Universidad Tecnológica Nacional

Facultad Regional Córdoba

Ingeniería en Sistemas de Información

Cátedra: Ingeniería de Software

Trabajo Práctico Nº 6:

"Requerimientos Ágiles - Implementación de User Stories"

Curso: 4k2

Docentes:
- Meles, Silvia Judith (Titular)
- Massano, Maria Cecilia (JTP)
- Robles, Joaquin Leonel (Ayudante 1ra)

GRUPO N°4:
- Urzaiz Zuain, Imanol Francisco [78183]
- Lizarralde Bressan, Delfina [69712]
- Fabro, Lorenzo Agustín [70430]
- Barrera, Luciano Martín [69920]
- de la Orden, Lourdes [70262]
- Rubiolo, Santiago Exequiel [67855]

Fecha de Presentación: 8/9/2020

Guía de Estilo

Esta es la guía de estilo oficial para el código específico de Vue. Si usa Vue en un proyecto, es una excelente referencia para evitar errores, "bikeshedding" y antipatrones. Sin embargo, no creemos que ninguna guía de estilo sea ideal para todos los equipos o proyectos, por lo que las desviaciones conscientes se fomentan en función de la experiencia pasada, la tecnología acumulada y los valores personales.

También evitamos las sugerencias sobre JavaScript o HTML en general. No nos importa si usa punto y coma o comas finales. No nos importa si su HTML utiliza comillas simples o comillas dobles para los valores de los atributos. Sin embargo, algunas excepciones existirán cuando descubramos que un patrón particular es útil en el contexto de Vue.
Próximamente, también proporcionaremos consejos para su ejecución. A veces simplemente tendrá que ser disciplinado, pero siempre que sea posible, trataremos de mostrarle cómo usar ESLint y otros procesos automatizados para simplificar el cumplimiento.

Finalmente, hemos dividido las reglas en cuatro categorías:

Categorías

Prioridad A: Esencial

Estas reglas ayudan a prevenir errores, así que apréndalas y cumpla con ellas a toda costa. Pueden existir excepciones, pero deben ser muy raras y sólo las deben realizar aquellas personas con conocimientos de JavaScript y Vue.

Prioridad B: Muy recomendable

Se ha descubierto que estas reglas mejoran la legibilidad y/o la experiencia del desarrollador en la mayoría de los proyectos. Su código aún se ejecutará si no las respeta, pero estas excepciones deben ser raras y estar bien justificadas.

Prioridad C: Recomendado

Donde existen opciones múltiples e igualmente buenas, se puede hacer una elección arbitraria para garantizar la coherencia. En estas reglas, describimos cada opción aceptable y sugerimos una opción predeterminada. Eso significa que puede sentirse libre de hacer una elección diferente en su propia base de código, siempre que sea coherente y tenga una buena razón para ello. ¡Por favor, justifique su elección! Al adaptarse al estándar de la comunidad, usted:

1. Entrena su cerebro para analizar más fácilmente la mayoría del código de comunidad que encuentre
2. Será capaz de copiar y pegar la mayoría de los ejemplos de código de comunidad sin modificación
3. A menudo descubrirá que las nuevas convenciones ya están amoldadas a su estilo de codificación preferido, al menos en lo que respecta a Vue

Prioridad D: Usar con precaución

Algunas características de Vue existen para adaptarse a casos excepcionales o migraciones menos "agresivas" desde una base de código heredada. Sin embargo, cuando se usan en exceso, pueden hacer que su código sea más difícil de mantener o incluso convertirse en una fuente de errores. Estas reglas arrojan luz sobre las características potencialmente riesgosas, y describen cuándo y por qué deberían evitarse.

Reglas de prioridad A: Esencial (prevención de errores)

Nombres de componentes de varias palabras ESENCIAL

Los nombres de los componentes siempre deben ser de varias palabras, a excepción de los componentes raíz de la "App".

Esto **evita conflictos** con elementos HTML existentes y futuros, ya que todos los elementos HTML son una sola palabra.

Incorrecto

```
Vue.component('todo', {
  // ...
})
export default {
  name: 'Todo',
  // ...
}
```

Correcto

```
Vue.component('todo-item', {
  // ...
})
export default {
  name: 'TodoItem',
  // ...
}
```

Componente data ESENCIAL

El componente `data` debe ser una función.

Al usar la propiedad `data` en un componente (es decir, en cualquier lugar excepto en `new Vue`), el valor debe ser una función que devuelve un objeto.

Incorrecto

```
Vue.component('some-comp', {
  data: {
    foo: 'bar'
  }
})
export default {
  data: {
    foo: 'bar'
  }
}
```

Correcto

```
Vue.component('some-comp', {
  data: function () {
    return {
      foo: 'bar'
    }
  }
})
// En un archivo .vue
export default {
  data () {
    return {
      foo: 'bar'
    }
  }
}
// Está bien usar un objeto directamente
// en una instancia raíz de Vue,
// ya que solo existirá esa única instancia.
new Vue({
  data: {
    foo: 'bar'
  }
})
```

Definiciones de Props ESENCIAL

Las definiciones de Props deben ser lo más detalladas posible.

En el código “commiteado”, las definiciones de props siempre deben ser lo más detalladas posible, especificando al menos su(s) tipo(s).

Incorrecto

```
// Esto esta bien solo cuando se prototipa
props: ['status']
```

Correcto

```
props: {
  status: String
}
// Mucho mejor!
props: {
  status: {
    type: String,
    required: true,
    validator: function (value) {
      return [
        'syncing',
        'synced',
        'version-conflict',
        'error'
      ].indexOf(value) !== -1
    }
  }
}
```

Key + v-for ESENCIAL

Siempre use `key` con `v-for`.

`key` con `v-for` es un requisito *perenne* en componentes, para mantener el estado del componente interno en el subárbol. Sin embargo, incluso para los elementos, es una buena práctica mantener un comportamiento predecible, como **constancia del objeto** en las animaciones.

Incorrecto

```
<ul>
  <li v-for="todo in todos">
    {{ todo.text }}
  </li>
</ul>
```

Correcto

```
<ul>
  <li
    v-for="todo in todos"
    :key="todo.id"
  >
    {{ todo.text }}
  </li>
</ul>
```

Evitar v-if con v-for ESENCIAL

Nunca use v-if en el mismo elemento que v-for.

Hay dos casos comunes en los que esto puede ser tentador:

- Para filtrar elementos en una lista (por ejemplo, v-for="user in users" v-if="user.isActive"). En estos casos, reemplace users con una nueva propiedad calculada que devuelva su lista filtrada (por ejemplo activeUsers).
- Para evitar renderizar una lista si debe estar oculta (por ejemplo, v-for="user in users" v-if="shouldShowUsers"). En estos casos, mueva el v-if a un elemento contenedor (por ejemplo,ul, ol).

Incorrecto

```
<ul>
  <li
    v-for="user in users"
    v-if="user.isActive"
    :key="user.id"
  >
    {{ user.name }}
  <li>
</ul>
<ul>
  <li
    v-for="user in users"
    v-if="shouldShowUsers"
    :key="user.id"
  >
    {{ user.name }}
  <li>
</ul>
```

Correcto

```
<ul>
 <li
  v-for="user in activeUsers"
  :key="user.id"
 >
  {{ user.name }}
 <li>
</ul>
<ul v-if="shouldShowUsers">
 <li
  v-for="user in users"
  :key="user.id"
 >
  {{ user.name }}
 <li>
</ul>
```

Scoping de Estilos de Componentes ESENCIAL

Para las aplicaciones, los estilos, en un componente `App` de nivel superior y en los componentes de "layout" pueden ser globales, pero todos los demás componentes siempre deben ser *scoped*

Esto es relevante sólo a **componentes single-file**. *No* requiere que se use el atributo **`scoped`**. El "scoping" podría ser a través de **CSS modules**, una estrategia basada en clases como **BEM**, ú otra biblioteca/convención.
Para las bibliotecas de componentes, sin embargo, se debería implementar una estrategia basada en clases en vez de usar el atributo `scoped`.

Esto hace que el "overriding" de los estilos internos sea más fácil, con nombres de clase legibles que no tienen una especificidad demasiado alta, y menos probable que generen conflictos.

Incorrecto

```
<template>
 <button class="btn btn-close">X</button>
</template>
```

```
<style>
.btn-close {
  background-color: red;
}
</style>
```

Correcto

```
<template>
  <button class="button button-close">X</button>
</template>

<!-- Usando el atributo `scoped` -->
<style scoped>
.button {
  border: none;
  border-radius: 2px;
}

.button-close {
  background-color: red;
}
</style>
<template>
  <button :class="[$style.button, $style.buttonClose]">X</button>
</template>

<!-- Usando CSS modules -->
<style module>
.button {
  border: none;
  border-radius: 2px;
}

.buttonClose {
  background-color: red;
}
</style>
<template>
  <button class="c-Button c-Button--close">X</button>
</template>
```

```
<!-- Usando la convención BEM -->
<style>
.c-Button {
  border: none;
  border-radius: 2px;
}

.c-Button--close {
  background-color: red;
}
</style>
```

Nombres de propiedades privadas ESENCIAL

Siempre use el prefijo `$_` para propiedades privadas en un plugin, mixin, etc. Luego, para evitar conflictos con código de otros desarrolladores, también incluya un *scope* (ej. `$_yourPluginName_`).

Incorrecto

```
var myGreatMixin = {
  // ...
  methods: {
    update: function () {
      // ...
    }
  }
}
var myGreatMixin = {
  // ...
  methods: {
    _update: function () {
      // ...
    }
  }
}
var myGreatMixin = {
  // ...
```

```
  methods: {
    $update: function () {
      // ...
    }
  }
}
var myGreatMixin = {
  // ...
  methods: {
    $_update: function () {
      // ...
    }
  }
}
```

Correcto

```
var myGreatMixin = {
  // ...
  methods: {
    $_myGreatMixin_update: function () {
      // ...
    }
  }
}
```

Reglas de prioridad B: Altamente Recomendadas (Mejorar la legibilidad)

Cada componente es un archivo ALTAMENTE RECOMENDADO

Siempre que un compilador pueda concatenar archivos, cada componente debería estar en su propio archivo.

Esto lo ayudará a usted a encontrar de una manera más rápida un componente cuando precise editarlo o verificar como se utiliza.

Incorrecto

```
Vue.component('TodoList', {
  // ...
})

Vue.component('TodoItem', {
  // ...
})
```

Correcto

```
components/
|- TodoList.js
|- TodoItem.js
components/
|- TodoList.vue
|- TodoItem.vue
```

Notación de nombres de componentes single-file ALTAMENTE RECOMENDADO

Los nombres de los archivos de los componentes single-file deben ser siempre PascalCase o siempre kebab-case.

El autocompletado de los editores de código funciona mejor cuando se utiliza PascalCase, ya que esta es consistente con la forma en que referenciamos componenten en JS(X) y plantillas, dónde sea posible. Sin embargo, nombres de archivos mixtos pueden crear problemas en sistemas de archivos insensibles a las mayúsculas y minúsculas, es por esto que utilizar kebab-case es perfectamente aceptable.

Incorrecto

```
components/
|- mycomponent.vue
components/
|- myComponent.vue
```

Correcto

```
components/
|- MyComponent.vue
components/
|- my-component.vue
```

Nombre de componentes base ALTAMENTE RECOMENDADO

Los componentes base (también conocidos como componentes de presentación, "tontos", o puros) que aplican estilos y convenciones específicas de la aplicación, deben comenzar con un prefijo específico, tal como `Base`, `App` o `V`.

Incorrecto

```
components/
|- MyButton.vue
|- VueTable.vue
|- Icon.vue
```

Correcto

```
components/
|- BaseButton.vue
|- BaseTable.vue
|- BaseIcon.vue
components/
|- AppButton.vue
|- AppTable.vue
|- AppIcon.vue
components/
|- VButton.vue
|- VTable.vue
|- VIcon.vue
```

Nombres de componentes de instancia única ALTAMENTE RECOMENDADO

Componentes que deben tener solamente una única instancia activa deben comenzar con el prefijo `The`, para denotar que solo puede haber una.

Esto no quiere decir que el componente solamente se utiliza en una única página, sino que solamente será utilizado una vez *por página*. Este tipo de componente nunca aceptan propiedades, dado que son específicas para la aplicación, no del contexto de la misma. Si usted encuentra la necesidad de añadir propiedades, puede ser un buen indicio de que se trata de un componente reusable que solamente se utiliza una vez por página, *por ahora*.

Incorrecto

```
components/
|- Heading.vue
|- MySidebar.vue
```

Correcto

```
components/
|- TheHeading.vue
|- TheSidebar.vue
```

Nombres de componentes fuertemente acoplados ALTAMENTE RECOMENDADO

Los componentes hijo que están fuertemente acoplados a su padre deben incluir el nombre del componente padre como prefijo.

Si un componente solo tiene sentido en el contexto de un componente padre, dicha relación debe ser evidente en su nombre. Dado que usualmente los editores organizan los archivos alfabéticamente, esto también deja ambos archivos cerca visualmente.

Incorrecto

```
components/
|- TodoList.vue
|- TodoItem.vue
|- TodoButton.vue
components/
|- SearchSidebar.vue
|- NavigationForSearchSidebar.vue
```

Correcto

```
components/
|- TodoList.vue
|- TodoListItem.vue
|- TodoListItemButton.vue
components/
|- SearchSidebar.vue
|- SearchSidebarNavigation.vue
```

Orden de las palabras en el nombre de los componentes ALTAMENTE RECOMENDADO

Los nombres de los componentes deben comenzar con la palabra de más alto nivel (muchas veces la más general) y terminar con palabras descriptivas.

Incorrecto

```
components/
|- ClearSearchButton.vue
|- ExcludeFromSearchInput.vue
|- LaunchOnStartupCheckbox.vue
|- RunSearchButton.vue
|- SearchInput.vue
|- TermsCheckbox.vue
```

Correcto

```
components/
|- SearchButtonClear.vue
|- SearchButtonRun.vue
|- SearchInputQuery.vue
|- SearchInputExcludeGlob.vue
|- SettingsCheckboxTerms.vue
|- SettingsCheckboxLaunchOnStartup.vue
```

Componentes con cierre automático ALTAMENTE RECOMENDADO

Componentes sin contenido deben cerrarse automáticamenete en componentes single-file, plantillas basadas en *strings*, y JSX - pero nunca en plantillas del DOM.

Los componentes que se cierran automáticamente no solo comunican que no tienen contenido, sino que garantizan que no deben tener contenido. Es la diferencia entre una página en blanco en un libro y una con el texto “Está página fue intencionalmente dejada en blanco”. También, su código es más limpio sin la *tag* de cerrado innecesaria.

Desafortunadamente, HTML no permite que los elementos personalizados se cierren automáticamente - sólo los **official "void" elements**. Es por eso que esta estrategia solo es posible cuando las plantillas Vue son compiladas antes de estar en el DOM, por lo que pueden servir HTML compatible con la especifiación.

Incorrecto

```
<!-- En componentes de un solo archivo, templates basados en string, y JSX -->
<MyComponent></MyComponent>
<!-- En plantillas del DOM -->
<my-component/>
```

Correcto

```
<!-- En componentes de un solo archivo, templates basados en string, y JSX -->
<MyComponent/>
<!-- En plantillas del DOM -->
<my-component></my-component>
```

Notación de nombres de componentes en templates ALTAMENTE RECOMENDADO

En la mayoría de los proyectos, los nombres de los componentes deben ser siempre PascalCase en componentes single-file y plantillas basadas en *string* - pero kebab-case en plantillas del DOM.

PascalCase tiene algunas ventajas sobre kebab-case:

- Editores pueden autocompletar nombres de componentes en plantillas, ya que en JavaScript también se utiliza PascalCase.
- `<MyComponent>` es más distintivo visualmente que un elemento HTML simple que `<my-component>`, porque hay dos caracteres distintos (las dos mayúsculas), en lugar de solo uno (el guión).
- Si usted utiliza cualquier elemento no Vue en sus plantillas, como un componente web, PascalCase asegura que sus componente Vue se mantendrán distinguibles visualmente.

Desafortunadamente, como HTML es insensible a las mayúsculas y minúsculas, plantillas del DOM deben utilizar kebab-case.

También tenga en cuenta que si usted ya ha invertido fuertemente en kebab-case, la consistencia con las convenciones de HTLM y la posibilidad de utilizar ese mismo enfoque en todos sus proyectos puede ser más importante que las ventajas mencionadas anteriormente. En dichos casos, **utilizar kebab-case en todos lados también es aceptable**.

Incorrecto

```
<!-- En componentes single-file y templates basados en string -->
<mycomponent/>
<!-- En componentes single-file y templates basados en string -->
<myComponent/>
<!-- En DOM templates -->
<MyComponent></MyComponent>
```

Correcto

```
<!-- En componentes single-file y templates basados en string -->
<MyComponent/>
<!-- En DOM templates -->
<my-component></my-component>
```

OR

```
<!-- En todos lados -->
<my-component></my-component>
```

Notación de nombres de componentes en JS/JSX ALTAMENTE RECOMENDADO

Los nombres de los componentes en JS/JSX siempre deben ser PascalCase, aunque pueden ser kebab-case dentro de *strings* en aplicaciones más simples, que solo utilizan registro global de componentes a través de `Vue.component`.

Incorrecto

```
Vue.component('myComponent', {
  // ...
})
import myComponent from './MyComponent.vue'
export default {
  name: 'myComponent',
```

```
  // ...
}
export default {
  name: 'my-component',
  // ...
}
```

Correcto

```
Vue.component('MyComponent', {
  // ...
})
Vue.component('my-component', {
  // ...
})
import MyComponent from './MyComponent.vue'
export default {
  name: 'MyComponent',
  // ...
}
```

Componentes con nombres completos ALTAMENTE RECOMENDADO

Nombres de componentes deben tener palabras completas, en vez de abreviaciones.

El autocompletado automático en los editores hace que el costo de escribir nombres largos muy bajo, mientras que la claridad que estos proveen es invaluable. En particular, las abreviaciones poco comunes deben ser evitadas.

Incorrecto

```
components/
|- SdSettings.vue
|- UProfOpts.vue
```

Correcto

```
components/
|- StudentDashboardSettings.vue
|- UserProfileOptions.vue
```

Notación de nombres de propiedades ALTAMENTE RECOMENDADO

Los nombres de propiedades siempre deben utilizar camelCase al declararse, pero kebab-case en plantillas y JSX.

Simplemente estamos siguiendo las convenciones de cada lenguaje. En Javascript, camelCase es más natural, mientras que HTML, kebab-case lo es.

Incorrecto

```
props: {
  'greeting-text': String
}
<WelcomeMessage greetingText="hi"/>
```

Correcto

```
props: {
  greetingText: String
}
<WelcomeMessage greeting-text="hi"/>
```

Elementos multi-atriubto ALTAMENTE RECOMENDADO

Elementos con múltiples atributos deben ocupar múltiples líneas, con un atributo por línea.

En Javascript, dividir objetos que poseen múltiples propiedades en varias líneas es considerado una buena práctica, porque es mucho más fácil de leer. Nuestras plantillas y **JSX** merecen la misma consideración.

Incorrecto

```
<img src="https://vuejs.org/images/logo.png" alt="Vue Logo">
<MyComponent foo="a" bar="b" baz="c"/>
```

Correcto

```
<img
  src="https://vuejs.org/images/logo.png"
  alt="Vue Logo"
>
```

```
<MyComponent
  foo="a"
  bar="b"
  baz="c"
/>
```

Expresiones simples en tempaltes ALTAMENTE RECOMENDADO

Plantillas de componentes deben incluir expresiones simples, con expresiones más complejas refactorizadas en propiedades computadas o métodos.

Las expresiones complejas en sus plantillas los tornan menos declarativos. Debemos enfocarnos en escribir *qué* debe aparecer, no en *cómo* estamos computando dicho valor. También, las propiedades computadas y métodos permiten que el código sea reutilizado.

Incorrecto

```
{{
  fullName.split(' ').map(function (word) {
    return word[0].toUpperCase() + word.slice(1)
  }).join(' ')
}}
```

Correcto

```
<!-- En un template -->
{{ normalizedFullName }}
// La expresión compleja ha sido movida a una propiedad computada
computed: {
  normalizedFullName: function () {
    return this.fullName.split(' ').map(function (word) {
      return word[0].toUpperCase() + word.slice(1)
    }).join(' ')
  }
}
```

Propiedades computadas simples ALTAMENTE RECOMENDADO

Propiedades computadas complejas deben ser separadas en propiedades más simples, siempre que sea posible.

Incorrecto

```
computed: {
  price: function () {
    var basePrice = this.manufactureCost / (1 - this.profitMargin)
    return (
      basePrice -
      basePrice * (this.discountPercent || 0)
    )
  }
}
```

Correcto

```
computed: {
  basePrice: function () {
    return this.manufactureCost / (1 - this.profitMargin)
  },
  discount: function () {
    return this.basePrice * (this.discountPercent || 0)
  },
  finalPrice: function () {
    return this.basePrice - this.discount
  }
}
```

Comillas en los valores de los atributos ALTAMENTE RECOMENDADO

Los valores de atributos HTML no vacíos siempre deben estar dentro de comillas (simples o dobles, la que no sea utilizada en JS).

Si bien los valores de atributos sin espacios no requiren comillas en HTML, esta práctica usualmente conduce a *evitar* espacios, lo que causa que los valores de los atributos sean menos legibles.

Incorrecto

```
<input type=text>
<AppSidebar :style={width:sidebarWidth+'px'}>
```

Correcto

```
<input type="text">
<AppSidebar :style="{ width: sidebarWidth + 'px' }">
```

Abreviación de directivas ALTAMENTE RECOMENDADO

Las abreviaciones de directivas (`:` para `v-bind` y `@` para `v-on:`) deben ser utilizadas siempre o nunca.

Incorrecto

```
<input
  v-bind:value="newTodoText"
  :placeholder="newTodoInstructions"
>
<input
  v-on:input="onInput"
  @focus="onFocus"
>
```

Correcto

```
<input
  :value="newTodoText"
  :placeholder="newTodoInstructions"
>
<input
  v-bind:value="newTodoText"
  v-bind:placeholder="newTodoInstructions"
>
<input
  @input="onInput"
  @focus="onFocus"
>
<input
  v-on:input="onInput"
  v-on:focus="onFocus"
>
```

Reglas de prioridad C: Recomendadas (Minimizando Elecciones Arbitrarias y Sobrecarga Cognitiva)

Orden de las opciones en un componente/instancia RECOMENDADO

Opciones de componentes/instancias deben ser ordenadas consistentemente.

Este es el orden que recomendamos para las opciones de los componentes. Estas están separadas en categorías, así usted sabe dónde agregar nuevas propiedades de *plugins*.

1. **Efectos colaterales** (desencadenan efectos fuera del componente)
   - el

2. **Consciencia Global** (requiere conocimiento más allá del componente)
   - name
   - parent

3. **Tipo de componente** (cambia el tipo del componente)
   - functional

4. **Modificadores de template** (cambia la forma en que el template es compilado)
   - delimiters
   - comments

5. **Dependencias de template** (*assets* utilizados en el template)
   - components
   - directives
   - filters

6. **Composición** (mezcla propiedades en las opciones)
   - extends
   - mixins

7. **Interfaz** (la interfaz del componente)
   - inheritAttrs
   - model
   - props/propsData

8. **Estado Local** (propiedades reactivas locales)
   - data
   - computed

9. **Eventos** (*callbacks* disparados por eventos reactivos)
   - watch
   - Eventos del ciclo de vida (en el orden que son invocados)
     - beforeCreate
     - created
     - beforeMount
     - mounted
     - beforeUpdate
     - updated
     - activated
     - deactivated
     - beforeDestroy
     - destroyed

10. **Propiedades no reactivas** (propiedades de instancia independientes del sistema de reactividad)
    - methods

11. **Renderización** (la descripción declarativa de la salida del componente)
    - template/render
    - renderError

Order de los atributos en un elemento RECOMENDADO

Orden de los atributos de elementos (incluyendo componentes) deben ser ordenadas consistentemente.

Este es el orden que recomendamos para las opciones de los componentes. Estas están separadas en categorías, así usted sabe dónde agregar nuevos atributos y directivas.

1. **Definición** (provee las opciones del componente)
   - `is`

2. **Renderización de Listas** (crea múltiples variaciones del mismo elemento)
   - `v-for`

3. **Condicionales** (sí el elemento es renderizado/mostrado)
   - `v-if`
   - `v-else-if`
   - `v-else`
   - `v-show`
   - `v-cloak`

4. **Modificadores de renderizado** (cambia la forma en la que el elemento se renderiza)
   - `v-pre`
   - `v-once`

5. **Consciencia Global** (requiere conocimiento más allá del componente)
   - `id`

6. **Atributos Únicos** (atributos que requieren valores únicos)
   - `ref`
   - `key`
   - `slot`

7. **Vinculación Bidireccional** (combinación de vinculación y eventos)
   - `v-model`

8. **Otros Atributos** (todos los no especificados)

9. **Eventos** (*event listeners* del componentes)
   - `v-on`

10. **Contenido** (sobreescribe el contenido del elemento)
    - `v-html`
    - `v-text`

Líneas vacías en las opciones de un componente/instancia RECOMENDADO

Usted puede añadir una línea vacía entre propiedades que se extienden en múltiples líneas, particularmente si las opciones no entrar en la pantalla sin *scrollear*.

Cuando un componente comienza a sentirse apretado o difícil de leer, añadir espacios entre propiedades que se extienden en múltiples líneas puede mejorar la legibilidad. En algunos editores, como Vim, opciones de formateo como esta pueden facilitar la navegación con el teclado.

Correcto

```
props: {
  value: {
    type: String,
    required: true
  },

  focused: {
    type: Boolean,
    default: false
  },

  label: String,
  icon: String
},

computed: {
  formattedValue: function () {
    // ...
  },

  inputClasses: function () {
    // ...
  }
}
// No tener espacios también es correcto, siempre y cuando
// el componente siga siendo fácil de leer y navegar.
props: {
```

```
  value: {
    type: String,
    required: true
  },
  focused: {
    type: Boolean,
    default: false
  },
  label: String,
  icon: String
},
computed: {
  formattedValue: function () {
    // ...
  },
  inputClasses: function () {
    // ...
  }
}
```

Orden de los elementos de nivel superior de un componente single-file RECOMENDADO

Componentes single-file siempre deben ordenar las etiquetas <script>, <template>, y <style> consistentemente, con <style> por último, ya que al menos una de las otras dos siempre es necesaria.

Incorrecto

```
<style>/* ... */</style>
<script>/* ... */</script>
<template>...</template>

<!-- ComponentA.vue -->
<script>/* ... */</script>
<template>...</template>
<style>/* ... */</style>

<!-- ComponentB.vue -->
<template>...</template>
<script>/* ... */</script>
<style>/* ... */</style>
```

Correcto

```
<!-- ComponentA.vue -->
<script>/* ... */</script>
<template>...</template>
<style>/* ... */</style>

<!-- ComponentB.vue -->
<script>/* ... */</script>
<template>...</template>
<style>/* ... */</style>
<!-- ComponentA.vue -->
<template>...</template>
<script>/* ... */</script>
<style>/* ... */</style>

<!-- ComponentB.vue -->
<template>...</template>
<script>/* ... */</script>
<style>/* ... */</style>
```

Reglas de prioridad D: Utilizar con Precaución (Patrones Potencialmente Peligrosos)

v-if/v-else-if/v-else sin key UTILIZAR CON PRECAUCIÓN

It's usually best to use key with v-if + v-else, if they are the same element type (e.g. both elements).

Por defecto, Vue actualiza el Dom de la manera más eficiente posible. Esto significa que, al cambiar entre elementos del mismo tipo, simplemente actualiza el elemento existente, en lugar de removerlo y añadir uno nuevo en su lugar. Esto puede traer **efectos colateral no deseados** si realmente estos elementos no deben ser consideramos el mismo.

Incorrecto

```
<div v-if="error">
 Error: {{ error }}
</div>
<div v-else>
 {{ results }}
</div>
```

Correcto

```
<div
 v-if="error"
 key="search-status"
>
 Error: {{ error }}
</div>
<div
 v-else
 key="search-results"
>
 {{ results }}
</div>
<p v-if="error">
 Error: {{ error }}
</p>
<div v-else>
 {{ results }}
</div>
```

Selector de elemento con `scoped` UTILIZAR CON PRECAUCIÓN

Los selectores de elementos deben ser evitados con `scoped`.

Prefiera selectores de clase sobre selectores de elementos en estilos `scoped`, ya que grandes números de selectores de elementos son lentos.

Incorrecto

```
<template>
 <button>X</button>
</template>

<style scoped>
button {
 background-color: red;
}
</style>
```

Correcto

```
<template>
  <button class="btn btn-close">X</button>
</template>

<style scoped>
.btn-close {
  background-color: red;
}
</style>
```

Comunicación implícita entre componentes padre-hijo UTILIZAR CON PRECAUCIÓN

Se debe preferir el uso de *props* y eventos para la comunicación entre componentes padre-hijo, en lugar de this.$parent o mutación de *props*.

Una aplicación Vue ideal es usada con *props* para abajo y eventos para arriba. Apegarse a esta convención hace que sus componentes sean más fáciles de entender. Sin embargo, hay casos border donde la mutación de *props* o el uso de this.$parent puede simplificar dos componentes que están fuertemente acopladas.

El problema es que también existen muchos casos *simples* donde estos patrones pueden ofrecer conveniencia. Cuidado: no se deje seducir por la conveniencia a corto plazo (escribir menos código) sobre la simpicidad (poder entender el flujo de su estado).

Incorrecto

```
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }
  },
  template: '<input v-model="todo.text">'
})
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }},
```

```
  methods: {
    removeTodo () {
      var vm = this
      vm.$parent.todos = vm.$parent.todos.filter(function (todo) {
        return todo.id !== vm.todo.id
      })
    }
  },
  template: `
    <span>
      {{ todo.text }}
      <button @click="removeTodo">
        X
      </button>
    </span>
  `
})
```

Correcto

```
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }
  },
  template: `
    <input
      :value="todo.text"
      @input="$emit('input', $event.target.value)"
    >
  `
})
Vue.component('TodoItem', {
  props: {
    todo: {
      type: Object,
      required: true
    }
```

```
  },
  template: `
    <span>
      {{ todo.text }}
      <button @click="$emit('delete')">
        X
      </button>
    </span>
  `
})
```

Manejo de estado sin utilizar flux UTILIZAR CON PRECAUCIÓN

Se debe preferir el uso de Vuex para el manejo del estado global, en vez de this.$root o un event bus global.

Manejar el estado en this.$root y/o utilizado un ***event bus*** **global** puede ser conveniente para casos simples, pero no son apropiados para la mayoría de las aplicaciones. Vuex no solo ofrece un lugar central para manejar el estado, sino que también ofrece herramiento para organizar, rastrear y depurar cambios de estado.

Incorrecto

```
// main.js
new Vue({
  data: {
    todos: []
  },
  created: function () {
    this.$on('remove-todo', this.removeTodo)
  },
  methods: {
    removeTodo: function (todo) {
      var todoIdToRemove = todo.id
      this.todos = this.todos.filter(function (todo) {
        return todo.id !== todoIdToRemove
      })
    }
  }
})
```

Correcto

```
// store/modules/todos.js
export default {
  state: {
    list: []
  },
  mutations: {
    REMOVE_TODO (state, todoId) {
      state.list = state.list.filter(todo => todo.id !== todoId)
    }
  },
  actions: {
    removeTodo ({ commit, state }, todo) {
      commit('REMOVE_TODO', todo.id)
    }
  }}
<!-- TodoItem.vue -->
<template>
  <span>
    {{ todo.text }}
    <button @click="removeTodo(todo)">
      X
    </button>
  </span>
</template>

<script>
import { mapActions } from 'vuex'

export default {
  props: {
    todo: {
      type: Object,
      required: true
    }},
  methods: mapActions(['removeTodo'])
}
</script>
```