Renderless Components: 5 Wild Experiments

I think that **renderless components have huge potential to transform the way we write components**, but we're still in the early stages of figuring out what that will look like.

I've been spending the last several weeks experimenting a lot with the different things that we can do with them. I've been pushing the limits pretty far, **going past the edges of what Vue is intended to do**.

Along the way I've discovered a few neat things that I'd like to share with you!

> **If you aren't already familiar with renderless components, you probably won't understand most of this article. Try reading Adam Wathan's great primer on the topic, but don't forget to come back!**

Here are the 5 renderless components we'll look at:

1. **Event** — Listen to events on the `window`
2. **State** — Keep track of state without using `data`
3. **Declarative Lifecycle Methods** — Declare `mounted`, `beforeDestroy`, etc. in our `<template>`
4. **Log** — Log a value to the console when it changes

5. **Interval** — Refresh part of your component at regular intervals

These demonstrate just a few of the things I've discovered — hopefully this will get you excited about the possibilities of renderless components as well!

Event Component

If you want to listen to an event outside of your component, for example on the `window`, you have to write a decent amount of boiler plate code.

What if you could add a component to your `template` in a single line that would take care of all that?

Well, that's what we'll be building!

Using it is as simple as this:

```
<Event event="click" @fired="handleClick" />
```

Anytime the `window` is clicked, it will call our method `handleClick`. Easy!

Boilerplate is no fun

Normally, if you wanted to listen to an event on the `window` you would need to do these things in your component:

- Bind any methods to make sure they have access to `this` (our instance of the Vue component)
- Add event listeners after the component mounts
- Clean up event listeners before the component is destroyed

That boilerplate might look something like this:

```
mounted() {
  // Bind context so we can access 'this'
  this.handleEvent = this.handleEvent.bind(this);

  // Attach event listener to the window
  window.addEventListener(
    this.event,
    this.handleEvent
  );
},

beforeDestroy() {
  // Clean up event listener
  window.removeEventListener(
    this.event,
    this.handleEvent
  );
},
```

You’d also need to do these things if you wanted to listen to events on **any** element outside of your Vue component.

It’s about 20 lines of code. Replacing it with a renderless component not only cleans it up, but makes it much more clear what our component is trying to do.

Renderless to the Rescue

We’ll start out simple with a very basic renderless component.

Instead of using scoped slots, we’ll just return `null` from our `render` method. This component won’t have any children, so no need to do anything fancy with scoped slots here.

```
render() {
  // We don't need to render
  // anything with this component
  return null;
}
```

Next, we’ll need to add in our `event` prop:

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

We’ll add in all of the boiler plate that we have from before.

All we want to do when the event is triggered is to emit our own event `fired` to our parent component. So we’ll start by adding our event handler:

```
methods: {
  handleEvent() {
    // Fire an event so our parent
    // can do something in response
    this.$emit('fired');
  }
}
```

Then we’ll add the event listener when we mount:

```
mounted() {
  // Attach event listener to the window
  window.addEventListener(
    this.event,
    this.handleEvent
  );
}
```

Because we use `this` in `handleEvent`, we need to bind the method first or we'll get this is undefined errors:

```
mounted() {
  // Bind context so we can access 'this'
  this.handleEvent = this.handleEvent.bind(this);

  // Attach event listener to the window
  window.addEventListener(
    this.event,
    this.handleEvent
  );
}
```

Finally, we need to make sure we clean up our event listener before the component is destroyed. Forgetting to do this leads to memory leaks, which can cause huge problems if you are reusing the component a lot!

Always remember to clean up any event handlers you create yourself.

Vue automatically clears up the ones that are created when listening to events on components, so don't worry about those.

```
beforeDestroy() {
  // Clean up event listener
  window.removeEventListener(
    this.event,
    this.handleEvent
  );
}
```

That’s it! We’ve built ourselves an awesome, declarative, renderless, event component!

Finished Component

Here is the full source for the component:

```
export default {
  props: {
    event: {
      type: String,
      required: true,
    }
  },

  mounted() {
    // Bind context so we can access 'this'
    this.handleEvent = this.handleEvent.bind(this);

    // Attach event listener to the window
    window.addEventListener(
      this.event,
      this.handleEvent
    );
  },

  beforeDestroy() {
    // Clean up event listener
    window.removeEventListener(
      this.event,
      this.handleEvent
    );
  },
  methods: {
    handleEvent() {
      // Fire an event so our parent
      // can do something in response
      this.$emit('fired');
    }
  },

  render() {
    // We don't need to render
    // anything with this component
    return null;
  }
}
```

State Component

We often use local state to keep track of toggled UI elements, form state, and other things.

Wouldn’t it be nice if we had a simpler way of keeping track of that?

If we wanted to **toggle a UI pane** open/closed:

```
<template>
  <State :initial-state="{ open: false }">
    <Pane
      slot-scope="{ state, update }"
      :open="state.open"
      @toggle="(val) => update({ open: val })"
    />
  </State>
</template>
```

Or if we needed to build the canonical **counter example**:

```
<template>
  <State :initial-state="{ count: 0 }">
    <div slot-scope="{ state, update }">
      <button @click="update({ count: state.count - 1 })">-</button>
      {{ state.count }}
      <button @click="update({ count: state.count + 1 })">+</button>
    </div>
  </State>
</template>
```

Isn’t composing behaviour fun?

Let’s get started building this `State` component then!

Using scoped slots in a render function

This component is less code than our previous `Event` example, but differs in one main way. This component **will** have children, so we need to render them somehow.

Our `render` method will look like this:

```
render() {
  return this.$scopedSlots.default({
    state: this.state,
    update: this.update,
  });
}
```

We access our component's scoped slots through `this.$scopedSlots`, and grab the default one. Vue let's us name scoped slots so we can have multiple, but here we only need the one.

By calling `this.$scopedSlots.default()`, we can pass in the data that will be provided to the scoped slot. In this case we are passing it `this.state` and `this.update`.

This is where our state comes from:

```
export default {
  props: {
    initialState: {
      type: Object,
      required: true,
    }
  },

  data() {
    return {
      state: this.initialState,
    }
  }

  //...
}
```

Using the `initialState` prop we can initialize our state — you won't find creatively named variables here, folks!

The last piece is our `update` method:

```
methods: {
  update(newState) {
    this.state = Object.assign(this.state, newState);
  }
},
```

If you're not familiar with how `Object.assign` works, it copies enumerable properties from one object to the next. In this case, we're using it to copy all of the properties of `newState` to `this.state`.

This means we only need to specify the properties we want to update when we call `this.update`. Any property not specified in `newState` will remain untouched in our state.

The final component

Putting everything together we get our renderless state component:

```
<script>
export default {
  props: {
    initialState: {
      type: Object,
      required: true,
    }
  },

  data() {
    return {
      // We can initialize our state using the
      // prop `initialState`
      state: this.initialState,
    }
  },

  methods: {
    update(newState) {
      // Copy all properties from newState on to
      // this.state, overriding anything on this.state
      this.state = Object.assign(this.state, newState);
    }
  },

  render() {
    // Pass our state and the update function into
    // our scoped slot so we can render children.
    return this.$scopedSlots.default({
      state: this.state,
      update: this.update,
    });
  }
}
</script>
```

Declarative Lifecycle Components

So far the components we've looked at have been fairly normal — things will start to get weirder soon.

In the spirit of **making everything declarative** — or at least as much as possible — why don't we try making a component that handles it's lifecycle methods declaratively?

Then we can do this in our template:

```
<template>
  <Mounted @mounted="fetchData" />
  {{ data }}
</template>
```

And as soon as our component mounts the data will be fetched. Cool, huh?

Building our component

It only takes use 2 simple steps.

First, we add our `render` function that returns `null`:

```
render() {
  return null;
}
```

Second, we need to emit a `mounted` event in our `mounted` lifecycle handler:

```
mounted() {
  // All we do is fire an event when the component mounts
  this.$emit('mounted');
}
```

Combining these we get a fairly simple component:

```
<script>
export default {
  mounted() {
    // All we do is fire an event when the component mounts
    this.$emit('mounted');
  },

  render() {
    // We don't need to render anything with this component
    return null;
  }
}
</script>
```

Not only can we do this with the `mounted` lifecycle, but we can easily make components for **any other lifecycle method** that we want.

Keep in mind that these events are actually triggered when the **child's lifecycle event** happens. However, because lifecycle events are triggered recursively, this should work as expected in most cases.

And as always, here is a sandbox to play around with.

Logging Component

Since we're doing all of these things declaratively now, it would be nice if we could **debug the values** that are being passed around inside of the `<template>`.

Unfortunately, we can't just drop in `console.log`s everywhere, and it doesn't work well to step through with a debugger.

Instead, let's build a basic `<Log>` component, which will let us do this:

```
<State :initial-state="{ count: 0 }">
  <template slot-scope="{ state, update }">
    <div>
      <!-- Add in logging -->
      <Log
        :value="state.count"
        :format="(val) => `Count is: ${val}`"
      />

      <button @click="update({ count: state.count - 1 })">-</button>
      {{ state.count }}
      <button @click="update({ count: state.count + 1 })">+</button>
    </div>
  </template>
</State>
```

Now, whenever `state.count` changes, we will print out a nicely formatted message in the console!

Let's log some things!

The main idea behind this component is to watch for when the `value` prop changes, and then log it to the console:

```
watch: {
  value() {
    // Whenever `value` changes, we'll log it
    console.log(this.value);
  }
}
```

This works, but it would be nice to have more control over what we log.

We can do this by passing in a function as a prop, `this.format`. This function will take the value we want to log and return a string.

Then we can use `this.format` to format our message:

```
watch: {
  value() {
    // Whenever `value` changes, we'll log it
    const toLog = this.format
      ? this.format(this.value)
      : this.value;
    console.log(toLog);
  }
}
```

Our props section for this component will look like this:

```
props: {
  // We don't care what type this prop is, since
  // all we're doing is logging it out.
  value: {},

  // Pass in a function to format your log message
  format: {
    type: Function,
  },
}
```

Normally it’s a good idea to specify the type of each prop, because it lets us **catch silly errors** much faster.

Here, however, we don’t care at all what value is being passed in.

Whatever we get passed we’ll log out. By **leaving the prop definition empty** we can specify that we’re expecting a `value` prop, without having to specify `type`, `default`, or any other field.

The final logging component

Here is the final source:

```
<script>
export default {
  props: {
    // We don't care what type this prop is, since
    // all we're doing is logging it out.
    value: {},

    // Pass in a function to format your log message
    format: {
      type: Function,
    },
  },

  watch: {
    value() {
      // Whenever `value` changes, we'll log it
      const toLog = this.format
        ? this.format(this.value)
        : this.value;
      console.log(toLog);
    }
  },

  render() {
    return null;
  }
}
</script>
```

Interval Component

Let’s take this all one step further and build a component that let’s us **define intervals declaratively**.

We’ll wrap up all of the `setInterval` boiler plate into a nice component, so all we have to do is:

```
<Interval :delay="1000">
  <div slot-scope="{}">
    <!-- Updates every second -->
    The time is {{ new Date().toLocaleTimeString() }}
  </div>
</Interval>
```

The child of this component will be updated at whatever `delay` you set it to.

You may have noticed we have to add in the extra `slot-scope="{}"` that isn’t really doing anything. We have to add that in because this is **starting to push the limits** of what Vue allows us to do easily.

I’m working on some ways around this, as well as a library that will make building all types of renderless components much easier — but first let’s see how it’s built!

Forcing a scoped slot to update

This whole component revolves around being able to **force a child to update** whenever we want it to. We could use key-changing here, but since we want to use a scoped slot we'll do something a little different.

A quick little review of Vue's reactivity.

When you use a prop inside of the `<template>`, Vue keeps track that it needs that value to render. If that prop is ever changed, Vue detects this and will re-render that component.

But our `<template>` section is just some sugar that gets compiled down to a `render` function at the end of the day, so the process works exactly the same inside of the `render` function.

All we need to do is register a reactive value in the render function, and then update it at regular intervals.

To do that we'll create a value, `tick`, that does just that:

```
data() {
  // Keep track of ticks
  return { ticks: 0 };
}
```

Then we'll make sure it gets tracked in our render function:

```
render() {
  return this.$scopedSlots.default({
    ticks: this.ticks,
  });
}
```

Really, we just need to access `this.ticks` in the render method, so this would work just as well:

```
render() {
  const copyTicks = this.ticks;
  return this.$scopedSlots.default({});
}
```

But I figure we might as well pass it into our slot — it may come in handy later!

Updating at regular intervals

As soon as our component mounts, we'll want to create an interval and set it up to update `tick` at the specified time intervals:

```
mounted() {
  // Set up the interval that increases the tick
  // and emits the event.
  this.timerId = setInterval(() => {
    this.ticks += 1;
    this.$emit('tick');
  }, this.delay);
}
```

We also emit an event for each tick, to give us some flexibility in how this component is used.

Don't forget to clean up your interval!

```
beforeDestroy() {
  // We have to make sure to clean up the interval
  // before the component is destroyed
  clearInterval(this.timerId);
}
```

The final component

Here is the source for the entire component:

```
<script>
export default {
  props: {
    // Specify how long the interval is
    delay: {
      type: Number,
      required: true,
    }
  },

  data() {
    // Keep track of ticks
    return { ticks: 0 }
  },

  mounted() {
    // Set up the interval that increases the tick
    // and emits the event.
    this.timerId = setInterval(() => {
      this.ticks += 1;
      this.$emit('tick');
    }, this.delay);
  },

  beforeDestroy() {
    // We have to make sure to clean up the interval
    // before the component is destroyed
    clearInterval(this.timerId);
  },

  render() {
    return this.$scopedSlots.default({
      ticks: this.ticks,
    });
  }
}
</script>
```

We’ve only scratched the surface

Now you’ve seen some of the cool things that renderless components can do.

Where do we go from here?

We have only just begun to start figuring out what is possible with renderless components. I believe there are huge wins we can unlock with them, because they allow us to write components in a whole new way.