Chapitre 5
Architecture web

Plan

- Architecture de haut niveau
- Patrons de conception d'interface
  - MVC
  - MVP
  - MVVM
- *Templating*
- Application: VueJS

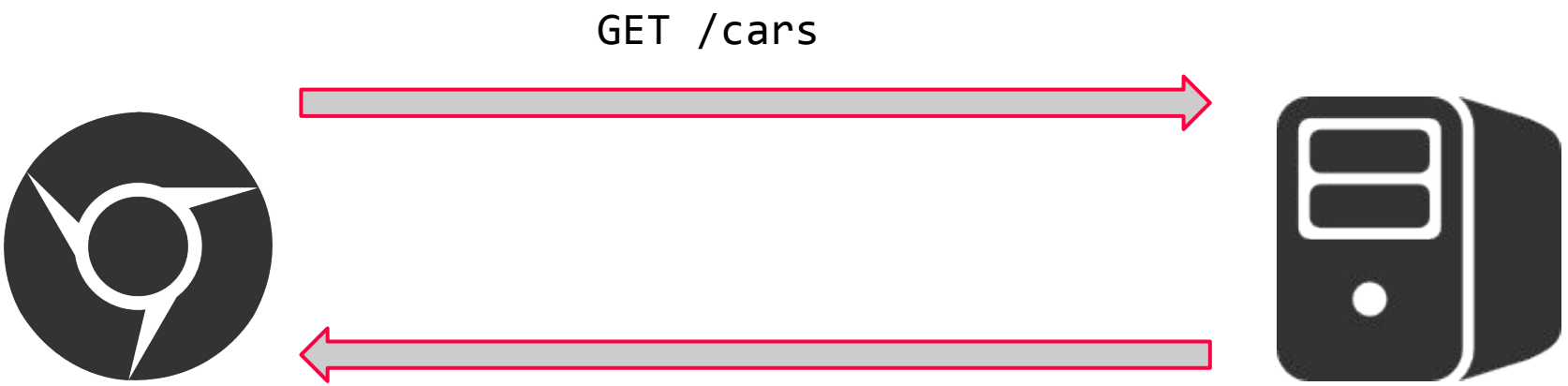
2 écoles de pensée…

Nous commençons à discuter d'**application** web…
et de haut niveau, il existe typiquement 2 façons de voir les choses.

- La manière 'classique' : server-side
- La manière 'moderne' : client-side

Server-side

Le client fait des requêtes REST vers le serveur, qui renvoie les pages HTML complètes!



Logique de présentation seulement : ne fait qu'afficher ce qu'il reçoit!

Fait tout le processing et génère la page complète!

Server-side

Existe encore beaucoup aujourd'hui

- ASP.NET
- Spring
- PHP
- Ruby on Rails
- Django

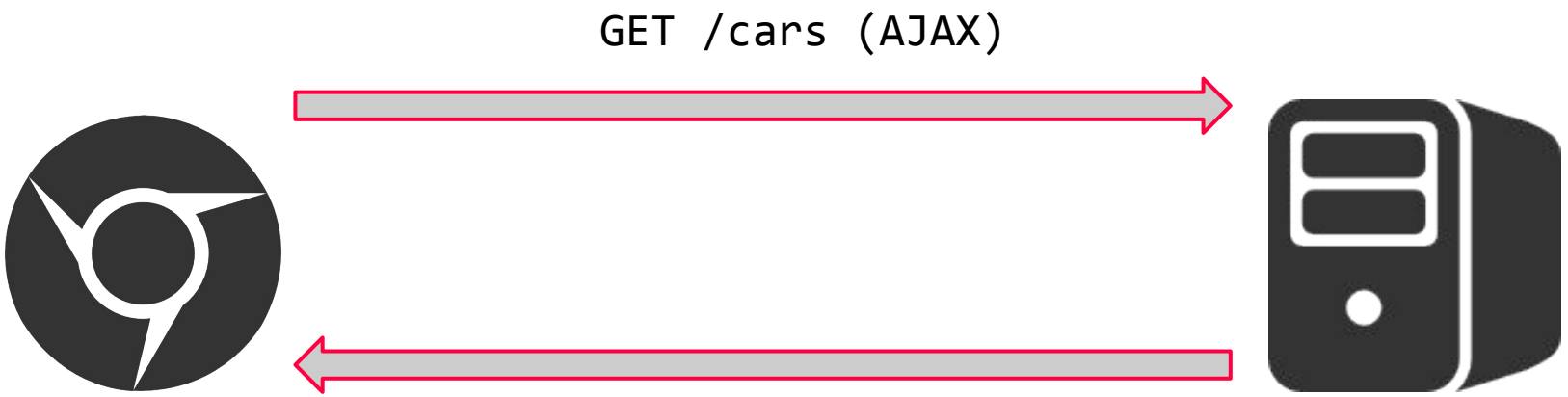
⇒ Très **performant** (évidemment, le navigateur n'a pas grand chose à faire…

⇒ Couple le **déploiement** du serveur et du client.

⇒ Demande un redéploiement complet pour une modification de style…

Client-side

Le client fait des requêtes AJAX vers le serveur qui n'expose qu'une API et renvoie du JSON.



Interprète les résultats et se charge de mettre à jour les éléments pertinents de la page.

```
{
    data: [{
            “make” : “Porsche”,
...
```

Génère simplement une réponse JSON pour la requête!

Client-side

La manière *server-side* est toujours d'actualité… dans certains cas.

La majorité des applications sont développées comme client-side… avec une twist hybride pour éviter les problèmes de CORS.

- Déploiements indépendants, équipes indépendantes!
- Interchangeabilité des deux parties

Client-side

Côté client, le but est d'arriver à une **single page application**.

- Pourquoi recharger les header/footer de ce monde alors qu'ils ne changent pas?
- Chaque particule de l'application peut utiliser l'API… ou non.
- Optimiser la performance au maximum, les changements au DOM sont extrêmements coûteux.

Patrons de conception

M V C
M V P
MVVM

Model View Controller

Model View Presenter

Model View ViewModel

MVC, MVP, MVVM

- Les 3 patrons d'apporter des solutions pour :
  - Découpler la logique des données et de l'interface
  - Séparation des responsabilités

- **Vue** (Interface Utilisateur)
- **Modèle** (Données affichées dans le UI)
- **« Colle »** (Gestion d'évènements, logique, etc)

- Vue et Modèle ont des définitions semblables dans les 3 patrons
- La « Colle » est la partie qui diffère.
  - C : Controller
  - P : Presenter
  - VM : View Model

Il s'agit de **design patterns d'interface!**

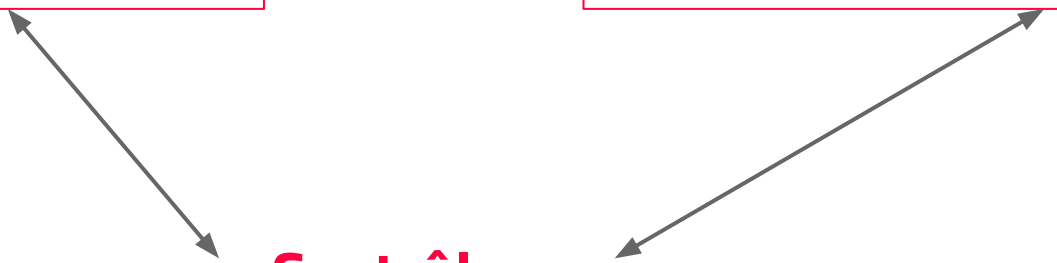
MVC, MVP, MVVM

Vue

```
<html>
     <head>
          <title>
               MyPage
          <title>

…
```

Modèle

```
{
    data: [{
              “make” : “Porsche”,
...
```



Contrôleur

Fait l’association entre les **vues** et les **modèles**… est donc responsable de la logique!

MVC, MVP, MVVM

- Modèle
  - Représentation des données/domaine
  - Ne connaît **PAS** les notions de vue, contrôleur, etc.
- Vue
  - Représentation (visuelle) des données
  - *Templates*
  - Ne connaît **PAS** les notions de modèle, contrôleur, etc.
- Controller, Presenter, ViewModel
  - Communique avec la vue et le modèle
  - Logique d'affaire

MVC

Model View Controller

(pas Model View Colle…)

MVC

- Inventé par des développeurs *Smalltalk* (1979)
- Une grande majorité de frameworks reposent sur ce patron
  - Django, Ruby on Rails, CakePHP, SpringMVC, ASP.NET MVC, Play!, etc.

* Dans les débuts de AngularJS (maintenant plus près de MVVM)

MVC

- **Contrôleur**
  - Lien entre l'utilisateur et l'application
  - Les évènements dans la vue lancent des actions que le contrôleur utilise pour modifier le modèle et déterminer quelle vue doit-être affichée ou mise-à-jour.
  - Il peut y avoir plusieurs vues pour un contrôleur et vice-versa.

- En server-side
  - Le contrôleur déterminer quelle vue doit être affiché
  - La vue envoie des évènements au contrôleur via une requête HTTP (URL) qui est acheminé (« routed ») au contrôleur approprié et à la méthode adéquate.
- En client-side
  - Les événements sont des événements JavaScript lancés par les interactions de l'usager.

MVC

MVC utilise activement les concepts déjà présentées auparavant (AJAX, REST etc.)

⇒ Il s'agit vraiment d'une couche architecturale **au-dessus** !

SpringMVC fonctionne selon la manière **classique**, c'est-à-dire que tout le traitement est fait server-side.

⇒ MVC peut être implémenté autant **server-side** que **client-side**!

(Nous allons faire du client-side, ça s'en vient…)

MVP

Model View Presenter

MVP

- Dérivé du MVC
- Popularisé par Microsoft ASP.NET
- Utilisé dans GWT, GWTP, Vaadin, Swing

- Communication avec la vue
  - Communication bidirectionnelle avec la vue
  - La vue communique avec le Presenter en appelant directement des fonctions du Presenter.
  - Le Presenter communique avec la vue en appelant une interface implémentée par la vue
  - Il y a (habituellement) un Presenter pour une vue.

M V VM

Model View ViewModel

MVVM

- Popularisé par Microsoft avec Windows Presentation Foundation (WPF) et Silverlight
- Utilisé dans KnockoutJs, KendoUI, KnockBack.JS, **VueJS**, AngularJS*, etc.

- Communication avec la vue
  - Communication bidirectionnelle avec la vue
  - Le ViewModel représente la vue (Les attributs du ViewModel correspondent plus à la vue qu'au modèle)
  - La vue est attaché (« bound ») au modèle. Dès qu'un changement au ViewModel est effectué, la vue est instantanément mise-à-jour (vue **active** au lieu de passive)

- La plupart des frameworks JavaScript modernes s'inspirent de ce patron (mais ne l'appliquent pas à 100%)

MVVM

Pas de contrôleur ou de présenteur?

⇒ Le ViewModel observe les changements du modèle et s'occupe de mettre à jour la vue en conséquence.

⇒ Typiquement moins de logique qu'un **contrôleur** traditionnel.

⇒ Ressemble au pattern **Observer**.

MVVM

Model

```
// Model
var data = {
      "Id": 1001,
      "SalePrice": 1649.01,
      "ListPrice": 2199.00,
      "ShortDesc": "Taylor 314CE",
      "Description": "Taylor 314-CE Left-Handed Grand Auditorium Acoustic-Electric
Guitar"
};
```

View

```
<h1 data-bind="text: shortDesc"></h1>
<div data-bind="text: description" class="descArea"></div>
<span class="label label-success" data-bind="text: formatCurrency(salePrice)"></span>
```

ViewModel

```
// ViewModel
var viewmodel = {
      id: ko.observable(data.Id),
      salePrice: ko.observable(data.SalePrice),
      listPrice: ko.observable(data.ListPrice),
      shortDesc: ko.observable(data.ShortDesc),
      description: ko.observable(data.Description),
      formatCurrency: function(value) {
            return "$" + value().toFixed(2);
      }
};
// Bind the ViewModel to the View using Knockout
ko.applyBindings(viewmodel);
```

http://jsfiddle.net/willyfortin/evFyQ/

MVVM

Comment est-ce que les mises à jour du modèle sont effectuées?

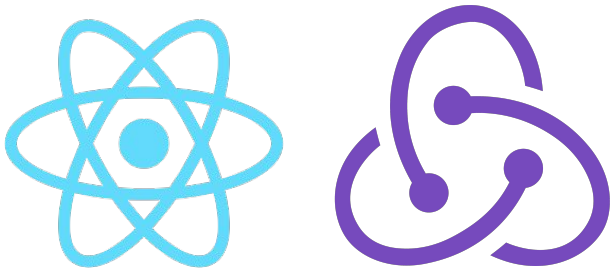
```
<div>
    You've clicked <span data-bind="text: numberOfClicks"></span> times
    <button data-bind="click: incrementClickCounter">Click me</button>
</div>

<script type="text/javascript">
    var viewModel = {
        numberOfClicks : ko.observable(0),
        incrementClickCounter : function() {
            var previousCount = this.numberOfClicks();
            this.numberOfClicks(previousCount + 1);
        }
    };
</script>
```

Les annotations **data-bind** permettent de définir les actions auxquelles le ViewModel réagissent! (exemple avec KnockoutJS)

React/Redux?

Certains d'entre vous ont peut-être déjà fait du **React/Redux**. Nous ne verrons pas ces *frameworks* dans le cours, mais notez bien que…

- React n'est **pas** un framework MVC.
- Redux est un framework utilisant le patron **Flux**. Il s'agit d'un paradigme un peu plus avancé que nous voyons dans la suite du cours.



MV*

- Ne pas se perdre dans les patterns
- Le but principal est la **séparation des responsabilités!**
- **Google a même déclaré AngularJS comme étant framework MVW (Model View Whatever)**
- Toujours de la magie -> Comprenez ce que vous faites!!

Igor Minar, Employé de Google à propos d'AngularJS

Templating

Templating

Le **templating** est un mécanisme permettant de mettre à jour facilement une page HTML avec de nouvelles informations.

L'engin de templating (il en existe de multiples) possède son propre interpréteur, et sert à remplacer des **annotations** spécifiques par des paramètres fournis.

⇒ Le résultat est du HTML, mais qui a été en quelque sorte "**compilé**".

Templating

En JavaScript

Framework permettant de transformer du “HTML” avec balises spécifiques en du HTML compréhensible par le navigateur…

- **Facilité de réutilisation**
- **Découpage**
- **Plus performant pour le navigateur -> ne pas recompiler de templates inutilement**

Templating

De multiples engins de templating existent :

- JSP (avec Spring)
- MVC Razor
- UnderscoreJS
- Handlebars : handlebarsjs.com
- Mustache

Templating

Exemple classique en JSP :

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ page session="false"%>
<html>
<body>
<c:import url="navbar.jsp" />
      <div class="container">
      <c:import url="header.jsp" />
      <div class="row-fluid">
            <h1 class="whitetext span6 well centered title">${title}</h1>
                  <c:if test="${searchresults != null}">
                  <span class="span12">
                        ${searchresults} result<c:if  test="${searchresults > 1}">s</c:if> found.
                  </span>
                  </c:if>
            <c:import url="carlisttemplate.jsp"></c:import>
      </div>
      </div>
      <c:import url="footer.jsp" />
</body>
</html>
```

⇒ Ressemble à du HTML, mais avec des balises spécifiques à l'engin courant.

Templating

Soit le cas d'utilisation suivant :

```
Vous avez ce HTML :
<div id=”cars”>
    <div>Toyota Corolla</div>
    <div>Toyota Yaris</div>
    <div>Toyota 4Runner</div>
</div>
```

Et vous désirez le **mettre à jour** suivant une requête au serveur… qui vous a renvoyé les informations suivantes :
[“Toyota Corolla”, “Toyota Yaris”]

Que feriez-vous avec vos connaissances actuelles?

Templating

Solution basée sur ce que nous avons vu :

1. Supprimer les éléments du div courant.

2. Créer des nouveaux éléments pour chaque résultat.

Pas très performant, et **beaucoup** de code pour peu de valeur.

Templating

Solution utilisant le templating (Handlebars)

1. Définition de la template

```
<script id="cars-template" type="text/x-handlebars-template">
   {{#each cars}}
   <div>{{this}}</div>
   {{/each}}
</script>
```

2. Définition de l'élément HTML

```
<div id="cars"></div>
```

Templating

Solution utilisant le templating (Handlebars)

3. Appel de la template

```
<div id="cars"></div>
<script>
   // Une seule fois au début. Compile la template.
   const source   = $("#cars-template").html();
   const template = Handlebars.compile(source);

   // Appel de la template avec le nouveau data. 2 seules lignes de code!
   const data = {cars : ["Toyota Corolla", "Toyota Yaris"]};
   $("#cars").html(template(data));
</script>
```

Templating

Solution utilisant le templating (Handlebars)

code complet :

```
<!DOCTYPE html>
<html>
<head>
   <title>Handlebars.js Demo</title>
   <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.10.1/jquery.min.js"></script>
   <script
src="http//cdnjs.cloudflare.com/ajax/libs/handlebars.js/1.3.0/handlebars.js"></script>
   <script id="cars-template" type="text/x-handlebars-template"> <!-- Ne pas oublier le tag
script -->
       {{#each cars}}
       <div>{{this}}</div>
       {{/each}}
   </script>

</head>
<body>
<div id="cars"></div> <!-- Remarquez comment le HTML devient très très simple -->
</body>
<script>
   var source   = $("#cars-template").html();
   var template = Handlebars.compile(source);

   var data = {cars : ["Toyota Corolla", "Toyota Yaris"]};
   $("#cars").html(template(data));
</script>
</html>
```

Templating

Mustache.js est particulièrement intéressant car il ne nécessite pas l'utilisation de **jQuery**…

- Recherchez toujours le *framework* le plus simple qui répond à vos besoins.

Exemples: https://github.com/janl/mustache.js

{{mustache.js}}

MVC - VueJS

Pourquoi Vue ?

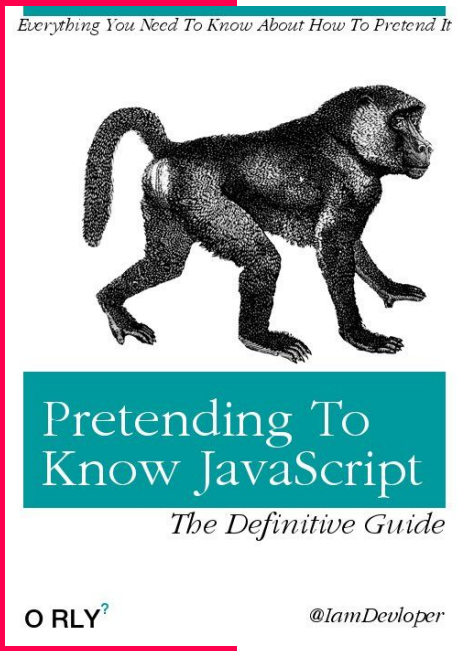
vs Angular, React, Ember ….

Vue.js s'avère une librairie plus qu'un framework, regroupant les outils pour permettre de faire des composants simples et réutilisables.

Les concepts amené dans Vue sont simples et universels. Apprendre Vue vous permettra d'appliquer des concepts semblables dans des frameworks ou librairie plus complexe comme Angular, React ou Ember.

Une autre option un peu moins moderne et moins *magique* mais très pédagogique restera toujours Backbone.js

> *Il est encore une fois important de noter que la compréhension du JavaScript pur est fondamental avant l'utilisation de framework.*



snipcart.com/blog/learn-vanilla-javascript-before-using-js-frameworks

Attention!
Nous utiliserons Vue version 2 dans le cadre du cours

Vue.js 2.0

Hello World

JS

```
new Vue({
    el: '#app',
    data: {
        message: 'Hello Vue.js!'
    }
})
```

HTML

```
<div id="app">
 <p>{{ message }}</p>
</div>
```

> *Although not strictly associated with the MVVM pattern, Vue's design was partly inspired by it. As a convention, we often use the variable vm (short for ViewModel) to refer to our Vue instance.*

vuejs.org/v2/guide/instance

Conditions

Les opérations dans les templates Vue sont toujours préfixés par `v-`

v-if

```
<div id="app-3">
   <p v-if="seen">Now you see me</p>
</div>
```

```
let app3 = new Vue({
   el: '#app-3',
   data: {
     seen: true
   }
})
```

https://jsfiddle.net/2u438meg/

Boucles

Les opérations dans les templates Vue sont toujours préfixés par v-

v-for

```
<div id="app-4">
   <ol>
       <li v-for="todo in todos">
       {{ todo.text }}
       </li>
   </ol>
</div>
```

```
let app4 = new Vue({
   el: '#app-4',
   data: {
       todos: [
           {text: 'Learn JavaScript'},
           {text: 'Learn Vue'},
           {text: 'Build something awesome'}
       ]
   }
})
```

https://jsfiddle.net/1k0rscah/

Évènements

Clicks, etc.

Les opérations dans les templates Vue sont toujours préfixés par v-

```
<div id="app-5">
   <p>{{ message }}</p>
   <button v-on:click="reverseMessage">Reverse Message</button>
</div>
```

```
let app5 = new Vue({
 el: '#app-5',
 data: {
   message: 'Hello Vue.js!'
 },
 methods: {
   reverseMessage: function () {
     this.message = this.message.split('').reverse().join('')
   }
 }
})
```

https://jsfiddle.net/62bcyxno/

Two-way binding

Entrée utilisateur

Les opérations dans les templates Vue sont toujours préfixés par v-

```
<div id="app-6">
 <p>{{ message }}</p>
 <input v-model="message">
</div>
```

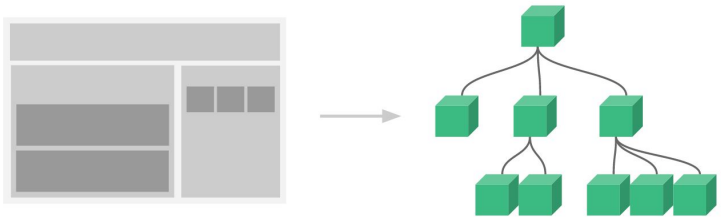
```
let app6 = new Vue({
 el: '#app-6',
 data: {
   message: 'Hello GLO-3102!'
 }
})
```

Imaginez ce simple composant en JavaScript pur! Vue nous simplifie grandement la vie.

https://jsfiddle.net/xn9uph1u/

Components

Pour développer une application complète de style **Single Page App**, on devra combiner plusieurs petites applications MVVM appelés Components.



vuejs.org/v2/guide/#Composing-with-Components

Components

```
<ol>
 <!-- Create an instance of the todo-item component -->
 <todo-item></todo-item>
</ol>
```

```
Vue.component('todo-item', {
 template: '<li>This is a todo</li>'
})
```

Évidemment, ceci donne un composant “statique”, donc le contenu sera toujours le même.

Components

Passer des paramètres aux components

```
Vue.component('todo-item', {
   props: ['todo'],
   template: '<li>{{ todo.text }}</li>'
 })

let app7 = new Vue({
   el: '#app-7',
   data: {
     groceryList: [
       { id: 0, text: 'Vegetables' },
       { id: 1, text: 'Cheese' },
       { id: 2, text: 'Whatever else humans are
supposed to eat' }
     ]
   }
})
```

```
<div id="app-7">
 <ol>
    <todo-item
       v-for="item in groceryList"
       v-bind:todo="item"
       v-bind:key="item.id">
   </todo-item>
  </ol>
</div>
```

https://jsfiddle.net/0h975o0g/

Components

Passer des paramètres aux components

- **data**
  - Propriété interne au component - non exposée à l'externe

- **props**
  - Propriétés exposées à l'externe du component - permet aux composants parent d'injecter des données

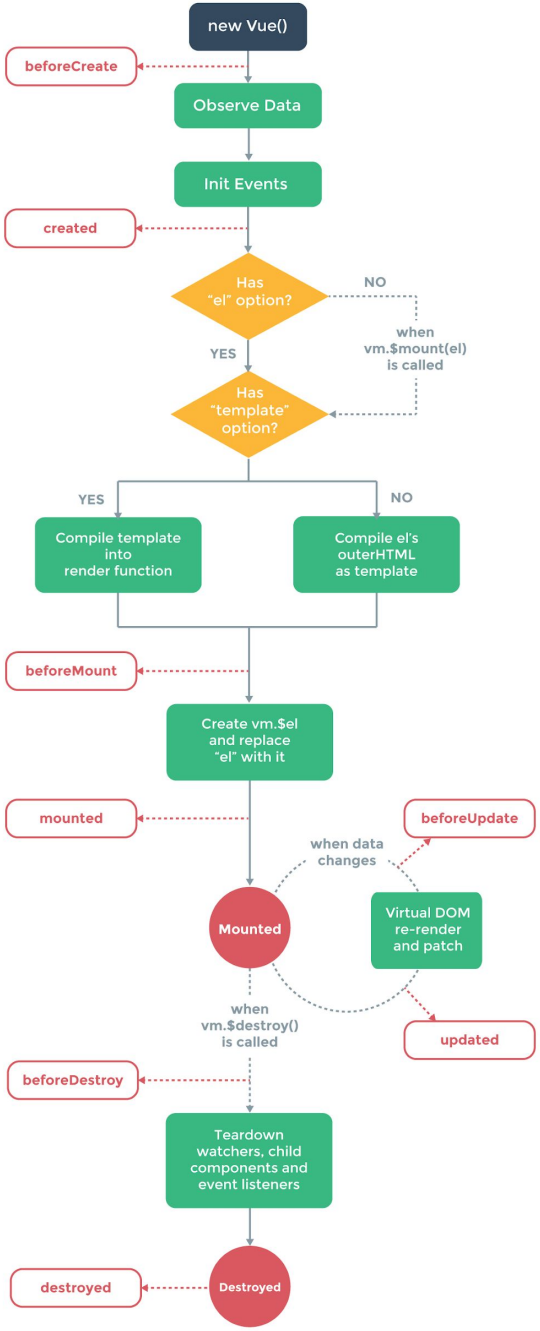
- **computed**
  - Propriétés internes composées à partir de d'autres propriétés

- **watch**
  - Fonctions internes qui réagissent à partir de changements au composant.

https://vuejs.org/v2/guide/computed.html#Computed-Properties

Cycle de vie d'un component

```
beforeCreate
created
beforeMount
mounted
beforeUpdate
updated
beforeDestroy
destroyed
```



Lifecycle

Exemple

```
var vm = new Vue({
   data: {
     a: 1
   },
   created: function () {
     // `this` points to the vm instance
     console.log('a is: ' + this.a)
   }
 })

 // -> "a is: 1"
```

Communication

Entre composants

Utilise la syntaxe $.emit. Ici un composant enfant (le composant **blog-post**):

```
<button v-on:click="$emit('enlarge-text', 0.1)">
  Enlarge text
</button>
```

Et son parent:

```
<blog-post
  ...
  v-on:enlarge-text="onEnlargeText"
></blog-post>
...

methods: {
  onEnlargeText: function (enlargeAmount) {
    this.postFontSize += enlargeAmount
  }
}
```

Routeur

Ajouter le support des URLs à votre application

Dans un contexte d'une single-page application, il n'y a qu'une seule *page* html, donc un seul URL pour accéder à l'application.

Il est fort probable que vous voudriez avoir des URLs uniques pour les différentes sections de votre application afin de permettre:

- Signets (bookmarks)
- Liens à partir d'autres pages/sites
- etc

Routeur

Ajouter le support des URLs à votre application

Il est possible de simuler des URL uniques avec un routeur. Nous allons utiliser le # (*hash*) de l'URL pour permettre de toujours charger le document index.html et interprété le hash avec le javascript pour afficher la page correspondante.

`window.location.hash` permet de lire la valeur du hash

`window.onhashchange` permet d'écouter sur les changement du hash

Il existe plusieurs librairies qui offrent ces fonctionnalités (et beaucoup plus).

Ex: vue-router, react-router, backbone-router, etc.

vue-router

Ajouter le support des URLs à votre application Vue JS

router.vuejs.org

vue-router

```
import Vue from 'vue';
import Router from 'vue-router';
import Home from '@/components/Home';
import Album from '@/components/Album';
import Artist from '@/components/Artist';

Vue.use(Router);

export default new Router({
 routes: [
   {
     path: '/',
     name: 'Home',
     component: Home,
   }, {
     path: '/artist',
     name: 'Artist',
     component: Artist
   }, {
     path: '/album',
     name: 'Album',
     component: Album
   }
 ],
});
```

vue-router

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


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

router.vuejs.org/en/essentials/dynamic-matching

Single File Component

Combiner CSS, HTML et JavaScript dans un fichier .vue

vuejs.org/v2/guide/single-file-component

Afficher des données du serveurs

Ajax avec “Promises”

```
<template>
 <ul v-if="posts && posts.length">
   <li v-for="post of posts">
     <p><strong>{{post.title}}</strong></p>
     <p>{{post.body}}</p>
   </li>
 </ul>
 <ul v-if="errors && errors.length">
   <li v-for="error of errors">
     {{error.message}}
   </li>
 </ul>
</template>

<script>
import axios from 'axios';

export default {
 data: () => ({
   posts: [],
   errors: []
 }),

 created() {
   axios.get(`http://jsonplaceholder.typicode.com/posts`)
   .then(response => {
     // JSON responses are automatically parsed.
     this.posts = response.data
   })
   .catch(e => {
     this.errors.push(e)
   })
 }
}
</script>
```

Afficher des données du serveurs

Ajax avec “async/await”

```
<template>
 <ul v-if="posts && posts.length">
   <li v-for="post of posts">
     <p><strong>{{post.title}}</strong></p>
     <p>{{post.body}}</p>
   </li>
 </ul>

 <ul v-if="errors && errors.length">
   <li v-for="error of errors">
     {{error.message}}
   </li>
 </ul>
</template>

<script>
import axios from 'axios';

export default {
 data: () => ({
   posts: [],
   errors: []
 }),

 async created () {
   try {
     const response = await axios.get(`http://jsonplaceholder.typicode.com/posts`)
     this.posts = response.data
   } catch (e) {
     this.errors.push(e)
   }
 }
}
</script>
```

Afficher des données du serveurs

Conseils

1. Remarquez que l'interaction avec le serveur est typiquement faite dans la méthode **created().**

   -> À cette étape, aucun DOM n'est encore créé, permet donc d'optimiser le rendering si la requête ne fonctionne pas, par exemple.

2. Assurez vous que vos appels serveurs sont fait dans des composants qui ne servent pratiquement qu'à ça.
   -> Divisez l'affichage des données!

Mixins

Permet de réutiliser des comportements

Voir https://vuejs.org/v2/guide/mixins.html

Permet de définir un comportement et de l'appliquer de manière générique au composant.

Existe aussi les **global mixins** -> permet de définir des comportements sur **tous** les composants de l'application.

Voir exemple complet ici:
https://css-tricks.com/using-mixins-vue-js/

VueX

State management

Un des problèmes classiques de Vue (ou tout autre framework de composants) est que plus la complexité de votre app augmente, plus vos composants deviennent **pollués!**

=> Les composants en haut de la chaîne multiplient les **props inutiles**!

Le pattern **flux** vient résoudre ce problème, mais il n'est vu que dans le cours 2. Ceci dit, sachez que VueX est relativement facile à intégrer dans votre application.

https://vuex.vuejs.org/

If your app is simple, you will most likely be fine without Vuex.

Exemples

- TodoMVC
  - todomvc.com/examples/vue
  - tastejs/todomvc/tree/gh-pages/examples/vue
- Hacker news clone
  - vuejs/vue-hackernews
- vuejs/awesome-vue

Tutoriel

- scotch.io/tutorials/build-a-to-do-app-with-vue-js-2

RTFM

vuejs.org/v2/guide

Excellents tutoriels vidéos complets sur laracasts

laracasts.com/series/learn-vue-2-step-by-step