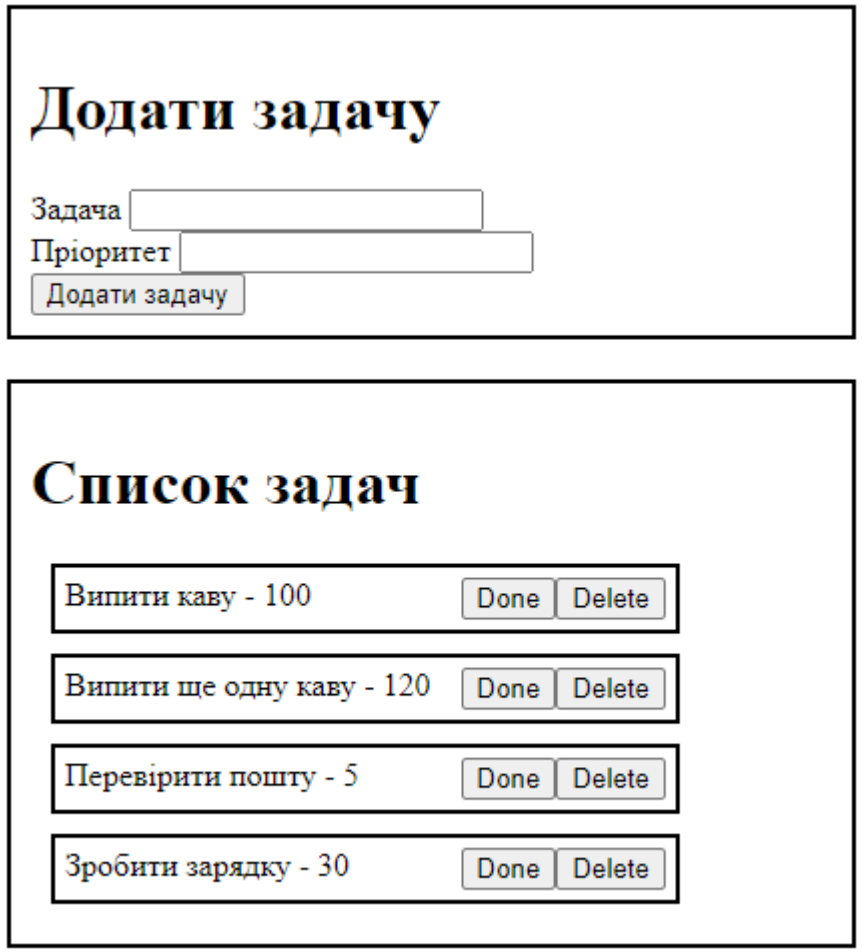
Vuex

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/

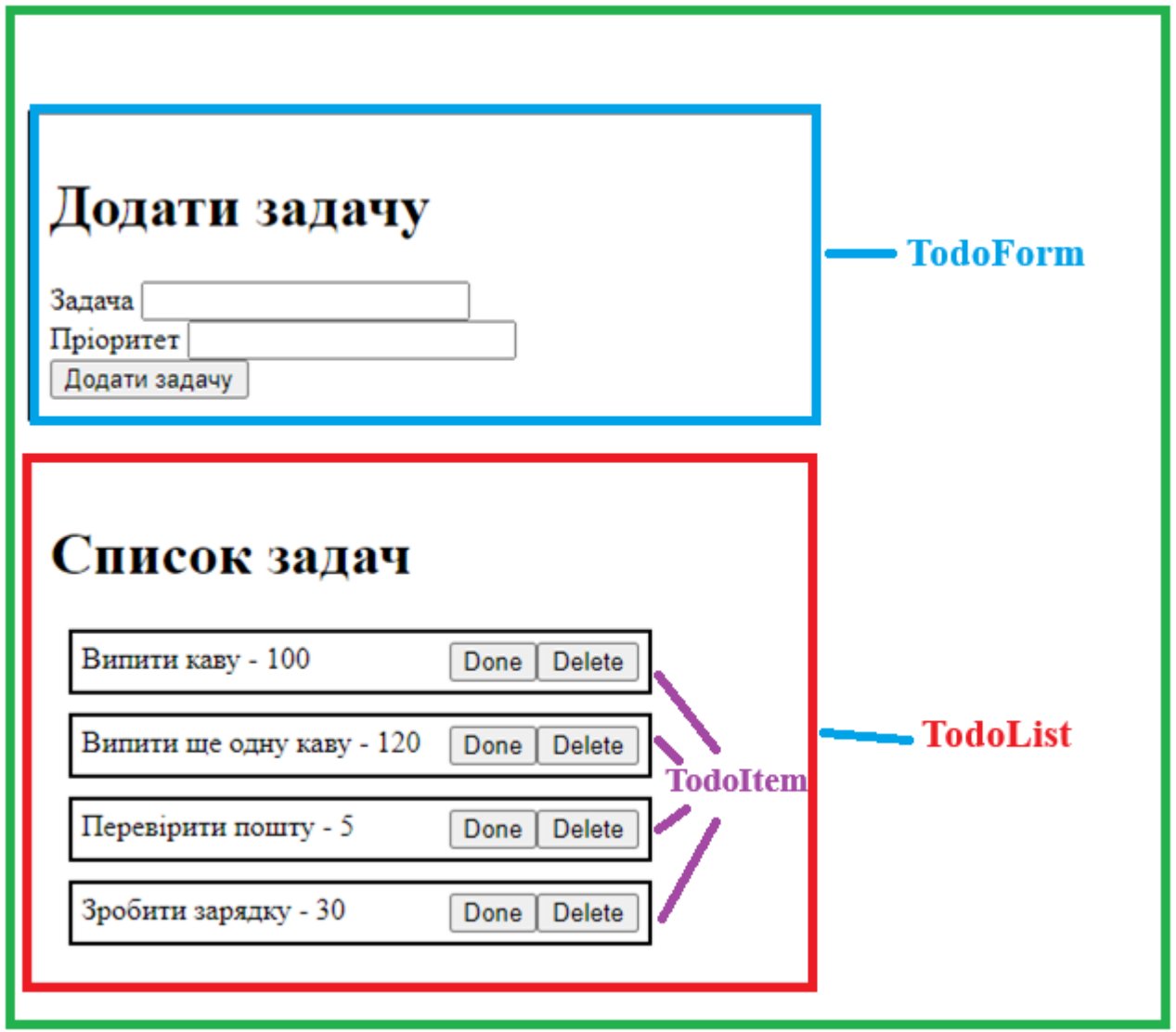
Для чого потрібе Vuex?

Розглянемо меджер задач



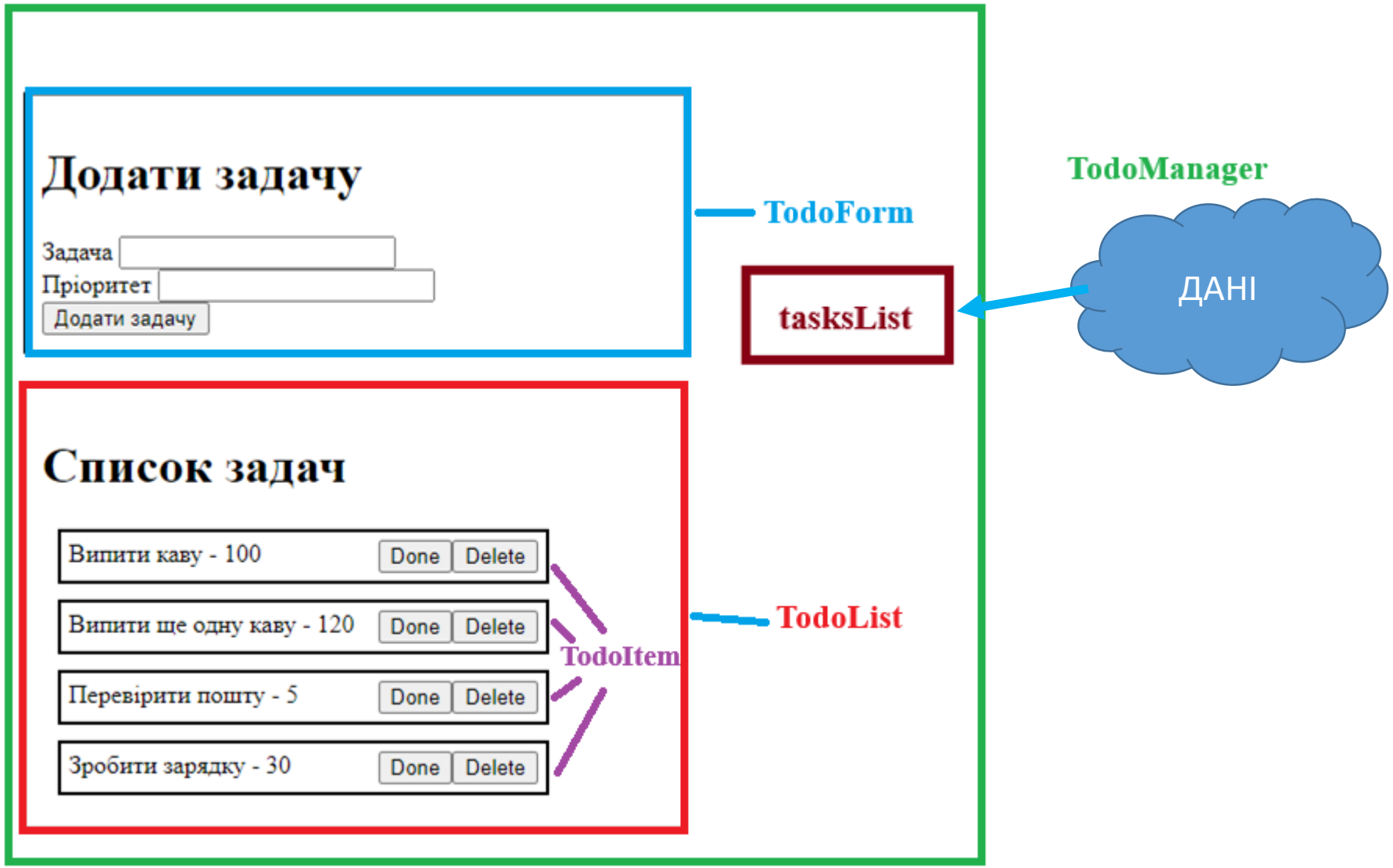
Для чого потрібе Vuex?

Розглянемо меджер задач



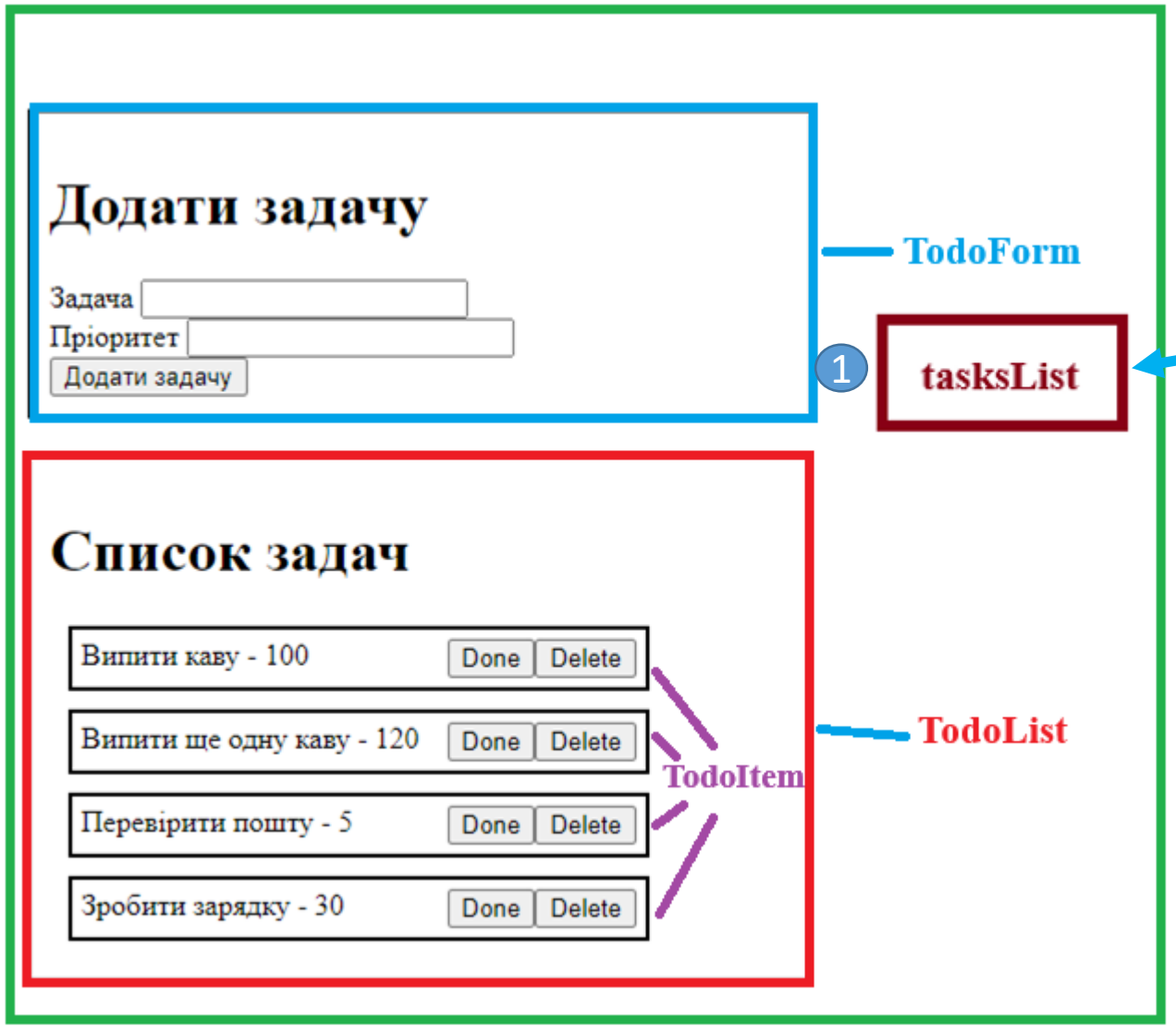
Для чого потрібе Vuex?

Розглянемо меджер задач



Для чого потрібе Vuex?

Розглянемо меджер задач

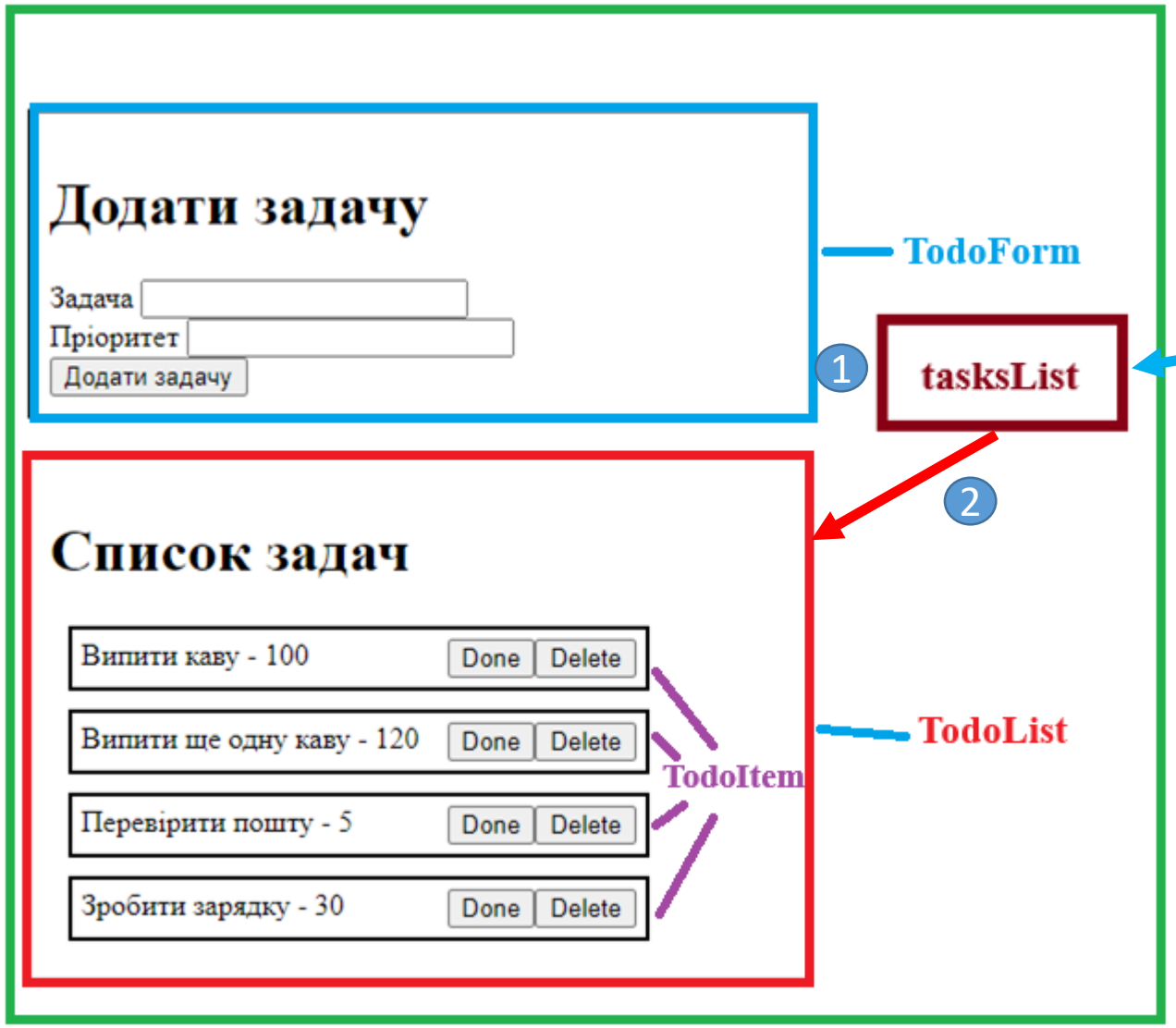


Як передати дані у TodoItem?

1) Масив tasksList міститься у TodoManager

Для чого потрібе Vuex?

Розглянемо меджер задач



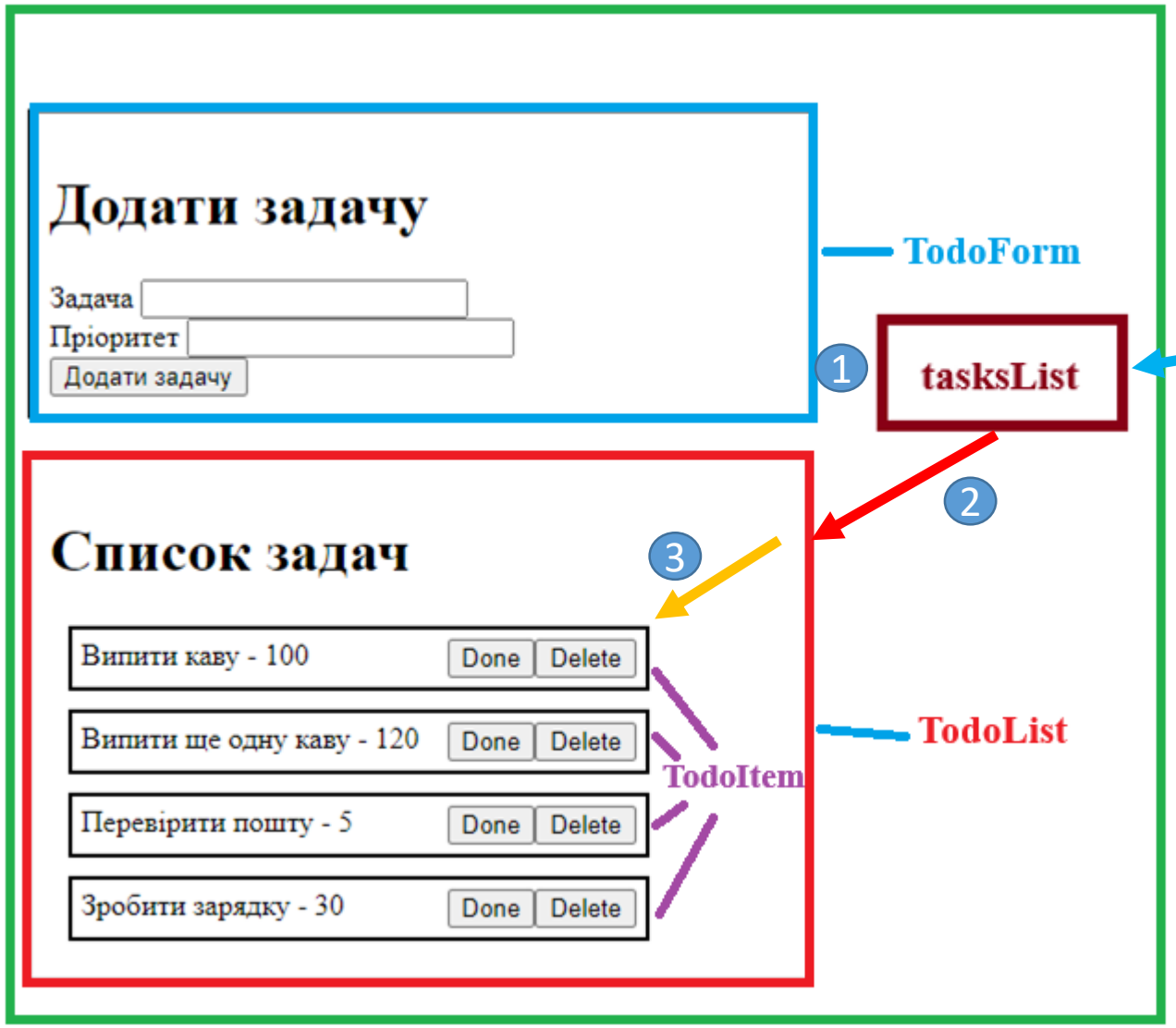
Як передати дані у TodoItem?

1) Масив tasksList міститься у TodoManager

2) TodoManager передає список задач у TodoList

Для чого потрібе Vuex?

Розглянемо меджер задач



Як передати дані у TodoItem?

1) Масив tasksList міститься у TodoManager
2) TodoManager передає список задач у TodoList
3) TodoList відображає елементи передаючи дані у дочірні елементи TodoItem

TodoManager.vue components U

components > TodoManager.vue > { } "TodoManager.vue" > scrip

```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>
```

1

TodoList.vue components U

src > components > TodoList.vue > { } "TodoList.vue" > template

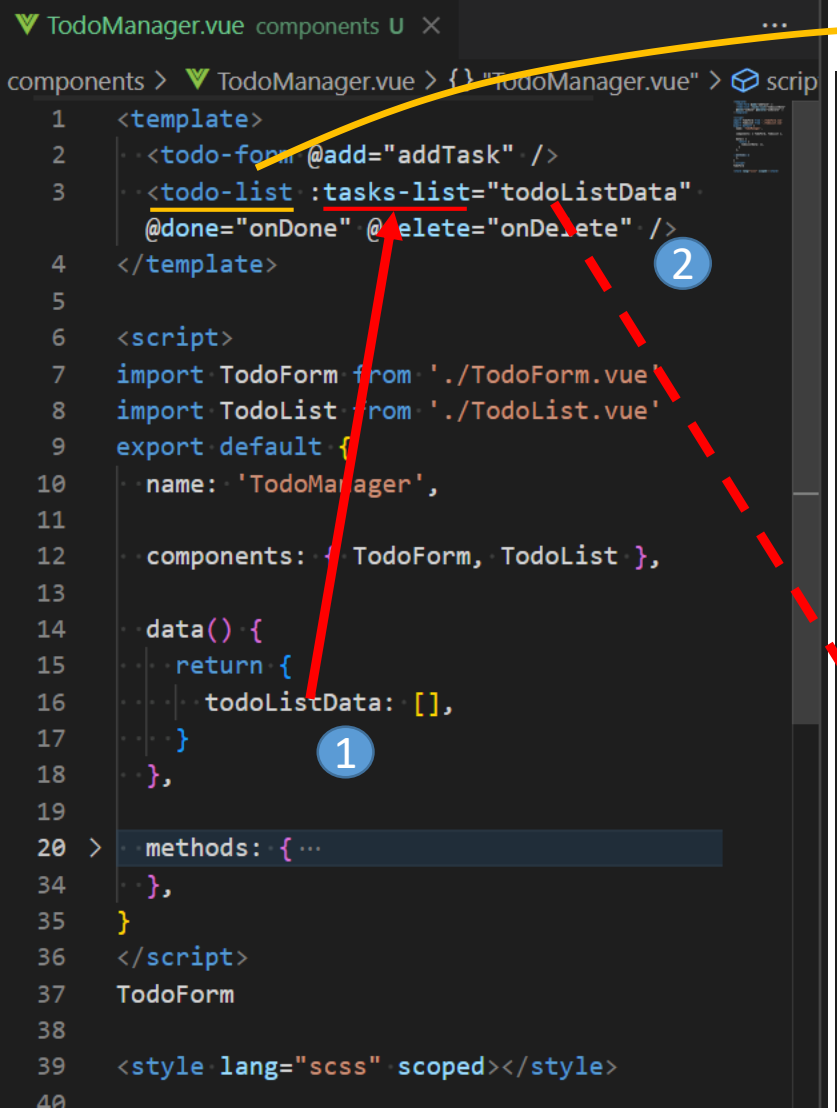
```vue
<template>
  <div class="list-container">
    <h1>Список задач</h1>
    <todo-item
      v-for="task in tasksList"
      :key="task.id"
      :task="task"
      @done="$emit('done', $event)"
      @delete="$emit('delete', $event)"
    />
  </div>
</template>
<script>
import TodoItem from './TodoItem.vue'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

  components: { TodoItem },
  emits: ['done', 'delete'],
  props: {
    tasksList: {
      type: Array,
      default: () => [],
    },
  }
}
</script>
```

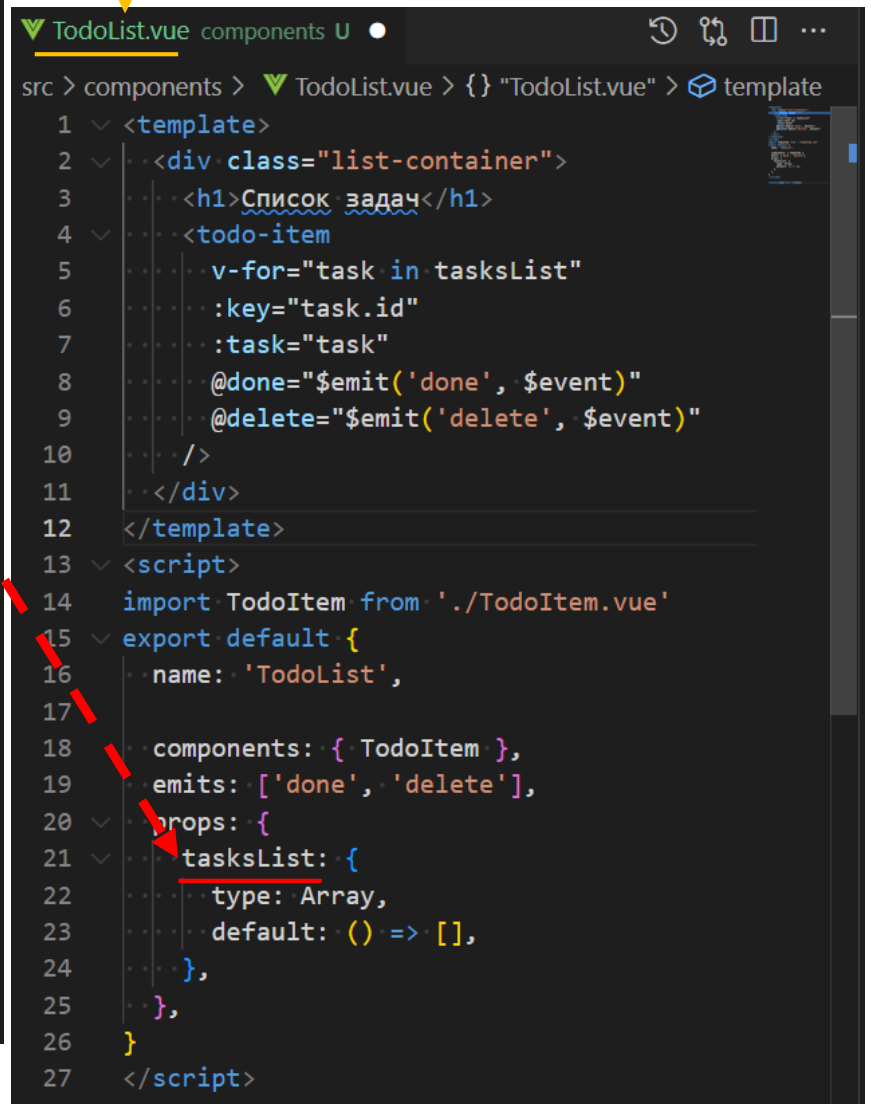
TodoItem.vue components U

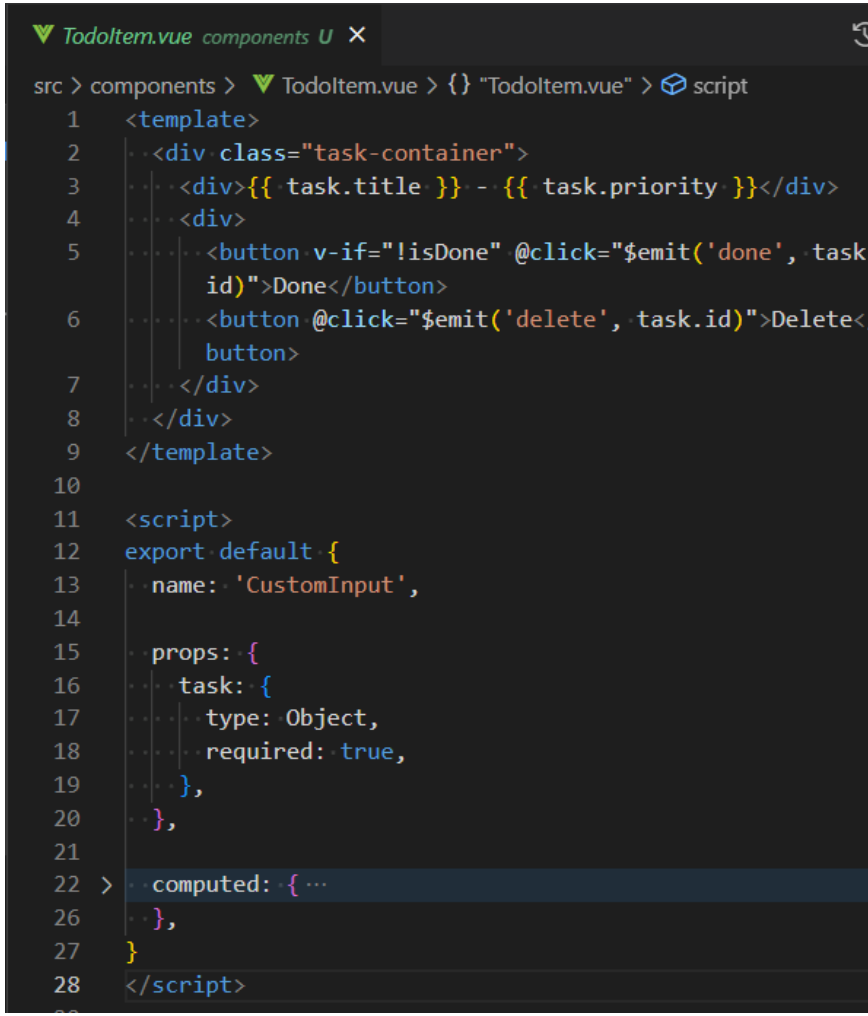
src > components > TodoItem.vue > { } "TodoItem.vue" > script

```vue
<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>
```

1) Масив tasksList міститься у TodoManager

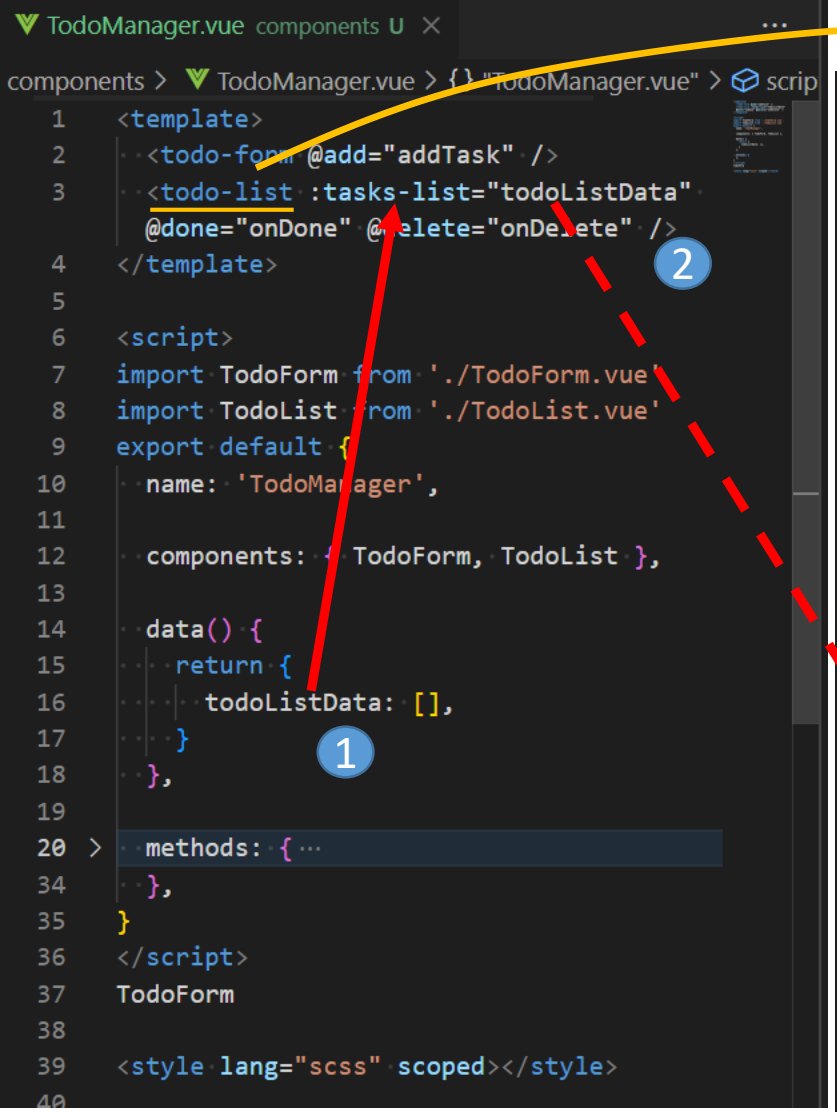


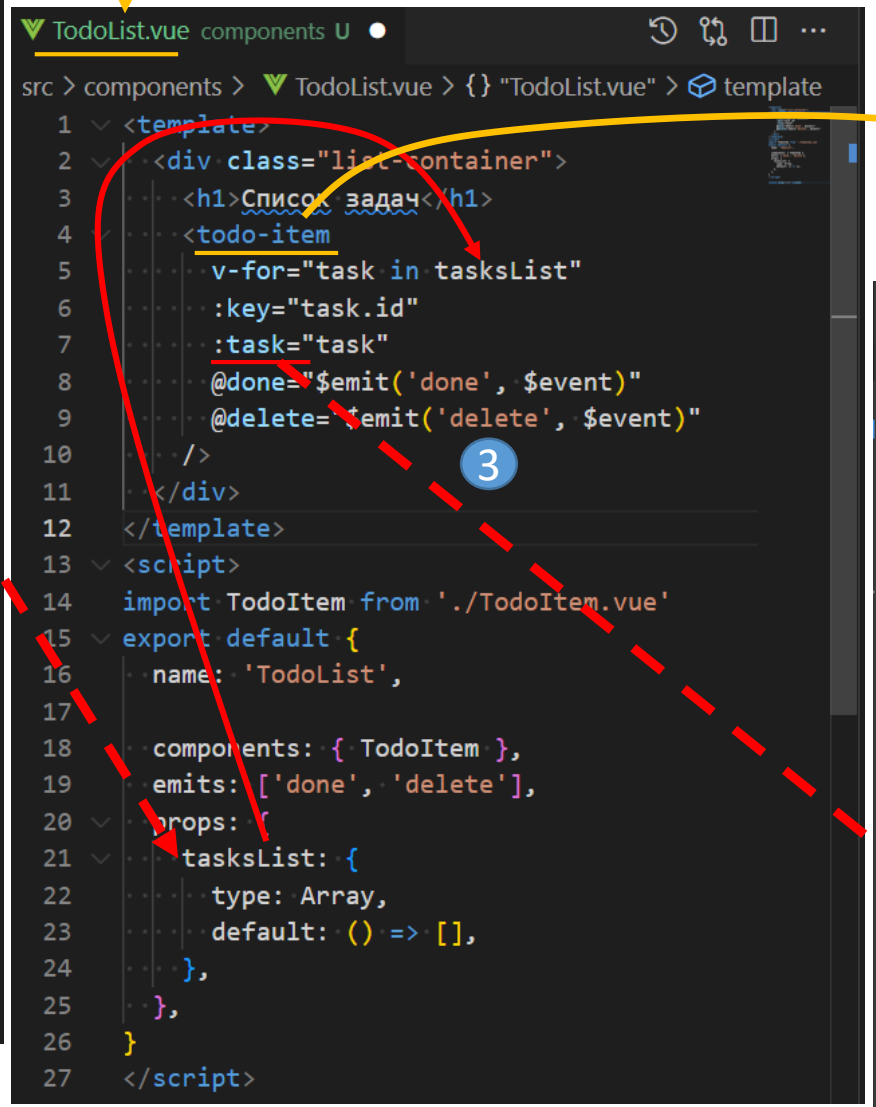


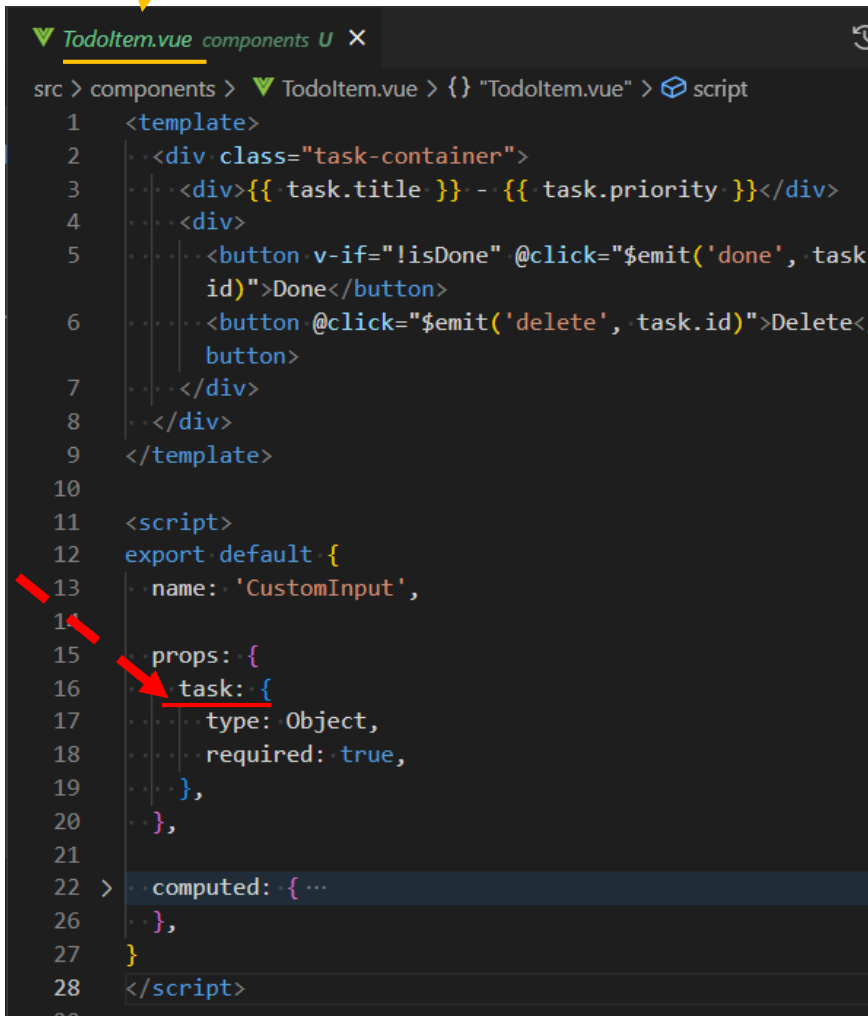


1) Масив tasksList міститься у TodoManager

2) TodoManager передає список задач у TodoList



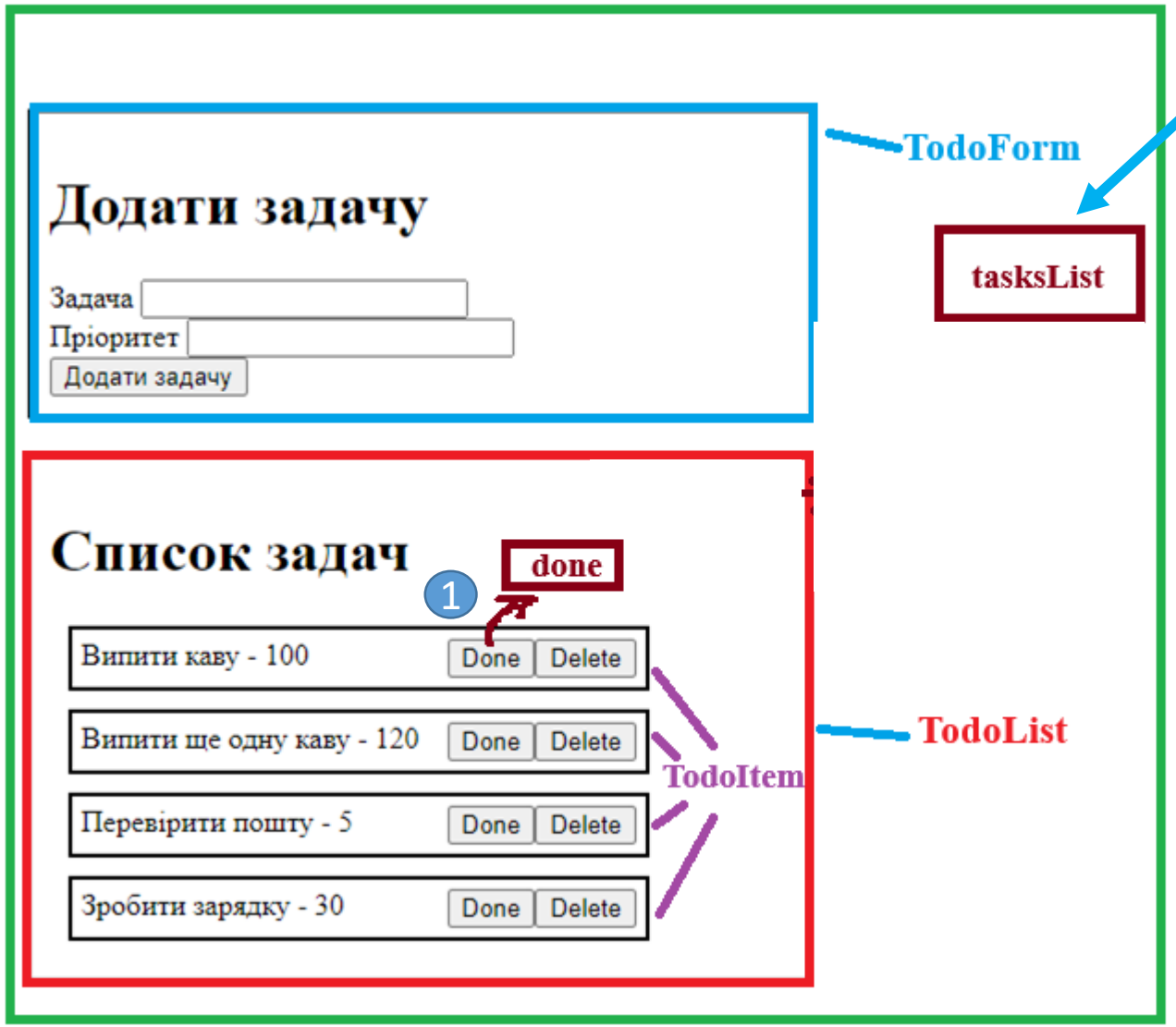




1) Масив tasksList міститься у TodoManager
2) TodoManager передає список задач у TodoList
3) TodoList відображає елементи

Для чого потрібе Vuex?

Розглянемо меджер задач



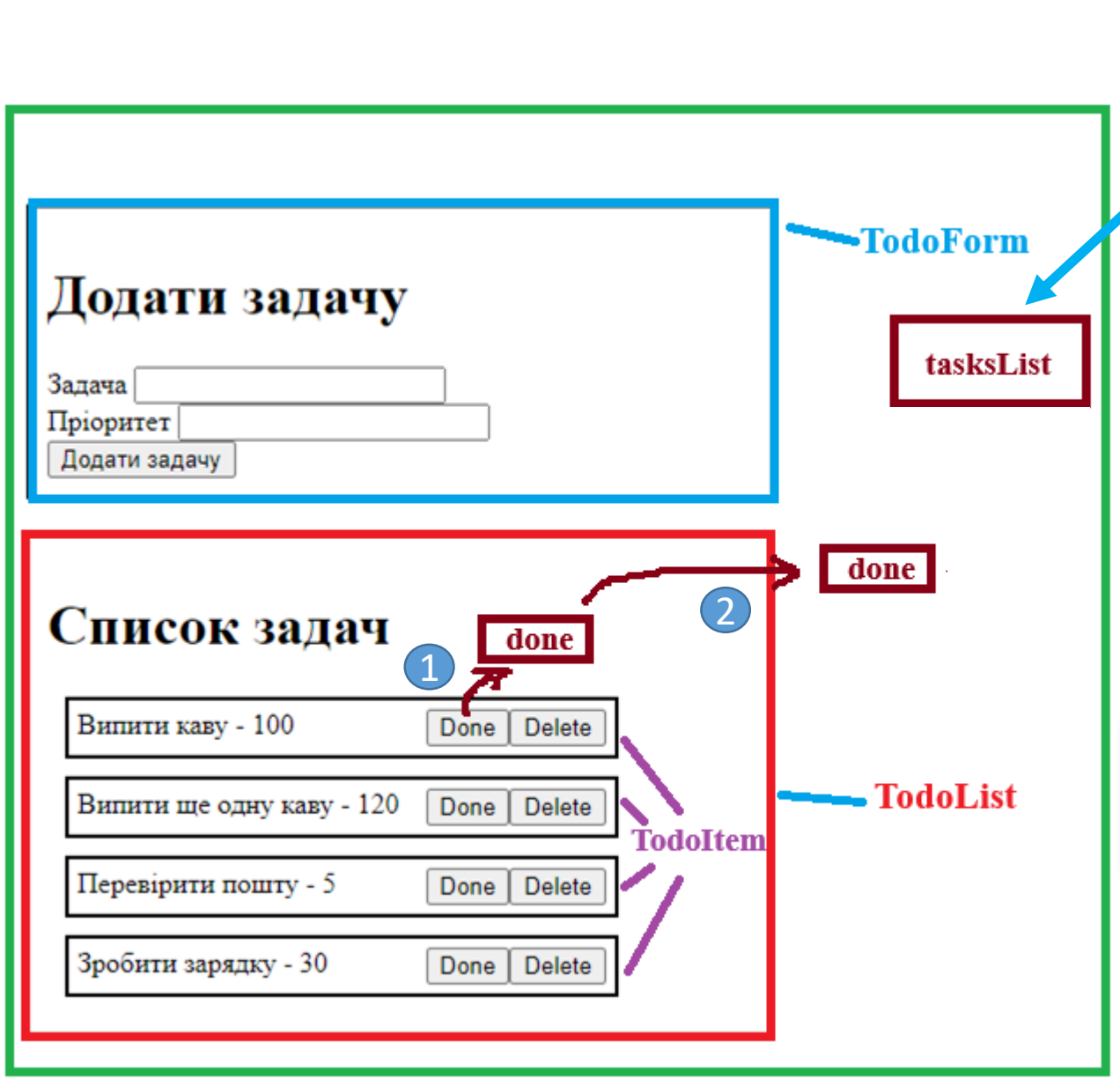
TodoManager

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



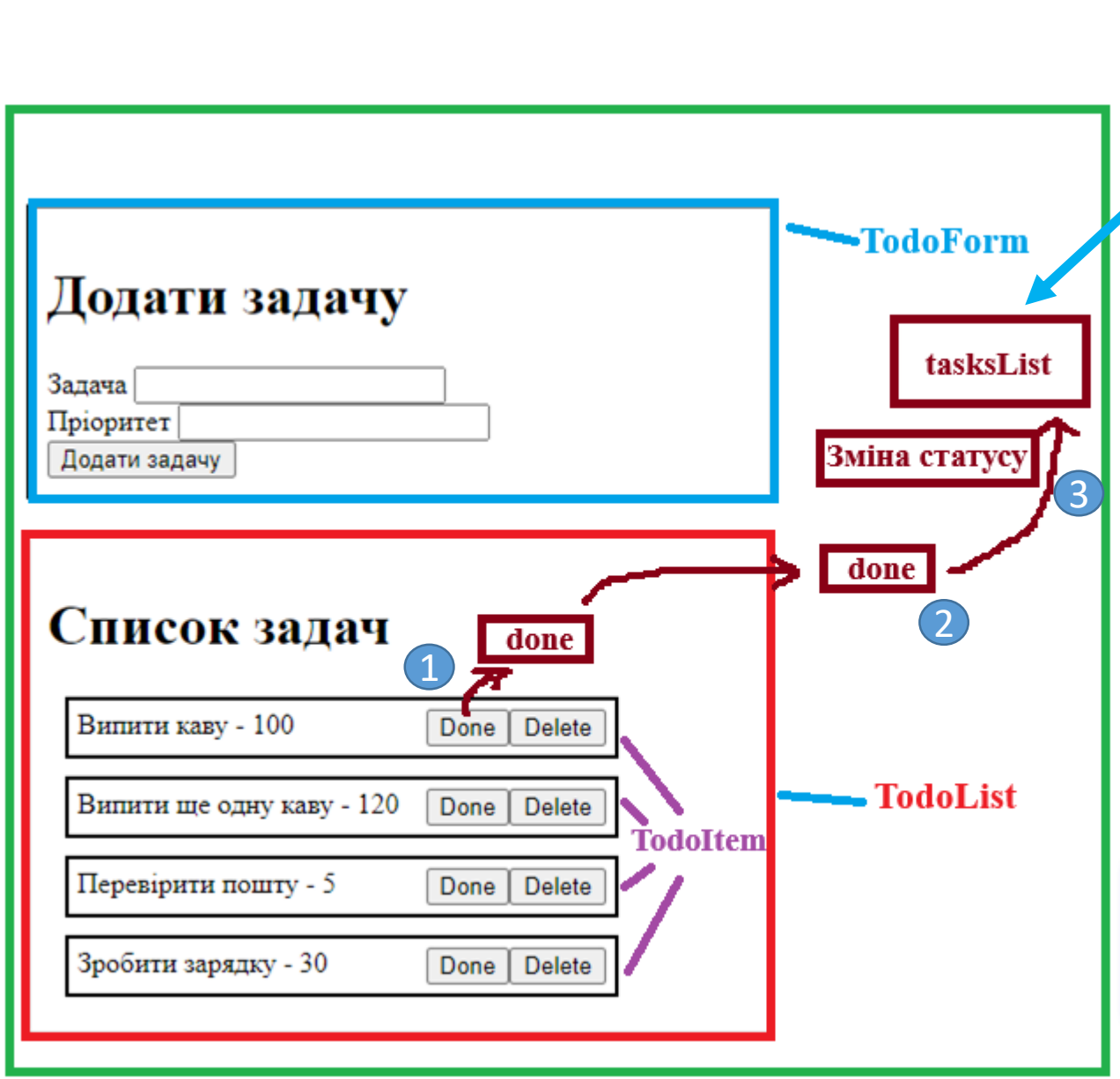
TodoManager

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



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”
3) TodoManager отримує подію “done” від дочірнього TodoList і робить зміни у списку

Для чого потрібе Vuex?

Розглянемо меджер задач

TodoManager.vue components U

```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 components U

```vue
<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)"
    />
  </div>
</template>
<script>
import TodoItem from './TodoItem.vue'
export default {
  name: 'TodoList',

  components: { TodoItem },
  emits: ['done', 'delete'],
  props: {
    tasksList: {
      type: Array,
      default: () => [],
    },
  },
}
</script>
```

TodoItem.vue components U

```vue
<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>
```

1

При необхідності зміни статусу задачі:

1) компонент TodoItem генерує подію “done”

Для чого потрібе Vuex?

Розглянемо меджер задач

TodoManager.vue components U

```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 components U

```vue
<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)"
    />
  </div>
</template>
<script>
import TodoItem from './TodoItem.vue'
export default {
  name: 'TodoList',

  components: { TodoItem },
  emits: ['done', 'delete'],
  props: {
    tasksList: {
      type: Array,
      default: () => [],
    },
  },
}
</script>
```

TodoItem.vue components U

```vue
<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>
```

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

TodoManager.vue components U

```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 components U

```vue
<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)"
    />
  </div>
</template>
<script>
import TodoItem from './TodoItem.vue'
export default {
  name: 'TodoList',

  components: { TodoItem },
  emits: ['done', 'delete'],
  props: {
    tasksList: {
      type: Array,
      default: () => [],
    },
  },
}
</script>
```

TodoItem.vue components U

```vue
<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>
```

При необхідності зміни статусу задачі:

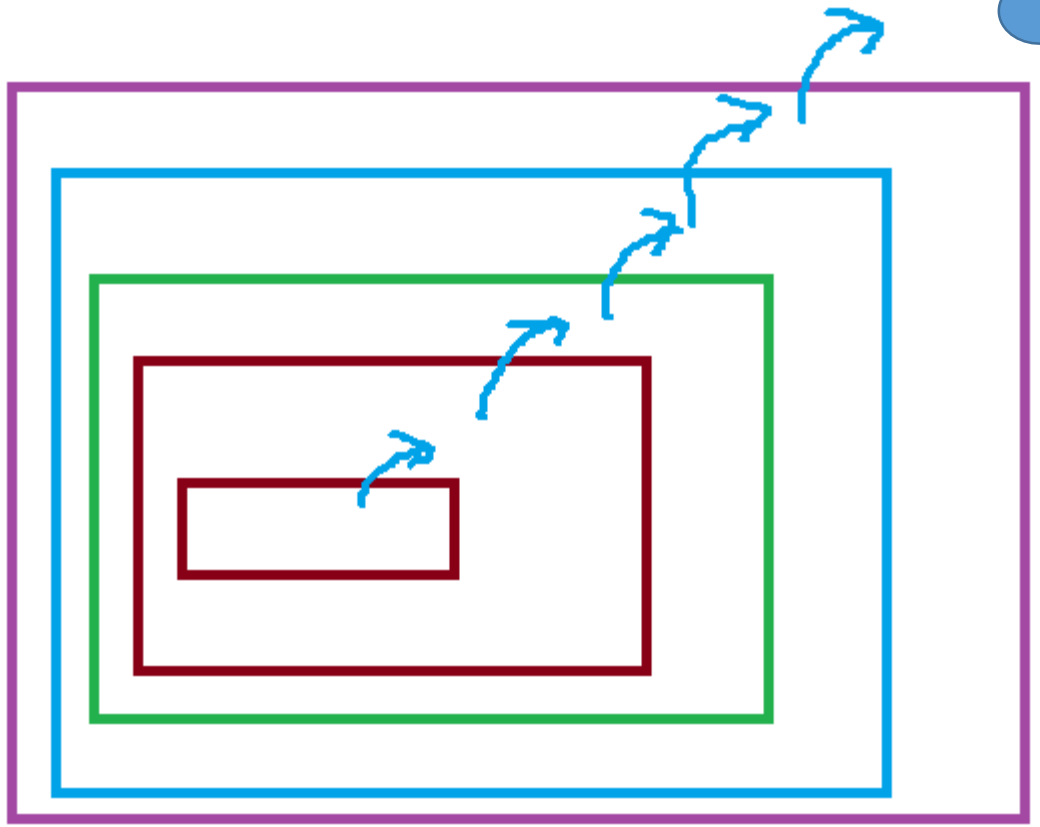
1) компонент TodoItem генерує подію “done”

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

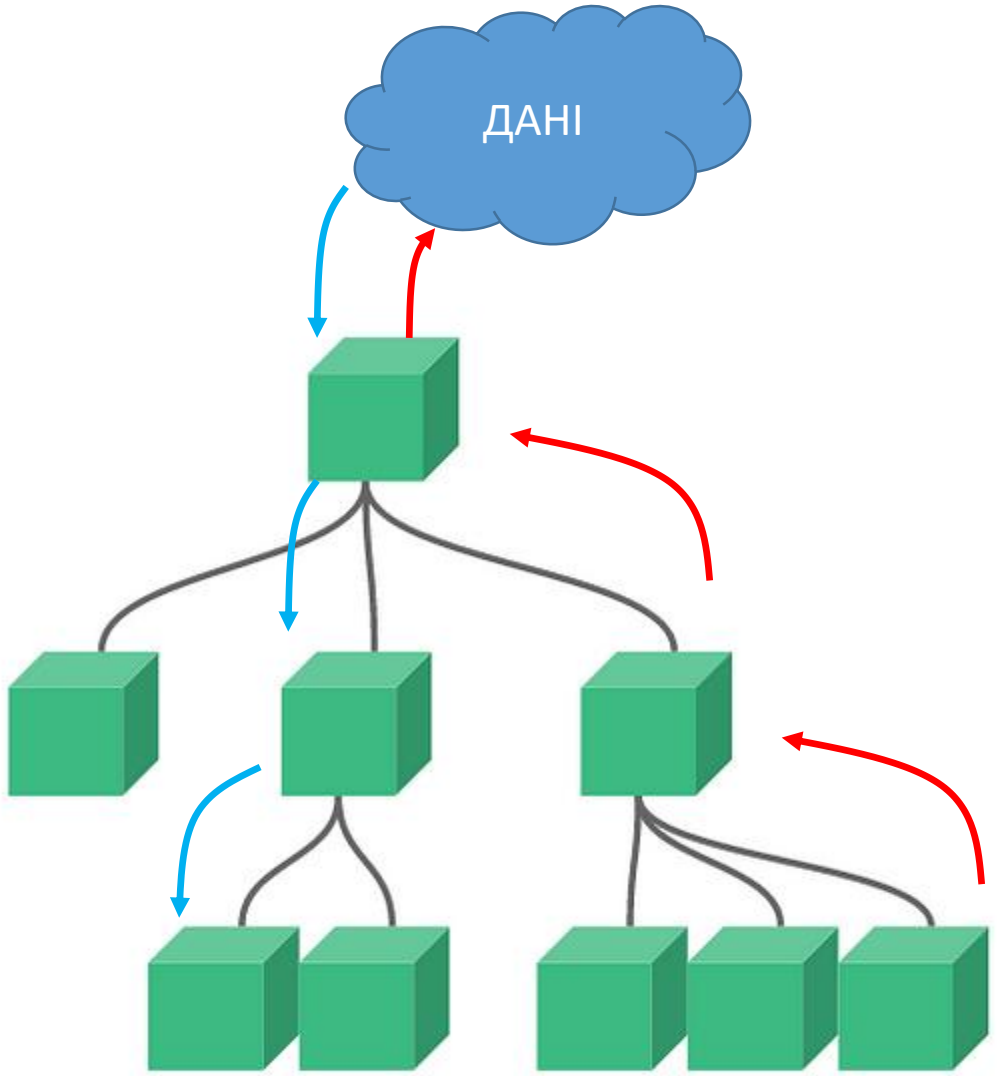
3) TodoManager отримує подію “done” від дочірнього TodoList і робить зміни у списку

Для чого потрібе Vuex?

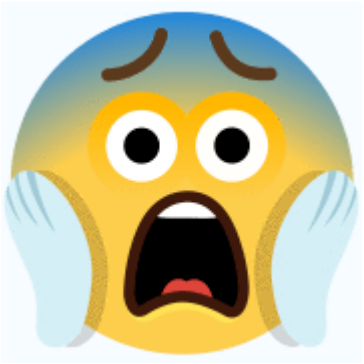
А що буде, якщо глибина буде ще більшою?



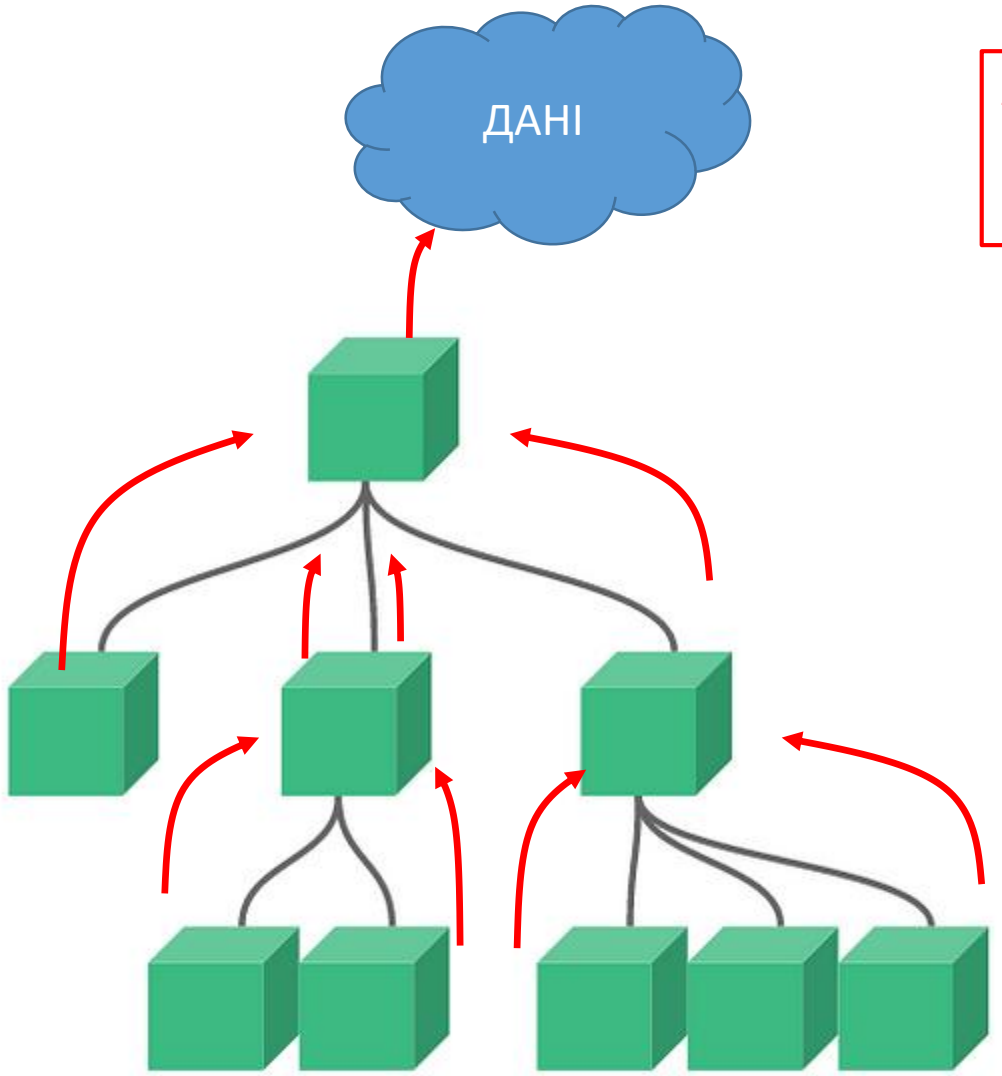
Для чого потрібе Vuex?



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



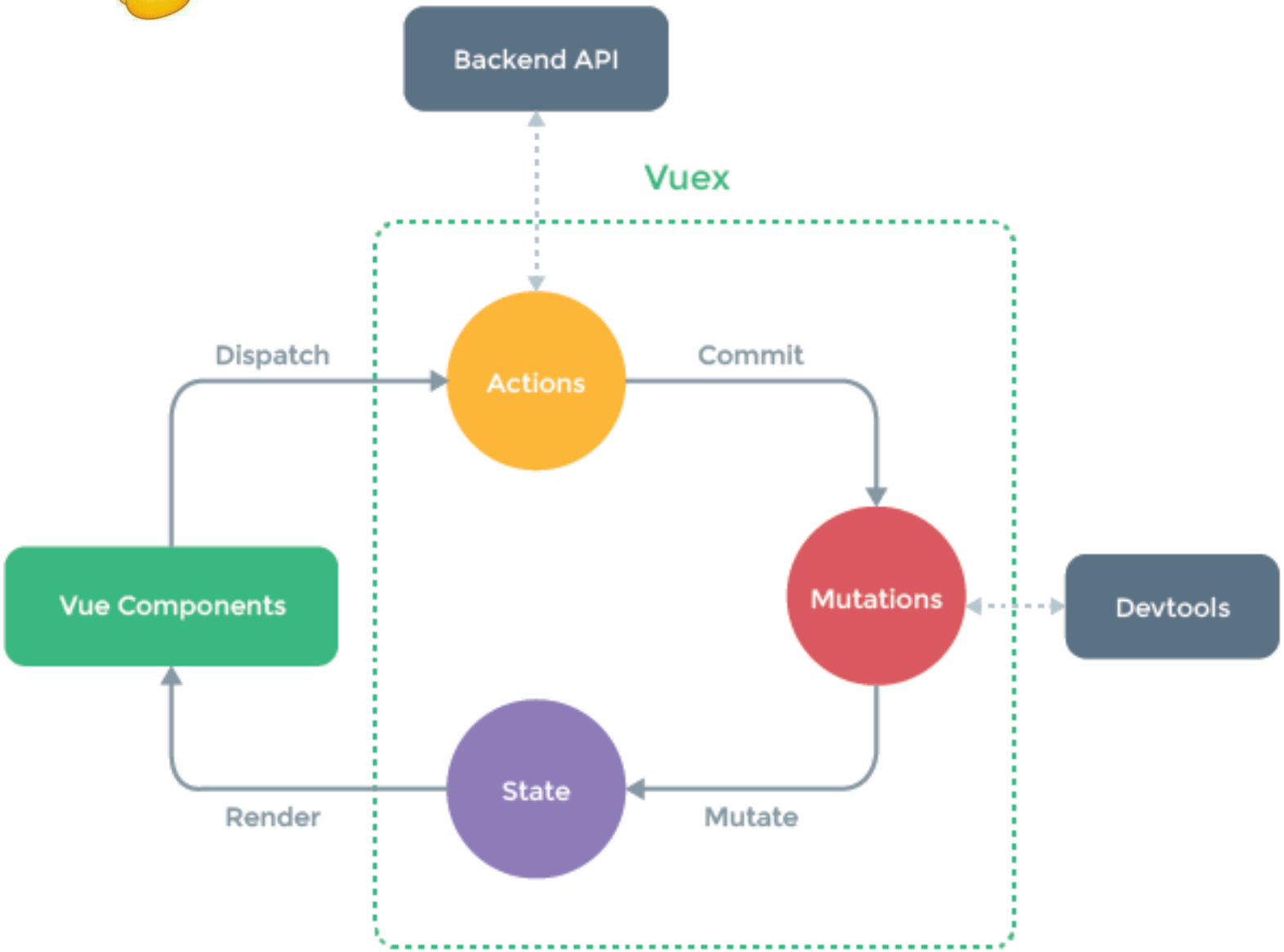
Для чого потрібе Vuex?



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

Для чого потрібе Vuex?

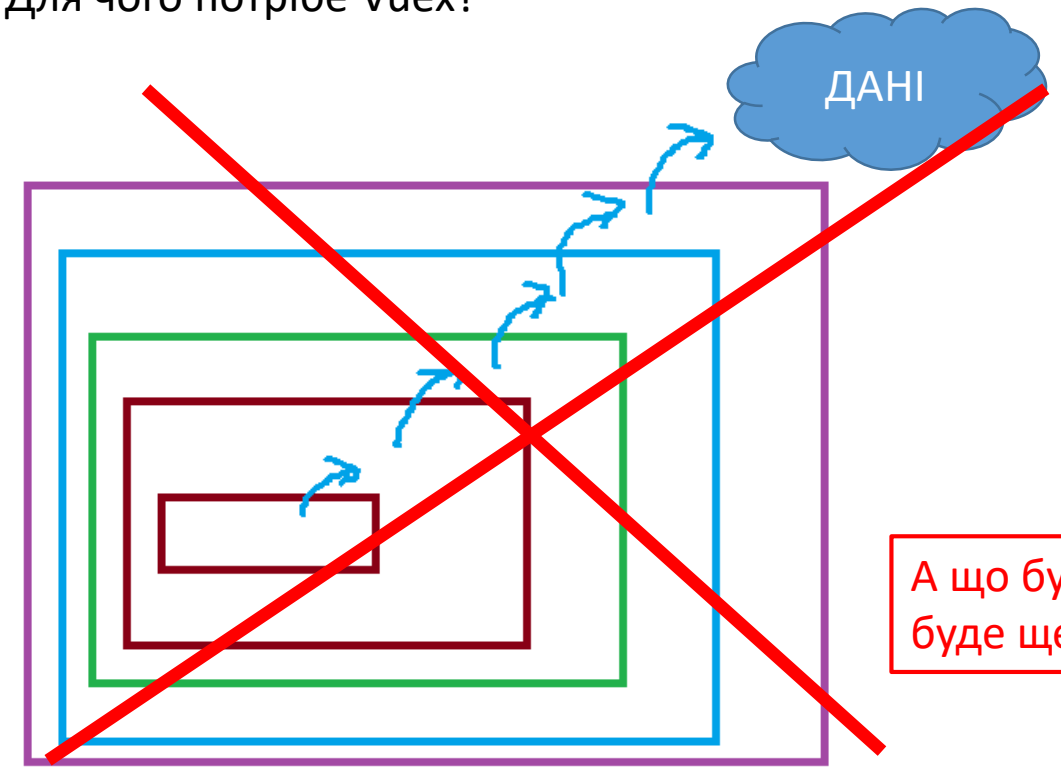
***Flux*-архитектура** – набір шаблонів програмування для побудови інтерфейсу веб-додатків заснований на одноннаправлених потоках даних



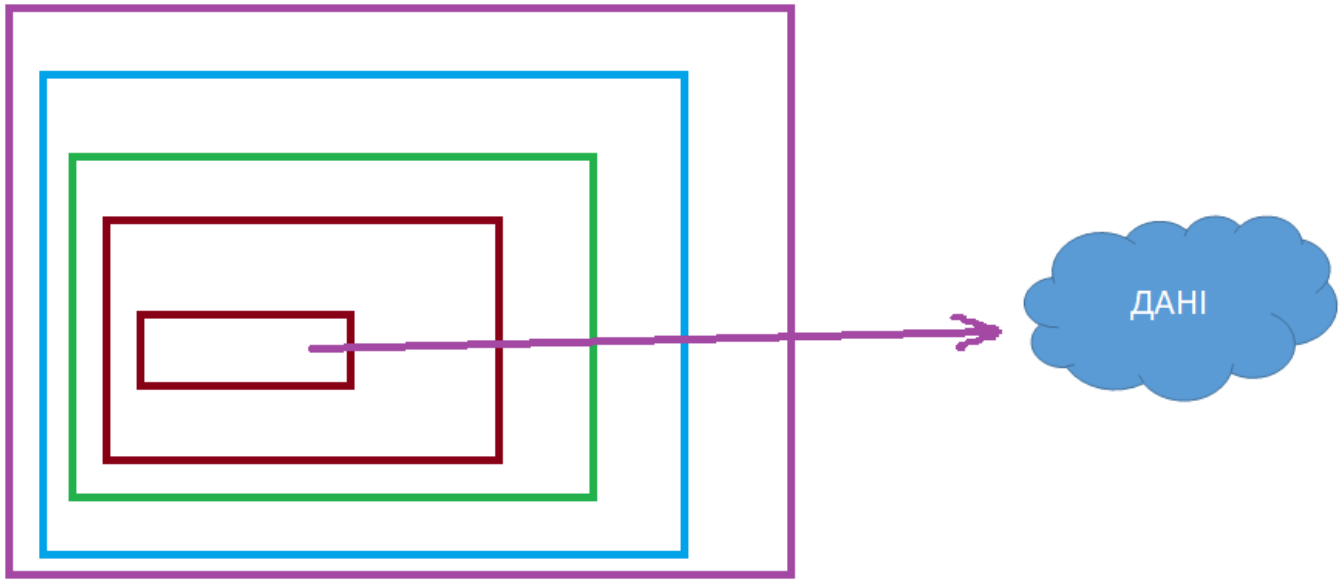
Принципи :

1. Існує єдине глобальне сховище ***State***
2. Всі компоненти можуть читати значення безпоседеньо 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?



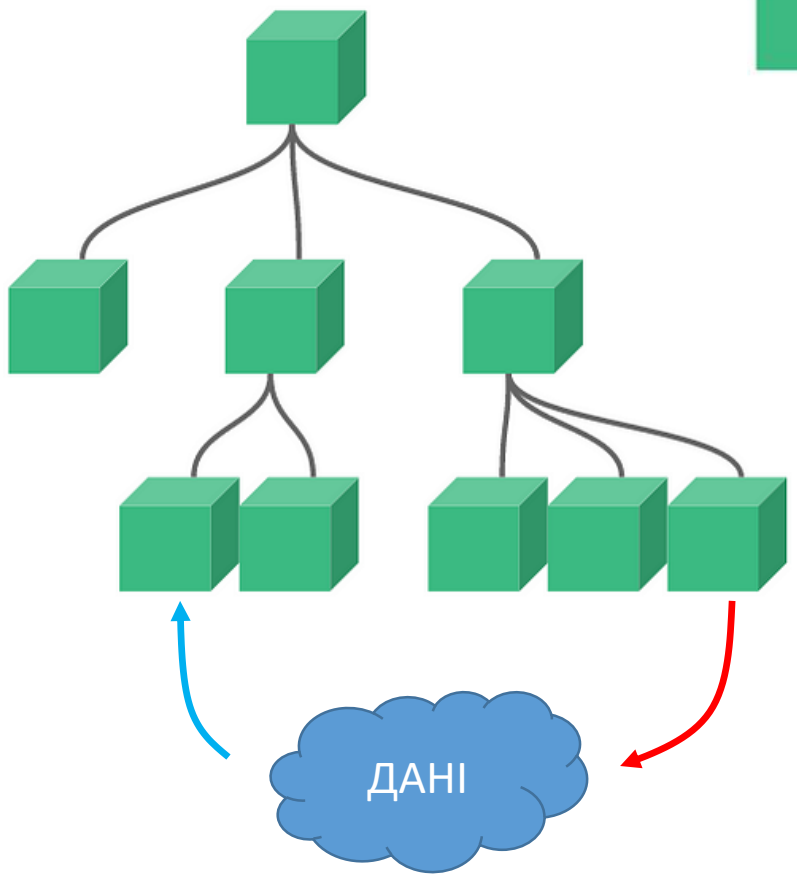
А що буде, якщо глибина буде ще більшою?

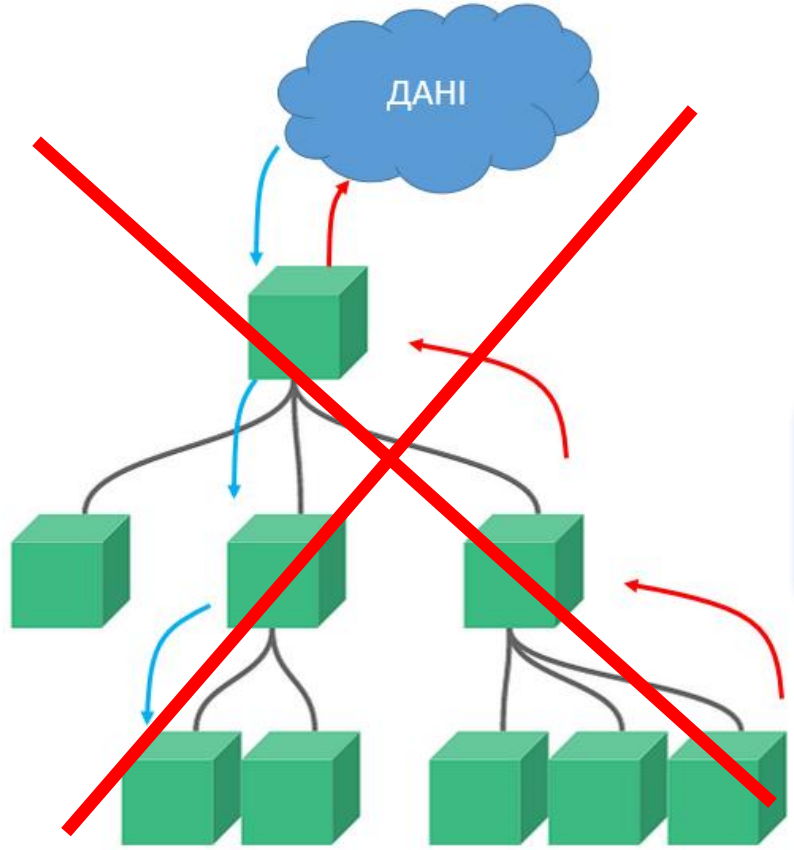


Принципи :

1. Існує єдине глобальне сховище ***State***
2. Всі компоненти можуть читати значення безпоседеньо 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?



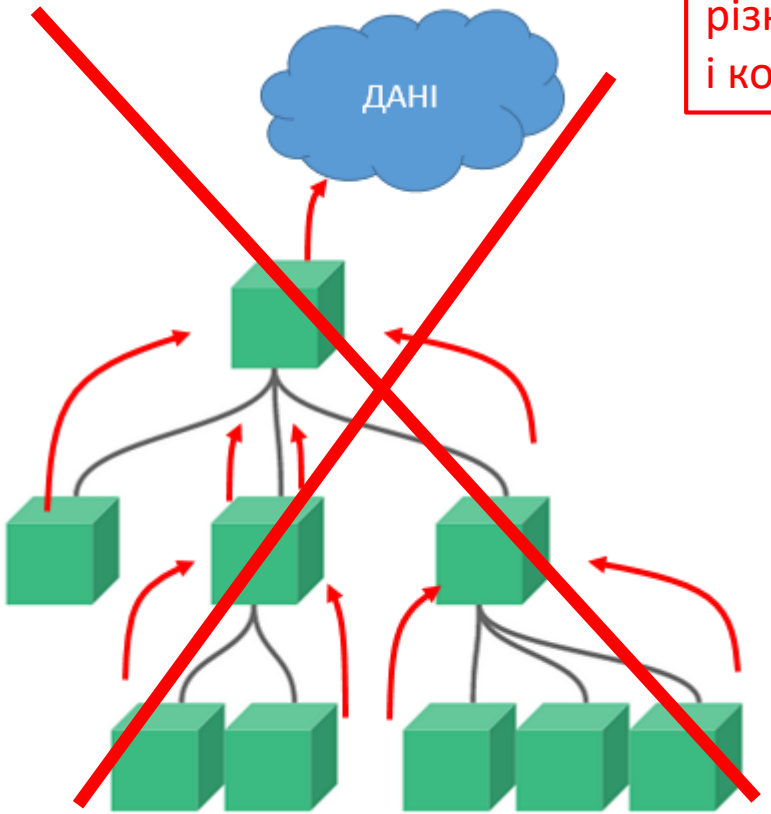


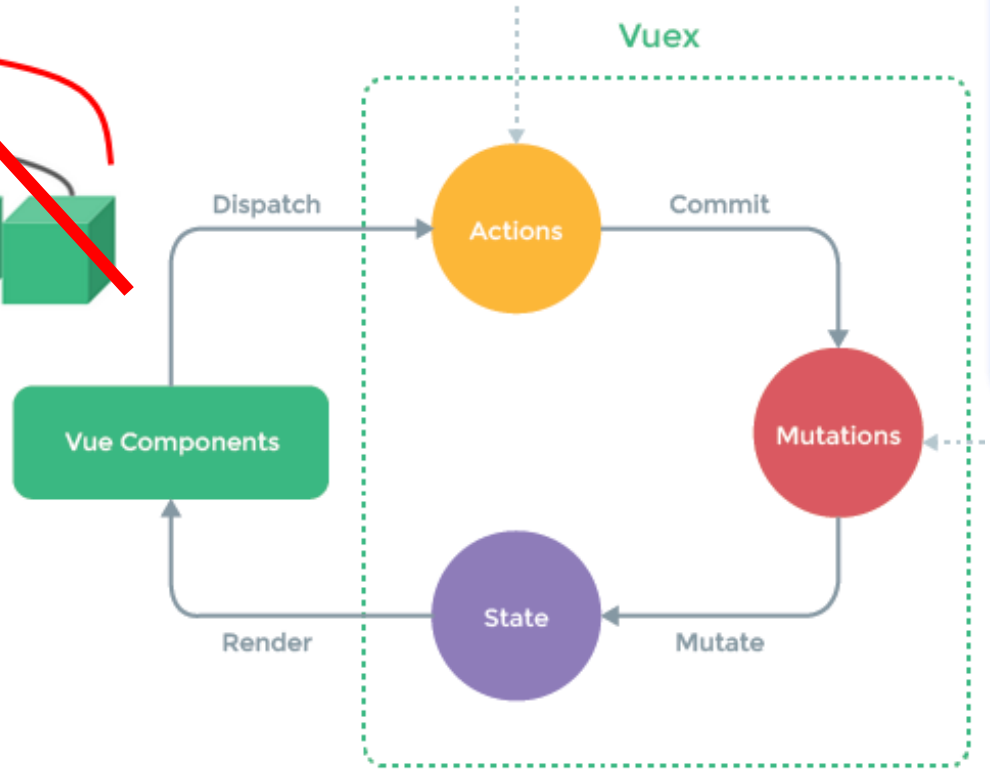
Принципи :

1. Існує єдине глобальне сховище ***State***
2. Всі компоненти можуть читати значення безпоседеньо 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?



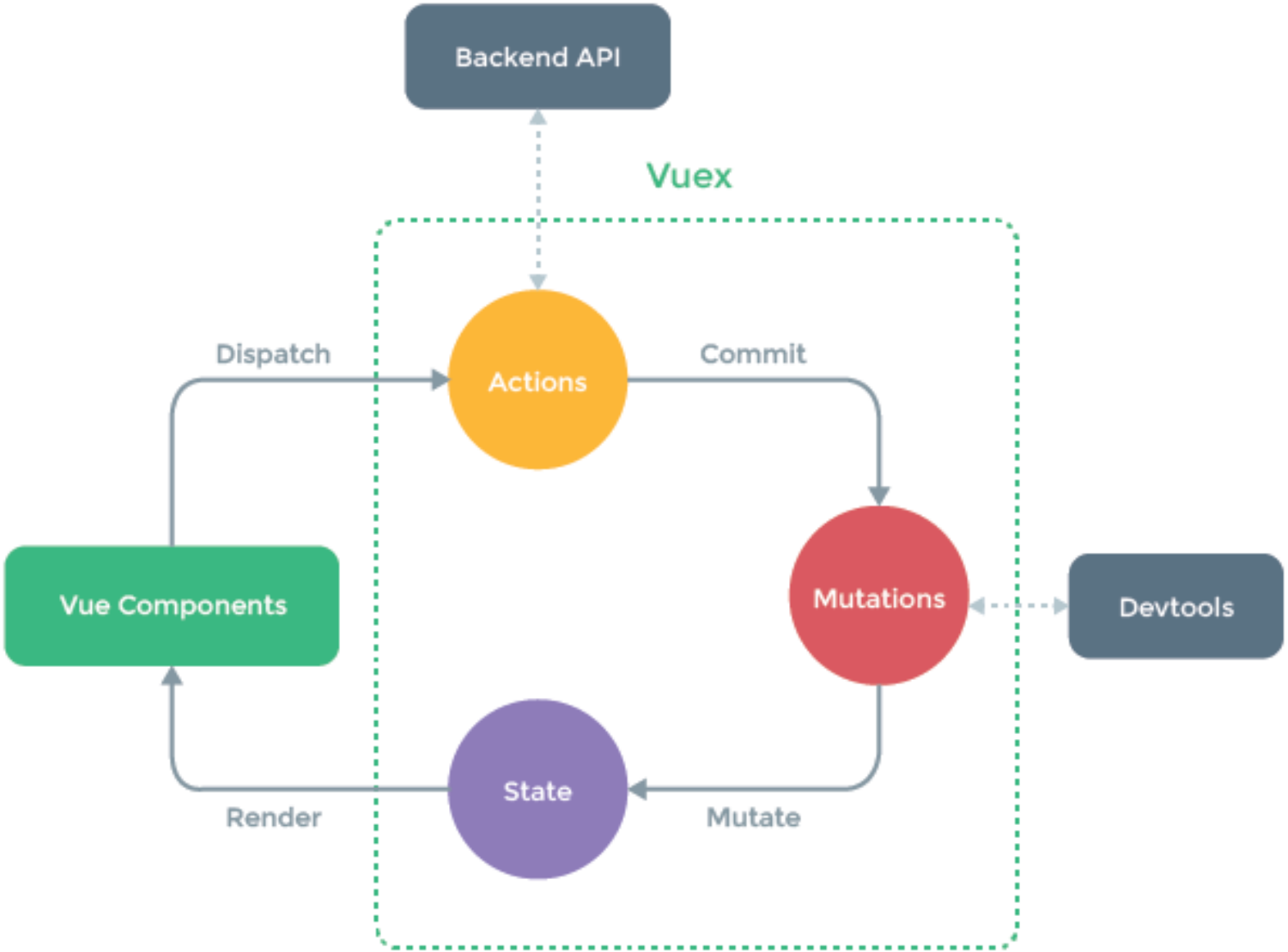


Принципи :

1. Існує єдине глобальне сховище ***State***
2. Всі компоненти можуть читати значення безпоседеньо 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?

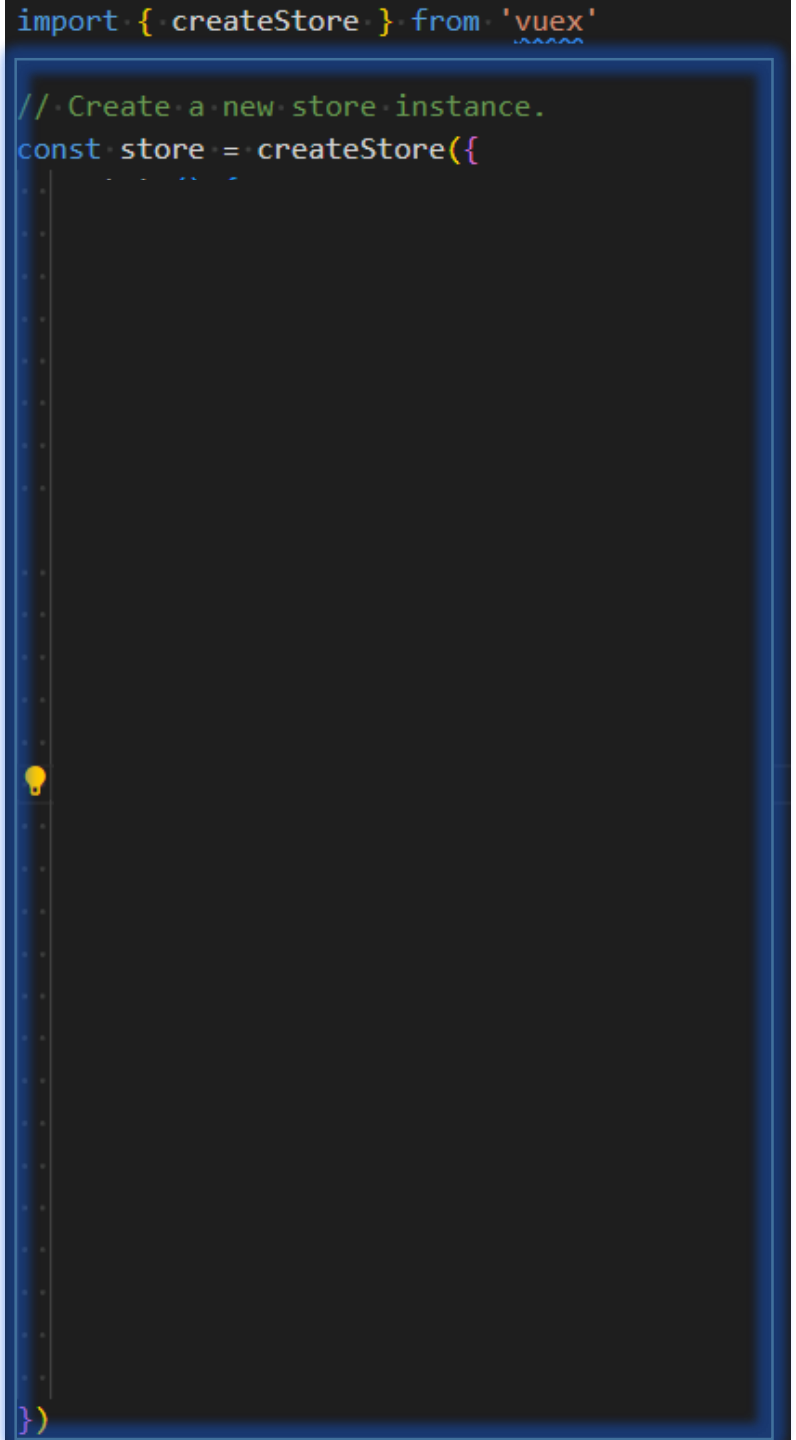
***Flux*-архитектура** – набір шаблонів програмування для побудови інтерфейсу веб-додатків заснований на одноннаправлених потоках даних



Принципи :

1. Існує єдине глобальне сховище ***State***
2. Всі компоненти можуть читати значення безпоседеньо 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({



})
```



- - - - 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 {
      властивсість1: поч.знач.1,
      властивсість2: 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 {
      властивсість1: поч.знач.1,
      властивсість2: 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 {
      властивсість1: поч.знач.1,
      властивсість2: 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 {
      властивсість1: поч.знач.1,
      властивсість2: 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 {
      властивсість1: поч.знач.1,
      властивсість2: 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')
    }
      . . . . . .
  }
})
```

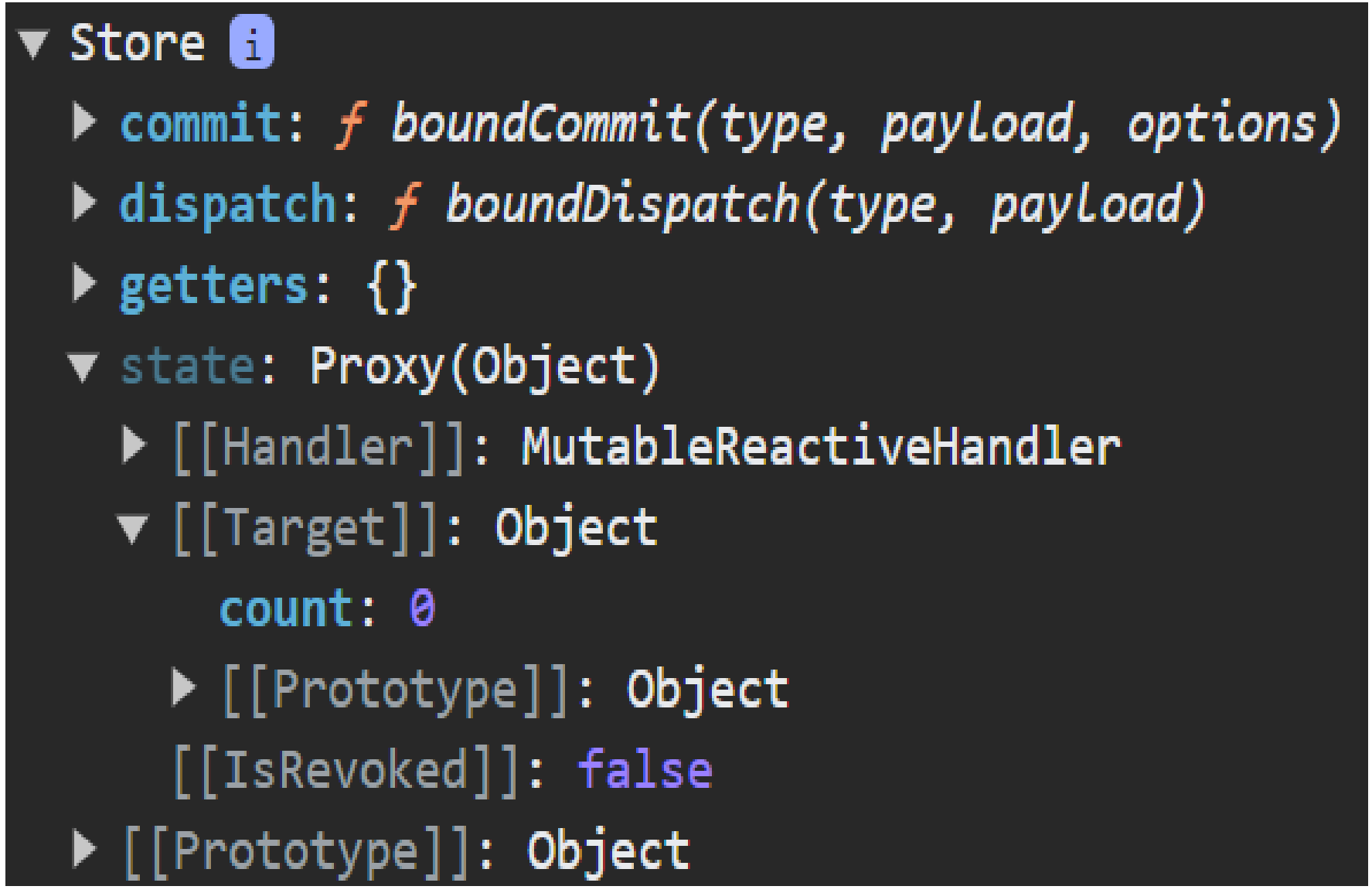
```
import { createStore } from 'vuex'

// 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



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

this . $store

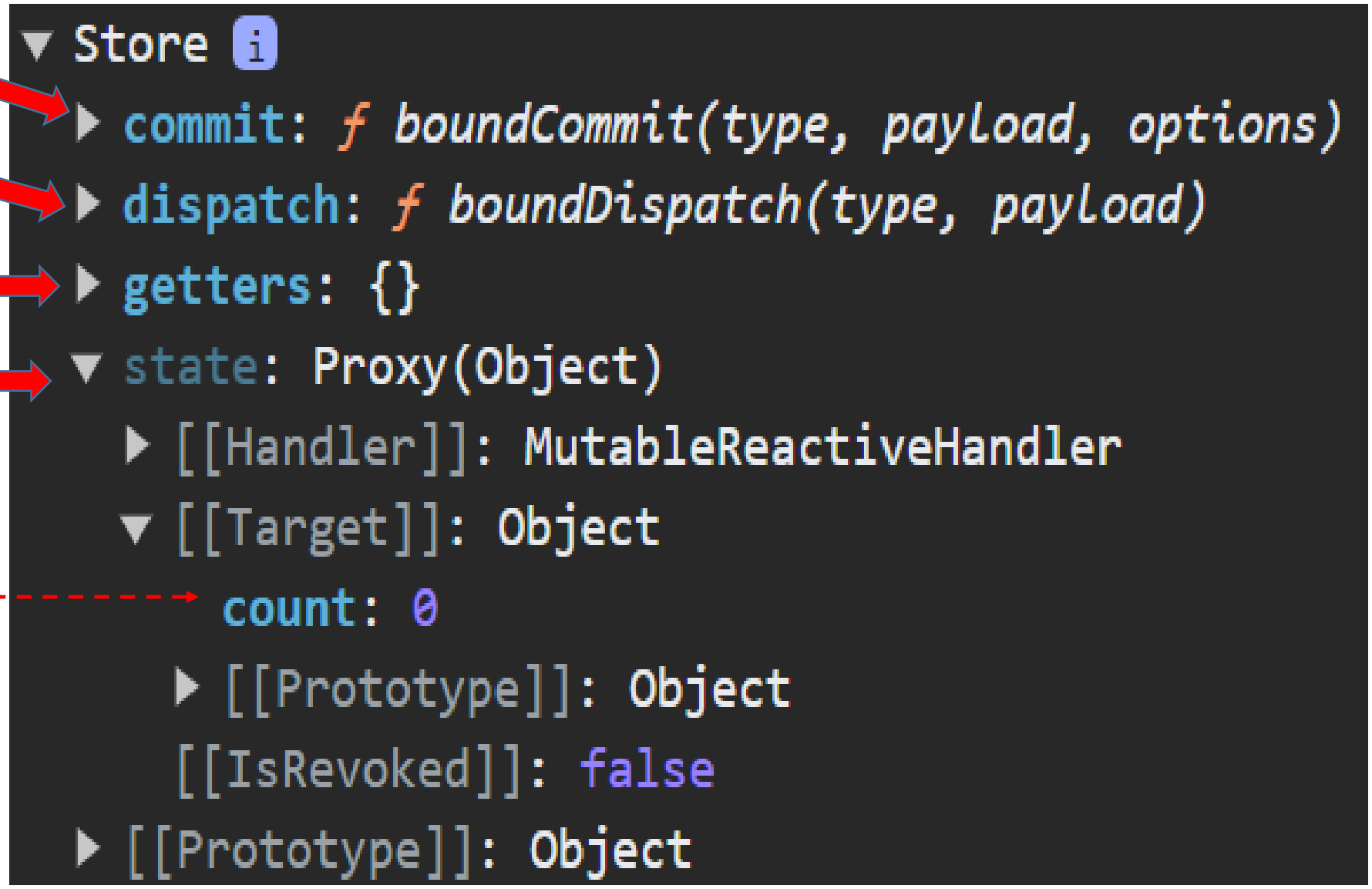
Для виклику мутацій

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>
```

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
```

Пряме звертання до гетерів 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 даних 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 гетерів з сховища (getters)

Фрагмент компонента

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

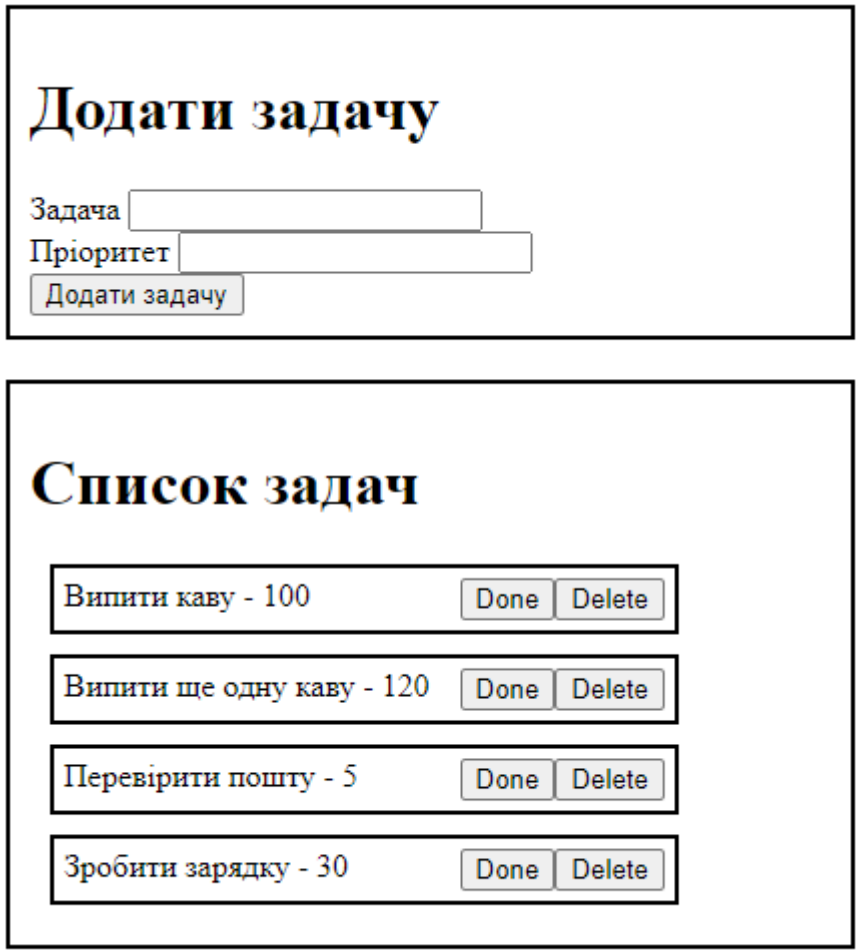
Загальна форма

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 Vuex



Використання гелперів для ***інтеграції*** компонентів сховища у компонети

Для ітеграції (вставки) значень з state → ***mapState***

Для ітеграції (вставки) гететрів → ***mapGetters***

Для ітеграції (вставки) мутацій-методів → ***mapMutations***

Для ітеграції (вставки) action-методів → ***mapActions***

https://vuex.vuejs.org/guide/state.html#the-mapstate-helper

Зчитування значень з сховища у компонентах (state)

Для ітеграції (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,

        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>




</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,

        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>




</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 ітеграції (вставки) гететрів → ***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. Імпортуємо 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 ітеграції (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>
```

----- 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,

        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'

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,

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

<script>
//1. Імпортуємо 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,

        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.

```
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
```

Для ітеграції (вставки) action-методів → ***mapActions***

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.

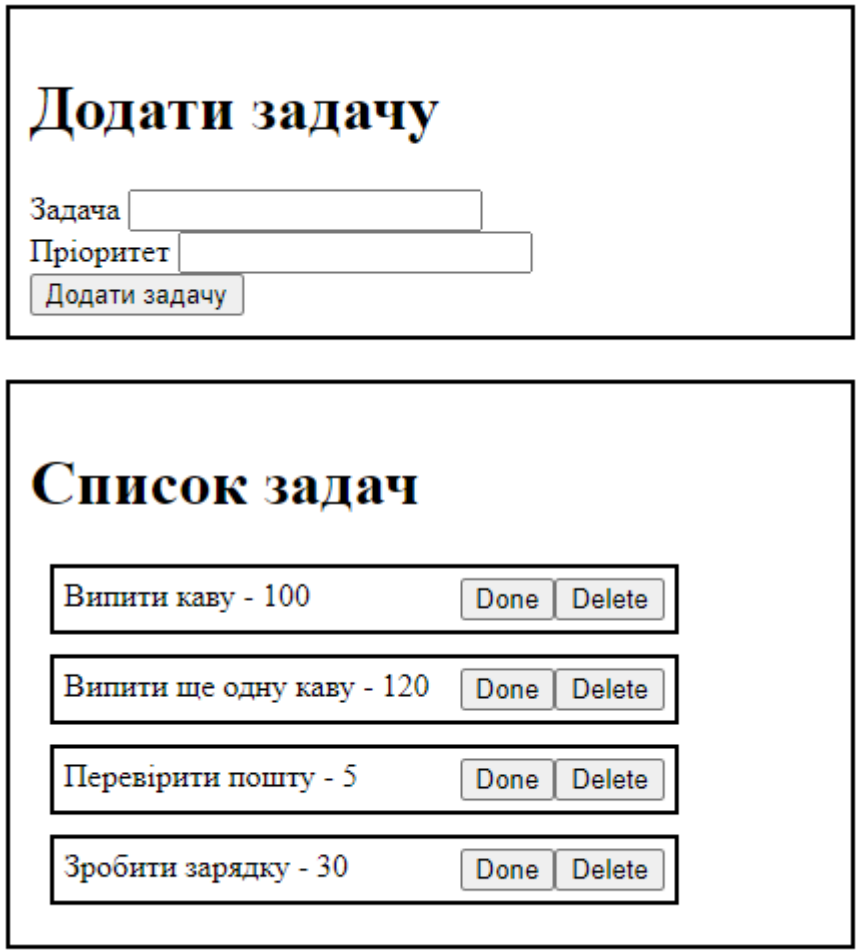
```
<template>
//3. Використовуємо action-методи
як звичайні методи
      . . . . . .
    <тег @подія="action-метод" />
      . . . . . .
</template>

<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,

        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 Vuex та гелперів



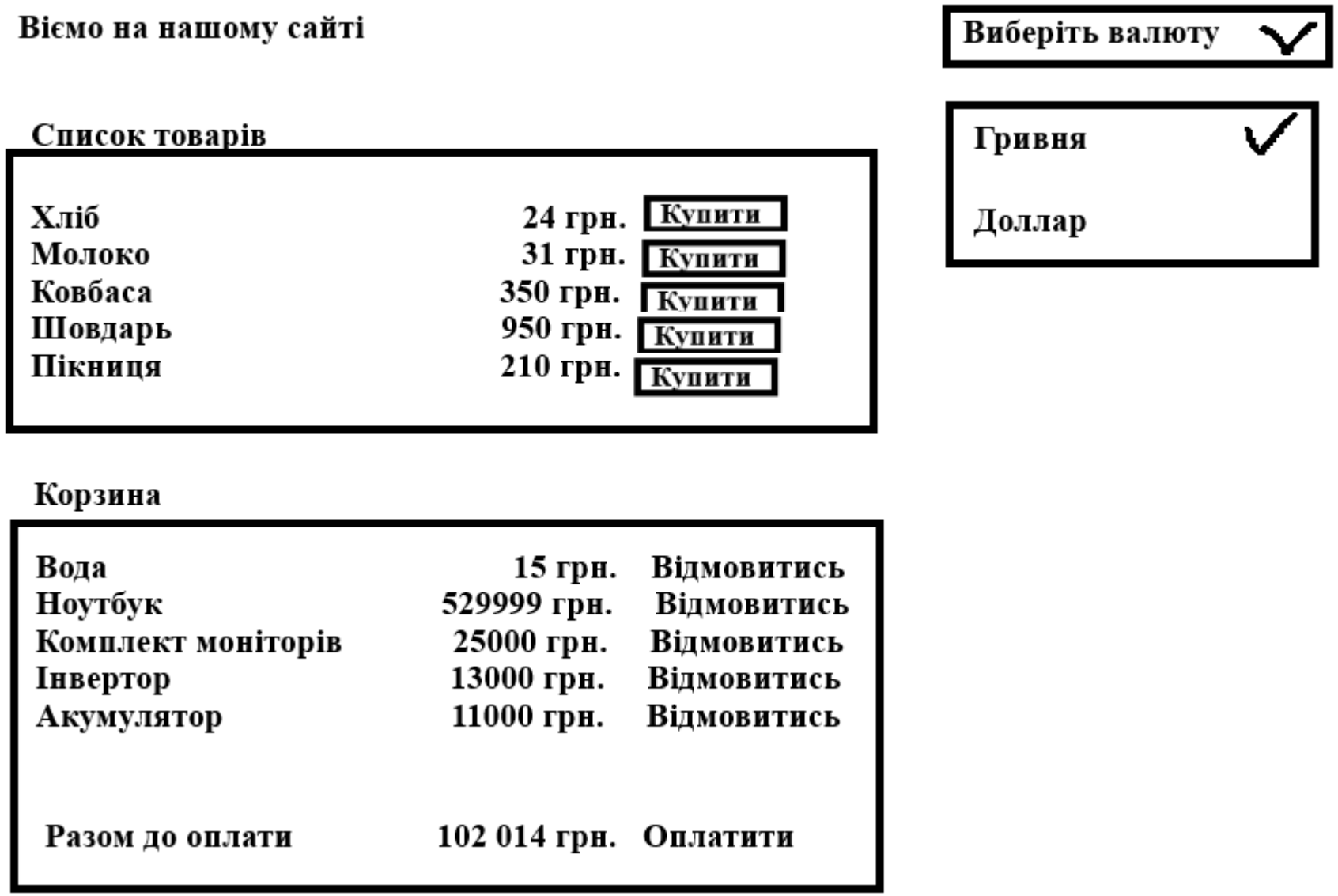
Список студентів **Система оцінювання** **Категорія**

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 Vuex



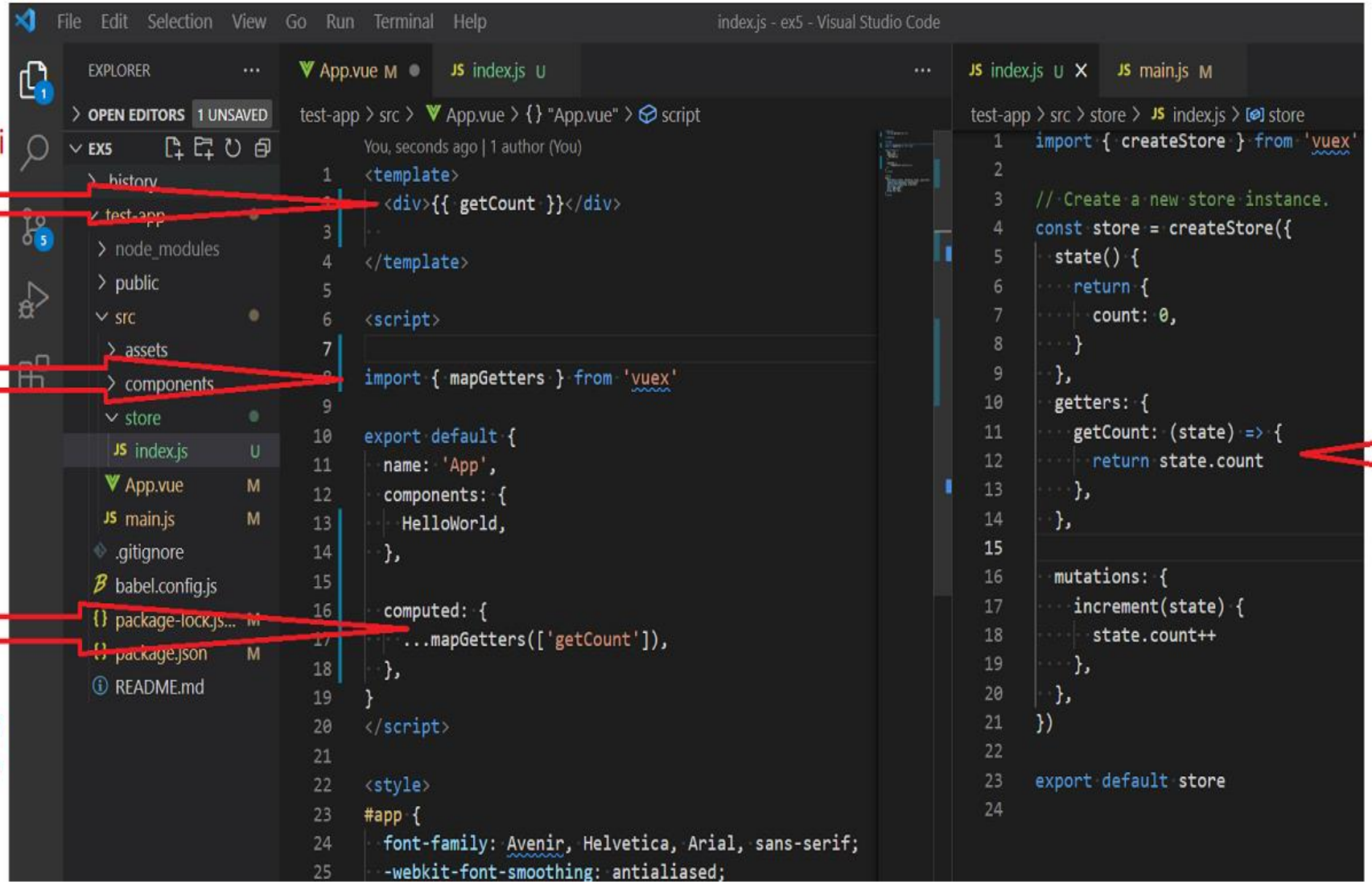
Ще додаткові слайди за аналогічними схемами

Використання геттерів

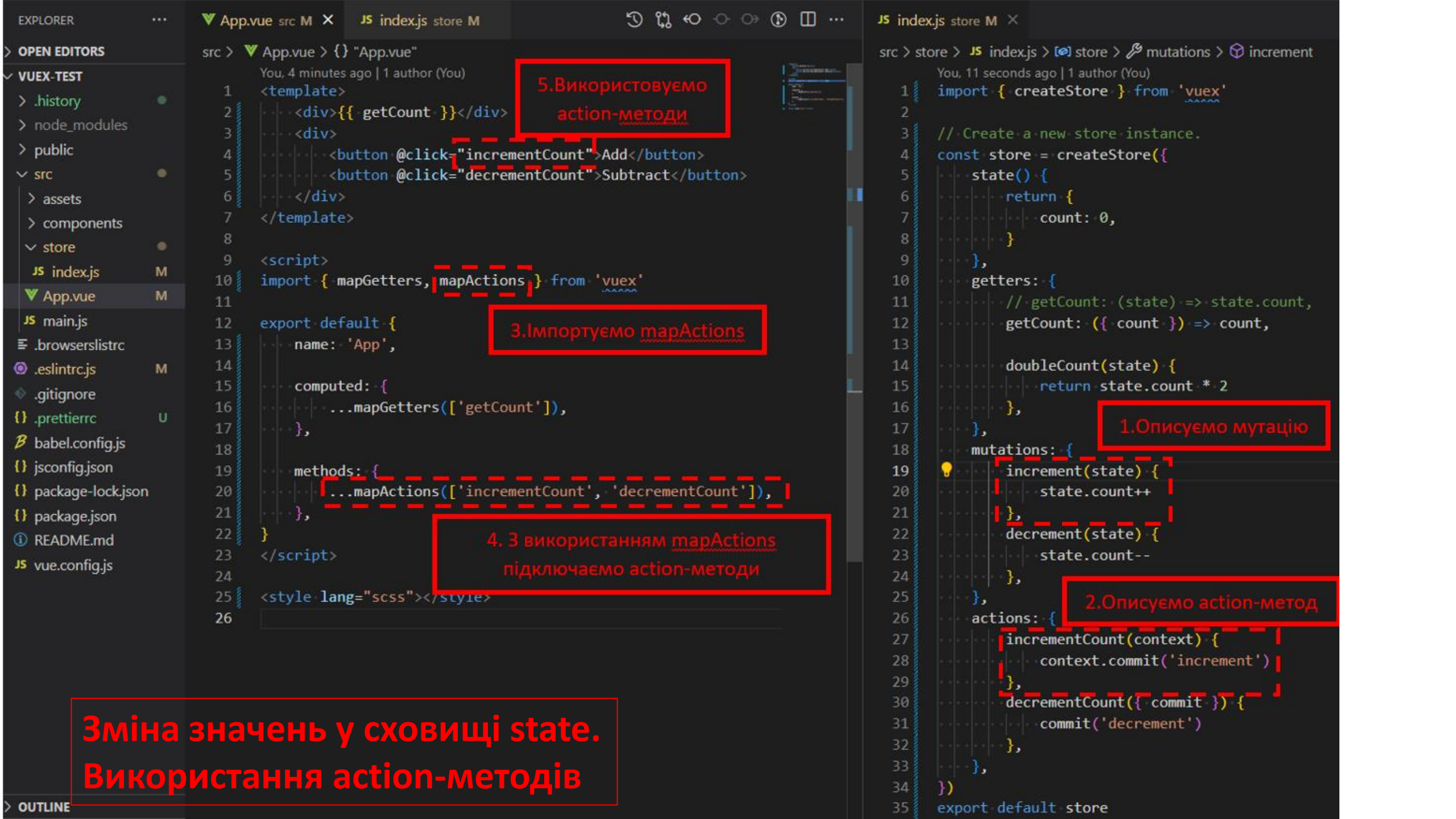
4. Використовуємо підключені геттери як звичайні властивості

2. Імпортуємо mapGetters

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



1. Описуємо getters


EXPLORER
OPEN EDITORS
VUEX-TEST
.history
node_modules
public
src
assets
components
store
index.js M
App.vue M
main.js
.browserslistrc
.eslintrc.js M
.gitignore
.prettierrc U
babel.config.js
jsconfig.json
package-lock.json
package.json
README.md
vue.config.js
OUTLINE
App.vue src M
index.js store M
src > App.vue > {} "App.vue"
You, 4 minutes ago | 1 author (You)
<template>
{{ getCount }}

<button @click="incrementCount">Add</button>
<button @click="decrementCount">Subtract</button>

</template>
<script>
import { mapGetters, mapActions } from 'vuex'
export default {
name: 'App',
computed: {
...mapGetters(['getCount']),
},
methods: {
...mapActions(['incrementCount', 'decrementCount']),
},
}
</script>
<style lang="scss"></style>
5.Використовуємо action-методи
3.Імпортуємо mapActions
4. З використанням mapActions підключаємо action-методи
index.js store M
src > store > index.js > store > mutations > increment
You, 11 seconds ago | 1 author (You)
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
1.Описуємо мутацію
2.Описуємо action-метод
Зміна значень у сховищі state.
Використання action-методів