Vue.js tutorial

José C. Campos

DI/UMinho
April 9, 2024

Contents

1 Introduction

This tutorial builds on top of the previous one, 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 previous tutorial as your starting point. As a backup option, the source code for a basic project is provided with the current tutorial.

The tutorial starts with some basic notions of Javascript, and then introduces the Vue.js framework. Vue.js builds on the MVVM[2] (Model–View–ViewModel) architectural pattern (see Figure 1) 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 focused on the user interface layer only, but the framework is also able to support sophisticated (single-page) Web applications.

Figure 1: The Model-View-ViewModel (MVVM) architectural pattern

2 Javascript basics

Contrary to HTML/CSS, Javascript is an imperative programming language. Javascript scripts correspond to code and functions, which are invoked in response to events in the page (e.g. when the page is loaded or when a user interacts with some control).

Javascript's syntax[3] is inspired by languages such as Java and C. Blocks of code are defined by "`{ }`", and "`;`" is used to end statements[4]. Javascript features the usual control structure: `if`, `while`, `do`/`while`, `for` and `switch`. To particular

[1] `https://vuejs.org/`, last visited 08/04/2023.

[2] Also known as Model–View–Binder — see, for example, `https://en.wikipedia.org/wiki/Model-view-viewmodel`, last visited 08/04/2024.

[3] For more on Javascript syntax see: `https://www.w3schools.com/js/js_syntax.asp`, last visited 26/04/2022.

[4] Strictly speaking semicolons are not mandatory in Javascript. However, its use is recom-

forms of for loops are `for/in` and `for/in`. The `for/in` statement loops through the enumerable property keys of an object. The `for/of` statement loops through the values of an iterable object. Examples of iterable objects are arrays and strings. Hence, the following two cycles would output `0, 1, 2` and `bananas, apples, pears`, respectively:

```
1 const fruits = ["bananas", "apples", "pears"];
2 for (f in fruits) console.log(f);
3 for (f of fruits) console.log(f);
```

Javascript is dynamically typed. What this means is that the types of variables are generally not known at compile time. See Section 2.2 for more on variables.

2.1 Running Javascript code

In order to check that everything is working as expected, some minimal examples can be put to work. The first proposed test is to create a popup message, using the `alert()` function (c.f. Figure 2), and the second is to print to the console, using `console.log()` (c.f. Figure 3).

You can do the above by placing the Javascript code inside a `<script>` tag on the HTML file. A better alternative is to place the code in external "`.js`" files (e.g. `script.js`), and load those into the page. To load `script.js`, the declaration `<script src="script.js"></script>` can be used.

The code placed in the `script.js` file will load once the browser loads the page referencing it. Code inside a function is only executed when the function is called. Code outside function definitions is automatically run. Thus, if you call the `alert()` method at the beginning of the `script.js` file, a popup window will automatically appear when the file is loaded. See the following `script.js` example file:

```
1 //code here will run automatically
2 alert("This is an alert message!");
```

mended. If not present, they are automatically inserted at the end of the line, which can cause problems (e.g. if an expression spans multiple lines, semicolons might be placed in the wrong place).

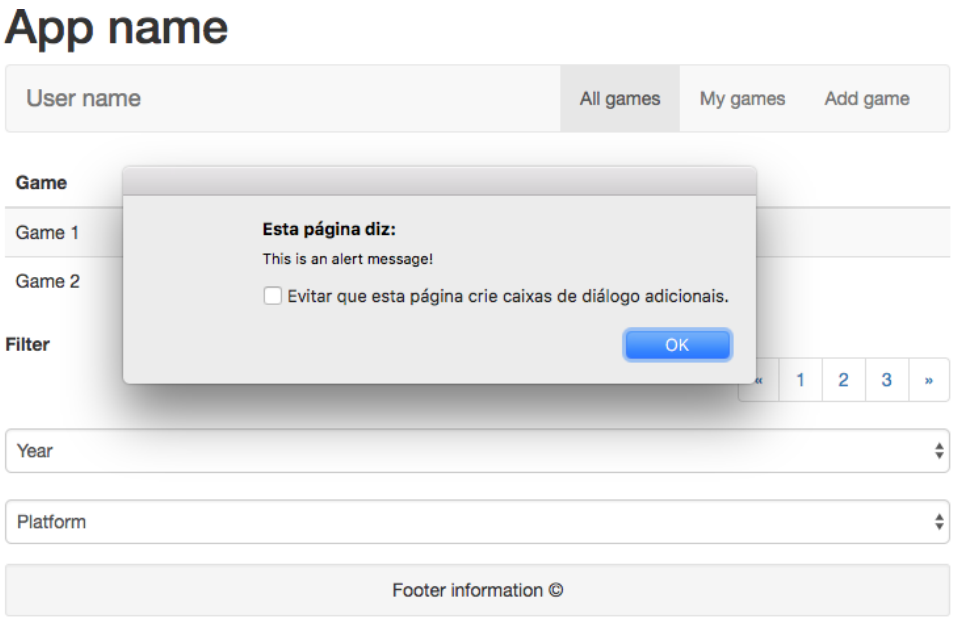

Figure 2: Output via an alert message.

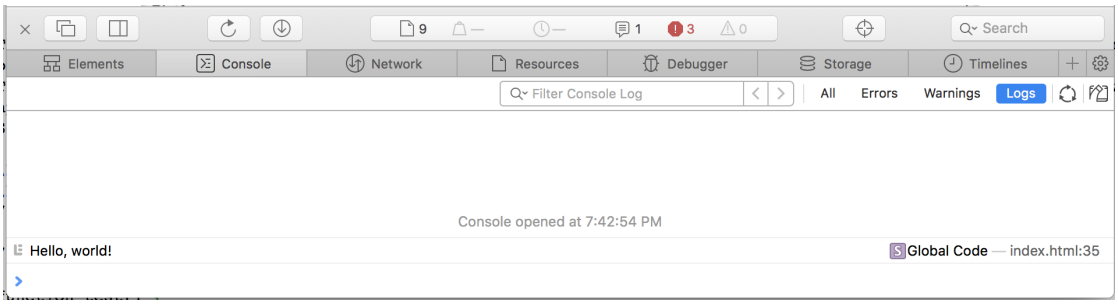

Figure 3: Output via the console.

```
3
4 function test() {
5     //this code will run when the function is invoked
6     console.log("Hello world!");
7 }
```

Modern browsers have developer tools to help in the development process. For instance in Google Chrome, Mozilla Firefox or Safari simply right-click in a page, and select inspect element. Such will show a new window or frame, where the console is accessible. This console will show errors, output messages and

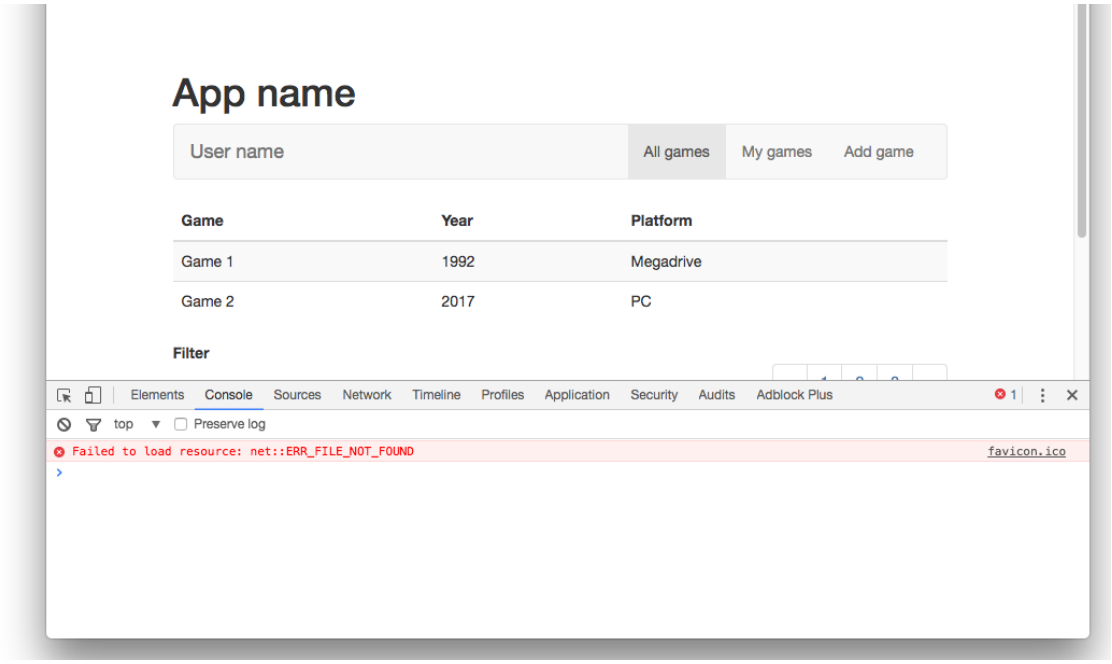

Figure 4: Javascript console with an error message.

allow you to interact with the browser (using Javascript). Figure 3 shows the Safari console displaying log messages. Figure 4 shows the Google Chrome console, with an error message indicating a missing file.

Tasks

1. Create the `scripts` folder and a `script.js` file inside it. Load the file at the end of the `index.html` file[5].

2. In the `script.js` file, create a popup message with "Hello, world!" as content. This popup should appear once the page is loaded. Create a function `sayHello()` which prints to the terminal "Hello, world!".

3. Load the `index.html` file and observe the alert message appear. Now invoke the `sayHello()` function through the console and check the output produced.

[5]For large applications, this allow the page to load faster, without having to wait for the Javascript code.

2.2 Javascript objects and classes

Javascript has the concept of object. Objects can be defined dynamically. For example, the following code defines an (empty) object and adds two fields (name and year) to it:

```
let game = {};
game.name = "Oxenfree";
game.year = 2017;
```

Javascript also supports the notion of classes and, as of ECMAScript 2015, a class can be defined as follows:

```
class Game {
    // constructor
    constructor(aName, aYear) {
        this._name = aName;
        this._year = aYear;
    }

    // getter for _name
    get name() {
        return this._name;
    }

    // setter for _name
    set name(v) {
        this._name = v;
    }

    // example method
    hasYear(aYear) {
         return this._year===aYear;
    }
}
```

Note that object properties (instance variables in "Java *speak*") do not have to be explicitly declared as part of the class definition (more below on this). They

are dynamically created when assigned some value in the constructor or methods (you should follow the convention to assign – i.e. *create* – all properties in the constructor). The keyword `this` is used to reference the object. As all other Javascript variables, instance attributes are dynamically typed (i.e., they have the type of the value they hold at each moment).

Notice the convention used to indicate private variables (the use of underscore), and the definition of getters and setters to access them. Using underscores was **just a convention** that was defined to support some form of encapsulation, since Javascript had no means of declaring a field or method private. However, encapsulation is not really guaranteed: while writing "`game.name`" uses the getter method, writing "`game._name`" accesses the field directly and will not trigger any error.

As of ECMAScript 2020, private fields and methods can be declared using the # prefix. The example above becomes:

```
class Game {
    // private fields
    #name; #year;

    // constructor
    constructor(aName, aYear) {
        this.#name = aName;
        this.#year = aYear;
    }
    ...
}
```

Attempting to access `game.#name` will now generate an error.

Using a class in Javascript is similar to Java:

```
let game = new Game("Oxenfree", 2016);
console.log(game.name); // uses the get method
console.log(game.hasYear(2017));
```

Note the `let` keyword to declare the `game` variable. Variables are implicitly *defined* the first type they are assigned. However, this is considered bad practice.

The declaration of variables (using `var`, `let` or `const`) is optional, but **highly recommended**. Declaring a variable has effects on the scope of the variable, and on whether it can later be declared again in the same scope[6]. Variables that are not explicitly declared, automatically gain a global scope (i.e. the variables become global variables)[7].

Tasks

4. Complete the class `Game` with the platform property, and add a method to check if a game is more recent than another (i.e. `game1.moreRecent(game2)`). Write your code in `scripts/game.js`.

2.3 Manipulating the DOM

Javascript has the capability of manipulating the DOM of the Web pages. This makes it possible to access the page elements, and access or modify their contents, appearance and even behaviour. When writing code for a Web page, we are in an event-based programming model. The browser controls execution and we provide event handlers to be executed when relevant events occur. We can think of event handlers in terms of two general types:

- even handlers used to react to browser events (e.g., a page has finished loading);
- even handlers used to react to user actions on the page (e.g., a button has been clicked).

In Section 3 we will look at the Vue.js framework, which facilitates defining and manipulating the contents of a web page. Nevertheless, it is worth noting that, in pure Javascript, manipulating the DOM consists of three steps:

First define which event[8] will trigger the behaviour. In order to achieve this, it is possible to specify, in the HTML elements, which Javascript functions are

[6] With `let` and `const` it cannot and will raise an error; with `var`, declaring a variable multiple times is allowed.

[7] For an explanation of Javascript scopes see: `https://www.w3schools.com/js/js_scope.asp`. Last visited 08/04/2024.

[8] List of possible events `https://www.w3schools.com/jsref/dom_obj_event.asp`. Last visited 08/04/2024.

associated with which events. E.g., to invoke a function when a mouse hovers some `<div>` we can write:

```
<div onmouseover="myFunction()">over here</div>
```

Second in the function, it is necessary to retrive from the DOM the element(s) of the page that will be accessed. This requires using DOM functions such as `document.getElementById()`[9]:

```
var elem = document.getElementById("myDiv");
```

Third the properties and functions of the retrieved element can then be used to achieve the desired effect[10]. For instance, modifying style properties can be done through the `style` property[11]:

```
elem.style.color = "blue";
```

Through this series of steps, it is possible to manipulate the page. However, as mentioned earlier, in Section 3 we will look at an alternative (more declarative) approach.

Tasks

An example is proposed, in order to understand the principles of DOM manipulation. The example is a counter of the number of clicks in a button.

5. In the HTML file, create the button to be clicked and a `<div>` where the number of clicks will be displayed. Give the `<div>` some `id` (e.g. `myId`).

6. In the `script.js` file:

   (a) Define a variable to act as counter and initialise it to zero[12].

[9] See the page `https://www.w3schools.com/jsref/dom_obj_document.asp` for more information on the attributes and API of the `document` object.

[10] See the page `https://www.w3schools.com/jsref/dom_obj_all.asp` for more information on DOM nodes' properties and API.

[11] See the page `https://www.w3schools.com/js/js_htmldom_html.asp` for more information on how to modify HTML.

[12] It is possible to avoid using a variable by working directly with the contents of the `<div>` (you can try it), however, this would be less MVC-like.

(b) Create a function which sets the contents of the just created `<div>` to the value of the counter (you can achieve this by setting the `innerHTML` property of the `<div>`) and increments it.

(c) Update the contents of the `<div>` to 0 (by invoking the function).

7. Back in the HTML, associate the `onClick` event of the button with your function.

   (a) do it in the HTML file, as explained above;

   (b) explore how to do it in the Javascript file instead.

8. Load the page in a browser and click the button multiple times, observing the effect on the `<div>`.

2.4 JSON

JSON (JavaScript Object Notation) is a popular format to exchange data in REST formats, and can be easily interpreted and manipulated by Javascript. JSON is a textual format which corresponds to key-value pairs. An example is as follows:

```
{
    "user": "username",
    "age": 10,
    "ids": ["1", "2", "3"],
    "contact": {
        "address": "street ...",
        "number": "n"
    }
}
```

The JSON[13] is derived from Javascript so JSON objects are defined inside `{ }`. Data is written as pairs of name/value using the syntax `"name":"value"`. Names must be strings). Value can be strings, integers, arrays (c.f. `ids` above) or even another object (e.g. `contact` above).

[13] See the following URL for more details `http://www.json.org/`. Last visited 08/04/2024.

In order to validate the syntax of a JSON object some tools can be used. The website `http://jsonlint.com/` is one example.

Converting a JSON string to a Javascript object is straightforward. The JSON class (part of Javascript) provides the method `parse`[14]. This method parses the text and creates an object with the corresponding structure. Next follows an example on how to parse and use JSON.

```
var json = "{\"name\":\"a user\",\"age\":\"20\"}";
var u = JSON.parse(json);
console.log(u.name);
```

Note that the object `u` is dynamically created, and its attributes automatically set. The inverse process can be achieved with the method `JSON.stringify()` .

3 Vue.js basics

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:

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

or

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

This will ensure that the latest version will be used[15].

[14] See the following URL for more details `https://www.w3schools.com/js/js_json_parse.asp`. Last visited 08/04/2024.

[15] This will load the latest stable version of Vue.js. At the time of initial writing, this was v2.6.12. The current version is v.3.4.21 – this tutorial should work with this version.

Vue.js can be installed locally using `npm` and there is an official build tool (Vite[16]) that provides quick scaffolding for projects. However, to start learning Vue.js, the simpler approach presented in the previous paragraph is recommended. Hence, we will use it for this tutorial.

It will be useful to install the Vue Devtools[17] extension in your browser. Chrome and Firefox are supported. There is also a standalone Electron app, but using the extensions is easier.

Tasks:

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

3.1 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 (we will use a property named `appName` to hold that name), the name of the logged in user (`userName`), and a list of games to display (`games`). In Vue.js this is expressed as[18]:

```
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 HTML element to be controlled is declared in

[16] `https://vitejs.dev/`, last visited 08/05/2024.

[17] `https://github.com/vuejs/devtools#vue-devtools`, last visited 08/05/2024.

[18] Calling the instance `vm` is a convention stemming from the fact that Vue.js builds on the MVVM architectural pattern (see footnote 2).

the `el` field of the object passed to the constructor function (in this case, the element with id `#gamesApp` is 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 in the data object are updated.

Tasks:

10. Create the `scripts` folder, if it does not yet exist, and the `gamesApp.js` file inside it. In the `gamesApp.js` file create the Vue instance.

11. 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>
```

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

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

3.2 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[19]. 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 -->
      ...
```

[19] 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.

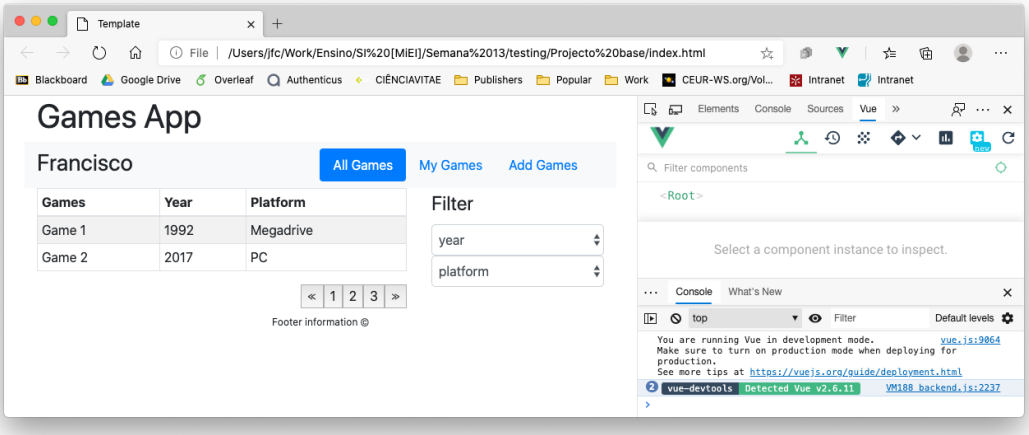

Figure 5: The App and user names defined in the Vue instance are displayed on the page (notice the Vue Devtools extension on the right).

Notice the reference to the `appName` property of the ViewModel inside the double curly braces ("Moustache" syntax). This is part of the template syntax[20].

Tasks:

13. 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 5 (but, probably, with no user name displayed – see next point).

14. 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"`"[21] in the command prompt. The user name "Francisco" should now appear.

3.3 Loops and conditionals

We will now deal with presenting the list of games.

[20] See the full template syntax at: `https://vuejs.org/v2/guide/syntax.html`, last visited 08/04/2024.

[21] 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:

15. Create the `Game` class. Then, update the games list of the Vue instance with a few games of your choice[22].

3.3.1 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:

[22] 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:

16. 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.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:

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

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.

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

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:

```
<nav aria-label="Page navigation" v-if="games.length>3">
  <ul class="pagination justify-content-end">
    ...
  </ul>
```

```
  </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 in its place, we could add an additional `div` with the v-else directive[23]:

```
<nav aria-label="Page navigation" v-if="games.length>3">
  ...
</nav>
<div class="d-flex justify-content-end" v-else>
  These are all the games!
</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:

18. 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.

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

4 Methods, Watchers and Computed properties

As mentioned above, you might have noticed that the years and platforms drop-down 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

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

achieve that we have three possibilities: using a *method*, using a *watcher* or using a *computed property*. We will now discuss each one of them.

4.1 Methods

One possibility is to define a method that calculates the list. This can be defined in the methods section of the Vue instance (see lines 8-18 below):

```
1  var vm = new Vue({
2      el: "#gamesApp",
3      data: {
4          appName: "Games App",
5          userName: "",
6          games: [...]
7      },
8      methods: {
9          listOfYears: function () {
10             var list = [];
11             this.games.forEach(function (g) {
12                 if (!(list.includes(g.year))) {
13                     list.push(g.year);
14                 }
15             });
16             return list;
17         }
18     }
19 })
```

The `listOfYears` 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, on the one hand, the list is always computed every time we want to (re)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.

4.2 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:

```
data: {
    ...
    years: []
},
watch: {
    games: function(newList) {
        this.years.length = 0; // clears the list
        for (g of newList) {
            if (!(this.years.includes(g.year))) {
                this.years.push(g.year);
            }
        }
    }
},
```

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:

```
data: {
    ...
    games: [new Game(1, "Asphalt 9: Legends", 2018, "Switch"),
            new Game(2, "AmongUs", 2019, "Tablet"),
            new Game(3, "Apex Legends", 2019, "PC")],
    years: []
```

```
    },
    watch: {
        games: function(newList) { ... }
    },
```

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 a function for the `created` lifecycle hook[24], which is called after the Vue instance is created, and use it to initialise the `games` attribute:

```
    data: {
        ...
        games: [],
        years: []
    },
    watch: {
        games: function(newList) { ... }
    },
    created: function () {
        this.games = [new Game(1, "Asphalt 9: Legends", 2018, "Switch"),
                      new Game(2, "AmongUs", 2019, "Tablet"),
                      new Game(3, "Apex Legends", 2019, "PC")]
    },
```

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[25]). Unless you need all the flexibility pro-

[24] For more on lifecycle hooks and the Vue instance lifecycle see: `https://vuejs.org/v2/guide/instance.html#Instance-Lifecycle-Hooks`, last visited 08/04/2024.

[25] For a simple example that illustrates the issue, see `https://vuejs.org/v2/guide/computed.html#Computed-vs-Watched-Property` (last visited 08/04/2024).

vided by watchers, it is preferable to use a simpler approach: computed properties.

4.3 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[26]. 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:

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

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

20. 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*".

5 Bidirectional data bindings

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, so that the HTML page shows the data present in the Vue instance. However, selecting a year or a platform in the filter still has no effect on which games are displayed. To make the filter work we need to make the Vue instance aware of which values are selected in the dropdown menus of the HTML page. 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[27]. Hence, if we define a Vue instance property `selYear`[28] (selected year), we can write in the HTML file:

```
1          <form>
2            <fieldset class="form-group">
3              <legend>Filter</legend>
4              <select class="form-select" 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 pre-

[27] For more on form input binding see `https://vuejs.org/v2/guide/forms.html`, last visited 08/04/2024.

[28] 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.

served. Hence, "year" will be a String, but all years' values will be numbers.

Tasks:

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

6 Fetching data asynchronously

Vue.js does not directly provide a means to perform HTTP 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[29]. We will use the latter, as it is the simpler solution.

Modern browsers support asynchronous HTTP requests 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 replaced.

A Promise is an object that represents the eventual conclusion of failure of an asynchronous operation[30]. The *traditional* approach to using Promises is to *register* functions to process the outcome of the Promise:

```
1 function f () {
2     fetch("http://...")
3     .then(function (response) {
4         // do something with response
5     })
6     .catch(function(error) {
7         // do something with error
8     });
9 }
```

`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.

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

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

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:

```
1 async function f () {
2     try {
3         const response = await fetch("http://...");
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() would become rejected and we would be able to catch it as in `f().catch(...)`.

A basic Fetch request to retrieve the list of games from a Web service provided at `https://my-json-server.typicode.com/joseccampos/gamesLibrary`[31] consists of (using the async/await syntax):

```
1 async function f () {
2     const response = await fetch("https://my-json-server.typicode.com/
          joseccampos/gamesLibrary/games");
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,

[31] The above service is based on `json-server` (`https://github.com/typicode/json-server`, last visited 20/04/2021). You can install and run `json-server` locally if you wish. Use the `db.json` file at `https://my-json-server.typicode.com/joseccampos/gamesLibrary/db`.

we again need to wait for the processing to complete (or fail). In this case, we chose not to handle errors.

Tasks:

22. 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 (see page 19), and make sure Game objects are created from the information received.

    In doing this, add a gamesURL property to the instance, to hold the URL of the service. Add also a message property, and use it on the web page to provide feedback about the success of loading the data.

7 Handling Events

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

Tasks:

23. 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).

24. Change your HTML table to display the games in `gamesPaginated[page]` only.

25. Change also the generation of the pagination link so that only buttons from 1 to the size of the `gamesPaginated` array are created[32] (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*[33], however, are still not functional, for that we will need to provide them with event handlers.

[32] Note: you can write `v-for="(val, idx)in list"` to have access to the indexes of the array.
[33] In fact they are not `button` tags, but links in the `pagination` Bootstrap component.

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[34]:

```
1  <a class="page-link" v-on:click="page = page+1">&raquo;</a>
```

It is possible to use modifiers to listen for events such as mouse clicks when the Control ou Meta key is pressed[35].

Note that the definition above lets the page number go to (invalid) numbers above the number of pages. To prevent that, you need to check that the current page is not the last one. The same applied when try to go back on the first page.

Tasks:

26. Program the Previous and Next keys to navigate the pages one by one. Make sure that the buttons do not allow going forward/backwards to non-existing pages. Add the functionality of Meta-click changing to the first/last page, respectively.

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

8 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 HTML 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:

[34] `v-on:click` can also be written as `@click`.

[35] See `https://vuejs.org/v2/guide/events.html#System-Modifier-Keys`, last visited 19/04/2021.

```
Vue.component('game-show', {
    props: ['game'],
    template: `
        <div>
            <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. In Vue 2.*x*, the template must contain a single root element. That is the reason for the `div` tag. The use of back quotes (`) to define the template allows us to write multi-line HMTL[36].

The main Vue instance can also have a template, in which case the template is displayed, not the HTML on the Web page[37].

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

[36]Note, however, that this is not supported in IE. If you want to avoid this issue, use " and escape the newlines with \.

[37]Try it!

the instance, which 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[38]. To set the selected game, we add an event handler to the table rows (look at line 8):

```
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:

28. 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.

29. Update the solution so that if a game is pressed a second time the `game-show` component is hidden[39]

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 back button of the `game-show` component is pressed, so that the `show`

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

[39] 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.

-game component is hidden, and the list of games and the filters are displayed again.

We must modify the button to emit an event when clicked:

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

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

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

Now, when the back event is caught, selGame becomes an empty object and, due to the v-if directives, the game-show component is hidden.

Tasks:

30. 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[40]:

```
1  Vue.component('games-filter', {
2      props: ['listOfGames', 'selYear', 'selPlat'],
3      computed: {
4          years: function () {...}, // moved here from...
5          platforms: function () {...}, // ...the main instance
6      data: function () {
7          return {
8              year: this.selYear, // local copy of prop
9              ...
10         }
11     },
12     template: `
```

[40]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.

```
        <form>
            ...
                    <select class="form-control"
                            v-model.number="year"
                            v-on:change="$emit('newyear', year)">
                        ...
                    </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>
```

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

9 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/guide/` for more information and to continue exploring Vue.js.