

Simple To Do App with Vue 3.x

Objective: Write a simple To Do app using Vue 3.x

Disclosure: This tutorial is based on the To Do app here <http://iamkumaran.github.io/vue-js/vue-js-todo-app-part-1.html> – but that one uses Vue 2.x.

1. I've updated it to use Vue 3.
2. I've also changed the names of several variables for clarity. You may want to review the original Vue 2.x tutorial for additional notes and explanation.
3. I've left out a few things not needed for the tutorial, such as the unique numerical ID of each item and the use of `:key`.
4. I've opted to *not* use Lodash for array manipulation and instead 'brute force' the deletion of an item at the end of the tutorial. This took me one extra line of code compared to importing the entire Lodash library.
5. Some of the above won't mean anything unless you go through the original Vue 2.x tutorial.

HTML and CSS

Take a moment to review the code for the starter HTML and CSS files and preview the HTML file in your browser. Notice Vue is already linked to in the HEAD.

JavaScript

Create a JS file as per the starter HTML code. You may want to code a temporary `alert()` or `console.log()` to ensure the JS file is properly linked.

v-text

Objective: Display element text

1. JS. Create a constant to describe options used by the app.

```
const SIMPLE_TODO_APP = {  
  data() {  
    return {  
      appTitle: 'Simple To Do App'  
    };  
  }  
};
```

2. JS. Create the app, mount it on the page, and refer to it as a constant for later use.

```
const APP = Vue.createApp(SIMPLE_TODO_APP).mount('#todo-app-container');
```

3. HTML. Render the `appTitle` property as the H1 text.

```
<section id="todo-app-container">  
  <h1 v-text="appTitle"></h1>  
  <form name="todo-form" method="post" action=""  
    <button type="button" v-text="add" v-on:click="addTask">
```

4. Test the page. It should look like this:

Simple To Do App

My Tasks

-

v-model

Objective: Create a two-way binding for the input field so that whatever is typed automatically updates Vue's data object.

1. JS. Edit the data object to include a new property. Initialize it as blank.

```
return {  
  appTitle: 'Simple To Do App',  
  itemToAdd: ''  
};
```

2. HTML. Add the same property name to the HTML to create the binding.

```
<form name="todo-form" method="post" action="">  
  <input name="add-todo" type="text" v-model="itemToAdd">  
  <button type="submit">Add</button>
```

3. HTML. Sanity check. Add {{ itemToAdd }} anywhere on the page (within the mount point). I've added it above the H1 (inside the app container/mount point). Type anything into the input field and the page will update in real-time as you type. Be careful to not click the Add button!

```
<section id="todo-app-container">  
  {{ itemToAdd }}  
  <h1 v-text="appTitle"></h1>  
  <form name="todo-form" method="post" action="">
```

I am typing something

2

Simple To Do App

1

I am typing something

Add

My Tasks

•

4. HTML. Delete {{ itemToAdd }} from above the H1.

Methods

Objective: Create a method (function) to add a new task item to the array.

1. JS. Add another property – an empty array to hold the To Do items.

```
data: {  
  return {  
    appTitle: 'Simple To Do App',  
    itemToAdd: '',  
    listOfToDoItems: []  
  };  
}
```

2. JS. Create a block for the data object's methods. The `methods` block should be the same level as the `data` block. So we'll need a comma after `data`'s closing curly brace.

```
    itemToAdd: '',  
    listOfToDoItems: []  
  };  
},  
  methods: {  
  }  
};
```

3. JS. Add a method named `addItem`, which is an anonymous function.

```
  },  
  methods: {  
    addItem: function () {  
    }  
  }  
};
```

4. JS. The anonymous function should append (push) an item to the end of the array. Remember that in this case, `this` refers to the object to which the method belongs (the `data` object).

```
methods: {  
  addItem: function () {  
    this.listOfToDoItems.push();  
  }  
}
```

5. JS. Push an object to the array. The object (representing a To Do item) has three properties. A unique numerical id, the item text, and a flag to tell if this task has been completed or not. Be sure to wrap the three properties inside curly braces.

```
methods: {  
  addItem: function () {  
    this.listOfToDoItems.push({  
      title: this.itemToAdd,  
      isComplete: false  
    });  
  }  
}
```

6. JS. Clear out the input field after adding an item. DO this once the push is complete.

```
methods: {  
  addItem: function () {  
    this.listOfToDoItems.push({  
      title: this.itemToAdd,  
      isComplete: false  
    });  
    this.itemToAdd = '';  
  }  
}
```

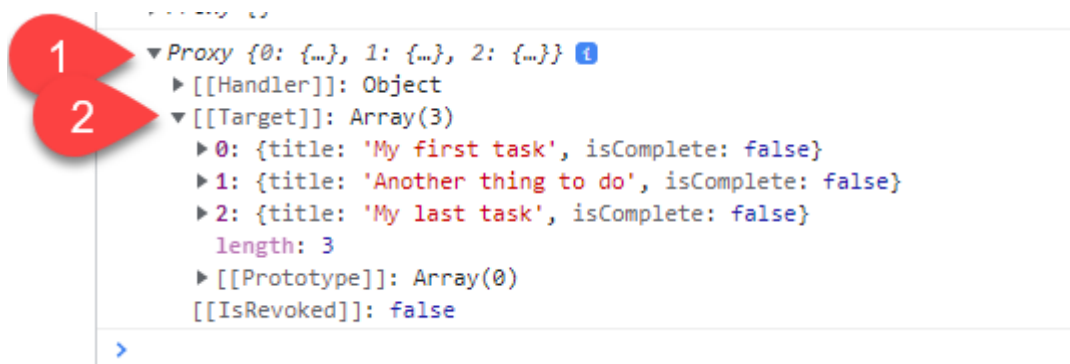
7. JS. Sanity check. After the push is complete and the input field is cleared, log the array.

```
    isComplete: false
  });
  this.itemToAdd = '';
  console.log(this.listOfToDoItems);
}
}
```

8. HTML. Edit the HTML to prevent the default form behavior (the Vue way to do `e.preventDefault()`) and instead run the `addItem` method.

```
<h1 v-text= "appTitle" ></h1>
<form name="todo-form" method="post" action="" @submit.prevent="addItem">
  <input name="add-todo" type="text" v-model="itemToAdd">
```

9. Type something into the input field and click the **Add** button. Do this a total of three times so that three items get saved to the array. Nothing displays on the page yet because we haven't coded that. But you can examine the *console log*. Expand **Proxy** and then expand **Target** and you should see your three tasks.



```
▼ Proxy {0: {…}, 1: {…}, 2: {…}} ⓘ
  ► [[Handler]]: Object
  ▼ [[Target]]: Array(3)
    ► 0: {title: 'My first task', isComplete: false}
    ► 1: {title: 'Another thing to do', isComplete: false}
    ► 2: {title: 'My last task', isComplete: false}
    length: 3
    ► [[Prototype]]: Array(0)
    [[IsRevoked]]: false
```

10. JS. Delete the `console.log` from the method.

v-for

Objective: Loop through the array of To Do items and render them on the page.

1. HTML. Edit the LI to repeat via `v-for`. This must be in the format *item in data-array* where *data-array* is the source data and *item* is the current array item. At the same time, extract the `title` property of the current item and use it as the text.

```
<ul>
  <li v-for="item in listOfToDoItems" v-text="item.title"></li>
</ul>
```

2. Test the page. Type in a few items and you should see them rendered on the page. While you're at it, leave the input field blank and click **Add**. Notice blank values still get added. We'll fix that next.

Simple Todo App

My Tasks

- Finish this tutorial
- Eat my vegetables
-
- Clean my room

v-bind

Objective: Validate against blank submissions.

1. JS. Add another property to flag errors.

```
const SIMPLE_TODO_APP = {  
  data() {  
    return {  
      appTitle: 'Simple Todo App',  
      itemToAdd: '',  
      listOfToDoItems: [],  
      hasError: false  
    }  
  };  
};
```

2. Within the `addItem` anonymous function, and before the array push, add a conditional that tests if the input field is blank. If it's blank, set the `hasError` property we just defined to `true` and exit the function. Otherwise, set it to `false` and let the function continue.

```
methods: {  
  addItem: function () {  
    if (!this.itemToAdd) {  
      this.hasError = true;  
      return;  
    } else {  
      this.hasError = false;  
    }  
  
    this.listOfToDoItems.push({  
      id: this.listOfToDoItems.length + 1,  
      text: this.itemToAdd  
    });  
  }  
}
```

3. HTML. Depending on the state of `hasError`, add `class="error"` to the input field or not.

```
<form name="todo-form" method="post" action="" @submit.prevent="addItem">  
  <input name="add-todo" type="text" v-model="itemToAdd" v-bind:class="{error: hasError}">  
  <button type="submit">Add</button>  
</form>
```

4. Test this. Type a few items and click **Add**. Then, leave the field blank and click **Add**. A blank item should not get added and the input field should display a red border. (Because of the starter CSS.) You can also examine the `class` of the input field in the Elements panel of the dev console.

Edit an Item

This section uses `event.target`, which you may remember from WEB-215's coverage of event bubbling and event delegation. That was part of the modal dialog box project. It also uses `$event`, which is a special variable in Vue.

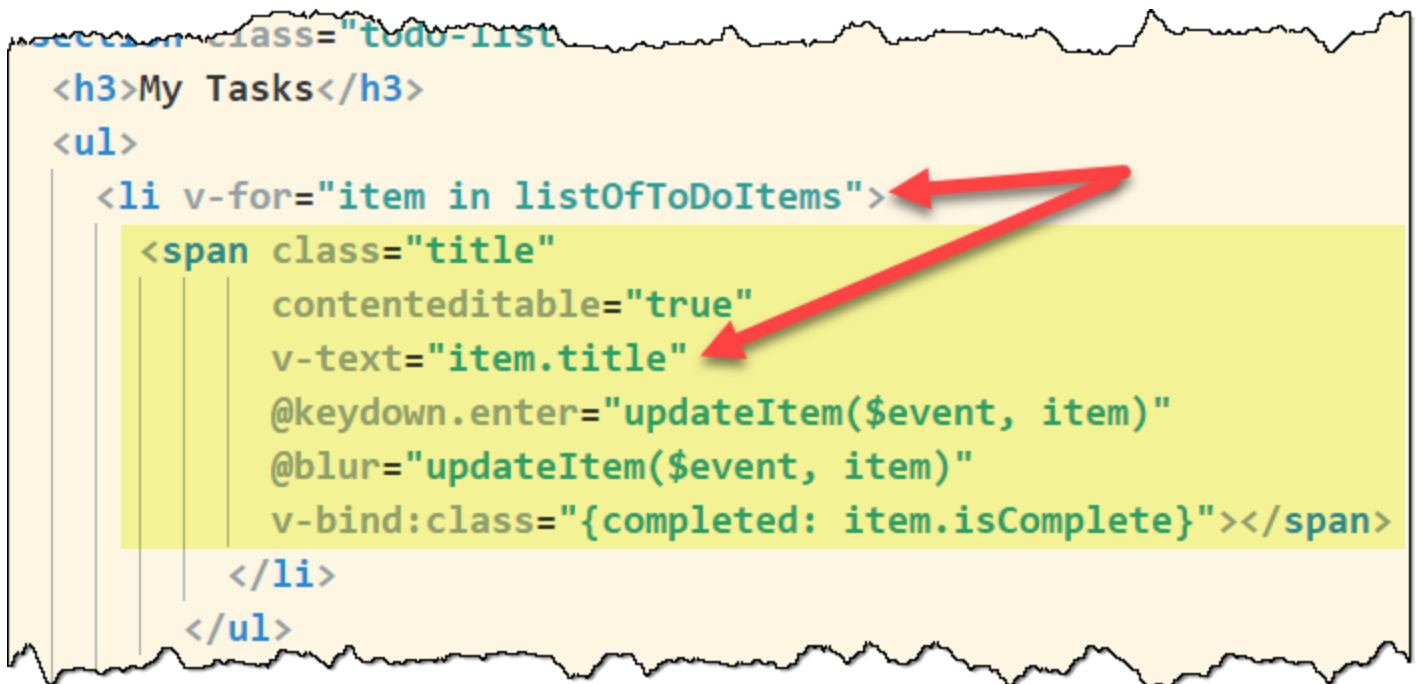
Overview:

To make an item editable, we need to have its text wrapped in a child element.

- A. Currently, we're using `v-text` on the `LI` tag. That overrides any text from children, so we'll need to create a child of `LI` and move `v-text` to that child.
- B. While we're at it, we'll use `contenteditable` to make the text editable as it's easier than swapping in a text field.
- C. We'll add event listeners for pressing the [enter] key and for moving away from the editable content (blur). When these events fire, a new function, `updateItem`, is called.
- D. We'll keep track of the `isComplete` state for future use, using `v-bind` to add a class just like we did with the previous error validation. If a task is complete, it gets `class="complete"`. But – we have any code yet to mark an item as complete. That will come later.

1. HTML.

- a. Delete `v-text` from the `LI`.
- b. Create a child `SPAN`.
- c. Load the `SPAN` up with the following attributes. I've placed each on their own line for readability.
- d. Note that the event listeners pass two arguments: the event, the current item.
- e. Note that `v-bind` adds `class="completed"` to the `SPAN` if the item's `isComplete` property is `TRUE`. We won't notice that until the next section. It's added to the code here because we're thinking ahead.



The image shows a snippet of Vue.js HTML template code. The code is as follows:

```
<h3>My Tasks</h3>
<ul>
  <li v-for="item in listOfToDoItems">
    <span class="title"
      contenteditable="true"
      v-text="item.title"
      @keydown.enter="updateItem($event, item)"
      @blur="updateItem($event, item)"
      v-bind:class="{completed: item.isComplete}"></span>
  </li>
</ul>
```

Two red arrows point from the right side of the image to the `` and `` tags, indicating the focus of the changes described in the text.

- JS. Add a new method (after `addItem`) to perform the update.

```

methods: {
  addItem: function () {
    if (!this.itemToAdd) {
      this.hasError = true;
      return;
    }
    // ...
  });
  this.itemToAdd = '';
},
updateItem: function (e, theItem) {
  e.preventDefault();
  theItem.title = e.target.innerText;
  e.target.blur();
}
}
};

```

3. Test it. Add a few items. Click one and you should be able to edit it. Your edit is saved when you either click out of the editable text or when you tap [enter].

Mark as Complete

This section uses a checkbox as the trigger allowing users to toggle an item as complete or not. The starter CSS file already has rules to style this using the `.completed` class.

1. **HTML.** Add a checkbox before the SPAN tag within the LI.
 - a. `@change` = when the checkbox is changed, run a function named `completeTask` and send it the `items` array as an argument
 - b. `v-bind` = If the item's `isComplete` property is `TRUE`, check the checkbox. If it's `FALSE`, clear the checkbox.

```
</ns>myTasks</ns>
<ul>
  <li v-for="item in listOfToDoItems">
    <input type="checkbox" @change="completeTask(item)"
      v-bind:checked="item.isComplete">
    <span class="title"
      contenteditable="true">
```

2. JS. Add a new method after `updateItem` to simply toggle the state of `isComplete`.

```
},  
updateItem: function (e, theItem) {  
  e.preventDefault();  
  theItem.title = e.target.innerText;  
  e.target.blur();  
},  
completeTask: function (theItem) {  
  theItem.isComplete = !theItem.isComplete;  
}  
}  
};
```

3. Test it. Add a few items. Check a checkbox and the item displays crossed out. Uncheck a box and the item is no longer crossed out.

Delete

Finally, we'll delete an item.

1. HTML. Add an X (the `×` character code) after the item within the LI.
 1. Give it a class of `remove` so it gets styled by our starter CSS.
 2. On click, run the function `removeTask`, passing it the item to be removed.

```
<span class="title"  
  contenteditable="true"  
  v-text="item.title"  
  @keydown.enter="updateItem($event, item)"  
  @blur="updateItem($event, item)"  
  v-bind:class="{completed: item.isComplete}"></span>  
<span class="remove" @click="removeTask(item)">&times;</span>  
</li>  
</ul>
```

2. JS. Add a new method after `completeTask` to delete the item.

```
e.target.blur(),
},
completeTask: function (theItem) {
  theItem.isComplete = !theItem.isComplete;
},
removeTask: function (theItem) {
  var index = this.listOfToDoItems.indexOf(theItem);
  this.listOfToDoItems.splice(index, 1);
}
}
};
```

- a) `var index` gets the position within the array of the item to delete.
- b) The next line uses `splice()` to delete 1 member from the array beginning at the index location just found. The syntax is `splice(startLocation, deleteCount)`. In this case, our deletion starts at the index of the item we want to delete and deletes 1 array member.

3. Test it. Create a few items. Then click the X 'button' to delete them one at a time.

Improvements

Yes – there are a ton of improvements that can be made. Like using `localStorage` to store the items rather than an array so that the list persists over browser sessions. I'll leave it to you to play with this project on your own.