Vue.js tutorial

José C. Campos

Sistemas Interactivos
Mestrado Integrado em Engenharia Informática
DI/UMinho
May 12, 2020

Contents

1 Introduction

This tutorial builds on top of the previous ones, by resorting to the Vue.js[1] framework in order to implement the user interface layer for the Games Library application. The goal is that you use the result of the Bootstrap tutorial as your starting point. As a backup option, the source code for a basic project is provided with the current tutorial.

The Vue.js builds on the MVVM[2] architectural pattern and provides features for rich and expressive web user interfaces. One of its main advantages is the fact that it is a progressive framework. That is, it can be incrementally adopted. The core of Vue.js is focussed on the user interface layer only, but the framework is also able to support sophisticated (single-page) Web applications.

2 Setup

In its most basic usage form, Vue.js does not require any specific setup. You can start using the framework simply by including Vue.js of off the Web on your webpage with either:

```
1 <!-- development version, includes helpful console warnings -->
2 <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

or

```
1 <!-- production version, optimized for size and speed -->
2 <script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

This will ensure that the latest version will be used. Alternatively, you can use the version provided with this tutorial.

Vue.js can be installed locally[3] using `npm` and there is an official command line interface (Vue CLI)[4] that provides quick scaffolding for projects. However, to

[1] `https://vuejs.org/`, last visited 27/04/2019.

[2] Model–view–viewmodel (also known as model–view–binder) — see, for example, `https://en.wikipedia.org/wiki/Model-view-viewmodel`, last visited 27/04/2019.

[3] `https://vuejs.org/v2/guide/installation.html`, last visited 27/04/2019.

[4] `https://cli.vuejs.org`, last visited 27/04/2019.

start learning Vue.js, the simpler approach presented in the previous paragraph is the recommended one. Hence, we will use it for this tutorial.

It will be useful to install the Vue Devtools[5] extension in your browser. Chrome and Firefox are supported. There is also a standalone Electron app, but using the extensions is easier (see installation and usage instructions at `https://github.com/vuejs/vue-devtools#vue-devtools`[6]).

Tasks:

1. Include the development version of Vue.js in your list games page (`index.html` in the case of the code provided with the tutorial).

3 Creating the Vue instance

At the core of a Vue.js application is a Vue instance. Hence, every application starts by creating one, using the `Vue` function. This corresponds to the ViewModel in the MVVM pattern.

In our case we will want to have a name for our app, the logged in user name and a list of games to display. In Vue.js this is expressed as[7]:

```
var vm = new Vue({
    el: "#gamesApp",
    data: {
        appName: "Games App",
        userName: "",
        games: []
    }
  })
```

A Vue instance manages a given HTML element in the webpage (the *View*), and contains a data object. The element is declared in the `el` field of the object passed to the constructor fun.tion (in this case, the element with id `#gamesApp` is

[5] `https://github.com/vuejs/vue-devtools#vue-devtools`, last visited 27/04/2019.
[6] Last visited 27/04/2019.
[7] Calling the instance `vm` is a connection stemming from the fact that Vue.js build on the MVVM architectural pattern (see footnote 2).

managed). The data object has the `appName`, `userName` and `games` properties. The properties of the data object are added to Vue's reactivity system and can be used in the view. When the values of those properties change, the view will be updated to match the new values. Similarly, if the values are changed in the view, the properties are updated.

Tasks:

2. Create the `scripts` folder and the `gamesApp.js` file inside it. In the `gamesApp.js` file create the Vue instance.

3. To use the above Vue instance in your app, add the script to the `index.html` webpage, **after** Vue.js:

```
<script src="scripts/gamesApp.js"></script>
```

4. Finally, add the `gamesApp` identifier to the top level `div` of the page:

```
    <div class="container" id="gamesApp">
```

4 Declarative rendering

Now that the `div` has become a view controlled by the Vue instance, we can declaratively render the data using a template syntax[8]. In order to display the app name, we can replace the current *hardwired* name with `{{appName}}`. The HTML becomes:

```
<div class="container" id="gamesApp">
  <div class="row">
    <div class="col">
      <h1>{{appName}}</h1>
    </div>
  </div> <!-- row -->
  ...
```

[8] As opposed to imperative approaches such as, e.g., Java Swing or plain Javascript, where event handlers have to be written to handle updates to view and model.

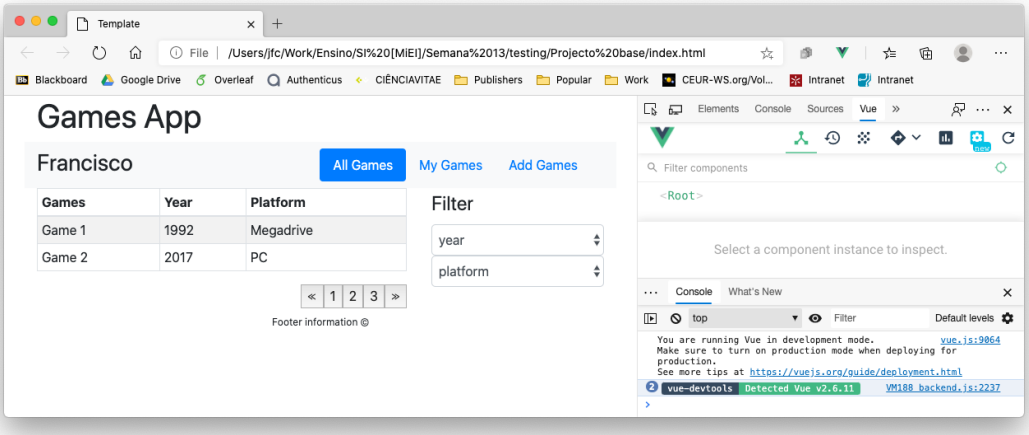

Figure 1: The App and user names defined in the Vue instance is displayed on the page (notice the Vue Devtools extension on the right).

Notice the reference to the `appName` property of the ViewModel inside the double curly braces ("Moustache" syntax). This is part of the template syntax[9].

Tasks:

5. Update your page to display both the app name and the user name defined in the Vue instance. The page should be similar to what is presented in Figure 1 (but, probably, with no user name displayed – see next point).

6. Since the `userName` property is empty, no user name is displayed. You can test the reactivity of the webpage by changing the property on the Vue Devtools extension. Simply write, e.g., "`vm.userName = "Francisco"`"[10] in the command prompt. The user name "Francisco" should now appear.

5 Loops and conditionals

We will now deal with presenting the list of games.

[9] See the full template syntax at: `https://vuejs.org/v2/guide/syntax.html`, last visited 28/04/2019.

[10] Notice how the properties defined in the data object become properties of the Vue instance.

Game class First we need to create the `Game` class to contain the game information. We define the class in `gamesApp.js` as follows:

```
class Game {
    // Private fields
    #id; #name; #year; #platform;

    //Constructor
    constructor(id, name, year, platform) {
        this.#id = id;
        this.#name = name;
        this.#year = year;
        this.#platform = platform;
    }

    //Getters
    get id() { return this.#id; }
    get name() { return this.#name; }
    get year() { return this.#year; }
    get platform() { return this.#platform; }
}
```

Tasks:

7. Create the `Game` class. Then, update the games list of the Vue instance with a few games of your choice[11].

5.1 Rendering arrays — the `v-for` directive

Currently, the list of games is statically defined in the HMTL file. At this point, we want to replace it by the list of games defined in the Vue instance.

The `v-for` directive supports iteration over a list of items (or an object's attributes). We can use it to display the list of games you previously defined in the Vue instance:

[11] You can create games using the defined constructor: `new Game(1, "Game 1", 1992, "Megadrive")`. The class will have to be declared before the View instance is!

```
<tbody>
  <tr v-for="g in games">
    <td>{{g.name}}</td><td>{{g.year}}</td><td>{{g.platform}}</td>
  </tr>
</tbody>
```

Tasks:

8. Define the list of games to be the contents of the `games` array of the Vue instance.

If you now reload the page, the new list of games will be displayed. In fact, every time the list of games changes, the page will be updated[12]. To test this, add a new game to the list in the Vue.js console:
`vm.games.push(new Game("amgus", "AmongUs", "2019", "Android")`.

We now want to setup the year and platform filters. The drop-down menus should list the years (respectively, platforms) present in the list of games.

Tasks:

9. Using the approach above, define the contents of the two dropdown menus (year and platform) in the filter.

5.2 Conditional rendering — the `v-if` and `v-show` directives

You might have noticed that the lists of years and platforms have duplicates (depending on the games you defined). Before we fix that, let us consider the pagination links.

The pagination links should only be displayed if needed (i.e. if the list of games does not fit one page). We can control whether the navigation `div` is displayed, or not, using the `v-if` directive:

[12] Due to limitations in JavaScript, there are two changes to the array that Vue.js is not able to detect. See: `https://vuejs.org/v2/guide/reactivity.html#Change-Detection-Caveats`, last visited 11/05/2020.

```
<nav class="d-flex justify-content-end" v-if="games.length>3">
  <form class="pagination">
    <button>&Lt;</a></li>
    <button>1</button>
    <button>2</button>
    <button>3</button>
    <button>&Gt;</button>
  </form>
</nav>
```

With the code above, the pagination will only be rendered if the list has more than three games. If we wanted to include some other information it is place, we could add an additional `div` with the v-else directive[13]:

```
<nav class="d-flex justify-content-end" v-if="games.length>2">
  ...
</nav>
<div class="d-flex justify-content-end" v-else>
  No need to sail away!
</div>
```

With `v-if` the content is not rendered (i.e. it is not placed in the DOM) if the condition is false. An alternative is to use `v-show`. The effect is similar but, in this case, the contents is always rendered and its visibility controlled with CSS.

As a rule of thumb, `v-show` should be preferred when you need to toggle visibility often (it only implies changing the style attribute), while `v-if` should be preferred when the condition is unlikely to change at runtime (the initial cost of rendering the element is avoided).

Tasks:

10. Using either `v-if` or `v-show`, control the visibility of the pagination links. Adjust the display threshold to be the number of games you defined.

[13] Note: there is also a `v-else-if` directive.

If you wish, you can now test the result by adding further games in the console, so that the links are displayed.

6 Methods, Watchers and Computed properties

As mentioned above you might have noticed that the years and platforms dropdown menus will have repeated entries, if multiples games have the same year or platform. In order to circumvent this, we need to calculate the list of years and the list of platforms, from the list of games, before we can display them. To achieve that we have three possibilities: using a *method*, using a *watcher* or using a *computed property*.

6.1 Methods

One possibility is to define a method that calculates the list. This can be defined in the Vue instance with the code below:

```
    methods: {
       listOfYears: function () {
           var list = [];
           this.games.forEach(function (g) {
               if (!(list.includes(g.year))) {
                   list.push(g.year);
               }
           });
           return list;
       }
   },
```

The function can now be used in the `v-for` directive to generate the dropdown with the list of years.

One issue with this approach, however, is that the computation of the list is not bound to the contents of the list of games. The result is that, one the one hand, the list is always computed when we want to generate the dropdown, even if the list of games has not changed; and, on the other hand, and more serious, when the list of games is updated the contents of the dropdown is not

updated unless explicitly rebuilt. While methods can be useful, clearly they are not the ideal approach in this case.

6.2 Watchers

An alternative approach is to use watchers (i.e. watched properties). They are useful when one property depends on another. To use this approach, we would define a new property (say, `years`) and define a *whatcher* on the `games` property to update `years`, when `games` changes. This is done as follows:

```
1    data: {
2        ...
3        years: []
4    },
5    watch: {
6        games: function(newList) {
7            this.years.length = 0;
8            for (g of newList) {
9                if (!(this.years.includes(g.year))) {
10                   this.years.push(g.year);
11               }
12           }
13       }
14   },
```

Now, every time `games` changes, the watcher function will run and update the `years` property.

An aspect to consider is that the function runs only when the watched attribute is updated. This means that it does not run when the Vue instance is created. Hence, the initial coherence between `games` and `years` is not guaranteed. Consider the following excerpt:

```
1    data: {
2        ...
3        games: [new Game(1, "Game 3", 2017, "Megadrive"),
4                new Game(1, "Game 1", 1992, "Megadrive"),
5                new Game(2, "Game 2", 2017, "PC")],
```

```
6         years: []
7     },
8     watch: {
9         games: function(newList) { ... }
10    },
```

With it, the page would present a list of games, but the years dropdown menu would be empty until some update to the list was made (you can try it by updating the list in the JavaScript console).

To solve the issue above we can define an event handler for the created event, which is triggered after the Vue instance is created, and use it to initialise the `games` attribute:

```
1     data: {
2         ...
3         games: [],
4         years: []
5     },
6     watch: {
7         games: function(newList) { ... }
8     },
9     created: function () {
10        this.games = [new Game(1, "Game 3", 2017, "Megadrive"),
11                      new Game(1, "Game 1", 1992, "Megadrive"),
12                      new Game(2, "Game 2", 2017, "PC")]
13    },
```

This way `years` is automatically updated, and the list of years is coherent with the list of games from the start.

While watchers are powerful, they can sometimes create repetitive code (e.g. when the derived attribute depends on more than one attributes – you need to declare multiple watchers[14]). Unless you need all the flexibility provided by watchers, it is preferable to use a simpler approach: computed properties.

[14]For a simple example that illustrates the issue, see `https://vuejs.org/v2/guide/computed.html#Computed-vs-Watched-Property` (last visited 28/04/2019).

6.3 Computed properties

With computed properties, instead of declaring `years` as a property of the Vue instance, you explicitly declare it as being computed at run time[15]. Look at lines 5 to 13 in the code below:

```
1    data: {
2        ...
3    },
4    computed: {
5        years: function () {
6            var list = [];
7            this.games.forEach(function (g) {
8                if (!(list.includes(g.year))) {
9                    list.push(g.year);
10               }
11           });
12           return list;
13       }
14   }
15 })
```

They define a new property on the Vue instance (`years`), which is computed from the list of games in `data`. Every time time the list of games changes, `years` is recalculated. However, in this case there is no actual attribute declared in `data` and, unlike in the case of the watcher functions, the function used to compute `years` has no side effects, making the code easier to understand and maintain.

Tasks:

11. Update your code to include the computed attribute `years` above and add a new computed attribute `platforms`.

12. Use these attributes to populate the dropdown menus of the filter. Keep the `year` and `platform` entries as the first entry in each menu to represent "*no selection*".

[15] cf. derived attributes in, e.g., UML.

7 Forms

At this point, we are able to display the data in the ViewModel. This was achieved by binding elements in the HTML page to attributes in the Vue instance. However, selecting an year or a platform in the filter still has no effect. To make the filter work we need to be able to determine which values are selected in the dropdown menus. For that we need to bind in the other direction: bind attributes of the Vue instance to elements in the HTML page.

In order to create bidirectional data bindings between form input fields (input, textarea, and select elements) and properties in the Vue instance, we use the `v-model` directive[16]. Hence, if we define a Vue instance attribute `selYear`[17] (selected year), we can write in the HTML file:

```
1         <form>
2           <fieldset>
3             <legend>Filter</legend>
4             <select class="form-control" v-model.number="selYear">
5               <option>year</option>
6               <option v-for="y in years">{{y}}</option>
7             </select>
8             ...
9           </fieldset>
10        </form>
```

The `.number` modifier makes Vue.js automatically typecast the value (a string) to a number. If the value in the string is not a number, its original value will be preserved. Hence, "year" will be a String, but all years' values will be numbers.

Tasks:

13. Use what you have learned until now to implement the filter functionality (it might be useful to have a computed attribute `gamesFiltered`).

[16] For more on form input binding see `https://vuejs.org/v2/guide/forms.html`, last visited 29/04/2019.

[17] Note that the initial value of the bound property should make sense for the input element. In this case we should define it as "year", making it the default value of the dropdown menu.

8 Fetching data asynchronously

Since version 2.0 Vue.js does not directly provide a means to perform asynchronous requests. Instead, it lets us choose the approach we want to use. There are essentially two alternatives: using a module (e.g. `vue-resource` or Axios) or using the native browser support[18]. We will use the latter, as it is the simpler solution.

Modern browsers support asynchronous HTTP request through the `fetch()` function. The Fetch API uses Promises, providing a simpler API when compared to the callbacks-based API of the `XMLHttpRequest` object it replaces.

A Promise is an object that represents the eventual conclusion of failure of an asynchronous operation[19]. The *traditional* approach to using Promises is to *register* functions to process the outcome of the Promise:

```
function f () {
    fetch("http://...")
    .then(function (response) {
        // do something with response
    })
    .catch(function(error) {
        // do something with error
    });
}
```

`then` is used in case of successful conclusion, `catch` in case of failure. Notice that the `response` object above is not a concrete value but a Stream where values will be produced.

While the above approach is simpler than using `XMLHttpRequest`, more recent browsers support an even simpler syntax: the async/await syntax. Using it the example becomes:

```
async function f () {
    try {
        response = await fetch("http://...");
```

[18] See `https://www.techiediaries.com/vuejs-ajax-http/` for a discussion (last visited 29/04/2019).

[19] We say the Promise *resolves* in case of success and *rejects* in case of error.

```
4         // do something with response
5     } catch (error) {
6         // do something with the error;
7     }
8 }
```

The async/await syntax is translated into promises. What the `await` keyword does is to wait for the promise returned by the function to resolve (or reject). Notice that `await` can only be used inside an `async` function. If in the code above we did not had the try/catch to handle errors, in case of error the promise generated by the call of the async function f() becomes rejected and we can catch it as in `f().catch(...)`.

Knowing that a Web service is provided at `http://ivy.di.uminho.pt:8080/GamesLibraryProvider`, using the async/await syntax, a basic Fetch request to retrieve the list of games consists of:

```
1 async function () {
2         const response = await fetch("http://ivy.di.uminho.pt:8080/
              GamesLibraryProvider/GamesService?action=list");
3         this.games = await response.json();
4     }
```

In line 2 we make a GET request to the relevant URL. Because `fetch` returns a Promise, we wait for it to be resolved, hence the `await`. When it finishes, the resolved value is placed in the `response` variable. In line 3, we obtain the JSON version of the response and store it in `this.games`. Because `response` is a Stream, we again need to wait for the processing to complete (or fail). In this case, we chose not to handle errors.

Tasks:

14. If you check the output of `http://ivy.di.uminho.pt:8080/GamesLibraryProvider/GamesService?action=list` you will notice two things: objects are being returned, but not Game objects; these objects have two additional fields (price and description). Update the Game class to include these additional fields.

15. Use the information from the Web service to populate the list of games in your application, replacing the statically defined games. Do this on the `created` property of the Vue instance, and make sure Game objects are created from the information received. In doing this, add a gamesURL attribute to the instance to hold the URL of the service. Add also a message attribute and use it on the web page to provide feedback about the success of loading the data.

9 Handling Events

We will now return to the pagination links and make pagination work. We start by building the infrastructure to support pagination.

Tasks:

16. Create a computed attribute `gamesPaginated` as an array of arrays of games. Define the maximum number of games in each sub-array (say 4) as an attribute of the Vue instance. Define also a `page` instance attribute to represent the page (sub-array) being displayed and set it to `0` (zero).
17. Change your HTML table to display the games in `gamesPaginated[page]` only.
18. Change also the generation of the pagination link so that only buttons from 1 to the size of the `gamesPaginated` array are created[20] (plus the Previous and Next buttons)

You should now only see the first 4 (or whatever number you defined) games of the list, the Previous and Next buttons, and the buttons corresponding to the available pages. The buttons, however, are still not functional, for that we will need to provide them with event handlers.

The `v-on` directive can be used to listen for and react to DOM events. We will use it to program what the buttons should do on click events. The "»" button should advance the page being displayed[21]:

[20] Note: you can write `v-for="(val, idx)in list"` to have access to the indexes of the array.
[21] `v-on:click` can also be written as `@click`

```
<button type="button" v-on:click="page += 1">&Gt;</button>
```

it is possible to use modifiers to listen for events such as mouse clicks when the Control ou Meta key is pressed[22].

Note that the definition above lets the page number go to (invalid) numbers above the number of pages. To prevent that, we need to disable it when the page number is the last one. This can be achieved by binding the disabled property of the button: `v-bind:disabled="page===gamesPaginated.length()"`.

Tasks:

19. Program the Previous and Next keys to navigate the pages one by one. Make sure that the buttons become disabled when appropriate. Add the functionality of Meta-click changing to the first/last page, respectively.

20. Now, program the numbered buttons to jump to their respective page.

10 Multiple components

One of the limitations with the current implementation is that there is no modularity. Typically we can decompose a user interface into components (in this case, the list of games, the filter, ...), yet we have a single ViewModel (the View instance) and a single View (the HTNL page). This means that all the code for the different components exists together. This hinders the maintainability and scalability of the implementation. In this section, the application will be decomposed into several components, starting with a component for a new feature: presenting the full information about a game.

A component is essentially a reusable instance with a name. We can define a simple component to display a game as:

```
Vue.component('game-show', {
    props: ['game'],
    template: `
        <div>
```

[22] See `https://vuejs.org/v2/guide/events.html#System-Modifier-Keys`, last visited 29/04/2019.

```
            <h3>{{game.name}}</h3>
            <p>
                Id: {{game.id}}<br />
                Year: {{game.year}}<br />
                Price: {{game.price}}<br />
                Description: {{game.description}}<br />
                Platform: {{game.platform}}
            </p>
            <button>Back</button>
        </div>
    `
  })
```

While components can have data, methods, etc. (they are instances), in this case two new concepts are introduced: `props` and `template`. The first, `props`, act as parameters to the component. We show how to bind them below. The second, `template`, is the HTML that will be generated when the component is used. The template must contain a single root element. That is the reason for the `div` tag. The use of ` to define the template allows us to write multi-line HMTL. Note, however, that this is not supported in IE. If you want to avoid this issue, use " and escape the newlines with \.

The main Vue instance can also have a template, in which case the template is displayed, not the HTML on the Web page[23].

We can now use this template in the HTML by writing (e.g., after the list of games):

```
        <game-show v-bind:game="selGame" v-if="selGame.id"></game-show>
```

Since we are binding the `game` property of the template to `selGame` (a property of the instance that holds the currently selected game – you will have to declare it!), the game in `selGame` will be displayed according to the template. Note, however, that, due to the `v-if` directive, this will only happen if variable selGame has an id attribute[24]. To set the selected game, we add an event handler to the table

[23] Try it!

[24] This is a quick way to ensure `selGame` holds a game.

rows (look at line 7):

```
1           <table class="table table-striped table-bordered table-sm">
2               <thead>
3                   <tr><th>Games</th><th class="extras">Year</th><th>
                        Platform</th></tr>
4               </thead>
5               <tbody>
6                   <tr v-for="g in gamesPaginated[page]"
7                       v-bind:key="g.id"
8                       v-on:click="selGame = g">
9                       <td>{{g.name}}</td><td>{{g.year}}</td><td>{{g.
                            platform}}</td>
10                  </tr>
11              </tbody>
12          </table>
```

Tasks:

21. Implement the `game-show` components and include it in the web page as shown above. Test it and notice how games can now be selected and displayed.

22. Update the solution so that if a game is pressed a second time the `game-show` component is hidden[25]

So far we have shown how to pass information into the template. In many cases we also need to send information from the component back to its parent. We can do that using events. As an example, we will want to notify the parent when the navigation buttons are pressed, so that the `show-game` component is hidden.

Consider the case of the "Back" button. We must modify the button to emit an event when clicked:

```
1           <button v-on:click="$emit('back')">Back</button>
```

[25]You can do it by checking that the game clicked is the same as the already selected game, and setting `selGame` to {} in that case.

The above will cause the button to emit a 'back' event. Then, we must handle the event in the `game-show` tag. It becomes:

```
        <game-show v-bind:game="selGame" v-if="selGame.id"
                   v-on:back="selGame={}"
        ></game-show>
```

Now, when the back event is caught, `selGame` becomes an empty object and, due to the `v-if` directive, the tag is removed from the page.

Tasks:

23. Refactor the filter component into a `games-filter` component. You will have to communicate back which year and platform have been selected, use the following skeleton solution[26]:

```
Vue.component('games-filter', {
    props: ['listOfGames', 'selYear', 'selPlat'],
    computed: {
        years: function () {...}, // moved here from...
        platforms: function () {...}, // ...the main instance
    data: function () {
        return {
            year: this.selYear, // local copy of prop
            ...
        }
    },
    template: `
        <form>
            ...
                    <select class="form-control"
                            v-model.number="year"
                            v-on:change="$emit('newyear', year)">
                        ...
```

[26]We define a `year` property to use in the dropdown menu, since we cannot change the property that is passed as a parameter. Note that the properties in `data` must be defined inside a function. This is so that multiple instances of the component might exist, each with its properties. Additionally, we use the `change` event of the `select` tag to emit and event with the selected value.

```
                    </select>
                    ...
        </form>
    `
})
```

The component is used thus:

```
<games-filter v-bind:list-of-games="games"
              v-bind:sel-year="selYear"
              v-on:newyear="selYear=$event"
              ...>
</games-filter>
```

24. Refactor the table and pagination into a `games-list` component.
25. Now handle visibility so that, when `game-show` is rendered, `games-list` and `games-filter` are not, and vice-versa.

11 Next steps

This tutorial covers only the essential aspects of Vue.js. It is meant at a starting point to allow you to develop a small application. Aspects such as code organisation, single-file components or routing are not covered. These require the installation of additional tools or modules, something that we wanted to avoid on this first contact with the framework.

One advantage of Vue.js is that you can start simple, and learn more advanced concepts as you need them/master the more basic ones. See the guide at `https://vuejs.org/v2/guide/` for more information and to continue exploring Vue.js.