VueJS

Part 2

MisterBit

Props validation

It is possible for a component to specify requirements for the props it is receiving. If a requirement is not met, Vue will emit warnings.

Instead of defining the props as an array of strings, you can use an object with validation requirements:

```
props: {
  // basic type check (`null` means accept any type)
  propA: Number,
  // multiple possible types
  propB: [String, Number],
  // a required string
  propC: {
     type: String,
     required: true
  },
  // a number with default value
  propD: {
    type: Number,
    default: 100
  },
  // object/array defaults should be returned
  //  from a factory function
  propE: {
    type: Object,
    default: function () {
      return { message: 'hello' }
    }
  },
  // custom validator function
  propF: {
    validator: function (value) {
      return value > 10
    }
  }
}
```

MisterBit

Props validation

The type can be one of the following native constructors:

- String
- Number
- Boolean
- Function
- Object
- Array

- In addition, type can also be a custom constructor function and the assertion will be made with an instanceof check.
- When a prop validation fails, Vue will produce a console warning (if using the development build).

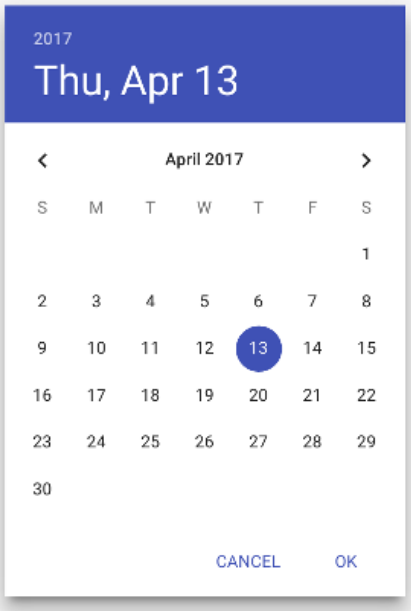
VueJS

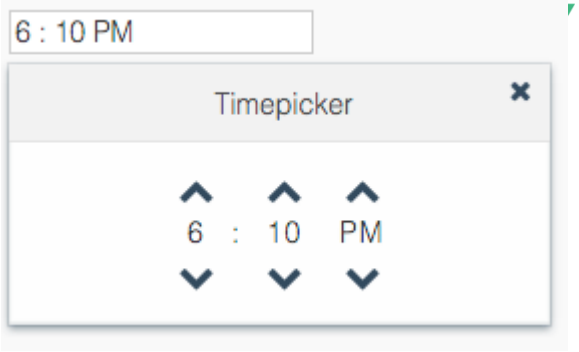
Custom controls

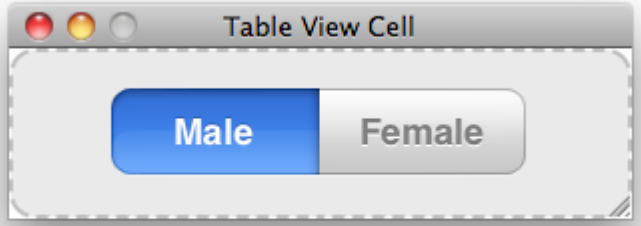
with v-model

Select a color:

MisterBit







HTML's built-in input types won't always meet your needs.

- Fortunately, Vue components allow you to build **reusable custom controls**
- Even better, they integrate nicely with v-model

Building a custom input

Its quite easy to create custom inputs that work with v-model:

```
// Syntatic Sugar:
<input v-model="something">

// Expanded to:
<input v-bind:value="something" v-on:input="something = $event.target.value">
```

So for a component to work with v-model, it must:

1. accept a *value* prop
2. emit an *input* event with the new value

See toggle-btn sample
Another Sample here

MisterBit

Filters

Apply common text formatting

- For more complex data transforms use Computed Properties instead.
- Filters can be chained, and can take arguments
- Here is a sample

```
new Vue({
  filters: {
    capitalize: function (value) {
      if (!value) return ''
      value = value.toString()
      return value.charAt(0).
             toUpperCase() +
             value.slice(1)
    }
  }
})
```

```
<tr v-bind:id="rawId | formatId"></tr>
      becomes:
<tr id="obj-3"></tr>
```

```
{{ message | capitalize }}
{{ message | capitalize(isMulti) }}
```

MisterBit

Methods, Filters & Computed

There is some similarity between them

- Use **methods** primarily for triggering state alterations. When you call a method, it generally implies some side effect.
- Use **filters** primarily for simple text formatting that needs to be reused all across your app. Filters should be pure - no side effects, just data in and data out.
- Use **computed** properties for local, component-specific data transforms. Similar to filters, computed getters should be pure.
- There is a special case where you want to compute something using a scope variable that is only available in the template e.g. a v-for alias), in such cases it's okay to use "helper methods" so that you can compute something by passing it arguments.

VueJS

Components

Content Distribution

Mister Bit

Content Distribution with Slots

When using components, it is often desired to compose them like this:

```
<great-box>
    Your {{thing}} is great!
</great-box>
```

- The <box> component has its own template and logic
- The <box> component accept content presented inside its mount target

- This is called **content distribution** (AKA “transclusion”).

Content Distribution with Slots

- Vue.js implements a content distribution API that is modeled after the current **Web Components spec draft**,
- using the special <slot> element to serve as distribution outlets for the original content.

```
<box>
  <box-header></box-header>
  <box-footer></box-footer>
</box>
```

See also this TABS sample

Compilation scope

Everything in the parent template is compiled in parent scope; everything in the child template is compiled in child scope.

```
<box>
  {{ parentData1 }}
  <box-header> {{ parentData2 }} </box-header>
  <box-footer></box-footer>
</box>
```

Single Slot

- Parent content will be **discarded** unless the child component template contains a <slot> outlet.
- The entire content fragment is inserted at its position in the DOM, replacing the slot itself.
- Anything originally inside the <slot> tags is (compiled in the child scope and) considered **fallback content**.
  - will only be displayed if the hosting element is empty and has no content to be inserted.

```
<div>
  <h1>I'm the parent title</h1>
  <child>
    <p>Some content</p>
    <p>Extra content</p>
   </child>
</div>
```

```
<div>
  <h2>I'm the child title</h2>
  <slot>
    Displayed if there is no content
    to be distributed.
  </slot>
</div>
```

Named Slots

- There is a name attribute, so we can have multiple slots with different names
- A named slot will match any element that has a corresponding slot attribute in the content fragment.
- There can still be one unnamed slot, which is the **default slot** that serves as a catch-all outlet for any unmatched content.
- If there is no default slot, unmatched content will be discarded.

```
<div>
  <h1>I'm the parent title</h1>
  <app-layout>
      <h1 slot="header">Page title</h1>
      <p>p1</p>
      <p>p2</p>
      <p slot="footer">Contact info</p>
   </app-layout>
</div>
```

```
<div class=“app-layout">
  <header>
      <slot name="header"></slot>
   </header>
   <main>
       <slot></slot>
   </main>
   <footer>
       <slot name="footer"></slot>
   </footer>
</div>
```

MisterBit

Authoring Reusable Components

- When authoring components, it's good to determine whether they are about to be reused
- It's OK for one-off components to be tightly coupled, but reusable components should define a clean public interface and make no assumptions about the context it's used in.
- The API for a Vue component comes in three parts:
  - **Props** allow the external environment to pass data into the component
  - **Events** allow the component to trigger side-effects in the external environment
  - **Slots** allow the external environment to compose the component with extra content

```
<my-component
  :foo="baz"
  :bar="qux"
  @event-a="doThis"
  @event-b="doThat">
        <img slot="icon" src="...">
        <p slot="main-text">Hello!</p>
</my-component>
```

MisterBit

Cheap Static Components with v-once

Rendering plain HTML elements is very fast in Vue, but sometimes you might have a component that contains **a lot** of static content.

In these cases, you can ensure that it's only evaluated once and then cached by adding the v-once directive to the root element, like this:

```
Vue.component('terms-of-service', {
template: `
<div v-once>
<h1>Terms of Service for date {{today}}</h1>
... a lot of static content ...
</div>`

})
```

components – DOM Template Parsing Caveats

There are restrictions that are inherent to how HTML works, because (at some cases) Vue retrieves the template content **after** the browser has parsed and normalized it.

```
<!– Problematic -->
<table>
    <my-row>...</my-row>
</table>

<!– Work around -->
<table>
    <tr is="my-row"></tr>
</table>
```

These limitations do not apply if you are using string templates from one of the following sources:

- <script type="text/x-template">
- JavaScript inline template strings
- .vue files

Therefore, **prefer using string templates whenever possible.**

Reactivity

In depth

Mister Bit

The Vue Instance

- When you instantiate a Vue instance, you need to pass in an **options object**
- It contain options for data, template, element to mount on, methods, lifecycle callbacks and more
- Vue instance **proxy** the properties found in its data object:

```
var data = { a: 1 }
var vm = new Vue({
data: data
})
vm.a === data.a // -> true
// setting the property also affects original data
vm.a = 2
data.a // -> 2
// ... and vice-versa
data.a = 3
vm.a // -> 3
```

The Vue Instance

```
var data = { a: 1 }
var vm = new Vue({
data: data
})
data.b = 8;  // -> b is not reactive!
```

- It should be noted that only these proxied properties are **reactive**.
- If you attach a new property to the instance after it has been created, it will not trigger any view updates (unless done in the *vue* way)
- We will discuss this further on.

The Vue Instance

Vue instances expose a number of instance properties and methods.

These properties and methods are prefixed with $

They are not used very often

```
var data = { a: 1 }

var vm = new Vue({
el: '#example',
data: data
})

vm.$data === data // -> true
vm.$el === document.getElementById('example') // -> true

// $watch is an instance method
vm.$watch('a', function (newVal, oldVal) {
    // this callback will be called when `vm.a` changes
})
```

Gotcha!

Don’t use *arrow functions* on an instance property or callback

- Wrong Code:
  vm.$watch('a', newVal => this.myMethod()).

- Arrow functions are bound to the parent context, and this.myMethod will be undefined.

Array change detection – Array Methods

Vue wraps an observed array's mutation methods so they will also trigger view updates. The wrapped methods are:

- push()
- pop()
- shift()
- unshift()
- splice()
- sort()
- reverse()

- Also, filter(), concat() and slice(), which do not mutate the original Array but return a new array.

Vue takes care of maximizing DOM element reuse, so replacing an array with another array containing overlapping objects is an efficient operation.

Array change detection – Array Caveats

- Due to limitations in JavaScript, Vue **cannot** detect the following changes to an array:
  1. When you directly set an item with the index, e.g. vm.**items**[idx] = newValue
  2. When you modify the length of the array, e.g. vm.**items**.length = newLength
- Remedies:
  1. Two options:
     - Vue.set(example1.**items**, indexOfItem, newValue)
     - app.**items**.splice(indexOfItem, 1, newValue)
  2. app.**items**.splice(newLength)

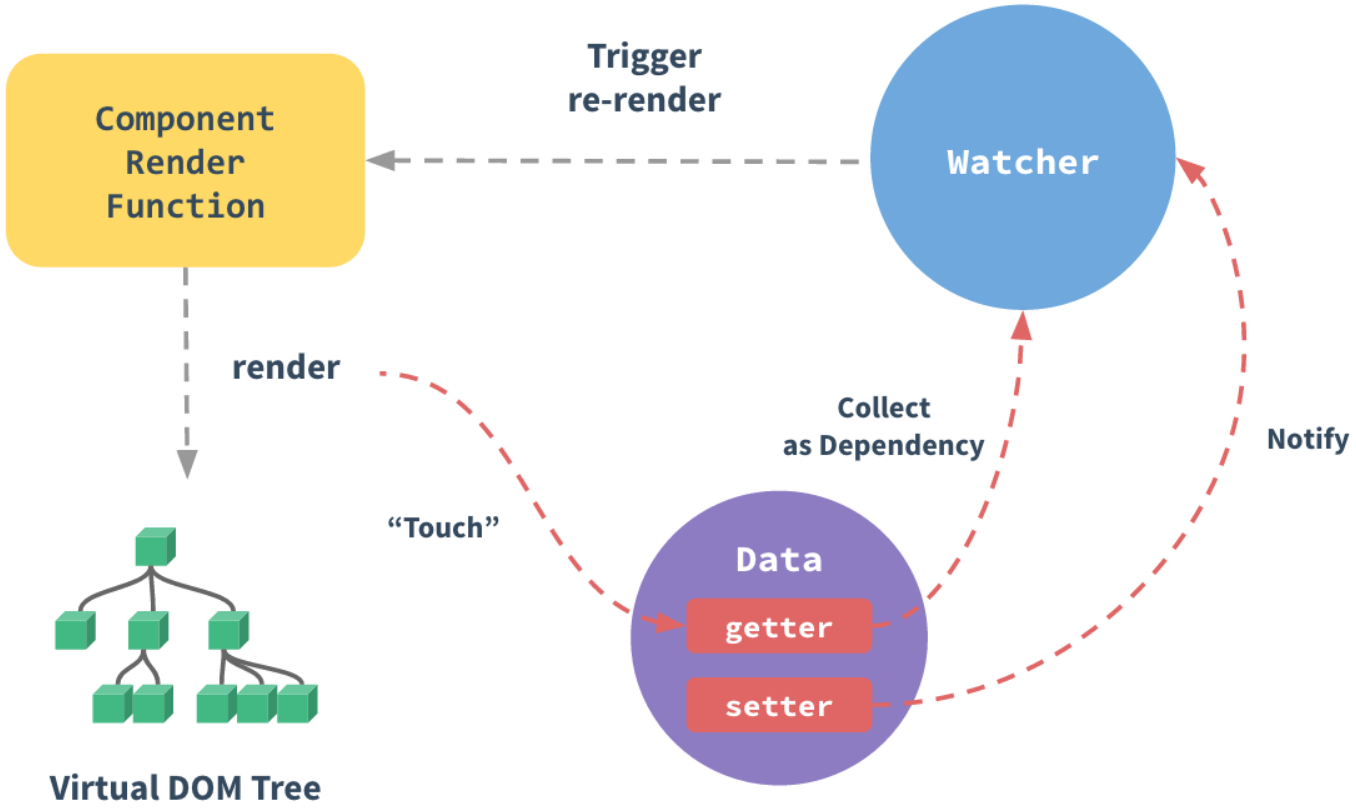
How Changes Are Tracked

- When you pass a plain JS object to a Vue instance as its data option, Vue will walk through all of its properties and convert them to getter/setters using Object.defineProperty.
  - This is an ES5-only and un-shimmable feature, which is why Vue doesn't support IE8 and below.

- The getter/setters are invisible to the user, but under the hood they enable Vue to perform:
- dependency-tracking and change-notification when properties are accessed or modified.

How Changes Are Tracked

- Every component instance has a corresponding (single) watcher instance, which records any properties “touched” during the component’s render as dependencies.
- Later on when a dependency’s setter is triggered, it notifies the watcher, which in turn causes the component to re-render.



Change Detection Caveats

- Vue **cannot detect property addition or deletion**.
- Since Vue performs the getter/setter conversion process during instance initialization, a property must be present in the data object in order for Vue to convert it and make it reactive.
- For example:

```
var vm = new Vue({
  data: {
    a: 1
  }
})
// `vm.a` is now reactive
vm.b = 2
// `vm.b` is NOT reactive
```

When needed, it's possible to add reactive properties like so:

```
Vue.set(vm.someObject, 'b', 2)
this.$set(this.someObject, 'b', 2)

// instead of `Object.assign(this.someObject, { a: 1, b: 2 })`
this.someObject = Object.assign({}, this.someObject, { a: 1, b: 2 })
```

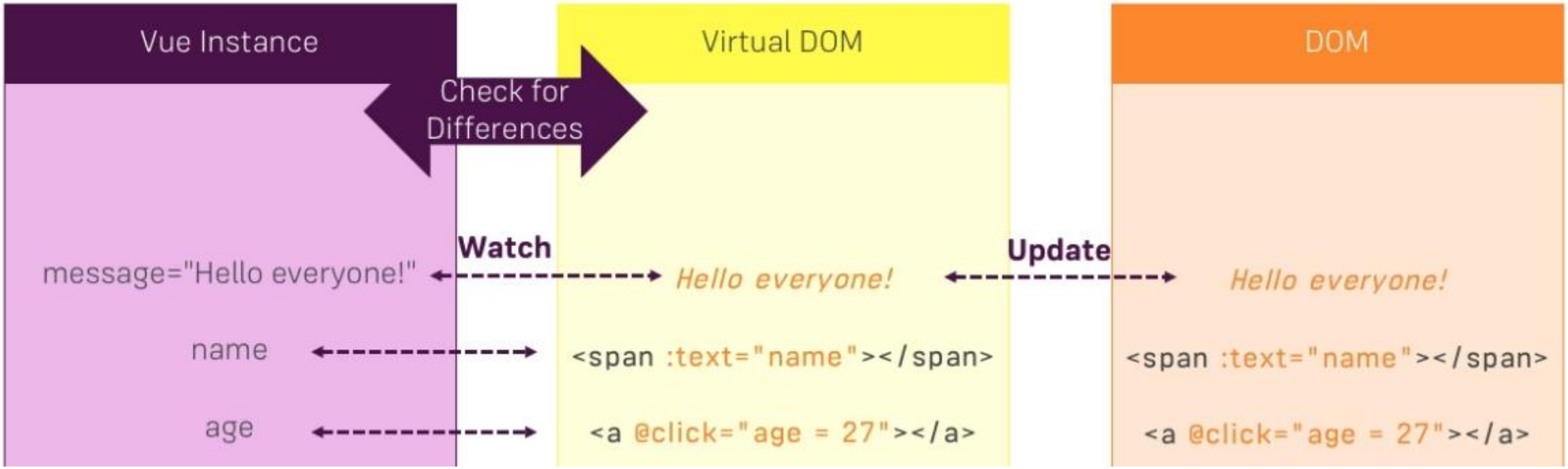
Async Update Queue

- Vue performs DOM updates **asynchronously**. Whenever a data change is observed, it will open a queue and buffer all the data changes that happen in the same event loop cycle.
- When its necessary to get your hands dirty and wait until Vue.js has finished updating the DOM after a data change, you can use Vue.nextTick

```
Vue.component('example', {
  template: '<span>{{ msg}}</span>',
  data() {
     return {
          msg: 'not updated'
     }
   },
   methods: {
      updateMsg () {
          this.msg = 'updated'
          console.log(this.$el.textContent) // => 'not updated'
          this.$nextTick(function () {
               console.log(this.$el.textContent) // => 'updated'
          })
       }
    }
})
```

MisterBit

VueJS DOM Updating



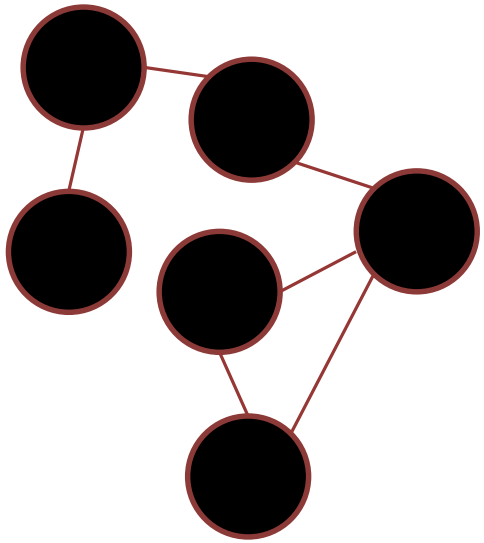
State Management

With Vuex

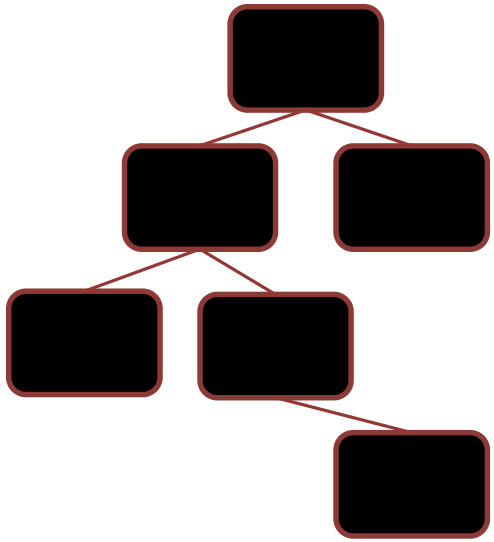
Overview

We make the model (state) of the application visible to user by the process we call rendering.

Model



DOM



Overview

In the past, it was not uncommon to have bits and pieces of state strewn across our application tucked inside of controllers, services, routes, directives, and even dom.

But now we often face the challenge of multiple components that need to alter the same state

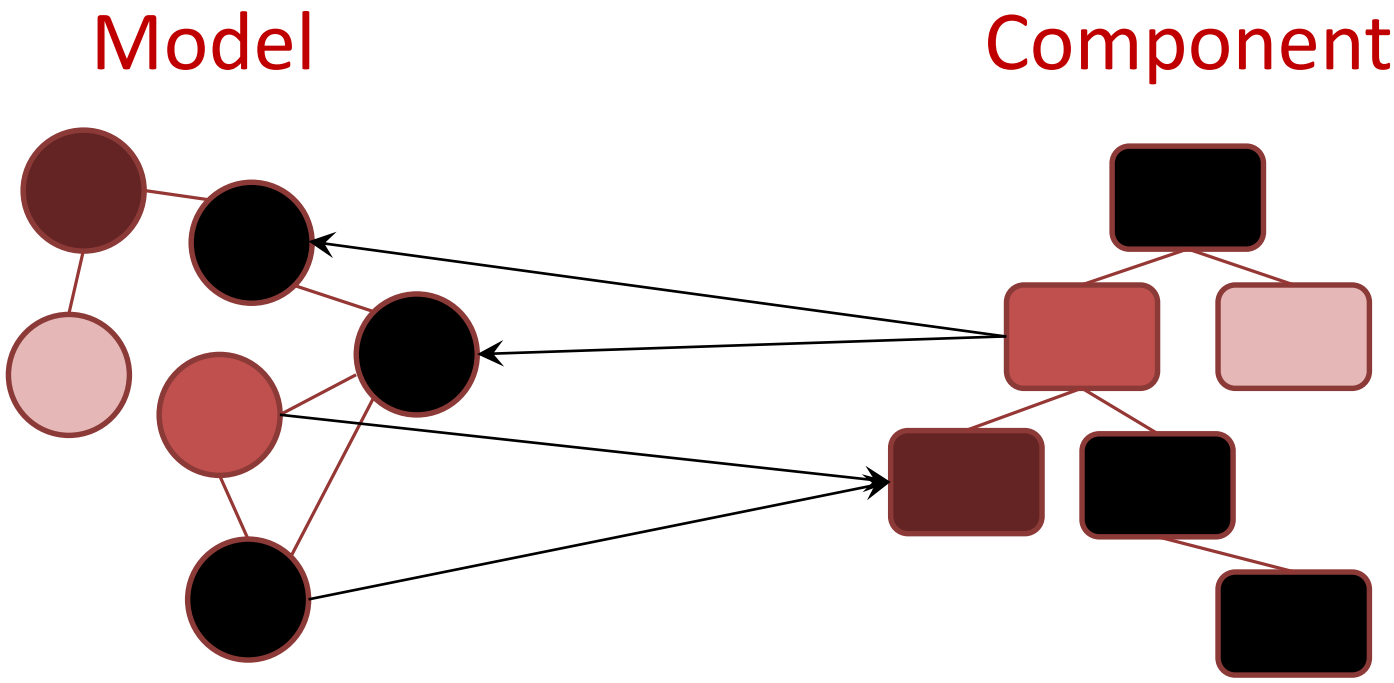
When the application grows, this approach is hard to scale.

Data changing over time is the issue here

- Back in the 90s it was easy…
- "Just refresh the page and new data is shown, yep*, its magic right there*."

Shared Mutable State is a pain

When *everything* can effect *anything*, we soon stop to understand what's going on.



What is it about?

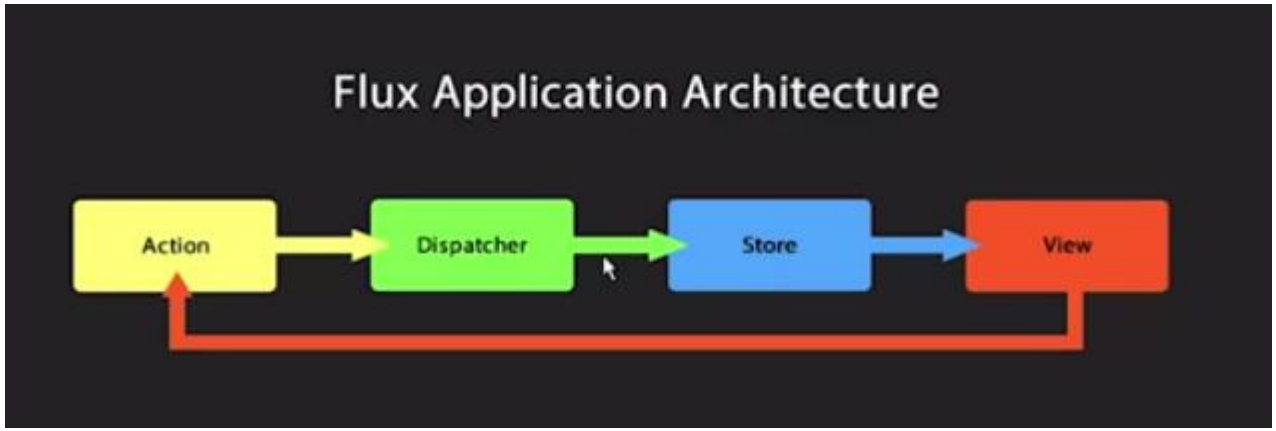
Building Frontends is hard because there is so much state

Much of this state has nothing to do with the server, examples: *isPlaying*, *currentDuration*, wizards.

In larger projects – It's becoming a *mess*

Enters Flux

- It is a design idea, an architecture, not a particular framework
- It is a one way flow of state
- where your components do not change the state of the application directly
- but goes through a dispatcher to do so.



Redux is an evolvement from Flux

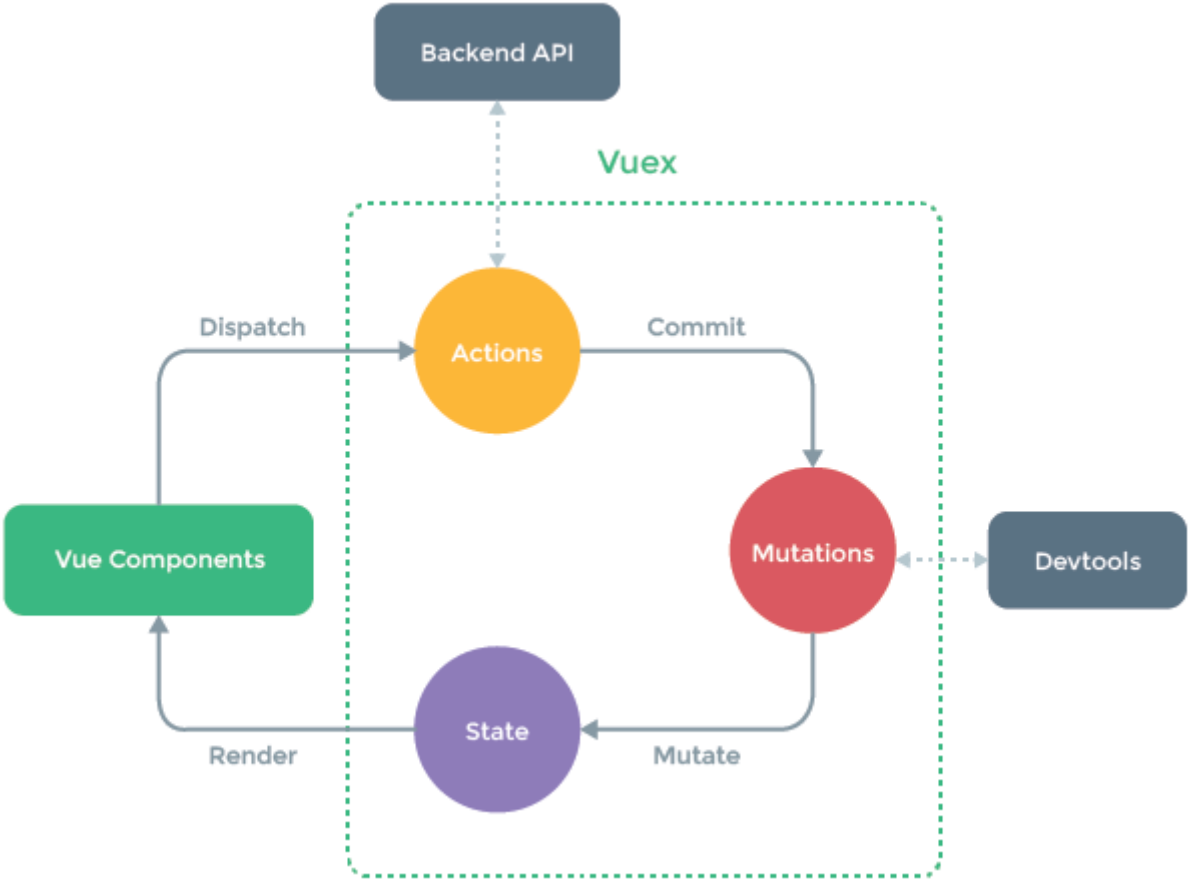
- Just as modern web frameworks like Angular permanently altered our jQuery-centric approach to app development, React has fundamentally changed the way that we approach state-management while using modern web frameworks.

- Redux is front and center of this shift as it introduced an elegant, yet profoundly simple way to manage application state.
- Born and raised at React
  and it works really well with Angular & Vue.

MisterBit

Unidirectional data flow

Extract the shared state out of the components, and manage it in a global singleton



Introducing Vuex

Vuex is an elegant state management library

It can be described in the following fundamental principles:

- The store holds all shared mutable state of the application
- Use mutations to update the state
- Use Actions for async operations

Vuex principles - store

The store is a single source of truth

```
const store = new Vuex.Store({
  strict: true,
  state: {
    count: 1
  }
})
```

Here is a component getting data from the store:

```
const CounterCmp = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () { return this.$store.state.count }
  }
}
```

Vuex principles - getters

You may use getters - reusable accessing logic on the state:

```
const store = new Vuex.Store({
  state: {
    count: 1
  },
 getters: {
      countForDispaly (state, otherGetters) {
        return state.count.toLocaleString()}
  }
})
```

```
const CounterCmp = {
  template: `<div>{{ count }}</div>`,
  computed: {
    count () { return this.$store.getters.countForDisplay }
  }
}
```

Vuex principles -Mutations

The only way to mutate the state is to commit a (synchronous) mutation

```
const store = new Vuex.Store({
   state: {
      count: 1
   },
   mutations: {
       increment (state, payload) { state.count += payload.amount }
   }
})
```

```
// inside some component:
this.$store.commit({ type: 'increment', amount: 10 })
```

Check out the simplest example

MisterBit

Vuex principles - Actions

For Asynchronous operations, use actions

```
actions: {
    incrementAsync ({ commit }, {amount}) {
        setTimeout(() => {
            commit({ type: 'increment', amount: 10 })
        }, 1000)
    }
}
```

```
// inside some component:
this.$store.dispatch({ type: 'incrementAsync', amount: 10 })
```

Check out the simplest example

MisterBit

Vuex principles - Actions

Its common to have the action return a promise, so the using component can be informed that the operation was done.

```
deleteCar(carId) {
   this.$store.dispatch({type: 'deleteCar', carId })
      .then(()=>{
            console.log('Car Deleted')
       })
}
```

Vuex principles - Modules

For bigger apps, divide the store into modules.

Each module can contain its own state, mutations, actions, getters, and even nested 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
```

MisterBit

Why Bother?

- Redux brings to the table constraints:
  - Single state tree
  - Actions to describe updates
  - Mutations as pure functions that apply the updates
- But what do we get back?

Debug workflow

- Log actions and states
- Find the bad state
- Check the action
  - If you see something funny, blame the component
- Fix the mutation
- Write a test

MisterBit

Everything is Data

- Simplified Debug workflow
- Persistence and Offline
- Recording user sessions
  - Good place for collecting analytics
- Error handling
  - Try-catch -> send state and action to server!
- Optimistic mutations
  - Update the UI immediately, If we get an error from server, undo.
- Collaborative editing is simple!
- Universal rendering

MisterBit

Sometimes,
Constraints
actually give you
Features!

Internal Mutable state is allowed

- Keeping application state in one immutable place is a good idea, but don’t go all the way…
  - Components can have local state that can only be updated when their inputs change or an event is emitted in their templates.
- So we allow mutable state, but in a very scoped form

Moving to the CLI

OK

```
.: downloading template
? Project name hybrid-base
? Project description A Vue.js project
? Author F1LT3R <al@pwn.io>
? Vue build standalone
? Install vue-router? Yes
? Use ESLint to lint your code? Yes
? Pick an ESLint preset Standard
? Set up unit tests Yes
? Pick a test runner jest
? Setup e2e tests with Nightwatch? Yes
? Should we run `npm install` for you after the project has been created? (recommended) npm

   vue-cli · Generated "hybrid-base".


# Installing project dependencies ...
```

MisterBit

Scoped style

Getting component scoped CSS is easy!

Try it out

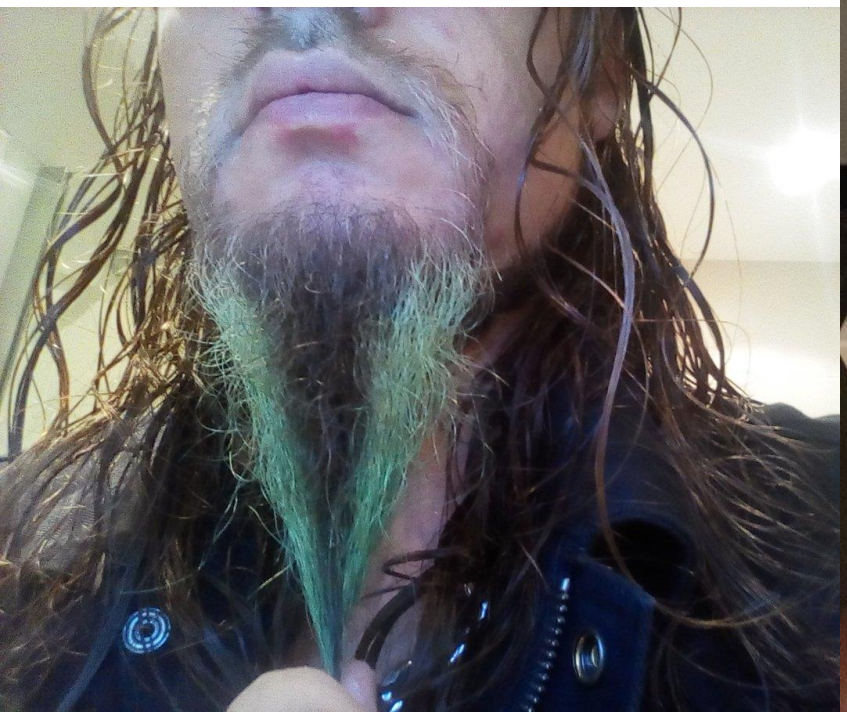
```
<style scoped></style>
```

NOTE:

- Scoped style is supported in .vue files, you can use BEM of other methods instead
- Scoped style is also applied to root element of children

VueJS

Now, go build an amazing app



Extra Stuff

Async Components

- In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it's actually needed.

- A recommended approach is to use async components together with **Webpack's code-splitting feature**:
- Read about it here.

Computed Setter

Computed properties are by default getter-only

- but you can also provide a setter when you need it:

```
computed: {
  fullName: {
  // getter
  get: function () {
    return this.firstName + ' ' + this.lastName
  },
  // setter
  set: function (newValue) {
    var names = newValue.split(' ')
    this.firstName = names[0]
    this.lastName = names[names.length - 1]
  }
 }
}
```

Functional Templates

For pure functional components that only rely on props, you can use:

```
<template functional>
<div>
   <h1>Pet: {{props.pet.name}}</h1>
</div>
</template>
```

USE IT:

```
<pet-preview :pet="myPet"></pet-preview>
```

Another way to achieve this is by setting:
`prop: true` in the routing for this component

In place edit

Forms and inputs are the way to go for updating content, but in some cases, you may want to use contenteditable (e.g. - WIX), here is a sample:

```
<div id="app">
   <span :class="{editable: editMode}"
         :contenteditable="editMode“
         @blur="editMode=false">
             Hello {{userName}}
   </span>
   <button v-show="!editMode"
           @click="toggleEdit">&#9998;</button>
</div>
```

see it in [action]

Mister Bit

Merging parent and child classes

When you use the class attribute on a custom component, those classes will be added to the component's root element.

Existing classes on this element will not be overwritten.

- The same is true for style bindings

```
Vue.component('my-component', {
    template: '<p class="foo bar">Hi</p>'
})
```

```
<my-component class="baz boo"></my-component>
```

```
<p class="foo bar baz boo">Hi</p>
```

Props with binding

All props form a **one-way-down** binding between the child property and the parent one:
when the parent property updates, it will flow down to the child, but not the other way around.

This prevents child components from accidentally mutating the parent's state, which can make your app's data flow harder to reason about.

Note! if the prop is an array or object, mutating the object or array itself inside the child **will** affect parent state (passed by reference)

Child Component Refs

- Despite the existence of props and events, sometimes you might still need to directly access a **child component** in JavaScript.
- To achieve this you can assign a reference ID to the child component using ref:

```
<div id="parent">
    <user-profile ref="profile"></user-profile>
</div>

var parent = new Vue({ el: '#parent' })
// access child component instance
var child = parent.$refs.profile
```

Refs are only meant as an **escape hatch** for *direct child manipulation, don't over use them*

Scoped Slots

- A scoped slot is a special type of slot that functions as a reusable template (that can be passed data to) instead of already-rendered-elements.
- In a child component, simply pass data into a slot as if you are passing props to a component:

```
<div class="child">
    <slot text="hello from child"></slot>
</div>
```

The value of scope is the name of a temporary variable that holds the props object passed from the child:

```
<div class="parent">
  <child>
    <template scope="props">
      <span>hello from parent</span>
      <span>{{ props.text }}</span>
    </template>
  </child>
</div>
```

Scoped Slots

A more typical use case for scoped slots would be a list component that allows the component consumer to customize how each item in the list should be rendered:

```
<my-awesome-list :items="items">
    <!-- scoped slot can be named too -->
    <template slot="item" scope="props">
        <li class="my-fancy-item">{{ props.text }}</li>
    </template>
</my-awesome-list>
```

And the template for the list component:

```
< ul>
   <slot name="item"
       v-for="item in items"
       :text="item.text">
           <!-- fallback content here -->
    </slot>
</ul>
```

Mixins

- Creating and Using Mixins is simple
  - Try a Local Mixin with a created() function
  - See how it does not overrides a create() in the component
- How Mixins get Merged?
  - component is last
- Creating a Global Mixin: Vue.mixin({})
  - Every instance in the app gets it – used by Vue plugins
  - You can use it to count how many Vue instances are created
- Mixins and Scope
  - Mixin with data() – data is not shared, every place get a new copy of the data

Directives

- VueJS provides a set of directives shipped in core (i.e. v-model, v-show)
- Vue also allows you to register your own custom directives.
- This is useful when you are adding some behavior to an element without adding any template markup
- You get low-level DOM access on plain elements

- See here

Render functions

- Vue recommends using templates to build your HTML in the vast majority of cases.
- There are situations however, where you really need the full programmatic power of JavaScript.
- That's where you can use the **render function**, a closer-to-the-compiler alternative to templates.
- See here

Side story about immutability

The Problem of mutable objects

- They make tracking changes hard, both for the developer and the framework.
- They force framework such as Angular to be conservative by default, which negatively affects performance.
- Im Vue, due to its reactivity, this is not a must.

Lets Immutable!

OK, so immutable objects are good.

Now, how do we make our objects immutable?

MisterBit

Const?

Consts don’t help:

```
const user = { name: ‘Puki', age: 40 };
user.age = 41

const users = [{ name: 'John', age: 27 }, { name: ‘Paul', age: 63}];
users.push({ name: ‘Ringo', age: 66})
```

MisterBit

Object.freeze?

ES6 Object.freeze() is only doing *shallow freeze*, not a great help.

Object.assign

Object.assign performs shallow copy

```
var user1 = {
    name: "User 1",
    score: 90,
    address: {
        city: 'Tel Aviv',
        street: 'Nahmani'
    }
};
var user2 = Object.assign({}, user1);
user2.height = 175;
user2.address.street = ‘Nordeo';
```

Object.assign

Object.assign() is useful as it accepts multiple arguments so we can “merge” multiple objects at the same time.

```
var user1 = {
    name: "User 1",
    score: 90,
    address: {
        city: 'Tel Aviv',
        street: 'Nahmani'
    }
};
var user2 = Object.assign({}, user1, {score: 75, address: {street: 'Nordeo'}})
```

Mutating array operations

Adding, removing and updating the people array in a mutable fashion:

```
let addPerson = (person: Person): void => {
    people.push(person);
};

let removePerson = (person: Person): void => {
    let index = people.indexOf(person);
    people.splice(index, 1);
};

let updatePersonAge = (person: Person, age: number): void => {
    person.age = age;
}
```

None mutating array operations

Now without mutating:

```
let addPerson = (people: Person[], person: Person): Person[] => {
    return [
    ...people,
        person,
    ];
};

let removePerson = (people: Person[], person: Person): Person[] => {
    let index = people.indexOf(person);
    return [
    ...people.slice(0, index),
    ...people.slice(index + 1)
    ];
};

let updatePersonAge = (people: Person[], person: Person, age: number): Person[] => {
    let index = people.indexOf(person);
    return [
    ...people.slice(0, index),
        Object.assign({}, person, {age}),
    ...people.slice(index + 1)
    ];
}
```

Summary

- In React / Angular we don’t mutate data
- Our alternatives:
  - Put a big sign in the room and punish developers for mutating?
  - Using clones (using things like the array spread operator, object spread operator, etc)
  - Use immutable data structures (i.e. immutable.js, mori.js, deepFreeze.js, etc)

- In Vue, we can forget about all this stuff and go have a beer.

VueJS Master!

a simple Framework for simple people

MisterBit