Component Communication Tutorial

In this tutorial, you'll make a todo list, but using separate components instead of one component that handles everything. To make the todo list, you'll:

- Move the data for the todo list into a Vuex datastore.
- Create a new component that creates a new todo item and adds it to the list.
- Create a new component that summarizes the todo list so a user can see how many todos they have left and how many are completed.

Your starting code is a simple todo list in the `TodoList.vue` component. This component is designed to only show todos and allow the user to interact with the todos. All of the data is currently hard-coded.

All of these components need access to the todo list data, but that is why you'll move the data into a Vuex datastore.

> Before you start, make sure to run `npm install` to install any dependencies.

Step One: Move the data to a Vuex datastore

First, you'll move the todo list data to the Vuex datastore. The Vuex datastore acts as a central location to keep all of the application's data and provide access to that data to all components in the application.

Looking at the `TodoList` component now, the data for the list is in the component's data function:

```
export default {
  data() {
    return {
        todos: [
          { name: 'Wake up', done: false, category: 'Home' },
          { name: '5 Minute Morning Movement', done: false, category: 'Home' },
          { name: 'Meditate', done: false, category: 'Home' },
          { name: 'Brush teeth', done: false, category: 'Home' },
          { name: 'Shower', done: false, category: 'Home' },
          { name: 'Answer email', done: false, category: 'Work' },
          { name: 'Stand up meeting', done: false, category: 'Work' },
          { name: 'Fix a bug', done: false, category: 'Work' },
        ]
    }
  }
}
```

To move this from this component to the Vuex datastore, open the file that defines the Vuex datastore. That file is at `src/store/index.js`:

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

```
Vue.use(Vuex)

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

When putting data in a Vuex datastore, the data goes into the `state` section of the datastore.

Remove the `todos` from the `data` property found in the component, and add them to the `state` property in the datastore. After you complete this step, the script block for the component will be empty, as shown below:

```
export default {

}
```

Once you complete this step, the state property in the Vuex store looks like this:

```
state: {
  todos: [
    { name: "Wake up", done: false, category: "Home" },
    { name: "5 Minute Morning Movement", done: false, category: "Home" },
    { name: "Meditate", done: false, category: "Home" },
    { name: "Brush teeth", done: false, category: "Home" },
    { name: "Shower", done: false, category: "Home" },
    { name: "Answer email", done: false, category: "Work" },
    { name: "Stand up meeting", done: false, category: "Work" },
    { name: "Fix a bug", done: false, category: "Work" }
  ]
},
```

Notice that `state` isn't a function, so you don't need to include a `return` statement. `state` is an object, and the values go directly in the curly brackets.

Now that the data is in the store, how do you get it back into your component?

Because the Vuex datastore is connected to the application in the `main.js` file, all the components have access to the datastore from the `this.$store` variable. This can be used in the `v-for` call to get the todo data kept in the datastore:

```
<li v-for="todo in $store.state.todos" v-bind:key="todo.name"  v-bind:class="{
'todo-completed': todo.done }">
    <input type="checkbox" v-model="todo.done"/>
    <span v-bind:class="{ completed: todo.done }">{{todo.name}}</span>
</li>
```

> Remember that in the HTML portion of the component, you don't have to use `this` to access component information. The same is true for referencing `this.$store` in your HTML.

Step Two: Add a component to the todos

Now you'll add a new component to the project.

To do that, create a new file under `src/components` called `NewTodo.vue`. If you type `vue` in that file, a list of snippets appears. Select the first snippet from the list of snippets displayed. This adds the following to the file:

```
<template>

</template>

<script>
export default {

}
</script>

<style>

</style>
```

Next, add a form that accepts a new todo name and a category:

```
<template>
  <section class="new-todo">
    <form>
      <input type="text" placeholder="Name">
      <select>
        <option value="">--- Select a category ---</option>
        <option value="Home">Home</option>
        <option value="Work">Work</option>
      </select>
      <button>Add</button>
    </form>
  </section>
</template>
```

Add the following CSS to make the form look better:

```
.new-todo {
    width:450px;
    background: #fff;
    margin: auto;
    font-family: 'Roboto Condensed', sans-serif;
    border-radius: 10px;
}
input, select, button {
  padding: 5px 5px;
  margin: 5px;
}
```

You can see with the form that there are two form elements: the input box for the name and the select for the category. Make a new data property to store those values:

```
export default {
  data() {
    return {
      newTodo: {
        name: '',
        category: '',
        done: false
      }
    }
  }
}
```

Next, add them to the HTML elements. Try doing this on your own before looking at the solution:

```
<input type="text" placeholder="Name" v-model="newTodo.name">
<select v-model="newTodo.category">
```

Now that the data is being captured, you need a way to save a new todo to the `todos` list in the datastore. Remember that the new component already has access to the Vuex datastore because all components automatically have access through the `this.$store` variable.

But adding data to a Vuex datastore isn't as straightforward as adding a line to the array. Changes to data should go through a Mutation.

Step Three: Add a mutation to the datastore

Open the `src/store/index.js` file. The store has the `todos` list. Now add a new mutation called `ADD_NEW_TODO()` that takes a todo object and adds it to the array:

```
mutations: {
  ADD_NEW_TODO(state, todo) {
    state.todos.push(todo);
  }
},
```

Now you can call this mutation from the `NewTodo` component. Remember that mutations aren't called like methods. They're committed to the store, much like a change in `git` is committed to the repository.

Next, add a new method called `saveTodo` to the `NewTodo` component that takes the entered todo, gives it a `done` status of `false` and then commits it to the datastore. It can then clear the data for the next todo to be added:

```
methods: {
  saveTodo() {
    this.newTodo.done = false;
    this.$store.commit('ADD_NEW_TODO', this.newTodo);
    this.newTodo = {
      name: '',
      category: '',
      done: false
    };
  }
}
```

Configure the form to call this method. Try this on your own first before looking at the answer below:

```
<form v-on:submit.prevent="saveTodo">
```

Now add the `NewTodo` component to the `App.vue` component. Open the `src/App.vue` file and in the JavaScript section, change the following:

```
import TodoList from './components/TodoList.vue';
import NewTodo from './components/NewTodo.vue';

export default {
  name: 'app',
  components: {
    TodoList,
    NewTodo
  }
}
```

Afterwards, add the component's HTML tag to the `App` component's HTML:

```
<div id="app">
  <todo-list />
  <new-todo />
</div>
```

Viewing the application now, you see the form underneath the todo list. When you fill out and submit the form, a new task is added to the list in the datastore, and the new task is displayed on the page with the other todos.

Step Four: Recreate checkbox handling in TodoList

There's a problem with the application.

Since data shouldn't be changed in the Vuex store outside of a mutation, `v-model` can no longer be used on data coming from a Vuex store. You'll notice that if you run this code in the browser and use any of the checkboxes, errors appear in the console even though the checkboxes still work.

Every change to Vuex data should go through a mutation, so the `TodoList` needs to change. To correct this, `v-model` is removed from the checkbox and replaced with `v-on:click`. When the checkbox is clicked, a method is called that commits a new mutation called `FLIP_DONE` that changes the value of the todo.

First, change the `TodoList` to change how the checkbox is handled:

```
<input type="checkbox" v-bind:checked="todo.done" v-
on:click="checkTodoBox(todo)"/>
```

This binds the status of the checkbox to the `todo.done` value from Vuex, but binds the action of the change to a method called `checkTodoBox()`. Now, write that method to send status changes to Vuex:

```
methods: {
    checkTodoBox(todo) {
        this.$store.commit('FLIP_DONE', todo);
    }
}
```

Then, in the `store/index.js` file, you can add the new mutation:

```
mutations: {
  ADD_NEW_TODO(state, todo) {
    state.todos.push(todo);
  },
  FLIP_DONE(state, todo) {
    todo.done = ! todo.done;
  }
},
```

Since the `todo` is already in the `state`, it can be referenced directly. And since its value is being changed in a mutation, the rules of Vuex are being followed when updating the value.

You see in the interface that everything works as it did before, but now an error won't show in the console because you're following the rule that states that all data changes should happen in a mutation.

Step Five: Add a component to show summary information

Now create a new component called `TodoSummary.vue` in the `src/components` folder. This component shows summary information about your list.

First, set up two boxes at the top to hold information for your "Home" todos and your "Work" todos:

```
<template>
  <section class="todo-summary">
      <div>
          <h3>Home</h3>
          <p>{{ completedHomeTodos }} / {{ totalHomeTodos }}</p>
      </div>
      <div>
          <h3>Work</h3>
          <p>{{ completedWorkTodos }} / {{ totalWorkTodos }}</p>
      </div>
  </section>
</template>
```

The following CSS can be used to make these boxes better fit with the layout:

```
<style>
.todo-summary {
    width:600px;
    background: #fff;
    margin: auto;
    font-family: 'Roboto Condensed', sans-serif;
    border-radius: 10px;
    display: flex;
    justify-content:space-evenly;
}

.todo-summary div {
    border: 1px black solid;
    padding: 20px;
    border-radius: 5px;
    display: inline-block;
    text-align: center;
    width: 40%;
}

.todo-summary div p {
```

```
    font-size: 2em;
    margin: 3px;
}
</style>
```

Then add new computed properties, using the state from the datastore, to calculate how many of each todo category are left to be completed.

Try to complete this without looking at the solutions below and then check your solution against the included one:

```
computed: {
    totalHomeTodos() {
        return this.$store.state.todos.filter((todo) => {
            return todo.category === 'Home';
        }).length;
    },
    completedHomeTodos() {
        return this.$store.state.todos.filter((todo) => {
            return todo.done === true && todo.category === 'Home';
        }).length;
    },
    totalWorkTodos() {
        return this.$store.state.todos.filter((todo) => {
            return todo.category === 'Work';
        }).length;
    },
    completedWorkTodos() {
        return this.$store.state.todos.filter((todo) => {
            return todo.done === true && todo.category === 'Work';
        }).length;
    },
}
```

Then add the `TodoSummary` component to the `App` component:

```
<template>
  <div id="app">
    <todo-summary />
    <todo-list />
    <new-todo />
  </div>
</template>

<script>
import TodoList from './components/TodoList.vue';
import NewTodo from './components/NewTodo.vue';
import TodoSummary from './components/TodoSummary.vue';
```

```
export default {
  name: 'app',
  components: {
    TodoList,
    NewTodo,
    TodoSummary
  }
}
</script>
```

You'll now see auto-updating summary information above the todo list.

Summary

In this tutorial, you learned how to:

- Implement an application that utilizes components that work together
- Access data stored in Vuex from multiple components
- Modify data stored in Vuex using mutations
- Use data stored in Vuex within computed properties