



# Vue Fundamentals – 70

## Managing state with Vuex



Peter Kassenaar –  
[info@kassenaar.com](mailto:info@kassenaar.com)

Peter Kassenaar

INCLUSIEF  
GRATIS  
WEBVERSIE  
VAN HET  
BOEK

Vue.js

Web Development Library



VANDUUREN MEDIA

Chapter 7,  
P. 198 e.v.

# What is State Management?

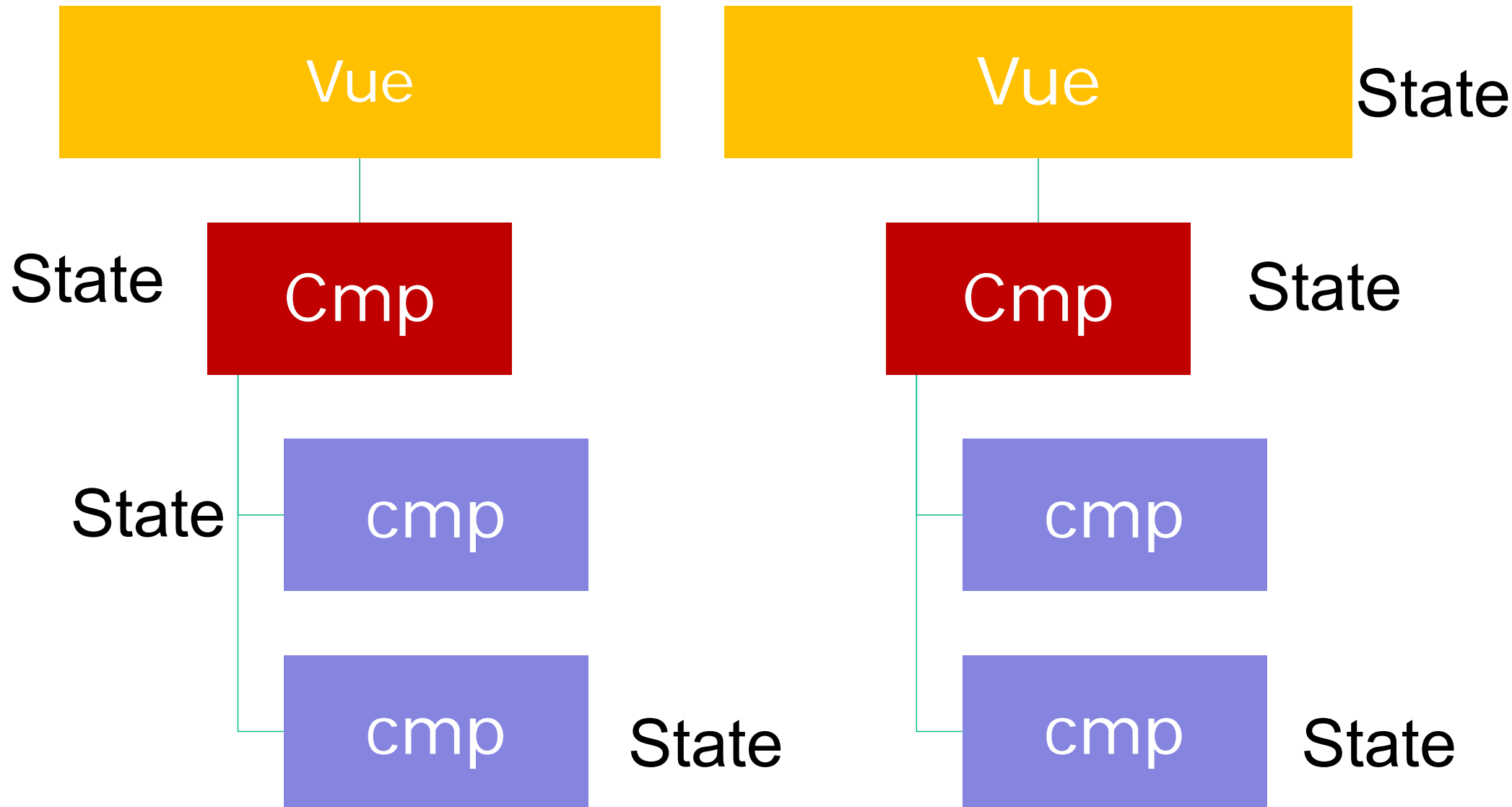
- Various design patterns, used for managing *state* (data in its broadest sense!) in your application.
- Multiple solutions possible – depends on application & framework



**Redux**



# State management without a store



# Benefits of using a store

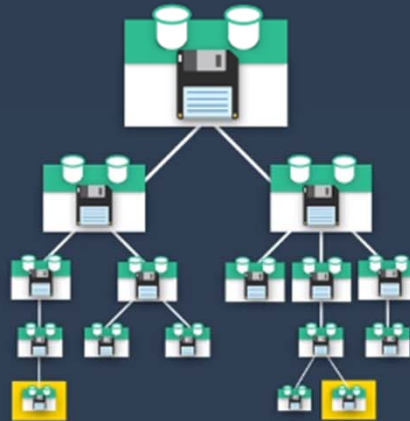
- State is only changed in a controlled way
- Component state is also driven from the store
- Based on immutable objects – b/c they are predictable
- Developer tools available to debug and see how the store changes over time
  - “Time travelling Developer tools”

# Complex data flow – use a Store

What is Vuex?



**Vuex:** State Management Pattern + Library

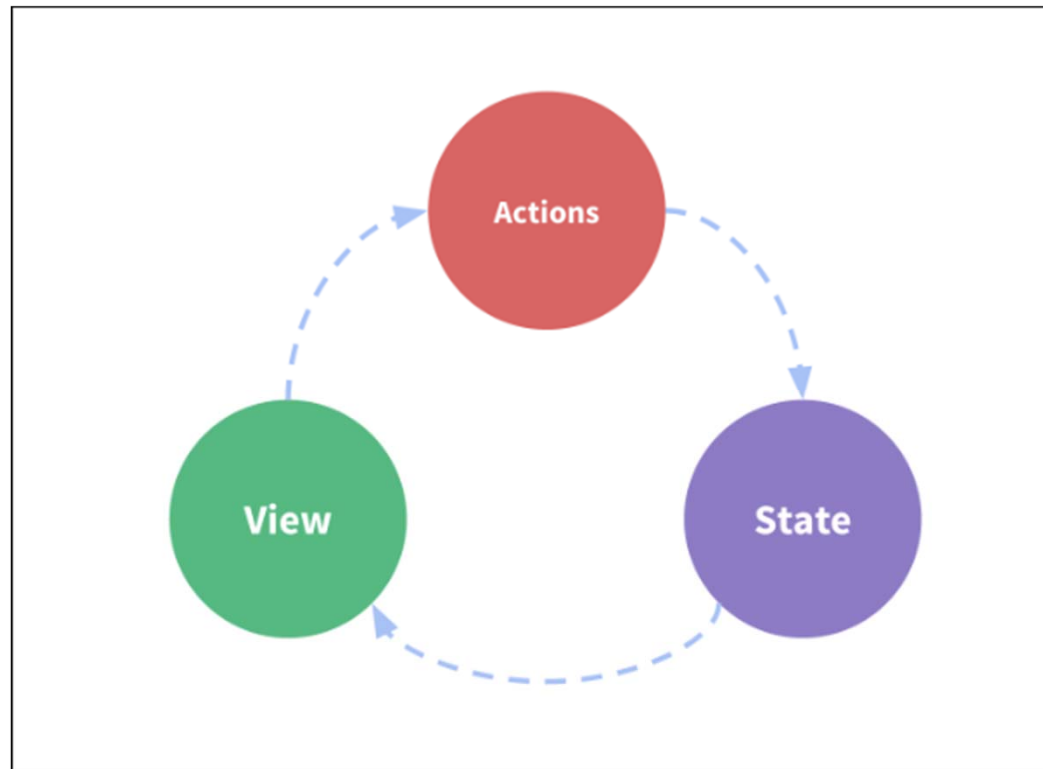


<https://vuex.vuejs.org/>

# One-way data flow

*“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.”*

# Simple one-way data flow

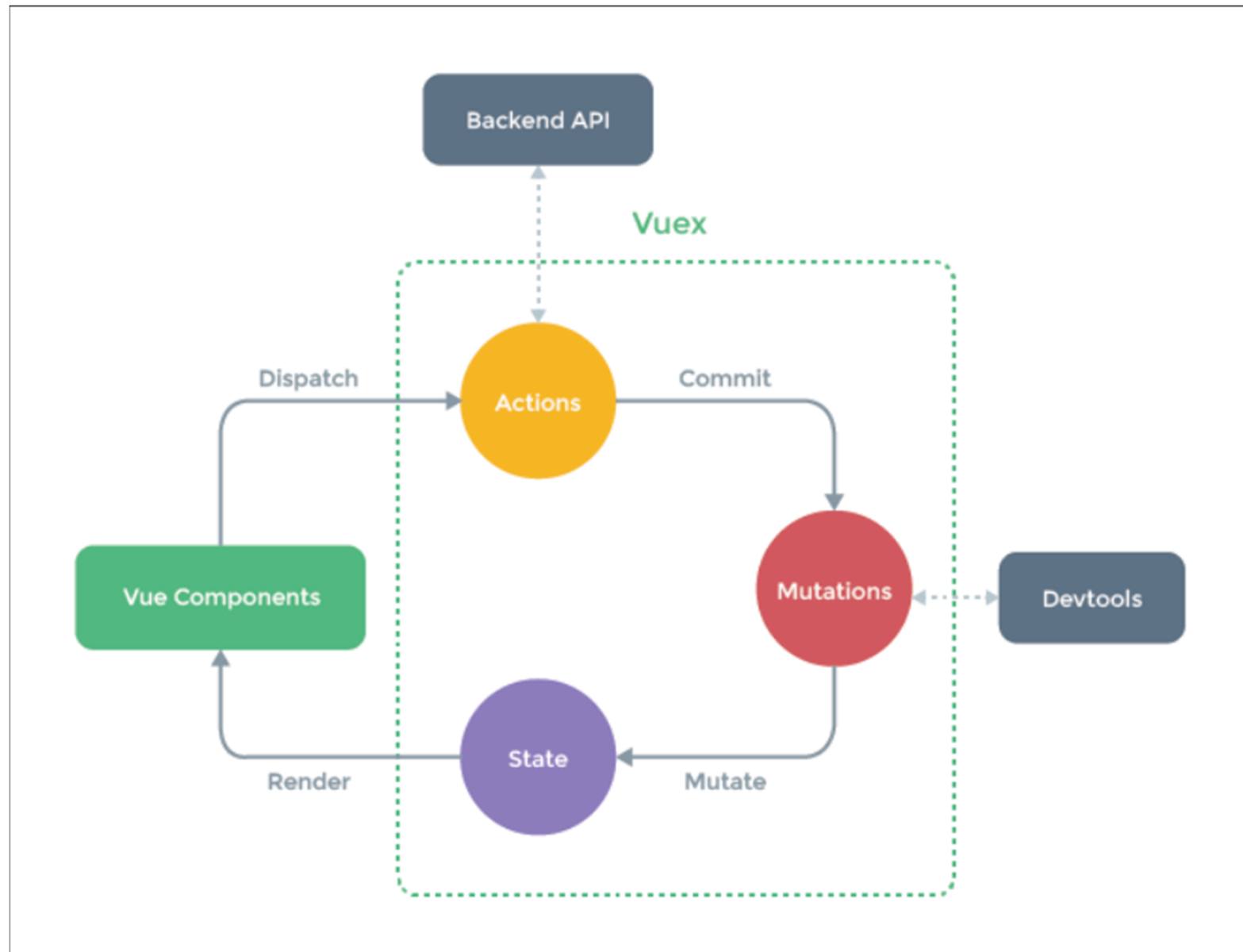




## Real life apps/store – more complex

- Solution: *extract* the shared state out of the components, and manage it in a global singleton
- The component tree becomes one big "view"
  - *any* component can access the state or trigger actions, no matter where they are in the tree
  - State and the store are a *single source of truth* for your application

# Store architecture



# Store concepts

- **Store** – holds a state object. Your single source of truth for the application
- **Mutations** – commit and track state changes
- **Actions** – update the state via mutations. It is bad practice to update the state directly
- **Getters** – access the state, to only get a portion of the state you are interested in
- **Dispatch** – dispatch an action from a vue component

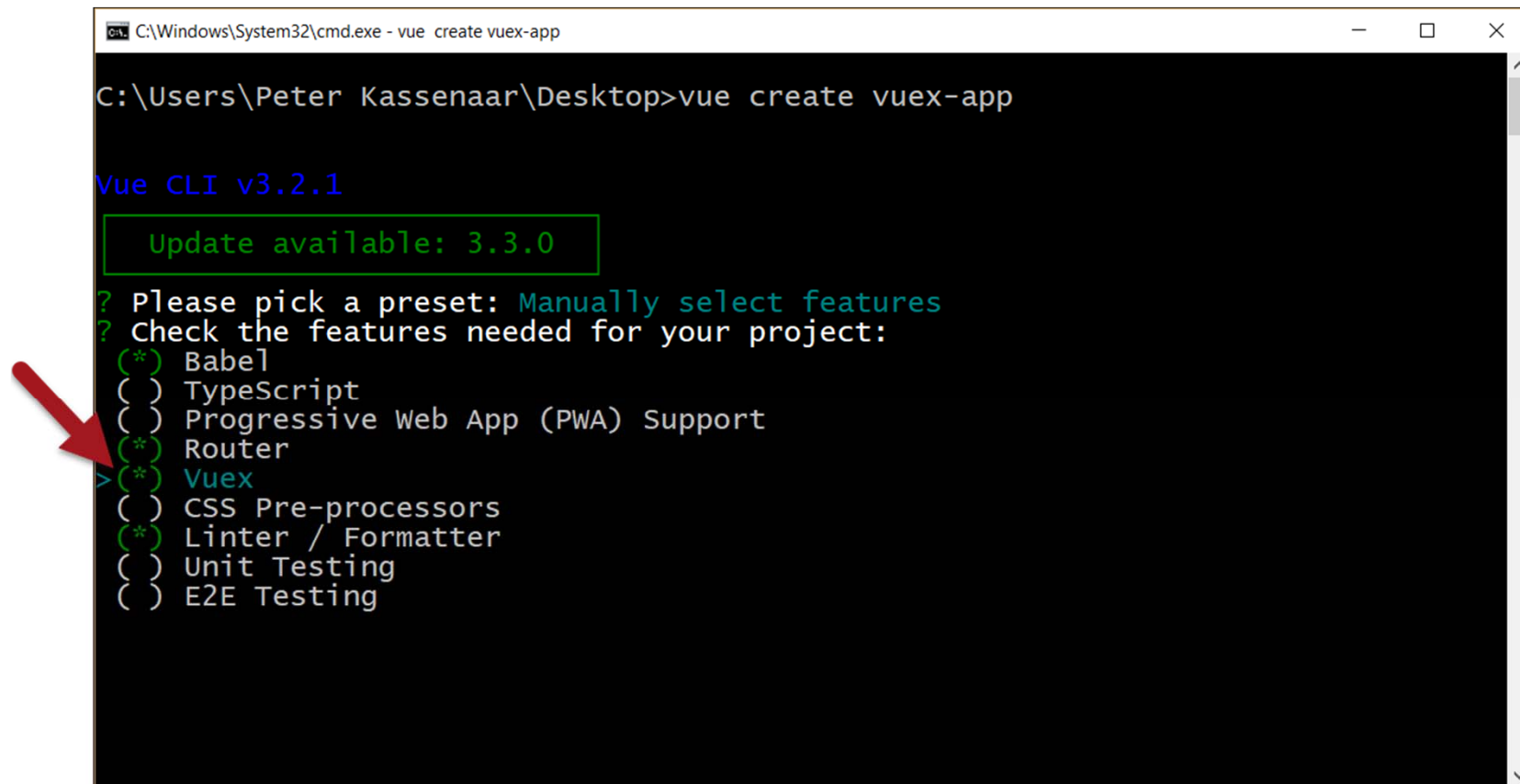


# Let's build a Store

Starting with the 'Hello World'-example for stores....

# Simplest example – a counter

- First : install vuex
  - From the CLI, by choosing Vuex as an option when creating a new project
  - Afterwards, by using `npm install --save vuex`.



```
C:\Windows\System32\cmd.exe - vue create vuex-app

C:\Users\Peter Kassenaar\Desktop>vue create vuex-app

Vue CLI v3.2.1
Update available: 3.3.0
? Please pick a preset: Manually select features
? Check the features needed for your project:
  (*) Babel
  ( ) TypeScript
  ( ) Progressive Web App (PWA) Support
  (*) Router
  > (*) Vuex
  ( ) CSS Pre-processors
  (*) Linter / Formatter
  ( ) Unit Testing
  ( ) E2E Testing
```

# Setting up your store

- Create a folder `../store`
- Create an `index.js` inside that folder
- Again, this is opinion – you can have a `store.js` file anywhere you want

```
// ../store/index.js
import Vue from 'vue';
import Vuex from 'vuex';

Vue.use(Vuex);

export default new Vuex.Store({
})
```

```
// main.js
import Vue from 'vue'
import App from './App.vue'
import store from './store'

Vue.config.productionTip = false;

new Vue({
  render: h => h(App),
  store
}).$mount('#app');
```

Tell `main.js` to use the store

# Creating a simple counter

- Store some data in the store
  - Create a `state` object – and initialize it!
  - Store cannot set properties that have no default value

```
export default new Vuex.Store({  
  state: {  
    counter: 0  
  },  
  ...  
})
```

# Add mutations

Mutations always take an existing `state` object, and the data you want to update (called `status` over here, often called `payload`)

```
export default new Vuex.Store({
  state: {
    counter: 0
  },
  mutations: {
    INCREMENT(state, status) {
      state.counter += status;
    },
    DECREMENT(state, status) {
      state.counter -= status;
    },
    RESET(state) {
      state.counter = 0;
    }
  },
  ...
})
```

You *can* call mutations directly by using `this.$store.commit()`, but it is not considered best practice to do so.



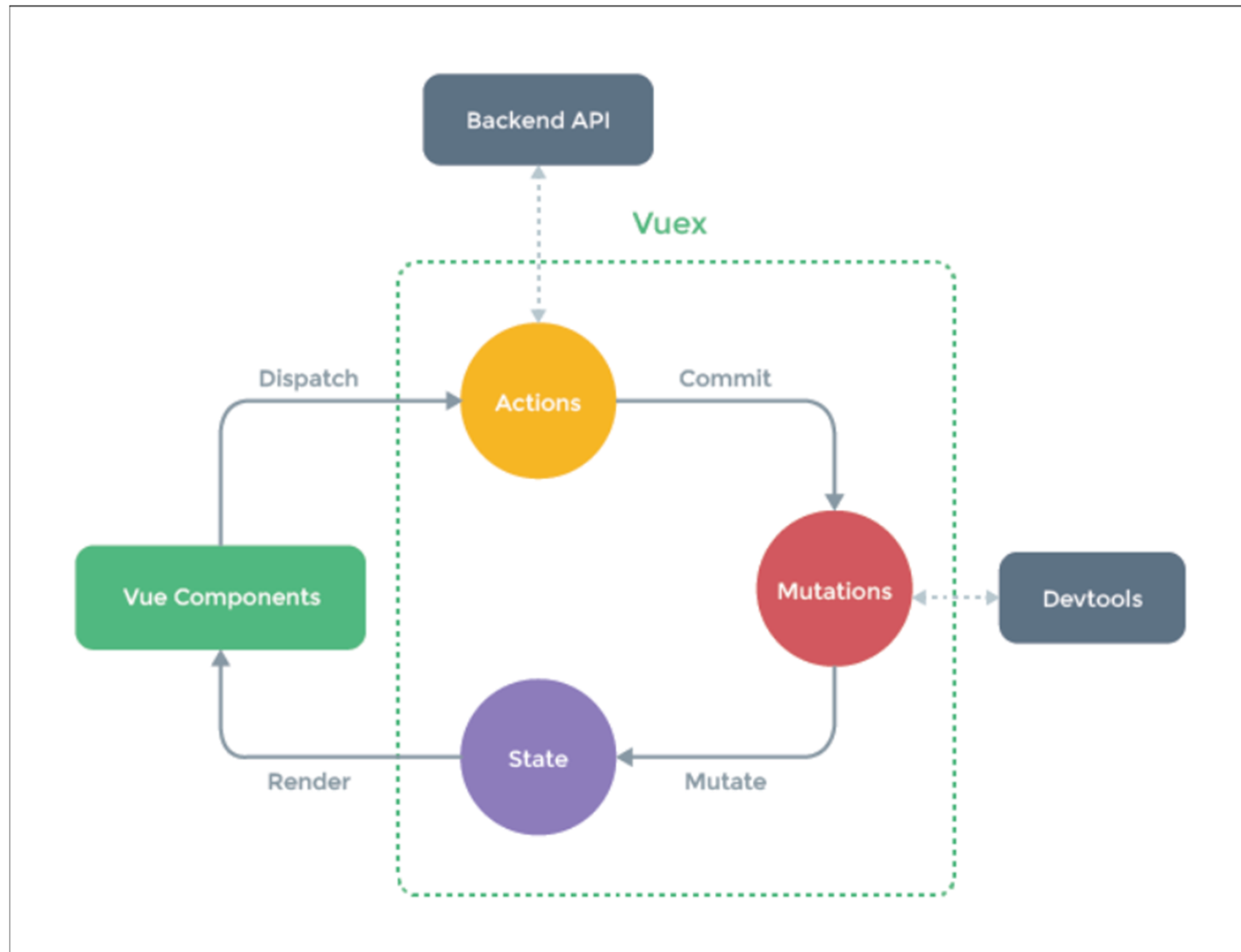
# Add actions

- It is best practice to call mutations through actions, so lets add them

```
export default new Vuex.Store({  
  ...  
  actions:{  
    increment(context, value){  
      context.commit('INCREMENT', value)  
    },  
    decrement(context, value){  
      context.commit('DECREMENT', value)  
    },  
    reset(context){  
      context.commit('RESET')  
    }  
  }  
})
```

In this simple use case our actions have the same name as the mutations, but this is not necessary. You *can* do multiple commits in one action.

# Store architecture



## Last step – create/update component(s)

- Add some simple UI to the component
- Create a computed property on the component to read the current state
  - it is updated as the state changes
- Create methods that call actions on the store
  - In this example: `increment()`, `decrement()` and `reset()`

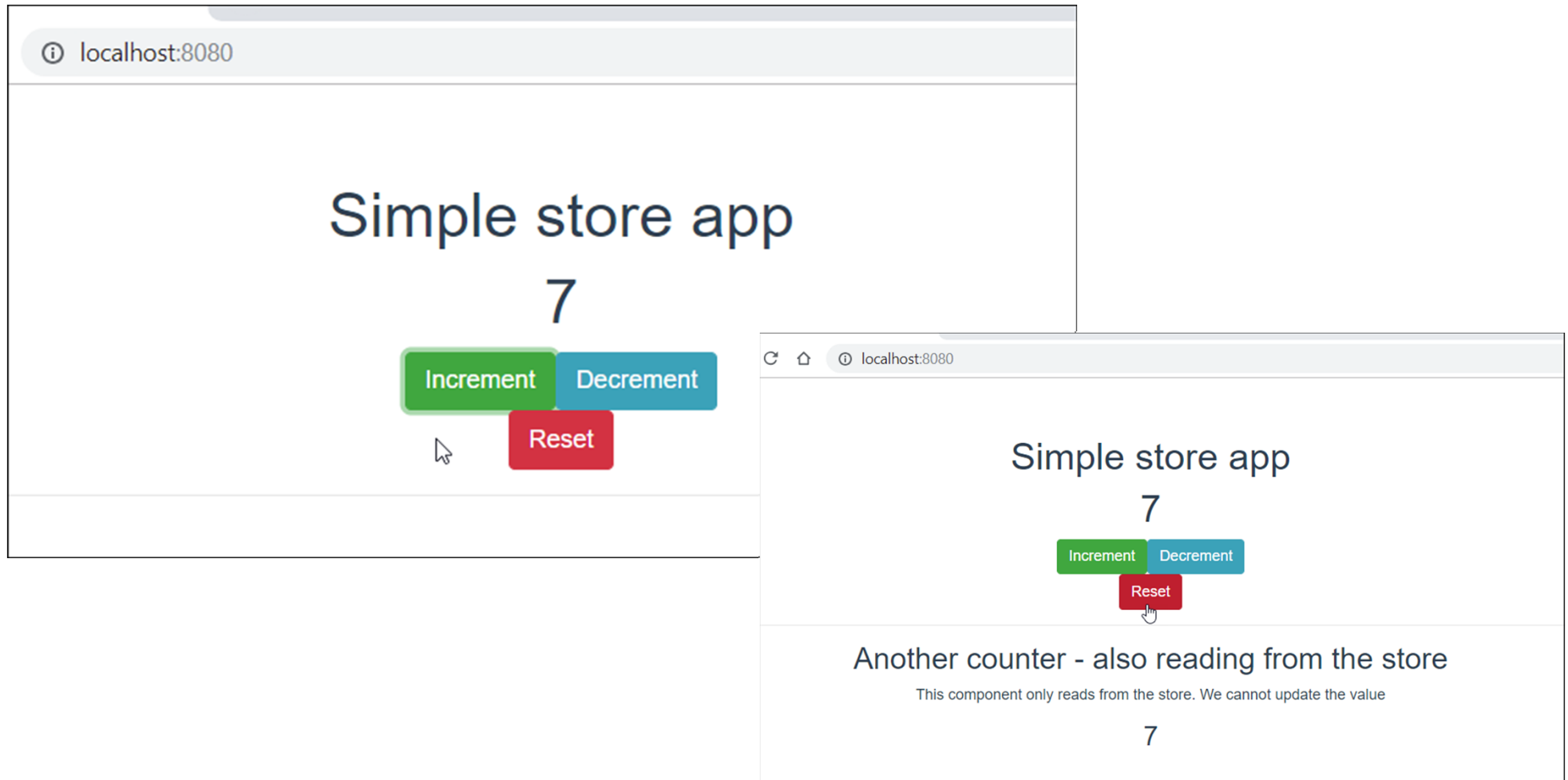
```
<div>  
  <button @click="increment()" class="btn btn-success">Increment</button>  
  <button @click="decrement()" class="btn btn-info">Decrement</button>  
  <br>  
  <button @click="reset()" class="btn btn-danger">Reset</button>  
</div>
```

# Call `this.$store.dispatch` on the component

```
<script>
  export default {
    name: "Counter",

    methods: {
      increment() {
        this.$store.dispatch('increment', 1)
      },
      decrement() {
        this.$store.dispatch('decrement', 1)
      },
      reset(){
        this.$store.dispatch('reset')
      }
    },
    computed:{
      counter(){
        return this.$store.state.counter;
      }
    }
  }
</script>
```

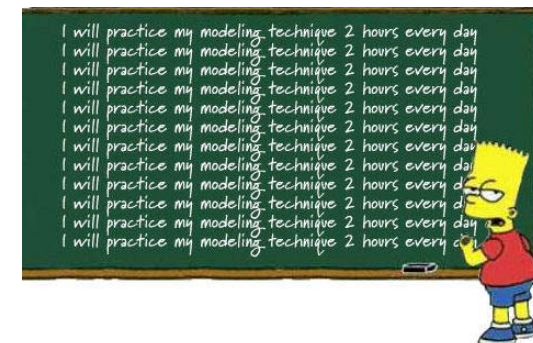
# Result



And after adding another component,  
also reading the state from the store

# Workshop

- Start a new application from scratch, using the CLI or add Vuex to an existing app
- Implement the `counter` example, shown in the previous slides
- Add a `state-variabele` message to the store:
  - Users can set the message via a textbox
  - The message is shown in the component and in another component
- Example: [../400-vuex-basics](#)



# Don't "Vuex" everything!

## ***"Components Can Still Have Local State -***

*Using Vuex doesn't mean you should put **all** the state in Vuex. If a piece of state strictly belongs to a single component, it could be just fine leaving it as local state."*

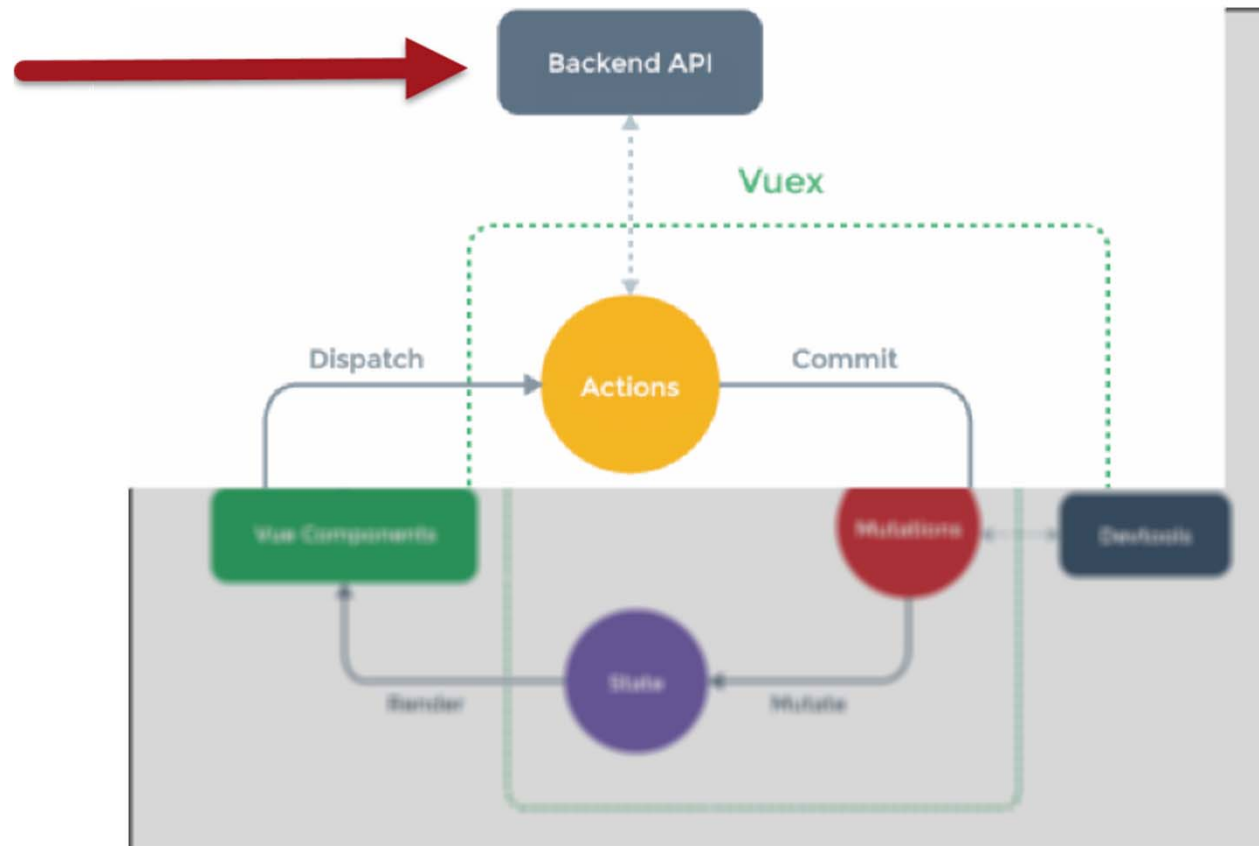


# More complex state operations

Talking to external API's, storing results in the Store



# More complex operations



# Using external API's

- Steps on creating and updating the store

1. Import `axios`

2. Create initial state

3. Create mutations to set the state

4. Create actions to call mutations and update the state.

- In the actions the `http-request` is made

Later, you update the component to alter the state

## Step 1 – import axios

- We're using `axios` here, so import that.
- Also set the url.
- File: `../store/index.js`

```
// store/index.js
...

import axios from 'axios';

const url = 'https://restcountries.eu/rest/v2/all';
```

## Step 2 – set initial state

```
export default new Vuex.Store({  
  state: {  
    loadingStatus: 'notloading',  
    countries: [],  
    errors: []  
  },  
  ...  
})
```

Of course you can add more properties to the state, if needed

## Step 3 – add mutations

```
export default new Vuex.Store({  
  ...  
  mutations: {  
    SET_LOADING_STATUS(state, payload) {  
      state.loadingStatus = payload;  
    },  
    SET_COUNTRIES(state, payload) {  
      state.countries = payload;  
    },  
    CLEAR_COUNTRIES(state) {  
      state.countries = []  
    },  
    ADD_ERROR(state, payload) {  
      state.errors = [...state.errors, payload]  
    }  
  }  
},  
  ...  
})
```

Mutations create a new `state` in the store via the second parameter, called `payload`

## Step 4 – add actions

```
actions: {
  fetchCountries(context) {
    // 1. Set loading status
    context.commit('SET_LOADING_STATUS', 'loading');
    // 2. Make http-request
    axios.get(url)
      .then(result => {
        context.commit('SET_LOADING_STATUS', 'notloading');
        context.commit('SET_COUNTRIES', result.data);
      })
      .catch(err => {
        context.commit('SET_LOADING_STATUS', 'notloading');
        context.commit('SET_COUNTRIES', []);
        context.commit('ADD_ERROR', err);
      })
  },
  clearCountries(context) {
    context.commit('CLEAR_COUNTRIES')
  }
},
```

Remember, `actions` don't update the store directly. They call `mutations` to commit data to the store. Note the use of the `context` parameter.

# Create component to view data from store

Template, ApiVuexComponent.vue

```
<template>
  <div>
    <h2>Countries via API - stored in Vuex store</h2>
    <button @click="fetchCountries()" class="btn btn-success">Fetch countries</button>
    <button @click="clearCountries()" class="btn btn-danger">Clear countries</button>
    <!--Loading indicator/spinner-->
    <div v-if="!loading">
      
    </div>
    <!--List with country data-->
    <ul class="list-group" v-if="countries && countries.length">
      <li class="list-group-item">
        <h4>{{ country.name }} </h4>
        ...
      </li>
    </ul>
  </div>
</template>
```

# Component logic

```
export default {
  name: "ApiVuexComponent",
  methods: {
    // 1. fetch all countries from the store
    fetchCountries() {
      this.$store.dispatch('fetchCountries')
    },
    // 2. clear countries from the store
    clearCountries() {
      this.$store.dispatch('clearCountries')
    },
  },
  computed: {
    countries() {
      return this.$store.state.countries;
    },
    loading(){
      return this.$store.state.loadingStatus === 'notloading'
    }
  }
}
```


Note – no data in this component anymore. Only store calls.

The computed properties are bound to the store state(s)







# Result

← → ↻ 🏠 ⓘ localhost:8080/vuex

 Home Country API - direct http Country API - vuex

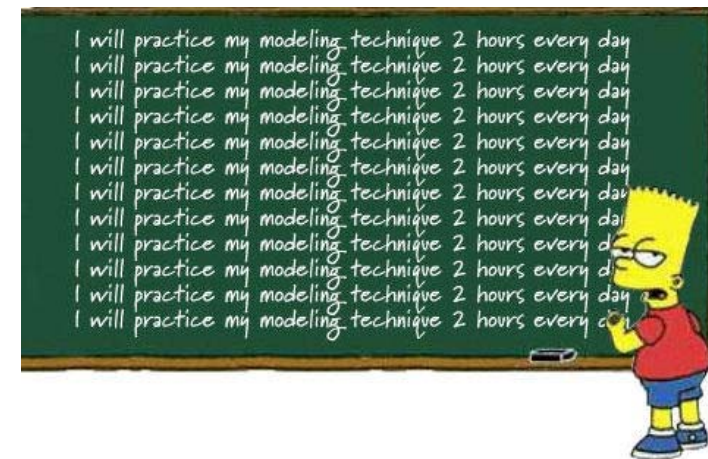
## Countries via API - stored in Vuex store

Fetch countries Clear countries

	Afghanistan Kabul, pop. 27657145
	Åland Islands Mariehamn, pop. 28875
	Albania Tirana, pop. 2886026
	Algeria Algiers, pop. 40400000

# Workshop

- Pick one of your own projects, or see example at:
  - `../410-vuex-countries`
- Create a small application using one of the API's in the file `JavaScript API's.txt`
- Store and fetch data in a store. Use for instance:
  - Pokemon API
  - Open Movie Database API
  - OpenWeatherMap API
  - ...





# Fetching details from the store

Working with `getters`

# Using Getters

- Sometimes you only need a *piece* of the state, for example a specific country
  - You can add a `find` function to every component that needs it:
  - ```
return this.$store.state.countries.find (c=> c.name ===  
  '<some-name>')
```
  - If multiple components need this logic, it's better to write a *getter*
- <https://vuex.vuejs.org/guide/getters.html>

# Writing a getter

- In our case: pass argument to the store by returning a function

```
// ../store/index.js
...
getters: {
  // only return the requested country from the store
  getCountry: (state) => (name) => {
    return state.countries.find(c => c.name === name)
  }
}
```

- Use the getter to retrieve value in a Detail Component
  - Example: `ApiVuexDetail.vue`

# Detail component – via Routeparameters

```
<!--ApiVuexDetail.vue-->
<template>
  <div v-if="country">
    <h2>Details for {{ country.name }}</h2>
    ...
  </table>
</div>
</template>

<script>
  export default {
    name: "ApiVuexDetail",
    created() {
      // get name from the route parameter.
      this.name = this.$route.params.name;
    },
    computed: {
      country() {
        return this.$store.getters.getCountry(this.name);
      }
    }
  }
</script>
```

# Benefits b/c now a store is used

- Data is already in the store – no additional http-call needed
- Use routing and parameters as you already know
- Study for yourself:
  - Property-Style Access
  - Method-Style Access (like we're using here)
  - The mapGetters helper

## The `mapGetters` Helper

The `mapGetters` helper simply maps store getters to local computed properties:

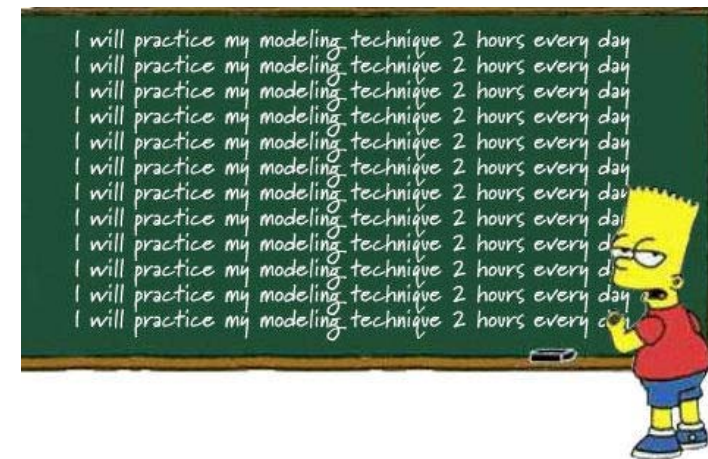
```
import { mapGetters } from 'vuex'

export default {
  // ...
  computed: {
    // mix the getters into computed with object spread operator
    ...mapGetters([
      'doneTodosCount',
      'anotherGetter',
      // ...
    ])
  }
}
```

<https://vuex.vuejs.org/guide/getters.html>

# Workshop

- Continue with your own project, or see example at:
  - `../410-vuex-countries`
- Create a detail option for your own project, using a getter
- Read the documentation on getters at <https://vuex.vuejs.org/guide/getters.html>





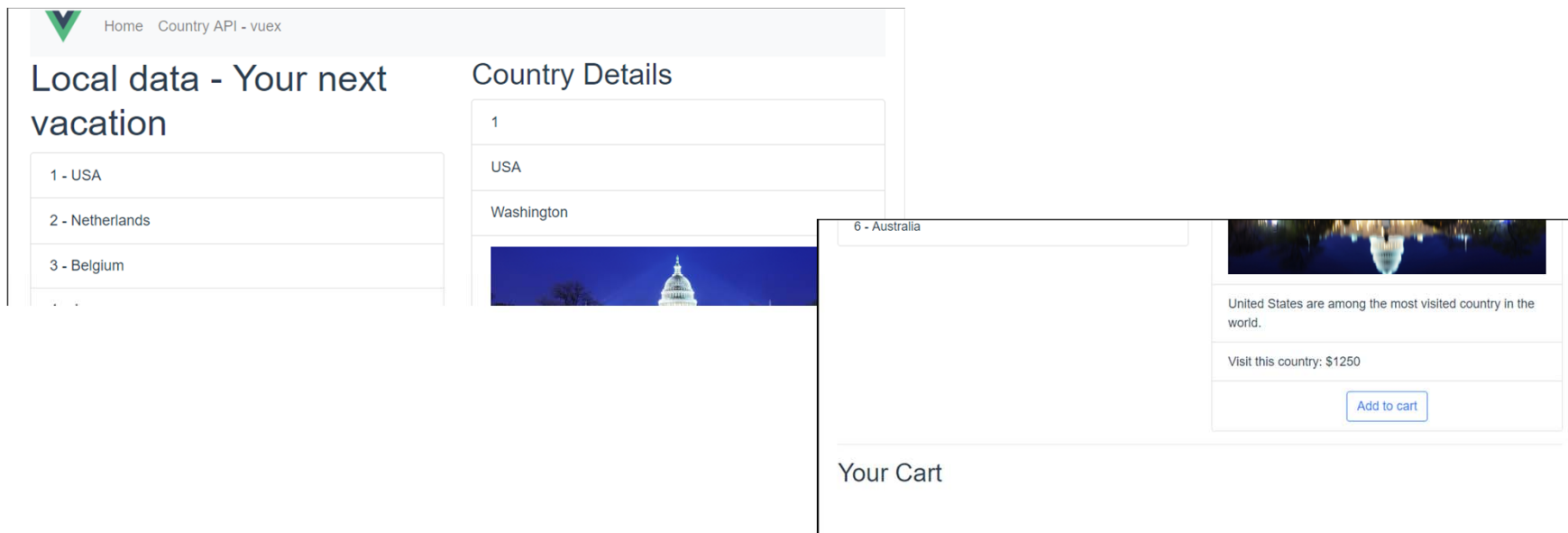


# Case – a country shopping cart

Booking trips to a country and saving them in a shopping cart in the store

# Going back to our static country data (for now)

- Use case
  - In our list of countries, every country has a `cost` property
  - When clicked, we see details and price of a trip to that country
  - We add an `Add to cart` button to every country
  - When clicked, a `ShoppingCart.vue` component is populated
  - The cart shows every item and offers a (fake) checkout option



# 1. Update the store

- Update the state with array of countries (here, called `items: []`)
- Add mutations to Add and Remove items and to Clear the cart
  - `ADD_TO_CART`
  - `REMOVE_FROM_CART`
  - `CLEAR_CART`
- Add Actions that call the various mutations
  - `addCountryToCart`
  - `removeCountryFromCart`
  - `checkout`
  - `clearCart`
- Add getters that return cart information
  - `cartProducts`
  - `cartTotalPrice`

`../store/index.js`

## 2. Create the ShoppingCart component

- Show a div if there are no items in the shopping cart
- Show a list or table if there are items in the shopping cart
- `../components/ShoppingCart.vue`

### Your Cart

|                |        |
|----------------|--------|
| Japan - 2895   | Delete |
| Brazil - 4500  | Delete |
| Total: \$ 7395 |        |

Checkout

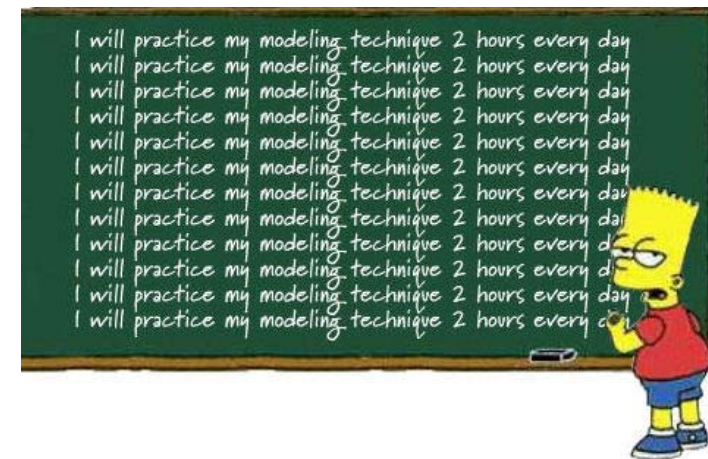
### 3. Update the CountryDetail component

- Show a div if there are no items in the shopping cart
- Show a list or table if there are items in the shopping cart
- `../components/ShoppingCart.vue`



# Workshop

- Create a simple e-commerce application using the store and adding a shopping cart.
  - Example: `.../420-vuex-shopping-cart`
- Study the example
- Update the example so you can get multiple trips to the same country (i.e. the same product) in the store
  - Create a counter or quantity field to record how many items of the product are in the cart



# Checkpoint

- You know what a Vuex State Management Store is
- You are familiar with store concepts like Actions, Mutations, Payload and Dispatching
- You know how to work with store Getters
- You can call `http` from your store and put retrieved data in a store