пріоритет

Список задач

| | | |
|---------------------------|-------------------------------------|---------------------------------------|
| Випити каву - 100 | <input type="button" value="Done"/> | <input type="button" value="Delete"/> |
| Випити ще одну каву - 120 | <input type="button" value="Done"/> | <input type="button" value="Delete"/> |
| Перевірити пошту - 5 | <input type="button" value="Done"/> | <input type="button" value="Delete"/> |
| Зробити зарядку - 30 | <input type="button" value="Done"/> | <input type="button" value="Delete"/> |

Для чого потрібе Vuex?

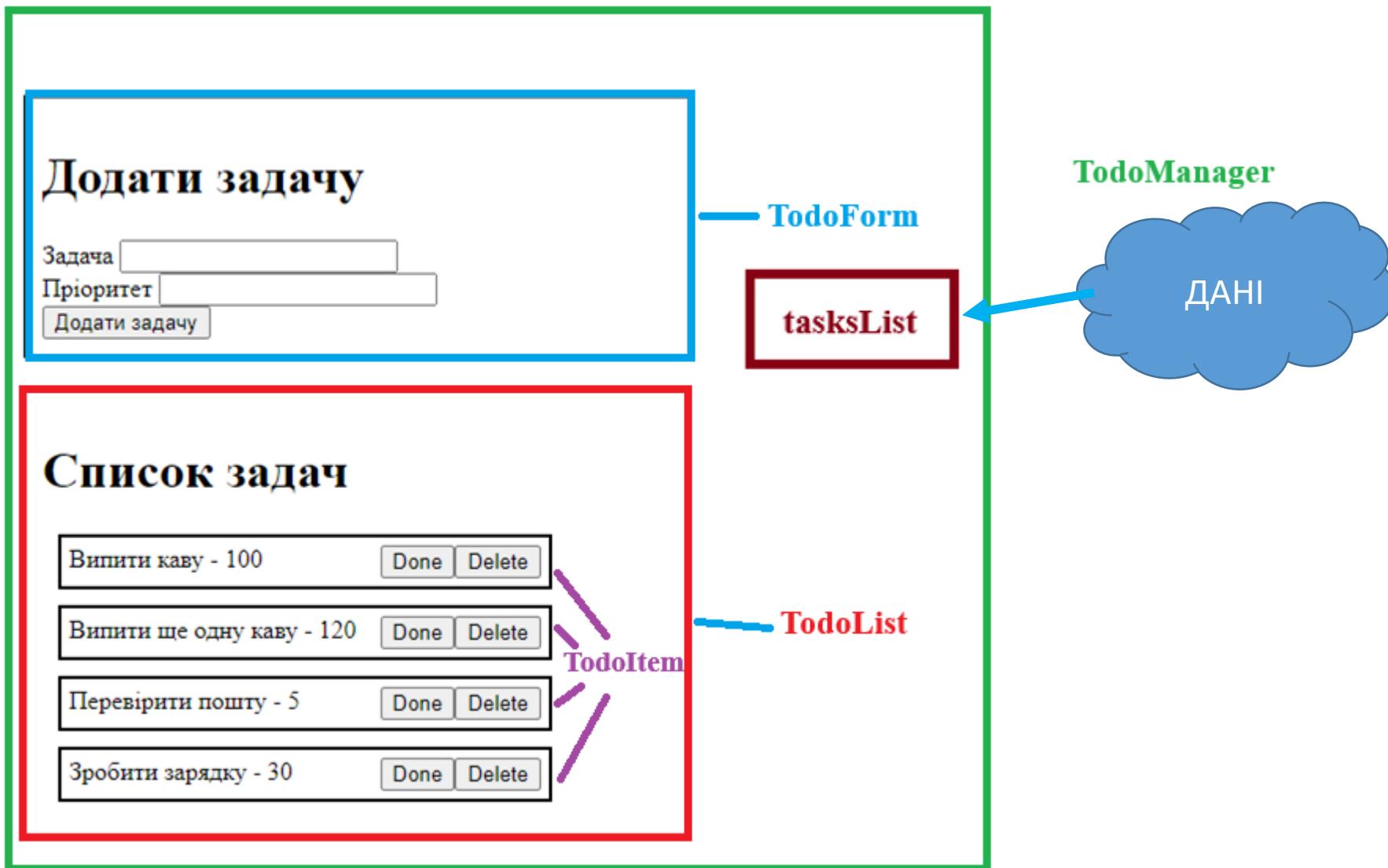
Розглянемо меджер задач



TodoManager

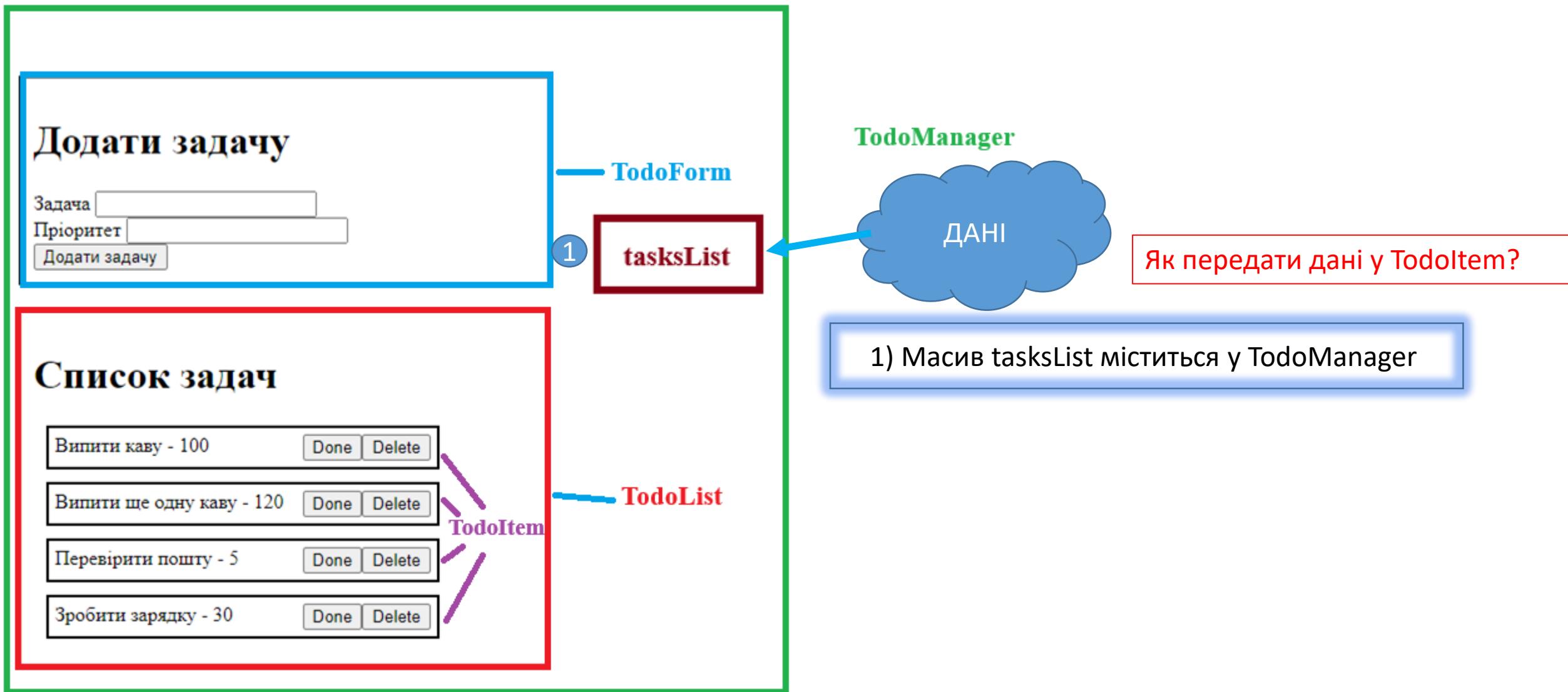
Для чого потрібе Vuex?

Розглянемо меджер задач



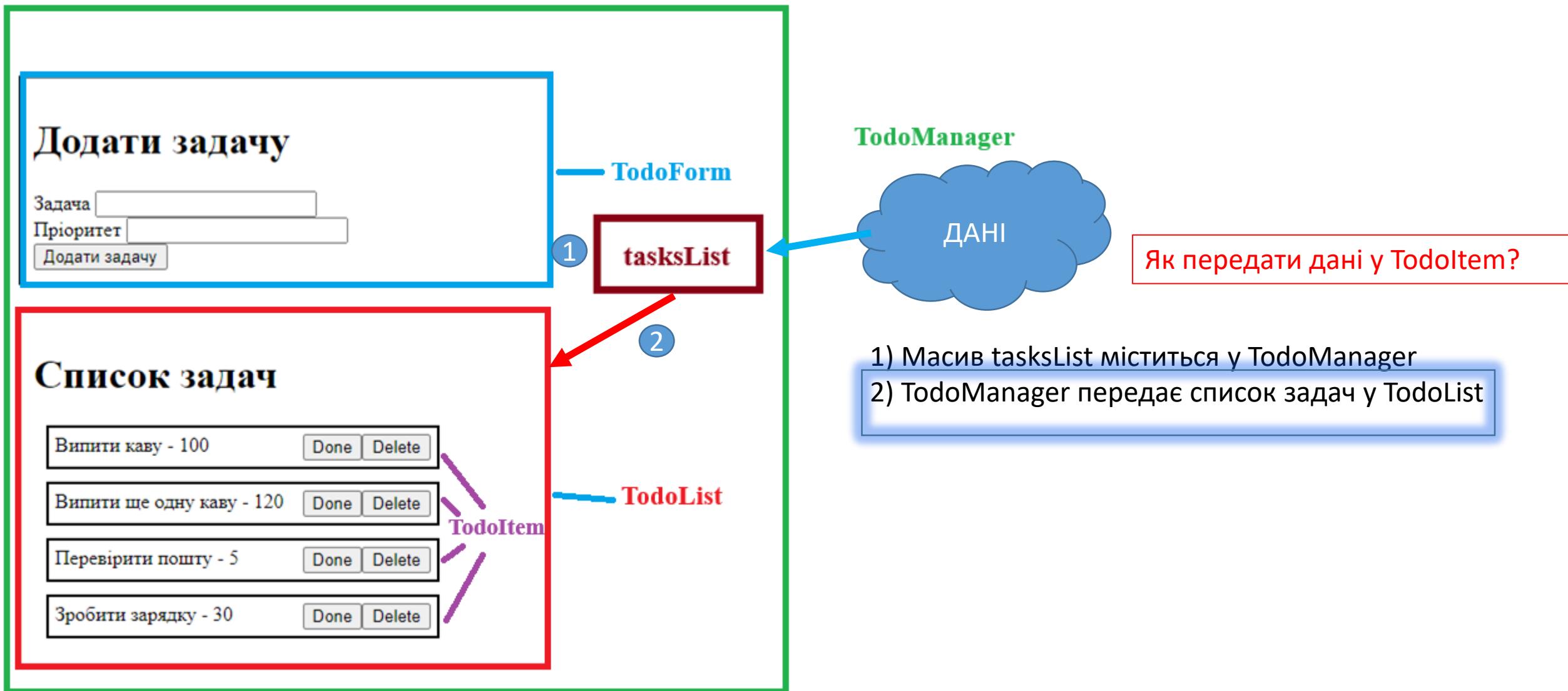
Для чого потрібе Vuex?

Розглянемо меджер задач



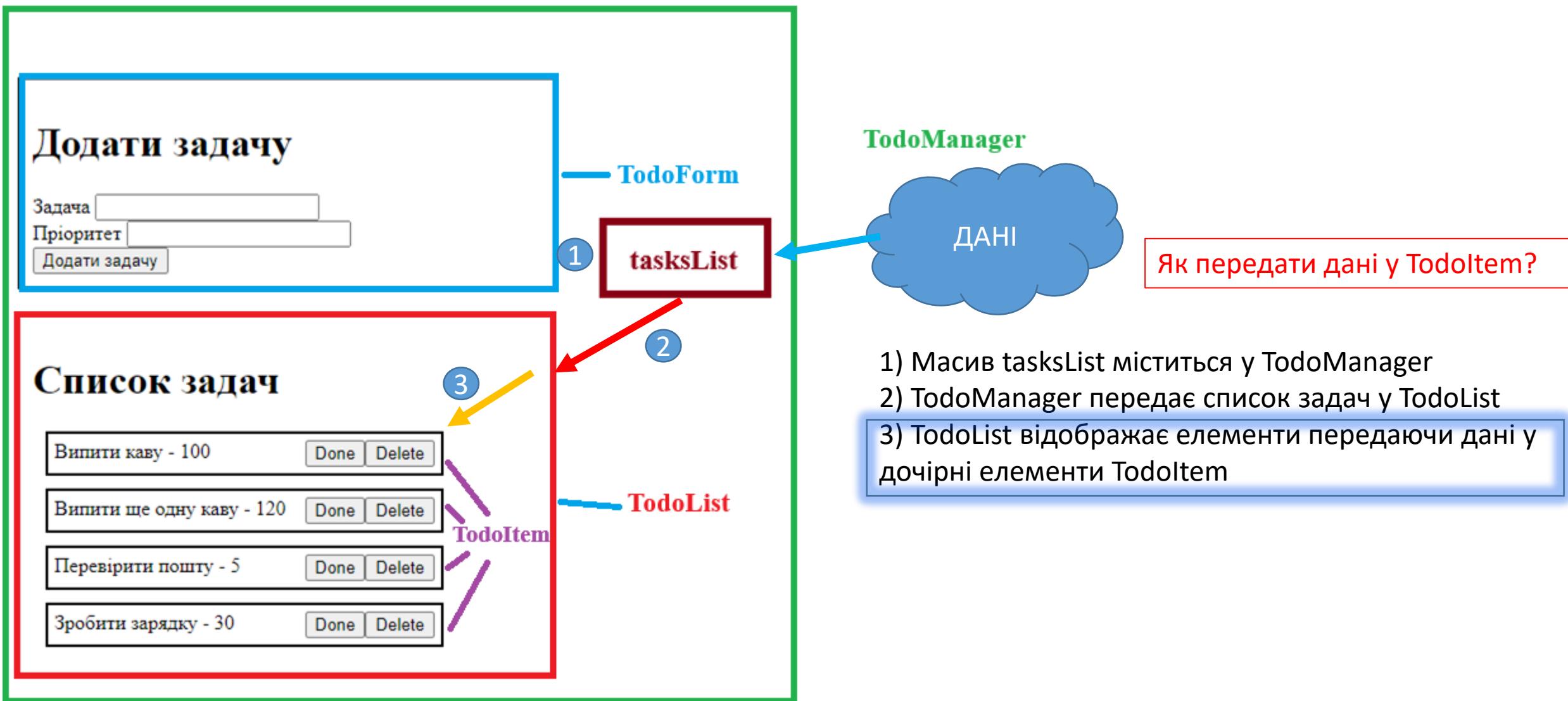
Для чого потрібе Vuex?

Розглянемо меджер задач



Для чого потрібе Vuex?

Розглянемо меджер задач



The image shows three code editors side-by-side, each displaying a different component of a Vue.js application:

- TodoManager.vue**:
A template section contains a `<todo-form @add="addTask" />` and a `<todo-list :tasks-list="todoListData" @done="onDone" @delete="onDelete" />`.
The script section imports `TodoForm` and `TodoList`, and defines a `todoListData` array.
A large blue arrow points from the `TodoManager` component towards the `TodoList` component.
- TodoList.vue**:
The template section includes a header `<h1>Список задач</h1>` and a loop `<todo-item v-for="task in tasksList" :key="task.id" :task="task" @done="$emit('done', $event)" @delete="$emit('delete', $event)" />`.
The script section imports `TodoItem` and defines the `TodoList` component with a `tasksList` prop.
- TodoItem.vue**:
The template section contains a `<div class="task-container"><div>{{ task.title }} - {{ task.priority }}</div><button v-if="!isDone" @click="$emit('done', task.id)">Done</button><button @click="$emit('delete', task.id)">Delete</button></div></div>`.
The script section defines the `CustomInput` component with a `task` prop.

1) Масив `tasksList` міститься у `TodoManager`

The diagram illustrates the data flow between three Vue components: TodoManager.vue, TodoList.vue, and TodoItem.vue.

TodoManager.vue (Left Panel):

- Contains a template with a `<todo-form @add="addTask" />` and a `<todo-list :tasks-list="todoListData" @done="onDone" @delete="onDelete" />`.
- An annotation labeled "1" points to the `todoListData: []` declaration in the data() method.
- An annotation labeled "2" points to the `@done="onDone" @delete="onDelete"` event handlers.

TodoList.vue (Middle Panel):

- Contains a template with a `<div class="list-container">` containing an `<h1>Список задач</h1>` and a `<todo-item v-for="task in tasksList" :key="task.id" :task="task" @done="$emit('done', $event)" @delete="$emit('delete', $event)" />`.
- An annotation labeled "1" points to the `components: { TodoItem }` declaration in the data() method.
- An annotation labeled "2" points to the `props: { tasksList: { type: Array, default: () => [] } }` declaration in the data() method.

TodoItem.vue (Right Panel):

- Contains a template with a `<div class="task-container">` containing `<div>{{ task.title }} - {{ task.priority }}</div>`, a `<button v-if="!isDone" @click="$emit('done', task.id)">Done</button>`, and a `<button @click="$emit('delete', task.id)">Delete</button>`.

1) Масив tasksList міститься у TodoManager

2) TodoManager передає список задач у TodoList

The diagram illustrates the data flow between three Vue components: TodoManager.vue, TodoList.vue, and TodoItem.vue.

TodoManager.vue (Left):

- Contains a `data()` method that returns an empty array: `todoListData: []`. (1)
- Contains a `methods` section with a `@add="addTask"` event handler. (2)
- Imports `TodoForm` and `TodoList`.

TodoList.vue (Middle):

- Contains a template with a `<todo-item>` component. (3)
- Contains a script section with a `v-for="task in tasksList"` loop.
- Contains a script section with `emits: ['done', 'delete']`.
- Contains a props section with `tasksList: { type: Array, default: () => [] }`.

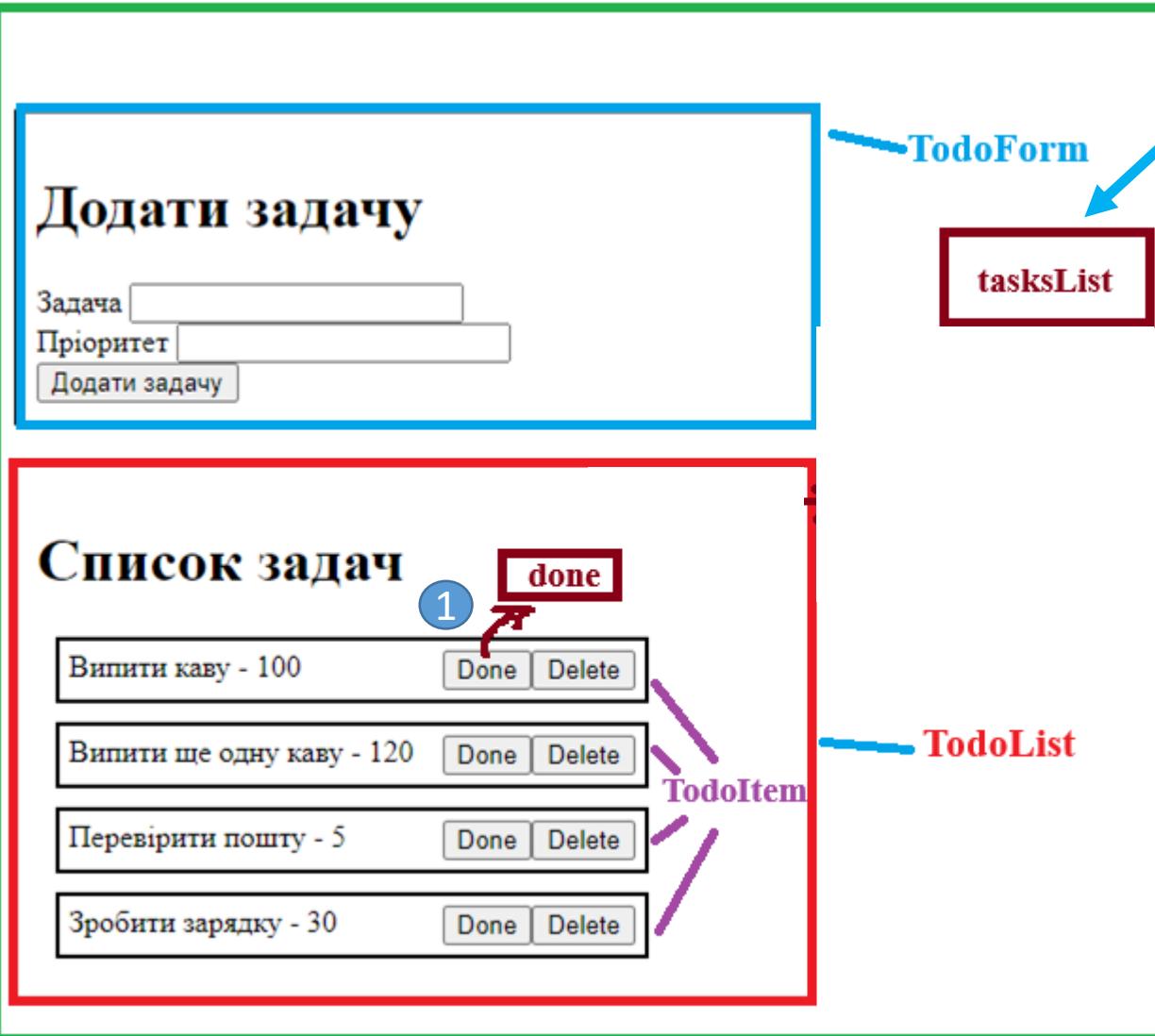
TodoItem.vue (Right):

- Contains a template with a `<div>` element containing `task.title` and `task.priority`.
- Contains a script section with `props: { task: { type: Object, required: true } }`.

- 1) Масив `tasksList` міститься у `TodoManager`
- 2) `TodoManager` передає список задач у `TodoList`
- 3) `TodoList` відображає елементи

Для чого потрібе Vuex?

Розглянемо меджер задач



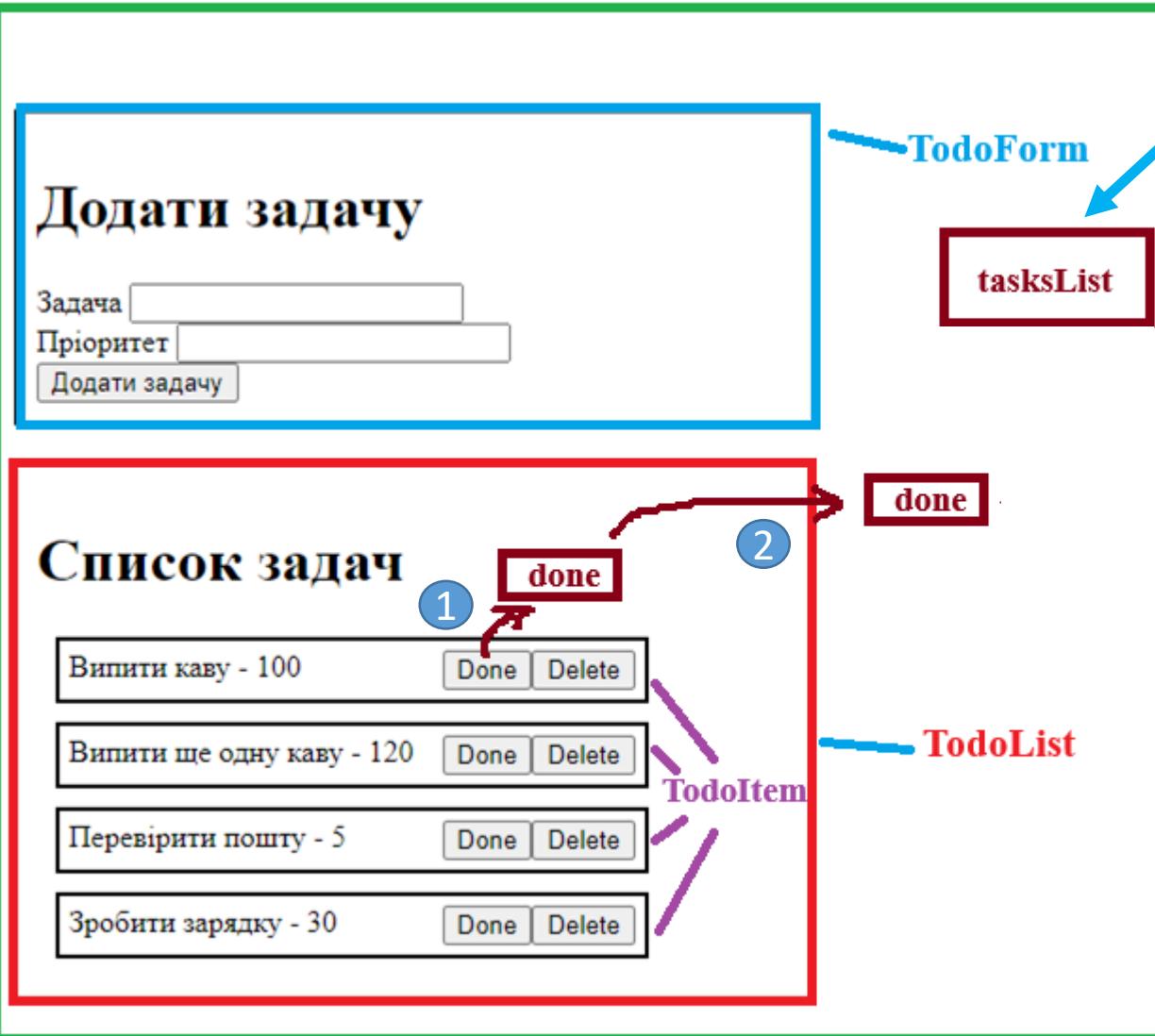
TodoManager

Для виконання операцій у випадку вкладених компонтів доводиться передавати події по ланцюжку вкладеності від дочірніх до батьківських

При необхідності зміни статусу задачі:
1) компонент **TodoItem** генерує подію “done”

Для чого потрібе Vuex?

Розглянемо меджер задач



TodoManager

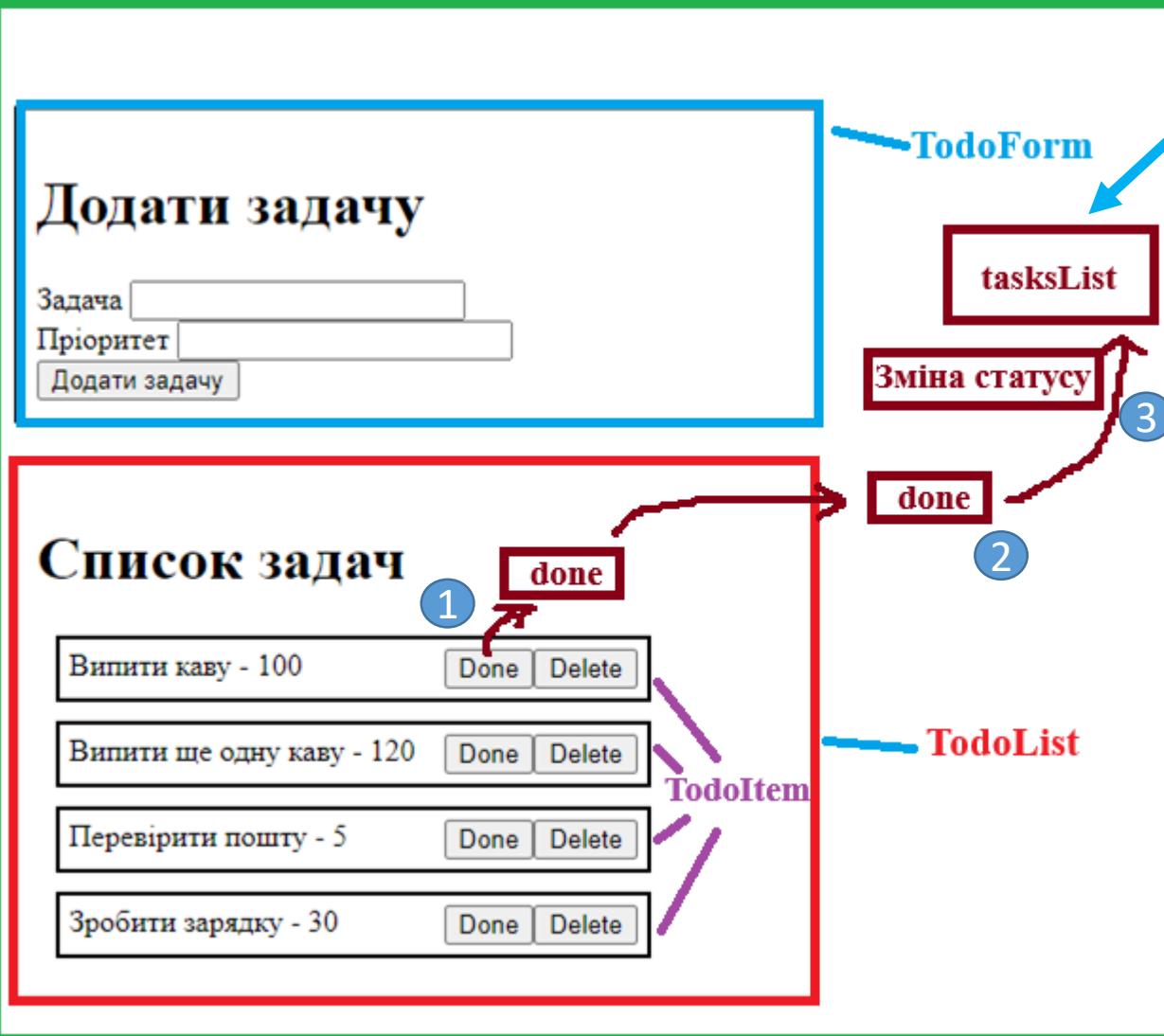
Для виконання операцій у випадку вкладених компонтів доводиться передавати події по ланцюжку вкладеності від дочірніх до батьківських

При необхідності зміни статусу задачі:

- 1) компонент **TodoItem** генерує подію “done”
- 2) батьківський компонент **TodoList** отримує подію “done” від дочірнього **TodoItem** і також генерує подію “done”

Для чого потрібе Vuex?

Розглянемо меджер задач



При необхідності зміни статусу задачі:

- 1) компонент **TodoItem** генерує подію “done”
- 2) батьківський компонент **TodoList** отримує подію “done” від дочірнього **TodoItem** і також генерує подію “done”
- 3) **TodoManager** отримує подію “done” від дочірнього **TodoList** і робить зміни у списку

Для чого потрібе VueX?

Розглянемо меджер задач

The screenshot shows three code editors side-by-side:

- TodoManager.vue**:

```
> <template>
  <todo-form @add="addTask" />
  <todo-list :tasks-list="todoListData"
    @done="onDone"
    @delete="onDelete" />
</template>
<script>
import TodoForm from './TodoForm.vue'
import TodoList from './TodoList.vue'
export default {
  name: 'TodoManager',
  components: { TodoForm, TodoList },
  data() {
    return {
      todoListData: []
    }
  },
  methods: {
  },
  </script>
TodoForm
<style lang="scss" scoped></style>
```
- TodoList.vue**:

```
src > components > <template>
  <div class="list-container">
    <h1>Список задач</h1>
    <todo-item
      v-for="task in tasksList"
      :key="task.id"
      :task="task"
      @done="$emit('done', $event)"
      @delete="$emit('delete', $event)">
    </todo-item>
  </div>
</template>
<script>
import TodoItem from './TodoItem.vue'
export default {
  name: 'TodoList',
  components: { TodoItem },
  emits: [ 'done', 'delete' ],
  props: {
    tasksList: {
      type: Array,
      default: () => []
    }
  }
}</script>
```
- Todolitem.vue**:

```
src > components > <template>
  <div class="task-container">
    <div>{{ task.title }} - {{ task.priority }}</div>
    <div>
      <button v-if="!isDone" @click="$emit('done', task.id)">Done</button>
      <button @click="$emit('delete', task.id)">Delete</button>
    </div>
  </div>
</template>
<script>
export default {
  name: 'CustomInput',
  props: {
    task: {
      type: Object,
      required: true
    }
  },
  computed: {
  }
}</script>
```

A yellow arrow points from the `@done` and `@delete` event handlers in **TodoList.vue** to the corresponding `$emit` calls in **Todolitem.vue**. A blue circle with the number "1" is placed over the `done` event handler in **Todolitem.vue**.

При необхідності зміни статусу задачі:

1) компонент Todolitem генерує подію “done”

Для чого потрібе VueX?

Розглянемо меджер задач

The diagram illustrates the event flow between three Vue components:

- TodoManager.vue** (highlighted with a red oval):
 - Emits a `$emit('done', $event)` when a task is marked as done.
 - Emits a `$emit('delete', $event)` when a task is deleted.
- TodoList.vue** (highlighted with a blue oval):
 - Receives the `'done'` event from `@done="$emit('done', $event)"`.
 - Receives the `'delete'` event from `@delete="$emit('delete', $event)"`.
- TodoItem.vue** (highlighted with a green oval):
 - Generates a `v-if="!isDone" @click="$emit('done', task.id)">Done` button.
 - Generates a `<button @click="$emit('delete', task.id)">Delete` button.

Annotations:

- Annotation 1:** A yellow arrow points from the `$emit('done', task.id)` call in **TodoItem.vue** to the `@done="$emit('done', $event)"` event handler in **TodoList.vue**.
- Annotation 2:** A yellow arrow points from the `$emit('delete', task.id)` call in **TodoItem.vue** to the `@delete="$emit('delete', $event)"` event handler in **TodoList.vue**.

```
TodoManager.vue code:  
1 <template>  
2   <todo-form @add="addTask" />  
3   <todo-list :tasks-list="todoListData" @done="onDone" @delete="onDelete" />  
4 </template>  
5  
6 <script>  
7   import TodoForm from './TodoForm.vue'  
8   import TodoList from './TodoList.vue'  
9   export default {  
10    name: 'TodoManager',  
11    components: { TodoForm, TodoList },  
12    data() {  
13      return {  
14        todoListData: [],  
15      }  
16    },  
17    methods: {  
18      ...  
19    },  
20  },  
21  </script>  
22  TodoForm  
23  <style lang="scss" scoped></style>
```

```
TodoList.vue code:  
1 <template>  
2   <div class="list-container">  
3     <h1>Список задач</h1>  
4     <todo-item  
5       v-for="task in tasksList"  
6       :key="task.id"  
7       :task="task"  
8       @done="$emit('done', $event)"  
9       @delete="$emit('delete', $event)"  
10      />  
11    </div>  
12  </template>  
13  <script>  
14    import TodoItem from './TodoItem.vue'  
15    export default {  
16      name: 'TodoList',  
17      components: { TodoItem },  
18      emits: [ 'done', 'delete' ],  
19      props: {  
20        tasksList: {  
21          type: Array,  
22          default: () => []  
23        },  
24      },  
25    },  
26  </script>
```

```
TodoItem.vue code:  
1 <template>  
2   <div class="task-container">  
3     <div>{{ task.title }} - {{ task.priority }}</div>  
4     <div>  
5       <button v-if="!isDone" @click="$emit('done', task.id)">Done</button>  
6       <button @click="$emit('delete', task.id)">Delete</button>  
7     </div>  
8   </div>  
9 </template>  
10 <script>  
11   export default {  
12     name: 'CustomInput',  
13     props: {  
14       task: {  
15         type: Object,  
16         required: true,  
17       },  
18     },  
19     computed: {  
20       ...  
21     },  
22   },  
23   </script>
```

При необхідності зміни статусу задачі:

- 1) компонент **TodoItem** генерує подію “done”
- 2) батьківський компонент **TodoList** отримує подію “done” від дочірнього **TodoItem** і також генерує подію “done”

Для чого потрібе Vuex?

Розглянемо меджер задач

The diagram illustrates the flow of the 'done' event across three Vue components:

- TodoManager.vue** (highlighted with a red circle):
 - Emits the 'done' event (step 3).
- TodoList.vue** (highlighted with a yellow circle):
 - Receives the 'done' event from TodoManager.vue.
 - Emits the 'done' event (step 2).
- TodoItem.vue** (highlighted with a blue circle):
 - Receives the 'done' event from TodoList.vue.
 - Contains logic to handle the 'done' event (step 1).

```
<template>
<div class="list-container">
  <h1>Список задач</h1>
  <todo-item
    v-for="task in tasksList"
    :key="task.id"
    :task="task"
    @done="$emit('done', $event)"
    @delete="$emit('delete', $event)">
  </todo-item>
</div>
</template>
<script>
import TodoItem from './TodoItem.vue'
export default {
  name: 'TodoList',
  components: { TodoItem },
  emits: ['done', 'delete'],
  props: {
    tasksList: {
      type: Array,
      default: () => []
    }
  }
}</script>
```

```
<template>
<div class="task-container">
  <div>{{ task.title }} - {{ task.priority }}</div>
  <button v-if="!isDone" @click="$emit('done', task.id)">Done</button>
  <button @click="$emit('delete', task.id)">Delete</button>
</div>
</template>
<script>
import TodoManager from './TodoManager.vue'
export default {
  name: 'CustomInput',
  props: {
    task: {
      type: Object,
      required: true
    }
  },
  computed: {
    isDone() {
      return this.task.isDone
    }
  }
}</script>
```

```
<template>
<div><todo-form @add="addTask" />
<todo-list :tasks-list="todoListData"
  @done="onDone"
  @delete="onDelete" />
</div>
</template>
<script>
import TodoForm from './TodoForm.vue'
import TodoList from './TodoList.vue'
export default {
  name: 'TodoManager',
  components: { TodoForm, TodoList },
  data() {
    return {
      todoListData: []
    }
  },
  methods: {
    addTask(task) {
      this.todoListData.push(task)
    },
    onDone(task) {
      const index = this.todoListData.indexOf(task)
      if (index !== -1) {
        this.todoListData.splice(index, 1)
      }
    },
    onDelete(task) {
      this.todoListData = this.todoListData.filter(item => item !== task)
    }
  }
}</script>
<style lang="scss" scoped></style>
```

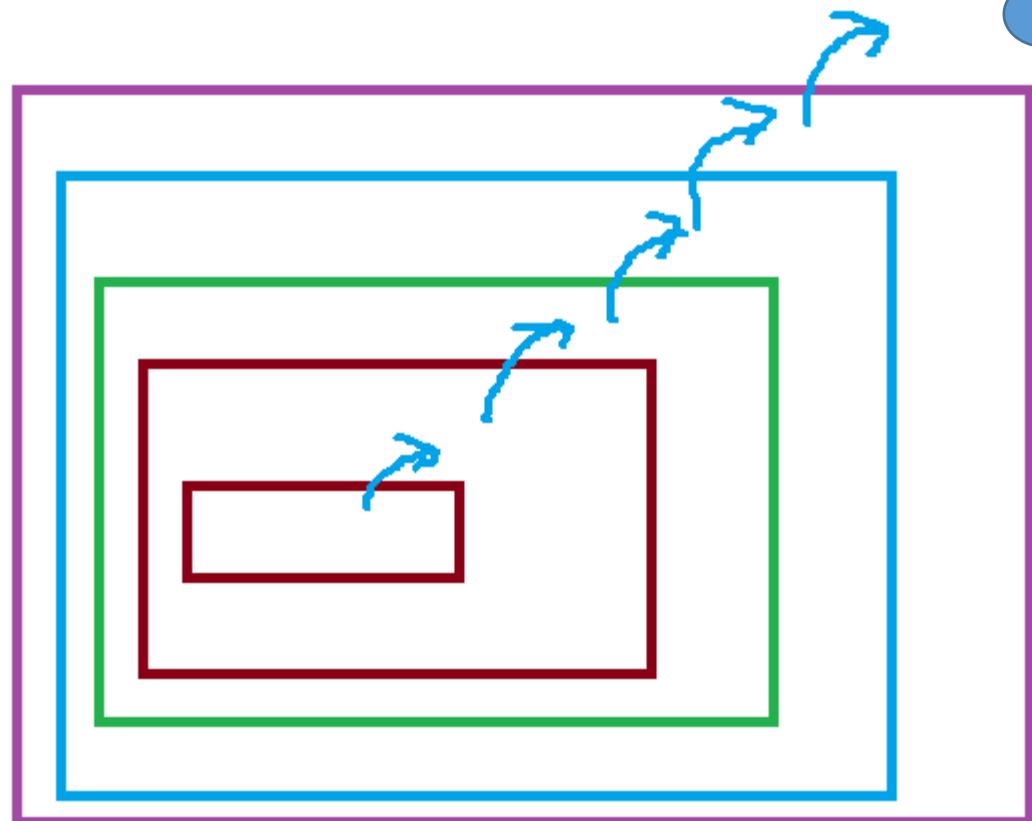
При необхідності зміни статусу задачі:

- 1) компонент TodoItem генерує подію "done"
- 2) батьківський компонент TodoList отримує подію "done" від дочірнього TodoItem і також генерує подію "done"
- 3) TodoManager отримує подію "done" від дочірнього TodoList і робить зміни у списку

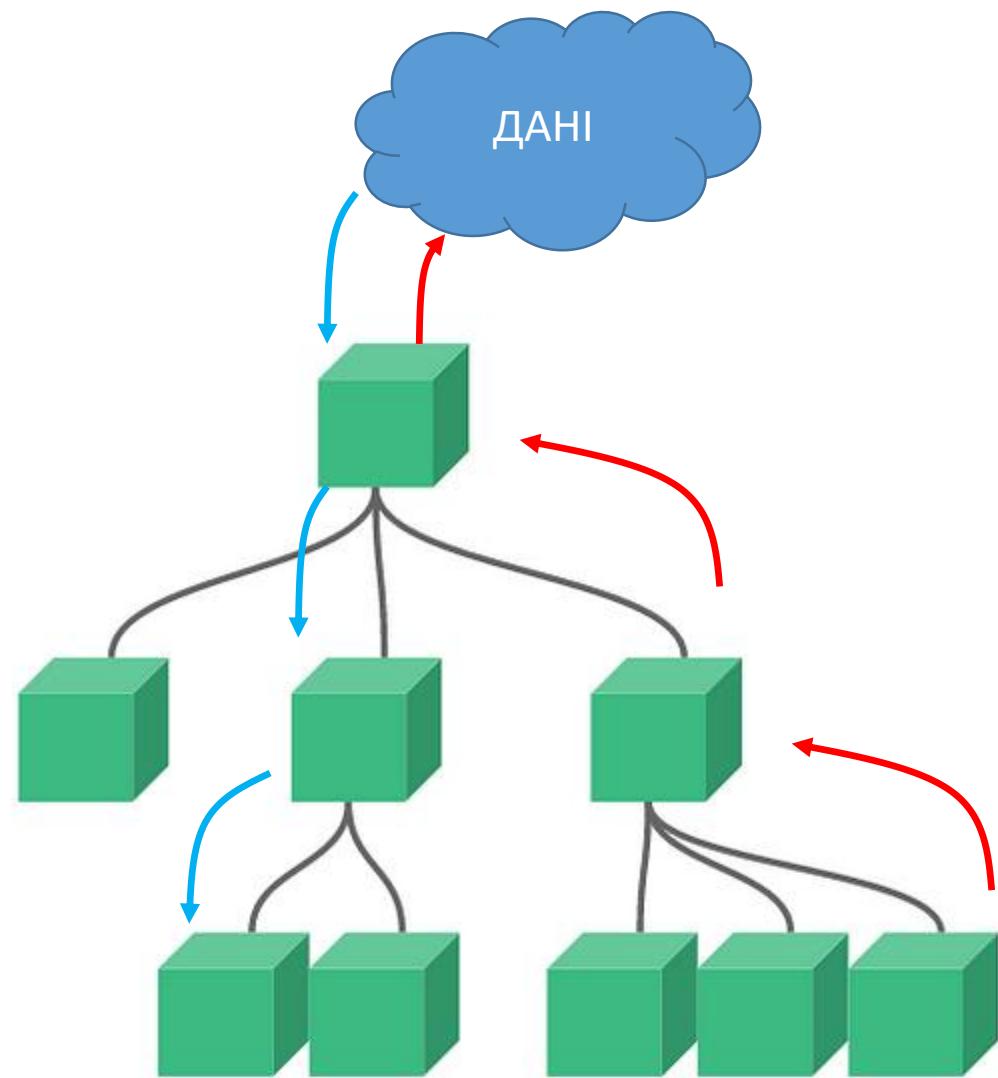
Для чого потрібе Vuex?



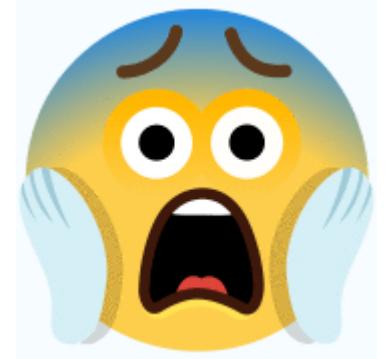
А що буде, якщо глибина
буде ще більшою?



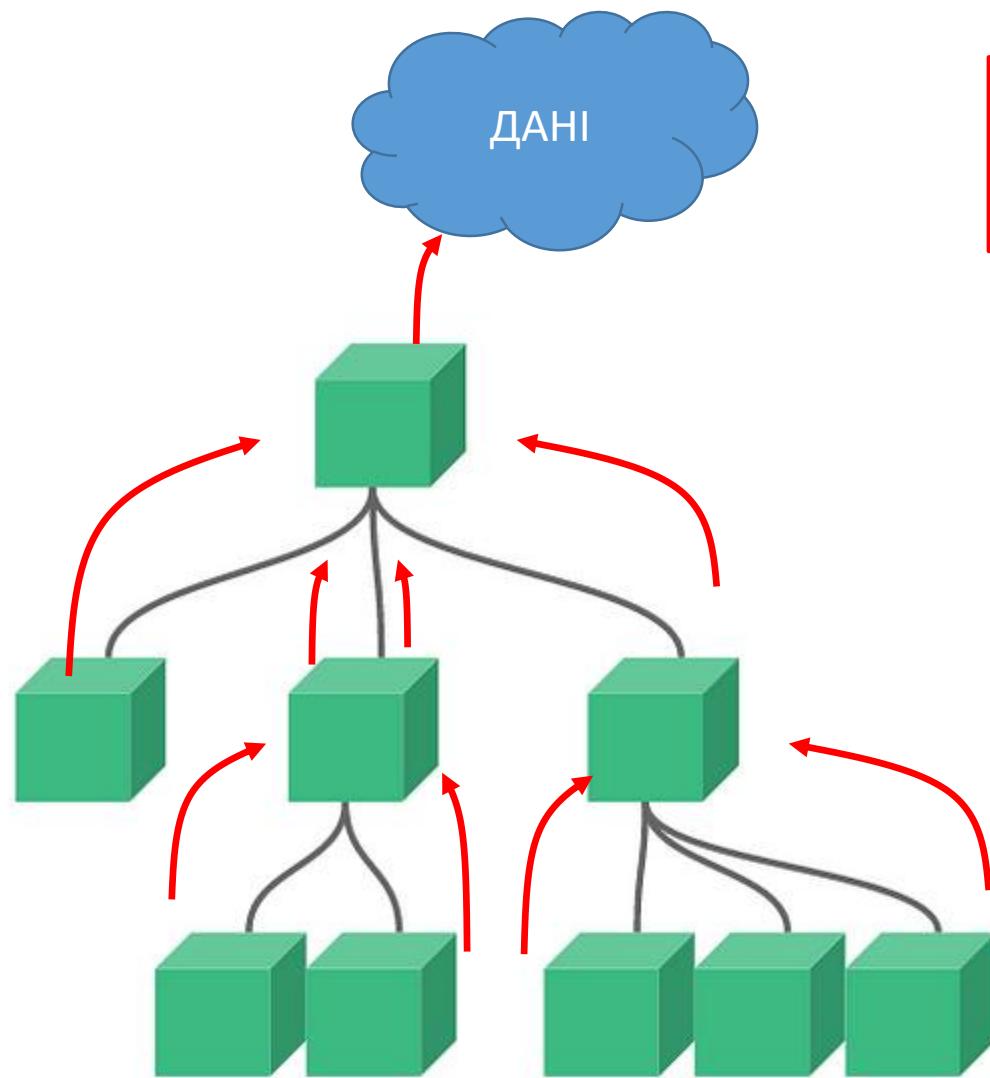
Для чого потрібе Vuex?



Що робити, якщо треба передати дані від одного вкладеного компонента до іншого?



Для чого потрібе VueX?



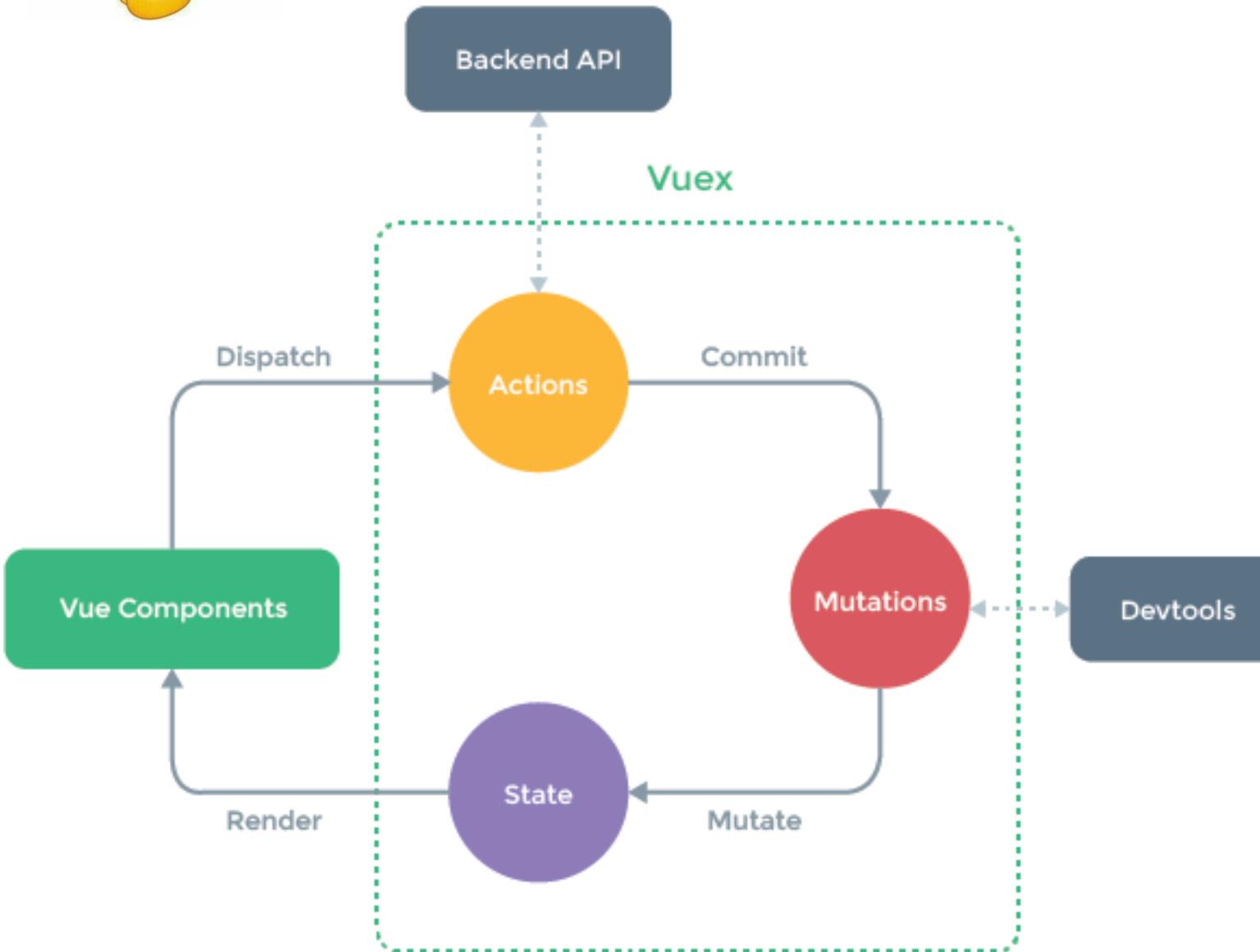
А що буде, якщо дані можуть змінюватись різними компонентами? Як це відслідковувати і контролювати?



Для чого потрібе Vuex?



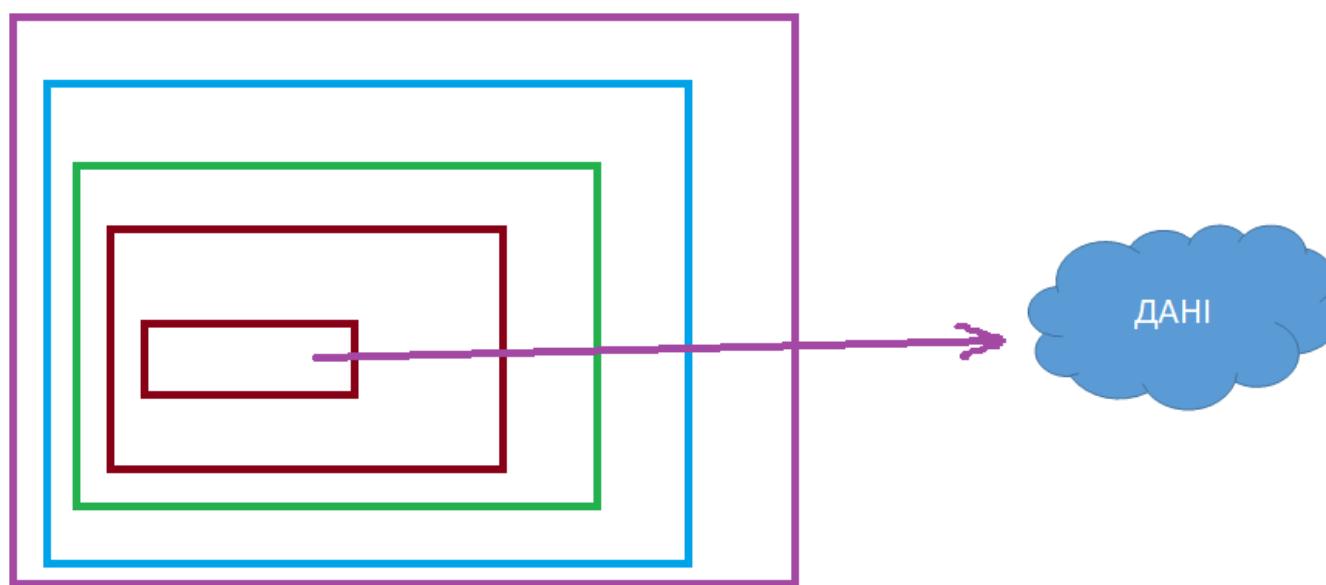
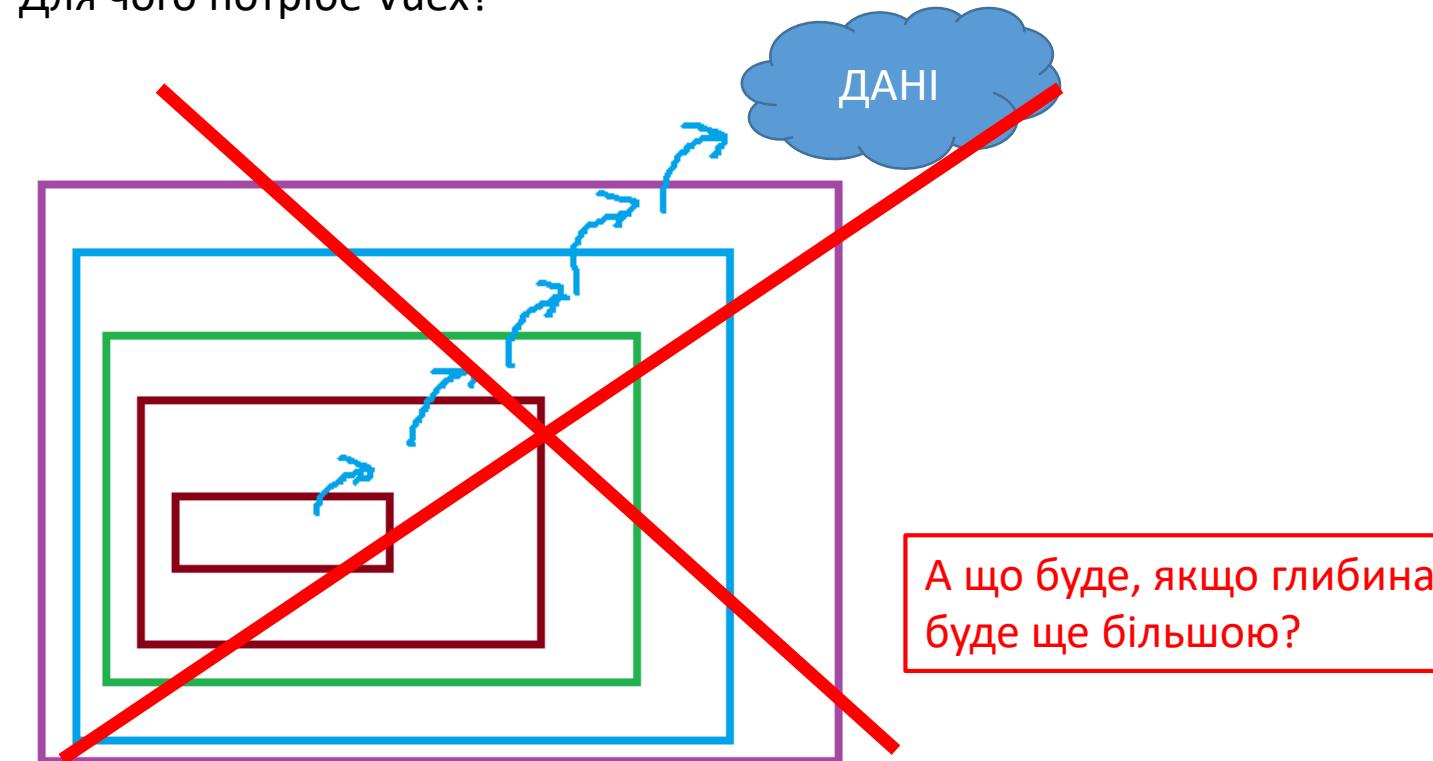
Flux-архітектура – набір шаблонів програмування для побудови інтерфейсу веб-додатків заснований на одоннаправлених потоках даних



Принципи :

1. Існує єдине глобальне сховище **State**
2. Всі компоненти можуть читати значення безпоседньо з сховища **State**
3. Компоненти не можуть безпосередньо змінювати значення у сховищі
4. Для зміни:
 - 1) будь-який компонент формує заявку (**dispatch**) для змін, тобто ініціює **action**
 - 2) **action** ініціює виконання змін у сховищі (**commit**), тобто **mutation**
 - 3) **mutation** виконують зміну у сховищі
5. Компоненти отримують оновлені дані і оновлюються за потреби

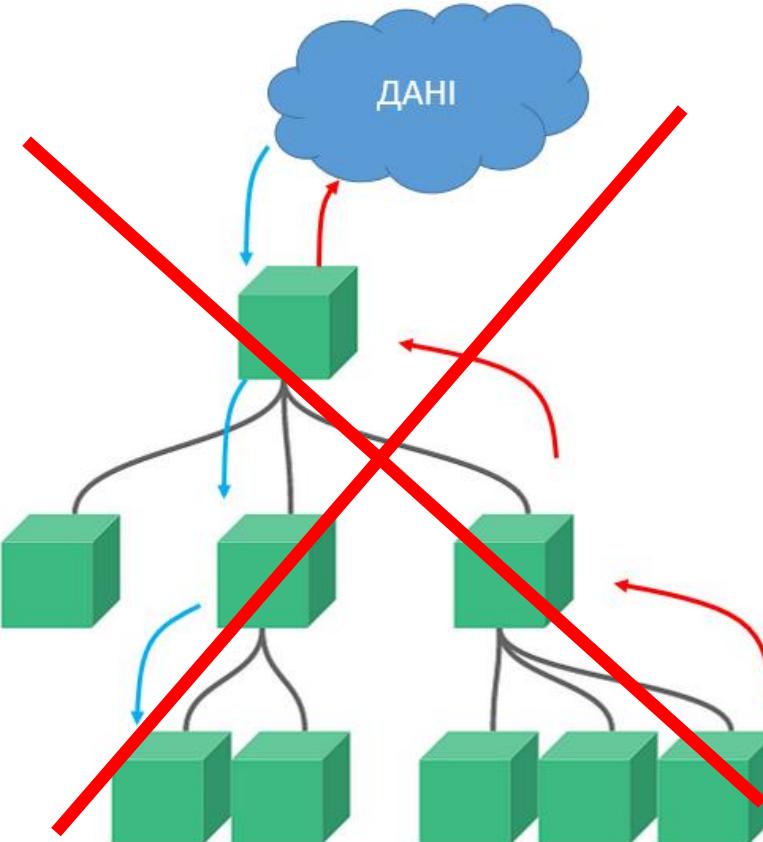
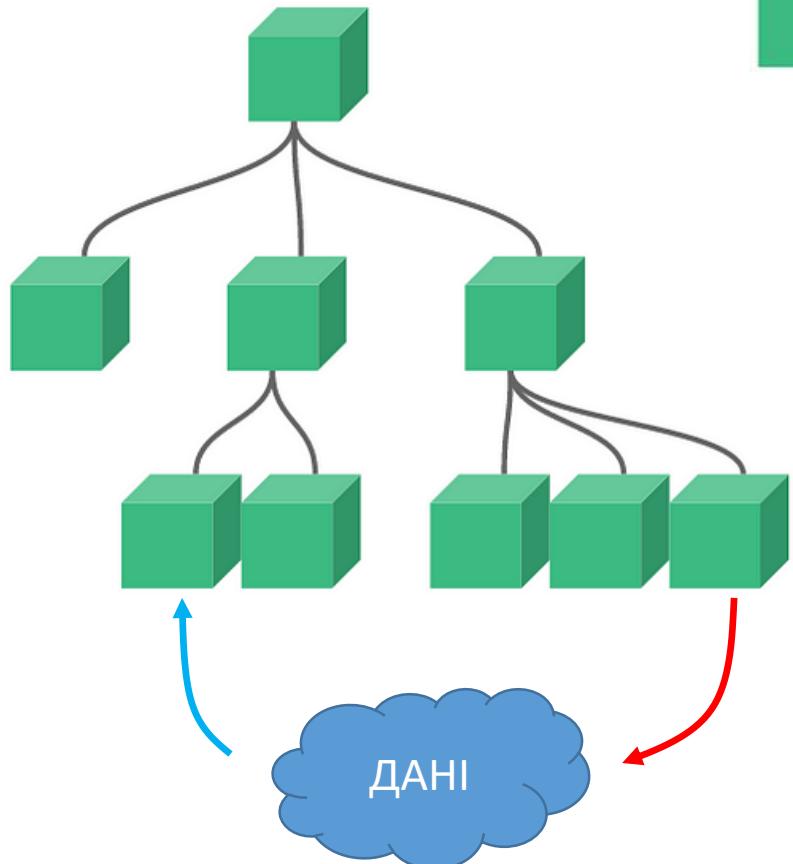
Для чого потрібе VueX?



Принципи :

1. Існує єдине глобальне сховище **State**
2. Всі компоненти можуть читати значення безпосередньо з сховища **State**
3. Компоненти не можуть безпосередньо змінювати значення у сховищі
4. Для зміни:
 - 1) будь-який компонент формує заявку для змін, тобто ініціює **action**
 - 2) **action** ініціює виконання змін у сховищі, тобто **mutation**
 - 3) **mutation** виконують зміну у сховищі
5. Компоненти отримують оновлені дані

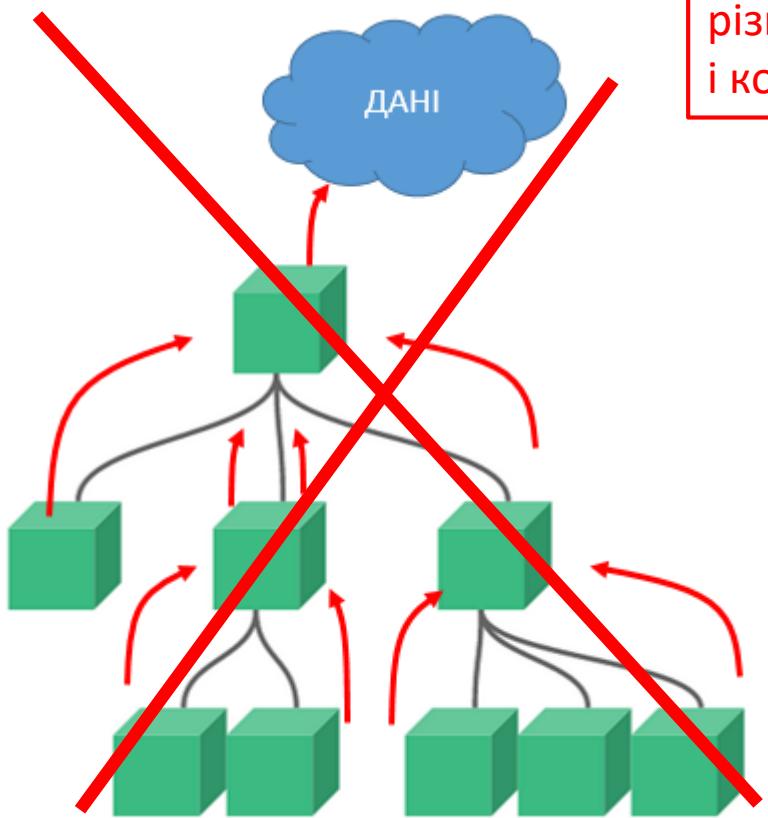
Що робити, якщо треба передати дані від одного вкладеного компонента до іншого?



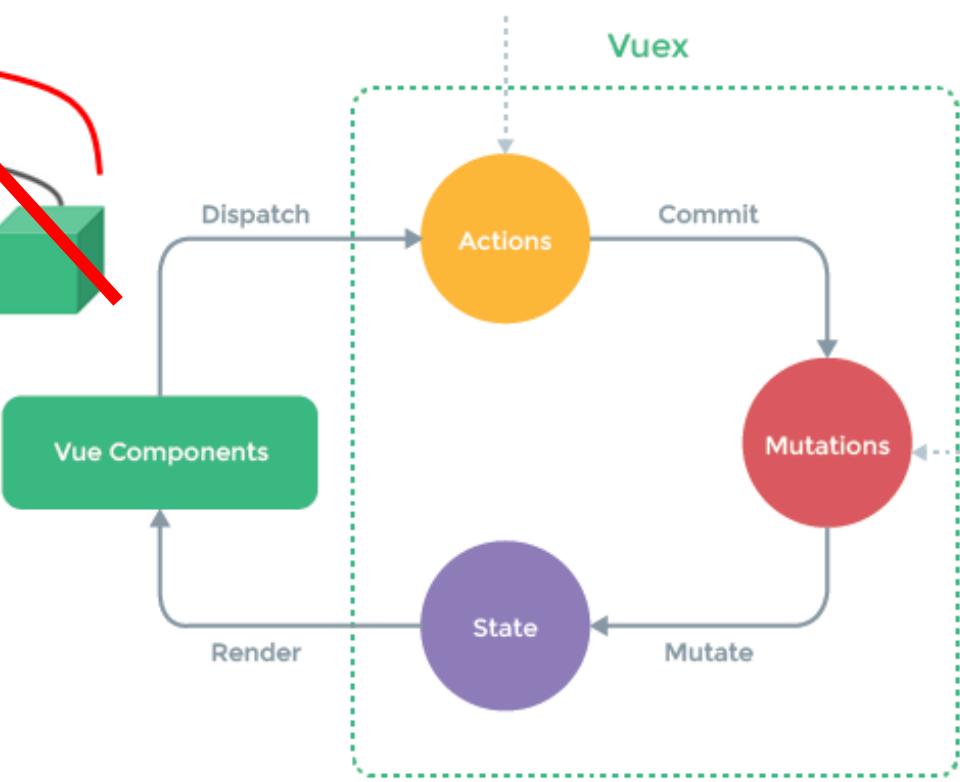
Принципи :

1. Існує єдине глобальне сховище *State*
2. Всі компоненти можуть читати значення безпоседньо з сховища *State*
3. Компоненти не можуть безпосередньо змінювати значення у сховищі
4. Для зміни:
 - 1) будь-який компонент формує заявку для змін, тобто ініціює *action*
 - 2) *action* ініціює виконання змін у сховищі, тобто *mutation*
 - 3) *mutation* виконують зміну у сховищі
5. Компоненти отримують оновлені дані

Для чого потрібе Vuex?



А що буде, якщо дані можуть змінюватись різними компонентами? Як це відслідковувати і контролювати?

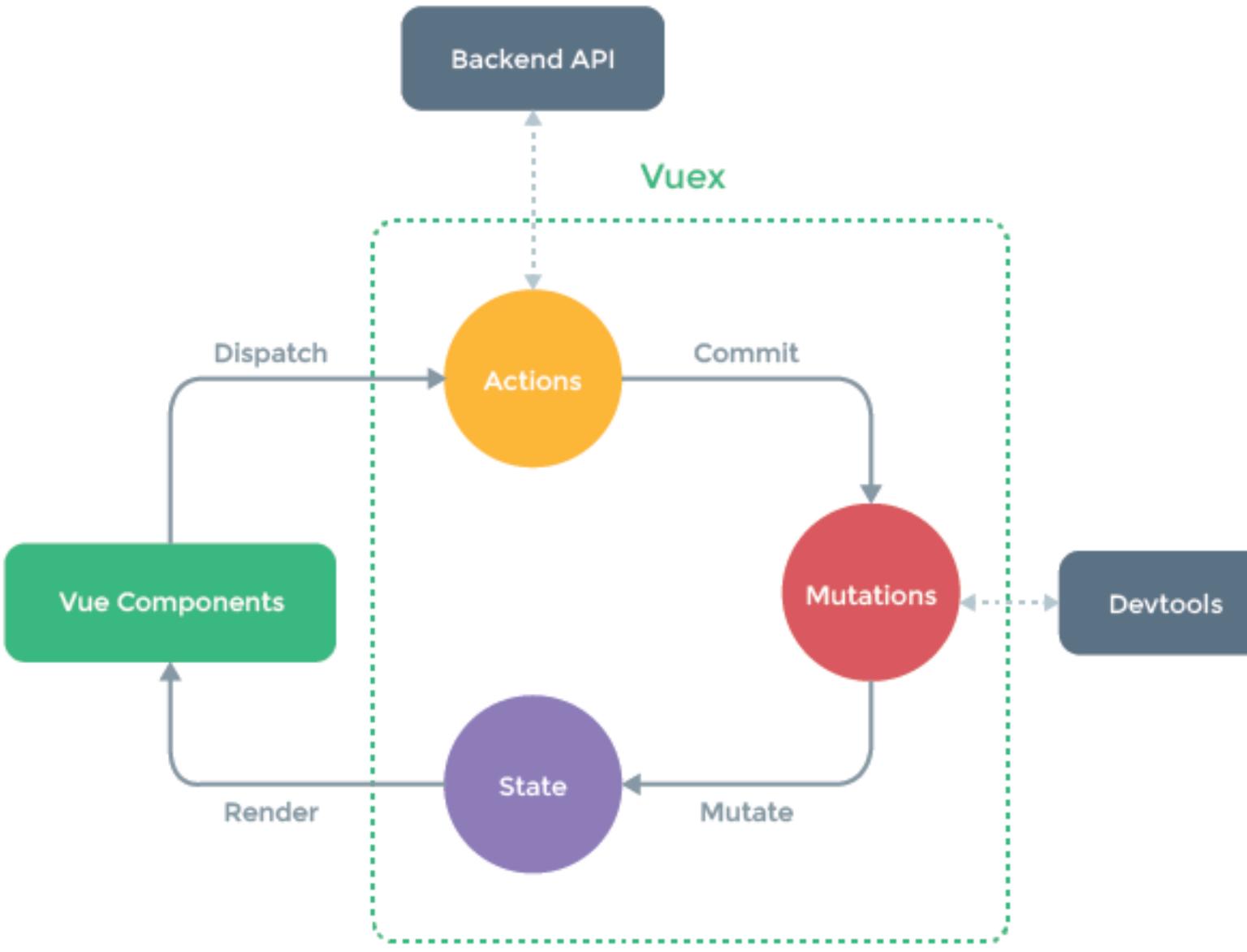


Принципи :

1. Існує єдине глобальне сховище **State**
2. Всі компоненти можуть читати значення безпоседньо з сховища **State**
3. Компоненти не можуть безпосередньо змінювати значення у сховищі
4. Для зміни:
 - 1) будь-який компонент формує заявку для змін, тобто ініціює **action**
 - 2) **action** ініціює виконання змін у сховищі, тобто **mutation**
 - 3) **mutation** виконують зміну у сховищі
5. Компоненти отримують оновлені дані

Для чого потрібе Vuex?

Flux-архітектура – набір шаблонів програмування для побудови інтерфейсу веб-додатків заснований на одонаправлених потоках даних



Принципи :

1. Існує єдине глобальне сховище **State**
2. Всі компоненти можуть читати значення безпосередньо з сховища **State**
3. Компоненти не можуть безпосередньо змінювати значення у сховищі
4. Для зміни:
 - 1) компонент формує заявку для змін, тобто ініціює **action**
 - 2) **action** ініціює виконання змін у сховищі, тобто **mutation**
 - 3) **mutation** виконують зміну у сховищі
5. Компоненти отримують оновлені дані

```
import { createStore } from 'vuex'
```

```
// Create a new store instance.  
const store = createStore({
```

```
)
```

```
import { createStore } from 'vuex'
```

```
// Create a new store instance.  
const store = createStore({
```

```
)
```

----- Опис сховища -----
1. Створюємо об'єкт сховища

```
import { createStore } from 'vuex'  
  
// Create a new store instance.  
const store = createStore({  
  //опис властивостей стану(аналог data)  
  state () {  
    return {  
      властивість1: поч.знач.1,  
      властивість2: поч.знач.2,  
      . . . . .  
    }  
  },  
})
```

```
import { createStore } from 'vuex'  
  
// Create a new store instance.  
const store = createStore({  
  state() {  
    return {  
      count: 0,  
    }  
  },  
})
```

- Опис сховища ----
1. Створюємо об'єкт сховища
 2. Описуємо властивості сховища, які хочемо відслідковувати

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  //опис властивостей стану(аналог data)
  state () {
    return {
      властивість1: поч.знач.1,
      властивість2: поч.знач.2,
      . . . . .
    }
  },
  //опис обчисл.власт.(аналог computed)
  getters: {
    обч.геттер1(state) {
      обчисл. знач. на основі state
    },
    . . . . .
  },
})
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
})
```

- Опис сховища ----
1. Створюємо об'єкт сховища
 2. Описуємо властивості сховища, які хочемо відслідковувати
 3. Опис методів-геттерів (аналог обчислювальних властивостей)

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  //опис властивостей стану(аналог data)
  state () {
    return {
      властивість1: поч.знач.1,
      властивість2: поч.знач.2,
      . . . . .
    }
  },
  //опис обчисл.власт.(аналог computed)
  getters: {
    обч.геттер1(state) {
      обчисл. знач. на основі state
    },
    . . . . .
  },
})
```

Повна форма містить 4 параметри

(state, getters, rootState, rootGetters)

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
})
```

- Опис сховища ----
1. Створюємо об'єкт сховища
 2. Описуємо властивості сховища, які хочемо відслідковувати
 3. Опис методів-геттерів (аналог обчислювальних властивостей)

Приклад. коли треба інші параметри

```
getters: {
  // ...
  геттер (state, getters) {
    використовуємо якийсь інший
    getters. інший_геттер
  }
}
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  //опис властивостей стану(аналог data)
  state () {
    return {
      властивість1: поч.знач.1,
      властивість2: поч.знач.2,
      . . . . .
    }
  },
  //опис обчисл.власт.(аналог computed)
  getters: {
    обч.геттер1(state) {
      обчисл. знач. на основі state
    },
    . . . . .
  },
  //опис методів зміни стану
  mutations: {
    метод_мутація1(state) {
      зміна властивостей state
    }
    . . . . .
  },
})
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
})
```

- Опис сховища ----
1. Створюємо об'єкт сховища
 2. Описуємо властивості сховища, які хочемо відслідковувати
 3. Опис методів-геттерів (аналог обчислювальних властивостей)
 4. Опис мутацій – методів, які можуть змінювати значення у state

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  //опис властивостей стану(аналог data)
  state () {
    return {
      властивість1: поч.знач.1,
      властивість2: поч.знач.2,
      . . . . .
    }
  },
  //опис обчисл.власт.(аналог computed)
  getters: {
    обч.геттер1(state) {
      обчисл. знач. на основі state
    },
    . . . . .
  },
  //опис методів зміни стану
  mutations: {
    метод_мутація1(state) {
      зміна властивостей state
    }
    . . . . .
  },
  //опис методів
  actions: {
    increment (context) {
      context.commit('increment')
    }
    . . . . .
  }
})
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  },
})
```

- Опис сховища ----
1. Створюємо об'єкт сховища
 2. Описуємо властивості сховища, які хочемо відслідковувати
 3. Опис методів-геттерів (аналог обчислювальних властивостей)
 4. Опис мутацій – методів, які можуть змінювати значення у state
 5. Опис action-методів – методів, які викликаються з компонентів для зміни значення сховища (не безпосередньо, а з використанням мутацій, які викликаються з використанням метода **commit ('назва мутації')**)

Доступ до сховища у компонентах. Пряме звертання до сховища з використанням this.\$store

this . \$store



```
▼ Store ⓘ
  ► commit: f boundCommit(type, payload, options)
  ► dispatch: f boundDispatch(type, payload)
  ► getters: {}
  ▼ state: Proxy(Object)
    ► [[Handler]]: MutableReactiveHandler
    ▼ [[Target]]: Object
      count: 0
      ► [[Prototype]]: Object
      [[IsRevoked]]: false
      ► [[Prototype]]: Object
```

Доступ до сховища у компонентах. Пряме звертання до сховища з використанням this.\$store

this . \$store



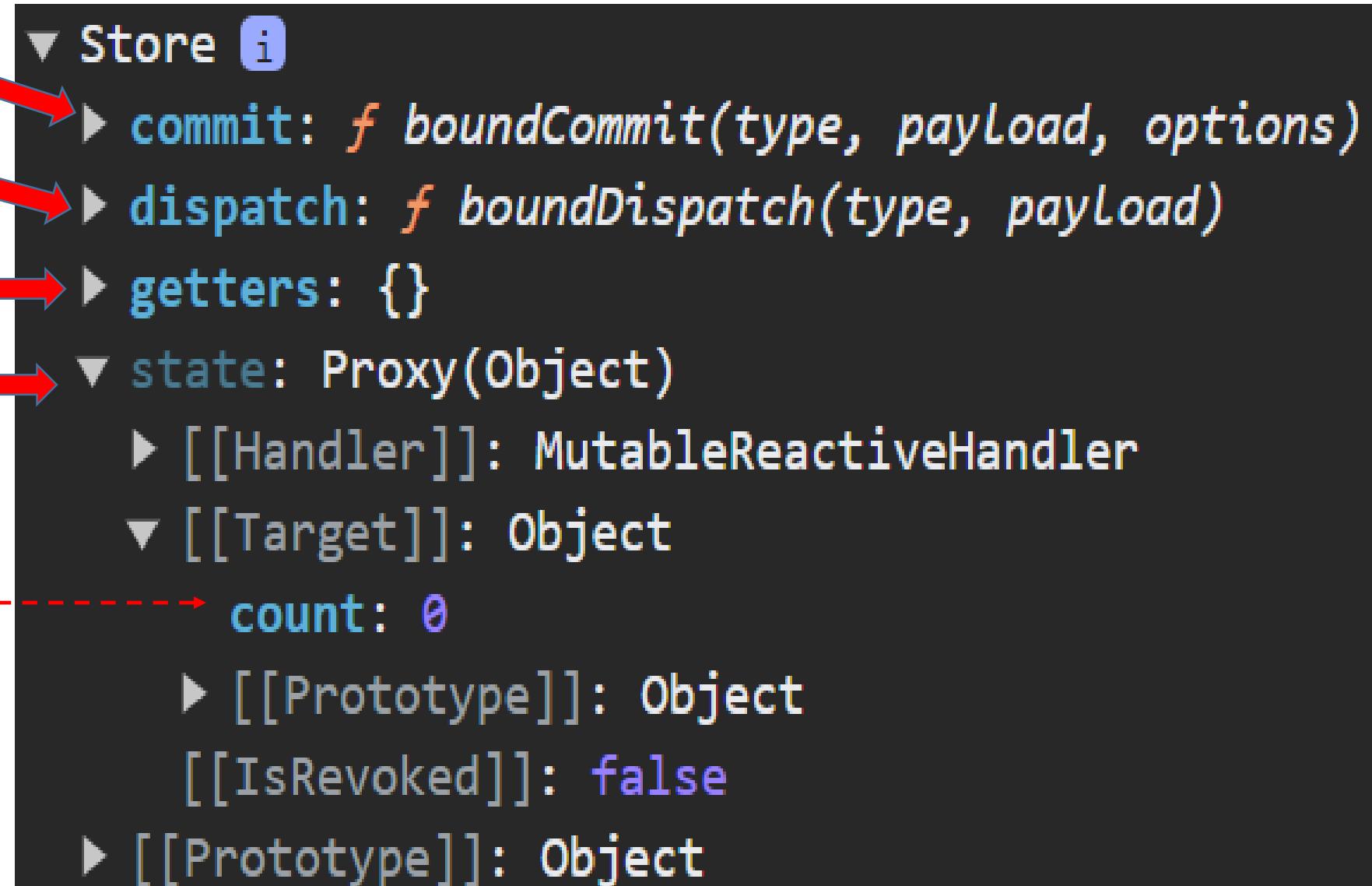
Для виклику мутацій

Для виклику action-методів

Для звертання до
геттера

Для прямого
звертання до
властивостей
state

this.\$store.state.count



Пряме зчитування властивостей з сховища (state)

Загальна форма

this.\$store.state. властивість

Фрагмент компонента

```
<template>
  <div>{{ $store.state.count }}</div>
</template>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Сховище

Пряме звертання до **геттерів** з сховища (getters)

Загальна форма

this.\$store.getters.**геттер**

Фрагмент компонента

```
<template>
  ...<div>{{ $store.getters.doubleCount }}</div>
</template>
```

Сховище

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Передача данных у геттери

Загальна форма

this.\$store.getters.**геттер(параметри)**

Сховище

Фрагмент компонента

```
<template>
  ...
  <div>{{ $store.getters.getTodoById(2) }}</div>
</template>
```

```
getters: {
  // ...
  getTodoById: (state) => (id) => {
    return state.todos.find(todo => todo.id === id)
  }
}
```

Пряме звертання до **геттерів** з сховища (getters)

Загальна форма

Фрагмент компонента

```
<template>
  ...
  <div>
    <button @click="$store.dispatch('incrementCount')">
      Add
    </button>
    <button @click="$store.dispatch('decrementCount')">
      Subtract
    </button>
  </div>
</template>
```

this.\$store.dispatch (*'action-метод', параметр*)

Сховище

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Приклад. Зробити менеджер задач з використанням Vue.js

Додати задачу

Задача

Пріоритет

Список задач

- Випити каву - 100
- Випити ще одну каву - 120
- Перевірити пошту - 5
- Зробити зарядку - 30

Використання гелперів для *інтеграції* компонентів сховища у компонети

Для інтеграції (вставки) значень з state



mapState

Для інтеграції (вставки) геттерів



mapGetters

Для інтеграції (вставки) мутацій-методів



mapMutations

Для інтеграції (вставки) action-методів



mapActions

Зчитування значень з сховища у компонентах (state)

Для інтеграції (вставки) значень з state



mapState

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зчитування значень з сховища у компонентах (state)

----- Загальна схема -----

```
<template>
</template>
<script>
//1. Імпортуємо mapState
import { mapState } from 'vuex'
</script>
}
```

----- Приклад -----

```
<template>
</template>
<script>
import { mapState } from 'vuex'

export default {
  name: 'App',
}
</script>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зчитування значень з сховища у компонентах (state)

----- Загальна схема -----

```

<template>
    ...
</template>

<script>
//1. Імпортуємо mapState
import { mapState } from 'vuex'

export default {
    ...
    //2. Додаємо властивості
    computed: {
        ...mapState(['властивість1',
                    'властивість2'], ...),
    },
    ...
}
</script>

```

----- Приклад -----

```

<template>
    ...
</template>

<script>
import { mapState } from 'vuex'

export default {
    name: 'App',
    computed: {
        ...mapState(['count']),
    },
}
</script>

```

```

import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
    state() {
        return {
            count: 0,
        }
    },
    getters: {
        // getCount: (state) => state.count,
        getCount: ({ count }) => count,
        doubleCount(state) {
            return state.count * 2
        },
    },
    mutations: {
        increment(state) {
            state.count++
        },
        decrement(state) {
            state.count--
        },
    },
    actions: {
        incrementCount(context) {
            context.commit('increment')
        },
        decrementCount({ commit }) {
            commit('decrement')
        },
    }
})
export default store

```

Зчитування значень з сховища у компонентах (state)

----- Загальна схема -----

```

<template>
//3. Використовуємо властивості як
звичайні обчислювальні властивості
    . . . . .
    <тег>{{ властивість1 }}</тег >
    . . . . .
    <тег :атрибут = “властивість2” />
    . . . . .
</template>

<script>
//1. Імпортуємо mapState
import { mapState } from 'vuex'

export default {
    . . . . .

//2. Додаємо властивості
    computed: {
        ...mapState(['властивість1',

        властивість2, ...),
    },
    . . . . .
}
</script>

```

----- Приклад -----

```

<template>
    . . . <div>{{ count }}</div>
</template>

<script>
import { mapState } from 'vuex'

export default {
    name: 'App',
    computed: {
        ...mapState(['count']),
    },
}
</script>

```

```

import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
    state() {
        return {
            count: 0,
        }
    },
    getters: {
        // getCount: (state) => state.count,
        getCount: ({ count }) => count,
        doubleCount(state) {
            return state.count * 2
        },
    },
    mutations: {
        increment(state) {
            state.count++
        },
        decrement(state) {
            state.count--
        },
    },
    actions: {
        incrementCount(context) {
            context.commit('increment')
        },
        decrementCount({ commit }) {
            commit('decrement')
        },
    }
})
export default store

```

Зчитування значень з сховища у компонентах. Використ. геттерів у компонентах.

Для інтеграції (вставки) геттерів



mapGetters

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зчитування значень з сховища у компонентах. Використ. геттерів у компонентах.

----- Загальна схема -----

```
<template>
```

```
</template>
```

```
<script>
```

//1. Імпортуємо *mapGetters*

```
import { mapGetters } from 'vuex'
```

```
export default {
```

```
</script>
```

```
<style lang="scss"></style>
```

----- Приклад -----

```
<template>
```

```
</template>
```

```
<script>
```

```
import { mapGetters } from 'vuex'
```

```
export default {
```

```
  name: 'App',
```

```
}
```

```
</script>
```

```
<style lang="scss"></style>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зчитування значень з сховища у компонентах. Використанням геттерів у компонентах.

- - - - Загальна схема - - - -

```
<template>

</template>

<script>
//1. Імпортуємо mapGetters
import { mapGetters } from 'vuex'

export default {
    . . . . . . . . .

//2. Підключаємо геттери
    computed: {
        ...mapGetters(['назва']),
    },
    . . . . . . . . .

}
</script>

<style lang="scss"></style>
```

----- *Приклад* -----

```
<template>
</template>

<script>
import { mapGetters } from 'vuex'

export default {
  name: 'App',
  computed: {
    ...mapGetters(['getCount']),
  },
}
</script>

<style lang="scss"></style>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  },
})
export default store
```

Зчитування значень з сховища у компонентах. Використанням геттерів у компонентах.

----- Загальна схема -----

```
<template>
//3. Використовуємо геттери як
звичайну обчислювальну властивість
    . . . . .
<тег>{{ геттер }}</тег >
    . . . . .
<тег :атрибут = “геттер” />
    . . . . .

</template>

<script>
//1. Імпортуюмо mapGetters
import { mapGetters } from 'vuex'

export default {
    . . . . .

//2. Підключаємо геттери
    computed: {
        ...mapGetters(['назва']),
    },
    . . . . .
}
</script>

<style lang="scss"></style>
```

----- Приклад -----

```
<template>
    . . . <div>{{ getCount }}</div>
</template>

<script>
import { mapGetters } from 'vuex'

export default {
    name: 'App',
    computed: {
        ...mapGetters(['getCount']),
    },
    . . . .
}

</script>

<style lang="scss"></style>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
    state() {
        return {
            count: 0,
        }
    },
    getters: {
        // getCount: (state) => state.count,
        getCount: ({ count }) => count,
        doubleCount(state) {
            return state.count * 2
        },
    },
    mutations: {
        increment(state) {
            state.count++
        },
        decrement(state) {
            state.count--
        },
    },
    actions: {
        incrementCount(context) {
            context.commit('increment')
        },
        decrementCount({ commit }) {
            commit('decrement')
        },
    }
})
export default store
```

Зміна значень у сховищі з компонентів. Використанням mutation-методів у компонентах.

Для інтеграції (вставки) мутацій-методів



mapMutations

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зміна значень у сховищі з компонентів. Використанням action-методів у компонентах.

- - - - - Загальна схема - - - - -

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

<script>
//1. Імпортуємо mapMutations
import { mapMutations } from 'vuex'
  ...
</script>
</script>
```

- - - - - Приклад - - - - -

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

<script>
import { mapMutations } from 'vuex'

export default {
  name: 'App',
  ...
}
</script>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зміна значень у сховищі з компонентів. Використанням action-методів у компонентах.

----- Загальна схема -----

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

<script>
//1. Імпортуємо mapMutations
import { mapMutations } from 'vuex'

export default {
  ...
  //2. Підключаємо методи-мутації
  methods: {
    ...mapMutations(['назва']),
  },
  ...
}
</script>
```

----- Приклад -----

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

<script>
import { mapMutations } from 'vuex'

export default {
  name: 'App',
  methods: {
    ...mapMutations(['increment', 'decrement']),
  },
</script>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зміна значень у сховищі з компонентів. Використанням action-методів у компонентах.

----- Загальна схема -----

```
<template>
//3. Використовуємо методи-мутації
як звичайні методи
    . . . . .
    <тег @подія="методи-мутації" />
    . . . . .

</template>

<script>
//1. Імпортуюмо mapMutations
import { mapMutations } from 'vuex'

export default {
    . . . . .
    . . . . .

//2. Підключаємо методи-мутації
methods: {
    ...mapMutations(['назва']),
},
    . . . . .
}
</script>
```

----- Приклад -----

```
<template>
<div>{{ $store.state.count }}</div>
<div>
    <button @click="increment">
        Add
    </button>
    <button @click="decrement">
        Subtract
    </button>
</div>
</template>

<script>
import { mapMutations } from 'vuex'

export default {
    name: 'App',
    methods: {
        ...mapMutations(['increment', 'decrement']),
    }
</script>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
    state() {
        return {
            count: 0,
        }
    },
    getters: {
        // getCount: (state) => state.count,
        getCount: ({ count }) => count,
    },
    mutations: {
        increment(state) {
            state.count++
        },
        decrement(state) {
            state.count--
        },
    },
    actions: {
        incrementCount(context) {
            context.commit('increment')
        },
        decrementCount({ commit }) {
            commit('decrement')
        },
    }
})
export default store
```

Зміна значень у сховищі з компонентів. Використанням action-методів у компонентах.

Для інтеграції (вставки) action-методів



mapActions

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зміна значень у сховищі з компонентів. Використанням action-методів у компонентах.

```
<template>
```

```
</template>
```

```
<script>
```

```
//1. Імпортуємо mapActions
```

```
import { mapActions } from 'vuex'
```

```
export default {
```

```
</script>
```

----- Приклад -----

```
<template>
  <div>{{ getCount }}</div>
</template>

<script>
  import { mapGetters, mapActions } from 'vuex'

  export default {
    name: 'App',

    computed: {
      ...mapGetters(['getCount']),
    },
  }
</script>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зміна значень у сховищі з компонентів. Використанням action-методів у компонентах.

```
<template>
```

```
</template>
```

```
<script>  
//1. Імпортуємо mapActions
```

```
import { mapActions } from 'vuex'
```

```
export default {
```

```
//2. Підключаємо action-методи
```

```
methods: {  
  ... mapActions(['action-метод']),  
},
```

```
}</script>
```

----- Приклад -----

```
<template>  
  <div>{{ getCount }}</div>  
</template>  
  
<script>  
  import { mapGetters, mapActions } from 'vuex'  
  
  export default {  
    name: 'App',  
  
    computed: {  
      ...mapGetters(['getCount']),  
    },  
  
    methods: {  
      ...mapActions(['incrementCount',  
        'decrementCount']),  
    },  
  }</script>
```

```
import { createStore } from 'vuex'  
  
// Create a new store instance.  
const store = createStore({  
  state() {  
    return {  
      count: 0,  
    }  
  },  
  getters: {  
    // getCount: (state) => state.count,  
    getCount: ({ count }) => count,  
  
    doubleCount(state) {  
      return state.count * 2  
    },  
  },  
  mutations: {  
    increment(state) {  
      state.count++  
    },  
    decrement(state) {  
      state.count--  
    },  
  },  
  actions: {  
    incrementCount(context) {  
      context.commit('increment')  
    },  
    decrementCount({ commit }) {  
      commit('decrement')  
    },  
  }  
}  
export default store
```

Зміна значень у сховищі з компонентів. Використанням action-методів у компонентах.

```
<template>
//3. Використовуємо action-методи
як звичайні методи
    . . . . .
    <тег @подія="action-метод" />
    . . . . .
</template>

<script>
//1. Імпортуємо mapActions
import { mapActions } from 'vuex'

export default {
    . . . . .
    . . . . .
    //2. Підключаємо action-методи
    methods: {
        ... mapActions(['action-метод']),
    },
    . . . . .
}
</script>
```

Приклад

```
<template>
    <div>{{ getCount }}</div>
    <div>
        <button @click="incrementCount">
            Add
        </button>
        <button @click="decrementCount">
            Subtract
        </button>
    </div>
</template>

<script>
import { mapGetters, mapActions } from 'vuex'

export default {
    name: 'App',
    computed: {
        ...mapGetters(['getCount']),
    },
    methods: {
        ...mapActions(['incrementCount',
            'decrementCount']),
    }
}
</script>
```

```
import { createStore } from 'vuex'

// Create a new store instance.
const store = createStore({
    state() {
        return {
            count: 0,
        }
    },
    getters: {
        // getCount: (state) => state.count,
        getCount: ({ count }) => count,
    },
    mutations: {
        increment(state) {
            state.count++
        },
        decrement(state) {
            state.count--
        },
    },
    actions: {
        incrementCount(context) {
            context.commit('increment')
        },
        decrementCount({ commit }) {
            commit('decrement')
        },
    }
})
export default store
```

Приклад. Зробити менеджер задач з використанням Vue.js та гелперів

Додати задачу

Задача

Пріоритет

Список задач

Випити каву - 100

Випити ще одну каву - 120

Перевірити пошту - 5

Зробити зарядку - 30

Список студентів**Система оцінювання****Категорія**

| ПІБ | Середній бал |
|-----------------|--------------|
| Коваль Тарас | 10 |
| Григоренко Іван | 9 |
| Маркович Оля | 11 |
| Василеко Іра | 12 |
| Пилип Тарас | 7 |
| Баба Галя | 35 |
| Дід Степан | 21 |

| |
|----|
| 12 |
| 5 |

| |
|-----------|
| Відмінник |
| Хорошист |
| Трійочник |
| Двійочник |
| Блатник |

Приклад. Розробити з використанням Vue.js

Віємо на нашому сайті

Список товарів

| | | |
|---------|----------|---------------------------------------|
| Хліб | 24 грн. | <input type="button" value="Купити"/> |
| Молоко | 31 грн. | <input type="button" value="Купити"/> |
| Ковбаса | 350 грн. | <input type="button" value="Купити"/> |
| Шовдарь | 950 грн. | <input type="button" value="Купити"/> |
| Пікніця | 210 грн. | <input type="button" value="Купити"/> |

Корзина

| | | |
|--------------------|-------------|--|
| Вода | 15 грн. | <input type="button" value="Відмовитись"/> |
| Ноутбук | 529999 грн. | <input type="button" value="Відмовитись"/> |
| Комплект моніторів | 25000 грн. | <input type="button" value="Відмовитись"/> |
| Інвертор | 13000 грн. | <input type="button" value="Відмовитись"/> |
| Акумулятор | 11000 грн. | <input type="button" value="Відмовитись"/> |

Разом до оплати

102 014 грн.

Виберіть валюту

Гривня
Доллар

Ще додаткові слайди за аналогічними схемами

Використання геттерів

4. Використовуємо
підключенні геттери
як звичайні властивості

2. Імпортуюмо
mapGetters

3. Вказуємо список
геттерів, які потрібно
використати у цьому
компоненті

1. Описуємо getters

```
File Edit Selection View Go Run Terminal Help
index.js - ex5 - Visual Studio Code
EXPLORER OPEN EDITORS 1 UNSAVED
App.vue M JS index.js U
test-app > src > App.vue > {} "App.vue" > script
You, seconds ago | 1 author (You)
1 <template>
2   <div>{{ getCount }}</div>
3 </template>
4
5
6 <script>
7   import { mapGetters } from 'vuex'
8
9   export default {
10     name: 'App',
11     components: {
12       HelloWorld,
13     },
14     computed: {
15       ...mapGetters(['getCount']),
16     },
17   }
18 </script>
19
20 <style>
21 #app {
22   font-family: Avenir, Helvetica, Arial, sans-serif;
23   -webkit-font-smoothing: antialiased;
24 }
```

```
JS index.js U X JS main.js M
test-app > src > store > JS index.js > [e] store
1 import { createStore } from 'vuex'
2
3 // Create a new store instance.
4 const store = createStore({
5   state() {
6     return {
7       count: 0,
8     }
9   },
10   getters: {
11     getCount: (state) => {
12       return state.count
13     },
14   },
15
16   mutations: {
17     increment(state) {
18       state.count++
19     },
20   },
21 })
22
23 export default store
24 }
```

App.vue

```
<template>
  <div>{{ getCount }}</div>
  <div>
    <button @click="incrementCount">Add</button>
    <button @click="decrementCount">Subtract</button>
  </div>
</template>

<script>
import { mapGetters, mapActions } from 'vuex'

export default {
  name: 'App',
  computed: {
    ...mapGetters(['getCount']),
  },
  methods: {
    ...mapActions(['incrementCount', 'decrementCount']),
  }
}
</script>

<style lang="scss"></style>
```

index.js

```
// Create a new store instance.
const store = createStore({
  state() {
    return {
      count: 0,
    }
  },
  getters: {
    // getCount: (state) => state.count,
    getCount: ({ count }) => count,
    doubleCount(state) {
      return state.count * 2
    },
  },
  mutations: {
    increment(state) {
      state.count++
    },
    decrement(state) {
      state.count--
    },
  },
  actions: {
    incrementCount(context) {
      context.commit('increment')
    },
    decrementCount({ commit }) {
      commit('decrement')
    },
  }
})
export default store
```

Зміна значень у сховищі state.
Використання action-методів

1. Описуємо мутацію

2. Описуємо action-метод

3. Імпортуємо mapActions

4. З використанням mapActions підключаємо action-методи

5. Використовуємо action-методи

