Universitat Autònoma de Barcelona

Vue.js Exercises

Setup

Unless otherwise specified, use the Vue project template in the virtual campus:

1. Uncompress the file.
2. Rename the uncompress folder to **`vue-project`**.
3. Go the directory **`vue-project`**.
4. Install the node modules specified in the `package.json` file by running the command: **`npm install`**
5. Copy the file **`App.vue`** into the file **`App#.vue`** where # corresponds to the exercise number you are working on.

Exercises

1. Create web page that counts from 0 to infinity. Create a vue instance with

   - a **`counter`** property in its data option, which is initially set to `0`,
   - a template that *interpolates* the **`counter`**, and
   - use the following piece of code to increase the counter.

   ```
   setInterval(() => this.counter++, 100);
   ```

   - Clear the interval when the vue instance is **`unmounted`**.

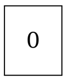


Figure 1: Resulting web page at its initial state.

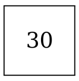


Figure 2: Resulting web page after 3 seconds.

2. Create a vue instance that has a template with

   - two **`<input>`** controls *bound* to variables **`a`** and **`b`**, respectively, and
   - after these two controls, an interpolation of the addition of **`a`** and **`b`**.

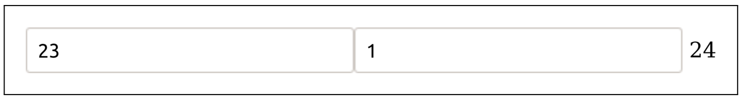


Figure 3: Example of the addition of `23` and `1`.

Note: the term *bound*, as employed here, means that there is a data binding between the variable and the form control. It may either be an one-way or a two-way binding (with **`v-bind`** or **`v-model`**, respectively).

3. Create a vue instance with a single **`<button>`** that disappears when clicked.

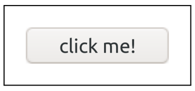


Figure 4: Initial state.

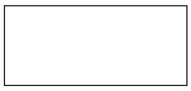


Figure 5: After clicking the button.

4. Modify the previous exercise so as when the button is clicked, instead of disappearing, it is disabled.

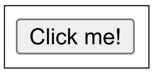


Figure 6: Initial sate.

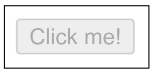


Figure 7: After clicking the button.

5. Create a vue instance with an empty `<input>` text box. The text box clears itself when its text length reaches 5 characters (or surpasses that number).

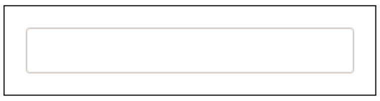


Figure 8: Initial state, and after writing five characters.

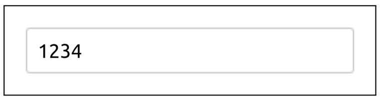


Figure 9: After writing four characters.

Think of two different ways to solve the problem.

6. Create a vue instance with an empty `<input>` text box. The text box turns red when keys are pressed, and restores its original color upon key release.

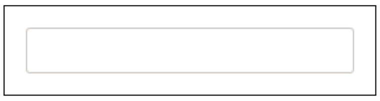


Figure 10: Initial state.

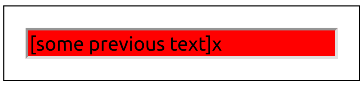


Figure 11: After pressing (but not releasing) the ‘x’ key.

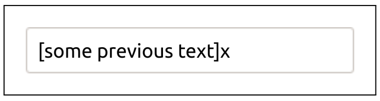


Figure 12: After releasing the ‘x’ key.

Hint: **`v-on:keydown`**, **`v-on:keyup`**.

7. Modify the previous exercise by binding the HTML attribute `class` to the CSS classes: **`pressed`** or **`released`** depending on if we are pressing a key or not. Use the following styles:

```
input.pressed{
  background-color: red;
}

input.released{
  background-color: aqua;

}
```

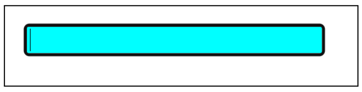


Figure 13: Initial state.

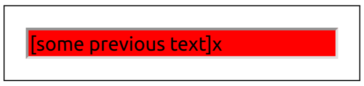


Figure 14: After pressing (but not releasing) the 'x' key.

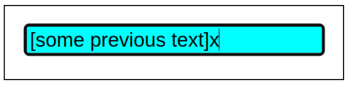


Figure 15: After releasing the 'x' key.

Hint: `v-on:keydown`, `v-on:keyup` '

8. Using the following template, create an instance that changes the 'redness' of the **AM I RED?** text according to the value in the range slider. Hide the **YES!** text when redness is under 70%.

```
<div>
  <div style="color: hsl(0,x%,50%)">AM I RED?</div>
  <input type="range" min="0" max="100">
  <div>YES!</div>
</div>
```

Note that in the CSS rule `color: hsl(0,x%,50%)`,when $x >= 70$ is when the color will be red.

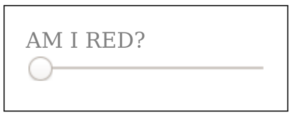


Figure 16: When 'redness' is `0`.

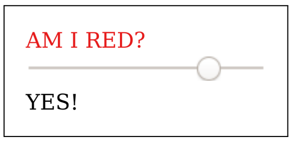
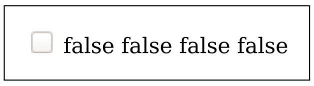


Figure 17: When 'redness' is `70`.

9. Create a vue instance with:

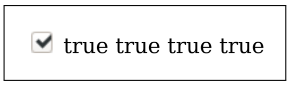
   - the properties **a**, **b**, **c** and **d** in its data option (initially set to `false`), and
   - a template with an `<input type=checkbox>` bound to **a**, followed by the interpolation of the four variables (**a**, **b**, **c** and **d**).

   Create a watch function for the variable **a** that sets **b** equal to **a**. Similarly, create a watch function for the variable **b** that sets **c** equal to **b**, and a watch function for the variable **c** that sets **d** equal to **c**.



☐ false false false false

Figure 18: Initial state.



☑ true true true true

Figure 19: After switching the checkbox.

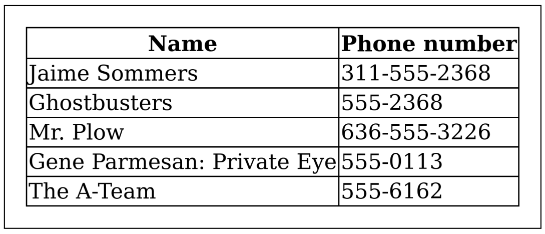
10. Modify the previous exercise so that there is just the property **`a`** in the **`data`** option and **`b`**, **`c`** and **`d`** are *computed* properties that follow the same behaviour above. Therefore, you will not need the *whatchers*.

11. Create a vue instance that displays the following phone book as shown in the accompanying figure.

```
[
  { name: 'Jaime Sommers', phone: '311-555-2368' },
  { name: 'Ghostbusters', phone: '555-2368' },
  { name: 'Mr. Plow', phone: '636-555-3226' },
  { name: 'Gene Parmesan: Private Eye', phone: '555-0113' },
  { name: 'The A-Team', phone: '555-6162' },
]
```

Employ the following CCS style.

```
table { border-collapse: collapse; }
table th,td { border: 1px solid black; }
```



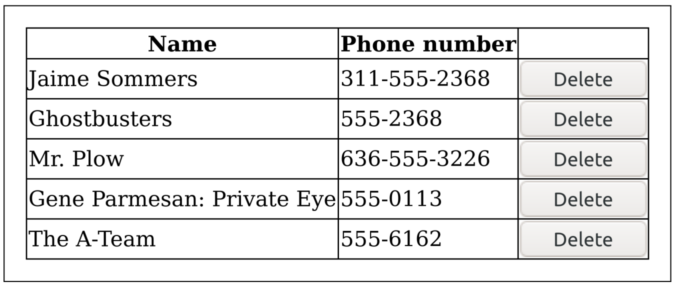
Name	Phone number
Jaime Sommers	311-555-2368
Ghostbusters	555-2368
Mr. Plow	636-555-3226
Gene Parmesan: Private Eye	555-0113
The A-Team	555-6162

Figure 20: How the phone book must be rendered.

Hints: **`v-for`** and the following html code:

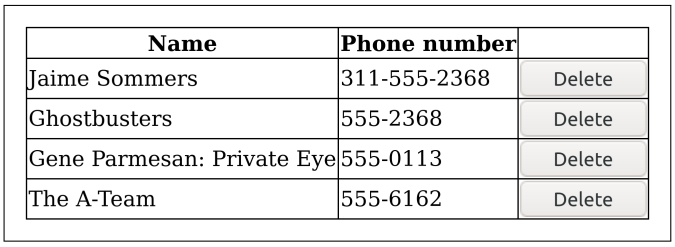
```
<table><tr><th>Name</th><th>Phone number</th></tr>
<tr><td>{{item.name}}</td><td></td></tr>
</table>
```

12. Extend the previous 'phone book' exercise by adding delete buttons. Add a third column with an individual delete button for each entry.



Name	Phone number	
Jaime Sommers	311-555-2368	Delete
Ghostbusters	555-2368	Delete
Mr. Plow	636-555-3226	Delete
Gene Parmesan: Private Eye	555-0113	Delete
The A-Team	555-6162	Delete

Figure 21: Initial state.



Name	Phone number	
Jaime Sommers	311-555-2368	Delete
Ghostbusters	555-2368	Delete
Gene Parmesan: Private Eye	555-0113	Delete
The A-Team	555-6162	Delete

Figure 22: After deleting the ‘Mr. Plow’ entry.

Hints:

- `list.splice`,
- `v-for="(item,index) in list"`.

13. Suppose this is a ‘semaphore’:

```
  <div class="trafficlight">
    <div class="light redOff"></div>
    <div class="light yellowOff"></div>
    <div class="light greenOn"></div>
    <button>Switch</button>
</div>
```

where the CSS style is defined in the component as:

```
<style>
  div.trafficlight{display: inline-block; width:30px;}
  div.light{height: 30px;}

  div.redOn{ background-color: red}
  div.redOff{background-color: indianred}
  div.yellowOn{background-color: yellow;}
  div.yellowOff{background-color: khaki;}
  div.greenOn{background-color: lime;}
  div.greenOff{background-color: seagreen;}
</style>
```

The CSS classes that represent that the lights are on are: **`redOn`**, **`yellowOn`** and **`greenOn`**; and the classes that represent that the lights are off are: **`redOff`**, **`yellowff`** and **`greenOff`**.

Create a web page that:

- renders the semaphore in a vue template,
- has a **`state`** variable, which is an integer representing which light is on, and
- has a **`<button>`** that switches the semaphore **`state`**.

A value of `0` for **state** denotes a green light, a value of `1` denotes a yellow light, and a value of `2` denotes a red light. The initial state is `0`.

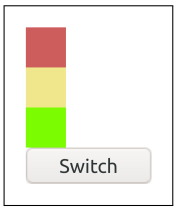


Figure 23: Initial state, and after 3 button clicks.

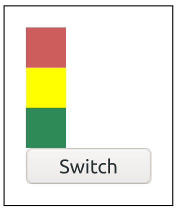


Figure 24: After 1 button click.

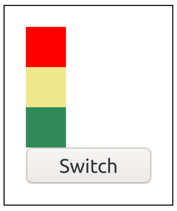


Figure 25: After 2 button clicks.

14. Based on the previous exercise, consider that the semaphore implements inline CSS styles:

```
<div style="display: inline-block; width:30px;">
  <div style="height: 30px; background-color: indianRed"></div>
  <div style="height: 30px; background-color: khaki"></div>
  <div style="height: 30px; background-color: seagreen"></div>
  <button>Switch</button>
</div>
```

Implement the same functionality as in the previous exercise but managing the inline CSS styles to turn on or off the light depending on the semaphore state.

Use the following css colors to represent when lights are on: **red**, **yellow**, and **lime**; and use the following to represent when lights are off: **indianRed**, **khaki**, and **seagreen**.

15. Create the component `<WordsToList>` that transforms words into list items. Words given through the **words** attribute are transformed into `<li>` elements inside an `<ul>`.

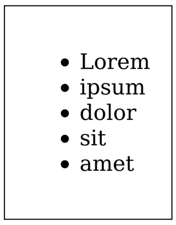
For example, `<WordsToList words="w1 w2 w3"></WordsToList>` is transformed into `<ul><li>w1</li><li>w2</li><li>w3</li></ul>`.

Hint:

```
- string.trim()
- string.split(' ')
```

Usage example:

```
<!-- App.vue -->
<template>
  <WordsToList words="Lorem ipsum dolor sit amet" />
</template>
```



- Lorem
- ipsum
- dolor
- sit
- amet

Figure 26: Result when **words** is "**Lorem ipsum dolor sit amet**".

16. Modify the previous exercise to use in the parent conmponent a text `<input>` element to introduce the words. While the user is typing words, they appear in the list. Make sure no bullet appears without any associated word.
17. Modify the previous exercise by adding a `<button>` element to perform the conversion from the input data to a word list only when the button is clicked.
18. Create the component `<Card>`, which is used to render user information. It is used as follows:

```
<script>
import Card from './components/Card.vue'
export default {
  components:{
    Card
  },
  data() {
    return {
      person: {
        name: 'My Name',
        picture: `data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAY
AAAAfFcSJAAAADUlEQVR42mM82Mz1HwAFqgJP3gasfwAAAABJRU5ErkJggg==`,
        email: 'me@somerandomdomain.com',
        phone: '+00 00 000 0000',
      }
    }
  },
}
</script>
<template>
  <div style="display:flex;">
    <!-- TODO: Use the component passing to it the person prop -->
  </div>
</template>
```

For the previous data, the component template has to yield this final result:

```
<div class="card">
  <div>
    <img src="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAAEAAAABCAY
          AAAAfFcSJAAAADUlEQVR42mM82Mz1HwAFqgJP3gasfwAAAABJRU5ErkJggg==">
  </div>
  <div><h1>My Name</h1></div>
  <div>me@somerandomdomain.com</div>
  <div>+00 00 000 0000</div>
</div>
```

Employ the following css style:

```
.card { font-family: Roboto; text-align: center; background: #ffbcbc;
  box-shadow: 6px 6px 8px #888; margin: 15px;}
.card div {padding: 10px; }
.card img { width: 100px; }
```

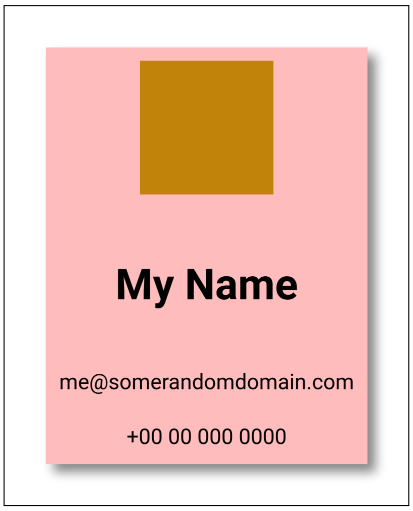


Figure 27: The `<Card>` component, as employed in the previous example.

19. Create a `<SwitchButton>` component as follows:

   - The rendered component has to resemble the following html snippet.

```
<div style="border:solid;display:inline-block">
  <button>ON</button>
  <button disabled>OFF</button>
</div>
```

   - When the 'ON' button is clicked, the `on` event is dispatched, the 'ON' button is disabled, and the 'OFF' button is enabled.
   - Similarly, when the 'OFF' button is clicked, the `off` event is dispatched, the 'OFF' button is disabled, and the 'ON' button is enabled.

   Regarding to the parent component:

   - It has a reactive variable: `state` that can be set to two possible values depending on the event triggered by the child component:

     'on' event: Sets the `state` variable to the string *'just turned on'*.

     'off' event: Sets the `state` variable to the string *'just turned off'*.

   - It interpolates the reactive variable `state` just beside the `SwitchButton` component.

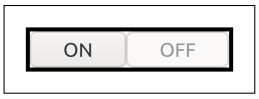


Figure 28: Initial state.

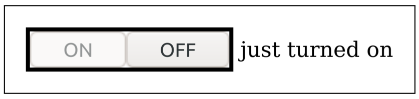


Figure 29: After clicking on the 'ON' button.

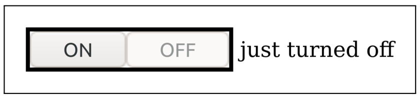


Figure 30: After clicking on the 'OFF' button.

20. Create a `<ColorSelector>` component as follows:

    - The rendered component has to resemble the following html snippet.

```
<div style="border:solid; display:flex;">
  <div style="background-color:#000; width:110px; height:110px;"></div>
  <div style="display:flex; flex-direction:column; padding:10px;">
    <div>R: <input type="range" min=0 max=255> red value</div>
    <div>G: <input type="range" min=0 max=255> green value</div>
    <div>B: <input type="range" min=0 max=255> blue value</div>
  </div>
</div>
```

    - When a new color is selected, the component has to emit a **`color`** event with the selected color value in css format (e.g. color=rgb(0,0,255)).

    Regarding to the parent component:

    - The template to be rendered is the following:

```
<template>
  <div style="border:solid red; display:flex;">
    <!-- Here goes the child component -->
    <div>TEXT</div>
    <!-- Make sure the TEXT changes the color to the selected one-->
  </div>
</template>
```

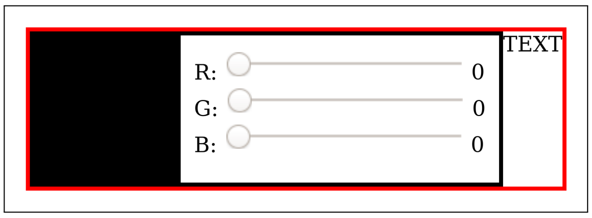


Figure 31: Initial state.

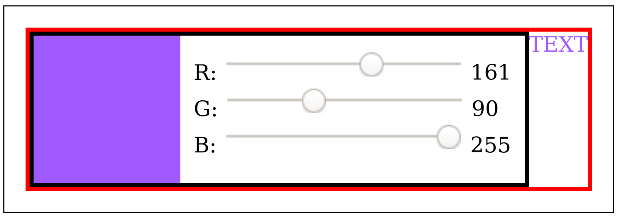


Figure 32: Result after some color selection.

21. Modify the previous exercise so that the parent component, instead of listening to the event **`color`** emited by the child component, uses the directive **`v-model`** to allow the child component to change the **`color`** variable. Modify the child component accordingly to this change.

22. Create a **DoubleCheckbox** component that has two checkbox. The parent component's template includes one standard **input checkbox** and the component **DoubleCheckbox**. When the user checks/unchecks the parent's checkbox, the two checkbox in the component get respectively checked/unchecked. If the user checks/unchecks the two checkbox in the component, the checkbox in the parent is checked/unchecked. If the two checkbox in the component does not have the same state (checked/unchecked), the parent checkbox remains unchecked.

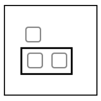


Figure 33: Initial state.

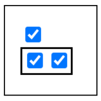


Figure 34: The user checks the input in the parent component or the user checks the two inputs in the **DoubleCheckbox** component.

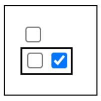


Figure 35: The user checks one of the inputs of the **DoubleCheckbox** component.

You can use the following template and style for the **DoubleCheckbox** component:

```
<template>
  <div>
    <input type="checkbox">
    <input type="checkbox">
  </div>
</template>

<style scoped>
  div{
    display: inline-block;
    border: solid black;
  }
  div input{
    padding-left: 4px;
  }
</style>
```

You can use the following template for the parent component:

```
<template>
  <div>
    <input type="checkbox">
  </div>
<!-- Here goes your component -->

</template>

<style>
</style>
```

23. Create an application that includes a component named **MagicInput** and a regular **input**. The **<MagicInput>** component works like a regular input text box (**<input type=text>**) except that it turns upper case letters into lower case letters and viceversa. The component has to support **v-model** as it will be bound to the parent component's reactive variable **buffer**.

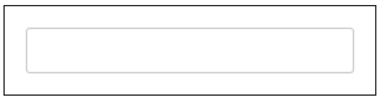


Figure 36: How an empty **<MagicInput>** text box should look like.

Note that the **<MagicInput>** has to display exactly what the user writes, and only change the text case in its model variable. For example, after typing the text **"Hola"** in the **<MagicInput**, the text box of the component has to display the text **"Hola"** and the value of **buffer** has to be **"hOLA"**.

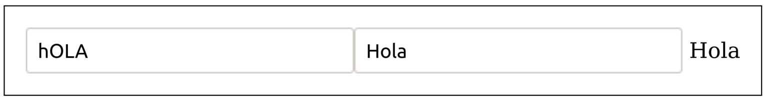
Similarly, when **buffer** is set to **"Test"** from the parent's input, the **MagicBox**'s input text has to display **"tEST"**.

Finally, the parent component also interpolates the reactive variable **buffer** at the end of the regular text input field.

Hints:

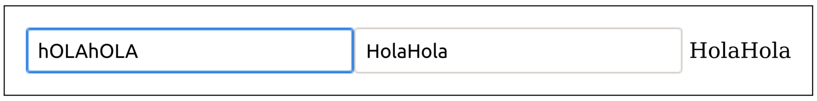
- Use the following case-switching snippet.

```
text.replace(/./g,
  x => x.toUpperCase() == x ? x.toLowerCase() : x.toUpperCase())
```



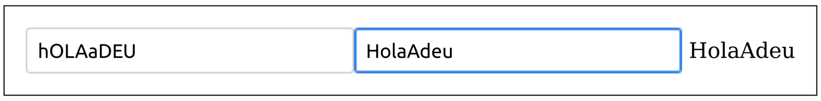
hOLA | Hola | Hola

Figure 37: Initial state.



hOLAhOLA | HolaHola | HolaHola

Figure 38: After appending 'hOLA' in the **<MagicInput>**.



hOLAaDEU | HolaAdeu | HolaAdeu

Figure 39: After appending 'Adeu' in the regular **<input>**. Note that the **<MagicInput>** is updated automatically.