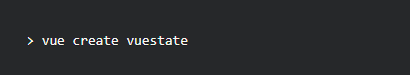
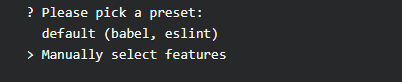
Vuex Simple Workshop

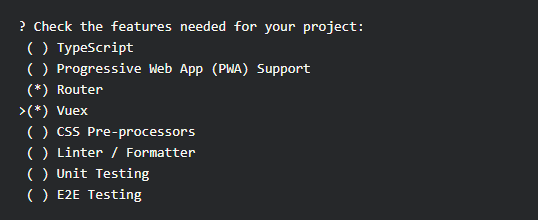
1. We`re going to use the Vue CLI to create our project. At the time of writing this, it`s at version 3.2.1. As long as your Vue CLI is at or beyond 3, you will be able to issue the vue create command to start the project.
2. In your project folder, run the following command:



1. Upon running this, you will be presented with the following prompt:

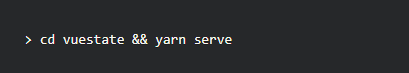


1. We want to manually select features, at which point you want to use your keyboard up and down keys and the spacebar to select the **router** and **vuex** as shown below:



1. Hit enter. It will ask you where you want to store config, and just leave it at *In dedicated config files*. Hit enter, select *n* if you don't want to save this preset, and it will begin installing the new vue app.

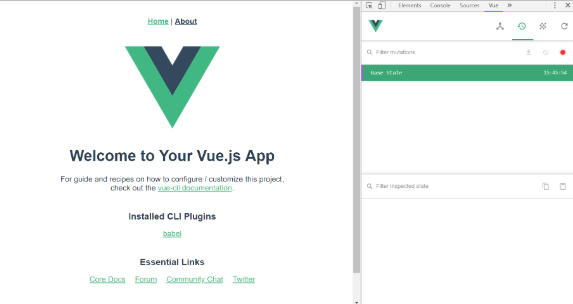
Hop into the folder and serve the app:



1. Great! Visit **http://localhost:8080** in your browser and you're all set.
2. The Vue Devtools Chrome Plugin

Although not a requirement for this tutorial, the [Vue DevTools](https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd) extension for Chrome helps you debug your Vue.js app, and allows us to see what's happening with the state when Vuex is used.

Install the extension and hit CTRL-SHIFT-I (i) and at the very end of the list of tabs, you will see "Vue". Click on it and select the clock icon (Vuex) and you should see something similar as shown below:



## Let`s create our vues. In folder C:\Users\user\Desktop\vuex\vuestate\src\views create Home.vue, About.vue, Books.vue and Contacts.vue.

## In Home, About and Contacts we have to write our html and css. Go to router.js and make the routes for the views like this:

## 

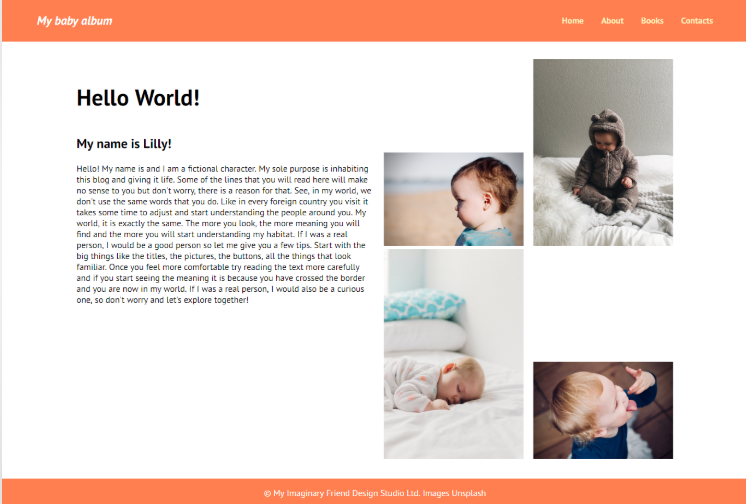
## Make the route for Books.vue too. Go to the file Books.vue and write the following html:

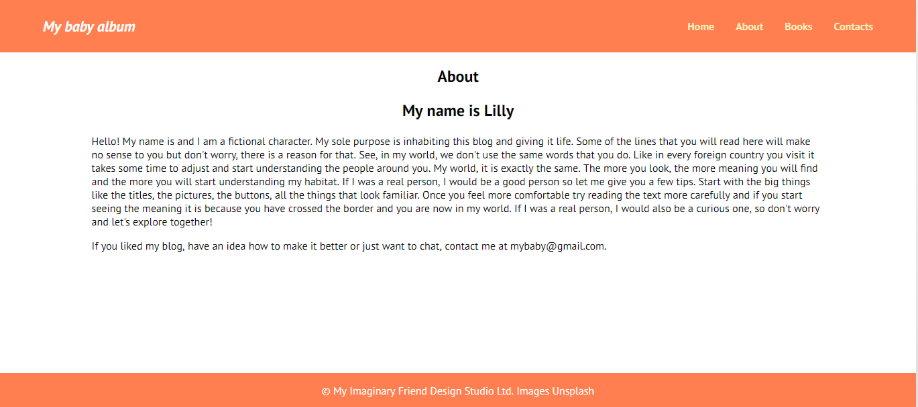
## 

## After that in the script tag import HelloWorld from components folder

## 

The home, about and contacts pages should look like this:





## 

## Defining a State Property. Open up our new Vue project in your code editor and visit the /src/store.js file. This is a file that's generated by the Vue CLI when we selected the Vuex option. This here is essentially Vuex. It's where we will define our data, mutations, actions, getters.

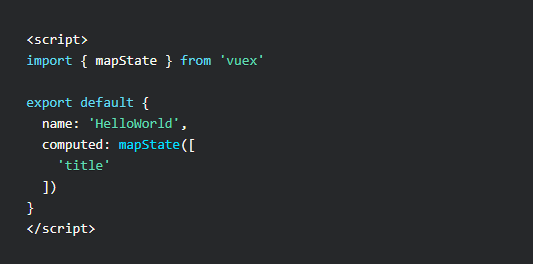
## For now, let's take a look at the State section. Like I mentioned earlier, the state is where your data is defined. So, let's define a property inside of state which will allow us to set a title for our app:

## 

Save this file and visit the **/src/components/HelloWorld.vue**. Adjust the template section as follows:



Next, in the *script* section, adjust it to match the following:



First we're importing **mapState**, which is a helper that allows us to access the state from Vuex. This will give us access to our title property we created earlier.

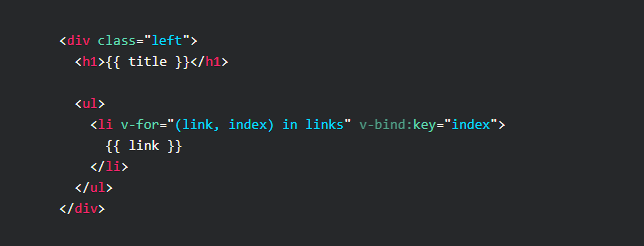
Then, we're adding **computed** and calling mapState and passing in an array with the name of the state property we defined.

This retrieves the value of title and also allows us to reference it by the same name title through interpolation in the template section.

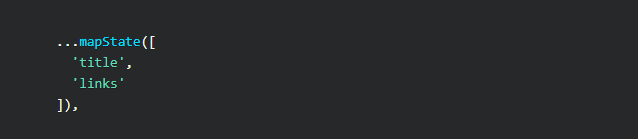
Great! Let's go back and define another state property as an array in **/src/store.js**:



And just in the same way, let's access it through mapState and render this list of links in the template:



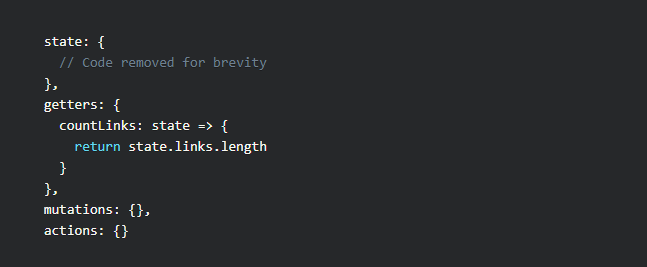
And in the script section:



1. Vuex Getters. To this point, we've been using the mapState helper to directly access the data contained in the state. This is fine when no computations need to be made on the data itself, but when you're dealing with multiple components, you may find yourself duplicating code unnecessarily when transformations need to be made on the data.

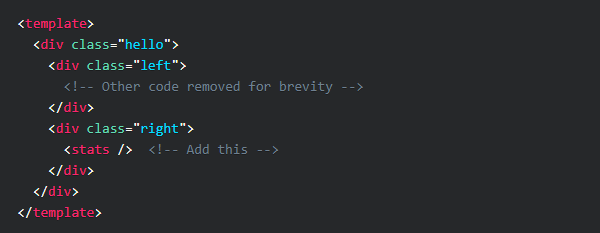
This is where Getters come in handy. Let's say for instance we want to count the number of links in our array. We can use a custom Getter for this.

Visit the /src/store.js and add the following:

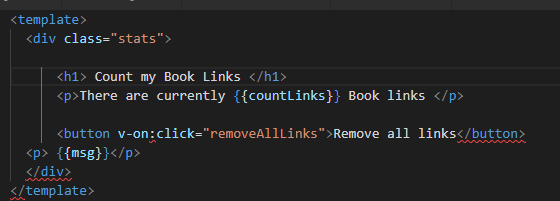


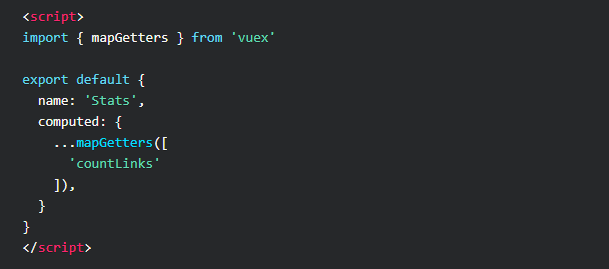
Going back to the original /src/components/HelloWorld.vue file, we could easily access this new getter from within this component, but I want to create another component from which we can access this getter simply for demonstration purposes. The whole point of Vuex is to allow multiple components to access the same state management, so I felt it would be necessary to create another component for this vuex tutorial.

With that said, let's make the following adjustments to our current .vue file:



Next, copy the current HelloWorld.vue file and rename it to Stats.vue. Inside of it, make the following adjustments:

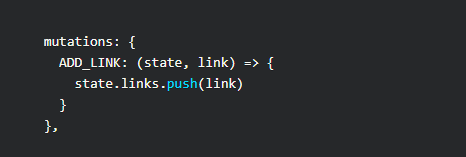




1. Vuex Mutations. To this point, we've only added and displayed data from the state. What about when we want to change the state data? That's where mutations come into play.

In our project, let's say we want to add a new link with a form input field.

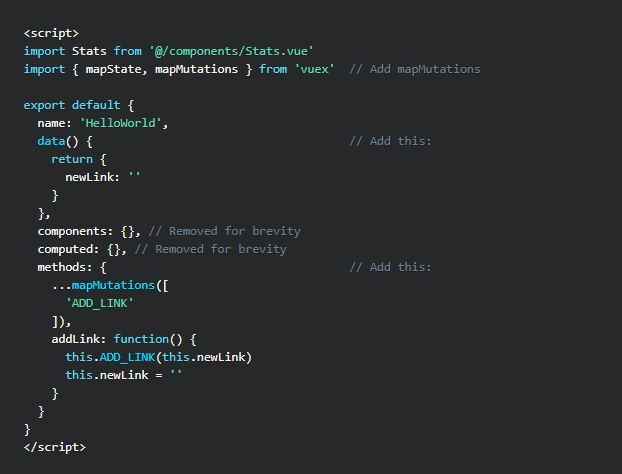
We have to first create a mutation in the /src/store.js file:



We're naming this mutation ADD\_LINK and passing in the state, and a payload named link. link will be the user-submitted value.

Then, we're simply pushing the link value to the links array defined in the state. We're just creating a form that will call a method **addLink** and binding the input class to a property called newLink.

Let's focus on the script section now:

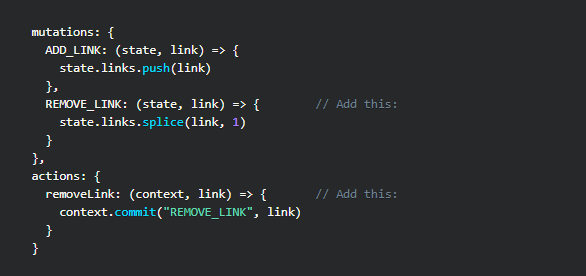


 The part we want to focus on is inside **methods: { }**. We're using the mapMutations helper to import our ADD\_LINK mutation, and then in the addLink() method, we call it and pass in this.newLink.

1. Vuex Actions. Calling mutations directly in the component is for synchronous events. Should you need asyncronous functionality, you use Actions.

This time, we'll use an Action to call upon a mutation that will remove a link.

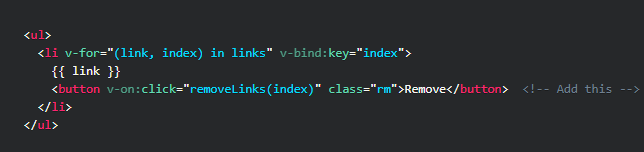
In **/src/store.js** add the following mutation and action:



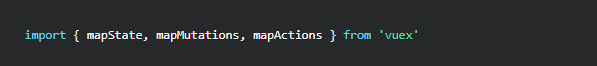
So, nothing special is happening with REMOVE\_LINK, but in actions, we're creating an action named removeLink in which the context (context simply provides you with the same methods and properties on the store instance), and the payload (link index) is passed.

Then we call **context.commit** where we call upon the REMOVE\_LINK mutation and pass to it the link index. This seems redundant, but it's necessary for asynchronous operations.

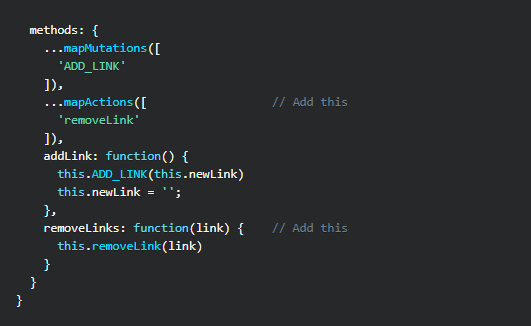
Next, visit HelloWorld.vue and add the following to the template:

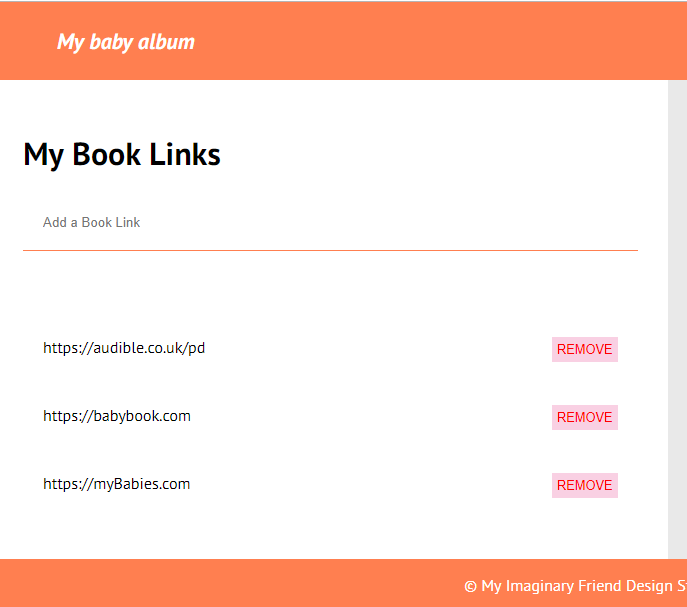


Then, we're going to include mapActions:



And adjust our methods section:

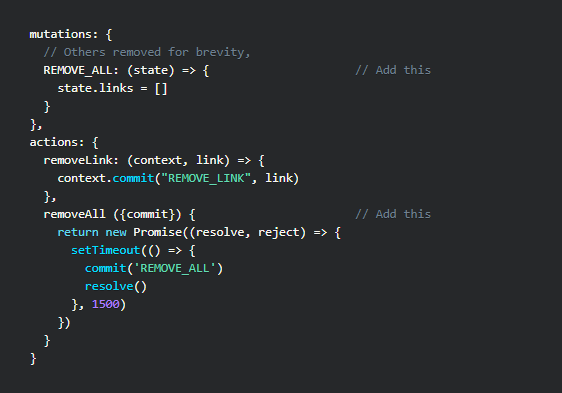


As you can see, it's set up to work in much the same way as calling mutations.Give it a go in the project:

Removing will now work.

This isn't that exciting though, let's try actually demonstrating the value of using Actions to perform asynchronous operations.

Going back to /src/store.js, let's create a new mutation and an action for removing all links with a click of a button:

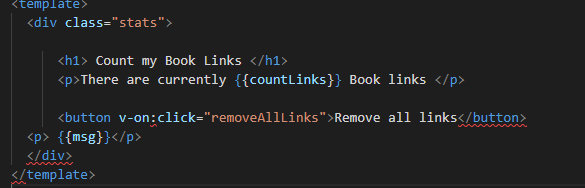


We have a new REMOVE\_ALL mutation that empties out the array.

Then, we have a new removeAll action, and this time we're using argument destructuring to pass in commit, which makes code more simple.

We're creating a promise and simulating an operation that could take time, and calling REMOVE\_ALL after 1.5 seconds.

We'll perform this operation in our **/src/components/Stats.vue** file:



In the script section:



The important part here is this.removeAll().then() - This will only be displayed once the action has been performed.

Try it out in the browser. Click **Remove all links** and in 1.5 seconds, the links will be removed and our message displayed!

