Model - View - View Model

- Patron de conception architectural
- Le but est de séparer les responsabilités entre les données (Modèle) et la vue (View). Cela est possible grâce à un lien entre ces deux éléments (View Modèle)
- Développé par Ken Cooper et Ted Peters pour Microsoft en 2005, comme une variation du modèle MVC [1] pour Windows
- Il est depuis utilisé dans de nombreux domaines comme le développement Web et Android

1. https ://learn.microsoft.com/en-us/archive/blogs/johngossman/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps

Modèle

- contient les données,
- définit la logique métier pour créer, supprimer et modifier des éléments
- indépendant du reste de l'application, et donc de l'interface utilisateur

Exemple : API (REST), BD ...

Vue

Dans la vue on retrouve tous les éléments de l'interface utilisateur : affichage des données et gestion des interactions avec les utilisateurs

Exemple : page web

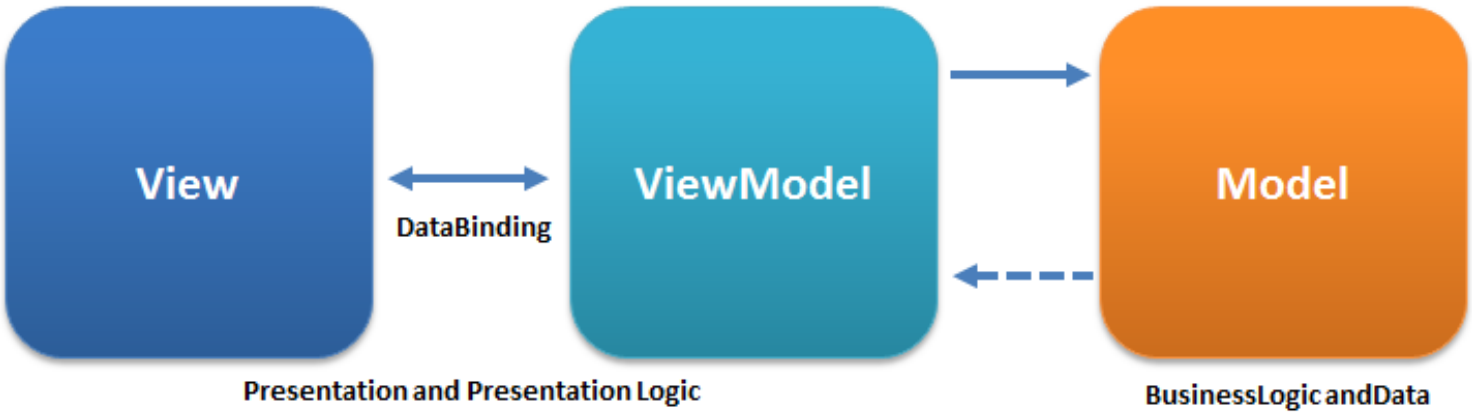
Comment faire le lien entre ces deux parties ?
C'est le rôle du *ViewModel*. Il se charge de faire le pont entre la couche *Vue* et la couche *Modèle*. Le *ViewModel* a deux objectifs

- Proposer à la Vue les données/propriétés à présenter. Il existe un binding automatique entre les propriétés de la vue et les propriétés de viewmodel
- Mettre à disposition de la Vue des fonctions permettant de modifier les données suites à des interactions avec l'interface

Comment faire le lien entre ces deux parties ?

ModelView

Le ModelView est donc une classe présentant un état des données du modèle, et des fonctions permettant de traiter les interactions de l'utilisateur avec la Vue.



- Ce design pattern n'est pas lié à un technologie, un type d'application ou à un langage
- Nous allons mettre en place ce design pattern dans le développement web à travers le framework **Vue.js**

Vue.js

- framework JavaScript open-source
- Construire des interfaces utilisateurs, dont notamment des applications single page
- Développé par Evan You en 2013, dernière version, 3.5 en novembre 2024
- Utilisé pour de nombreux sites dont Gitlab.com

Examinons un exemple de code

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <script type="importmap">
      {"imports": {
          "vue": "https://unpkg.com/vue@3/dist/vue.esm-browser.js"
      }}
    </script>
  </head>
<body>
<div id="app">{{ message }}</div>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }
    }
  }).mount('#app')
</script>
</body>
</html>
```

On peut distinguer trois parties dans cette page

- En-tête : la bibliothèque Vue est importée
- `div`, un élément est placé entre double { { } } (appelé moustache)
- Script : Pour ce premier exemple, le script est inclus directement dans la page. Mais il serait préférable par la suite de le stocker à part

Script

- Chaque application Vue débute avec la création d'une instance de l'application avec de la fonction `createApp`
- Cette fonction prend un composant, le composant racine. Dans notre exemple, nous allons définir directement le composant. Dans des projets plus importants, le composant sera défini dans un autre fichier, et pourra contenir lui même des composant.
- Ce composant contient des propriétés, définies dans `data` . La syntaxe est celle de JSON

On a ici une liaison (binding) entre la propriété définie dans le script, et l'élément défini dans la page. On introduit ici le concept de liaison entre la vue et la vuemodel.

Script

- Pour que l'instance de l'application soit fonctionnellle, il faut quelle soit montée
- la méthode `mount` est appelée une fois le composant créé, et prend en argument un élement du DOM

Il est tout à fait possible de définir une liste dans nos données et de l'utiliser dans notre vue

```
<div id="app">
    <h2>{{ title }}</h2>
    <ul>
      <li>{{ items[0] }}</li>
      <li>{{ items[1] }}</li>
     </ul>
</div>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    data() {
      return {
        items: ['Courses', 'Apprendre REST'],
        title: 'Ma liste de taches'
      }
    }
  }).mount('#app')
</script>
```

Méthodes

- En plus des données, il est possible de définir des méthodes. Pour cela il suffit de définir les méthodes dans la section `methods`
- Les méthodes peuvent manipuler les données

```
<div id="app" class="container">
  <p>{{ message }}</p>
  <span class="input-group-btn">
  <button v-on:click="renverse"
    class="btn btn-default"
    type="button">Renverse</button>
  </span>
</div>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    data() {
      return {
        message: 'Hello Vue.js!'
      }
    },
    methods: {
      renverse: function () {
        this.message = this.message.split('').reverse().join('')
      }
    }
  }).mount('#app')
</script>
```

- On peut voir que la fonction a accès aux attributs de notre objet, par l'intermédiaire de `this`
- Grâce à la directive `v:on:click` il est possible de relier la méthode à un évènement (ici le clique)
- La syntaxe abrégé de `v:on:evenement` est `@evenement`

```
<div id="app">
  <span v-bind:title="message">
    Survolez moi pour observer mon titre dynamiquement chargé !
  </span>
</div>
</body>
</html>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    data() {
      return {
        message: 'Cette page a été chargée le ' + new Date()
      }
    }
  }).mount('#app')
</script>
```

Pour les attributs, la syntaxe précédente (moustache) ne fonctionne pas, il est nécessaire d'utiliser des directives vue. Ici `v-bind`

- ~~`title={{message}}`~~
- `v-bind:title="message"` ou
- en version abrégé `:title="message"`
- liaison unidirectionnelle

```
<div id="app" class="container">
<input v-model="message"> {{ message }}
</div>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    data() {
      return {
        message: 'Hello Vue.js!'
      }
    }
    }
  }).mount('#app')
</script>
```

- Avec la directive `v-model` la liaison (binding) entre la vue et le viewmodel est bidirectionnel. Si l'un des deux change, l'autre est modifié automatiquement
- Utile sur les éléments d'input de la vue
- Peu de code est nécessaire

Autres directives :

- `v-if` : expression conditionnelle qui permet d'afficher ou non un élément
- `v-for` : boucle for

```
<div v-if="yesOrNo">Affichage ou non</div>

 <li v-for="element in liste-elements">
```

```
<div id="app" class="container">
<h2>{{ title }}</h2>
  <ol>
    <li v-for="todo in todos">
      <div class="checkbox">
        <label>
          <input type="checkbox">
          {{ todo.text }}
        </label>
        <div class="alert alert-success" v-if="todo.checked">
         Done
        </div>
      </div>
    </li>
  </ol>
</div>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    data() {
      return {
        todos: [
      { text: 'Apprendre ES6', checked: true },
      { text: 'Apprendre Vue', checked: false },
        ],
      title: 'Mes taches urgentes:'
      }
    }
  }).mount('#app')
</script>
```

- `template` : avec cet attribut, il est possible de définir un template. Vous spécifier dans cet attribut une chaine de caratère contenant de l'HTML avec du CSS

Un objet Vue a un plusieurs propriétés prédéfinies qui permettent d'appeler des fonctions sous certaines circonstances, c'est à dire des moments dans le cycle de vie du composant (lifecyle hooks).

- beforeCreate,
- beforeMount,
- mounted,
- beforeUpdate,
- updated,
- activated,
- beforeDestroy
- destroyed.

```
<body>
  <div id="app" class="container">
    <p>{{ seconds }}</p>
    <span class="input-group-btn">
    <button v-on:click="stoppe"
      class="btn btn-default"
      type="button">Stop</button>
    </span>
  </div>

  <script type="module">
    import { createApp } from 'vue'
    createApp({
      data() {
        return {
          seconds: 0
        }
      },
      methods: {
        stoppe: function() {
          clearInterval(this.$interval)
            }
      },
      mounted() {
        this.$interval = setInterval( () => {
          this.seconds++;
        }, 1000)
      },
      beforeUnmount(){
        clearInterval(this.$interval);
      }
    }).mount('#app')
  </script>
```

Il est possible de définir ses propres composants, avec son propre tag, avec la propriété `components`

```
<body>
<div id="app">
  <mon-composant></mon-composant>
</div>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    data() {
      return {
        message: 'Hello Vue!'
      }},
      components: {
        'mon-composant': {
          template: `<section> Hello Vue.js !</section>`
        }
      }
    }
  ).mount('#app')
</script>
</body>
```

- Il est possible de rendre notre composant parametrable, en ajoutant la propriété `props`. Dans cette proprité on définit les différents paramètres de notre nouveau tag
- Un paramètre peut être assigné avec une valeurs statique (une constante), ou dynamiquement avec un `v:bind`

```
<body>
<div id="app">
  <mon-composant :texte="texte"
                 classe="alert alert-info">
  </mon-composant>
</div>
<script type="module">
  import { createApp } from 'vue'
  createApp({
    components: {
      'mon-composant': {
        props: ['classe', 'texte'],
        template: '<section :class="classe"> {{ texte }}</section>',
      }
    },
    data(){
      return {
        texte: 'Salut Vue.js',
      }
    }
  }).mount('#app')
</script>
</body>
</html>
```