Vue.js - The Progressive JavaScript Framework

—

AxonActive Workshop team

Agenda

- Vue introduction
- Core concept
  (Event handling, Component, Form input)
- Vue Router
- VueX

Vue introduction

- Vue (/vju:/, like view) is a progressive framework

adoptable

focused

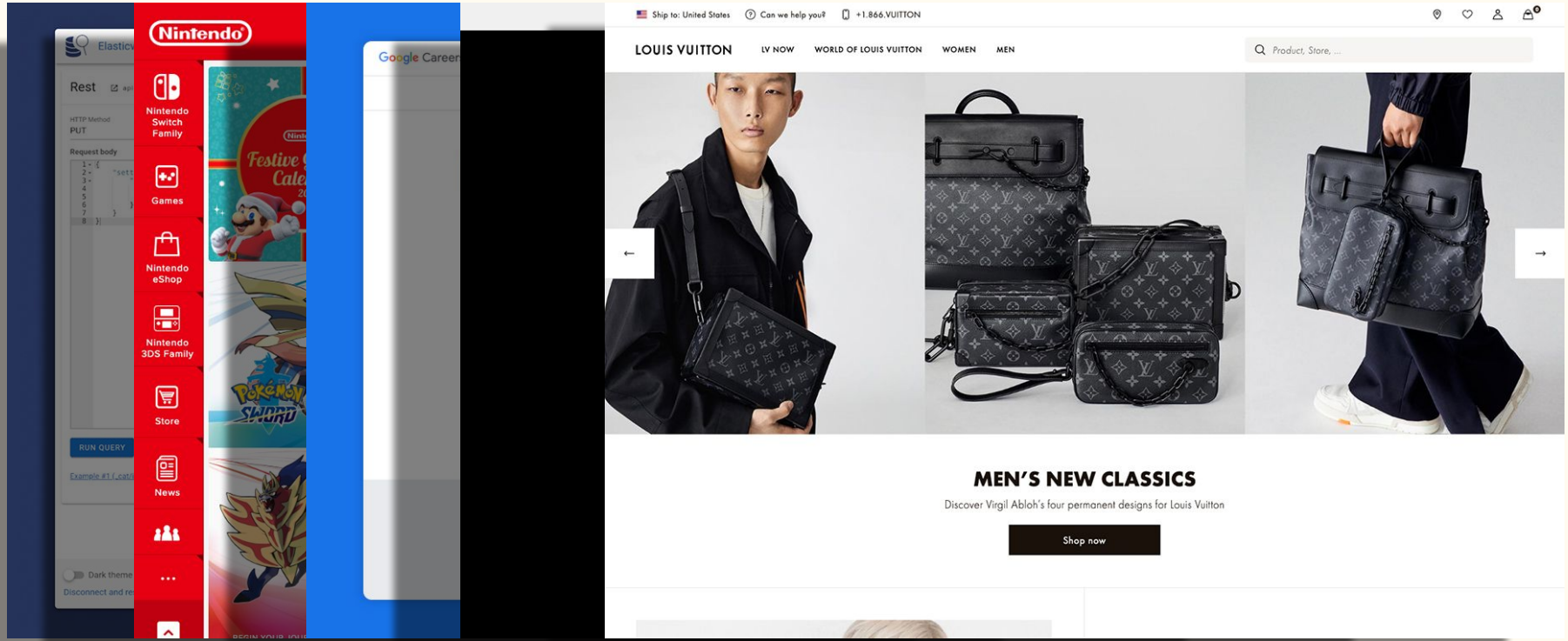
integrable

Single-Page Applications

When to use Vue?

- When Your App Is Full of Animations and Interactive Elements
- When You Need Seamless Integration with Multiple Apps
- When You Want To Prototype Without Advanced Skills

Who is using Vue



Basic concepts

- Conditional Rendering
- List Rendering

Conditional Rendering

- The directive **v-if** is used to conditionally render a block. The block will only be rendered if the directive's expression returns a truthy value.
- **v-else**
- **v-if-else**

```
<div v-if="type === 'A'">
  A
</div>
<div v-else-if="type === 'B'">
  B
</div>
<div v-else-if="type === 'C'">
  C
</div>
<div v-else>
  Not A/B/C
</div>
```

List Rendering

- We can use the **v-for** directive to render a list of items based on an array

```
<ul id="array-rendering">
    <li v-for="item in items">
        {{ item.message }}
    </li>
</ul>
```

```
Vue.createApp({
    data() {
        return {
            items: [{ message: 'Foo' }, { message: 'Bar' }]
        }
    }
}).mount('#array-rendering')
```

Event handling

- Listen to Events
- Event Modifiers

Listen to Events

- Use **v:on** or shorthand **@** to listen events
- Example with single event:

```
<div id="basic-event">
   <button @click="counter += 1">Add 1</button>
   <p>The button above has been clicked {{ counter }}
times.</p>
</div>
```

Listen to Events

- Example with complex single event:

```
<div id="event-with-method">
    <!-- `greet` is the name of a method defined below -->
    <button @click="greet">Greet</button>
</div>
```

```
<div id="inline-handler">
    <button @click="say('hi')">Say hi</button>
    <button @click="say('what')">Say what</button>
</div>
```

Listen to Events

- Example with multiple events:

```
<!-- both one() and two() will execute on button click -->
<button @click="one($event), two($event)">
   Submit
</button>
```

Event Modifiers

- It is a very common need to call **event.preventDefault()** or **event.stopPropagation()** inside event handlers
- It would be better if the methods can be purely about data logic
  - `.stop`
  - `.prevent`
  - `.capture`
  - `.self`
  - `.once`
  - `.passive`

Event Modifiers

```
<!-- the click event's propagation will be stopped -->
<a @click.stop="doThis"></a>

<!-- the submit event will no longer reload the page -->
<form @submit.prevent="onSubmit"></form>

<!-- modifiers can be chained -->
<a @click.stop.prevent="doThat"></a>

<!-- just the modifier -->
<form @submit.prevent></form>
```

Event Modifiers

```
<!-- use capture mode when adding the event listener -->
<!-- i.e. an event targeting an inner element is handled here before being handled
<div @click.capture="doThis">...</div>

<!-- only trigger handler if event.target is the element itself -->
<!-- i.e. not from a child element -->
<div @click.self="doThat">...</div>
```

Key Modifiers

- When listening for keyboard events, we often need to check for specific keys. Vue allows adding key modifiers for **v-on** or **@** when listening for key events

```html
<!-- only call `vm.submit()` when the `key` is `Enter` -->
<input @keyup.enter="submit" />
```

```html
<input @keyup.page-down="onPageDown" />
```

Key Modifiers

- Key Aliases:

- `.enter`
- `.tab`
- `.delete` (captures both "Delete" and "Backspace" keys)
- `.esc`
- `.space`
- `.up`
- `.down`
- `.left`
- `.right`

- `.ctrl`
- `.alt`
- `.shift`
- `.meta`

Key Modifiers

.exact modifier: allows control of the exact combination of system modifiers needed to trigger an event.

```html
<!-- this will fire even if Alt or Shift is also pressed -->
<button @click.ctrl="onClick">A</button>

<!-- this will only fire when Ctrl and no other keys are pressed -->
<button @click.ctrl.exact="onCtrlClick">A</button>

<!-- this will only fire when no system modifiers are pressed -->
<button @click.exact="onClick">A</button>
```

Why Listeners in HTML?

- It's easier to locate the handler function implementations
- ViewModel code can be pure logic and DOM-free
- When a ViewModel is destroyed, all event listeners are automatically removed

Form Input Bindings

- Basic usage
- Value bindings
- Modifiers

Form Input Bindings

- Use **v-model** to create two-way data bindings
- **v-model** will ignore the initial data

Form Input Bindings

- Example for Text:

```
<input v-model="message" placeholder="edit me" />
<p>Message is: {{ message }}</p>
```

- Example for Checkbox, boolean value:

```
<input type="checkbox" id="checkbox" v-model="checked" />
<label for="checkbox">{{ checked }}</label>
```

Form Input Bindings

- Example for Multiple checkboxes, bound to the same array:

```
<div id="v-model-multiple-checkboxes">
    <input type="checkbox" id="jack" value="Jack"
v-model="checkedNames" />
    <label for="jack">Jack</label>
    <input type="checkbox" id="john" value="John"
v-model="checkedNames" />
    <label for="john">John</label>
    <input type="checkbox" id="mike" value="Mike"
v-model="checkedNames" />
    <label for="mike">Mike</label>
    <br />
    <span>Checked names: {{ checkedNames }}</span>
</div>
```

```
Vue.createApp({
    data() {
        return {
            checkedNames: []
        }
    }
}).mount('#v-model-multiple-checkboxes')
```

Value Bindings

- Use **v-bind** to bind the value to a dynamic property or bind the input value to non-string values
- Example for Checkbox:

```
<input type="checkbox" v-model="toggle" true-value="yes" false-value="no" />
```

```
// when checked:
vm.toggle === 'yes'
// when unchecked:
vm.toggle === 'no'
```

Value Bindings

- Example for Radio:

```
<input type="radio" v-model="pick" v-bind:value="a" />
```

```
// when checked:
vm.pick === vm.a
```

Value Bindings

- Example for Select Options:

```
<select v-model="selected">
  <!-- inline object literal -->
  <option :value="{ number: 123 }">123</option>
</select>
```

```
// when selected:
typeof vm.selected // => 'object'
vm.selected.number // => 123
```

Modifiers

- `.lazy` synced data after "change" instead of "input"

```
<input v - model.lazy="msg" />
```

- `.number` input will be automatically typecast as a number

```
<input v-model.number="age" type="number" />
```

- `.trim` whitespace from input will be trimmed automatically

```
<input v-model.trim="msg" />
```

Component Basics

- What are Components ?
- Passing data to Child Components with Props
- Listening to Child Components events

What are Components ?

- The most powerful features of VueJS
- Reusable code

Base example

```
<template>
  <h4>{{ title }}</h4>
</template>

<script>
  export default {
        name : "BlogPost",
        props : ['title']
    }
</script>

<style lang="scss" scoped>
</style>
```

How to use ?

- Component can be reused as many times as you want :

```
<template>
  <BlogPost></BlogPost>
  <BlogPost></BlogPost>
  <BlogPost></BlogPost>
</template>

<script>
  import BlogPost from 'BlogPostComponent.vue'
  export default {
    components : [BlogPost]
  }
</script>
```

- Each time you use a component, a new instance of it is created.

Props

- Props are custom attribute you can register on a component
- How to passing data to component :

```
<template>
  <BlogPost title="My journey with Vue"></BlogPost>
  <BlogPost title="Blogging with Vue"></BlogPost>
  <BlogPost title="Why Vue is so fun"></BlogPost>
</template>
```

Result 

My journey with Vue

Blogging with Vue

Why Vue is so fun

Listening to Child Component Events

- Communication with the parent component
- Usage :
  - The parent component listen to any event on the child component :

```
<BlogPost ... @enlarge-text="postFontSize += 0.1"></BlogPost>
```

  - The child component can emit an event on itself :

```
<button @click="$emit('enlargeText')">
    Enlarge Text
</button>
```

Listening to Child Component Events

- Emit a value with an Event:
  - The parent component:

```
<BlogPost ... @enlarge-text="postFontSize += 0.1"></BlogPost>
```

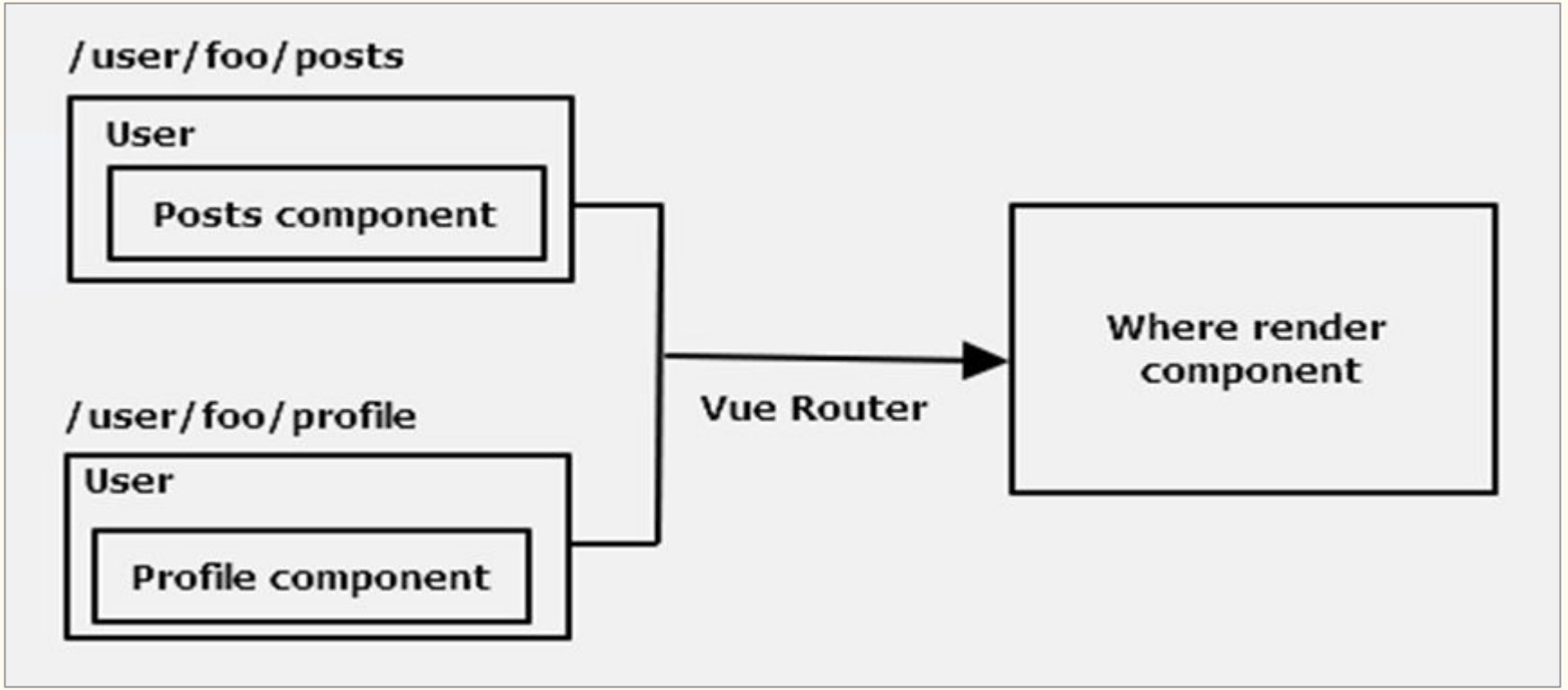
  - The child component:

```
<button @click="$emit('enlargeText', 0.1)">
   Enlarge Text
</button>
```

What is Vue Router?

- It is a official router for Vue.js.
- It integrates with Vue.js core to make building SPA with Vue.js a breeze.
- Features include:
  - Nested routes mapping
  - Dynamic Routing
  - Modular, component-based router configuration
  - Route params, query, wildcards
  - .....

What is Vue Router?



How to use?

```
// 1. Define route components.
const Foo = { template: '<div>foo</div>' }
const Bar = { template: '<div>bar</div>' }

// 2. Define some routes
const routes = [
  { path: '/foo', component: Foo },
  { path: '/bar', component: Bar }
]

// 3. Create the router instance and pass the `routes` option
const router = VueRouter.createRouter({,
 History: VueRouter.createWebHasHistory(),
  routes: routes
})

// 4. Create and mount the root instance.
const app = Vue.createApp({});
app.use(router);
app.$mount('#app');
```

How to use?

- **Router-link**
  - Render an <a> tag with the correct href attribute
  - It change the URL without reloading the page and handle URL generation as well as its encoding.

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="app">
        <h1>Hello App!</h1>
        <router-link to="/foo">Go to Foo</router-link>
        <router-link to="/bar">Go to Bar</router-link>
</div>
```

How to use?

- **Router-view**
  - Display the component that corresponds to the url.
  - You can put it anywhere to adapt it to your layout.

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="app">
        <h1>Hello App!</h1>
        <p>
        <router-link to="/foo">Go to Foo</router-link>
        <router-link to="/bar">Go to Bar</router-link>
        </p>
         <router-view></router-view>
</div>
```

Dynamic Route Matching with Params

- Define route:

```
const routes = [
        // dynamic segments start with a colon
        { path: '/user/:id', component: User }
]
```

- In component:

```
const User = {
        template: '<div>User {{ $route.params.id }}</div>'
}
```

Dynamic Route Matching with Params

- You can have multiple params in the same route, and they will map to corresponding fields on `$route.params`

pattern	matched path	$route.params
/users/:username	/users/eduardo	{ username: 'eduardo' }
/users/:username/posts/:postId	/users/eduardo/posts/123	{ username: 'eduardo', postId: '123' }

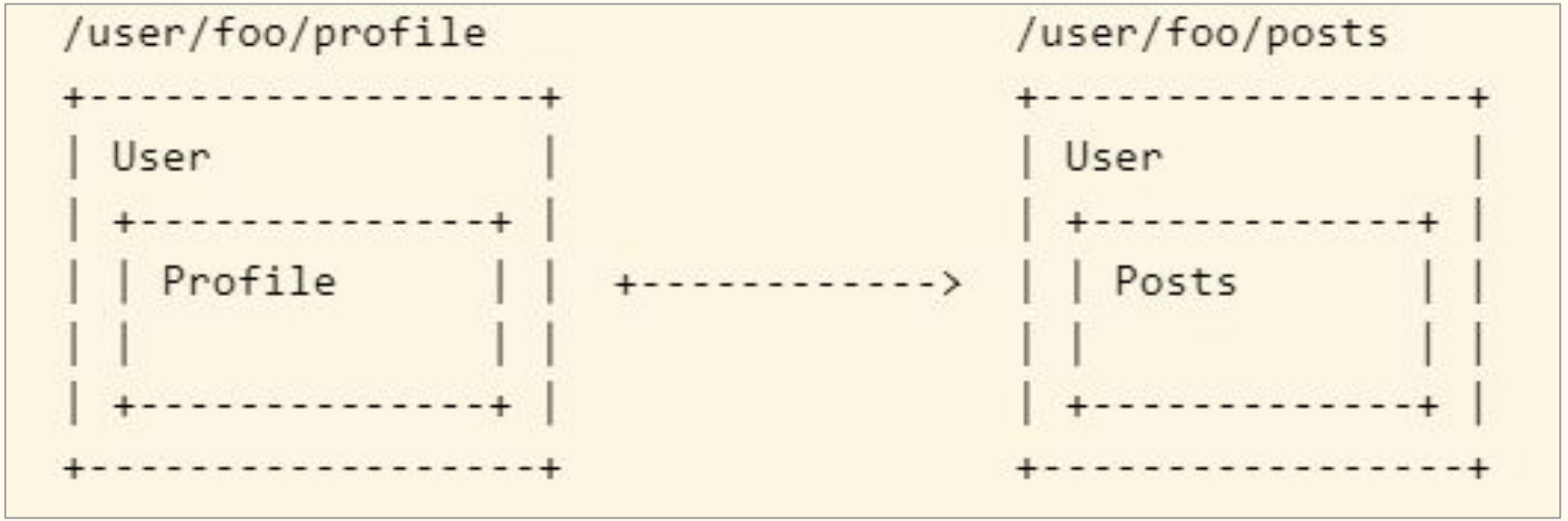
Reacting to Params Changes

- When the user navigates from `/users/johnny` to `/users/jolyne`, the same component instance will be reused => don't call the lifecycle hooks.
- Two simple ways to handle:

```
const User = {
  template: '...',
  created() {
    this.$watch(
        () => this.$route.params,
        (toParams, previousParams) => {
          // react to route changes
        })
  },
};
```

```
const User = {
  template: '...',
  props:{},
  async beforeRouteUpdate(to, from, next) {
    // react to route changes...
    // don't forget to call next()
    this.userData = await fetchUser(to.param.id)
    next();
  }
}
```

Nested Routes



Nested Routes

```
<div id="app">
        <router-view></router-view>
</div>
```

```
const routes = [
   {
     path: '/user/:id', component: User,
     children: [
       { path: 'profile', component: UserProfile },
       { path: 'posts', component: UserPosts }
     ]
   }
]
```

```
const User = {
  template: `
  <div class="user">
    <h2>User {{ $route.params.id }}</h2>
    <router-view></router-view>
  </div>`
}
```

Nested Routes

```
<script src="https://unpkg.com/vue/dist/vue.js"></script>
<script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

<div id="app">
  <p>
    <router-link to="/user/foo">/user/foo</router-link>
    <router-link to="/user/foo/profile">/user/foo/profile</router-link>
    <router-link to="/user/foo/posts">/user/foo/posts</router-link>
  </p>
  <router-view></router-view>
</div>
```

Programmatic Navigation

- The way to navigate programmatically:

Declarative	Programmatic
<router-link :to="...">	router.push(...)

- Some examples:

```
router.push('home')
router.push({ path: 'home' })

// named route
router.push({ name: 'user', params: { userId: '123' } })

// with query, resulting in /register?plan=private
router.push({ path: 'register', query: { plan: 'private' } })
```

Named Routes

- Alongside the `path`, you can provide a `name` to any route.
- Define route:

```
const routes = [{
        path: '/user/:userId',
        name: 'user',
        component: User},
]
```

- Use in `router-link` or `router.push`

```
<router-link :to="{ name: 'user', params: { userId: 123 }}">User</router-link>
```

```
router.push({ name: 'user', params: { userId: 123 } })
```

Named Views

- A `router-view` without a name will be given `default` as its name.

```
<router-view class="view one"></router-view>
<router-view class="view two" name="LeftSideBar"></router-view>
<router-view class="view three" name="RightSideBar"></router-view>
```

```
const router = createRotuer({
  routes: [{
          path: '/',
          components: {
              default: Home,
              LeftSideBar: LeftSideBar,
              RightSideBar: RightSideBar
          }
    }]
});
```

Passing Props to Route Components

We can replace

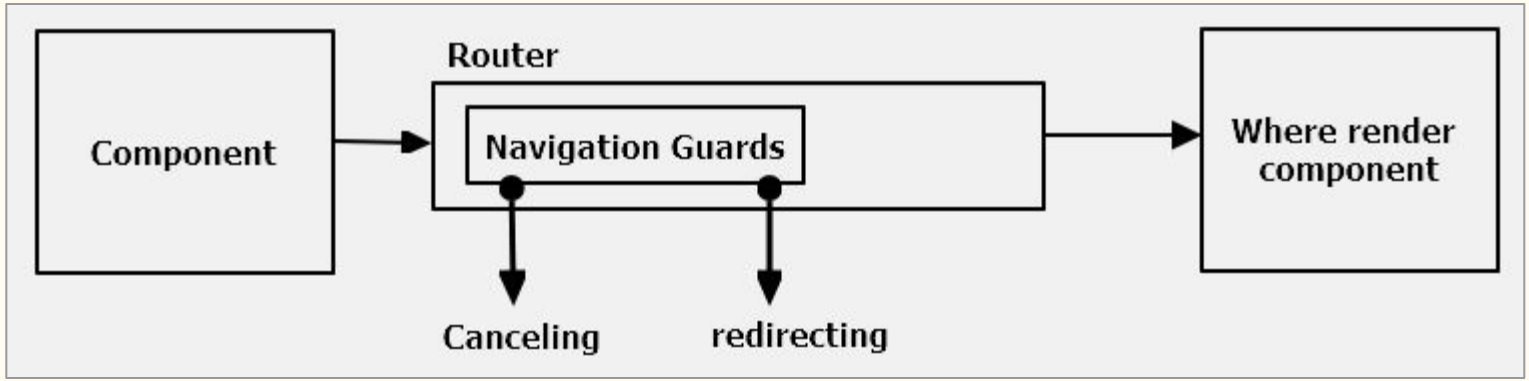
```
const User = {
        template: '<div>User {{ $route.params.id }}</div>'
}
const router = createRouter({
        routes: [{ path: '/user/:id', component: User }]
})
```

by

```
const User = {
  props: ['id'],
  template: '<div>User {{ id }}</div>'
}
const router = createRouter({
  routes: [
    { path: '/user/:id', component: User, props: true },
  ]
})
```

Navigation Guards

As the name suggests, the navigation guards provided by vue-router are primarily used to guard navigations either by redirecting it or canceling it



There are 3 ways to hook into the route:

- Global Before Guards.
- Per-Route Guard.
- In-Component Guards.

Navigation Guards - Global Before Guards

- **beforeEach:** whenever a navigation triggered

```
const router = createRouter({ ... })

router.beforeEach((to, from, next) => {
  // ...
})
```

to: Route: the target Route Object being navigated to.
from: Route: the current route being navigated away from.
next: Function: this function must be called to resolve the hook.

- **next()**: move on to the next hook in the pipeline.
- **next(false)**: abort the current navigation.
- **next('/')**: redirect to a different location.
- **next(error):** aborted navigation and passed to callbacks registered via router.onError()

Navigation Guards - Global Before Guards

- **beforeResolve:** Called right before the navigation is confirmed, after all in-component guards and async route components are resolved

- **afterEach:** Not affect the navigation

```
router.afterEach((to, from) => {
  // ...
})
```

Navigation Guards - Per-Route Guard

- **beforeEnter:**

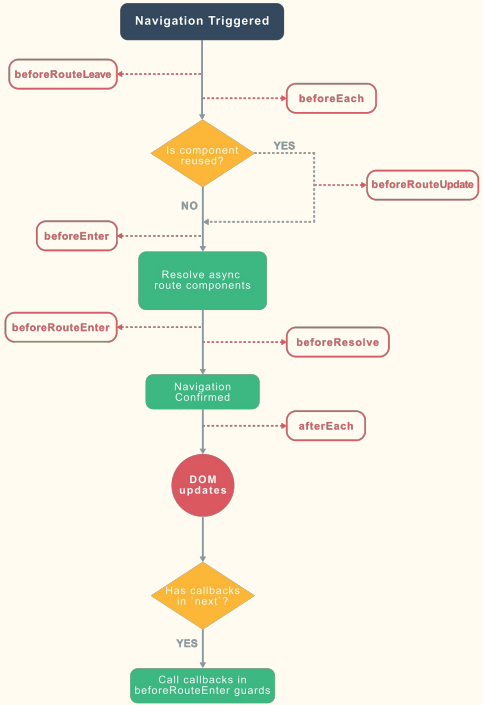
```
const router = createRouter({
  routes: [{
          path: '/foo',
          component: Foo,
          beforeEnter: (to, from, next) => {
          // ...
     }
   }]
});
```

Navigation Guards - In-Component Guards

- **beforeRouteEnter:** called before the route that renders this component is confirmed (component did not render yet).
- **beforeRouteUpdate:** called when the route that renders this component has changed (dynamic route matching params).
- **beforeRouteLeave:** called when the route that renders this component is about to be navigated away from (e.g ask to save data).

```
const Foo = {
  template: `...`,
  beforeRouteEnter(to, from, next) { // called before the route that renders this component is confirmed.
  },
  beforeRouteUpdate(to, from, next) { // called when the route that renders this component has changed.
  },
  beforeRouteLeave(to, from, next) { // called when the route that renders this component is about to be navigated away from.
  }
}
```

Navigation Guards - Overview



Route Meta Fields

```
const router = new VueRouter({
  routes: [{
      path: '/foo',
      component: Foo,
      children: [{
            path: 'bar',
            component: Bar,
            // a meta field
            meta: { requiresAuth: true }
        }]
   }]
});
```

```
router.beforeEach((to, from, next) => {
  if (to.matched.some(record => record.meta.requiresAuth)) {
    // this route requires auth, check if logged in
    // if not, redirect to login page.
  }
  next() // make sure to always call next()!
})
```

Transitions

```
<!-- use a dynamic transition name -->
<transition :name="transitionName">
  <router-view></router-view>
</transition>
```

```
// then, in the parent component,
// watch the `$route` to determine the transition to use
watch: {
  '$route'(to, from) {
    const toDepth = to.path.split('/').length
    const fromDepth = from.path.split('/').length
    this.transitionName = toDepth < fromDepth ? 'slide-right' : 'slide-left'
  }
}
```

Data Fetching

- Fetching After Navigation

```
<template>
  <div class="post">
    <div v-if="loading" class="loading">
      Loading...
    </div>

    <div v-if="error" class="error">
      {{ error }}
    </div>

    <div v-if="post" class="content">
      <h2>{{ post.title }}</h2>
      <p>{{ post.body }}</p>
    </div>
  </div>
</template>
```

```
export default {
  data() {
    return {
      loading: false,
      post: null,
      error: null
    }
  },
  created() { this.fetchData() },
  watch: { '$route': 'fetchData' },
  methods: {
    fetchData() {
      this.error = this.post = null
      this.loading = true
      const fetchedId = this.$route.params.id
      getPost(fetchedId, (err, post) => {
        // where call api
      })
    }
  }
}
```

Data Fetching

- Fetching Before Navigation

```
export default {
  data() {
    return {post: null,  error: null}
  },
  beforeRouteEnter(to, from, next) {
    getPost(to.params.id, (err, post) => {
      next(vm => vm.setData(err, post))
    })
  },
  beforeRouteUpdate(to, from, next) {
    this.post = null
    getPost(to.params.id, (err, post) => {
      this.setData(err, post)
      next()
    })
  },
  methods: {
    setData(err, post) {
      if (err) {
        this.error = err.toString()
      } else {
        this.post = post
      }
    }
  }
}
```

- Fetching After Navigation

```
export default {
  data() {
    return {loading: false, post: null, error: null}
  },
  created() {
    this.fetchData()
  },
  watch: {
    '$route': 'fetchData'
  },
  methods: {
    fetchData() {
      this.error = this.post = null
      this.loading = true
      const fetchedId = this.$route.params.id
      // replace `getPost` with your data fetching util / API wrapper
      getPost(fetchedId, (err, post) => {
        if (this.$route.params.id !== fetchedId) return
        this.loading = false
        if (err) {
          this.error = err.toString()
        } else {
          this.post = post
        }
      })
    }
  }
}
```

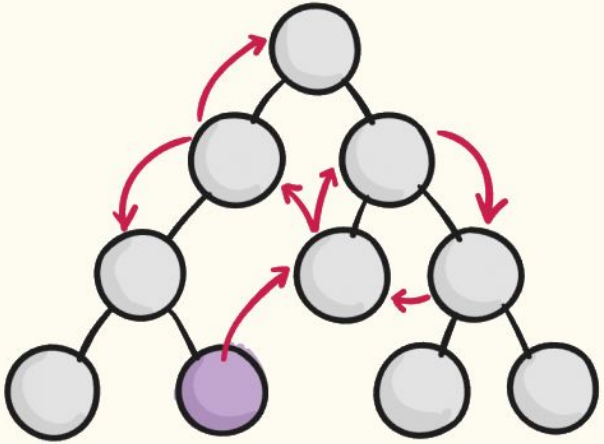
What is state?

- **State** is simply an object that contains the properties that need to be shared within the application. E.g:
  - ***List of customers***, ***products*** from database is a type of state.
  - ***Events*** in the browser or the ***color*** or a ***div element*** also a state.

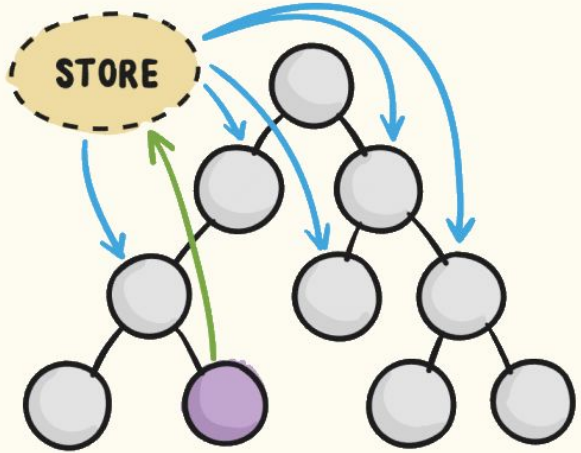
What is state management ?

- **State management** is the implementation of a Design Pattern, which help us can **synchronize the state** of the application **throughout all components** of the application.

What is state management ?



WITHOUT
state management



WITH
state management

Benefit of state management ?

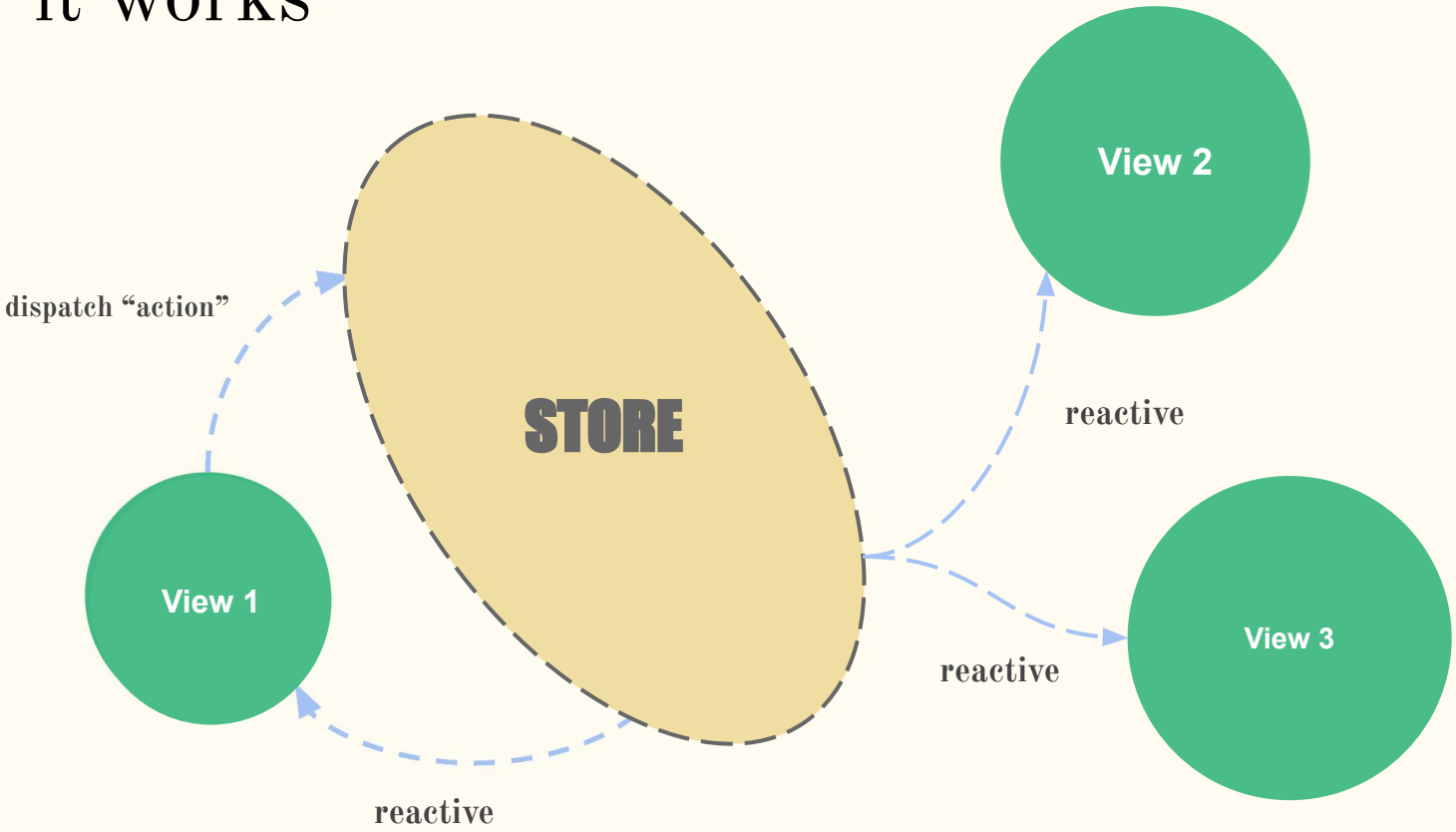
WITHOUT state management

- No central place to hold the state. Need to access the single state or data from different places.
- **More** HTTP **requests** need to sent to the back-end for fetching and retrieval of the data.
- **More logic** stay together with the view, difficult to read, maintain.

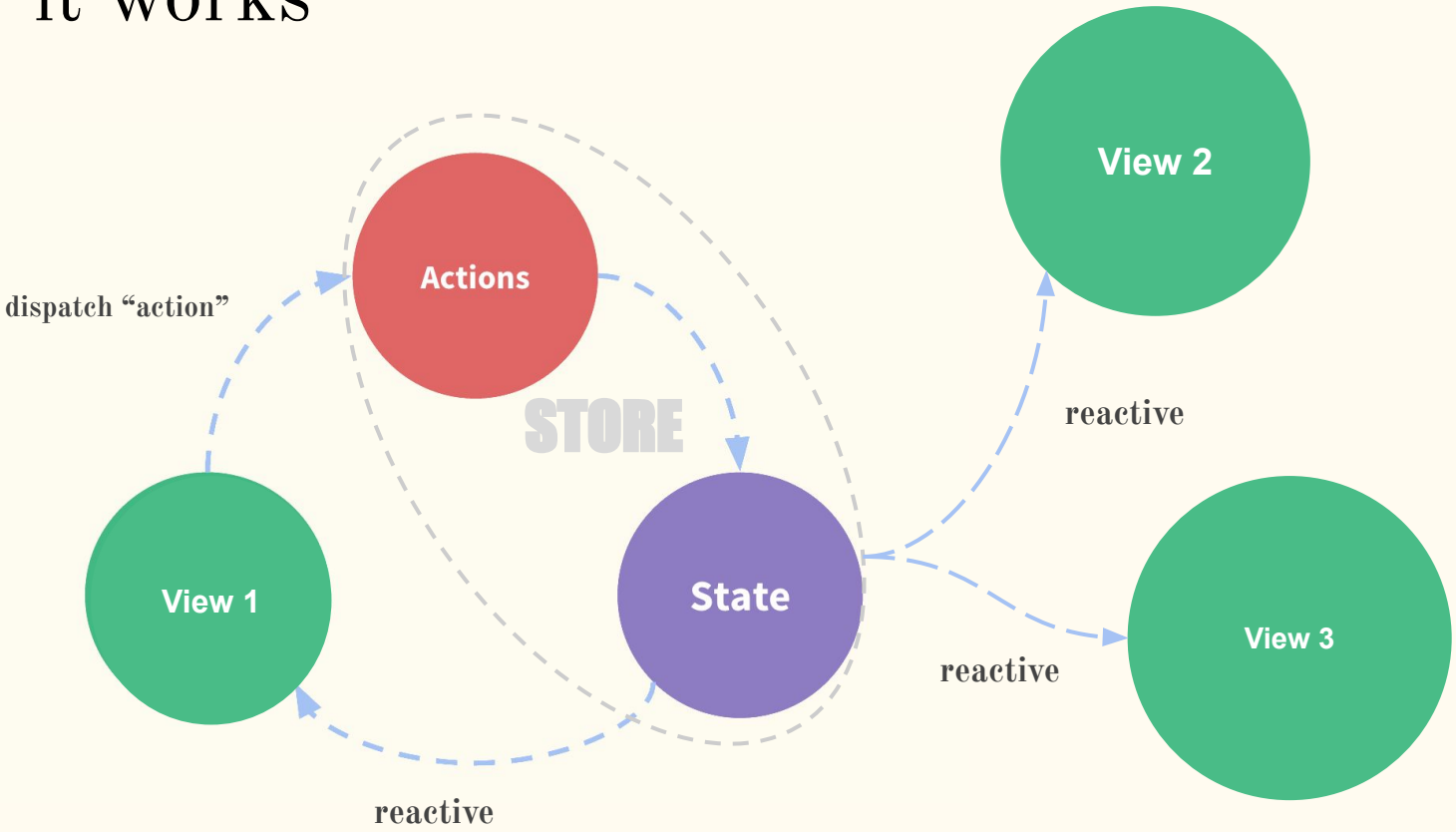
WITH state management

- State of the whole application is present at a **single place** which named **Store.**
- **Reduces** the HTTP **requests** sent to the back-end for fetching and retrieval of the data.
- **Centralize the code,** easy to maintain.

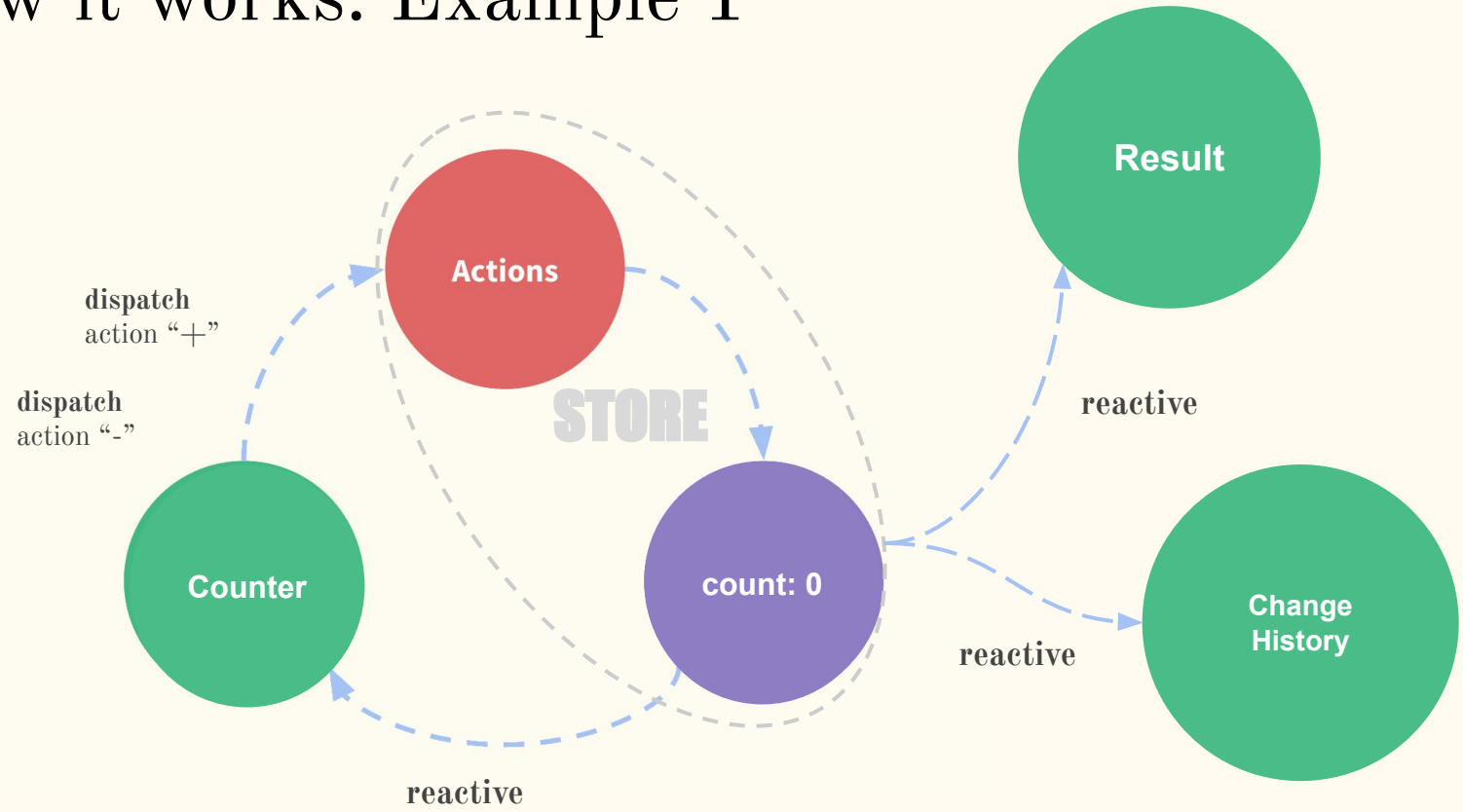
How it works


View 2
dispatch “action”
STORE
reactive
View 1
View 3
reactive
reactive

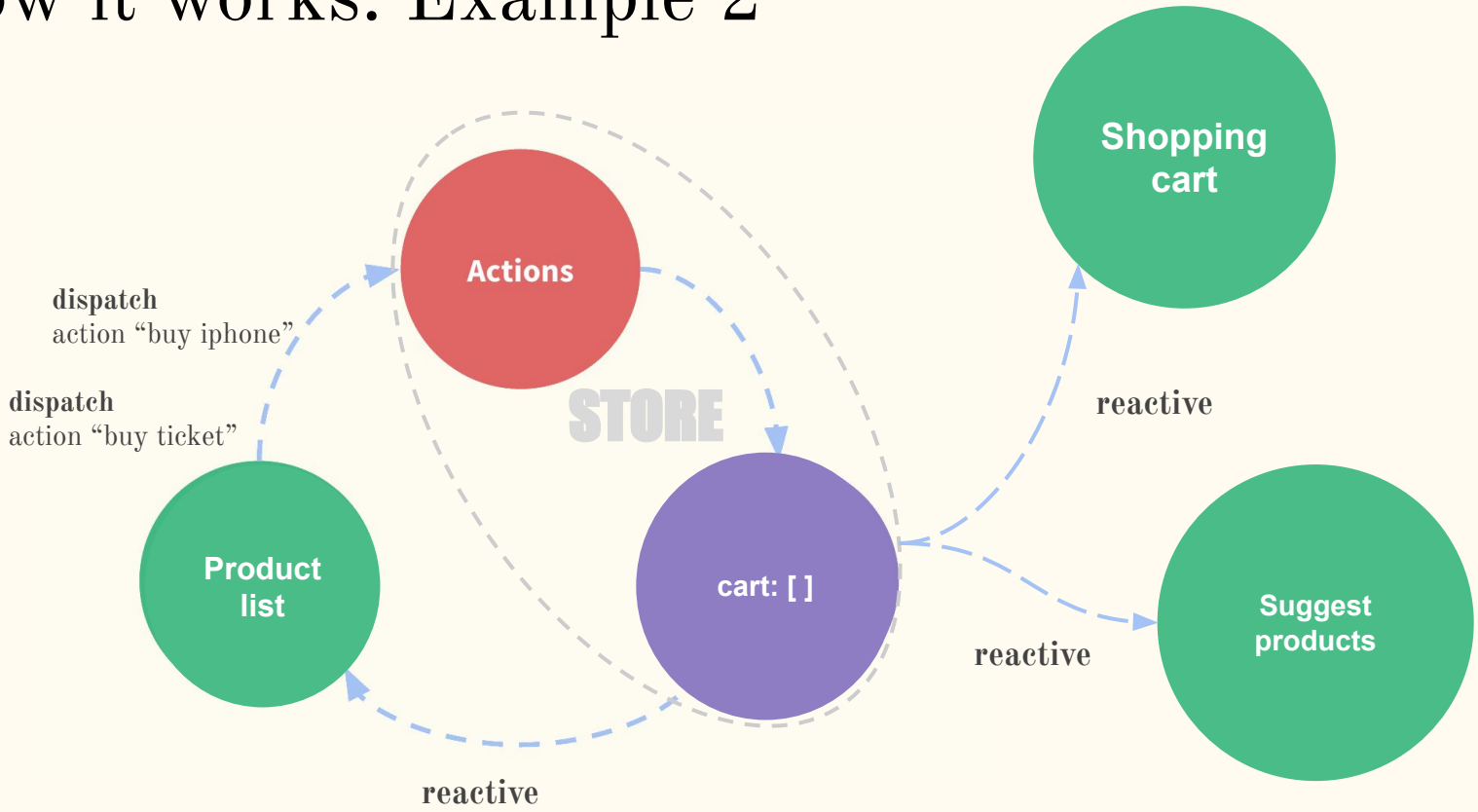
How it works


View 2
Actions
dispatch “action”
STORE
reactive
View 1
State
View 3
reactive
reactive

How it works: Example 1



How it works: Example 2



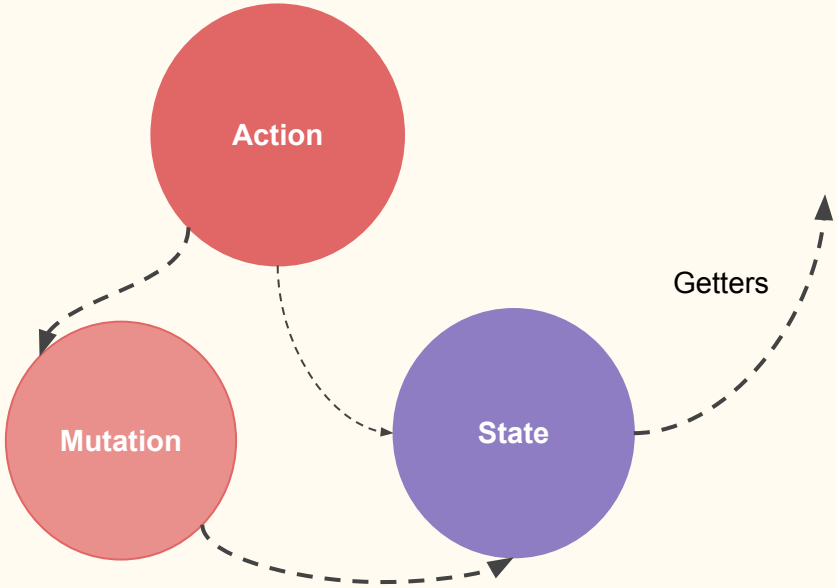
State management tools

Redux

Core concept

There are 5 core concepts in VueX:

- State
- Getters
- Mutations
- Actions
- Modules



VueX - State

```
export default new Vuex.Store({
  state: {
    toDos: JSON.parse(localStorage.getItem("key")),
    count: 0
  }
})
```

```
$this.store.state.doDo
```

VueX - Getters

```
export default new Vuex.Store({
  state: {
    toDos: JSON.parse(localStorage.getItem("key")),
    count: 0
  },
  getter: {
    doneToDos: (state) => {
  }
      return state.toDos.filter(m => m.completed == true)
    }
})
```

VueX - Mutation

```
mutations: {
  increment(state, param) {
    state.count += param.number
  }
}
```

```
this.$store.commit('increment', 9)
```

VueX - Actions

```
action: {
  increment({commit}) {
    commit('increment')
  }
}
```

```
this.$store.dispatch('increment')
```

VueX - Modules

```
const moduleA = {
  state: {...},
  mutations: {...},
  actions: {...},
  getters: {...}
}
const moduleB = {
  state: {...},
  mutations: {...},
  actions: {...}
}
```

```
const store = new Vuex.Store({
  modules: {
    a: moduleA,
    b: moduleB
  }
})

store.state.a //-> `moduleA`'s state
store.state.b //-> `moduleB`'s state
```

Thanks for listening!