Why the Composition API

There's been some confusion over the new Vue 3 composition API. By the end of this lesson it should be clear why the limitations of Vue 2 have led to its creation, and how it solves a few problems for us.

There are currently three limitations you may have run into when working with Vue 2:

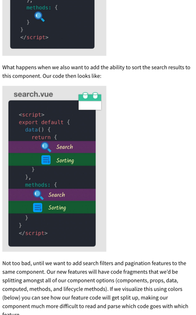
- As your components get larger readability gets difficult.
- The current code reuse patterns all come with drawbacks.
- Vue 2 has limited TypeScript support out of the box.

I will go into detail with the first two, so it's apparent what problem the new API solves.

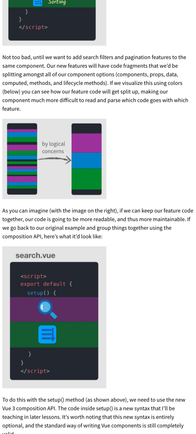
Large components can be hard to read & maintain.

To wrap our head around this problem lets think about a component that takes care of searching the products on our website.

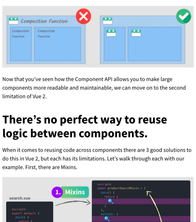
The code for this component, using the standard Vue component syntax, is going to look like this:



What happens when we also want to add the ability to sort the search results to this component. Our code then looks like:



Not too bad, until we want to add search filters and pagination features to the same component. Our new features will have code fragments that we'd be splitting amongst all of our component options (components, props, data, computed, methods, and lifecycle methods). If we visualize this using colors (below) you can see how our feature code will get split up, making our component much more difficult to read and parse which code goes with which feature.



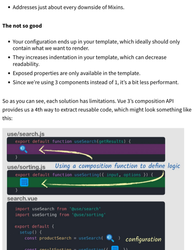
As you can imagine (with the image on the right), if we can keep our feature code together, our code is going to be more readable, and thus more maintainable. If we go back to our original example and group things together using the composition API, here's what it'd look like:

To do this with the setup() method (as shown above), we need to use the new Vue 3 composition API. The code inside setup() is a new syntax that I'll be breaking in later lessons. It's worth noting that this new syntax is entirely optional, and the standard way of writing Vue components is still completely valid.

I know when I first saw this, I wondered, "Wait, does this mean I create a gigantic setup method, and put all my code in there? How can that be how this works?"

No, don't worry, this isn't what happens. When you organize components by features using the composition API, you'll be grouping features into composition functions that get called from your setup method, like so:

Using this our components can now be organized by logical concerns (also known as "features"). However, this doesn't mean that our user interface will be made up of fewer components. You're still going to use good component design patterns to organize your applications:



Now that you've seen how the Component API allows you to make large components more readable and maintainable, we can move on to the second limitation of Vue 2.

There's no perfect way to reuse logic between components.

When it comes to reusing code across components there are 3 good solutions to do this in Vue 2, but each has its limitations. Let's walk through each with our example. First, there are Mixins.

The good

- Mixins can be organized by feature.

The not so good

- They are **conflict-prone**, and you can end up with property name conflicts.
- **Unclear relationships** on how mixins interact, if they do.
- **Not easily reusable** if I want to configure the Mixin to use across other components.

This last item leads us to take a look at **Mixin Factories**, which are functions that return a customized version of a Mixin.

As you can see above, Mixin Factories allow us to customize the Mixins by sending in a configuration. Now we can configure this code to use across multiple components.

The good

- **Easily reusable** now that we can configure the code.
- We have **more explicit relationships** of how our Mixins interact.

The not so good

- Namespacing requires strong conventions and discipline.
- We still have implicit property additions, meaning we have to look inside the Mixin to figure out what properties it exposes.
- There's no instance access at runtime, so Mixin factories can't be dynamically generated.

Luckily, there's one more solution that can often be the most useful, **Scoped Slots**:

The good

- Addresses just about every downside of Mixins.

The not so good

- Your configuration ends up in your template, which ideally should only contain what we want to render.
- They increases indentation in your template, which can decrease readability.
- Exposed properties are only available in the template.
- Since we're using 3 components instead of 1, it's a bit less performant.

So as you can see, each solution has limitations. Vue 3's composition API provides us a 4th way to extract reusable code, which might look something like this:

Now we're creating components using the composition API inside functions that get imported and used in our setup method, where we have any configuration needed.

The good

- We're writing less code, so it's easier to pull a feature from your component into a function.
- It builds on your existing skills since you're already familiar with functions.
- It's more flexible than Mixins and Scoped Slots since they're just functions.
- Intellisense, autocomplete, and typings already work in your code editor.

The not so good

- Requires learning a new low-level API to define composition functions.
- There are now two ways to write components instead of just the standard syntax.

Hopefully, the "why" behind the composition API is now clear to you, I know it wasn't clear to me at first. In the next lesson I'll be diving into the new syntax for composing components.

Teleport

Vue's component architecture enables us to build our user interface into components that beautifully organize our business logic and presentation layer. However, there are some instances where one component has some html that needs to get rendered in an alternative location. For example:

1. Styles that require fixed or absolute positioning and z-index. For example, it's a common pattern to place UI components (like modals) right before the `</body>` tag to ensure they are properly placed in front of all other parts of the webpage.
2. When our Vue application is running on a small part of our webpage (or a widget), sometimes we may want to move components to other locations in the DOM outside of our Vue app.

Solution

The solution Vue 3 provides is the **Teleport** component. Previously this was named "Portal", but the name was changed to Teleport so not to conflict with the future `<portal>` element which might some day be a part of the HTML standard. The Teleport component allows us to specify template html (which may include child components) that we can send to another part of the DOM. I'm going to show you some very basic usage, and then show you how we might use this in something more advanced. Let's start by adding a `div` tag outside of our Vue app, in our basic Vue CLI generated app:

/public/index.html

```
    ...
    <div id="app"></div>
    <div id="end-of-body"></div>
  </body>
</html>
```

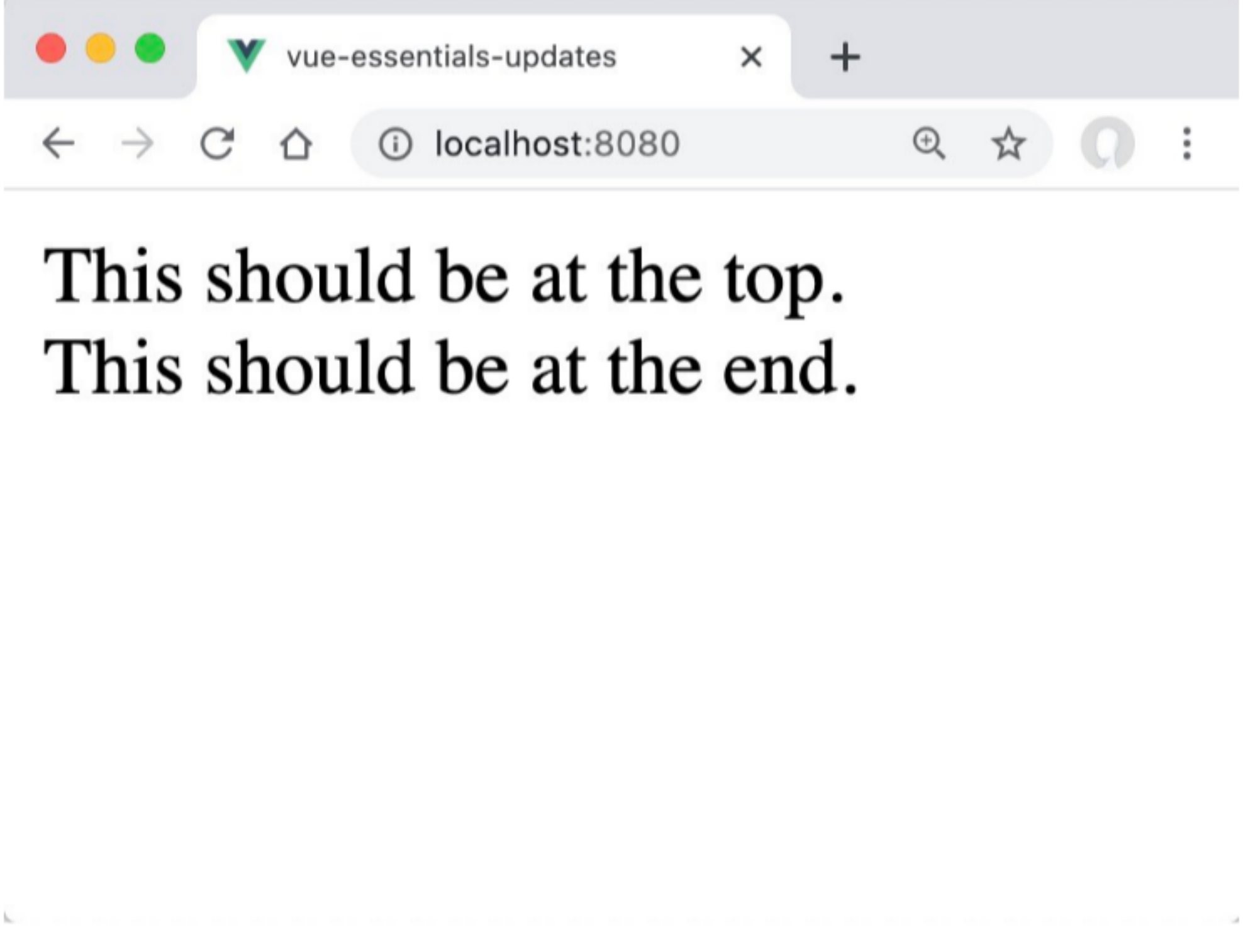
Then let's try teleporting some text to this `#end-of-body` div from inside our Vue application to slightly outside the application.

/src/App.vue

```
<template>
  <teleport to="#end-of-body">
    This should be at the end.
  </teleport>
  <div>
    This should be at the top.
  </div>
</template>
```

Notice the teleport line where we specify the div we want to move our template code to, and if we did this right, the text at the top should be moved to the bottom. Sure enough, it does:

(width=300)



Teleport Options for To

Our `to` attribute simply needs to be a valid DOM query selector. Aside from using the `id` like I did above, here are three more examples.

Class selector

```
<teleport to=".someClass">
```

Data selector

```
<teleport to="[data-modal]">
```

Using a data attribute our target div might look like:

Dynamic selector

If you needed you could even bind a dynamic selector, adding the colon.

```
<teleport :to="reactiveProperty">
```

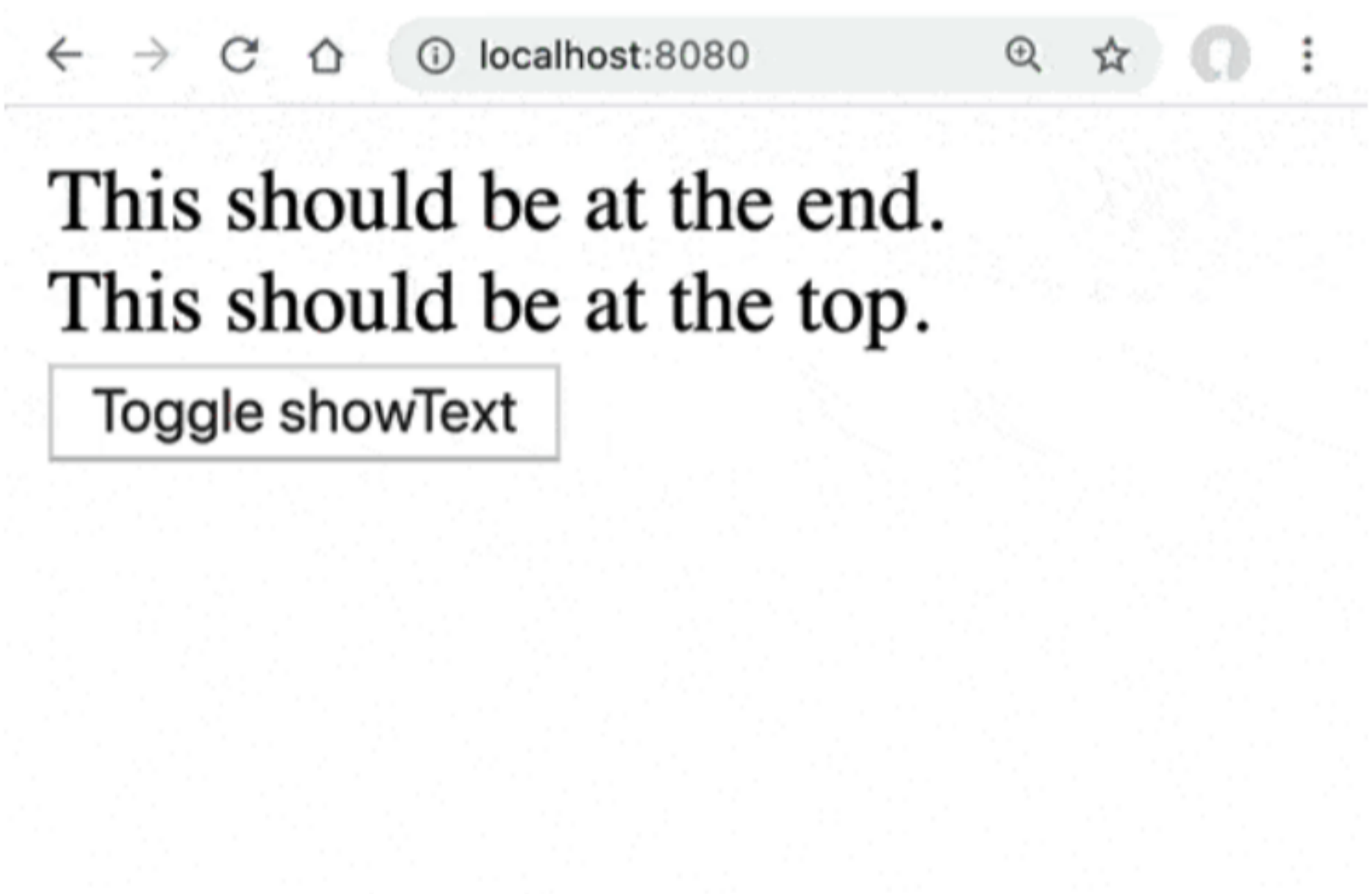
Disabled State

Modals and other pop-ups often start hidden until they are displayed on the screen. For that reason, teleport has a disabled state where the content stays inside the original component. It's not until teleport is enabled that it will be moved to the target positioning. Let's update the code to be able to toggle `showText`, like so:

```
<template>
  <teleport to="#end-of-body" :disabled="!showText">
    This should be at the end.
  </teleport>
  <div>
    This should be at the top.
  </div>
  <button @click="showText = !showText">
     Toggle showText
  </button>
</template>
<script>
export default {
  data() {
    return {
      showText: false
    };
  }
};
</script>
```

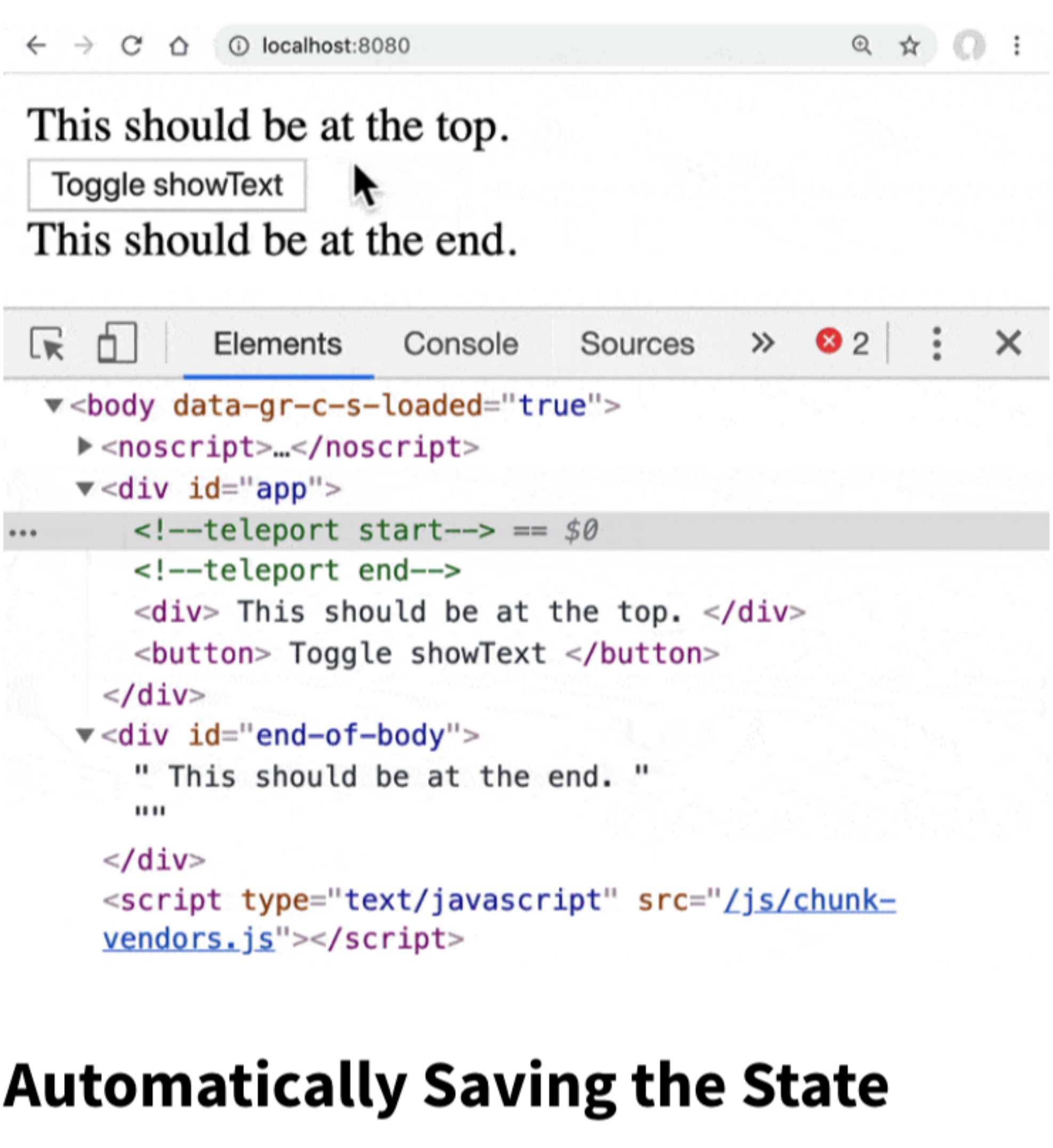
As you can see, the content inside teleport gets moved from inside the component, to outside the component as we toggle:

<01-disable.gif width=250>



If we inspect the source realtime, we can see that the content is actually being moved in the DOM from place to place.

<02-devtools.gif width=367>



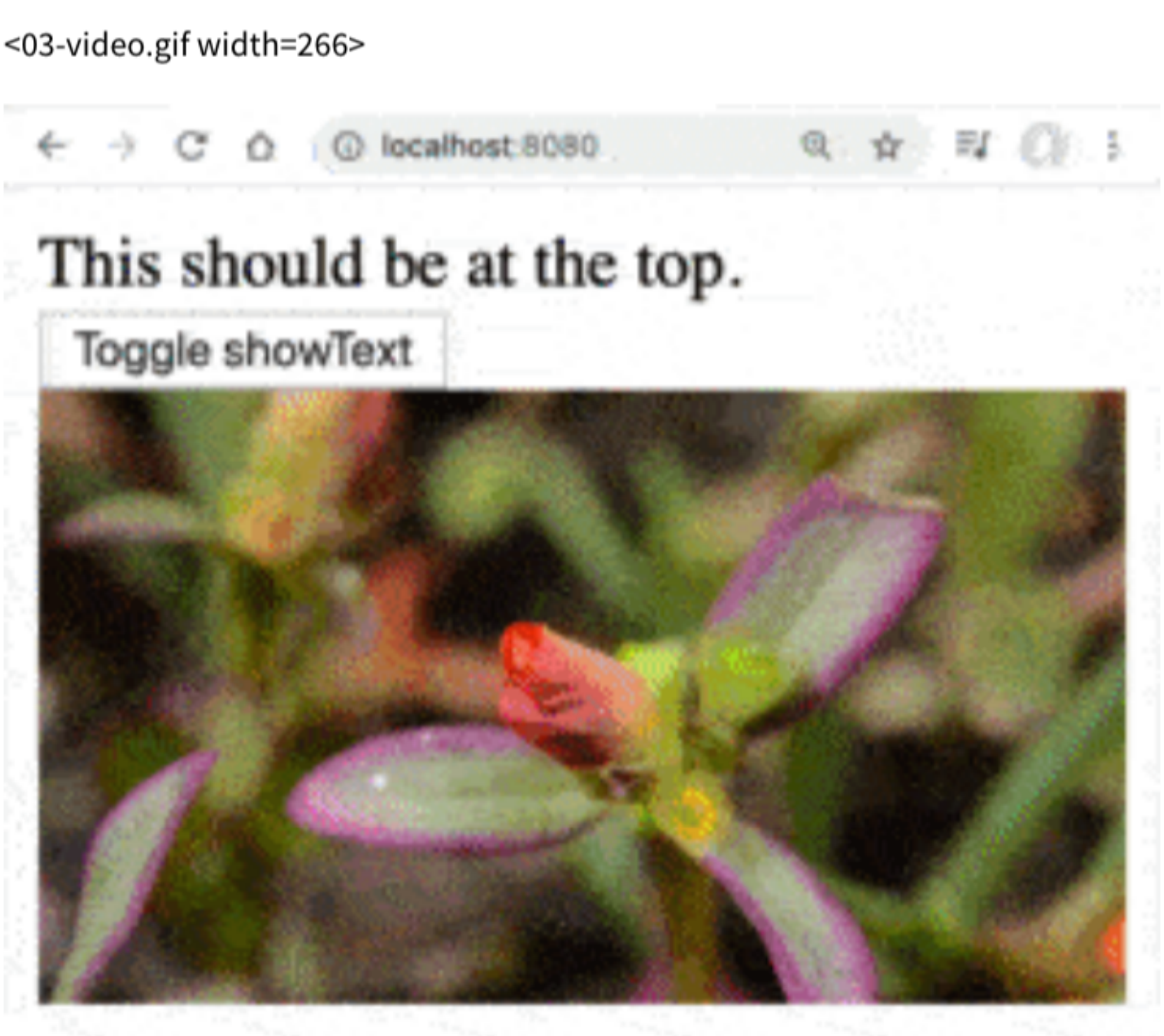
Automatically Saving the State

When teleport goes from disabled to enabled, the DOM elements are re-used, so they completely retain the existing state. This can be illustrated by teleporting a playing video.

```
<template>
  <teleport to="#end-of-body" :disabled="!showText">
    <video autoplay="true" loop="true" width="250">
      <source src="flower.webm" type="video/mp4">
    </video>
  </teleport>
  <div>
    This should be at the top.
  </div>
  <button @click="showText = !showText">
      Toggle showText
  </button>
</template>
<script>
export default {
  data() {
    return {
      showText: false
    };
  }
};
</script>
```

As you can see in the video below, the state of the video as it moves between locations remains the same.

<03-video.gif width=266>



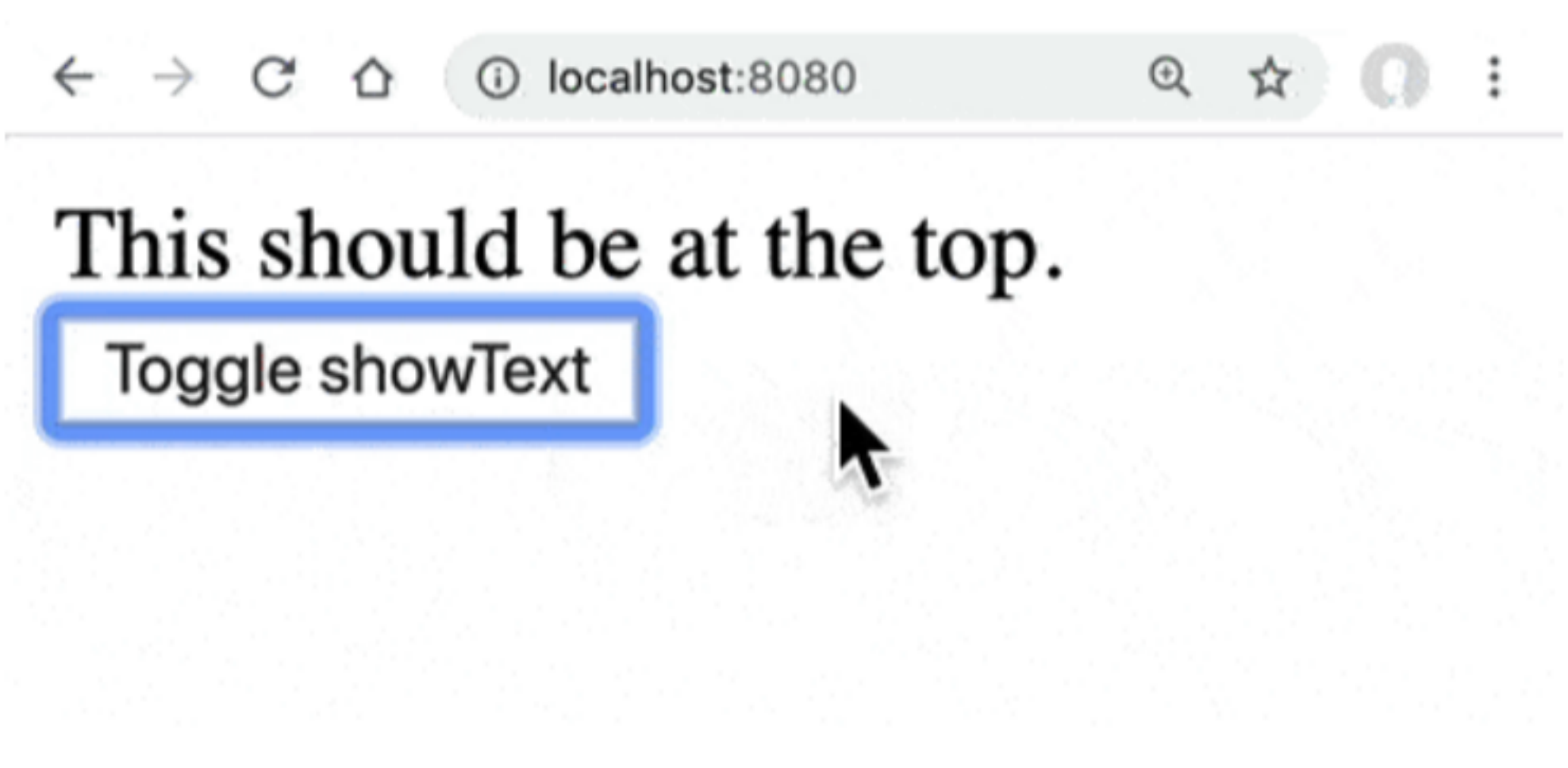
Hiding the Text

If the content we had inside teleport was a modal, we probably wouldn't want to show it until it was active. Right now "This should be at the end." is displaying inside our component, even when showText is false. We can disable this from showing by simply adding a v-if.

```
<template>
  <teleport to="#end-of-body" :disabled="!showText" v-if="showText">
      This should be at the end.
  </teleport>
  ...
```

Now our text only shows up when showText is true, and thus teleported to the bottom of the page.

<04-v-if width=250>



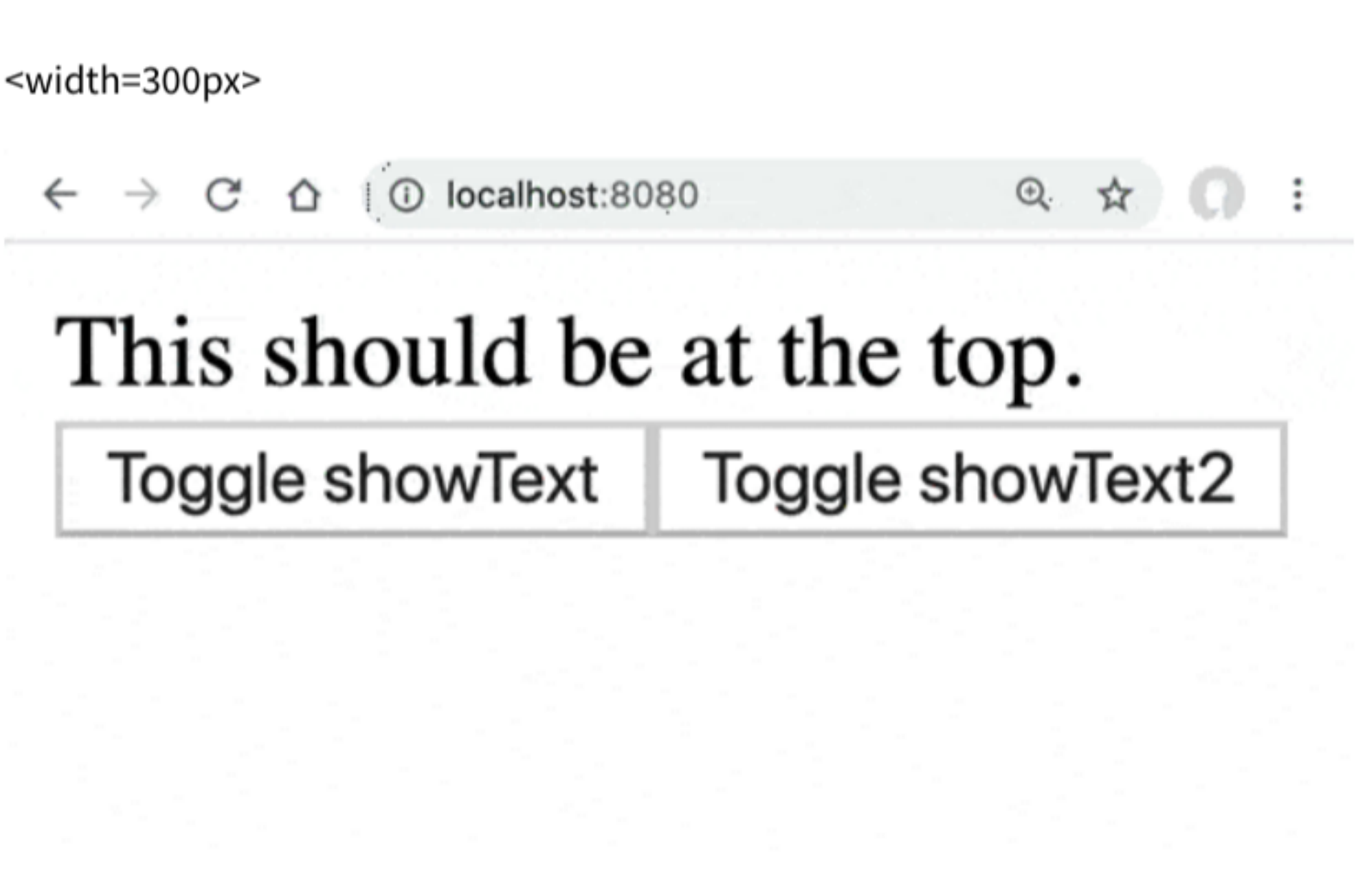
Multiple Teleports into the Same Place

This made me wonder, what happens when you teleport two things into the same place? I can can see (especially with modals) how you might want to teleport more than one thing. Let's give it a try with our overly simple example, simply creating a showText2.

```
<template>
  <teleport to="#end-of-body" :disabled="!showText" v-if="showText">
    This should be at the end.
  </teleport>
  <teleport to="#end-of-body" :disabled="!showText2" v-if="showText2">
    This should be at the end too.
  </teleport>
  <div>
    This should be at the top.
  </div>
  <button @click="showText = !showText">
      Toggle showText
  </button>
  <button @click="showText2 = !showText2">
      Toggle showText2
  </button>
</template>
<script>
export default {
  data() {
    return {
      showText: false,
      showText2: false
    };
  }
};
</script>
```

You can see from the video below that it works as you'd expect, adding the content as it's toggled. It's interesting to see that it's simply appending the element based on which one is clicked first.

<width=300px>



Conclusion

As you can see, using teleport provides you a way to keep your code in the same component, while moving pieces of it into other parts of your page. Aside from the obvious solution of using this for modals which need to appear on top of the rest of the page, and placed right above your `</body>` tag, I'm excited to see how else this Vue 3 feature is used in practice.

For a more detailed written description, check out the RFC.

Thank you for working through our first Vue 3 course. Next up I'll be working on a Vue 3 Reactivity course, explaining some of the core concepts of Vue 3's new Reactivity engine.

Setup & Reactive References

At this point, you're probably wondering what the new composition API syntax looks like, and we're about to dive in. First, we want to be clear about when you might use it, and then we'll learn about the setup function and Reactive References or refs. Also, you may want to grab Vue Mastery's Vue 3 Cheat Sheet if you haven't yet.

Disclaimer: If you haven't caught on yet, the composition API is purely additive, and nothing is deprecated. You can code Vue 3 the same way you code up Vue 2.

When to use the Composition API?

When any of the following are true:

- You want optimal TypeScript support.
- The component is too large and needs to be organized by feature.
- Need to reuse code across other components.
- You & your team prefer the alternative syntax.

Disclaimer #2: The examples that follow are quite simple. Constructing components this simple using the Composition API would be unnecessary, but make it easier to learn.

Let's start with a very simple component written using our normal Vue 2 API, which is valid in Vue 3.

```
<template>
  <div>Capacity: {{ capacity }}</div>
</template>
<script>
export default {
  data() {
    return {
      capacity: 3
    };
  }
};
</script>
```

Notice I have a simple `capacity` property, which is reactive. Vue knows to take each of the properties in the object returned by the data property, and make them reactive. This way, when these reactive properties get changed, components that use this property gets re-rendered. Duh, right? In the browser we see:

Using the Setup Function

In Vue 3 using the Composition API, I would start by writing that `setup` method we've seen before:

```
<template>
  <div>Capacity: {{ capacity }}</div>
</template>
<script>
export default {
  setup() {
    // more code to write
  }
};
</script>
```

The `setup` executes before any of the following options are evaluated:

- Components
- Props
- Data
- Methods
- Computed Properties
- Lifecycle methods

It's also worth mentioning that the `setup` method does not have access "this", unlike other Component options. In order to get access to the properties we normally would access using this, `setup` has two optional arguments. The first is props which is reactive and can be watched, as such:

```
import { watch } from "vue";
export default {
  props: {
    name: String
  },
  setup(props) {
    watch(() => {
      console.log(props.name);
    });
  }
};
```

The second argument is context, that has access to a bunch of useful data:

```
setup(props, context) {
  context.attrs;
  context.slots;
  context.parent;
  context.root;
  context.emit;
}
```

But let's get back to our example. Our code needs a reactive reference:

Reactive References

```
<template>
  <div>Capacity: {{ capacity }}</div>
</template>
<script>
import { ref } from "vue";
export default {
  setup() {
    const capacity = ref(3);
    // additional code to write
  }
};
</script>
```

`const capacity = ref(3)` is creating a "Reactive Reference." Basically it's wrapping our primitive integer (3) in an object, which will allow us to track changes. Remember, previously our `data()` option already wrapping our primitive (capacity) inside an object.

Aside: *The composition API allows us to declare reactive primitives that aren't associated with a component, and this is how we do it.*

One last step, we need to explicitly return an object with properties our template will need to render properly.

```
<template>
  <div>Capacity: {{ capacity }}</div>
</template>
<script>
import { ref } from "vue";
export default {
  setup() {
    const capacity = ref(3);
    return { capacity };
  }
};
</script>
```

This returned object is how we expose which data we need access to in the `renderContext`.

Being explicit like this is a little more verbose, but it's also intentional. It helps with longer-term maintainability because we can control what gets exposed to the template, and trace where a template property is defined. We now have what we started with:

Using Vue 3 with Vue 2

It's worth noting that you can use the Vue 3 Composition API with Vue 2 today by using this plugin. Once you install and configure it on a Vue 2 application, you'd use the same syntax I'm teaching you above with one small change. Instead of

```
import { ref } from "vue";
```

you would write

```
import { ref } from "@vue/composition-api";
```

That's how I tested all the above code, in case you're wondering.

Next, we'll learn how to write component methods using this new syntax.

Lifecycle Hooks

You're probably familiar with Vue lifecycle hooks, which give us the ability to run code when a component reaches a particular state in execution. Let's review the typical LifeCycle hooks as you know them:

- **beforeCreate -** Called immediately after instance is initialized, before options are processed.
- **created -** Called after the instance has been created.
- **beforeMount -** Right before mounting of the DOM begins
- **mounted -** Called when the instance has been mounted (browser updated).
- **beforeUpdate -** Called when reactive data has changed, before the DOM is re-rendered.
- **updated -** Called when reactive data has changed, and the DOM has been re-rendered.
- **beforeDestroy -** Called right before the Vue instance is destroyed.
- **destroyed -** Called after the Vue instance has been destroyed.

There are two newer Vue 2 LifeCycle methods you may not be familiar with:

- **activated -** Used for , when a component inside is toggled on.
- **deactivated -** Used for , when a component inside is toggled off.
- **errorCaptured -** Called when an error from any descendent component is captured.

For more detailed explanations check out the API documentation on LifeCycle hooks.

Unmounting in Vue 3

In Vue 3 `beforeDestroy()` can also be written as `beforeUnmount()`, and `destroyed()` can be written as `unmounted()`. When I asked Evan You about these changes, he mentioned it's just a better naming conventions, because Vue *mounts* and *unmounts* components.

Composition API LifeCycle Methods

In Vue 3's Composition API we can create callback hooks inside `setup()` by adding **on** to the LifeCycle method name:

```
import {
  onBeforeMount,
  onMounted,
  onBeforeUpdate,
  onUpdated,
  onBeforeUnmount,
  onUnmounted,
  onActivated,
  onDeactivated,
  onErrorCaptured
} from "vue";

export default {
  setup() {
    onBeforeMount(() => {
      console.log("Before Mount!");
    });
    onMounted(() => {
      console.log("Mounted!");
    });
    onBeforeUpdate(() => {
      console.log("Before Update!");
    });
    onUpdated(() => {
      console.log("Updated!");
    });
    onBeforeUnmount(() => {
      console.log("Before Unmount!");
    });
    onUnmounted(() => {
      console.log("Unmounted!");
    });
    onActivated(() => {
      console.log("Activated!");
    });
    onDeactivated(() => {
      console.log("Deactivated!");
    });
    onErrorCaptured(() => {
      console.log("Error Captured!");
    });
  }
};
```

You might notice that two hooks are missing. `beforeCreate` and `created` are not needed when using the Composition API. This is because `beforeCreate()` is called right before `setup()` and `created()` is called right after `setup()`. Thus, we simply put code inside `setup()` that would normally be in these hooks, such as API calls.

Two New Vue 3 LifeCycle Methods

There are two more additional watchers coming in Vue 3. These have not been implemented with the Vue 2 Composition API plugin (as I'm writing this), so you can't play with them without using Vue 3 source.

- **onRenderTracked** - called when a reactive dependency is first being accessed in the render function, during render. This dependency will now be tracked. This is helpful to see which dependencies are being tracked, for debugging.
- **onRenderTriggered** - Called when a new render is triggered, allowing you to inspect what dependency triggered a component to re-render.

I'm excited to see what sort of optimization tools can be created with these two hooks.