Compoñentes Vue

Introdución

A forma utilizada ata agora para traballar con Vue é útil cando os proxectos son pequenos. No momento en que estes empezan a medrar, hai que buscar un mecanismo para xestionar de forma fácil o proxecto e automatizar as tarefas na medida que sexa posible.

Para conseguilo utilizarase a sintaxe **SFC** (Single File Components), que permite encapsular a lóxica do compoñente Vue (JavaScript), o contido (HTML) e os estilos (CSS) nun único ficheiro con extensión ***.vue**.

Para crear unha páxina web baseada en SFCs é necesaria unha ferramenta que traduza o código en HTML, JavaScript e CSS. Para iso utilizarase a ferramenta **Vite**.

Vite (rápido en francés) é unha ferramenta lixeira e rápida que axiliza e facilita a creación e desenvolvemento de proxectos web modernos. Está desenvolvida por Evan You, o creador de Vue. Proporciona un servidor de desenvolvemento, un comando para empaquetar o proxecto e xerar código óptimo para produción. Encargarase de traducir o código SFC ao correspondente HTML, JavaScript e CSS.

Vite é unha ferramenta de desenvolvemento que non é exclusiva de Vue. Ten múltiples pantillas predeseñadas para crear proxectos utilizando diferentes tecnoloxías: vanilla, vue, react, typescript, … Ademais, ten plantillas mantidas pola comunidade que inclúen outras ferramentas e frameworks, como por exemplo django-vite: Integration of ViteJS in a Django project.

Para usar **Vite** é necesario ter instalado Node.js, contorna de execución de JavaScript que permite executar código JavaScript fóra do navegador.

O método recomendado para crear un proxecto Vue é utilizar a ferramenta create-vue, que está baseada en **Vite**. Create-vue é a ferramenta oficial para a creación dun proxecto Vue coa estrutura estándar de arquivos e carpetas.

Para crear un novo proxecto Vue coa ferramenta Vite usarase o seguinte comando:

```
npm create vue@latest
```

Durante a súa execución preguntará polas opcións de confguración. A modo de exemplo utilizarase as opcións por defecto:

```
Vue.js - The Progressive JavaScript Framework

✔ Project name: … vue-project
✔ Add TypeScript? … No / Yes
✔ Add JSX Support? … No / Yes
✔ Add Vue Router for Single Page Application development? … No / Yes
✔ Add Pinia for state management? … No / Yes
✔ Add Vitest for Unit Testing? … No / Yes
✔ Add an End-to-End Testing Solution? › No
✔ Add ESLint for code quality? › No

Scaffolding project in /media/DATOS/repositorios/2024-DWCC-cparis/clase/vue-project...

Done. Now run:

  cd vue-project
  npm install
  npm run dev
```

Unha vez o proxecto estea creado, haberá que installar as dependencias e lanzar o servidor:

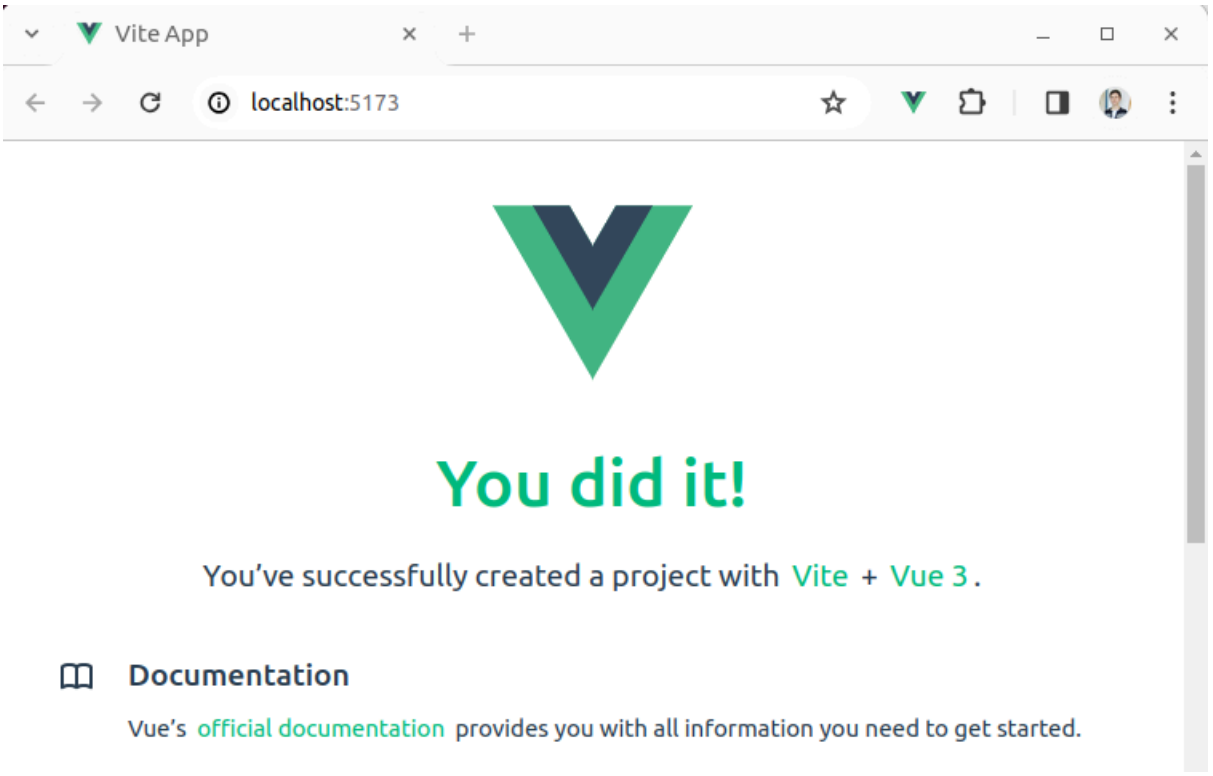
```
cd <your-project-name>
npm install
npm run dev
```

NOTA: recordar ter configurado o ficheiro .gitignore para non subir arquivos innecesarios ao repositorio.

Unha vez finalice o comando anterior, unha mensaxe indicará o enderezo onde se estará executando o servidor, neste caso:

```
VITE v6.1.0  ready in 465 ms

➜  Local:   http://localhost:5173/
➜  Network: use --host to expose
➜  Vue DevTools: Open http://localhost:5173/__devtools__/ as a separate window
➜  Vue DevTools: Press Alt(⌥)+Shift(⇧)+D in App to toggle the Vue DevTools
➜  press h + enter to show help
```

Probar a acceder a dita dirección dende un navegador e comprobar que a aplicación web se está executando. É necesario manter o servidor en execución durante o desenvolvemento da aplicación.



Nalgúns proxectos Vite créase o ficheiro de configuración **vite.config.js** onde se poden configurar detalles de funcionamento do empaquetador do proxecto. Dependendo das opcións escollidas á hora da creación do proxecto, este ficheiro pode ser diferente e incluso non existir.

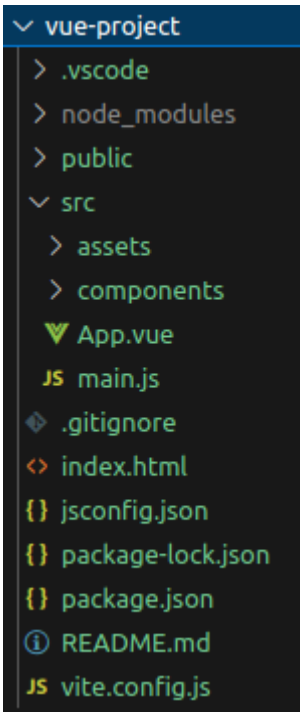
Unha vez que a aplicación estea lista para distribuir haberá que compilala e xerar o código para produción. Isto faise executando o seguinte comando:

```
npm run build
```

Para máis información, consultar Production Deployment | Vue.js.

Estrutura do proxecto

No apartado anterior, o comando para a creación dun proxecto Vite, creou unha estrutura de carpetas e arquivos. A continuación detállanse os máis importantes:



- **vite.config.js**: configuración de Vite.
- **package.json:** configuración do proxecto e dependencias. Os principais parámetros son:
    - **type**: tipo de sistema de módulos a utilizar. Con **module** usa ESM (versión do 2015), en caso contrario usa CommonJS (usados en Node.js).
    - **scripts**: colección de scripts do proxecto.
        - **dev**: lanza o servidor web de vite e configura vue para a contorna de desenvolvemento.
        - **build**: crea os ficheiros **JS** e **CSS** dentro de dist/ con todo o código da aplicación.
        - **preview:** facer unha previsualización local da versión de produción.
    - **dependencies**: colección de paquetes que utiliza o aplicación en producción.
    - **devDependencies**: colección de paquetes que utiliza a aplicación durante a fase de desenvolvemento.
- **node_modules/**: paquetes necesarios do proxecto.
- **index.html**: html con un div onde se cargará o compoñente (id=”app”).
- **public/**: lugar onde deixar elementos estáticos que non pasan por vite.
- **src/**: todo o código a desenvolver:
    - **assets/**: CSS, imaxes, etc. Elementos estáticos que vite procesará.
    - **components/**: conterá os ficheiros .vue dos compoñentes.
    - **router/**: carpeta cos ficheiros do router, se se usa vue-router.
    - **store/**: carpeta cos ficheiros do store se se usa pinia ou vuex.
    - **views:** se se usa vue-router, aquí poranse os compoñentes que constitúan unha vista da aplicación.
    - **App.vue**: compoñente principal e inicial do proxecto. Aquí cargarase a cabeceira, o menú, … e os diferentes compoñentes.
    - **main.js**: ficheiro JavaScript principal que crea a instancia de Vue e carga o compoñente principal chamado App.vue e renderízao en #app.

O ficheiro **index.html** contén o seguinte código:

```
<div id="app"></div>
<script type="module" src="/src/main.js"></script>
```

No ficheiro **/src/main.js** está o seguinte código. Observar que se usan módulos para importar Vue e que tamén se poden importar os ficheiros CSS:

```
import './assets/main.css'

import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

Os ficheiros **.vue** son ficheiros especiais que pemiten crear compoñentes Vue incluíndo no mesmo ficheiro o código HTML (na etiqueta <template>), CSS (na etiqueta <style>) e JavaScript (na etiqueta <script>).

O ficheiro **src/App.vue** é o compoñente principal da aplicación, o que contén o deseño da páxina principal. Trátase dun ficheiro SFC (*Single File Component*) que contén dentro da etiqueta <template> o código que se renderizará no div#app de **index.html.** Tamén pode conter outros compoñentes almacenados en ficheiros *.vue do directorio src/components.

No ficheiro **src/App.vue** móstranse imaxes cos logos. Observar que as imaxes están almacenadas en **src/assets/**

Cos ficheiros de configuración e fonte, Vite é capaz de construír o código final que será interpretado por un navegador.

Primeira páxina SFC

Partindo do proxecto base creado con Vite no apartado anterior, vaise borrar todo o contido para crear unha páxina simple con Vue.

Para iso, no ficheiro **App.vue** eliminarase todo o contido dentro de <template>, <script> e <style>, incluídos os atributos **setup** e **scoped**. O ficheiro **App.vue** debe quedar asi:

```
<script></script>

<template></template>

<style></style>
```

NOTA: os compoñentes de Vue poden ser escritos utilizando dous estilos diferentes de API: **Options API** e **Composition API**. A aplicación creada por defecto por Vite utiliza Composition API (<script setup>), mais nós estamos habituados a usar Options API, onde a lóxica dos compoñentes se define usando un obxecto. Neste documento vaise seguir usando **Options API**.

Observar no seguinte código que cando se usa o estilo Composition API, o elemento script leva un atributo **setup:**

<script> **// Options API** export default { // register child component } </script>	<script **setup**> **// Composition API** </script>

Tamén se eliminará o contido das carpetas **src/assets** e **src/components**. Dado que o ficheiro css se importaba dende JavaScript, tamén haberá que eliminar a liña onde se importaba no ficheiro **main.js:**

```
~~import './assets/main.css'~~

import { createApp } from 'vue';
import App from './App.vue';

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

NOTA: pode ser interesante personalizar a configuración de VS Code para traballar con Vue. Por exemplo, para formatear o código usando Prettier, engadirase a seguinte liña no ficheiro **settings.json**:

```
{
   …
  "[vue]": {
    "editor.defaultFormatter": "esbenp.prettier-vscode"
  },
}
```

Pode probarse a lanzar o servidor (**npm run dev**) e comprobar que se mostra unha páxina web baleira.

Se o servidor se está executando, toda modificación nos ficheiros será compilada por Vue automaticamente, o que provocará a actualización da web. Por exemplo, probar a engadir unha cabeceira no ficheiro **App.vue**:

```
<script></script>

<template>
  <h1>Hello World!</h1>
</template>

<style></style>
```

Tamén se pode escribir código propio de Vue no ficheiro **App.vue**:

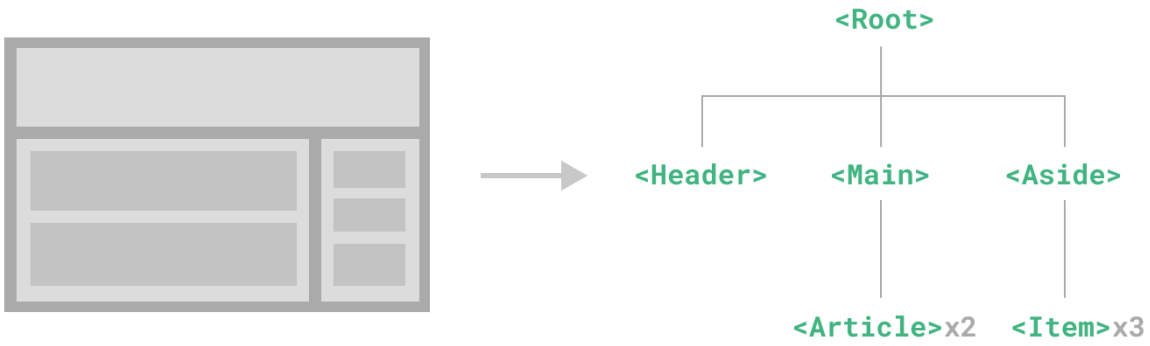
```
<script>
export default {
  data() {
    return {
      message: 'Ola mundo!',
      count: 0,
    };
  },
};
</script>

<template>
  <h1>{{ message }}</h1>
  <button @click="count++">You clicked me {{ count }} times.</button>
</template>

<style></style>
```

Observar no exemplo anterior que **export default** fai posible que no ficheiro **main.js** se importe o obxecto necesario para crear a aplicación con **createApp**.

Compoñentes

Os compoñentes permiten dividir a interface gráfica en pezas independentes e reutilizables. Cada compoñente terá o seu propio contido e lóxica de funcionamento, independente do resto da aplicación.

É común que unha aplicación estea organizada nunha árbore de compoñentes aniñados, facendo que a aplicación sexa escalable e fácil de manter:



Vue implementa o seu propio modelo de compoñentes permitindo encapsular contido personalizado e lóxica en cada compoñente. É dicir, cada compoñente estará formado por:

- Estrutura e información (HTML)
- Aspecto visual (CSS)
- Funcionalidade (JavaScript).

Para definir un compoñente usarase o formato **SFC** (Single File component), que almacenará os tres aspectos anteriores nun único ficheiro con extensión *.vue. Unha vez definido o compoñente, será posible utilizalo como unha etiqueta HTML na aplicación.

```
<ButtonCounter />
<ButtonCounter ></ButtonCounter >
```

Dado que **ButtonCounter** non é unha etiqueta HTML válida, Vue interpreta que debe existir un compoñente **ButtonCounter** co seu HTML, CSS e JavaScript, comproba de onde ten que importalo e utilízao na páxina xerando o código HTML asociado de forma optimizada.

Ademais de poder crear compoñentes personalizados, Vue ten un conxunto de compoñentes xa creados, como **<Teleport>** ou **<keepAlive>**, que poden ser utilizados en calquera aplicación.

Vue permite que os compoñentes dunha aplicación poidan comunicarse entre si intercambiando información.

Definir un compoñente

Cando se utilizan ferramentas de ensamblaxe os compoñentes defínense en ficheiros independentes con extensión .vue, con sintaxe SFC, nos que o código do compoñente se describe utilizando as etiquetas <template>, <script> e <style>.

Continuando co proxecto de Vue creado en apartados anteriores, vaise crear o compoñente **src/components/FoodItem.vue** co seguinte contido:

```
<script>
export default {
  data() {
    return {
      name: 'Apples',
      message: 'I like apples',
    };
  },
};
</script>

<template>
  <div>
    <h2>{{ name }}</h2>
    <p>{{ message }}</p>
  </div>
</template>

<style></style>
```

O código da etiqueta <script> comeza por **export default**, para que o obxecto definido poida ser importado dende outro ficheiro.

Rexistrar un compoñente

Un compoñente Vue necesita estar rexistrado para que poida ser utilizado por Vue, é dicir, para que Vue saiba onde encontrar a súa implementación. Primeiramente importarase coa sentencia **imprt** e despois rexistrarase usando o método **.component()**.

Continuando co exemplo anterior, utilizarase o seguinte código para rexistrar o compoñente no ficheiro **main.js**:

```
import { createApp } from 'vue';
import App from './App.vue';
import FoodItem from './components/FoodItem.vue';

const app = createApp(App);
app.component('FoodItem', FoodItem);
app.mount('#app');
```

NOTA: reescribíronse as últimas liñas do ficheiro **main.js** para almacenar o obxecto devolto por createApp() nunha constante e poder acceder aos seus métodos.

Agora xa é posible utilizar o compoñente dentro da etiqueta **<template>** de **App.vue**:

```
<script>
export default {
  data() {
    return {
      message: 'Ola mundo!',
      count: 0,
    };
  },
};
</script>

<template>
  <h1>{{ message }}</h1>
  <button @click="count++">You clicked me {{ count }} times.</button>
  <FoodItem />
  <FoodItem />
</template>

<style>
#app > div {
  border: dashed black 1px;
  display: inline-block;
  margin: 10px;
  padding: 10px;
  background-color: lightgreen;
}
</style>
```

Pode probarse a executar o servidor e cargar a páxina web nun navegador.

En SFC recoméndase que os nomes dos compoñentes sigan a sintaxe PascalCase (**<FoodItem />)** para diferencialos das etiquetas nativas de HTML. Vue SFC é un formato compilado, polo que cada vez que encontre unha etiqueta dun compoñente, o código será traducido.

Se os compoñentes se escriben directamente no DOM (in-DOM templates), hai que cumprir as restricións HTML dos navegadores e utilizar kebab-case (**<food-item/>**). Estas limitacións non se aplican a SFC, nin a **template: ‘ ’.** Máis información.

No exemplo anterior fíxose un **rexistro global** dun compoñente en Vue:

```
// main.js
import { createApp } from 'vue';
import App from './App.vue';
import MyComponent from './components/MyComponent.vue';

const app = createApp(App);
app.component('MyComponent', MyComponent);
app.mount('#app');
```

Un compoñente rexistrado globalmente poderá ser usado dentro da etiqueta <template> de calquera compoñente da aplicación:

```
<MyComponent/>
<MyComponent/>
```

Os compoñentes tamén poden ser **rexistrados de forma local**. Un compoñente rexistrado localmente só estará dispoñible no compoñente onde se rexistre. Non estará dispoñible nos seus compoñentes descendentes.

O rexistro local faise usando a opción **components:**

```
<script>
import ComponentA from './ComponentA.vue'

export default {
  components: {
    ComponentA
  }
}
</script>

<template>
  <ComponentA />
</template>
```

Exercicios:

1. Crea un proxecto Vue onde o compoñente raíz da aplicación teña unha propiedade que sexa unha mensaxe de texto, por exemplo “ola mundo”. Configura o código HTML para mostrar a mensaxe. Engade ao HTML un botón de tal forma que cada vez que se pulse transforme a mensaxe de minúscula a maiúscula ou viceversa. Engade outro botón que permita engadir un carácter, por exemplo “!” á mensaxe de texto.

2. Crea un proxecto Vue con un compoñente a reutilizar que conteña unha propiedade que almacene o número de veces que se pulsou nun botón. Engade ao compoñente un botón que incremente o contador anterior en unha unidade cada vez que se pulse. O texto que aparece no botón debe informar do número de veces se pulsou.

   Engade no compoñente raíz da aplicación 3 instancias do compoñente anterior. ¿O valor do contador é compartido polos compoñentes?

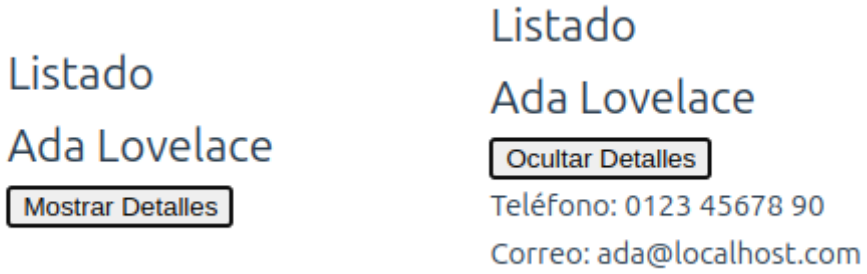
3. Crea un proxecto Vue para mostrar unha lista de contactos. A información dos contactos debe estar nun compoñente separado (**Contact**) no que se debe gardar a información dunha persoa: nome, teléfono e correo electrónico, como mínimo.

   De momento, inicializa manualmente os datos persoais dunha soa persoa no compoñente Contact.

   Engade dúas instancias do compoñente **Contact** na compoñente raíz da aplicación.

   Configura o compoñente **Contact** para que mostre o nome da persoa e teña un botón para mostrar/ocultar o resto da información.

   Cando se lance a aplicación debe mostrar en pantalla os seguintes datos. Observar que o botón permite mostrar/ocultar os detalles:



Propiedades Props

Un compoñente permite definir un elemento reutilizable dunha páxina web. Estes compoñentes comparten o aspecto visual, mais sería interesante que se poidese personalizar o contido, recibíndoo como parámetro. Por exemplo, nunha aplicación dun blog que utilizase un compoñente para definir os posts, estes poderían recibir como parámetro a información a mostrar. Isto é posible utilizando **props**.

Props (abreviatura de propiedades) son atributos personalizados que se poden rexistrar nun compoñente e que se utilizan para pasarlle información. Para poder utilizar **props** no compoñente hai que declarar a lista de propiedades que o compoñente pode aceptar usando a opción **props**. Esta declaración informa a Vue que o compoñente espera recibir esas propiedades.

As propiedades poden declararse como un **array de strings**:

```
<!-- BlogPost.vue -->
<script>
export default {
 props: ['title', 'likes'],
}
</script>

<template>
  <h4>{{ title }}</h4>
</template>
```

Tamén é posible declarar props usando a **sintaxe de obxectos**. Para cada propiedade, a chave é o nome da propiedade e o valor é a función construtora do tipo de dato. Isto permite documentar o compoñente e axuda a evitar erros ás persoas desenvolvedoras para saber o tipo de dato a pasar:

```
export default {
  props: {
    title: String,
    likes: Number
  }
}
```

Cando se pasa un valor a un atributo prop, convértese nunha propiedade da instancia do compoñente. O valor da propiedade está accesible no compoñente como se fose unha propiedade máis.

Un compoñente pode ter tantas propiedades como sexa necesario.

Unha vez que a prop está rexistrada, pode enviarse información como un atributo personalizado do compoñente:

```
<!--  .App.vue -->
<script>
import BlogPost from "./components/BlogPost.vue";

export default {
  // rexistro do compoñente
  components: {
    BlogPost,
  },
};
</script>

<template>
  <BlogPost title="My journey with Vue" />
  <BlogPost title="Blogging with Vue" />
  <BlogPost title="Why Vue is so fun" />
</template>
```

Nos exemplos vistos ata agora pasouse un valor estático a unha propiedade. Tamén se pode asignar o valor dinamicamente usando **v-bind** ou a súa abreviatura “**:”.** Exemplo:

```
<!-- Dynamically assign the value of a variable -->
<BlogPost :title="post.title" />
```

Así, nunha aplicación de posts será habitual ter un array de posts no compoñente raíz. Pode utilizarse v-for e v-bind nestes casos para enlazar dinamicamente os valores das propiedades. Pode probarse o exemplo en funcionamento.

```
<script>
export default {
  data() {
    return {
      posts: [
        { id: 1, title: 'My journey with Vue' },
        { id: 2, title: 'Blogging with Vue' },
        { id: 3, title: 'Why Vue is so fun' }
      ]
    }
  }
}
</script>

<template>
  <BlogPost v-for="post in posts" :key="post.id" :title="post.title"></BlogPost>
</template>
```

Os exemplos anteriores utilizan props con Strings. Tamén é posible pasar outros tipos de datos. Ver diferentes exemplos na documentación.

Se se queren pasar todas as propiedades dun obxecto como props, pode usarse v-bind sen argumentos. Exemplo:

```
export default {
  data() {
    return {
      post: {
        id: 1,
        title: 'My Journey with Vue'
      }
    }
  }
}
```

```
<BlogPost v-bind="post" />
<!-- Sería equivalente a  -->
<BlogPost :id="post.id" :title="post.title" />
```

Nomes longos de propiedades utilizan a nomenclatura **camelCase**, evitando ter que usar aspas (‘) cando se usen como propiedades dun obxecto, ademais de permitir ser referenciadas directamente en expresións na template porque son considerados identificadores válidos en JavaScript:

```
<script>
export default {
  props: {
    title: String,
    greetingMessage: String,
  },
};
</script>

<template>
  <p>{{ greetingMessage }}</p>
</template>
```

Tecnicamente pode usarse a sintaxe camelCase ao pasar propiedades a un compoñente fillo. Sen embargo, a convención é usar kebab-case para seguir a sintaxe dos atributos HTML:

```
<MyComponent greeting-message="hello" />
```

Comunicación unidireccional

As propiedades **props** permiten a comunicación unidireccional entre un compoñente ascendente e o descendente. Se a propiedade se actualiza no compoñente pai, tamén se actualizará no descendente, pero non ao revés. Un intento de modificación dunha propiedade nun compoñente descendente, provocará un erro.

NOTA: debido a que os obxectos e arrays se pasan por referencia, no compoñente descendente poderían cambiarse as súas propiedades, mais debe evitarse facelo.

Hai dous casos de uso nos que interesa modificar a propiedade:

- A propiedade utilízase para **establecer un valor inicial** e o compoñente quere usala despois como unha propiedade máis do compoñente. Neste caso é mellor definir unha propiedade local inicializada ao valor de prop:

```
export default {
  props: ['initialCounter'],
  data() {
    return {
      // counter only uses this.initialCounter as the initial value;
      // it is disconnected from future prop updates.
      counter: this.initialCounter
    }
  }
}
```

- A propiedade pásase como **un valor que é necesario transformar**. Neste caso, é mellor definir unha propiedade calculada usando o valor da propiedade:

```
export default {
  props: ['size'],
  computed: {
    // computed property that auto-updates when the prop changes
    normalizedSize() {
      return this.size.trim().toLowerCase()
    }
  }
}
```

Validación de props

Os compoñentes poden especificar requirimentos que deben cumprir as propiedades, como a especificación do tipo de datos visto no apartado anterior. Se o requirimento non se cumpre, Vue mostra un aviso na consola do navegador. Isto é especialmente útil durante a fase de desenvolvemento do compoñente e tamén cando se usan compoñentes de terceiros.

Tamén é posible especificar validacións de propiedades, para o que se utiliza un obxecto onde se definen os requirimentos de validación, en lugar de usar array de Strings. Exemplo:

```
export default {
  props: {
    // Basic type check (`null` and `undefined` values will allow any type)
    propA: Number,
    // Multiple possible types
    propB: [String, Number],
    // Required string
    propC: {
      type: String,
      required: true
    },
    // Number with a default value
    propD: {
      type: Number,
      default: 100
    },
    // Custom validator function
    propF: {
      validator(value) {
        // The value must match one of these strings
        return ['success', 'warning', 'danger'].includes(value)
      }
    },
  }
}
```

O tipo de dato pode ser un dos seguintes construtores nativos: **String, Number, Boolean, Array, Object, Date, Function, Symbol** e **Error.**

Características adicionais:

- Todas as propiedades son opcionais por defecto, a menos que se especifique **required: true**.
- A ausencia dunha propiedade opcional, que non sexa Boolean, terá un valor **undefined.**
- A ausencia dunha propiedade Boolean será considerado que ten un valor **false**. Isto pode modificarse utilizando un valor por defecto.
- Se se especifica un valor por defecto, será usado cando a propiedade é **undefined**, é dicir, cando non se pase a propiedade ou o seu valor sexa **undefined.**

Se a validación non se cumpre, Vue mostra un aviso por consola.

NOTA: as propiedades son validadas antes de crear a instancia do compoñente, polo que as propiedades de instancia (p.ex: data, computed, etc) non estarán dispoñibles para **default** e **validator**.

As propiedades booleanas teñen un comportamento especial, permitindo simular o comportamento dos atributos booleanos. No seguinte exemplo declárase unha propiedade booleana dun compoñente e no lado da dereita exemplifícase como usar o compoñente:

export default { props: { disabled: Boolean } }	<!-- equivalent of passing :disabled="true" --> <MyComponent disabled /> <!-- equivalent of passing :disabled="false" --> <MyComponent />

Exercicios:

1. Exercicio titorial.
2. Modifica o exercicio do listado de contactos do apartado anterior para que o array de persoas estea definido no compoñente pai e se pase a información ao compoñente fillo utilizando props. No compoñente raíz da aplicación debes crear un array con dúas persoas mínimo indicando o nome, teléfono e correo electrónico.

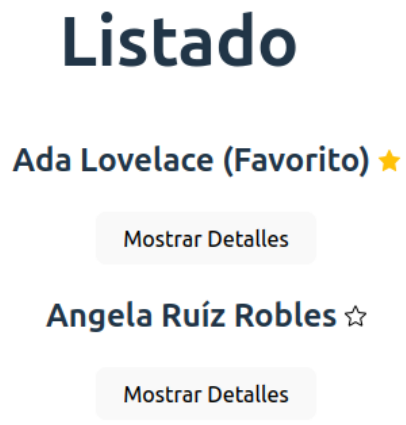
   Crea unha aplicación que mostre a información dos contactos utilizando o compoñente Contact.

   Implementa a validacion de props.
3. Engade ao exercicio anterior información de se un contacto é favorito ou non. Esta información almacénase no array do compoñente raíz e pásase ao compoñente Contact como prop.

   Mostra visualmente se é un contacto favorito ou non. Podes utilizar texto ou imaxes, como se mostra a continuacion.

   Engade tamén ao compoñente descendente a funcionalidade que permita modificar se un compoñente é favorito ou non. Fai que **visualmente** se actualice a información de se é ou non favorito. De momento non se explicou como cambiar este dato no compoñente ascendente.

   Exemplo:



Eventos

Os compoñentes poden emitir eventos personalizados utilizando o método **$emit**. Isto permite que un compoñente poida notificar os cambios ao seu ascendente.

```
<!-- MyComponent -->
<button @click="$emit('someEvent')">click me</button>
```

O método **$emit()** tamén está dispoñible na instancia do compoñente:

```
<!-- MyComponent -->
export default {
  methods: {
    submit() {
      this.$emit('someEvent')
    }
  }
}
```

No compoñente ascendente usarase a directiva **v-on:nome-evento,** ou a súa abreviatura **@nome-evento,** para escoitar o evento personalizado:

```
<!-- App.vue -->
<template>
  <MyComponent @some-event="callback" />
</template>
```

No compoñente raíz pode usarse un método para manexar o evento:

```
<!-- App.vue -->
<script>
export default {
  …
  methods: {
    incrementarContador() {
      this.count = this.count + 1;
    },
  },
};
</script>

<template>
  <ChildComponent @some-event="incrementarContador"></ChildComponent>
  <p>{{ count }}</p>
</template>
```

O compoñente pode declarar explicitamente os eventos que emitirá usando a opción **emits.** Aínda que é opcional esta declaración, é recomendable para documentar mellor o compoñente**:**

```
<!-- MyComponent -->
export default {
  emits: ["someEvent"],
};
```

NOTA: se un evento nativo (por exemplo, click) é definido na opción **emits**, o listener só escoitará os eventos click emitidos polo compoñente e non responderá ao evento click nativo.

NOTA: a diferencia dos eventos nativos do DOM, os eventos emitidos polos compoñentes non se propagan. Só é posible escoitar eventos emitidos por descendentes directos.

Algunhas veces é útil pasar un valor específico xunto co evento emitido. Isto lógrase pasando **argumentos extra a $emit**:

```
<!-- MyComponent -->
<button @click="$emit('someEvent', 1)">click me</button>
```

```
<!-- App.vue →
<script>
export default {
  …
  methods: {
    incrementarContador(n) {
      this.count = this.count + n;
    },
  },
};
</script>

<template>
  <MyComponent @some-event="incrementarContador"></ChildComponent>
  <p>{{ count }}</p>
</template>
```

Os parámetros extra pasados a $emit() despois do nome do evento, serán pasados ao manexador de eventos. Por exemplo, a instrución $emit(“someEvent”, 1, 2, 3) fará que a función manexadora do evento reciba tres argumentos.

Ao igual que se pode facer a validación de props, tamén é posible facer a **validación de eventos** emitidos utilizando a sintaxe de obxectos. Isto é moi útil na **fase de desenvolvemento** de software e, sobre todo, se se traballa en equipo.

Para engadir a validación, asígnase ao evento unha función que recibe os argumentos pasados e devolve un booleano indicando se o evento é válido ou non:

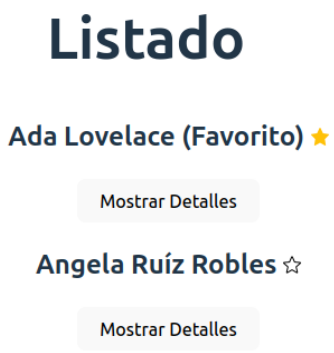
```
<!-- MyComponent -->
<script>
export default {
  emits: {
    // No validation
    otherEvent: null,

    // Validate submit event
    someEvent: (n) => {
      if (n > 5) return true;
      else {
        console.warn('Número demasiado baixo');
        return false;
      }
    },
  },
};
</script>

<template>
  <button @click="$emit('someEvent', 4)">click me</button>
</template>
```

Igual que os nomes dos compoñentes e propiedades, os nomes dos eventos proporcionan unha transformación automática entre camelCase e kebab-case. Recoméndase usar a nomenclatura adecuada á linguaxe onde se usan: nun atributo HTML usarase kebab-case, nunha propiedade JavaSript usarase camelCase e nunha etiqueta HTML usarase PascalCase.

Exercicios:

1. Exercicio titorial.

2. Modifica o exercicio do listado de persoas para engadir a funcionalidade de “Cambiar favorito”. Configura un botón no compoñente descendente que permita modificar se unha persoa é favorita ou non, de tal forma que a información se actualice no compoñente ascendente.



3. Amplía a aplicación do exercicio anterior coas seguintes funcionalidades:

    a. Engade un novo compoñente que conteña un formulario para dar de alta un novo contacto. Engade este novo compoñente ao compoñente raíz da aplicación e fai a implementación necesaria para que ao dar de alta un novo contacto, se engada á lista de contactos do compoñente raíz.
    b. No compoñente que visualiza os datos de cada persoa (Contact) engade un botón que permita eliminar ese contacto. O contacto debe borrarse do array do compoñente raíz da aplicación.

4. Crea unha nova aplicación que teña dous compoñentes descendentes. No compoñente raíz está a información dunha persoa: nome e ano de nacemento.

    O primeiro compoñente a crear debe mostrar o nome e ano de nacemento desa persoa. O outro compoñente debe mostrar un formulario cuberto cos datos persoais desa persoa. Este formulario debe permitir a modificación dos datos, actualizándose o compoñente que os visualiza cando isto suceda.

    Configura a validación de props de tal forma que o ano de nacemento sexa un número e sexa obrigatorio.

Fallthrough Attributes

Un atributo fallthrough é un atributo ou un manexador de eventos pasado a un compoñente que non está declarado explicitamente en **props** nin en **emits**. Exemplos típicos destes atributos son **class, style** e **id**.

Cando un compoñente renderiza só un elemento raíz, os atributos fallthrough son automaticamente engadidos como atributos do elemento raíz. Por exemplo, dado un compoñente <MyButton> co seguinte código HTML:

```
<!-- template of <MyButton> -->
<button>click me</button>
```

Cando se utiliza o compoñente anterior utilizando a seguinte instrución:

```
<MyButton class="large" />
```

O código que xera Vue é:

```
<button class="large">click me</button>
```

Observar que <MyButton> non declara o atributo class como **props,** mais engádese automaticamente ao elemento raíz do compoñente <MyButton>.

Se o elemento raíz do compoñente descendente xa ten os atributos **class** ou **style,** os atributos fallthrough serán fusionados cos que xa tiña o elemento. Exemplo:

```
<!-- template de <MyButton> -->
<button class="btn">click me</button>
```

```
<!-- compoñente ascendente -->
<MyButton class="large" />
```

```
<!-- código HTML xerado por Vue -->
<button class="btn large">click me</button>
```

Para os manexadores de eventos **v-on** aplícase a mesma regra:

```
<MyButton @click="onClick" />
```

O manexador do evento click engadirase ao elemento raíz de MyButton, neste caso un botón. Cando se pulse o botón **executarase o método onClick do compoñente ascendente**. Se o botón de MyButton xa ten un listener rexistrado, executaranse os dous.

Se o compoñente descendente ten á súa vez como nodo raíz outro compoñente, os atributos fallthrough recibidos son automaticamente reenviados ao segundo nivel de descencencia. No seguinte exemplo MyButton reenvia os atributos automaticamente a BaseButton:

```
<!-- root template -->
<MyButton class="large" />
```

```
<!-- template of <MyButton> -->
<BaseButton />
```

```
<!-- template of <BaseButton> -->
<button>click me</button>
```

NOTA: fallthrough attributes non inclúen atributos declarados como **props**, nin listeners de eventos declarados con **v-on** por <MyButton>. Noutras palabras, as propiedades e os listeners son consumidos por <MyButton>

Estilos con CSS

Vue permite aplicar estilos CSS de diferentes formas: usar un **ficheiro externo** CSS, usar estilos **globais** en SFC (ficheiros .vue) ou usar estilos **locais** en SFC.

Vue permite utilizar un ficheiro externo CSS e aplicalo de forma global á aplicación. A ferramenta de creación de proxectos Vite crea os ficheiros CSS **src/assets/main.css** e **src/assets/base.css.** Observar que o ficheiro **main.css** se importa dende **src/main.js** coa instrución **import './assets/main.css'.** Esta instrución fai que o ficheiro de estilos se colla na fase de construción do proxecto e se engada ao sitio web creado.

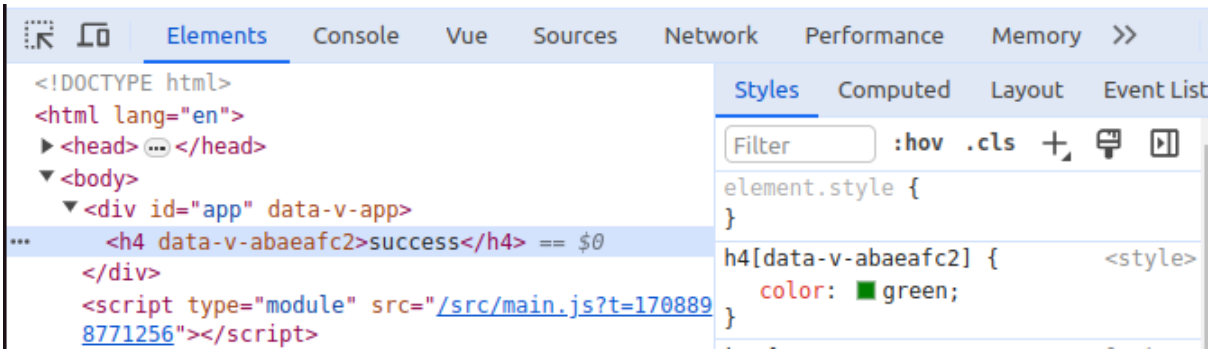
Cando nun ficheiro SFC se utiliza a etiqueta **<style>** os estilos así definidos son globais da aplicación. Se se queren definir estilos a aplicar nun único compoñente hai que engadir o atributo **scoped**:

```
<style scoped></style>
```

Para que isto funcione, Vue inclúe un atributo xerado de forma aleatoria durante a tradución de código, tanto no compoñente como nas regras de estilo. Así, observar como se mostrará o seguinte código nas ferramentas de desenvolvemento do navegador:

```
<template>
  <h4>{{ title }}</h4>
</template>

<style scoped>
h4 {
  color: green;
}
</style>
```

A seguinte imaxe amosa como se traduciu o código anterior:



A continuación móstranse dous exemplos de utilización de estilos globais e locais:

- Exemplo que utiliza estilos globais que afectan a todos os compoñentes.
- Exemplo que utiliza estilos locais que se aplican a un só compoñente.

Ao usar **scoped**, os estilos definidos no compoñente ascendente, non se herdan no compoñente descendente. Sen embargo, o nodo raíz do compoñente descendente si que se verá afectado tanto polos estilos definidos no nodo ascendente como no propio compoñente.

É posible utilizar estilos globais e locais no mesmo compoñente:

```
<style>
/* global styles */
</style>

<style scoped>
/* local styles */
</style>
```

A decisión de creación dun compoñente independente tamén se pode basear nos estilos a aplicar. Por exemplo, pode decidirse crear un compoñente para a cabeceira da páxina e definir os estilos a aplicar no propio compoñente con scoped. Os estilos así definidos non afectarán ao resto da aplicación.

Slots

Os compoñentes poden aceptar **props,** que son valores JavaScript de calquera tipo. Sería interesante que tamén poidesen recibir contido HTML, por exemplo, para poder pasar un fragmento HTML a un compoñente descendente e que o renderice integrado no seu propio código HTML.

Así, por exemplo, sería posible utilizar o compoñente FancyButton coa seguinte sintaxe:

```
<FancyButton>
  Click me! <!-- slot content -->
</FancyButton>
```

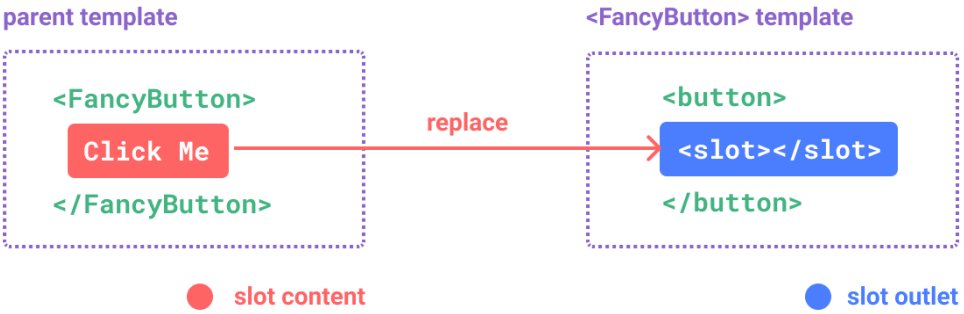
Sendo o código de **FancyButton.vue** o seguinte:

```
<template>
  <button class="fancy-btn">
    <slot></slot>    <!-- slot outlet -->
  </button>
</template>
```

O elemento **<slot>** indica o oco onde se debe colocar o contido proporcionado polo compoñente ascendente. Tras a compilación, o resultado final no DOM será o que aparece a continuación. Ver exemplo en funcionamento.

```
<button class="fancy-btn">Click me!</button>
```

O exemplo anterior pode ilustrarse coa seguinte imaxe:



Co uso de slots, o compoñente <FancyButton> é responsable de renderizar o botón (e o seu estilo), mentres que o contido é proporcionado polo compoñente ascendente.

O contido dun slot non está limitado só a texto, pode ser calquera template válida. Por exemplo, pódense pasar múltiples elementos e incluso outros compoñentes. Ver exemplo en funcionamento:

```
<FancyButton>
  <span style="color:red">Click me!</span>
  <AwesomeIcon name="plus" />
</FancyButton>
```

Ao usar slots, o compoñente <FancyButton> é máis flexible e reutilizable. Pode usarse en diferentes lugares con diferente contido, pero co mesmo estilo CSS.

Ver outro exemplo de utilización de slots en w3schools.

Hai casos nos que é útil especificar un **contido por defecto** dun slot para usalo no caso de que non se proporcione un contido específico. O seguinte exemplo crea un botón personalizando o texto a mostrar, de tal forma que se non se pasa ese texto, mostrará por defecto “Submit”:

```
<!-- FancyButton.vue -->
<button type="submit">
  <slot>
    Submit <!-- Contido por defecto -->
  </slot>
</button>
```

Na seguinte táboa móstranse á esquerda diferentes formas de utilizar o compoñente e á dereita especifícase como se renderiza realmente. Ver exemplo en funcionamento:

<SubmitButton /> <SubmitButton>Save</SubmitButton>	<button type="submit">Submit</button> <button type="submit">Save</button>

O **contido dun slot** ten acceso aos datos dentro do **ámbito do compoñente ascendente,** porque é aí onde está definido. Pola contra, o contido dun slot non ten acceso aos datos do

compoñente descendente. No seguinte exemplo, ambos {{message}} mostrarán o mesmo contido.

```
<span>{{ message }}</span>
<FancyButton>{{ message }}</FancyButton>
```

Slots con nome

Cando nun compoñente se engaden varias veces as etiquetas <slot></slot>, o contido recibido do compoñente ascendente duplicarase en cada slot. Ver exemplo.

Algunhas veces é útil ter varios ocos (slots) nun mesmo compoñente, mais con diferente contido. Para poder ter slots con diferente contido utilizarase o atributo especial denominado **name,** que permitirá asignar un identificador único ao slot e así determinar onde renderizar o contido. Un slot sen atributo **name** terá implicitamente o nome **"default"**. Exemplo:

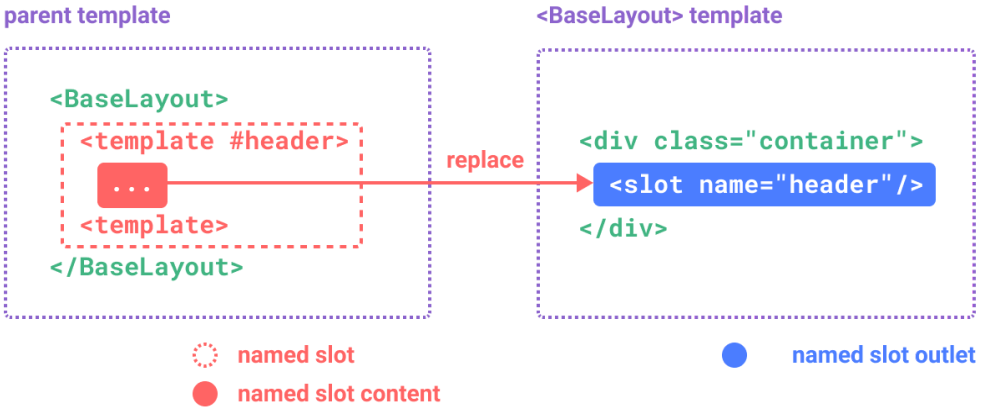
```
<!-- BaseLayout.vue -->
<div class="container">
  <header>
    <slot name="header"></slot>
  </header>
  <main>
    <slot></slot>
  </main>
  <footer>
    <slot name="footer"></slot>
  </footer>
</div>
```

No compoñente ascendente é necesario un mecanismo para pasar varios fragmentos de contido e especificar en cal dos slots se colocará cada un. Para isto utilizarase o elemento **<template>** coa directiva **v-slot** e o nome do **slot** ao que se refire. A directiva **v-slot** ten a forma abreviada **#,** polo que os dous exemplos seguintes son equivalentes. Ver exemplo en funcionamento**:**

```
<!-- compoñente ascendente -->
<BaseLayout>
  <template v-slot:header>
    <h1>Here might be a page title</h1>
  </template>
  <template v-slot:default>
     <p>Main content.</p>
     <p>Another paragraph.</p>
   </template>
   <template v-slot:footer>
     <p>Here's some contact info</p>
   </template>
</BaseLayout>
```

```
<!-- compoñente ascendente -->
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>
  <template #default>
    <p>Main content.</p>
    <p>Another paragraph.</p>
  </template>
  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

Para pasar o contido do slot por defecto usarase **<template v-slot:default>** ou **<template #default>.**



Cando un compoñente ten declarados slots por defecto e slots con nome, todo nodo que non estea dentro dun template considérase que é contido do slot por defecto. Polo tanto, o exemplo anterior pode ser reescrito como se indica a continuación. Na columna da dereita indícase como se renderizará o código:

```
<BaseLayout>
  <template #header>
    <h1>Here might be a page title</h1>
  </template>

  <!-- implicit default slot -->
  <p>A paragraph for the main content.</p>
  <p>And another one.</p>

  <template #footer>
    <p>Here's some contact info</p>
  </template>
</BaseLayout>
```

```
<div class="container">
  <header>
    <h1>Here might be a page title</h1>
  </header>
  <main>
    <p>A paragraph for the main content.</p>
    <p>And another one.</p>
  </main>
  <footer>
    <p>Here's some contact info</p>
  </footer>
</div>
```

Paso de información a un slot

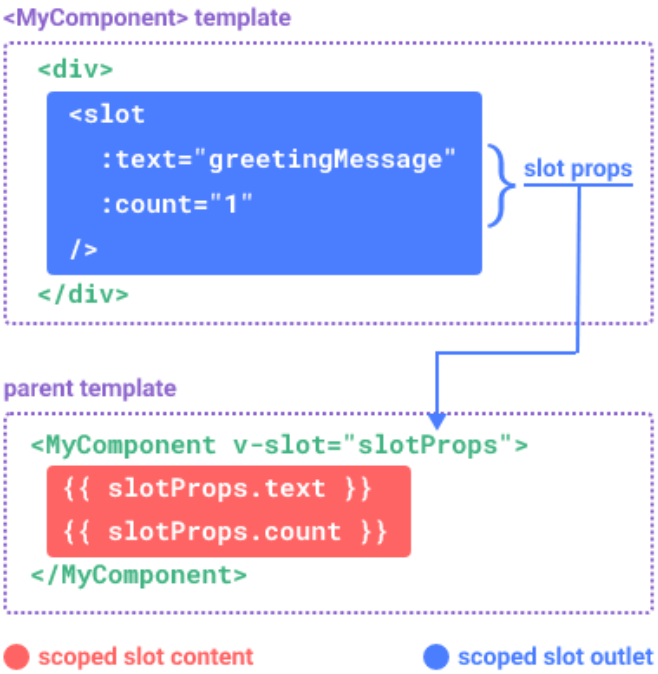
O contido dun slot non ten acceso aos datos do compoñente fillo. Sen embargo, hai casos en que sería útil que o contido dun slot puidese usar datos tanto do compoñente ascendente como do descendente. Para conseguir isto, establécese un mecanismo para que o compoñente descendente poida pasar información ao ascendente para renderizarla. De feito, este mecanismo é igual a como se pasan props a un compoñente:

```
<!-- <MyComponent> -->
<div>
  <slot :text="greetingMessage" :count="1"></slot>
</div>
```

No compoñente ascendente as propiedades recíbense nun obxecto especial que contén todas as propiedades enviadas. Este obxecto estará accesible dende a directiva **v-slot** (pode escollerse calquera nome para este obxecto).

No seguinte exemplo pode verse o código para utilizar o compoñente anterior e recibir as propiedades en **slotProps** no compoñente pai. O valor de slotProps é un **obxecto** con todas as propiedades pasadas ao slot. Ver exempo en funcionamento:

```
<MyComponent v-slot="slotProps">
  {{ slotProps.text }} {{ slotProps.count }}
</MyComponent>
```



Pode usarse **desestruturación de obxectos** co código anterior:

```
<MyComponent v-slot="{ text, count }">
  {{ text }} {{ count }}
</MyComponent>
```

A información que se pasa dende o compoñente fillo pode ser estática (non é unha variable) ou dinámica. Exemplo:

```
<!-- <MyComponent>-->
<template>
  <slot staticText="This text is static" :dynamicText="text"></slot>
</template>

<script>
export default {
  data() {
    return {
      text: 'This text is from the data property',
    };
  },
};
</script>
```

Cando se usan slots con nome, as propiedades que pasa o slot declararanse no compoñente ascendente con **v-slot:name="slotProps"** ou coa sintaxe abreviada **#name="slotProps"**. O nome do slot non será incluído nas propiedades. Exemplo:

```
<!-- BaseLayout.vue -->
<div class="container">
  <header>
    <slot name="header" message="hello"></slot>
  </header>
  <main>
    <slot message="default"></slot>
  </main>
  <footer>
    <slot name="footer" message="footer"></slot>
  </footer>
</div>
```

```
<!-- Compoñente ascendente -->
<BaseLayout>
  <template #header="headerProps">
    {{ headerProps }}
  </template>
  <template #default="defaultProps">
    {{ defaultProps }}
  </template>
  <template #footer="footerProps">
    {{ footerProps }}
  </template>
</BaseLayout>
```

Ver outro exemplo en funcionamento.

Exercicio:

1. Exercicio titorial.

2. Crea unha aplicación que conteña un botón que, ao pulsarse, mostre unha ventá modal (que se creará como un compoñente novo).

   A ventá modal terá un aspecto parecido ao da seguinte imaxe:

   Observa que o compoñente para a ventá modal ten definidos 3 slots con valores por defecto. Se se pasa contido ao slot, mostrarase o contido personalizado.

   Cando se pulse o botón Ok da ventá modal, esta ocultarase.

   Podes partir do seguinte código base para a ventá modal, que configura os estilos CSS para que apareza superposta a calquera contido da páxina:

```
<div class="modal-mask">
  <div class="modal-container">
  </div>
</div>
```

```
.modal-mask {
  position: fixed;
  z-index: 9998;
  top: 0;
  left: 0;
  width: 100%;
  height: 100%;
  background-color: rgba(0, 0, 0, 0.5);
  display: flex;
}

.modal-container {
  width: 300px;
  margin: auto;
  padding: 20px 30px;
  background-color: #fff;
  border-radius: 2px;
  box-shadow: 0 2px 8px rgba(0, 0, 0, 0.33);
}
```

Teleport

O compoñente <Teleport> é un compoñente proporcionado por Vue para teletransportar unha parte da template dun compoñente a outro nodo do DOM, fóra da xerarquía dese compoñente.

O exemplo máis común de uso de **<Teleport>** é a creación dunha ventá modal nun compoñente. Aínda que o código e a lóxica da ventá modal pertenzan a un compoñente, semanticamente non debería estar colocada aí no DOM, pois pode dar problemas ás tecnoloxías asistenciais. Ademais, tamén podería haber problemas coa aplicación dos estilos CSS ao estar a ventá modal aniñada noutros compoñentes.

Observar o DOM creado no exercicio da ventá modal e comprobar que a ventá modal aparece xusto despois do botón no DOM. Semanticamente non debería estar aí colocado. Máis ben, debería estar colocado no **body**.

O compoñente <Teleport> proporciona unha forma de mover un compoñente no DOM. Ten unha propiedade **to** á que hai que pasarlle un selector CSS ou un nodo DOM ao que se engadirá o compoñente. Exemplo:

```
<template>
  <button id="show-modal" …>Show Modal</button>

  <Teleport to="body">
    <!-- use the modal component -->
    <Modal …>
      …
    </Modal>
  </Teleport>
</template>
```

O código anterior permite mover a ventá modal ao compoñente body:

NOTA: o elemento ao que se vai teletransportar o compoñente debe estar no DOM cando se monte o compoñente que se está desprazando. Recoméndase colocalo nun elemento

fóra da aplicación Vue. Se se despraza a un elemento renderizado por Vue, hai que asegurarse de que ese elemento estea montado antes de facer o <Teleport>.

Compoñentes dinámicos

Algunhas veces resulta útil poder cambiar dinamicamente entre compoñentes, como nunha interface con pestanas. Os compoñentes dinámicos permiten implementar esta funcionalidade. É dicir, a partir de certos eventos mostrarase un compoñente e ocultarase outro. Ver exemplo.

Para definir compoñentes dinámicos utilizarase o elemento **<component>** co atributo especial **is.** O valor pasado a **is** será o nome dun compoñente rexistrado:

```
<script>
import Home from './Home.vue';
import Posts from './Posts.vue'
import Archive from './Archive.vue'

export default {
  components: {
    Home,
    Posts,
    Archive
  },
  data() {
    return {
      currentTab: 'Home',
      tabs: ['Home', 'Posts', 'Archive']
    }
  }
}
</script>

<template>
  …
  <!-- Component changes when currentTab changes -->
  <component :is="currentTab"></component>
</template>
```

Cando se cambia entre compoñentes utilizando o mecanismo anterior, o compoñente desmóntase ao ser substituído por outro. Isto provoca que os cambios feitos no seu estado se perdan. Cando o compoñente se volva a mostrar, crearase unha nova instancia co estado inicial.

<KeepAlive> é un compoñente integrado en Vue que permite manter en caché as instancias de compoñentes cando se cambia dinamicamente entre eles. Exemplo:

```
<!-- Inactive components will be cached! -->
<KeepAlive>
  <component :is="activeComponent" />
</KeepAlive>
```

Pode verse un exemplo onde non se utiliza <keepAlive> e se observa que ao cambiar entre compoñentes pérdese o estado almacenado. Tamén está dispoñible o mesmo exemplo con <keepAlive> onde se observa que o estado dos compoñentes se mantén.

Por defecto, <keepAlive> mantén en caché todas as instancias dos compoñentes, aínda que isto pode personalizarse utilizando as propiedades **include** e **exclude**. Ver exemplos:

- Exemplo con include.
- Exemplo con exclude.

```
<KeepAlive include="CompOne,CompThree" exclude="CompTwo">
  <component :is="activeComp"></component>
</KeepAlive>
```

Tamén é posible limitar o número máximo de instancias de compoñentes que poden se cacheados utilizando a propiedade **max** de <keepAlive>. Manteranse en caché as últimas instancias accedidas. Exemplo.

```
<KeepAlive :max="2">
  <component :is="activeComp"></component>
</KeepAlive>
```

Exercicio:

1. Crea un compoñente que conteña un contador e mostre o seu valor. Engade un botón para incrementar o contador.

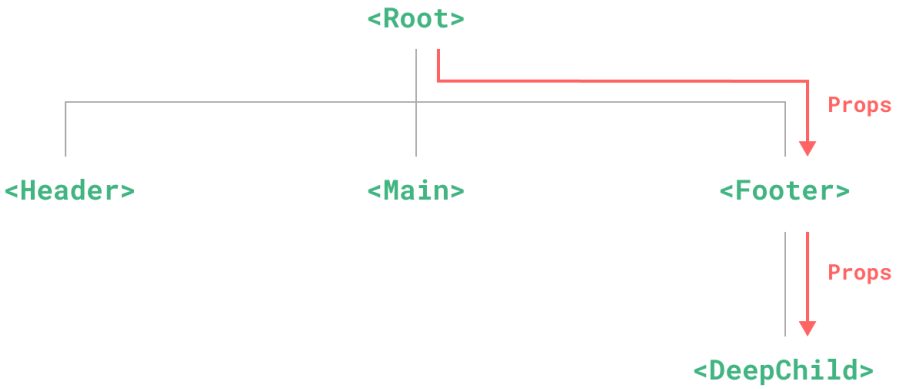
   Crea outro compoñente que teña unha caixa de texto de texto e ademais mostra o valor da caixa de texto noutro elemento HTML.

   Crea unha aplicación raíz que permita, mediante radiobuttons cambiar entre os dous compoñentes anteriores. Fai que o estado do compoñente se manteña aínda que se cambie o compoñente a mostrar.

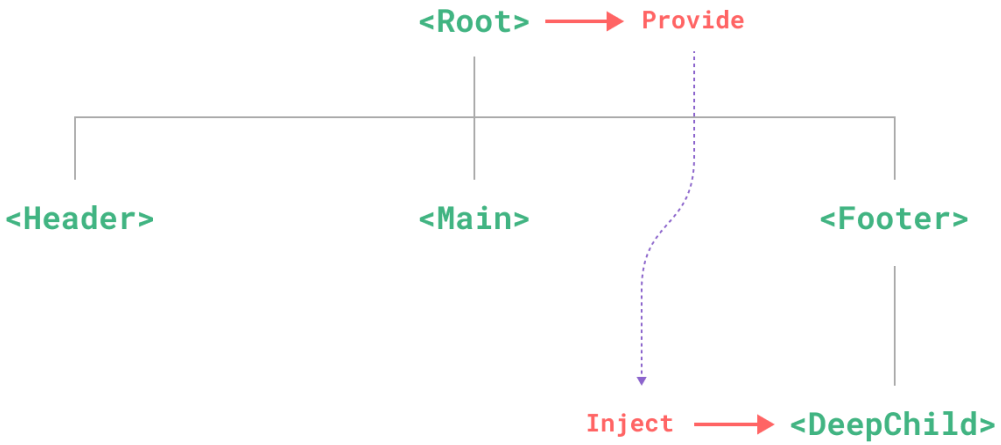
Provide/Inject

Normalmente, cando se pasa información dun compoñente ao seu descendente, utilízanse **props**. Sen embargo, pode darse a situación de ter unha árbore de compoñentes aniñados e necesitar información dun ascendente lonxano. Se se utilizan props, haberá que pasar a

mesma propiedade por toda a cadea. Aínda que os compoñentes intermedios non necesiten esa propiedade, terán que declarala e pasala aos descendentes:



Utilizando **provide** e **inject**, proporciónase unha solución a este problema. Un compoñente ascendente pode funcionar como **provedor de dependencias** para todos os seus descendentes. Calquera compoñente descendente, independente do seu nivel de descendencia, pode inxectar as dependencias proporcionadas polos provedores da súa cadea de ascendentes:



Para proporcionar información aos descendentes utilízase a opción **provide**:

```
export default {
  provide: {
    message: 'hello!'
  }
}
```

Para cada propiedade do obxecto **provide**, a chave é usada polos descendentes para localizar o valor correcto a inxectar.

Se se necesita proporcionar información do estado, por exemplo información declarada en data(), **provide** debe ser unha función:

```
export default {
  data() {
    return {
      message: 'hello!'
    }
  },
  provide() {
    // use function syntax so that we can access `this`
    return {
      message: this.message
    }
  }
}
```

NOTA: O código anterior non fai a inxección reactiva.

Para inxectar a información proporcionada por un compoñente ascendente utilízase a opción **inject**:

```
export default {
  inject: ['message'],
  created() {
    console.log(this.message) // injected value
  }
}
```

NOTA: as inxeccións, ao igual que as props, son de **só lectura**.

A opción **inject** resólvese antes que o estado do propio compoñente, polo que se pode acceder ás propiedades inxectadas dende data():

```
export default {
  inject: ['message'],
  data() {
    return {
      // initial data based on injected value
      fullMessage: this.message
    }
  }
}
```

Exemplo completo en funcionamento de provide + inject.

Cando se usa a **sintaxe de array** para **inject**, a propiedade inxectada está dispoñible no compoñente coa mesma chave coa que é declarada. Se se quere usar unha chave local diferente hai que usar a **sintaxe de obxectos** para **inject.** No seguinte exemplo o compoñente buscará unha propiedade chamada **message** que será usada como **this.localMessage** dentro do compoñente:

```
export default {
  inject: {
    /* local key */ localMessage: {
      from: /* injection key */ 'message'
    },
    count: {},  // se non se quere cambiar o nome á propiedade
  }
}
```

Por defecto, **inject** asume que a propiedade inxectada é proporcionada por algún compoñente ascendente. En caso de que non sexa así, producirase un aviso en tempo de execución. Para evitalo, pode proporcionarse un valor por defecto para inject, similar ao funcionamento dos valores por defecto de props:

```
export default {
  // object syntax is required when declaring default values for injections
  inject: {
    message: {
      from: 'message', // this is optional if using the same key for injection
      default: 'default value'
    }
  }
}
```

Tamén é posible prover funcións en provide para que os compoñentes descendentes poidan invocalas:

```
export default {
  provide() {
    // use function syntax so that we can access `this`
    return {
      imprimir: this.imprimir,
    };
  },
  methods: {
    imprimir() {
      console.log("Imprimir mensaxe " + this.message);
    },
  }
};
```

Para que as inxeccións sexan enlazadas de forma reactiva co provedor é necesario proporcionar unha propiedade calculada usando a función **computed().** Ver exemplo en funcionamento:

```
import { computed } from 'vue'

export default {
  data() {
    return {
      message: 'hello!'
    }
  },
  provide() {
    return {
      // explicitly provide a computed property
      message: computed(() => this.message)
    }
  }
}
```

NOTA: observar a primeira liña onde se importa a función computed para que o código funcione.

Aínda que existe a funcionalidade de provide/inject, poden seguir usándose props para pasar información aos compoñentes. Recoméndase usar provide/inject cando a cadea de comunicación da información é longa.

Exercicio:

1. Crea unha aplicación que permita xestionar unha lista de recursos web.

   Un recurso web é un obxecto coas propiedades id, título, descrición e unha ligazón web.

   A aplicación permitirá listar os recursos almacenados, mostrando en pantalla o título, descrición e ligazón, sobre a que se poderá pulsar para navegar á súa páxina.

   A aplicación tamén debe permitir engadir novos recursos, proporcionando un formulario para introducir todos os seus datos e dar de alta un novo obxecto.

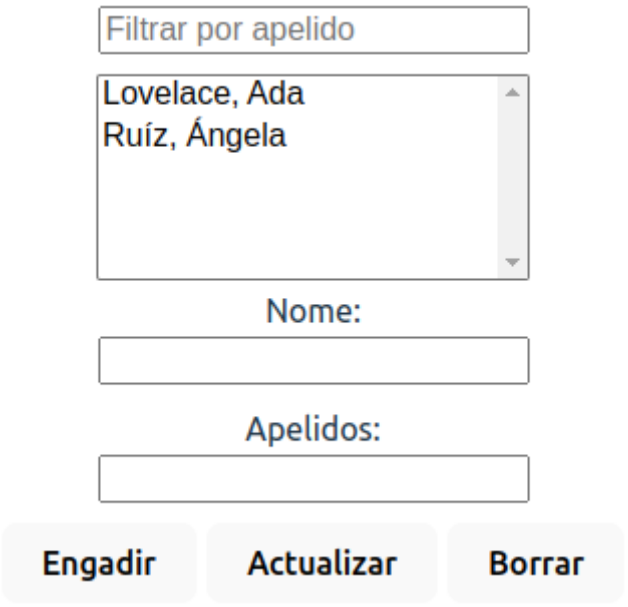
   Utiliza compoñentes dinámicos para seleccionar a funcionalidade a mostrar da aplicación. É dicir, a aplicación debe permitir seleccionar entre mostrar a lista de recursos ou o formulario para engadir un novo recurso.

   Despois de engadir un novo recurso á lista, debe mostrarse en pantalla a lista de recursos.

   No formulario para engadir recursos, comproba que ningún campo está baleiro e no caso de que sexa así mostra unha ventá modal informando da situación. Utiliza <teleport> para mostrar a ventá modal.

   Engade un botón a cada recurso para poder eliminalo.

2. Crea unha aplicación que mostre unha lista de persoas coma a da seguinte imaxe:



   A aplicación debe permitir engadir, actualizar e borrar elementos da lista.

   Ao pulsar sobre un elemento da lista cubriranse automaticamente os campos do formulario co nome e apelidos do elemento seleccionado, permitindo a súa actualización.

   Cando se escribe algo na caixa de texto "Filtrar por apelido", filtraranse os elementos da lista e só se mostrarán aqueles que teñan o apelido que comece polo texto escrito na caixa de texto.

3. Crea unha aplicación que mostre información dos últimos commits dun repositorio de forma similar a como se ve a información dos commits en GitHub.

   A ligazón anterior anterior mostra os últimos commits da rama **main** do repositorio **vuejs/core.** Entre outra información móstrase a mensaxe do commit, o avatar do usuario, o login, información da data do commit e ligazóns ao propio commit.

   GitHub proporciona unha API para consultar información dun repositorio. En concreto, utilizando a URL https://api.github.com/repos/{owner}/{repo}/commits poderá descargarse a información dos commits. Pode personalizarse a petición especificando a rama co parámetro **sha** e tamén limitar o número de commits utilizando o parámetro **per_page.** Así, utilizando a seguinte URL recupereranse os últimos 3 commits da rama **main** do repositorio **core** do usuario **vuejs** de GitHub https://api.github.com/repos/vuejs/core/commits?per_page=3&sha=main.

   Podes empezar facendo unha páxina que consulte a información dun repositorio e rama concreta. Despois, podes engadir un formulario para que a persoa usuaria escriba o login e repositorio a consultar e tamén o nome da rama.

4. Crea unha aplicación Vue que conteña un botón tal que ao pulsalo faga unha petición á API https://random-data-api.com/api/v2/users. Esta API devolve

información aleatoria de usuarios, incluído un avatar. Procesa a información recibida e mostra o avatar e algunha información relevante máis recibida.

Múltiples apps de Vue vs. Múltiples compoñentes

Pode usarse Vue para controlar **partes** de HTML ou para construír SPAs (Single Page Applications).

Cando se controlan múltiples partes independentes de HTML, normalmente úsanse múltiples apps (creadas cada unha co método createApp()).

Por outro lado, cando se crea un SPA, normalmente créase unha app raíz (só se usa unha vez createApp()) e a interface constrúese con múltiples compoñentes.

Aínda que poden usarse compoñentes cando se teñen varias apps, non é o habitual.

As apps de Vue son independentes unhas das outras, non podendo comunicarse entre elas, é dicir, non poden compartir información. Aínda que existen mecanismos para saltar esta limitación, non é a forma “oficial” de facelo.

Os compoñentes, polo contrario, ofrecen certos mecanismos de comunicación que permiten intercambiar información entre eles. Polo que é posible construír unha interface con compoñentes que se intercomunican utilizando unha app raíz que contén múltiples compoñentes.

v-model nun compoñente personalizado

A directiva **v-model** de Vue úsase para implementar unha ligazón bidireccional en elementos de formularios. Así, o seguinte código fai que no input se mostre o valor da propiedade **searchText** e, ademais, cando o valor do elemento do formulario se cambie, a propiedade actualízase automaticamente:

```
<input v-model="searchText" />
```

O código anterior é traducido por Vue polo seu equivalente usando atributos enlazados e manexadores de eventos:

```
<input
  :value="searchText"
  @input="(event) => (searchText = event.target.value)"  />
```

Cando se utiliza **v-model** nun compoñente, pasa algo similar ao anterior. Así, o seguinte código:

```
<CustomInput v-model="searchText" />
```

será traducido por Vue polo seguinte código:

```
<CustomInput
   :modelValue="searchText"
   @update:modelValue="(newValue) => (searchText = newValue)"  />
```

Observar que o código xerado por Vue pasa a propiedade **modelValue** ao compoñente descendente e este emite o evento personalizado **update:modelValue**.

Para que o anterior funcione, o compoñente descendente debe:

- enlazar o atributo value dun elemento <input> coa propiedade **modelValue**.
- cando se lance o evento input, emitir o evento personalizado **update:modelValue**

Ver exemplo:

```
<!-- CustomInput.vue -->
<script>
export default {
  props: ['modelValue'],
  emits: ['update:modelValue']
}
</script>
<template>
  <input
    :value="modelValue"
    @input="$emit('update:modelValue', $event.target.value)"  />
</template>
```

Probar o exemplo anterior en funcionamento.

A directiva **v-model** nun compoñente tamén pode recibir un argumento (title no seguinte exemplo):

```
<MyComponent v-model:title="bookTitle" />
```

Neste caso, en lugar dos valores por defecto para a propiedade (**modelValue)** e o evento (update:modelValue), o compoñente descendente espera recibir unha propiedade denominada **title** e emitir o evento **update:title.** Ver exemplo en funcionamento**:**

```
<!-- MyComponent.vue -->
<script>
export default {
  props: ['title'],
  emits: ['update:title']
}
</script>

<template>
  <input
    type="text"
    :value="title"
    @input="$emit('update:title', $event.target.value)"
  />
</template>
```

Coa utilización de argumentos na directiva v-model é posible pasar múltiples valores para unha mesma instancia dun compoñente. Ver exemplo en funcionamento.

```
<UserName
  v-model:first-name="first"
  v-model:last-name="last"
/>
```

Exercicio:

1. Crea unha aplicación que conteña un compoñente raíz con un formulario.

   Crea un compoñente personalizado que conteña a definición dun control personalizado a incluír no formulario, utilizando como base o seguinte código:

   ```
   <template>
     <ul>
       <li>
         <button type="button">Baixo</button>
       </li>
       <li>
         <button type="button">Medio</button>
       </li>
       <li>
         <button type="button">Alto</button>
       </li>
     </ul>
   </template>
   ```

Cada vez que se pulse sobre un dos botóns no compoñente descendente, debe notificarse ao formulario sobre a opción pulsada.

O formulario utilizará o compoñente personalizado anterior e deberá saber en todo momento que botón se pulsou.

Modifica os estilos do compoñente personalizado para que ao pulsar sobre un botón cambie o seu aspecto. Por exemplo, podes cambiar a cor de texto e do borde, para saber visualmente que elemento está seleccionado.

Cando se envíe o formulario, debe mostrarse por consola a opción seleccionada no compoñente personalizado.

Fases do ciclo de vida

Cada instancia dun compoñente Vue pasa por unha serie de fases durante o seu ciclo de vida, que se corresponden coas distintas operacións que realiza: establecer a observación dos datos, compilar o template, montar a instancia no DOM, actualizar o DOM cando hai cambios nos datos, etc.

Cando o compoñente cambia de fase executa unha función específica, denominada **lifecycle hook,** función na que se pode engadir código personalizado para realizar accións específicas.

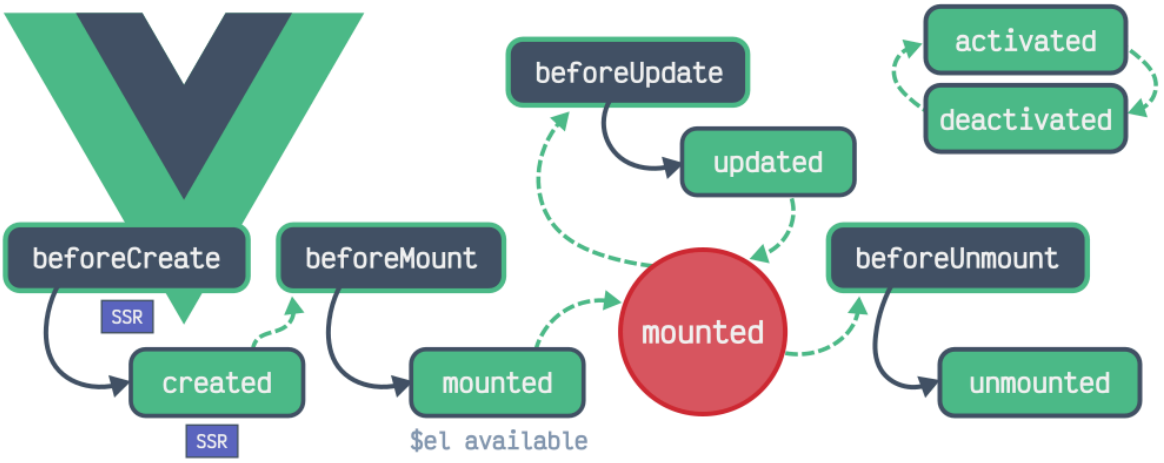
Por exemplo, a función **mounted** pode utilizarse para executar código despois de que o compoñente rematou de facer o renderizado inicial e de crear os nodos do DOM:

```
export default {
  mounted() {
    console.log(`the component is now mounted.`)
  }
}
```

Os **lifecycle hooks** máis utilizados son: **mounted**, **updated** e **unmounted**.

En todas as funcións **lifecycle hooks** pode usarse a palabra reservada **this,** que apunta á instancia activa do compoñente. Isto significa que debe evitarse usar funcións frecha ao declarar funcións de lifecycle hooks, pois dende unha función non se poderá acceder á instancia do compoñente usando **this.**

O seguinte diagrama mostra o ciclo de vida cun compoñente de Vue. O círculo vermello simboliza o momento no que o compoñente está cargado e mostrado na páxina web, é dicir, o compoñente está **montado**.



Ligazón a outra imaxe do ciclo de vida dun compoñente.

As fases do ciclo de vida polas que pasa un compoñente Vue son:

- **beforeCreate**: execútase antes de que se inicialice a instancia do compoñente, polo que non estarán definidas as propiedades do compoñente: data, computed properties, methods e listeners.
- **created**: execútase despois de que a instancia remate de procesar todas as opcións relativas ao seu estado. Cando se execute estarán dispoñibles data, computed properties, methods e watchers. Neste punto poden facerse peticións HTTP para recibir datos ou inicializar valores de data.
- **beforeMount**: execútase inmediatamente antes de montar o compoñente, polo que non están creados os nodos do DOM.
- **mounted**: execútase despois de que o compoñente sexa montado (todos os compoñentes fillos están montados, o DOM creado e insertado no contedor pai). A partir deste punto o compoñente é visible.
- **beforeUpdate**: execútase antes de que o compoñente actualice o DOM debido a un cambio no estado reactivo.
- **updated**: execútase despois de que o compoñente actualizase o DOM debido a un cambio no estado reactivo. **NOTA**: non modificar o estado do componente neste punto, pois produciríase un bucle infinito.
- **beforeUnmount**: execútase antes de que a instancia do compoñente se desmonte e se elimine do DOM.
- **unmounted**: execútase despois de que o compoñentes fose desmontado e eliminado do DOM, xusto cando o compoñente e todos os fillos se desmontaron.
- **activated** e **deactivated**: execútanse cando un compoñente dinámico en caché é engadido ou eliminado. Execútanse cando o compoñente se utiliza con <keepAlive>.

En código, cada unha destas fases maniféstase con unha función especial chamada **hook** de ciclo de vida. Esta función é executada por Vue cando o compoñente se encontra en dita

fase, polo que pode programarse a execución de calquera instrución en cada unha destas funcións, como facer unha petición HTTP, establecer un timer, etc.

```
  <input type="text" ref="userText">
  <button @click="setText">Set text</button>
  <p>{{ message }}</p>
```

```
export default {
  data() {
    return {
      message: "",
    };
  },
  methods: {
    setText() {
      this.message = this.$refs.userText.value;
    },
  },
  beforeCreate() {
    console.log("beforeCreate");
  },
  created() {
    console.log("created");
  },
  beforeMount() {
    console.log("beforeMount");
  },
  mounted() {
    console.log("mounted");
  },
  beforeUpdate() {
    console.log("beforeUpdate");
  },
  updated() {
    console.log("updated");
  },
  beforeUnmount() {
    console.log("beforeUnmount");
  },
  unmounted() {
    console.log("unmounted");
  },
};
```

```
// aínda que non é habitual desmontar un compoñente, faise como exemplo
setTimeout(function () {
  app.unmount();
}, 3000);
```

O código anterior mostra un exemplo de utilización de funcións hook e pode comprobarse na consola a orde na que se executan.

Exercicios:

1. Exercicio titorial.

Referencias

Para a elaboración deste material utilizáronse, entre outros, os recursos que se enumeran a continuación:

- JavaScript | MDN
- Client-side web APIs - Learn web development | MDN
- JavaScript Tutorial - w3schools
- Desarrollo Web en Entorno Cliente | materials
- Javascript en español - Lenguaje JS
- The Modern JavaScript Tutorial
- Eloquent JavaScript
- Vue.js
- Getting started with Vue - Learn web development | MDN
- Vue Mastery
- Vue.js 3 Fundamentals, a FREE Vue.js Course
- Vue CLI