Formation Vue.JS

Réaliser un projet Vue.JS en TDD

Auteur: **Jérémie Amsellem** // Consultant Sécurité Offensive

`https://hack.courses/Programming/Vue.JS`

Le principe de base du TDD

Le **TDD** (Test Driven Development) est une manière de développer des logiciels qui met **les tests** (notamment unitaires) **en priorité** par rapport aux premières phases d'écriture de code.

On écrit d'abord les tests et **ensuite** on commençe à coder les fonctionnalités correspondantes.

Dans une logique métier, en architecturant votre projet vous commençerez par dégager des **besoins** auquel il doit répondre.

Besoins qui deviendront des **spécifications**, et ces **spécifications** seront prêtes à être énoncée sous forme de **tests**.

Les étapes du TDD

- **1** - Écrire un test
- **2** - Lancer le test (vérifier qu'il échoue)
- **3** - Écrire le code nécessaire pour que le test passe
- **4** - Optimiser le code écrit

Les Tests Unitaires

Un test unitaire c'est quoi?

Un **test unitaire** a pour but (comme son nom l'indique) de **tester une "unité" d'un programme**.

Ce qu'on appelle une "unité" dans un programme peut être une **méthode, une fonction, une classe, même quelque lignes seulement**...

Si l'"unité" qui doit être testée dans un test unitaire n'est pas nécessairement définie et que le concept est relatif, le but d'un test unitaire est quand à lui assez précis !

À quoi ça sert ?

Le but d'un test unitaire est de tester, que pour un paramètre **A** (le paramètre d'une fonction, une variable, un évènement), l'unité de code testée ait le comportement **B** attendu.

L'avantage est de pouvoir s'assurer à chacune des étapes du développement que les fonctionnalités du projet sont conformes à la manière dont elles ont été architécturées et également de pouvoir éviter et repérer de nombreuses régressions dans le code !

Exemple

Prenons le cas d'une fonction **checkPassword** qui servirait à valider le mot de passe d'un utilisateur.

```
function checkPassword(password) {
}
```

Elle doit lever une exception dans ces conditions :

- Le mot de passe fait moins de 8 caractères
- Le mot de passe est un mot du dictionnaire
- Le mot de passe ne comporte que des chiffres

Si le mot de passe respecte ces règles, checkPassword devra alors retourner **true**.

Nous pourrions écrire les tests de la manière suivante :

```
describe('checkPassword', () => {
    // We start by checking the first rule
    it("shouldn't allow passwords shorter than 8 chars",
        expect(checkPassword("test1")).toThrow();
    });
    // Then the second one
    it("shouldn't allow words from the dictionary", () =>
        expect(checkPassword("aerodynamic")).toThrow();
    });
    // And finally the third one
    it("shouldn't allow only numeric passwords", () => {
        expect(checkPassword("8008135707")).toThrow();
    });
    // A valid password should be accepted
    it("should return true when a valid password is given'
        expect(checkPassword("g00dP455w0rd!")).to.equal(tr
    });
})
```

Premier contact avec Vue.js

Qu'est-ce que Vue.js

Vue.JS est un **framework Web** qui a été crée en 2017 par **Evan You** (ex Googler).

C'est un **logiciel libre** (license **MIT**), qui se veut diffèrer des frameworks front-ends **monolithiques** comme Angular par le fait qu'il se limite **exclusivement à la définition et l'affichage de composants graphiques**.

Il est inspiré du Modèle View View-Model (**MVVM**).

Prérequis et installation

Il est nécessaire pour utiliser et installer les **outils en ligne de commande de Vue.js** d'avoir **Node.js et npm** installés.

Une fois ceux-ci configurés sur votre machine il faut utiliser la commande

> (sudo) npm install -g @vue/cli

Pour installer l'interface en ligne de commande de Vue.js.
Si l'installation s'est correctement déroulée vous devriez pouvoir utiliser la commande **vue**.

Utilisation de vue-cli

Commandes principales

- **create** [options] app-name - *Créer un nouveau projet*
- **serve** [options] (fichier) - *Lancer le serveur HTTP d'un projet*
- **build** [options] (fichier) - *Générer un build de production*
- **init** [options] template app-name - *Créer un projet à partir d'un des templates de https://github.com/vuejs-templates*
- **ui** - *Démarre l'interface utilisateur de gestion des projets Vue.js*

Le concept de "composant"

Les composants Vue.js s'inspirent librement des **spécifications des WebComponents**.

Pour résumer ce qu'est un composant: un composant est un bloc **réutilisable** et **pouvant lui même contenir d'autres composants** associant :

- Du style (CSS)
- Un template (HTML)
- De la logique (JavaScript)

L'**intérêt principal** de l'utilisation de composants est **architectural**, ils permettent d'**éviter de dupliquer du code** et d'avoir des **templates HTML lisibles**.

Mount et ShallowMount

Le module *@vue/test-utils* inclut plusieurs classes et méthodes qui facilitent les tests unitaires des composants.

Notamment les fonctions **mount et shallowMount** qui permettent d'instancier un composant à l'intérieur d'un **wrapper**, ce qui rend possibles les intéractions avec des composants au sein des tests.

```
const wrapper = mount(MyComponent)
assert(wrapper.name() === 'MyComponent')
```

La différence entre mount et shallowMount est que shallowMount remplace les méthodes et composants enfants par des *stubs* : des éléments vides de remplacement.

Les principales méthodes de l'objet Wrapper

Un wrapper possède plusieurs méthodes nous permettant d'intéragir avec un composant monté, voici une liste non exhaustive de celles dont nous nous servirons :

- **vm** : Contient l'instance de l'objet Vue crée.
- **find(*CSSselector*)** : Chercher un élément par son sélecteur
- **findAll(*CSSselector*)** : Chercher plusieurs éléments
- **setData(*data*)** : Mettre à jour la **data** d'un composant
- **setProps(*props*)** : Mettre à jour les **props** d'un composant
- **html()** : Retourne une chaîne de texte contenant la représentation HTML du composant
- **text()** : Retourne le texte brut compris dans le composant

Hands-on!

- Création de notre premier projet Vue.js
- Présentation du projet Message Board
- Spécifications et premières modifications

Les principales options des composants

- **name** : Le nom du composant
- **data** : Une fonction qui retourne les attributs du composant
- **methods** : Les méthodes explicitées du composant
- **components**: Les dépendances du composant
- **props** : Les attributs passés par le composant parent
- **computed** : Les valeurs à traiter avant leur affichage

La liaison de données (data-binding)

Le premier moyen de lier une information entre deux composants est le **data-binding parent vers enfant**.

On envoie une information d'un composant **parent** vers **un autre composant** qui est utilisé dans son template (son composant **enfant**).

```
<template>
        <enfant></enfant>
</template>
<script>
export default { name: "Parent" }
```

Utilisation des props

Pour qu'un composant puisse reçevoir des informations de ses parents, il est nécessaire de définir les attributs qui pourront lui être envoyés.

C'est le rôle de la méta-donnée "props": elle contient des définitions de l'ensemble des paramètres qu'un composant pourra accepter.

> **Note**: Essayez d'être aussi précis que possible dans la définition de vos props

```
export default {
        props: {
                author: {
                        type: String,
                        required: true,
                        validator: () => {}
                },
                content: String }}
```

La liaison de données parent vers enfant

Plusieurs manières d'afficher de la donnée dynamiquement dans le template d'un composant

- **moustaches** :

```
<h1>{{ data }}</h1>
```

- **v-text** :

```
 <h1 v-text="data"></h1> // n'est pas watchée
```

- **interpolation d'expressions dans des moustaches** :

```
<h3>{{ data + new Date() }}</h3>
```

Les slots

Il est également possible de faire passer des informations de template d'un composant parent vers un composant enfant en utilisant des slots.

Pour ce faire, on utilise une balise **slot** dans le composant enfant à l'endroit où on souhaite afficher le template passé par le parent.

Dans le composant parent il suffira de mettre un template HTML entre les balises du composant enfant, comme ceci :

```
<my-component>
        <h1>This will replace the slot in MyComponent</h1>
</my-component>
```

Les directives

Les directives **sont des composants ne comportant pas de template**.

Elles permettent d'associer de la logique à un élément de template **déjà existant**.

Il est possible de définir nos propres directives, en créant un objet Vue.directive (nous y reviendrons) mais il existe déjà un certain nombre de directives **prédéfinies dans le framework**.

Les directives Vue.js prédéfinies

- **v-text** : Permet de lier une variable au contenu 'text' de l'élément sélectionné
- **v-bind (alias ‘:’)** : Permet de lier une variable locale à un attribut d'un élément de votre template
- **v-if** / **v-else-if** / **v-else** : Permet d'ajouter des conditions à l'ajout d'un élément dans le DOM
- **v-show** : Permet d'afficher ou non un élément (display en CSS) en fonction d'une condition
- **v-for** : Permet d'itérer sur les éléments d'un tableau
- **v-on** (alias ‘@’) : Permet de lier un comportement dans notre composant à un évènement
- **v-model** : Permer de lier la valeur d'un input utilisateur (input, checkbox, radio, select…) à une variable de notre composant

Le v-on en détail

La directive Vue.js v-on (ayant pour alias @) permet d'associer un événement JavaScript avec du comportement défini en JavaScript.

Par exemple pour déclencher une fonction au clic sur un bouton on fera comme ceci dans un composant :

```
<template>
  <button v-on:click="doSomething()"></button>
</template>
<script>
  export default {
        doSomething() {
                alert('Button clicked');
        }
  };
</script>
```

Hands-on!

- Création de données de Mock
- Création de notre composant MessagesList
- Premières intéractions entre composants

La liaison de donnée enfant vers parent

Nous allons maintenant voir comment faire passer une information du composant enfant vers le composant parent.

Le paradigme est ici légèrement différent, on va utiliser le système d'évènements pour remonter des informations du composant enfant vers le composant parent.

On déclenche notre évènement dans le composant enfant avec la méthode

```
this.$emit('hello')
```

Il suffira ensuite d'écouter cet évènement dans le composant parent avec la directive **v-on**.

```
<parent v-on:hello="doSomething()"/>
```

Utilisation avancée du wrapper

*Il est maintenant temps de voir d'autres méthodes du **wrapper** qui vont nous servir dans nos prochains tests :*

- **contains(*selector or component*)** : Retourne **true** si un élément est contenu dans le composant. Sinon **false**.
- **wrapper.emitted()** : Retourne un objet contenant les évènements émis par le composant.
- **wrapper.trigger()** : Permet de déclencher un évènement dans le **DOM** de l'élément sélectionné.

> **Note**: wrapper.trigger peut être utilisé à la suite d'un find(), par exemple :

```
wrapper.find('button').trigger('click')
```

Le cycle de vie des composants

L'**affichage**, la **mise à jour** et la **destruction** des composants Vue.js sont des actions gérées par le framework, nous n'avons pas à spécifier lorsqu'il faut ajouter ou retirer un composant du DOM.

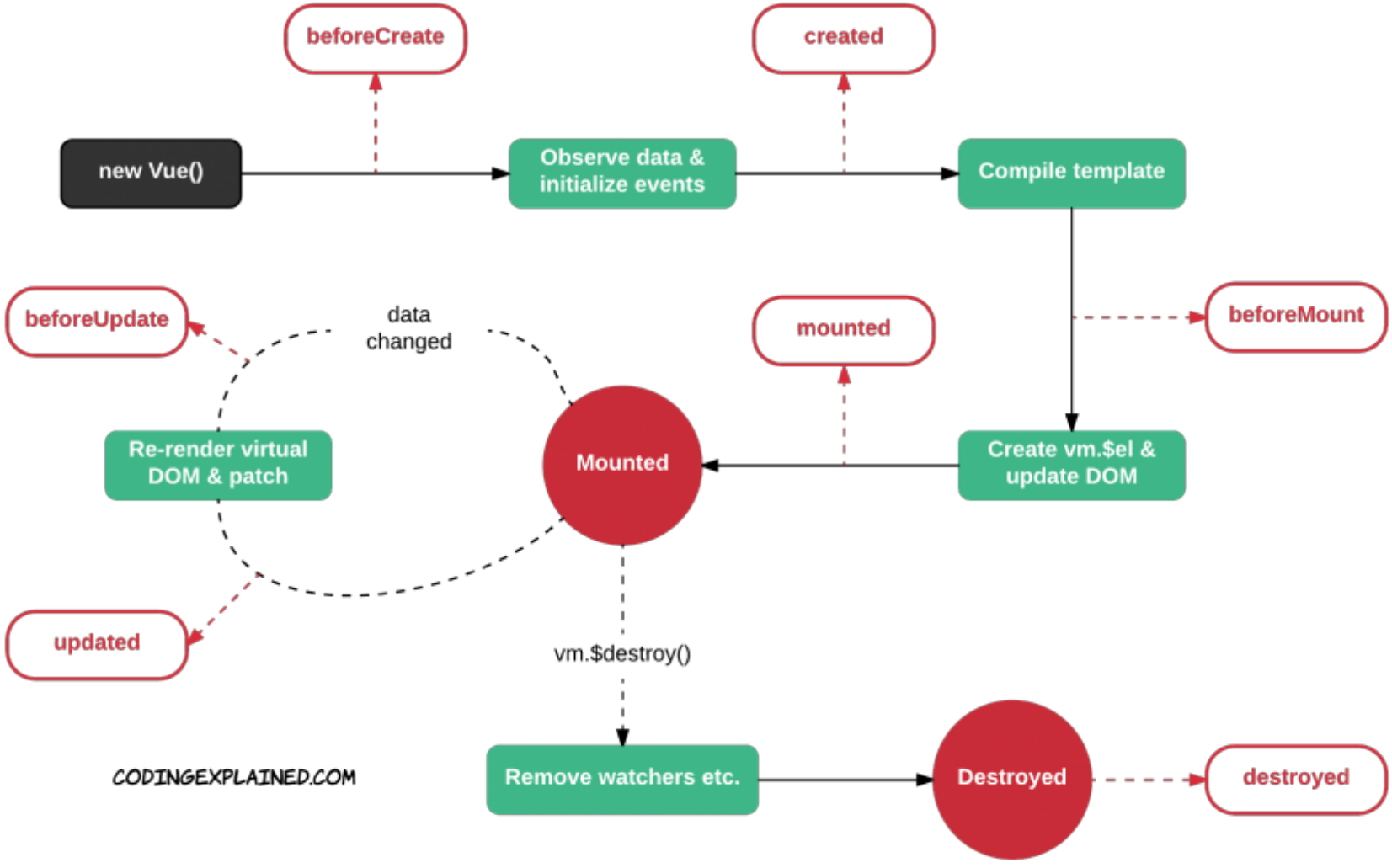
En revanche, Vue.js nous permet d'effectuer certaines actions à des moments précis du cycle de vie de nos composants à l'aide de **crochets (hooks)**.

Pour les utiliser, il suffit d'implémenter les méthodes éponymes à l'intérieur de notre composant.

Les hooks qui nous seront les plus utiles sont :

Created, Mounted, Updated et Destroyed.

Schéma du cyle de vie d'un composant



Le concept de Store

(State Management Pattern)

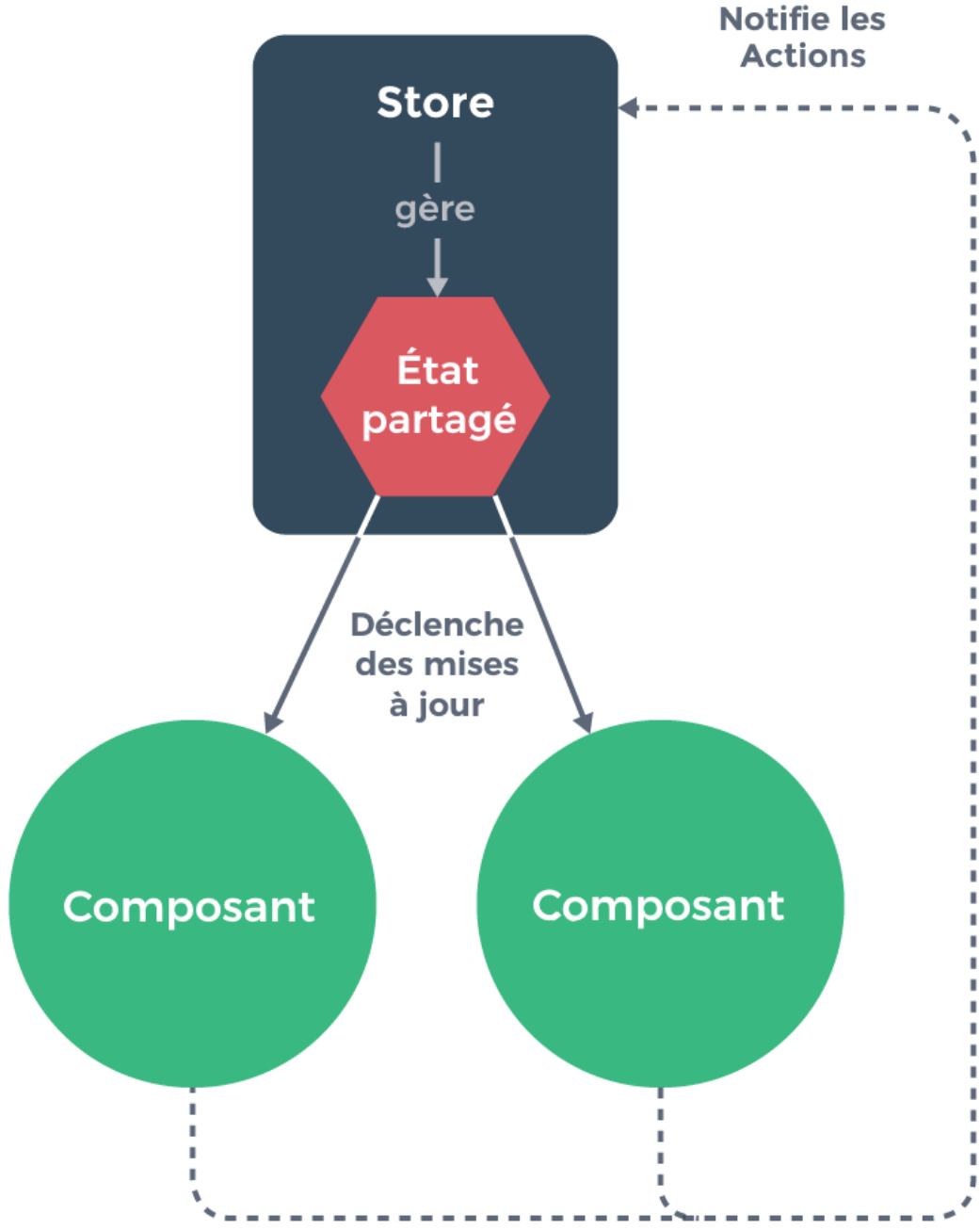
Vue.js utilise **Vuex** comme système de gestion d'état.
Vuex est similaire à Redux dans son fonctionnement en de nombreux points :

Vuex et Redux permettent de créer des **stores**.

Un **store** sert à garder un **état (state)** en mémoire au sein d'une WebApplication.

Tous nos **composants** peuvent effectuer des **modifications (mutations)** sur cet état.

Les **composants** peuvent à tout moment accéder à l'**état** du store et être **notifiés** en cas de modifications de celui-ci.


Notifie les
Actions
Store
gère
État
partagé
Déclenche
des mises
à jour
Composant
Composant

Utilisation de Vuex

Vuex est un module séparé du module core de Vue.js.

Il faut l'installer (si ce n'est pas déjà fait)

> npm install vuex --save

et ensuite ajouter le module à notre projet Vue en ajoutant dans notre main.js :

```
import Vuex from 'vuex'
Vue.use(Vuex)
```

Nous allons maintenant pouvoir créer notre premier **store**!

Le routage

Le routeur de Vue.js est un module séparé du module *core* Vue.js.

Une fois qu'il est installé il permet d'instancier un objet **Router**, dont le constructeur prend en paramètre une liste des routes de notre application sous forme de tableau.

Chaque route doit au minimum décrire un **path** et y associer un **composant** ou une **redirection**.

Il est d'usage d'ajouter **un nom** à chacune des routes crées.

> **Note**: On peut utiliser des wildcards (*) sur les **path** des routes. Ce qui permet par exemple de matcher toutes les routes invalides avec `{path: '*'}`

Hands-On!

- Création d'un store et de mutations
- Ajout des parties utilisateur de l'application

Le mapState

Maintenant que nous avons un peu plus en main l'utilisation de **Vuex**, nous pouvons aborder les **mapGetters** et **mapState** qu'il propose.

mapState permet de mapper des propriétés de l'état d'un store à une propriété *computed* d'un composant.

Il prend en paramètre un tableau :

```
export default {
        computed: mapState(['login', 'avatar', ])
        // this.login && this.avatar
}
```

Ou un tableau associatif, permettant de nommer les propriétés *computed* :

```
computed: mapState({userLogin: 'login',
userAvatar: 'avatar' })
```

Le mapGetters

Nous utiliserons en priorité le **mapGetters**, il est conseillé d'utiliser des getters plutôt que directement le **state** d'un store !

Celui-ci fonctionne comme **mapState** mais associe un **getter** à **une propriété computed**.
La syntaxe est aussi quelque peu différente, on utilisera le *spread* d'ES6 ici car mapGetters retourne un tableau:

```
export default {
        computed: {... mapGetters(
                        ['getLogin',
                         'getAvatar']) }
        // this.getLogin && this.getAvatar
}
```

Liaison de classes et styles

Nous avons jusqu'ici eu une utilisation relativement classique de la directive v-bind.

Il est en réalité également possible de s'en servir pour associer du style et des classes à des propriétés JavaScript.

Vue.js permet de gérer simplement les modifications de style et de classes à l'aide d'objets, plutôt que de chaînes de caractères.

Par exemple, pour changer la classe et la couleur du texte d'un élément en fonction d'attributs d'un composant :

```
<div :style="{color: myCustomColor}">
  <span :class="{selected: isSpanSelected}"></span>
</div>
```

Le pattern EventBus

Jusqu'à présent nous avons émis nos évènements directement depuis nos composants.
C'est la manière basique d'écrire une gestion d'évènements en Vue.js, nous allons maintenant aborder le pattern **EventBus** qui propose de centraliser la gestion des évènements dans un seul composant Vue.js.

L'intêret est de faire communiquer facilement des composants sans qu'ils aient une relation de parenté.

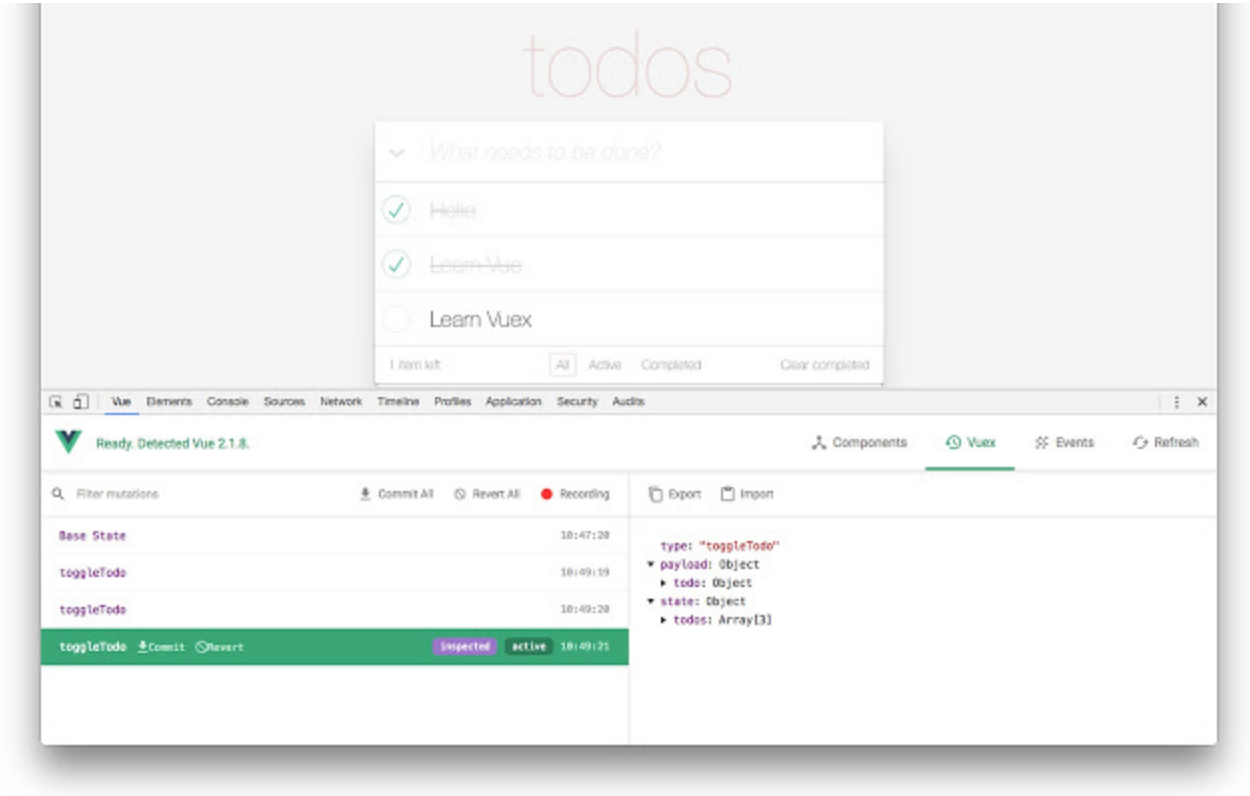
```
export default new Vue()
```

```
import eventBus from '@/eventBus'
eventBus.$emit('refresh') // Pour l'envoi
```

```
eventBus.$on('refresh', () => {...}) // Pour l'écoute
```

Outils de Débogage

Il existe une suite d'outils *officielle* pour Vue.js : les **Vue DevTools**, disponibles sous forme d'extension **Chrome** et **Firefox**.



Une fois l'extension installée, un onglet **Vue** est disponible dans les outils de développement du navigateur.

axios

Axios est une bibliothèque JavaScript pouvant être utilisée côté client et serveur (en Node.js) et qui permet de faire des requêtes réseau.

Vue.js ne prenant pas en charge la gestion des requêtes **HTTP**, nous utiliserons **axios** pour effectuer nos échanges.

```
import axios from 'axios'

axios
.get('https://hack.courses')
.then(response => console.log(response))
.catch(e => console.error(e))
```

Les mixins

Il n'est pas nécessaire de réecrire 100% du code de vos composants si certaines parties sont similaires.

Vue.js intègre un système de **mixins** très simple à prendre en main.

Vous pouvez séparer la partie du code que vous souhaitez partager dans une **mixin** :

```
const annoyingMixin = {
        created() {
                console.log(this.name + ' created');
        }
}
```

Il suffit ensuite de l'ajouter à la propriété **mixins** d'un composant :

```
export default { mixins: [annoyingMixin] }
```

Les modules Vuex

Jusqu'ici nous nous sommes contentés de mettre toutes les informations et mutations de notre **store** dans un module unique.

Notez qu'il est également possible de découper un **store Vuex** en modules, chaque module possédant ses propres **mutations, actions, getters, modules** et son **état**.

On va définir un module de la sorte :

```
const module = {
            state: {...},
            mutations: {...},
            getters: {...}
}
```

Et l'ajouter à notre store :

```
const store = new Vuex.Store({
        modules: { a: module } //  => store.state.a
```

Écriture de directives personnalisées

Nous avons souvent utilisé les directives prédéfinies de Vue.js, il est également possible de créer nos propres directives personnalisées.

La syntaxe est légèrement différente à la création d'un composant :

```
Vue.directive('bold', {/*directive content*/})
```

Les directives possèdent également un cycle de vie différent des composants, avec les **hooks** suivants :

- **bind(*element, bindings*)** : Appellé la première fois qu'une directive est liée à un élément
- **inserted(*element, bindings*)** : Appellé quand l'élément lié a été injecté dans son parent.
- **update(*element, bindings*)** : Appellé après chaque changement du composant contenant la directive.

Element et bindings ?!

Ces deux paramètres sont passés aux **hooks** des directives Vue.js.

Element contient une référence à l'élément du **DOM** sur lequel est appliquée la directive.

Bindings est un objet plus vaste : il contient plusieurs informations sur la liaison entre notre directive et l'élément auquel elle est rattachée :

- **name** : le nom de la directive
- **value** : la valeur passée à la directive
- **expression** : l’expression liée en tant que chaine de caractères
- **arg** : l’argument passé à la directive
- **modifiers** : un objet contenant les modificateurs de la directive

Filtres

Vue.js permet de créer des filtres qui pourront être utilisés pour formater des informations avant de les afficher.

Les filtres s'utilisent comme ceci dans un template :

```
{{ unixTimeStamp | dateFilter }}
```

La définition d'un filtre utilise **Vue.filter** et utilise une fonction de transformation à laquelle sera passée la valeur à filtrer.

```
Vue.filter('dateFilter', (value) => moment(value).format(
```

WebSockets

Les **WebSockets** ont un fonctionnement inspiré des **sockets** TCP UNIX.

Elles sont basées sur le protocole **HTTP** et permettent d'établir une connexion continue entre un client et un serveur.

Il existe une **API** JavaScript permettant de créer simplement une WebSocket et d'écouter ses évènements.

```
const ws = new WebSocket(url);
ws.onmessage = (message) => console.log(message);
```

Fichier de style global

Dans une logique "**D**on't **R**epeat **Y**ourself", il est possible grâce à WebPack de définir un fichier de style global pour notre application.

Les modifications de configuration de notre projet Vue.js se feront dans un fichier **vue.config.js**.

Pour ajouter un fichier SCSS global, il faudra y ajouter :

```
module.exports = {
  css: {
    loaderOptions: {
      sass: {
        data: `@import "@/style/global.scss";`
      }}}
```

Ce qui forcera WebPack à charger le fichier /src/style/global.scss

> **Note**: Il est évidemment possible de charger plusieurs fichiers de style

Les transitions

Vue.js possède un système de gestion des transitions d'entrée et de sortie liées au **v-if et v-show**.

La première étape de leur utilisation est de spécifier qu'un élément nécessitera une transition sur son apparition ou sa disparition en utilisant le composant **transition** :

```
<transition name="fade-in">
        <div v-if="show">Hello World</div>
</transition>
```

Il va maintenant falloir définir la transition nommée "fade-in" que nous allons utilisée.

Nous avons plusieurs classes à disposition, qui seront utilisées pour définir l'état d'un élément dans sa transition. Ce sont :

- **{transition}-enter** : *État de départ, avant que l'élément soit inséré.*
- {transition}-enter-active : *État actif appliqué pendant toute la phase d'animation **enter**.*
- **{transition}-enter-to** : *État de fin du enter, vient remplacer le {transition}-enter.*
- **{transition}-leave** : *État de départ, avant que l'élément soit retiré*
- {transition}-leave-active : *État actif appliqué pendant toute la phrase d'animation **leave**.*
- **{transition}-leave-to** : *État de fin du leave, vient remplaçer le {transition}-leave.*

Exemple de transition CSS utilisant les classes de transition

```
.fade-in-enter-to {
  opacity: 1;
}
.fade-in-enter-active, .fade-leave-active {
  transition: opacity .5s;
}
.fade-in-enter {
  opacity: 0;
}
```

Déploiement

Pour déployer votre projet en production, c'est très simple avec Vue.js, il suffit d'utiliser le script

> npm run build

Celui-ci va minifier/compiler l'ensemble de vos fichiers pour tout assembler dans un dossier **dist** contenant votre projet prêt à être déployé en production sur votre serveur web!