PAU CASESNOVES
Centre Integrat de Formació Professional

Desenvolupament d'aplicacions web

Desenvolupament web a l'entorn client

UT 8: Vue.js

Joan Pons Tugores

Índex de continguts

Bastiments de client

Javascript és pràcticament la única opció per programar el client d'una aplicació web. Per això des de fa temps han sorgit bastiments(frameworks) o llibreries que intenten facilitar aquesta tasca.

Un dels veterans és jQuery que ja hem vist una mica. A més de facilitar la tasca de programar per a qualsevol navegador, jQuery UI, una de les seves extensions, introdueix una sèrie de components per implementar la interfície d'usuari.

Els frameworks han anat evolucionant, n'han sorgit de nous, ... de manera que avui en dia és molt estrany veure aplicacions web fetes en Javascript pur i dur, el que anomenen Javascript Vanilla.

La imatge del costat pertany a un article,

https://hackernoon.com/how-it-feels-to-learn-javascript-in-2016-d3a717dd577f#.mzolt5dqc

que xerra de l'angoixant que pot ser començar amb Javascript i trobar-te amb la multitud de llibreries i bastiments que hi ha disponibles. L'article és de l'any 2016, però la situació no ha minvat, tot el contrari.

SPA

No sé que vos fan pensar aquestes sigles, per un desenvolupador web tenen un sol significat: Single Page Application.

Una aplicació de pàgina única consisteix en un únic document HTML en el que s'allotjarà tota la pàgina. No haurem de demanar més documents HTML al servidor. Els diferents components de l'aplicació es generen a través d'alguna llibreria Javascript, de manera que pràcticament l'únic que s'ha de demanar al servidor són les dades que ha de mostrar el client.

Molts frameworks Javascript faciliten aquesta tasca: Angular, React, Meteor, Vue.js, ...

Vue.js

Vue.js (pronunciat com a view) és un framework Javascript de codi obert pensat per a desenvolupar interfícies d'usuari i aplicacions SPA. Va ser creat per *Evan You,* que treballava per Google en projectes amb Angular i va decidir fer un framework molt més lleuger que Angular que suportàs el que a ell realment li agradava d'Angular.

La primera versió de *Vue* va ser llançada el 2014.

La part central de *Vue*, el core, està centrada en la gestió de la vista de l'aplicació, en el desenvolupament de la interfície d'usuari, encara que podem afegir-hi altres llibreries com *vue-router* per gestionar el *routing*, el pas d'uns components a d'altres, o *pinia* per gestionar l'estat de l'aplicació, entre d'altres.

Les característiques bàsiques d'aquest framework són:

- *Components*: Són una extensió dels elements HTML i permeten encapsular codi reutilitzable. Un component inclou una plantilla "HTML", codi Javascript, que inclou les dades associades al component, i els estils que s'apliquen al component.

Un component pot ser tan senzill com un element d'una llista o tan complicat com un mòdul d'una aplicació.

- *Plantilles*: Utilitza unes plantilles basades en HTML que permeten enllaçar el DOM pintat a la pàgina amb les dades del component. *Vue* compila les plantilles en funcions de renderitzat d'un DOM virtual. Això permet renderitzar els components en memòria abans d'enviar-los al navegador, i calcular el mínim de canvis que haurà de fer el navegador per actualitzar la pàgina.
- *Reactivitat*: Cada component de *Vue* monitoritza els canvis en les dades de les que depèn de manera que en canviar les dades s'actualitza el component a la pàgina, i també, quins són els components que ha d'actualitzar en cada cas.
- *Routing*: Una SPA necessita modificar el seu contingut en resposta a les accions dels usuaris basada en la url actual. Aquesta funcionalitat no està inclosa a *Vue.js*, però si a la llibreria *vue-route*r.

Un exemple

Per aplicacions molt senzilles en *Vue* no necessitam més que incloure *vue* com un script més a la nostra pàgina. Un petit exemple:

```
<!DOCTYPE html>
<html lang="ca">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>

<div id="app">{{ message }}</div>
```

```
<script>
    const { createApp } = Vue

   const aplicacio= createApp({
        data() {
            return {
                message: 'Hello Vue!'
            }
        }
    }).mount('#app')
</script>
</body>
</html>
```

Si picam el codi en un arxiu html i l'obrim al navegador tendrem una aplicació Vue molt senzilla en funcionament.

- L'element <div id="app>... serà el lloc on inclourem la nostra aplicació Vue.
- L'expressió {{ message }} indica a Vue on incloure el contingut de la variable amb aquest nom.

A l'script cream un objecte Vue i li passam un objecte amb una funció, *data*, que conté les dades incloses al sistema de reactivitat. Totes les dades incloses en crear l'aplicació seran supervisades i els elements del DOM que les contenguin seran actualitzats quan el seu valor canviï.

Després, amb *mount()* li deim a quin objecte de la pàgina *index.html* montarem l'aplicació vue. Per norma l'id utilitzat és app.

Si assignam el que torna *createApp* a una variable,

```
const aplicacio=Vue.createApp({
        data() {
            return {
                message: 'Hello Vue!'
            }
        }
    }).mount('#app')
```

podrem provar la reactivitat obrint la consola del navegador (F12) i escriure

```
aplicacio.message="Un altre missatge";
```

En pitjar Enter hauria de canviar el contingut de la pàgina.

En complicar un poc més les aplicacions, per exemple afegint llibreries com *pinia*, ... necessitarem *node* i totes les eines que l'envolten.

Entorn de desenvolupament

Vue necessita node i *npm*, i altres eines. Per preparar un entorn local podeu seguir les instruccions dels següents enllaços:

- node: https://nodejs.org/en/download/
- npm: El gestor de paquets de node. Si no ve inclòs amb node el podeu baixar a https://www.npmjs.com/get-npm
- vue: npm install vue
- vite: Eina per a construir els projectes. npm create vue@latest instal·la i executa *create-vue* l'eina oficial de Vue per crear l'esquelet del projecte.
- Si utilitzau Chrome, és útil l'extensió Vue.js devtools https://chrome.google.com/webstore/detail/vuejs-devtools/nhdogjmejiglipccpnnnanhbledajbpd?hl=es

Una altra opció és utilitzar un ide que doni soport a aplicacions *Vue.js*. Per exemple *VS Code* amb l'extensió *Vue – Official extension*, *Webstorm* inclou soport per *Vue*. ...

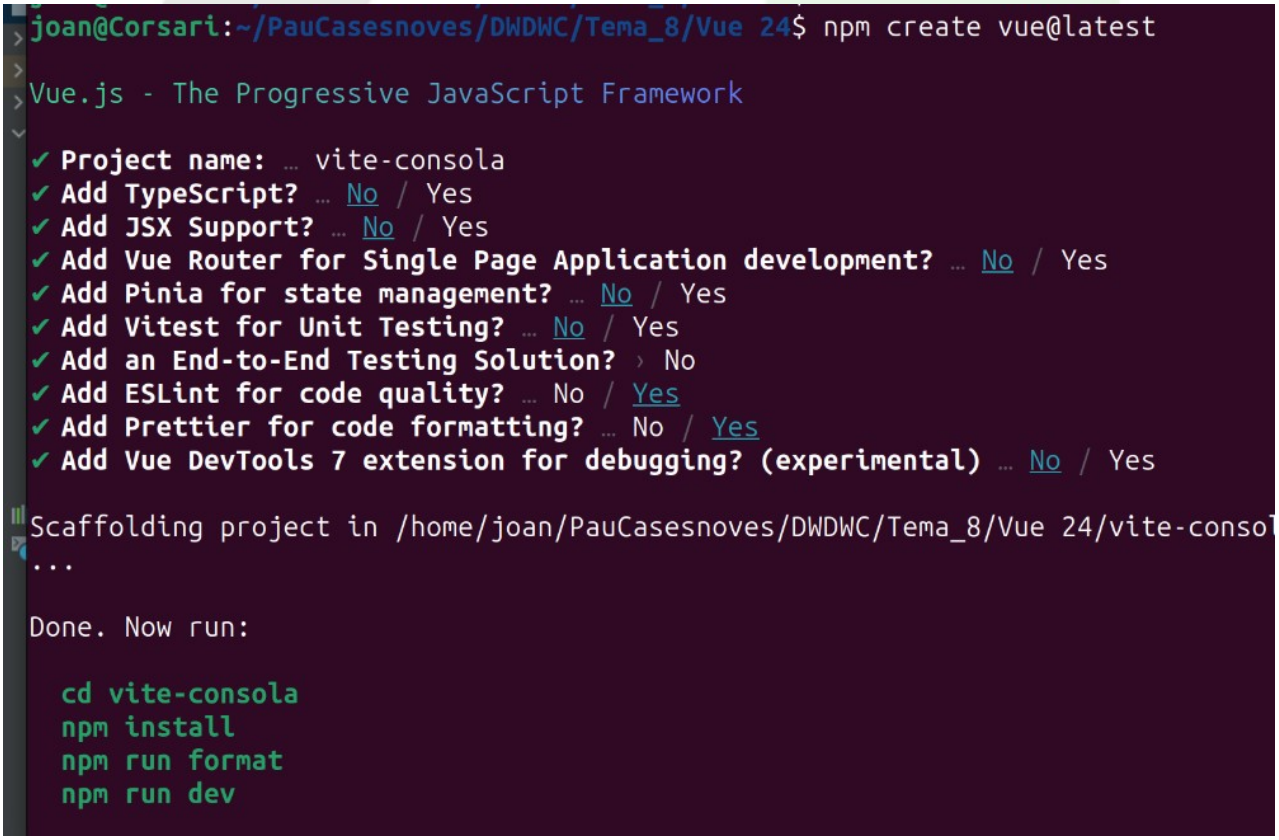
Crear un projecte Vue

Els iDE's segurament inclouen un assistent per crear l'estructura del projecte, encara que la forma recomanada és utilitzar l'eina vite des de consola.

Des de la carpeta pare d'on voleu crear el projecte heu d'executar la comanda

```
npm create vue@latest
```

L'script ens demanarà que volem utilitzar en el nostre projecte. De moment la configuració podria ser la següent:



Encara no utilitzarem ni rutes ni gestió de l'estat, tampoc utilitzarem cap llibreria per fer tests al codi, i podem activar ESLint perquè ens ajudi a trobar errades al codi i Prettier per formatar el codi.

En acabar, l'script vos dona les instruccions que heu de seguir a continuació:

Canvia al directori del projecte

```
cd nom-projecte
```

Al projecte es genera un fitxer anomenat *package.json* amb la configuració del projecte i les dependències que té, els paquets que necessita.

Executa la comanda següent per instal·lar els paquets dels que depèn el nostre projecte:

```
npm install
```

Si voleu donar format als fitxers de codi del projecte (recomanable):

```
npm run format
```

I si voleu posar en marxa l'aplicació:

```
npm run dev
```

Mentre estigui actiu tendreu accés a l'aplicació a la url que vos mostra.

Vue bàsic

Vue disposa de dues API's per treballar, l'*Options* i la *Composition*. En aquesta unitat utilitzarem l'API *Composition*.

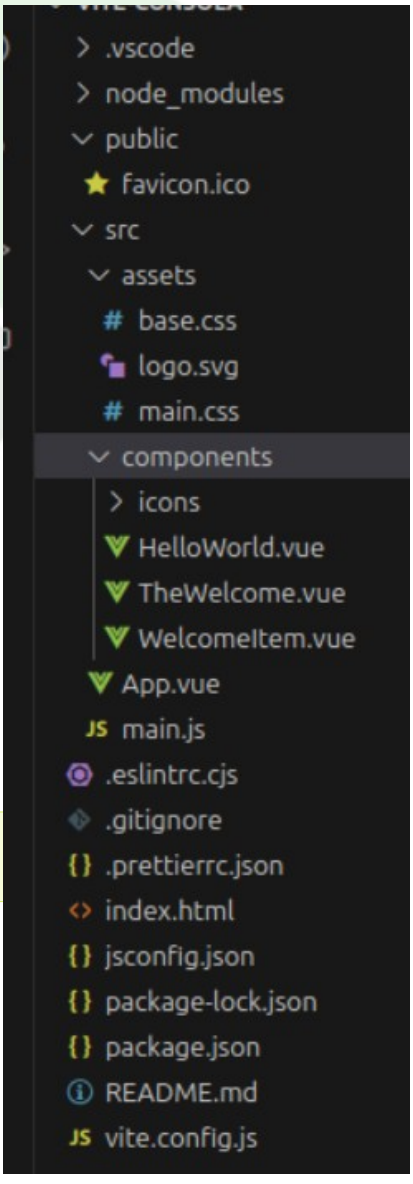
Moltes característiques d'aquesta API no poden ser executades directament al client. El projecte necessitarà una compilació a un servidor *node* abans d'enviar-se al client.

Si volem desplegar l'aplicació a un servidor web necessitarem executar npm run build. El codi que generi aquesta acció serà el que haurem de desplegar.

Estructura d'una aplicació Vue.js

Molt poques vegades les aplicacions seran tan simples com l'exemple anterior amb un sol arxiu *html*.

A la imatge del costat podeu veure l'estructura d'un projecte creat al *codesanbox*:



- **public**: Conté els elements estàtics de l'aplicació, s'han de referenciar amb la ruta absoluta i no són empaquetats dins l'aplicació. Pot incloure l'*index.html*, encara que aquest fitxer també pot ser a la carpeta principal del projecte.
- **index.html**: Inclou l'element on hi haurà l'aplicació Vue.js

```
<div id="app"></div>
```

- **src**: El nostre codi: El seu contingut s'empaquetarà i és accessible a través de rutes relatives.
  - assets: imatges, ...

- components: Els components que utilitzarem a l'aplicació.
- App.vue: És el component principal de l'aplicació. Està dividit en tres parts:
  - script: conté l'objecte Javascript amb les dades, els mètodes, ...
  - template: la plantilla que es transformarà en codi HTML.
  - style: el format CSS que s'aplica al *template*.

```
<script setup>
        import HelloWorld from './components/HelloWorld.vue'
        import TheWelcome from './components/TheWelcome.vue'
</script>

<template>
  <header>
    <img alt="Vue logo" class="logo" src="./assets/logo.svg"
                width="125" height="125" />
    <div class="wrapper">
        <HelloWorld msg="You did it!" />
    </div>
  </header>

  <main>
    <TheWelcome />
  </main>
</template>

<style scoped>
header {
  line-height: 1.5;
```

```
}

.logo {
  display: block;
  margin: 0 auto 2rem;
}
</style>
```

- main.js: Crea la instància Vue i l'assigna a l'element d'*index.html*

```
import { createApp } from "vue";
import App from "./App.vue";
import './assets/main.css'

createApp(App).mount("#app");
```

- **components**: Dins aquesta carpeta hi tendrem els distints components (els veurem més envant) amb els que montarem la nostra aplicació. Cada component té la seva plantilla, la part d'script i els estils.

```
<script setup>
defineProps({
  msg: {
    type: String,
    required: true
  }
})
</script>

<template>
  <div class="greetings">
```

```
    <h1 class="green">{{ msg }}</h1>
    <h3>
      You've successfully created a project with
      <a href="https://vitejs.dev/" target="_blank" rel="noopener">Vite</a> +
      <a href="https://vuejs.org/" target="_blank" rel="noopener">Vue 3</a>.
    </h3>
  </div>
</template>

<style scoped>
h1 {
  font-weight: 500;
  font-size: 2.6rem;
  top: -10px;
}

h3 {
  font-size: 1.2rem;
}

.greetings h1,
.greetings h3 {
  text-align: center;
}
</style>
```

- **package.json**: Les propietats del projecte i els paquets que necessita. Amb aquest fitxer *npm* pot gestionar les dependències del projecte, per exemple, descarregant els fitxers que facin falta.

Quan executam npm install per afegir un paquet al projecte s'actualitza aquest fitxer.

El fitxer main.js

En aquest fitxer cream la instancia de *Vue* passant-li com a paràmetre el component *App* declarat a *App.vue* i la montam dins l'element de *index.html* amb l'id que passam a *mount*.

Montar l'aplicació vol dir que tota la sortida que generi *Vue* anirà dins d'aquest element.

Un exemple d'aquest fitxer:

```
import { createApp } from 'vue' //La funció createApp
import App from './App.vue' //El component arrel de l'aplicació

//Crea l'aplicació. Li passam el component arrel.
const aplicacio=createApp(App)

//Montam l'aplicació a l'element d'index.html amb aquest identificador.
aplicacio.mount('#app')
```

Dins d'aquest fitxer hi poden anar altres part de la inicialització de l'aplicació Vue, com per exemple l'*store* per mantenir l'estat de l'aplicació.

Qualsevol configuració que volguem fer a l'aplicació s'ha de fer abans del *mount*.

App.vue

Conté el component arrel de l'aplicació. Té definida una part de plantilla, *template*, amb l'html del component, una altra part d'script, on es declara tot el necessari per el funcionament del component i la part d'estils aplicables a aquest component.

El component definit a aquest script és el que es passa d'argument a *createApp* al *main.js*.

```
<script setup>
import { reactive } from 'vue'

const state = reactive({ count: 0 })
function increment() {
  state.count++
}
</script>

<template>
  <button @click="increment">
   Pitjat  {{ state.count }} vegades
  </button>
</template>

<style>
        #app {
                font-family: Avenir, Helvetica, Arial, sans-serif;
                -webkit-font-smoothing: antialiased;
                -moz-osx-font-smoothing: grayscale;
                text-align: center;
                color: #2c3e50;
```

```
            margin-top: 60px;
        }
</style>
```

El fet d'afegir *setup* a l'script té conseqüències importants:

- S'executa cada vegada que es crea una instància del component.
- Tot el que es defineixi a l'script (variables, constants, funcions, ...) i tot el que s'importi (funcions, components, ...) és directament accessible des de la part de *template* del component.

Hooks del cicle de vida de la instància(només per tasques especialitzades)

Cada component de *Vue* passa per una sèrie de fases d'inicialització quan es creat. Els *lifecycle hooks* no són més que funcions que s'executaran quan el component arribi a una determinada fase.

Són mètodes que s'executen quan l'aplicació, es creada, quan es montada, ...

```
<script setup>
    import { onMounted } from 'vue'
    onUpdated( () => {
        console.log("updated")
    })
</script>
```

Aquests hooks són *beforeCreate*, *created*, *beforeMounted*, *mounted*, *beforeUpdate*, *updated*, *beforeUnmount*, *unmounted.*

S'han d'importar a l'script i es defineixen directament dins de l'script. Cada *hook* rep com a paràmetre una funció que s'executarà quan es dispari aquest hook.

Podeu trobar més informació, inclòs un diagrama de seqüència dels distints hooks a la url https://vuejs.org/guide/essentials/lifecycle.html#lifecycle-diagram

Sintaxi de *template*

Les plantilles utilitzen sintaxi HTML. Qualsevol plantilla Vue pot ser interpretada per qualsevol navegador o parser HTML.

Vue les converteix en funcions Javascript altament optimitzades que interactuen amb el DOM. Juntament amb el sistema de reactivitat Vue pot calcular els mínims canvis a fer al DOM per actualitzar la pàgina quan l'estat de l'aplicació canvia.

Mustaches {{ }}

Interpolació de text. Aquesta notació permete incloure text dinàmicament dins la plantilla. Normalment dins {{ }} hi posarem una de les propietats definides al *setup*. Per exemple,

```
<span>Message: {{ msg }}</span>
```

substituirà {{ msg }} per el valor de la propietat msg definit dins l'*script*. A més, si el valor d'aquesta propietat canvia s'actualitzarà la part corresponent de la pàgina per mostrar el canvi.

El text és escapat, és a dir, si msg inclou elements html en pantalla es veuran aquests elements, no s'interpretaran.

Si tenim

```
msg = "<strong>Has pitjat el botó</strong>";
```

veurem a la pàgina

Has pitjat el botó

Si volem que s'interpreti l'html haurem d'utilitzar una directiva, v-html

```
<p v-html="msg"></p>
```

I veurem a la pàgina

Has pitjat el botó

Els *mustaches* no es poden utilitzar per donar valor als atributs.

Dins els *mustaches* hi podem posar una expressió Javascript, és a dir, codi Javascript que torna un valor.

```
<span>Següent valor: {{ numero + 1 }} </span>
```

o

```
<span>Següent valor: {{ calculaSeguent() }} </span>
```

Directives v-

Les directives són atributs especials amb el prefixe *v-* Els seus valors són una expressió Javascript. La seva tasca és aplicar canvis al DOM en reacció a canvis a la seva expressió.

```
<p v-if="altern">Només es veu quan <em>altern=true</em></p>
```

Segons el valor de l'expressió el paràgraf apareixerà o no a la pàgina.

Algunes de les directives són:

- **v-bind**: S'utilitza per assignar valors als atributs dels elements. L'acompanya un argument per indicar quin atribut es vol modificar.

  ```
  <button v-bind:disabled="altern">Botó inútil</button>
  <div v-bind:id="dynamicId"></div>
  ```

  Es pot abreujar amb :

  ```
  <button :disabled="isButtonDisabled">Button</button>
  <div :id="dynamicId"></div>
  ```

  L'atribut tendrà el valor de l'expressió. En el cas dels atributs booleans, si l'expressió és false l'atribut no s'inclourà.

  Pot incloure expressions javascript. Per exemple,

  ```
  <div :id="`list-${id}`"></div>
  ```

- **v-on**: permet assignar listeners als esdeveniments dels elements. Tenen com a argument l'esdeveniment. El seu valor pot ser una crida a una funció o el nom de la funció.

```
<a v-on:click="doSomething"> ... </a>
```

  Es pot abreujar amb @:

```
<a @click="doSomething"> ... </a>
```

- **v-model**: Permet enllaçar propietats amb elements d'un formulari, de forma bidireccional, de manera que si es modifica la propietat es modifica el contingut de l'element i si es modifica l'element es modifica la propietat.

  Mes endavant veurem els formularis amb detall.

- **v-if**: inclou l'element o no segons el valor de l'expressió associada a l'*if*. També tenim l'*else*.

```
<p v-if="altern">Només es veu quan <em>altern=true</em></p>
<p v-else>Només es veu quan <em>altern=false</em></p>
```

- **v-for**: Crea tants elements com ítems contengui la llista associada. Cada element creat amb la iteració ha de tenir un identificador assignat amb *v-bind:key="id"*.

  Per exemple si *a l'script* tenim definit un array:

```
llistaNoms:["Pere","Joan","Maria","Margalida"],
```

  Els podem incloure a la pàgina amb el *v-for*

```
<p v-for="nom in llistaNoms" v-bind:key="nom">{{nom}}</p>
```

Fonaments de la reactivitat

Podem crear un objecte o array reactiu utilitzant la funció *reactive()*. Aquesta funció torna un *proxie*, un objecte Javascript amb la particularitat que *Vue* és capaç de seguir els accessos i les mutacions d'aquest objecte.

```
<script setup>
import { reactive } from 'vue'

const state = reactive({ count: 0 })

function increment() {
  state.count++
}
</script>

<template>
  <button @click="increment">
    {{ state.count }}
  </button>
</template>
```

La funció *reactive()* té dues limitacions importants:

1. Només es pot utilitzar amb objectes i amb arrays, no amb tipus primitius com number, string, boolean, ...
2. Perquè funcioni el sistema de reactivitat, sempre hem de mantenir la mateixa referència a l'objecte reactiu: Si l'assignam a una nova variable, el passam com a paràmetre a una funció, ... es perd la reactivitat.

Per evitar aquestes limitacions *Vue* disposa de la funció *ref()*. Funciona amb qualsevol tipus de dades. Torna l'argument que rep dins un objecte amb la propietat *value*.

```
const count = ref(0)

console.log(count) // { value: 0 }
console.log(count.value) // 0

count.value++
console.log(count.value) // 1
```

A la plantilla no fa falta utilitzar el .value:

```
<p>{{ count }}</p>
```

Propietats computades

Hem vist que podem utilitzar expressions Javascript directament, però això té uns quants contres, per exemple no podem posar blocs de codi més enllà d'una expressió, no podem reutilitzar l'expressió, el codi es complica, ...

Les propietats computades són una alternativa. És defineixen dins l'script utilitzant la funció *computed* que hem d'importar de 'vue'..

```
const state = reactive({cadena:'Hola Vue!'});

const cadenaGirada = computed(() => {
        return state.cadena.split("").reverse().join("");
})
```

Si utilitzam aquesta propietat computada tendrem el contingut de *cadena* al revés.

```
<p>Missatge invertit: "{{ cadenaGirada }}"</p>
```

En el nostre exemple mostraria !euV olaH. A l'aplicació ara tenim la propietat *cadena* i la propietat computada *cadenaGirada*.

La propietat computada pertany a l'entorn de reactivitat: En aquest cas, quan canviï el valor de *cadena* canviarà també el de *cadenaGirada* i s'actualitzarà la pàgina.

Podríem haver creat un mètode en lloc d'una propietat commutada. Mostraria el mateix.

```
function missatgeGirat(){
        return state.cadena.split("").reverse().join("");
}
```

Però hi ha una **diferència** important. Una propietat computada només s'avaluarà, s'executarà la funció per calcular el seu valor, quan les seves dependències hagin canviat. En el nostre cas, mentre *cadena* no es modifiqui, es mantendrà en caché el valor de *missatgeGirat*.

En canvi, si ho declaram com un mètode, cada vegada que s'ha de tornar a generar el DOM s'executarà la funció.

Renderització condicional

Com ja hem vist, amb la directiva *v-if* podem incloure l'element que la conté o no segons el valor de l'expressió associada a l'if. També tenim l'else.

```
<h1 v-if="ok">Sí</h1>
<h1 v-else>No</h1>
```

La directiva *v-else* ha de seguir una directiva *v-if* per tenir efecte.

Si el que volem mostrar o amagar amb if és més d'un element podem utilitzar l'element <*template*> per contenir tot el que volem que afecti l'if. En cap cas es generarà un element HTML <template>.

```
<template v-if="ok">
        <h1>Título</h1>
        <p>Párrafo 1</p>
        <p>Párrafo 2</p>
</template>
```

També tenim la directiva *v-else-if* per encadenar condicions. Sempre ha d'anar darrera un *v-if* o un altre *v-else-if.*

```
<div v-if="type === 'A'">
                Tipus A
</div>
<div v-else-if="type === 'B'">
                Tipus B
</div>
<div v-else-if="type === 'C'">
                tipus C
</div>
<div v-else>
                Si no es A, B o C
</div>
```

La directiva *v-show* també permet mostrar un element o no depenent d'una condició. La diferència és que mentre que *v-if* crea o no l'element, *v-show* només canvia el valor de l'atribut *display* de l'element.

v-show no funciona amb <*template*> ni té res similar a l'else.

***v-show* té un cost d'alternar entre true i false molt més baix que *v-if*.** Si estam davant una condició que canviarà molt sovint millor utilitzam *v-show* que *v-if*. Com a contrapartida, sempre es genera l'element, encara que inicialment la condició sigui falsa; en canvi, v-if no el genera fins que la condició sigui certa.

Renderització iterativa

Quan volem mostrar una llista, un array, repetint elements HTML per a cada element de la llista, podem utilitzar *v-for*.

```
<script setup>
	const altern = ref(true)
	const llista = reactive([{id: 1, nom: "Joan"}, {id: 2, nom: "Margalida"}])
</script>
<template>
	<ul>
		<li v-for="persona in llista" v-bind:key="persona.id">
			{{ persona.nom }}
		</li>
	</ul>
</template>
```

En aquest cas es generarà un *<li>* per a cada element de *items*. Si ens fa falta, també podem accedir a l'index de cada element:

```
<ul>
    <li v-for="(persona, index) in llista" v-bind:key="persona.id">
		{{ persona.nom }} ( {{ index }} )
    </li>
</ul>
```

També podem utilitzar *v-for* per iterar a través de les propietats d'un objecte.

```
<ul>
	<li v-for="valor in persona">    {{ valor }}  </li>
</ul>
```

A l'script

```
const persona: reactive({
                    nom: 'Jo',
                    mateix: 'Mateix',
                    edad: 15
                })
}
```

I si volem accedir als noms de les propietats de l'objecte:

```
<div v-for="(value, key) in persona">
        {{ key }}: {{ value }}
</div>
```

Si volem que el *v-for* afecti a diversos elements podem utilitzar <template> com feiem amb *v-if*.

Finalment, podem utilitzar un rang en el lloc d'una llista.

```
<span v-for="n in 10">{{ n }}</span>
```

Mostrarà els sencers d'1 a 10.

Esdeveniments

Amb la directiva *v-on*, abreujada com a @, podem assignar lògica als esdeveniments del DOM.

```
<script setup>
		pitjar(event) {
			this.altern = !this.altern;
			if (this.altern) {
				this.missatge = "Hola, Vue!";
			} else {
				this.missatge = "<strong>Has pitjat el botó</strong>";
			}
		}
</script>
...
<template>
		<button type="button" @click="pitjar()">Alternar missatge</button>
</template>
```

Encara que no sigui recomanable també es pot posar codi Javascript directament dins la directva *v-on*:

```
<button v-on:click="counter += 1">Add 1</button>
```

Modificadors

Moltes vegades dins el codi dels mètodes que responen a un esdeveniment hem de cridar els mètode preventDefault, stopPropagation, ... Vue ens dona la manera de indicar aquestes accions sense haver-les de programar als listeners dels esdeveniment. Són els modificadors

```
<form v-on:submit.prevent="onSubmit"></form>
```

En aquest cas s'executarà el mètode *preventDefault* a l'inici de la funció onSubmit. Els modificador són:

- .**stop**: stopPropagation()
- .**prevent**: preventDefault()
- .**capture**: tercer paràmetre de addEventListener(). Utilitza *tickling* i no *bubbling.*
- .**self**: Només s'executa si les propietats *target* i *currentTarget* de l'element tenen el mateix valor.
- .**once**: El listener només s'executa la primera vegada que es dispara l'esdeveniment.

Modificadors per esdeveniments de teclat

Podem especificar quina tecla volem que dispari l'esdeveniment amb els modificadors *enter*, *tab*, *delete*, *esc*, *space*, *up*, *down*, *left* i *right.*

Per exemple, si volem que un esdeveniment es dispari només en pitjar *Enter*:

```
<input @keyup.enter="submit" />
```

Formularis

Per enllaçar elements d'un formulari amb les dades de *Vue* utilitzam *v-model*. L'enllaç es bidireccional, és a dir, si modificam la propietat s'actualitza l'element del formulari i si modificam el contingut de l'element del formulari es modifica la propietat.

Es pot utilitzar en elements input, select i textarea. Vue ignorarà els valors dels atributs value, checked o selected establerts a l'element, al seu lloc utilitzarà el proporcionat per v-model.

Input

En aquest exemple enllaçam un input amb la propietat *missatge*.

```
<form>
   <p><label>Missatge: </label><input type="text" v-model="missatge"></p>
</form>
```

En modificar el camp de text s'actualitza immediatament el valor de la dada *missatge* i com que forma part del sistema de reactivitat, s'actualitzarà per tot allà on s'utilitzi, incloses les propietats computades.

I al revés, si de qualque manera es modifica el valor de *missatge* s'actualitzarà el valor de l'*input*.

textarea

Amb un textarea seria d'aquesta forma:

```
<textarea v-model="missatge"></textarea>
```

select

El valor de la propietat ha de coincidir amb el del *value* de l'element *option* seleccionat.

```
<script setup>
import { ref } from 'vue'

const triat = ref('')
</script>
<template>
	<select v-model="triat">
		<option disabled value="">Tria un element</option>
		<option value="1">Joan</option>
		<option value="2">Margalida</option>
		<option value="3">Pere</option>
	</select>
	<p>Seleccionat: {{ triat }}</p>
</template>
```

checkbox

Per propietats booleanes. Podem associar un sol checkbox a una propietat simple:

```
<p><input type="checkbox" id="checkbox" v-model="altern"></p>
```

O podem associar múltiples checkbox a un array, v-model fa referència a un array:

```
<script setup>
import { ref } from 'vue'

const llista=ref([{id:1,nom:"Jo Mateix"},{id:2,nom:"Tu També"},{id:3,nom:"Un Altre"}]);
```

```
const seleccionats = ref([])
</script>

<template>
    <p v-for="persona in llista" v-bind:key="persona.id">
            <label>{{persona.nom }}</label><input type="checkbox"
                    v-model="seleccionats" v-bind:value="persona.id">
        </p>
        <p>{{seleccionats}}</p>
</template>
```

radio buttons

v-model s'associa a una propietat simple. Es marca el radio tal que el seu valor coincideix amb *v-model.*

```
<script setup>
    import { ref } from 'vue'

    const                    llista=ref([{id:1,nom:"Mallorca"},{id:2,nom:"Menorca"},
{id:3,nom:"Eivissa"},
            {id:4,nom:"Formentera"}])
    const seleccionat = ref(0)
</script>

<template>
    <p v-for="illa in llista" v-bind:key="illa.id">
            <label>{{illa.nom }}</label>
            <input type="radio"  v-model="seleccionat" :value="illa.id">
    </p>
    <p>{{seleccionat}}</p>
```

```
</template>
```

Modificadors

v-model també permet modificadors. Són els següents:

- **.lazy**: Per defecte cada vegada que l'usuari modifica el contingut de l'input es sincronitza el seu valor amb la propietat. Posant el modificador lazy la sincronització te lloc en disparar-se l'esdeveniment *change*.

```
<h4>Modificadors</h4>
<p>
        <label>Missatge lazy: </label>
        <input type="text" v-model.lazy="missatge">
</p>
<p>    {{ missatge}}  </p>
```

- **.number**: Els inputs sempre es tracten com a string. Amb aquest modificador l'entrada es tracta com a numèrica. Si *parseFloat()* aplicat al contingut de l'element pot tornar un valor numèric torna aquest valor numèric, sinó torna una cadena de text.

```
<script setup>
        function mostraTipus(){    alert(typeof(quantitat.value)); }
</script>
<template>
<h4>Modificadors</h4>
<p>
   <label>Quantitat: </label>
   <input type="number"  v-model="quantitat" @change="mostraTipus">
</p>
<p>    {{ quantitat}}  </p>
```

```
</template>
```

- **.trim**: Retalla automàticament les entrades de l'usuari, es a dir, elimina els espais en blanc a l'inici i al final de l'entrada:

```
<input v-model.trim="msg">
```

Els modificadors es poden concatenar:

```
<h4>Modificadors</h4>
<p><label>Missatge lazy: </label>
        <input type="text" v-model.lazy.trim="missatge"></p>
```