Why Animate

You've probably heard that web animations can improve the user experience of your Vue application. But do you know what you should be animating, and how to do that in Vue? In Vue Mastery's **Animating Vue** course, we'll explore what makes an effective web animation and learn the simplest ways to start animating your interfaces so your users can better navigate, and enjoy, your application.

Before learning *how* to animate in Vue, let's understand *why* we should, by exploring how we can use animation to improve our user experience.

Directing Focus

In today's world, the human mind is constantly bombarded by information. We're sending emails, receiving texts, scrolling through feeds… our attention is pulled in an endless amount of directions. So when a user arrives at your app, it's likely their brain is already swirling with information they've been processing throughout the day. Our job, as user interface builders, is to quickly orient and direct our users' attention, guiding them on how to effectively use our app.

We can harness the power of animation to focus our user's attention. And once we have it, we can direct it where we want it to go.

Inspiring Action

What is the first thing you want your users to do when they reach your application? It may be a line of copy you want them to read, or a button you want them to click… some first step you want them to take. By effectively utilizing animation, you can remove distraction and inspire that specific action.

In this example, our call to action is a button we want the user to click. By using motion, we have removed distraction and eliminated any confusion in the user's mind about where they should be focusing, and how they should start engaging with our app.

The reason motion is so effective at drawing attention is due to a primal human instinct. Whether for hunting down dinner, or avoiding becoming another animal's dinner, visual sensitivity to motion is a core process of the human brain that has helped us survive and evolve throughout our time here on this planet.

As developers, we can tap into this sensitivity to visual motion in order to direct our users' attention to the elements we want them focusing on so they're more likely to take the action we want them to take. But with great power comes great responsibility, so we should be using motion in a graceful way that mimics nature.

Creating Flow

If I begin telling you what the next reason to animate is, and I end up telling you there are 100 billion stars in our galaxy, that's called a *non sequitur*. Where I ended up abruptly shifted away from where I started. Unfortunately, non sequiturs are very common in poorly designed web interfaces. What does that look like? Have you ever clicked on a button or link, and everything you were just looking at disappeared in a flash, only to be replaced with an entirely new page? This can be disorienting and cause your user to have to re-orient to your site every time something like this happens. Over time, this causes cognitive fatigue in your users.

Instead of breaking context, we can harness the power of animation to create a seamless flow as the user navigates around our application. We can morph elements into one another, we can transition between pages and components in a way that feels natural.

We can show the user where that menu they just had open went, by animating it into its collapsed state. We can use animation to create a sense of physical space and time within or application, where elements transition between positions and states over a predictable duration. When this is implemented well, the user gains a sense of familiarity with the "world" of our app. They understand where things live within it so they better remember how to find what they need when they need it, and they understand how the app behaves so they don't get disoriented , frustrated, or exhausted by it.

Instead, they can be delighted, which brings me to the last major reason I believe we ought to be animating our interfaces.

Delight & Surprise

With an endless amount of apps out there, many of which may be rather routine and dull, our users will appreciate thoughtful touches that make your app more fun and pleasing to use. If you make your user smile, you've just gained a fan.

These thoughtful touches could be anything from a customized loading spinner, to a super clean and helpful form that eases the burden of filling in a bunch of fields, to a button that communicates a success or error state.

When done well, these delightful touches will make your app more memorable, can differentiate you from your competition, and can even help you hold your user's attention for longer than if you hadn't gone that extra mile. But it's worth sharing a word of caution here: it is easy to go overboard and animate *too much*. Animations are like adding spice to a dish. Too little, and the dish is bland; too much, and the dish is ruined. Like any good meal, an effectively animated interface must be well balanced.

Let's ReVue

In the following lessons, we'll look at several ways we can harness the power of Vue to create animations that just about any web app can use. While there are many ways to animate in Vue and it can get quite advanced and complex, the goal of this course isn't to show you every one of the many options out there. The goal is to provide you with the tools and insight you'll need to start adding simple, practical animations to your Vue apps that direct focus, inspire action, create flow, and delight and surprise.

Transitions

Now that we understand the value of animating our interfaces, we're almost ready to start coding. But first, we need to understand how Vue's **transition** component works, then we'll build our first simple transition animation.

Vue Transitions

Before we begin, it's worth mentioning that depending on whom you ask, they might say that transitions and animations are categorically different things. But for the sake of this course, we'll consider a transition to be a simple animation.

A transition is exactly what it sounds like. It's when something *transitions* from _____ to _____. From *off* the screen to *on*, from *here* to *there*, from *open* to *closed*, from *visible* to *invisible*, and so on. As such, transitions can provide the user visual feedback about how something is changing.

In Vue, we use the built-in `transition` component, which serves as a wrapper that gives us classes we can hook into during the lifecycle of the transition. Let's explore how we harness those classes to define entering and leaving transitions.

Default Style

When designing a transition with Vue's `transition` component, we'll first want to ask ourselves this question: **What should the default style be?** In other words, how should the element/component appear when it is *not* transitioning?

To make this more clear, let's look at a simple example, where we want a box to transition on and off the screen and fade its visibility as it transitions.

What should its default style be? How should it appear when it is not transitioning? In this case, we'd want it to be `opacity: 1` by default because that is the style we are transitioning *to* and *away from* as the visibility fades in and fades out.

You may be aware that an HTML element's default style is already `opacity: 1` So in this case, we don't need to define a default style for the element we are transitioning, since `opacity: 1` is already defined for us by the browser. Actually, in the majority of cases you won't need to define the default style because that style is already how the element will appear. Imagine if you were transitioning an element's scale from `10%` to `100%` An element's scale is already `100%` by default, so we wouldn't need to bother explicitly defining that again.

There are caveats to this, where we need to set a default style that is different from how the browser would style the element by default, but we'll get to that later.

Once we are clear on what the element (or component's) default style is, we can then make use of the `transition` component's built-in classes to design the transition *to* and *away from* that default style. In other words: we can build our **entering** and **leaving** transitions.

Entering Transition

The next question we ought to ask ourselves when designing a transition is: **What should the starting style be?**

Since we want the square to go from invisible to visible, we'll need it to start at `opacity: 0`. We put that style inside of the `v-enter` class, to set the transition's **starting** style.

```
.v-enter { /* starting style */
    opacity: 0;
}
```

Now, when the element enters the DOM, it will start off completely invisible and then transition away from that style (`0`) and towards our default style (`1`). This brings us to the next question we should ask ourselves when defining a transition: **What should the active style be?**

In this case, we need to know: How long should the transition take? Should we speed up or slow down the transition during the course of the transition? We use the `v-enter-active` class to define the behavior of the transition while it is *active*, while it is *happening*, specifying things like its duration and easing. If easing functions are new to you, you can learn more about them here.

```
.v-enter { /* starting style */
    opacity: 0;
}

.v-enter-active { /* active entering style */
    transition: opacity 2s ease-in;
}
```

Here, we've specified that we are transitioning the `opacity` property and set the duration of the transition to last `2s` and gave it an `[ease-in](https://cubic-bezier.com/#.42,0,1,1)` curve, meaning it will fade in slowly then get faster as it approaches `1`.

Leaving Transition

Now that our box is on the screen, how do we transition if off? In the entering transition, we needed to define the starting state, which was `opacity: 0` For the leaving transition, we need to define the **ending** state, which brings us to the next question: **What should the ending style be?**

Since we are transitioning *away from* visible (`1`)and going towards invisible (`0`), we'd set that ending style in the `v-leave-to` class.

```
.v-enter { /* starting style */
    opacity: 0;
}

.v-enter-active { /* active entering style */
    transition: opacity 2s ease-in;
}

.v-leave-to { /* ending style */
    opacity: 0;
}
```

Now, the box will transition away from the default state of `1` to our ending state of `0`. And this brings us to the final major question we need to ask ourselves when defining a transition: **What should the active leaving style be?**

Similar to the entering transition, we define this in the `v-leave-active` class, setting its duration, easing, etc.

```
.v-enter { /* starting style */
    opacity: 0;
}

.v-enter-active { /* active entering style */
    transition: opacity 2s ease-in;
}

.v-leave-active { /* active leaving style */
    transition: opacity 2s ease-out;
}

.v-leave-to { /* ending style */
    opacity: 0;
}
```

Now that we understand the mechanics of a Vue's transition classes, let's use Vue's `transition` component to build our first practical transition.

To Follow Along

You can download the starting code in the Lesson Resources on this page to follow along. Just remember to run `npm install` to get all the dependencies.

A Simple Transition

Whenever something abruptly pops into the DOM, it can be a bit disorienting to our user and they may not immediately know what changed on the screen. A simple fix for this is to *fade* the element into view over time to provide them context about what is changing. Let's take the concepts from our `opacity` example from earlier and implement a simple fade transition in our example app.

Let's say we have a modal. It could be a login modal, a configuration modal, etc. When a ****button is clicked, the modal fades in. Then the modal itself has button, and when that is clicked, the modal fades away.

Our starting template looks like:

src/views/Home.vue

```
<template>
    <div>
      <button @click="toggleModal">Open</button>

    <div v-if="isOpen" class="modal">
      <button @click="toggleModal">Close</button>
    </div>

    </div>
</template>
```

And the script section looks like:

src/views/Home.vue

```
<script>
export default {
  data() {
    return {
      isOpen: false
    }
  },
  methods: {
    toggleModal() {
      this.isOpen = !this.isOpen
    }
  }
}
</script>
```

We have an `isOpen` data property, which we toggle between `true` and `false` when the `toggleModal` method is run. Because we have `v-if="isOpen"` on our modal `div`, the modal will appear and disappear whenever the **open** and **close** buttons are pressed.

By wrapping the modal in a `transition` component, we can then create a transition for it as it opens and closes.

src/views/Home.vue

```
<template>
    <div>
      <button @click="toggleModal">Open</button>

         <transition name="fade"> // <-- named transition
        <div v-if="isOpen" class="modal">
          <button @click="toggleModal">Close</button>
        </div>
      </transition>

    </div>
</template>
```

Named Transitions

Notice how we used the `name` attribute to give the transition a name of `fade`. This allows us to prepend our transition's classes with that name (`fade-enter` instead of `v-enter`). Named transitions help us stay organized as our app scales, and makes our transitions more reusable. We may want to use this `fade` transition on other elements throughout our app, which is why we ought to be naming our transitions based on what they *do* instead of what element they target. We could've named this transition `modal` but that name just describes this one specific use case, and we may want to `fade` things that aren't modals.

Entering Transition

Now we can make use of our named transition classes to create the **enter** transition, which entails defining the starting style. Remember the questions we should be asking ourselves?

What should the starting style be?

src/views/Home.vue

```
.fade-enter { /* starting style */
  opacity: 0;
}
```

What should the active entering style be?

src/views/Home.vue

```
.fade-enter { /* starting style */
  opacity: 0;
}

.fade-enter-active { /* entering style */
  transition: opacity .5s ease-out;
}
```

Within `.fade-enter-active` we defined how we want the CSS `transition` to behave, specifying what property we're transitioning (`opacity`) how long the transition's duration is (`.5s`) and the timing function (`ease-out`).

Leaving Transition

Now that our **enter** transition is built, we can create our **leaving** transition.

What should the ending style be?

src/views/Home.vue

```
.fade-leave-to { /* ending style */
  opacity: 0;
}
```

What should the active leaving style be?

src/views/Home.vue

```
.fade-leave-active { /* leaving style */
  transition: opacity .5s ease-out;
}
```

As you can see, the transition's **ending** style (`.fade-leave-to`) has `opacity` at `0` and the **leaving** state (`.fade-leave-active`) contains the same CSS transition as our entering transition. Because it's the same, we can condense our styles like so:

src/views/Home.vue

```
.fade-enter {
  opacity: 0;
}


.fade-enter-active,
.fade-leave-active {
  transition: opacity .5s ease-out;
}

.fade-leave-to {
  opacity: 0;
}
```

Additional Transition Classes

If you look at the documentation for Vue's Enter/Leave transitions, you'll also find the `v-enter-to` and `v-leave` classes. The reason we didn't cover them in this lesson is because the style we were transitioning to (`opacity: 1`) was already the default style of our element. The same goes for what we were transitioning away from (`opacity: 1`). That is why we did not need to explicitly set our opacity to 1 in `v-enter-to` or `v-leave`. You'll only ever need to use these classes when the style you are transitioning to (`v-enter-to`) or away from (`v-leave`) is different from the inherent style of the element OR if you run into browser compatibility issues, where these classes may come in handy for you.

Let's ReVue

In this lesson, we've covered what the nature of a transition is, explored the mechanics of Vue's transition component and its built-in classes, then built our first simple transition, using these questions to guide our decisions:

- **What should the default style be?**
- **What should the starting style be?**
- **What should the active entering style be?**
- **What should the ending style be?**
- **What should the active leaving style be?**

In the next lesson, we'll look at using these same concepts to create a page transition using Vue Router.