

Vue

Tema 2

Componentes

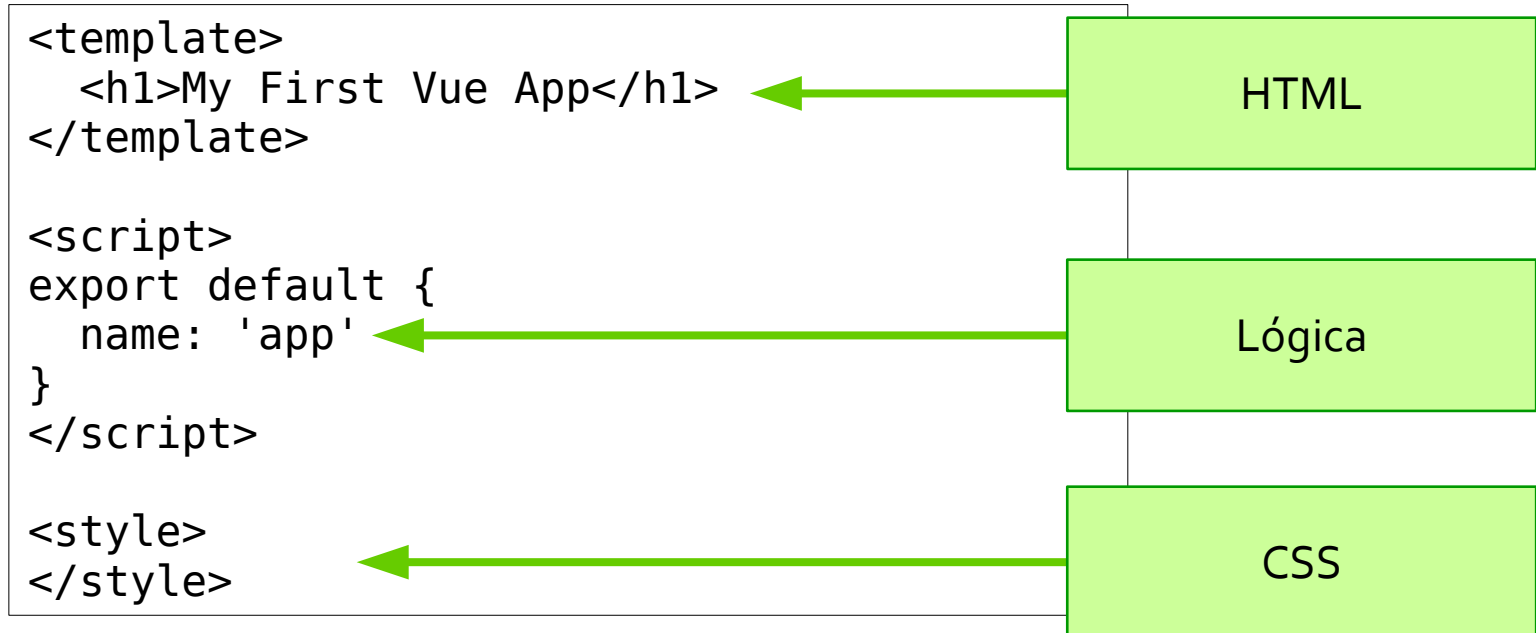
Componentes

Componentes en Vue

- Un componente una **nueva etiqueta HTML** con una **vista** y una **lógica** definidas por el desarrollador
- Fichero `.vue` con 3 secciones:
 - La **vista** es una plantilla (*template*) en HTML con elementos especiales y un CSS
 - La **lógica** es un código JavaScript vinculado a la vista

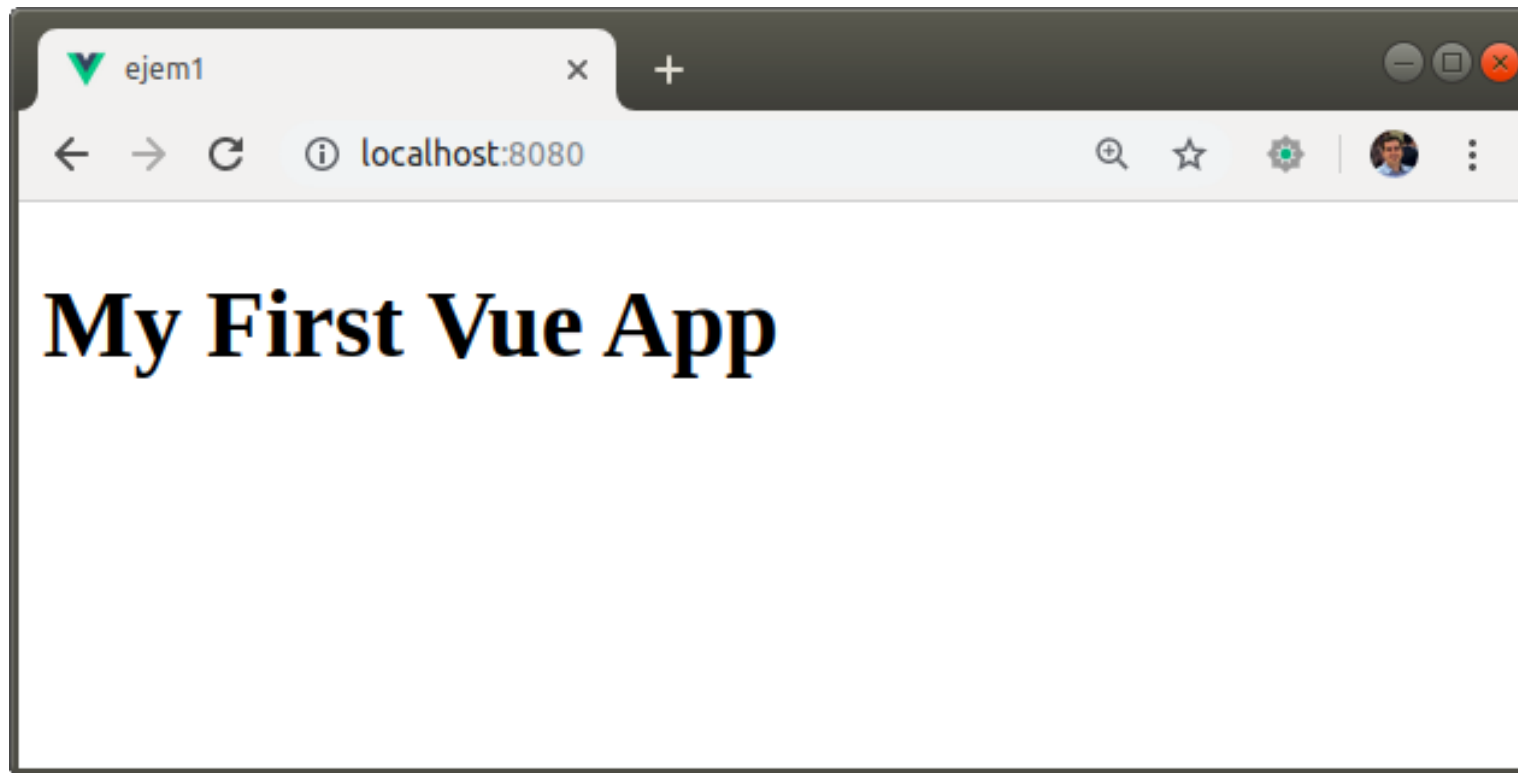
Componentes en Vue

App.vue



Componentes

ejem1



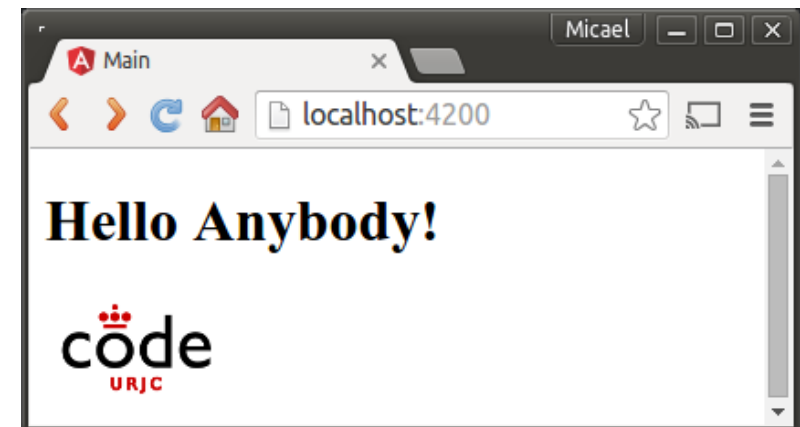
Visualización de una variable

App.vue

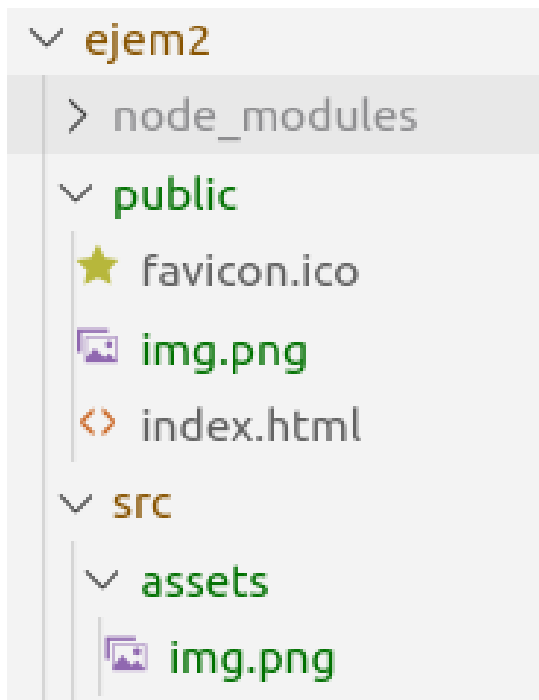
```
<template>
  <div>
    <h1>Hello {{name}}!</h1>
    
  </div>
</template>

<script>
export default {
  name: 'app',
  data: () => {
    return {
      name: 'Anybody',
      imgUrl: 'img.png'
    }
  }
}
</script>
```

La vista del componente (HTML) se genera en función de sus datos (campo **data**)



Recursos de la app



- Los recursos (imágenes, fonts..) pueden colocarse en dos sitios:
 - **public:** Serán copiados a la raíz de la carpeta de producción. No se procesan (optimización, inline...)
 - **src/assets:** Se procesan (optimización, inline...) si se encuentran referenciados en los templates

Ejecución de lógica

App.vue

```
<template>
  <div>
    <h1>Hello {{name}}!</h1>
    <button v-on:click="setName('John')">Hello John</button>
  </div>
</template>

<script>
export default {
  name: "app",
  data: () => {
    return {
      name: "Anybody"
    }
  },
  methods: {
    setName: function(name) {
      this.name = name;
    }
  }
};
</script>
```

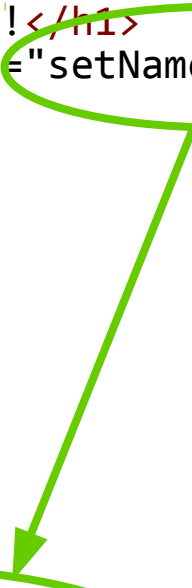
Se puede ejecutar un método ante un evento producido en la vista del componente

Ejecución de lógica

App.vue

```
<template>
  <div>
    <h1>Hello {{name}}!</h1>
    <button v-on:click="setName('John')">Hello John</button>
  </div>
</template>

<script>
export default {
  name: "app",
  data: () => {
    return {
      name: "Anybody"
    }
  },
  methods: {
    setName: function(name){
      this.name = name;
    }
  }
};
</script>
```



Se puede ejecutar un método ante un evento producido en la vista del componente

Ejecución de lógica

App.vue

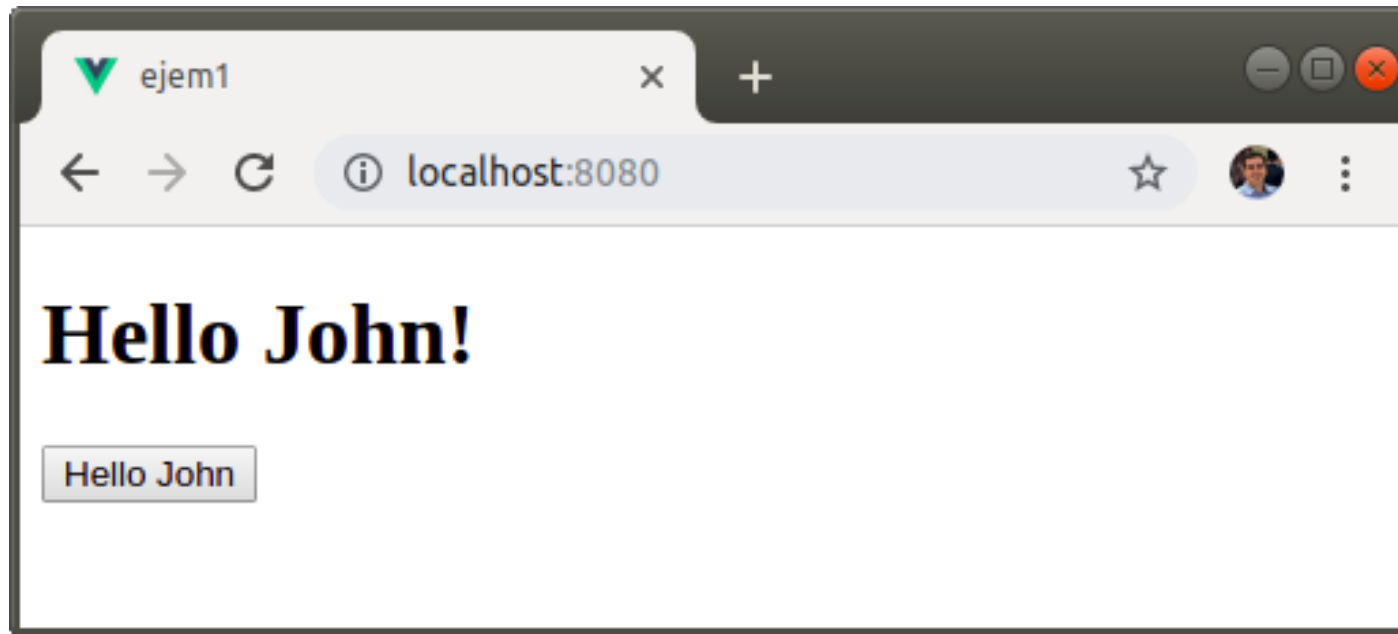
```
<template>
  <div>
    <h1>Hello {{name}}!</h1>
    <button v-on:click="setName('John')">Hello John</button>
  </div>
</template>
```

```
<script>
export default {
  name: "app",
  data: () => {
    return {
      name: "Anybody"
    }
  },
  methods: {
    setName: function(name){
      this.name = name;
    }
  }
};
</script>
```

Se puede definir cualquier **evento** disponible en el **DOM** para ese elemento

Ejecución de lógica

Se puede ejecutar un método ante un evento producido en la vista del componente



Sintaxis de los templates

Versión compacta (*shorthand*)

```

```



```

```

```
<button v-on:click="setName('John')">Hello John</button>
```



```
<button @click="setName('John')">Hello John</button>
```

Sintaxis de los templates

```
<h1>Hello {{name}}!</h1>  
  
<button @click="setName('John')">  
    Hello John  
</button>
```

{{ data }}

Valor de una propiedad de **data** del componente en el texto

:prop="data"

Valor de una propiedad de **data** como valor de una propiedad del elemento HTML

@event="method()"

Se llama al método cuando se produce el evento

Datos enlazados (*data binding*)

Un campo de texto se puede “enlazar” a un atributo
Atributo y campo de texto están sincronizados

```
<template>
  <div>
    <input type="text" v-model="name">

    <h1>Hello {{name}}!</h1>

    <button @click="setName('John')">
      Hello John
    </button>
  </div>
</template>
```

```
<script>
export default {
  name: "app",
  data: () => {
    return {
      name: "Anybody"
    }
  },
  methods: {
    setName: function(name){
      this.name = name;
    }
  }
};
</script>
```

Datos enlazados (*data binding*)

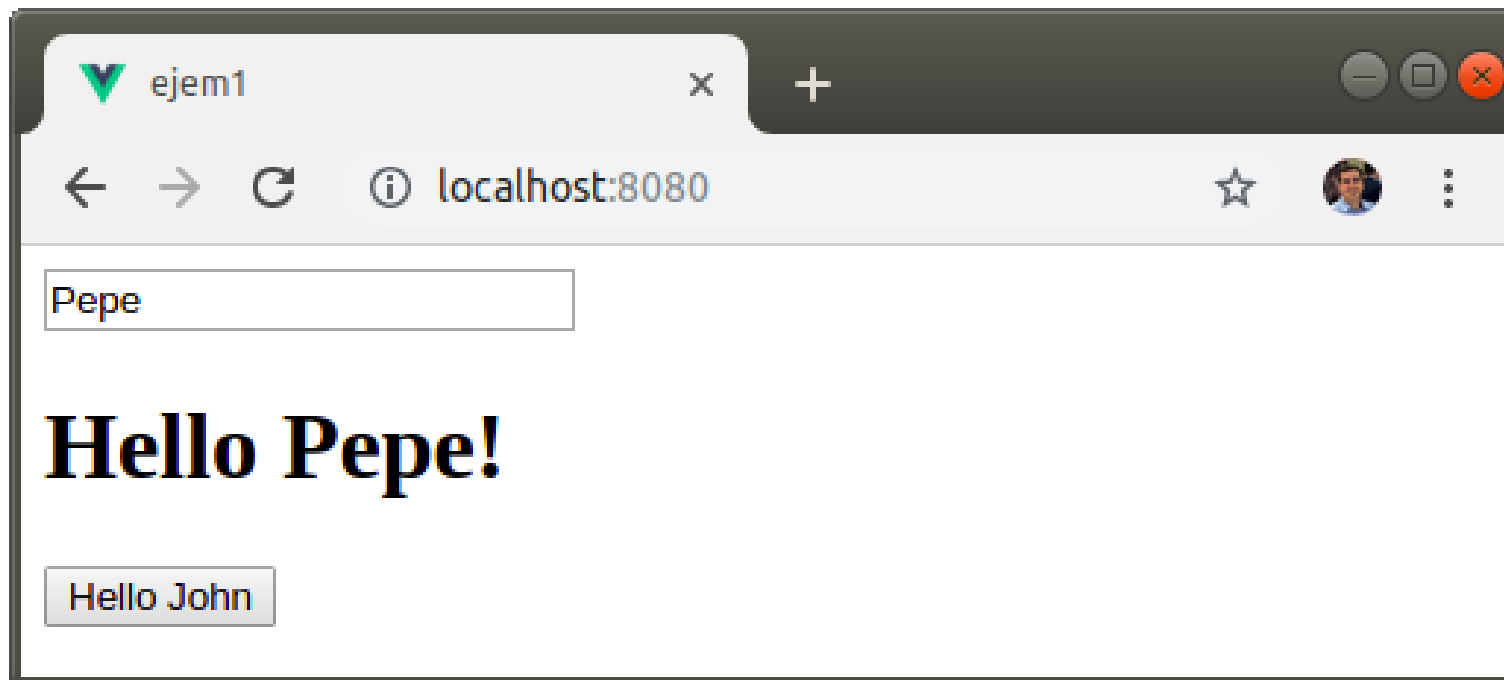
Un campo de texto se puede “enlazar” a un atributo
Atributo y campo de texto están sincronizados

```
<template>
  <div>
    <input type="text" v-model="name">
    <h1>Hello {{name}}!</h1>
    <button @click="setName('John')">
      Hello John
    </button>
  </div>
</template>
```

```
<script>
export default {
  name: "app",
  data: () => {
    return {
      name: "Anybody"
    }
  },
  methods: {
    setName: function(name){
      this.name = name;
    }
  }
};
</script>
```

Datos enlazados (*data binding*)

Un campo de texto se puede “enlazar” a un atributo
Atributo y componente están sincronizados



Propiedades computadas

Cuando una propiedad está derivada de otra, se define como propiedad computada y sólo se actualiza cuando se actualiza la original

```
<script>
export default {
  name: "app",
  data: () => {
    return {
      name: "Anybody"
    }
  },
  computed: {
    upperName: function(){
      return this.name.toUpperCase();
    }
  },
  methods: {
    setName: function(name){
      this.name = name;
    }
  }
};
</script>
```

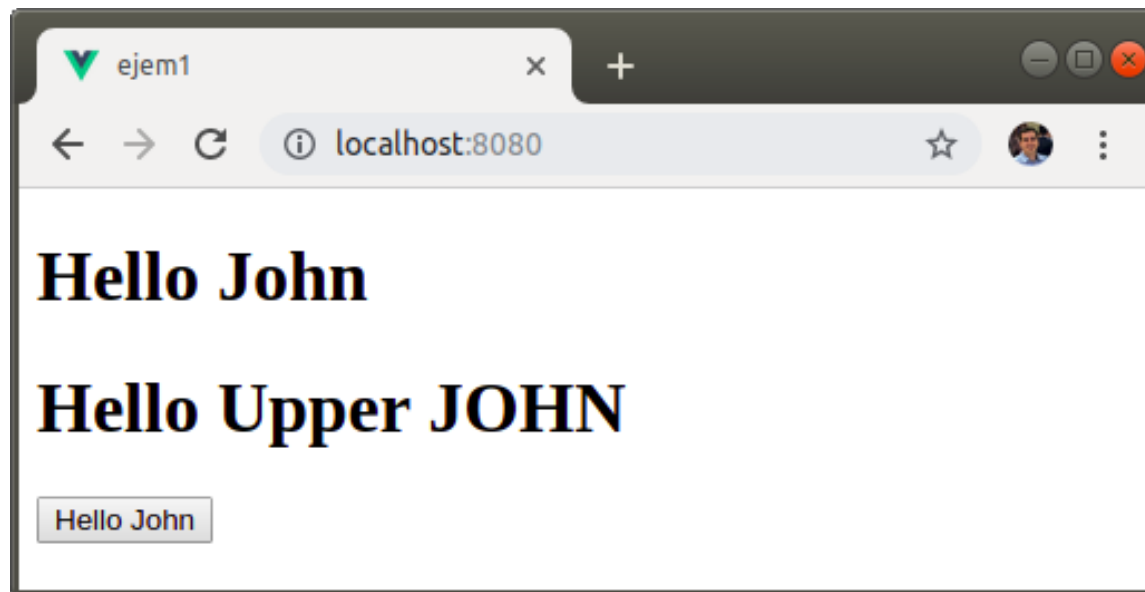
```
<template>
  <div>

    <h1>Hello {{name}}</h1>
    <h1>Hello Upper {{upperName}}</h1>

    <button @click="setName('John')">
      Hello John
    </button>
  </div>
</template>
```

Propiedades computadas

Cuando una propiedad está derivada de otra, se define como propiedad computada y sólo se actualiza cuando se actualiza la original



Templates

- Los **templates** permiten definir la vista en función de la información del componente
 - HTML en bruto
 - Visualización condicional
 - Repetición de elementos

- **HTML en bruto**
 - Usando `{{ }}` el contenido HTML se escapa
 - Para mostrar HTML en la página, se usar la directiva `v-html`

```
<p>Using mustaches: {{ rawHtml }}</p>  
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

Using mustaches: `This should be red.`

Using v-html directive: **This should be red.**

- **Visualización condicional**

- Se puede controlar si un elemento aparece o no en la página dependiendo del valor de un atributo de la clase usando la **directiva ngIf**
- Por ejemplo dependiendo del **valor del atributo** booleano visible

```
<p v-if="visible">Text</p>
```

- También se puede usar una **expresión**

```
<p v-if="num == 3">Num 3</p>
```

- **Visualización condicional**
 - Se puede mostrar un contenido si la expresión no se cumple (**else**) y si se cumple otra (**else-if**)

```
<div v-if="isValid">valid content</div>  
<div v-else>invalid content</div>
```

```
<div v-if="type === 'A'">A</div>  
<div v-else-if="type === 'B'">B</div>  
<div v-else-if="type === 'C'">C</div>  
<div v-else>Not A/B/C</div>
```

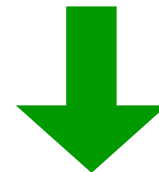
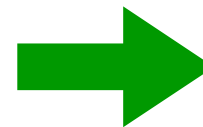
- **Visualización condicional**
 - Se pueden mostrar varios elementos si se cumple una expresión (el template no aparece en el dom)

```
<template v-if="ok">  
  <h1>Title</h1>  
  <p>Paragraph 1</p>  
  <p>Paragraph 2</p>  
</template>
```


- Repetición de elementos
 - Es posible visualizar el contenido de un array con la **directiva v-for**
 - Se define cómo se visualizará cada elemento

```
<div v-for="elem in elems" :key="elem.id">{{elem.desc}}</div>
```

```
elems: [  
  { id:0, desc: 'Elem1', checked: true },  
  { id:1, desc: 'Elem2', checked: true },  
  { id:2, desc: 'Elem3', checked: false }  
]
```



```
<div>Elem1</div>  
<div>Elem2</div>  
<div>Elem3</div>
```

- **Repetición de elementos**

- Para que un cambio en el array se actualice en el DOM de forma eficiente, cada objeto debe tener un identificador
- Ese identificador se define con la propiedad sintética **key**

```
<div v-for="elem in elems" :key="elem.id">{{elem.desc}}</div>
```

- Aspectos a tener en cuenta al cambiar un array visualizado con **v-for**

<https://vuejs.org/v2/guide/list.html#Array-Change-Detection>

- **v-if con v-for**

- No se debe usar un v-if en el mismo elemento que v-for
- Para ello se usa un **propiedad computada** que filtre el array

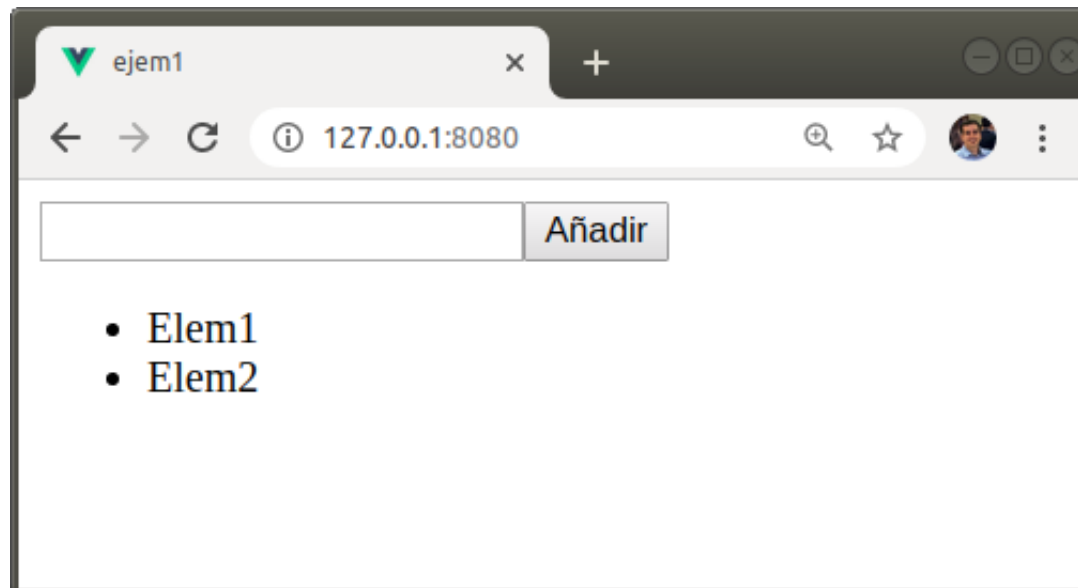
```
filteredElems: function(){  
    return this.elems.filter(elem => elem.checked);  
}
```

- Si se quiere mostrar condicionalmente el array, el v-if debe estar en un elemento padre

```
<ul v-if="elems.length">  
    <li v-for="elem in elems" :key="elem.id">{{elem.desc}}</li>  
</ul>  
<p v-else>No elems!</p>
```

Ejercicio 1

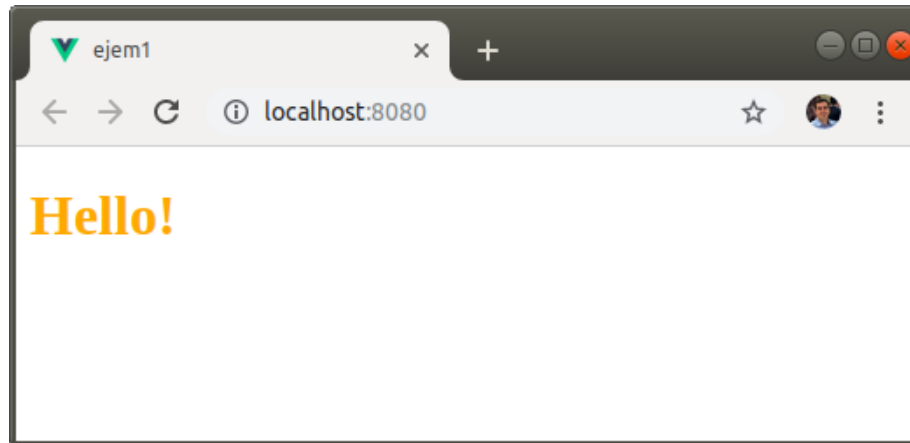
- Implementa una aplicación con un **campo de texto** y un **botón de Añadir**
- Cada vez que se pulse el **botón**, el **contenido** del campo de texto se **añadirá al documento**



Estilos CSS

- Existen varias formas de definir un CSS en Vue ^{ejem7}
 - Globalmente asociado al index.html
 - Local al componente en la sección <style> del fichero .vue

- Definir CSS en Vue localmente
 - En el fichero .vue



```
<template>
  <div>
    <h1>Hello!</h1>
  </div>
</template>

<script>
export default {
  name: "app"
}
</script>

<style scoped>
h1 {
  color: orange;
}
</style>
```

- **Controlar la clase o estilo de un elemento**
 - Hay muchas formas de controlar los estilos de los elementos
 - Asociar la **clase** de un elemento a un atributo de tipo string
 - Activar una **clase concreta** con un atributo boolean
 - Asociar la **clase** de un elemento a un atributo de tipo mapa de string a boolean
 - Asociar un **estilo concreto** de un elemento a un atributo

<https://flaviocopes.com/vue-css/>

- Asociar la clase de un elemento a un atributo string
 - Cambiando el valor del atributo se cambia la clase del elemento
 - Por ejemplo, la clase del elemento h1 se cambia modificando la propiedad **titleClass**

```
<h1 :class="titleClass">Hello!</h1>
```

- Asociar la clase de un elemento a un atributo string

```
<template>
  <div>
    <h1 :class="titleClass">Hello!</h1>
    <button @click="toRed()">To Red</button>
  </div>
</template>
```

```
<style scoped>
  .orange {
    color: orange;
  }
  .red {
    color: red;
  }
</style>
```

```
<script>
export default {
  name: "app",
  data: function(){
    return {
      titleClass: "orange"
    }
  },
  methods: {
    toRed: function(){
      this.titleClass = "red"
    }
  }
}
</script>
```

- Activar una clase concreta con un atributo boolean
- Activa o desactiva una clase red con el valor del atributo booleano **isRed**

```
<h1 :class="{ red: isRed }">Hello!</h1>
```

- Se puede usar para varias clases

```
<h1 :class="{ red: isRed, underline: isUnderlined }">Hello!</h1>
```

- Activar una clase concreta con un atributo boolean

```
<template>
  <div>
    <h1 :class="{ red: isRed }">Hello!</h1>

    <h1 :class="{ red: isRed, underline: isUnderlined }">
      Hello!
    </h1>

    <button @click="toRed()">To Red</button>
  </div>
</template>
```

```
<style scoped>
.red {
  color: red;
}
.underline {
  text-decoration: underline;
}
</style>
```

```
<script>
export default {
  name: "app",
  data: function(){
    return {
      isRed: false,
      isUnderlined: false
    }
  },
  methods: {
    toRed: function(){
      this.isRed = true;
      this.isUnderlined = true;
    }
  }
}
</script>
```

- Asociar la clase de un elemento a un mapa
 - Para gestionar varias clases es mejor usar un objeto con propiedades como nombre de la clase a boolean

```
<h1 :class="titleStyle">Hello!</h1>
```

```
data: function(){  
  return {  
    titleStyle: {  
      red: false,  
      underline: false  
    }  
  }  
}
```

```
methods: {  
  toRed: function(){  
    this.titleStyle.red = true;  
    this.titleStyle.underline = true;  
  }  
}
```

- Asociar la clase de un elemento a un array
- Se puede usar un array de propiedades con el valor de las propiedades como clases CSS

```
<h1 :class="[color, decoration]">Hello!</h1>
```

```
data: function(){  
  return {  
    color: "orange",  
    decoration: "italic"  
  }  
}
```

```
methods: {  
  toRed: function(){  
    this.color = "red",  
    this.decoration = "underline"  
  }  
}
```

- **Asociar un estilo concreto a un atributo**
- En algunos casos es mejor cambiar el estilo directamente en el elemento con un objeto

```
<h1 :style="titleStyle">Hello!</h1>
```

```
data: function(){  
  return {  
    titleStyle: {  
      color: "orange",  
      "text-decoration": undefined,  
      fontStyle: "italic"  
    }  
  }  
}
```

```
methods: {  
  toRed: function(){  
    this.titleStyle.color = "red",  
    this.titleStyle["text-decoration"]="underline",  
    this.titleStyle.fontStyle = undefined;  
  }  
}
```

- **Asociar un estilo concreto a un atributo**
 - O con la propiedad directamente en el template

```
<h1 :style="{ color: color }">Hello!</h1>
```

```
data: function(){  
  return {  
    color: "orange"  
  }  
}
```

```
methods: {  
  toRed: function(){  
    this.color = "red"  
  }  
}
```


Formularios

- Existen diversas formas de controlar formularios en Vue
 - Vincular un control del formulario a un atributo del componente (***data binding***)
 - Otros mecanismos **avanzados** para generación dinámica de formularios (no los veremos)

- *Data binding* en campo de texto
 - Se vincula el control a un atributo del componente con la directiva **v-model**
 - Cualquier cambio en el control se refleja en el valor del atributo (y viceversa)

```
<input type="text" v-model="name">  
<textarea v-model="name"></textarea>  
  
<p>{{name}}</p>
```

```
name: ""
```



The diagram illustrates data binding in Vue.js. It shows two boxes. The top box contains three lines of HTML code: `<input type="text" v-model="name">`, `<textarea v-model="name"></textarea>`, and `<p>{{name}}</p>`. The bottom box contains a JavaScript object definition: `name: ""`. Green circles highlight the `name` attribute in the `v-model` directives and the `name` property in the object. Green arrows point from the `name` property in the object to each of the `name` attributes in the HTML code, indicating the bidirectional data binding.

- ***Data binding* en checkbox (boolean)**
 - Cada control se asocia con v-model a un atributo booleano y su valor depende de si está “checked”

```
<input type="checkbox" v-model="vue"/> Vue  
<input type="checkbox" v-model="javascript"/> JS
```

```
vue: true,  
javascript: false
```

- *Data binding* en checkbox (objetos)
- Cada control se asocia con **v-model** a un atributo booleano de un objeto de un array

```
<span v-for="item in items" :key="item.id">  
  <input type="checkbox" v-model="item.selected" />  
  {{item.value}}  
</span>
```

```
items: [  
  { id:0, value: "Item1", selected: true },  
  { id:1, value: "Item2", selected: false }  
]
```

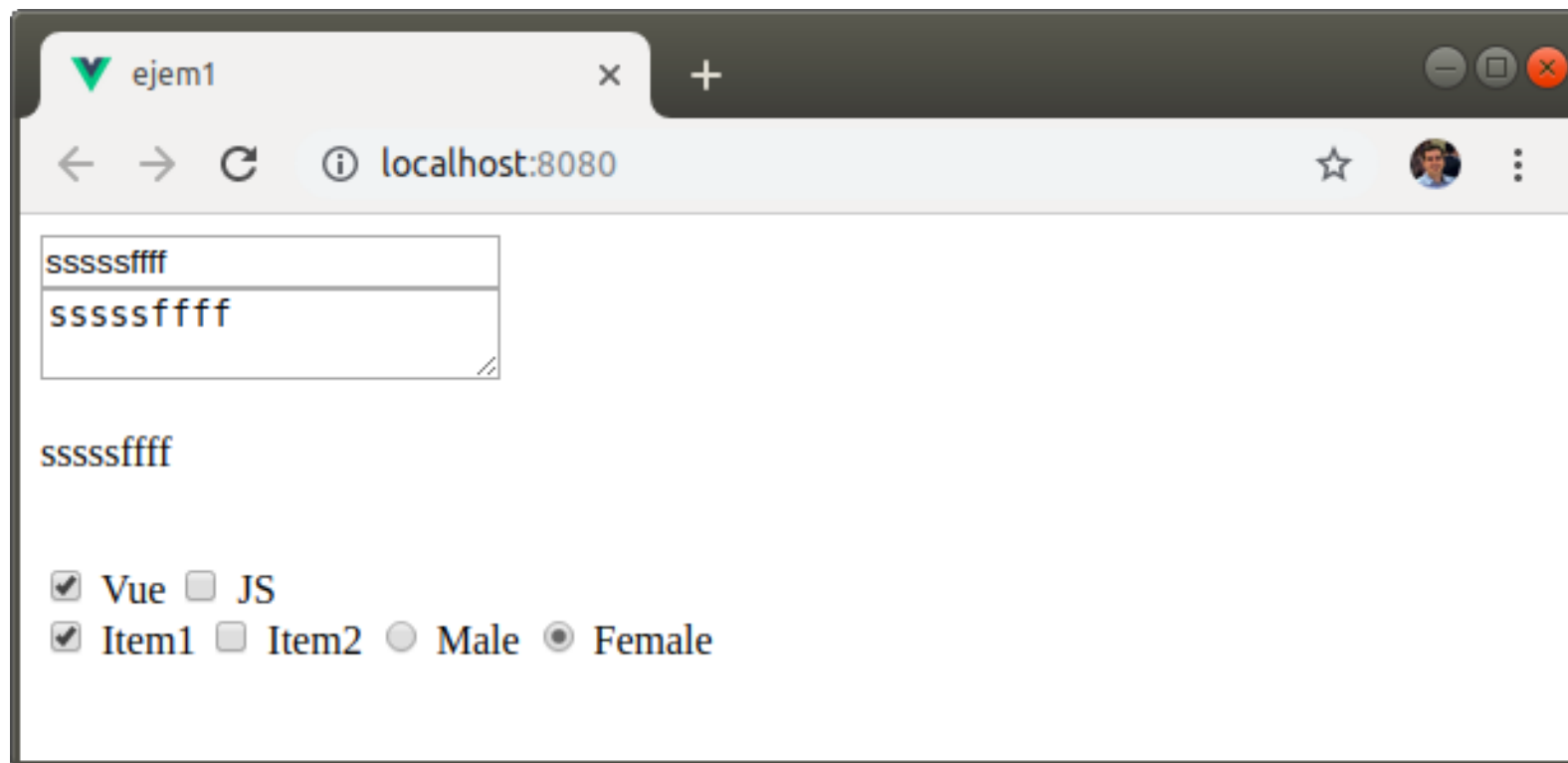
- *Data binding* en botones de radio
 - Todos los botones del mismo grupo se asocian al mismo atributo con `[(ngModel)]`
 - El valor del atributo es el "value" del control

```
<input type="radio" name="gender"  
  v-model="gender" value="Male"> Male  
<input type="radio" name="gender"  
  v-model="gender" value="Female"> Female
```

gender: "Female"

Formularios

ejem14

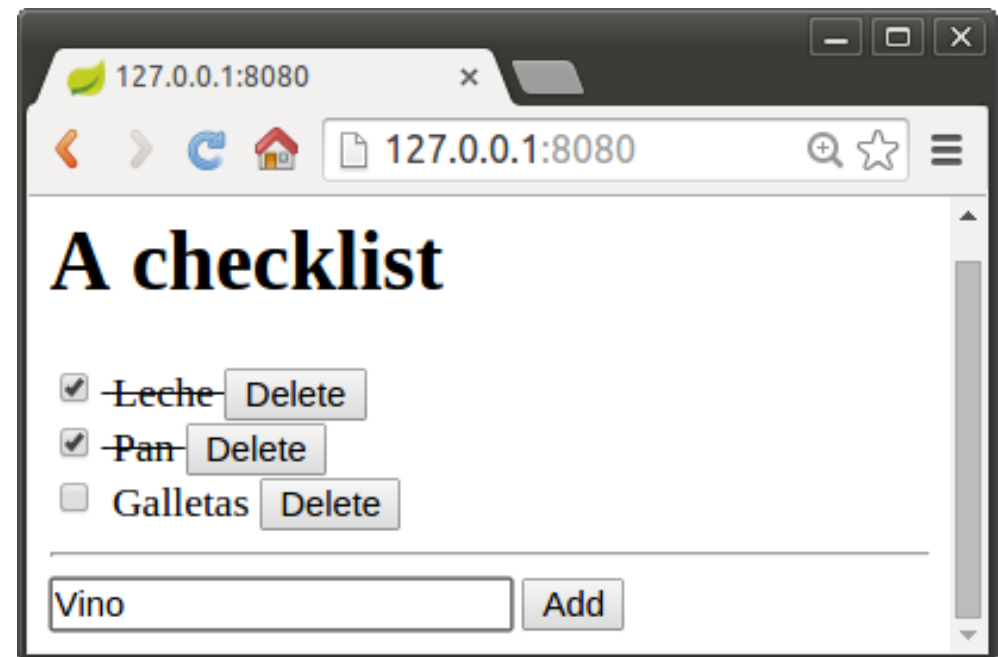
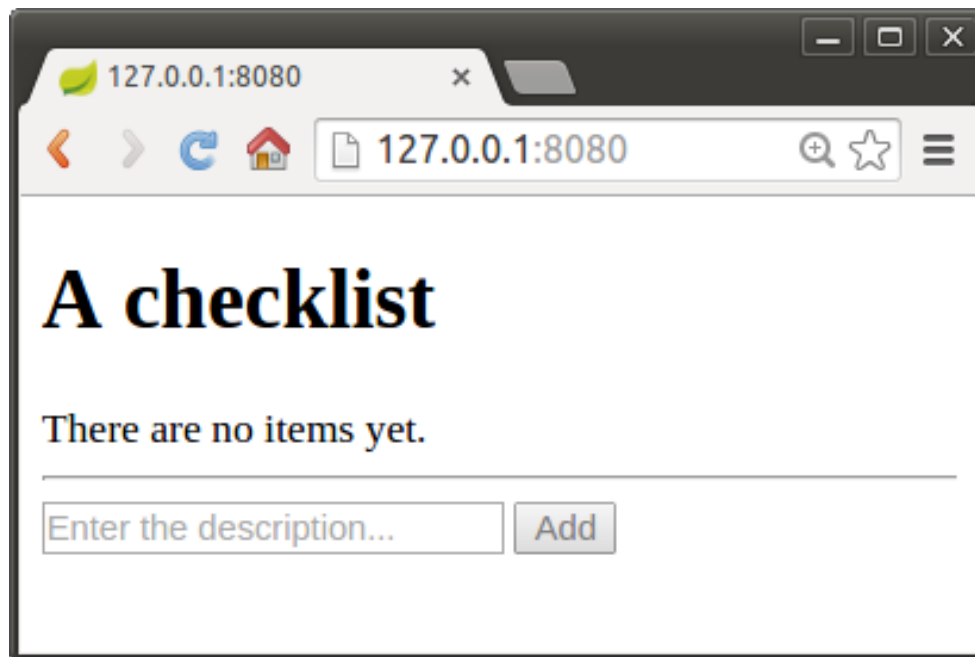


A screenshot of a web browser window titled 'ejem1' showing a form at 'localhost:8080'. The form contains the following elements:

- Two stacked text input fields, both containing the text 'sssssfff'.
- A text area containing the text 'sssssfff'.
- Two checkboxes: 'Vue' (checked) and 'JS' (unchecked).
- Three radio buttons: 'Item1' (checked), 'Item2' (unchecked), and 'Male' (unchecked).
- A radio button 'Female' (checked).

Ejercicio 2

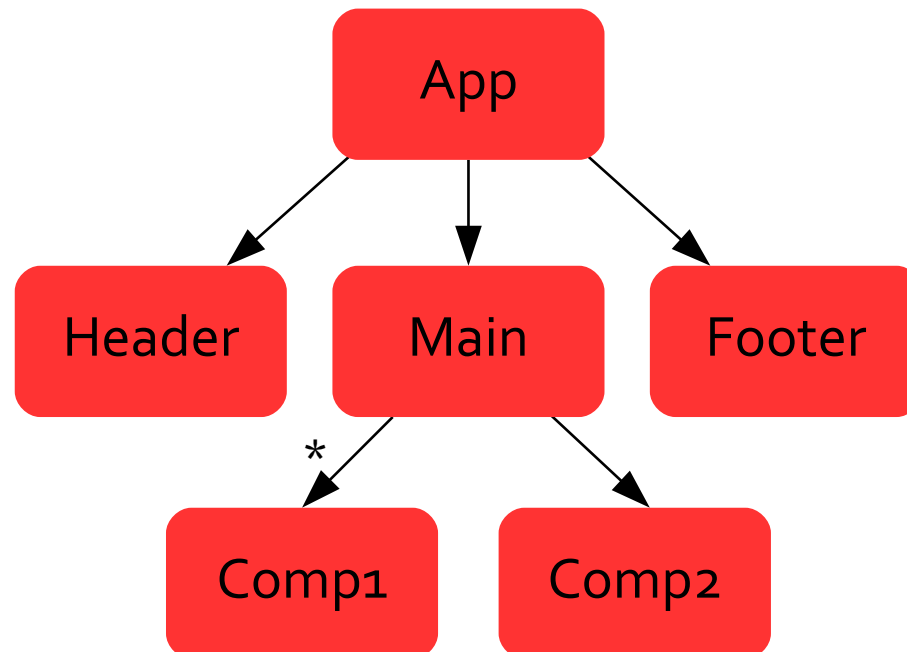
- Implementa una aplicación de **gestión de tareas**
- Las tareas se mantendrán en **memoria**



Composición de componentes

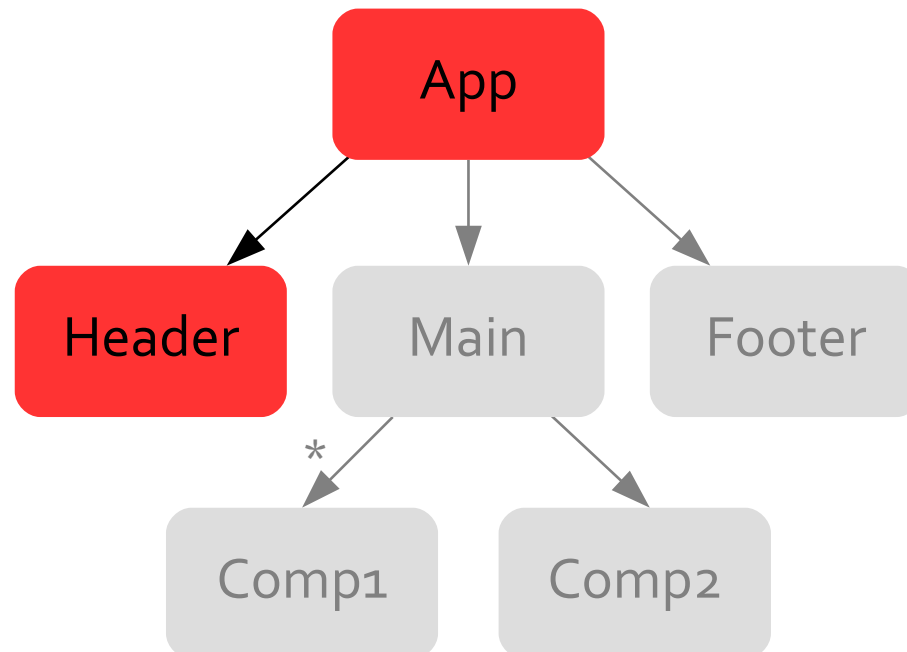
Árboles de componentes

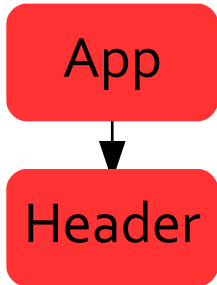
En Vue un componente puede estar formado por más componentes formando un árbol



Árboles de componentes

En Vue un componente puede estar formado por más componentes formando un árbol





Árboles de componentes

```
<h1>Title</h1>  
<p>Main content</p>
```



```
<app-header></app-header>  
<p>Main content</p>
```

```
<app-header>
```

```
<h1>Title</h1>
```

Árboles de componentes

App



Header

App.vue

```
<template>
  <div>
    <app-header/>
    <p>Main content</p>
  </div>
</template>

<script>
import AppHeader from './components/AppHeader.vue'

export default {
  name: "app",
  components: {
    AppHeader
  }
}
</script>
```

AppHeader.vue

```
<template>
  <h1>Title</h1>
</template>

<script>
export default {
  name: "app-header"
}
</script>
```

Composición de componentes

Árboles de componentes

App

Header

App.vue

```
<template>
  <div>
    <app-header/>
    <p>Main content</p>
  </div>
</template>

<script>
import AppHeader from '../components/AppHeader.vue'

export default {
  name: "app",
  components: {
    AppHeader
  }
}
</script>
```

Para incluir un componente se usa su **name**

AppHeader.vue

```
<template>
  <h1>Title</h1>
</template>

<script>
export default {
  name: "app-header"
}
</script>
```

Composición de componentes

Árboles de componentes

App

Header

App.vue

```
<template>
  <div>
    <app-header/>
    <p>Main content</p>
  </div>
</template>

<script>
import AppHeader from '../components/AppHeader.vue'

export default {
  name: "app",
  components: {
    AppHeader
  }
}
</script>
```

Para incluir un
componente se usa
su **name**

AppHeader.vue

```
<template>
  <h1>Title</h1>
</template>

<script>
export default {
  name: "app-header"
}
</script>
```

Árboles de componentes

- Al cargar la app en el navegador, en el árbol **DOM** cada componente incluye en su **elemento** el contenido de la **vista** (HTML)

```
▼ <div>  
  <h1>Title</h1>  
  <p>Main content</p>  
</div>
```


- Comunicación entre un **componente padre** y un **componente hijo**
 - Configuración de propiedades (Padre → Hijo)
 - Envío de eventos (Hijo → Padre)

Configuración de propiedades (Padre → Hijo)

- El componente padre puede especificar **propiedades** en el componente hijo como si fuera un elemento **nativo HTML**

```
<template>
  <div>
    <app-header title="Title!!"/>
    <p>Main content</p>
  </div>
</template>
```

Configuración de propiedades (Padre → Hijo)

App.vue

```
<template>
  <div>
    <app-header title="Title!!"/>
    <p>Main content</p>
  </div>
</template>

...
```

AppHeader.vue

```
<template>
  <h1>{{title}}</h1>
</template>

<script>
export default {
  name: "app-header",
  props: {
    title: undefined
  }
}
</script>
```

Configuración de propiedades (Padre → Hijo)

App.vue

```
<template>
  <div>
    <app-header title="Title!!"/>
    <p>Main content</p>
  </div>
</template>

...
```

AppHeader.vue

```
<template>
  <h1>{{title}}</h1>
</template>

<script>
export default {
  name: "app-header",
  props: {
    title: undefined
  }
}
</script>
```

Envío de eventos (Hijo → Padre)

- El componente hijo puede generar eventos que son atendidos por el padre como si fuera un elemento **nativo HTML**

La variable `$event` apunta al evento generado

```
<app-header @shown='shownTitle($event)' />
<p>Main content</p>
```

Composición de componentes

Envío de eventos (Hijo → Padre)

ejem17

App.vue

```
<template>
  <div>
    <app-header @shown="shownTitle($event)"/>
    <p>Main content</p>
  </div>
</template>

<script>
import AppHeader from './components/AppHeader.vue'

export default {
  name: "app",
  components: {
    AppHeader
  },
  methods: {
    shownTitle: function($event){
      // eslint-disable-next-line
      console.log(`Shown title: ${$event}`)
    }
  }
}
```

AppHeader.vue

```
<template>
  <div>
    <h1 v-if="show">Title</h1>
    <button @click="toggle()">Toggle</button>
  </div>
</template>

<script>
export default {
  name: "app-header",
  data: function(){
    return {
      show: true
    }
  },
  props: {
    title: undefined
  },
  methods: {
    toggle: function(){
      this.show = !this.show;
      this.$emit('shown', this.show);
    }
  }
};
```

Composición de componentes

Envío de eventos (Hijo → Padre)

ejem17

App.vue

```
<template>
  <div>
    <app-header @shown="shownTitle($event)"/>
    <p>Main content</p>
  </div>
</template>

<script>
import AppHeader from './components/AppHeader.vue'

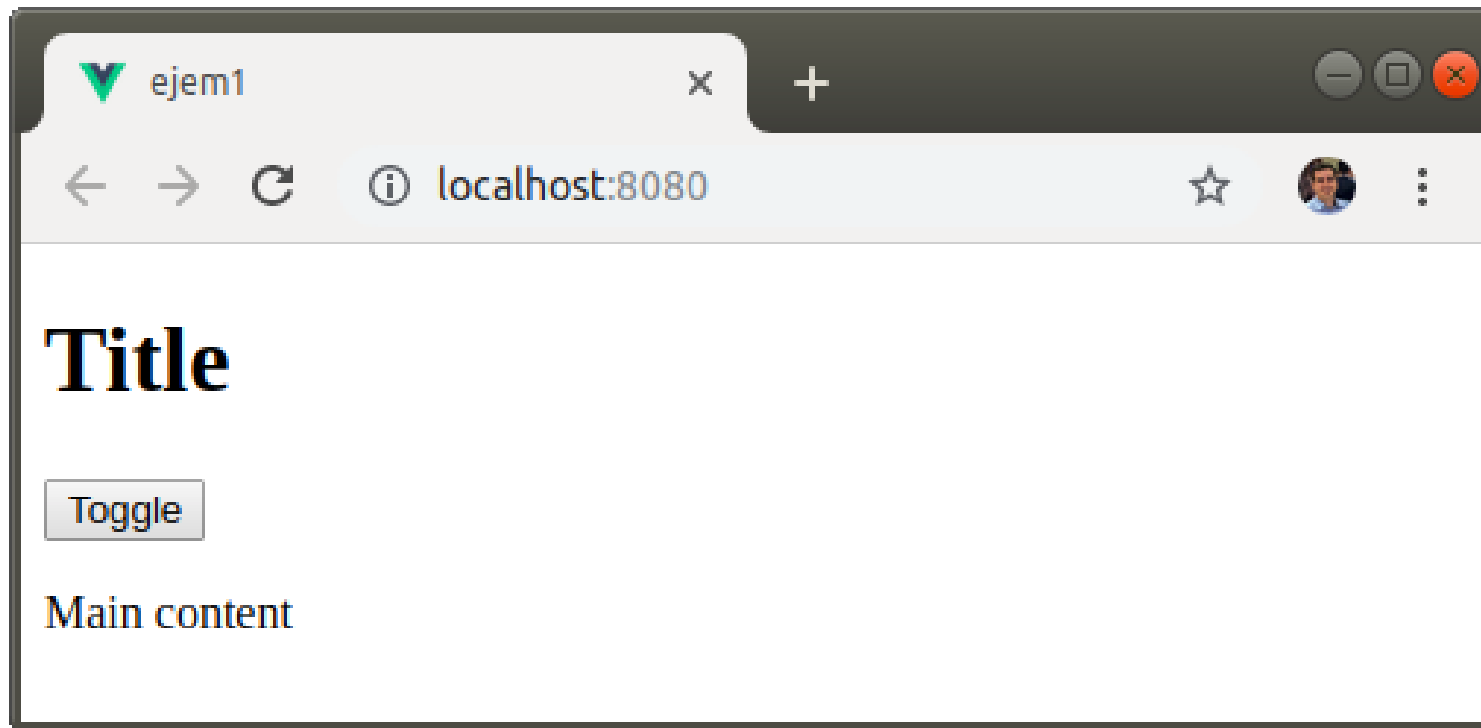
export default {
  name: "app",
  components: {
    AppHeader
  },
  methods: {
    shownTitle: function($event){
      // eslint-disable-next-line
      console.log(`Shown title: ${$event}`)
    }
  }
}
```

AppHeader.vue

```
<template>
  <div>
    <h1 v-if="show">Title</h1>
    <button @click="toggle()">Toggle</button>
  </div>
</template>

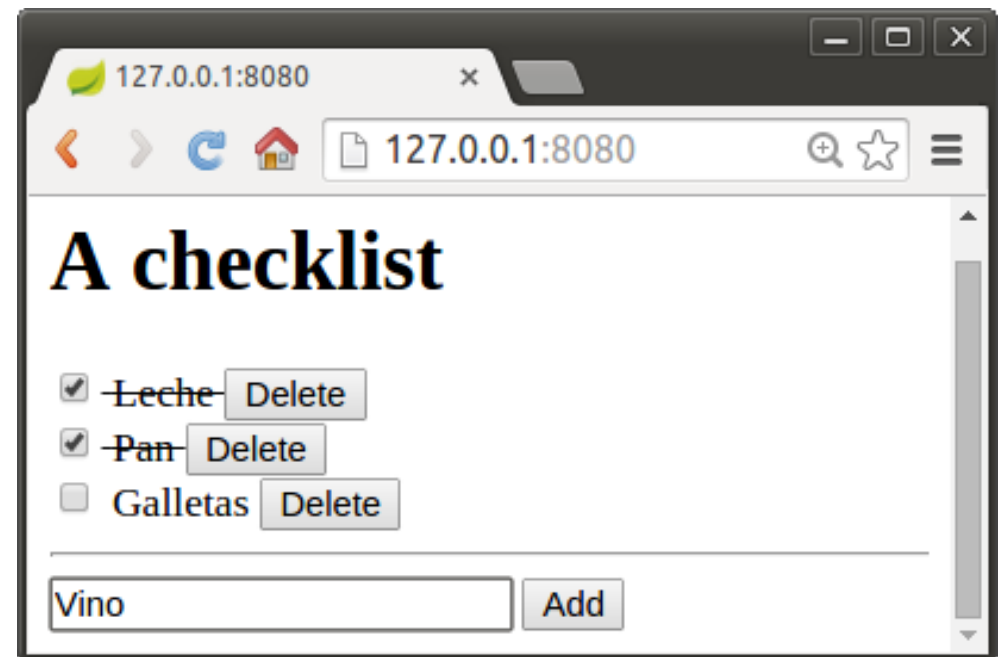
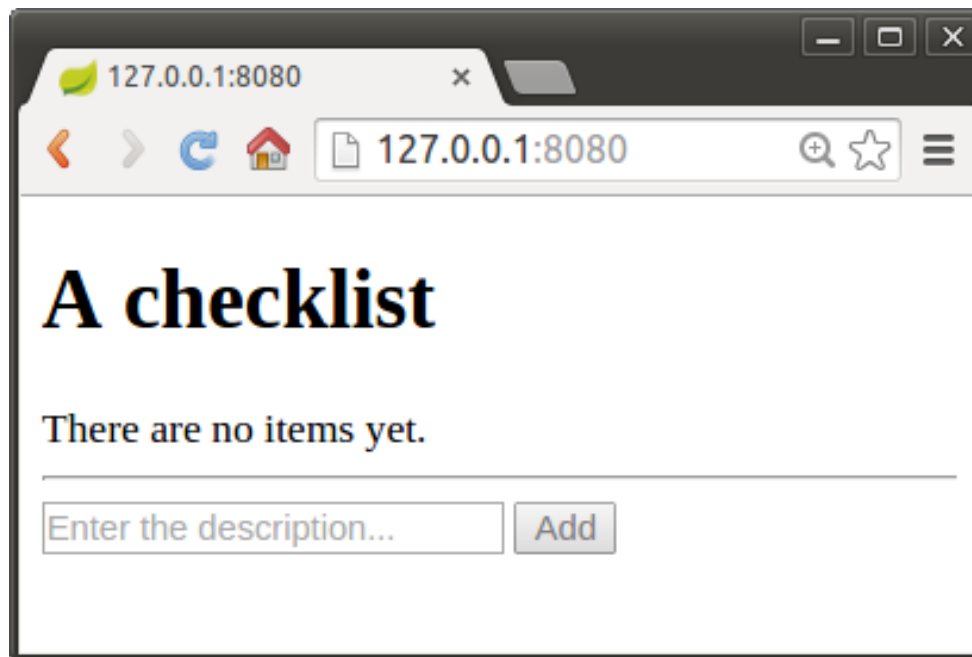
<script>
export default {
  name: "app-header",
  data: function(){
    return {
      show: true
    }
  },
  props: {
    title: undefined
  },
  methods: {
    toggle: function(){
      this.show = !this.show;
      this.$emit('shown', this.show);
    }
  }
};
</script>
```

Envío de eventos (Hijo → Padre)



Ejercicio 2

- Refactoriza la aplicación de **gestión de tareas** para que cada tarea sea un componente



- ¿Cuándo crear un nuevo componente?
 - El ejercicio y los ejemplos son **excesivamente sencillos** para que compense la creación de un nuevo componente **hijo**
 - En casos reales se crearían nuevos componentes:
 - Cuando la lógica y/o el *template* sean suficientemente **complejos**
 - Cuando los componentes hijos puedan **reutilizarse** en varios contextos