Unit 5 - Vue.js

Classwork: The classwork for this unit should be saved in a new folder: **your_repo/unit_5/**
Homework: Homework should be saved here: **your_repo/homework/unit_5/**

*** Day 01 ***

Vue.js - Intro

We will take a break from Django, so we can learn Vue.js, but we will use Django later to serve the requests done by Vue.js and have a full stack application.

I saw the majority of students did not implement a good amount of JS components in your final project for the first semester. I understand you, maybe JS is not easy. Or maybe you did not have enough time.

I have good news about the frontend development. We are going to use a JS framework called Vue.js. You remember a **framework is a library that you can use to create an application**: like Django is a web framework built in Python.

We already learned some JS basics. This will be useful for Vue.js, as we have to write JS code. But Vue.js is going to help us to have a structure and take care of a lot of the JS in terms of refreshing some sections of the html by itself when data is going to change. This will be very helpful to create a more complicated and interactive frontend.

You can use Vue.js in some sections of your html page, for example in the menu bar, or you can use it for a whole page of your application. You can also have Vue.js in one or multiple pages. Or you can have your entire application done with Vue.js and this moment it will be called a single page application (SPA).

Also, there are 2 ways to use Vue.js in your application:

1. The first one, which is the easiest way, but limited in its use (especially in using some Vue.js libraries). It imports Vue.js into our HTML, so the developer writes the Vue code along with HTML.
2. The second way involves building the application that requires setting up a configuration for our Vue application, and especially being able to make it work with our Django application.

We are going to learn both ways.

First, let's go for the easiest configuration, where we do not need to build a Vue.js application.

Let's create the files: index.html and our_app.js

In our html, we will have a div with the id="app": `<div id="app">{{ message }}</div>`

index.html:

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>Movie Theater</title>
</head>

<body>
    <div id="app">{{ message }}</div>
    <script type="module" src="our_app.js"></script>
</body>

</html>
```

{{}} have more or less the same behavior than in Django.

As you can see the html code calls the script our_app.js which imports 2 functions from Vue.js: create the Vue.js app, and mount it on the div id="app".

our_app.js:

```
import { createApp, ref } from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'

 createApp({
   setup() {
     const message = ref('Hello Vue!')
     return {
       message
     }
   }
 }).mount('#app')
```

Save your classwork here: **your_repo/unit_5/movie_theater/your_files_here**

Let's use the createApp differently now.

We are going to insert some values inside a list, using an input field, and click on a button to insert this value inside the list, and display the list in a tag:

```
<div id="app">
    <div>
      <input type="text" id="my_value" v-model="inputValue" />
      <button v-on:click="addValue">Add value</button>
    </div>
    <br/>
    <ul>
      <li v-for="value in myList">{{ value }}</li>
    </ul>
</div>
```

And our Javascript code:

```
import { createApp } from 'https://unpkg.com/vue@3/dist/vue.esm-browser.js'

createApp({
 data() {
   return {
     myList: [],
     inputValue: ""
   };
 },
 methods: {
   addValue() {
     this.myList.push(this.inputValue);
     this.inputValue = "";
   },
 }

}).mount('#app')
```

In our Vue app, we are going to introduce a data section and a methods section. We are using the structure of the Vue js framework.

In the data section, which is a function, we are going to return the initial state of our data, and define the name of our variables.

In the methods section, we define the function that our application will use.

Notice, I'm initializing a string and a list in the data section.

And in our html, I'm setting up an input field:
<input type="text" id="my_value" v-model="inputValue" />

We need to connect our html element with our vue js app, and for that, Vue.js introduces a lot of keys that begin usually with **v-something**, and there will be a shortcut with **@ or :** (we will see that later).

Here with an input field, you can connect the input with a Vue.js data variable with the key **v-model**. If you change the value in the html side, the value in javascript will be updated automatically. If you change the value in the javascript side, you will see it display automatically on the html. This is the magic of Vue js, it is going to take care of a lot of the implementation for us, and coding in Vue js will help us to let our creativity flow, while it takes care of the annoying stuff.

In our Vue app, we have defined our method addValue. It is going to take the value from the input field, and then add it to the list, and then we reinitialize the inputValue.
We connect this method to the html button with the key **v-on:click** (which means, when we click on this button, use this vue js method).

And then, we are going to display the list. We create a <ul> element, and we are going to create a vue js loop inside using <li> elements, with the key **v-for.** We are looping over the list myList using for each iteration the value 'value', and we are going to insert this value inside the <li> elements with {{ }}.

What do you think about this simple example?

Test the code on your side, check the html when you add a value on the list.

Homework (It is already posted on the website):

Research: How to retrieve the variables values myList and inputValue in the console (developer tool)? Once you succeed, change the inputValue, and push a value to myList. What does it happen when you do that? Is the html automatically updated?

Write your answers here: **your_repo/homework/unit_5/02_08_variables_console.txt (md or pdf file)**

*** Day 02 ***

Vue.js - Intro

Let's get a html link from a Vue variable:

```
data() {
  return {
    myList: [],
    inputValue: "",
    htmlLink: "https://vuejs.org",
  };
},
```

How should we include the variable htmlLink in this html code Click here to go to the vue website and learn more about it ?

Yesterday, we used some useful **v- keys**. We have a key to connect the attributes of elements our vue app, it is **v-bind**, and then you can write javascript code inside the "", so here we just put our variable name:

```
<p><a v-bind:href="htmlLink">Click here to go to the vue website and learn more about it</a></p>
```

Let's check your webpage to verify the link was added.

Instead of writing **v-bind:href**, you can also write **:href.**

Between the {{}}, you can call Vue.js data variables. Also, you can also call simple javascript code, like {{ 1 + 1}}, or {{ Math.random() }}.

What about writing simple code like this {{ if (true) 'bob' }}? Well, that will not work. You may try it and check the error in the console, it throws an error about the expected expression, got keyword 'if'.

Anything inside {{ }} should be a simple one call, not complex calls.

I am adding a randomNumber method in our vuejs app:

```
methods: {
  addValue() {
    this.myList.push(this.inputValue);
    this.inputValue = "";
  },
  randomNumber() {
    return Math.random();
  }
}
```

And calling it from our html using {{}}:

{{ randomNumber() }} This will work, which is great because this functionality could be very useful when creating our websites.

Do not forget when you want to use a data variable from our vuejs app inside a method in your vuejs app, you need to use the keyword **this**. Remember we used that in the code yesterday for inputValue and myList.

Return html code from Vue:

In case you need to return some html code from a vuejs method.

In your vue file:
htmlText: " This is some html code coming from vue js "

In html:
{{ htmlText }}

You will see that vuejs displays the text in on your web page, and is not considered as html code. We need to introduce a new vue key: **v-html**. So let's use it, we need it to insert as an attribute of an element, we can use a div for example (but you can use any element that suits your needs).

We have learned how to define variables and methods in vue js, and be able to interact with it in the html code, make our html interface more interactive, and it was easy to do it.

Let's learn some events now.

Let's add a variable: clickCounter: 0

And two buttons: Add and Remove

```
<div>
    <button>Add</button>
    <button>Remove</button>
    <p>clickCounter: {{ clickCounter }}</p>
</div>
```

****

Exercise:

We learned already how to create an event for a button. Let's do it again, this time when you click on "Add", the clickCounter should increase, and with the "Remove" button it should decrease.

For this exercise do not create an extra method, you must do it in the html tag only. Remember you can write JS in the element attribute (the one which connects with vuejs, with a key v-....)

****

Let's add this to your html:

```
<p>{{ Math.random() }}</p>
<p>{{ randomNumber() }}</p>
```

Change the value everytime we click on Add and Remove. That means vuejs is recalculating these expressions every time that there is a change in the data (you can try it by changing values from the console, you will see that vue js recalculates and refreshes values on the webpage). This is important to remember, and we will have an alternative when you would like to avoid this behavior for a method.

****

Exercise:

Now, replace the v-... attribute you added to your buttons with a method.You must define your methods in the JS file. When you call a vue method from html if you do not include (), Vue will execute the method anyway (myMethod() or myMethod)

****

****

Exercise:

Now let's add an argument to your methods. Example: increaseCounter(step), increaseCounter(step) step will be added or subtracted from your counter variable. Make steps for user input.

****

*** Day 03 ***

Vue.js - Events and Modifiers

Let's add an input field that will be displayed in a html element. An event will help us to do that:

```
<p>Enter you name: <input type="text" v-on:input="setName($event, 'Smith')" ></p>
<p>Your Name is: {{ name }}</p>
```

$event => reserved name to get access to the built-in event by the browser.

In your JS file, add:

```
setName(event, lastName) { // the event you get it from the browser
        this.name = event.target.value + ' ' + lastName;
},
```

Where are you going to add the previous code? Is there anything else that should be added to this file?

Forms:

Create a form in html:

```
<form>
        <input type="text">
        <button>Sign up</button>
</form>
```

There is a problem here when you click on the button the entire page refreshes. It is because the request is sent to the server. There are two ways prevent this to happen:

```
<form v-on:submit="submitForm">
        <input type="text">
        <button>Sign up</button>
</form>
```

In your JS file in the section methods, you may add:

```
methods: {
        …..
        submitForm(){
                alert('Submitted!');
```

```
        }
}
```

But, the page still gets reloaded. Data in other fields gets lost.

There is a JS method we can use in Vue event.preventDefault(). That will prevent other data from being refreshed.

```
methods: {
        …..
        submitForm(event){
                event.preventDefault();
                alert('Submitted!');
        }
}
```

Vue has a nicer way to do that. There are some built-in modifiers in Vue to change an event behavior. Add a dot (.) after v-on:submit. if you want a modifier. Example: v-on:submit.prevent, it will prevent the browser default (refresh pages)

```
<form v-on:submit.prevent="submitForm">
        <input type="text">
        <button>Sign up</button>
</form>

methods: {
        …..
        submitForm(){
                alert('Submitted!');
        }
}
```

We can add modifiers to events to say when we want them to occur. Let's say, we want to increase or decrease the values in our previous example only when the user right clicks on the buttons. Let's add the modifier to our buttons (v-on:click.right):

```
<button v-on:click.right="increaseCounter(counterStep)">Increase</button>
<button v-on:click.right="decreaseCounter(counterStep)">Decrease</button>
```

With that we want to tell Vue that it should only react when there is a right click.

*** Day 04 ***

Vue.js

Topics: More Events, Two Way Binding, Computed Properties

Let's add a keyup event to an input field. Let's use the name text field from our previous lesson as an example.

HTML:

```
<p>Enter you name: <input type="text" v-on:input="setName($event, 'Smith')"
v-on:keyup="setName2"></p>
<p>Your Name is: {{ name }}</p>
<p>Your name, but with another event: {{ name2 }}</p>
```

Then in the JS file:

- Add the data variable name2: ""
- In method, add:

```
setName2() {
    this.name2 = this.name;
  },
```

Q: What happened?

Now, let's add something on the keyup event to make it trigger when the enter key is pressed:

```
<p>Enter you name: <input type="text" v-on:input="setName($event, 'Smith')"
v-on:keyup.enter="setName2"></p>
```

You may use **v-once** to refresh an element once.

```
<p>{{ Math.random() }}</p>
<p v-once>{{ randomNumber() }}</p>
```

You will see the value of randomNumber() refreshed only once when you charge the page.

Two Way Binding

We are listening to an event coming out of the input element (input event) and at the same time, we are writing the value back to the input element through its value attribute. We communicate in both directions.

Let's add a Reset Input button, to clear an input field. We need to add a method to perform that action.

HTML:

```
<input type="text" v-model="name">
<button v-on:click="resetInput">Reset Input</button>
```

JS:

```
resetInput() {
   this.name = '';
}
```

As you can see with the two-way binding we lost the functionality of having the last name added to the name. So, you can create another method for that if you would like:

HTML:

```
<p>Your Full Name is: {{ outputFullname() }}</p>
```

JS:

```
outputFullname() {
        return this.name + " " + "Smith";
},
```

Let's add Smith only if the name is not empty:

```
outputFullname() {
        if (this.name === "")
                return '';
        return this.name + " " + "Smith";
},
```

However if anything else changes on the web page Vue will refresh outputFullname(), and probably you do not want that.

Computed Properties

These are like methods. The big difference is that those will get updated only if some dependencies change. Add the following to your Vue code at the same level of data and methods.
JS

```
data: {
        …
}
computed: {
        fullname() { // It is a method, but we are going to use it as a property.

                if (this.name === "")
                        return '';
                return this.name + " " + "Smith";
        },
 },
Methods: {
        …
}
```

In your HTML file, you do not call fullname(), instead you point to it fullname => without(). And Vue will call it for you. Basically, you use computed methods as properties, that is why you should point to and name them as a property.

Vue will check if fullname is a property (data section), if it is not there, Vue will check the computed section and execute it.

Now, if you refresh any other element the name will not be refreshed. In our example, VUE will execute fullname only in a dependency (this.name) changes.

For output performance, it is better to use computed properties than methods. Use methods, when you want to refresh elements anytime that another element changes in the web page.

*** Day 05 ***

Watchers

A watcher is a function you can tell Vue to execute only when dependencies change. This definition is the same as computed properties. Let's learn the differences.

Inside a watch, you can define a method with the same name as a data variable. For example, we can write a method name in our code because we have a name variable.

```
watch:{
   name() {
     this.name2 = this.name + " - This is setup with a watcher. So
cool and so fast."
   },
 },
```

Try it.
This is much better than when we use events in JS.

You can also define it like this:

```
watch:{
   name(value) {
     this.name2 = value + " - This is setup with a watcher. So cool
and so fast"
   },
 },
```

If by any reason, you need the previous value, you can write your code like this:

```
watch:{
   name(value, oldValue) {
      this.name2 = value + " - This is set up with a watcher. So cool
and so fast. Previous value: oldValue: " + oldValue
   },
 },
```

Different tasks for watcher and computed properties

A watch and a computed property are similar. However, a **computed property has the ability to check over multiple dependencies but it always returns a value** that you will display in a certain HTML section. A **watch can only follow one variable data, but it does not need to return a value**, you can set up different actions, change values. You could also do that with a computed property, but computed properties are primarily created to return a value that you display on your HTML page.

Computed properties will be used more to return a result you want to show on HTML.
Watcher will be used when you want to have some logic and modify some data variables.

Summary

Methods are used for event binding (example: click on a html element) or data binding (return a value and put it on the HTML code with {{ }})
Do not forget, if you use a method that returns a value used in the html, it will be executed every time that a field/value changes on the web page.

Computed properties will be used for data binding (return a value and put it on the html with {{ }}). These properties will be executed only when one of its dependencies change (and not when any data variable changes like for methods)

Watcher will be used to execute some logic/action that depends on a data variable.

Dynamic Styling

Using inline styles

Let's add a div in the HTML code. We are going to modify a bordercolor.

```
<div style="margin-top:20px;border-style:solid;border-color:#ccc">
       A div
</div>
```

Add the following data variables and use them in the v-bind:style that we are going to write.

```
data() {
   return {
     …
     borderColor: "#ccc",
     borderColorInd: false,
   };
```

Reminder => `:style` is for `v-bind:style`

```
<div :style="{
       'margin-top':'20px',
       'border-style':'solid',
       'border-color':borderColor
     }">
```

For events, we can use the **v-on**, which we typically shorten to the **@ symbol,** to listen to DOM events and run some JS when they're triggered. The usage would be **v-on:click="handler"** or with the shortcut, **@click="handler"**.

****

Exercise 5.1

Let's create an event. When we click on the div, the border color should change from #ccc to red (alternate between these two colors). Your HTML should look like this:

```
<div :style="{
       'margin-top':'20px',
       'border-style':'solid',
       'border-color':borderColor
     }"
       @click = "changeBorderColor"
      >
       A div
     </div>
```

Note: `'border-color':borderColor` could be replaced by `borderColor:borderColor`

What should we do now?

****

Exercise 5.2

You can have an alternative solution for exercise 5.1 using the variable borderColorInd. Let's do the same as the previous exercise, but now you must use the borderColorInd variable with values false/true. Write a different method, so you can keep the previous one as reference.

```
<div :style="{
      'margin-top':'20px',
      'border-style':'solid',
      borderColor:borderColorInd ? 'red': '#ccc'
    }"
      @click = "revertBorderColorInd"
     >
      A div
    </div>
```

****

This last exercise was done using **CSS inline styling.**

Using a CSS class

Let's create a css file, **styles.css**

Reminder: Include the CSS file in the head section of the HTML:

```
<link rel="stylesheet" href="styles.css">
```

Add the following code in the CSS file:

```
.grey_border {
   margin-top: 20px;
   border-style: solid;
   border-color: #ccc;
}
```

HTML:

```
<div :class="'grey_border'"
       @click = "revertBorderColorInd"
      >
       A div
     </div>
```

You should see the gray border.

Why do we have to write "grey_border" as a string using quotations? It is because the `v-bind:` (`or shortcut :`) acts as JS inside, so it is not a VUE variable, so we need to put the word as a string. Otherwise, VUE is going to look for this variable inside in the data section.

We can define the class to add the red border color as we did for the styling, with the **question mark ?.**

****

Exercise 5.3

Write another CSS class that will define the color red border, and then set it in the :class (the color that is going to have priority on the CSS is the one defined last).

HTML:

```
<div :class="borderColorInd ?'grey_border red_border': 'grey_border'"
     @click = "revertBorderColorInd"
    >
     A div
   </div>
```

****

You do not need to change the VUE code, the event is the same.

Good, it is working. However, we are not going to use a VUE.JS feature for a CSS class, where we can define the :class as an object and give the value for each class a true or false (or in this case, use a method or a data value)

Let's see how it works:

```
<div :class="{grey_border:true, red_border: borderColorInd}"
       @click = "revertBorderColorInd"
      >
       A div
     </div>
```

Here we go, now the syntax is clean and clear, we know when each class has to appear with its condition.

We can also go a little bit further, and as we always have the grey_border (always true), then we are going to statically put as regular HTML class, and have also the :class (you can have both at the same time)

```
<div class="grey_border" :class="{red_border: borderColorInd}"
       @click = "revertBorderColorInd"
      >
       A div
     </div>
```

Clean and nice!!!!

When you have more complex cases, you can also return the CSS classes with computed properties.

In the VUE code, in computed properties:

```
redBorder() {
     return {red_border: this.borderColorInd}
   },
```

HTML:

```
<div class="grey_border" :class="redBorder"
       @click = "revertBorderColorInd"
      >
       A div
</div>
```

Here we have nothing special in the computed properties, but if you need to add some conditions, a computed property could help by returning the object classes.

You can also return an array of class object in case you need to have this flexibility:

```
<div :class="['grey_border', redBorder]"
       @click = "revertBorderColorInd"
      >
       A div
</div>
```

*** Day 06 ***

Conditionals

The directive **v-if** is used to conditionally render a block.

The **v-else-if** serves as an "else if block" for v-if. It can also be chained multiple times.

A **v-else** element must immediately follow a v-if or a v-else-if element - otherwise it will not be recognized.

In previous lessons, we added buttons to increase/decrease a counter in our web page. You should already have this code in your html file:

```
<button v-on:click.right="increaseCounter(counterStep)">Increase</button>
<button v-on:click.right="decreaseCounter(counterStep)">Decrease</button>
<input type="number" step="1" v-model="counterStep" />
<p>clickCounter: {{ clickCounter }}</p>
```

Let's replace the <p> (code in red) with the following code:

```
<p v-if="clickCounter != 0">clickCounter: {{ clickCounter }}</p>
<p v-else>No counter</p>
```

Do not forget to right-click on increase to make clickCounter increase, and show the functionality of v-if and v-else.

We can also add a condition on a list element. We have a list already defined in our code, look for myLIst, and add this condition:

```
<div v-if="myList.length > 0">
     My List:
     <ul>
       <li v-for="value in myList">{{ value }}</li>
     </ul>
</div>
```

And you are going to see this div element only if myList has elements.

There is an alternative, you can use **v-show** instead of v-if:

```
<div v-show="myList.length > 0">
     My List:
     <ul>
       <li v-for="value in myList">{{ value }}</li>
     </ul>
</div>
```

What is the difference between v-show and v-if?
Which one will perform the task faster?

V-for

We already learned that **v-for** acts as a loop. If you **traverse an array**, you can get more information than just the value of an element. For example, you can also get the index of the item in the array.

```
<div v-show="myList.length > 0">
      My List:
      <ul>
            <li v-for="(value, index) in myList">{{ value }} - {{ index }}</li>
      </ul>
</div>
```

You can also use loops to **iterate through object properties:**

```
<p>Object Properties</p>
<ul>
       <li v-for="(value, key, index) in {name: "Peter", age: 15}">{{ value }} : {{ value }} - {{index}} </li>
</ul>
```

Looping through a **range of numbers**:

```
<p>Array of numbers</p>
<ul>
       <li v-for="num in 5">{{ num }}</li>
</ul>
```

Removing list items:

We are going to add a listener to the element in the list with @click and call the method removeItem(index). Notice we are sending the index as parameter:

HTML:

```
<div v-show="myList.length > 0">
	My List:
	<ul>
		<li v-for="(value, index) in myList" @click="removeItem(index)">{{ value }} - {{ index }}</li>
	</ul>
</div>
```

VUE:

```
removeItem(index) {
	this.myList.splice(index, 1); // splice find an element at index and remove it
},
```

It was easy, that way we do not have to render the list every single time we delete an item from there.

Lists and Keys

I would like to have a paragraph in each element of the list, so I can add an input field. So let's do the following:

```
<div v-show="myList.length > 0">
	My List:
	<ul>
		<li v-for="(value, index) in myList" @click="removeItem(index)">
			<p>{{ value }} - {{ index }}</p>
			<input type="text">
		</li>
	</ul>
</div>
```

Did it work?

Let's try:

```
<div v-show="myList.length > 0">
      My List:
      <ul>
            <li v-for="(value, index) in myList" @click="removeItem(index)">
                  <p>{{ value }} - {{ index }}</p>
                  <input type="text" @click.stop>
            </li>
      </ul>
</div>
```

By adding, @click.stop, you will stop the event.

Bug!!!!!

- Let's add two elements to your list.
- Add something in the first element input.
- Delete the first element.
- What happened?

How to fix it?

VUE updates the list when you add/remove elements and updates the DOM as well. VUE tries to optimize the DOM update by reusing the elements. So, if we have 2 DOM elements (one for each item in the list) and I delete the first one VUE will not render the entire list, which means it will not delete the first DOM element, but it will just move things into the first DOM element. Just the dynamic elements {{}} will be changed, but not the field input.

Since all the elements we are using to display the list are <li> there is no way to identify the item. There is an extra feature we can add to our <li> tabs, and it is **:key.** VUE will detected that key which must be a unique id. **It is a good idea to use :key with v-for.**

Index is not a good idea to be used as an identifier because it will get updated after adding/removing elements from the list. When we will be dealing with the backend, a unique id from a database will work.

For now, I will use the value as identifier (will not work if you have a duplicate value):

```
<div v-show="myList.length > 0">
      My List:
      <ul>
            <li v-for="(value, index) in myList" :key="value"
      @click="removeItem(index)">
                  <p>{{ value }} - {{ index }}</p>
                  <input type="text" @click.stop>
            </li>
      </ul>
</div>
```