Vue Best Practices

Best Practices

Motivation

Let us see some best practices

Best Practices

In the following slides are a handful best practices which can be useful in your day-to-day developer life.

Always use :key inside v-for

Using the key attribute with the v-for directive helps your application be constant and predictable whenever you want to manipulate the data.

```
<template>
    <!-- BAD -->
<div v-for="product in products">{{ product }}</div>

<!-- GOOD! -->
<div v-for="product in products" :key="product.id">{{ product }}</div>
</template>
```

Use kebab-case for events

When it comes to emitting custom events, it’s always best to use kebab-case.

This is because in the parent component, it is the same syntax we used to listen to that event (consistency).

```
this.$emit("close-window");
```

Props camelCase / kebab-case in templates

This best practice simply just follows the conventions for each language. **In JavaScript, camelCase is the standard and in HTML, it's kebab-case.**

Luckily for us, **VueJS converts between kebab-case and camelCase for us**.

```
<template>
  <PopupWindow title-text="hello world" />
</template>
<script>
export default {
  props: {
    titleText: String,
  },
}
</script>
```

Props camelCase / kebab-case in templates

This best practice simply just follows the conventions for each language. **In JavaScript, camelCase is the standard and in HTML, it's kebab-case.**

Luckily for us, **VueJS converts between kebab-case and camelCase for us**.

```
<template>
  <PopupWindow title-text="hello world" />
</template>
<script>
export default {
  props: {
    titleText: String,
  },
}
</script>
```

Data should always return a function

When declaring component data, the data option should always return a function.

If it does **not**, and we just simply return an object, then that **`data will be shared across all instances of the component.`**

Bad:

```
export default {
    data: {
        name: "My Window",
        articles: [],
    },
};
```

Good:

```
export default {
    data() {
        return {
            name: "My Window",
            articles: [],
        };
    },
};
```

Don’t call a method on created AND watch

Common mistake Vue developers make is they unnecessarily call a method in **`created and watch`**.

The thought behind this is that we want to run the watch hook as soon as a component is initialized.

```
<script>
// BAD!
  export default {
   created: () {
    this.handleChange()
   },
   methods: {
    handleChange() {
     // stuff happens
    }
   },
   watch () {
    property() {
     this.handleChange()
    }
   }
  }
</script>
```

Don’t call a method on created AND watch

All we have to do is restructure our watcher a little bit and declare two properties:

- handler (newVal, oldVal) – this is our watcher method itself
- immediate: true – this makes our handler run when our instance is created

```
<script>
export default {
 methods: {
  handleChange() {
   // stuff happens
  }
 },
 watch () {
  property {
   immediate: true
   handler() {
    this.handleChange()
   }
  }
 }
}
</script>
```

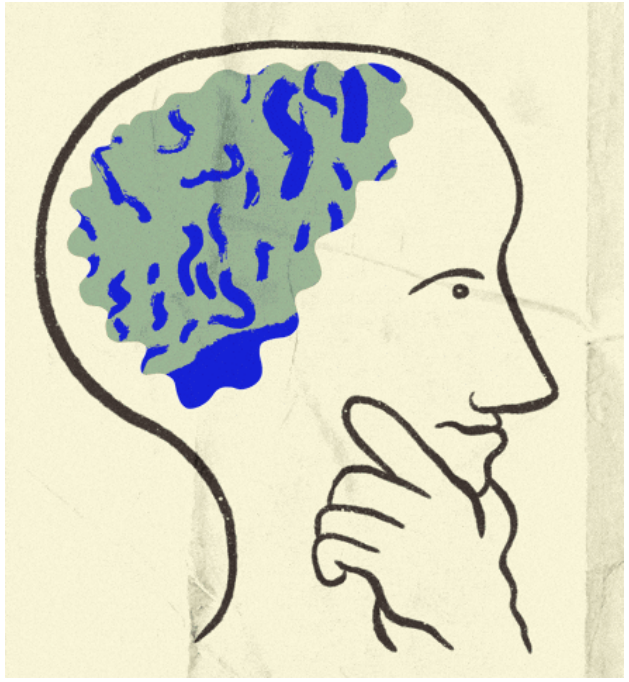
Don’t call a method on created AND watch

```
<script>
  export default {
   created: () {
    this.handleChange()
   },
   methods: {
    handleChange() {
     // stuff happens
    }
   },
   watch () {
    property() {
     this.handleChange()
    }
   }
  }
</script>
```

```
<script>
export default {
 methods: {
  handleChange() {
   // stuff happens
  }
 },
 watch () {
  property {
   immediate: true
   handler() {
    this.handleChange()
   }
  }
 }
}
</script>
```

Patterns & Antipatterns

Next, let us talk about some lesser known patterns worth knowing.



Cheap Static Components with “v-once”

Rendering plain HTML elements is very fast in Vue, but sometimes you might have a component that contains a lot of static content.

```
Vue.component('terms-of-service', {
  template: `
    <div v-once>
      <h1>Terms of Service</h1>
      ... a lot of static content ...
    </div>
  `
})
```

Recursive Components

Components can recursively invoke themselves in their own template. However, they can only do so with the name option.

If you’re not careful, recursive components can also lead to infinite loops:

```
1 name: 'stack-overflow',
2 template: '<div><stack-overflow></stack-overflow></div>'
```

Smarter watchers

Imagine we are working in a search input component, so let's say we want to fetch on created, then watch the input.

```
created() {
    this.fetchUserList()
},
watch: {
    searchText() {
        this.fetchUserList()
    }
}
```

Smarter watchers

We can improve this code by making the watcher accept method names as strings making it look like this:

```
created() {
    this.fetchUserList()
},
watch: {
    searchText: 'fetchUserList'
}
```

This reduces a few lines and makes the code cleaner. 🤗

Smarter watchers

The next improvement we can make is by making the watcher call themselves on created:

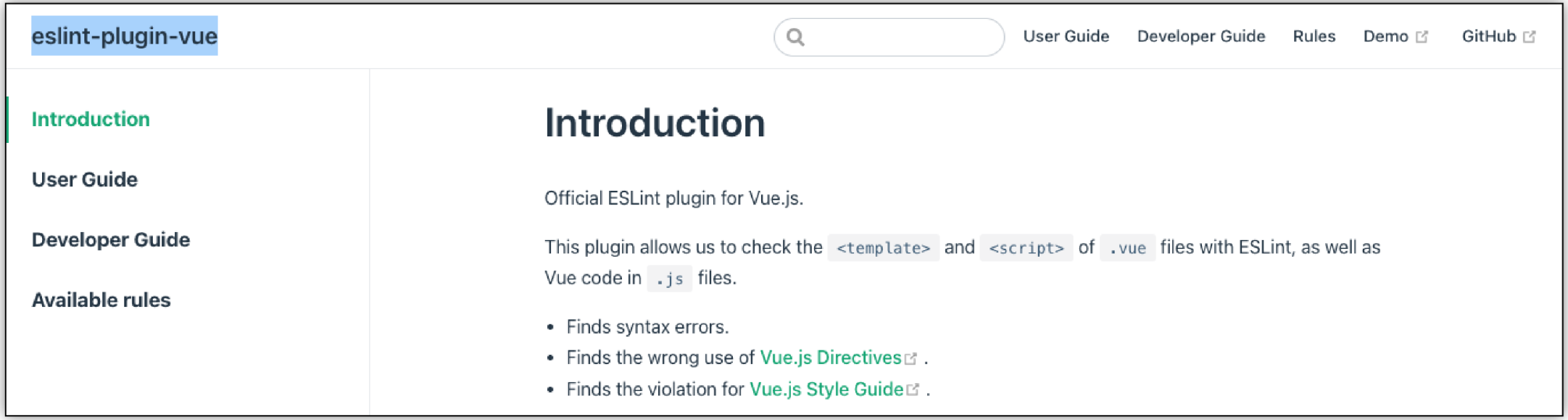
```
watch: {
    searchText: {
        handler: 'fetchUserList',
        immediate: true
    }
}
```

Smarter watchers

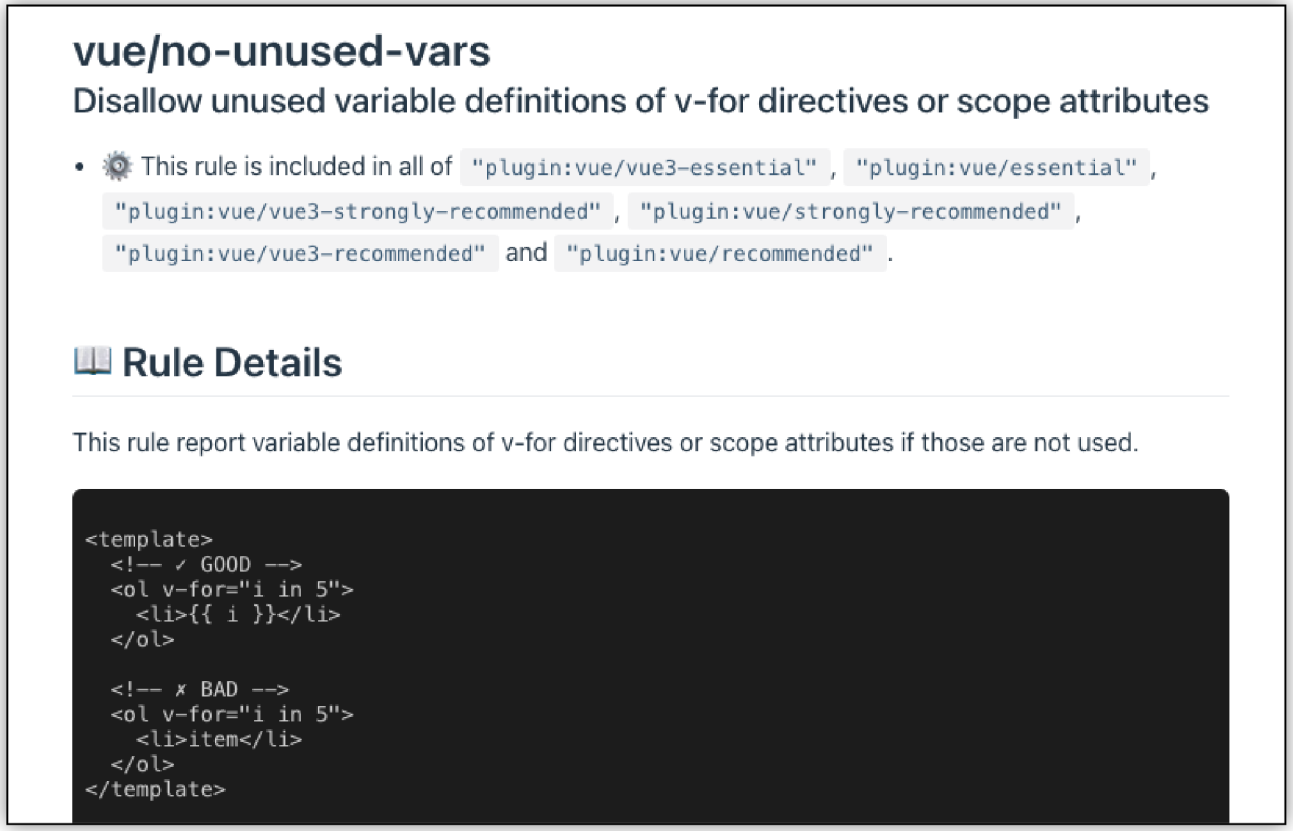
Let's analyze some topics:

- handler: Is the function (or string that has the name of the method) that we want to call.
- immediate: When true means that we don't need to use created hook anymore because the handler will be called as soon as the component is ready, it will be immediately.

eslint-plugin-vue



eslint-plugin-vue



vue/no-unused-vars

Disallow unused variable definitions of v-for directives or scope attributes

- ⚙️ This rule is included in all of `"plugin:vue/vue3-essential"`, `"plugin:vue/essential"`, `"plugin:vue/vue3-strongly-recommended"`, `"plugin:vue/strongly-recommended"`, `"plugin:vue/vue3-recommended"` and `"plugin:vue/recommended"`.

📖 Rule Details

This rule report variable definitions of v-for directives or scope attributes if those are not used.

```
<template>
  <!-- ✓ GOOD -->
  <ol v-for="i in 5">
    <li>{{ i }}</li>
  </ol>

  <!-- ✗ BAD -->
  <ol v-for="i in 5">
    <li>item</li>
  </ol>
</template>
```

End

End

That was all for this chapter