Introduction

One of the best things that modern JavaScript frameworks like Vue allow us to do is break down our websites and applications into manageable components. However, as our sites and apps get bigger and scale, it is easy to fall into certain pitfalls that will make managing your components difficult.

In this course, I will be showing you a variety of best practices and techniques that will help you:

1. Create components that are easier to use and maintain

2. Avoid common mistakes that make components more prone to bugs

3. Feel empowered to make architectural decisions that are best for your app.

In this course, I will be sharing a variety of best practices and techniques for helping you manage your components. However, before we progress any farther, there are three things I need you to understand before you progress through this course.

1. These are guidelines. Not rules.

2. Your opinion and experience matter too.

3. Choose what works best for you and your team.

In other words, if you feel like you have good reason to believe a best practice or technique is not a good fit for your app, then you should trust your instincts and move forward with your solution. Sometimes a technique or best practice that might work well in many contexts can actually be an anti-pattern given another context.

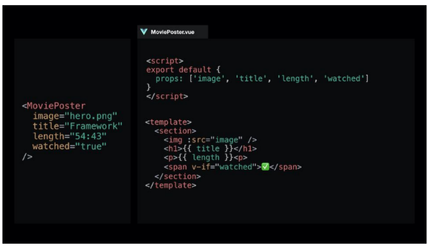
As a result, the important thing is that you understand the potential trade-offs so you can make the best decision possible for your application.

With that said, I look forward to sharing this knowledge with you. See you inside the course!

Props: Fundamentals

Introduction

When it comes to passing information down from one component to another, props are the first technique that most Vue developers encounter. By defining a `props` property on a component, it can then receive data.



Defining Props: Best Practices

The Array Syntax

When defining props, many developers are initially exposed to the Array syntax.

```
<script>
export default {
  name: 'Movie',
  props: ['title', 'length', 'watched']
}
</script>
```

```
<template>
	<section>
		<h1>{{ title }}</h1>
		<p>{{ length }} <span v-if="watched">✅</span><p>
	</section>
</template>
```

While there is nothing wrong with this definition and will work in production, there are some pitfalls that exists with this method:

- Although the prop names may seem intuitive based on its meaning, it leaves it open for interpretation which can often lead to bugs
- For example, what happens when a developer forgets to include a prop that is technically required for the component to render properly? A Movie component without a title would look rather silly wouldn't it?
- Another issue that arises is that the definition of each prop is vague. In the example we have above, should `length` be a number? Should it be a formatted string (i.e., `1:28`)? But then which format should it be (i.e., `1 hr 28 min`)
- And when registering whether the movie has been watched before, what's the proper way to tell it's been watched (e.g., Yes? Y? Watched? true?)

Let's not forget that this component is only responsible for rendering the props. Can you imagine what kind of bugs would show up if more complex logic was involved? 😱

The Object Syntax

Instead, for most scenarios, we should define our props using the Object syntax. This allows us to define three key prop attributes that allow you to answer three fundamental questions:

- `type` : What data type(s) can be expected?
- `required` : Is the prop is required or not
- `default` : Is there default content that accounts for most scenarios so we don't have to repeat ourselves multiple time?

```
<script>
export default {
	props: {
		length: {
			type: Number,
			required: true,
			default: 90
		}
	}
}
</script>
```

While you are probably aware of some of the basic JavaScript data types:

- String
- Number
- Boolean
- Array
- Object

There are a few more that you should know about!

- Date
- Function
- Symbols

And if you want to define multiple data types, it's as simple as using an Array syntax!

```
<script>
export default {
	props: {
		length: {
			type: [Number, String],
			required: true,
			default: 90
		}
	}
}
</script>
```

Some of you might also be thinking, "If there is a `default` property defined, is a `required` property really needed?" And you'd be right! The reality is that when you have a default prop, you don't need the required prop.

```
<script>
export default {
	props: {
		length: {
			type: Number,
			default: 90
		}
	}
}
</script>
```

Conclusion

To review, the Array syntax methodology can be prone to bugs as the application scales, but this is a completely valid method that can be very useful when prototyping and such. However, whenever possible, it's considered a best practice to define your props using the Object syntax.

Finally, when defining your props, remember to answer the three fundamental questions:

- What data type(s) can be expected?
- Is the prop required?
- Can you provide default data to account for most scenario?

Remember that props are useful for providing detailed specifications on how to use a component, but this is also it's downside as this does not allow flexibility on the developer's part.