From Vue 2 to Vue 3

Vue 3 is packed full of cool new tools and gizmos to empower your applications even further. As developers the more of these features we know and understand, the more powerful and feature rich applications we are able to deliver.

Vue 3 doesn’t actually come with a lot of breaking changes in relation to Vue 2, but due to it being a complete rewrite, some breaking changes were unavoidable and necessary.

If you have an application that is currently using props, receiving attributes and/or emitting events to communicate with a parent this course is for you.

Disclaimer. This course is intended for developers that already have experience with Vue 2. I’m going to assume that you already know how to make your own custom components, and that you are very comfortable with how core API features like v-model, props, emits and listeners work.

Furthermore I’m not going to cover the Composition API in this course. If you want a refresher or you’re completely new to it, we have a fantastic course called **Vue 3 Essentials** that I highly recommend you check out first.

Having said that, let’s talk about what you’re going to learn in this course!

First we’ll take a deep dive in to the new `v-model`. There’s a lot of exciting new features to be learned, like the new defaults for binding value and emiting events.

We’ll also cover multi `v-model` bindings into a single component (yes that’s a thing now!), and how to create your own custom v-model modifiers.

Later on in the course, we’ll take a look at the new `$attrs`. We’ll see how the removal of `$listeners` plays a part into component development, and the importance of being able to control `class` and `style` as fall-through attributes now.

Finally, we’re going to take a look at multi-root components and its caveats, like controlling attribute fall-through and the infamous `inheritAttrs`, when and why to use it.

All of these concepts are super general and can be applied to most of the components out there in the wild.

Ready to jump right in?

See you in the first lesson!

The new v-model

As you probably know, `v-model` allows us to very quickly and easily capture an input’s value into the state of our application. Every time the user types or interacts with an input, `v-model` will let the parent know so that it can update our state.

In Vue 3, `v-model` has gone through a redesign that gives us more power and flexibility when defining how this double binding should be done.

Kicking it off with Native inputs

Let’s start by looking at a native input element.

```
<template>
        <input type="text" />
</template>
```

In Vue 2, whenever you add a `v-model` declaration to a native input element, the compiler produces a block of code that handles the correct input value and event to be listened to.

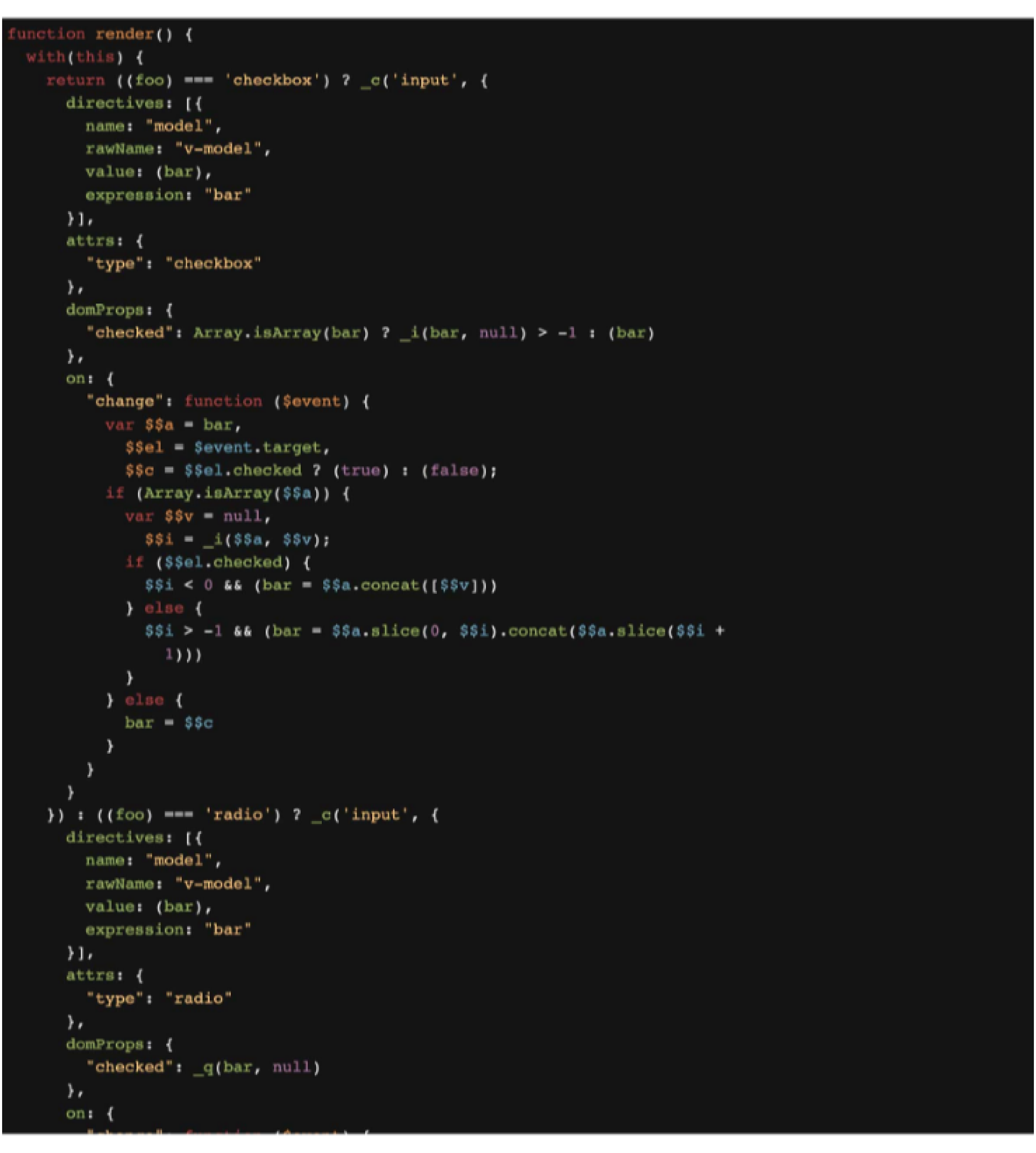
```
<input type="text" v-model="myValue" />

export default {
  data() {
    myValue: null
  }
}
```

This strategy works fairly well, but what if our component uses a dynamic value to set the type of input?

I’m sure you’ve been in a situation where you have created an input component that can either be a type of `input` for someone’s name, for example, and that with the change of a property you use it as a type `email` for their email address.

How does it compile?

Because Vue 2 cannot “guess” what type of element this is going to be at runtime, due to the possibility of the data changing at any given time, the Vue 2 compiler is forced to output a very lengthy and verbose block of code to handle *every possible scenario.*



```
function render() {
  with(this) {
    return ((foo) === 'checkbox') ? _c('input', {
      directives: [{
        name: "model",
        rawName: "v-model",
        value: (bar),
        expression: "bar"
      }],
      attrs: {
        "type": "checkbox"
      },
      domProps: {
        "checked": Array.isArray(bar) ? _i(bar, null) > -1 : (bar)
      },
      on: {
        "change": function ($event) {
          var $$a = bar,
            $$el = $event.target,
            $$c = $$el.checked ? (true) : (false);
          if (Array.isArray($$a)) {
            var $$v = null,
              $$i = _i($$a, $$v);
            if ($$el.checked) {
              $$i < 0 && (bar = $$a.concat([$$v]))
            } else {
              $$i > -1 && (bar = $$a.slice(0, $$i).concat($$a.slice($$i +
                1)))
            }
          } else {
            bar = $$c
          }
        }
      }
    }) : ((foo) === 'radio') ? _c('input', {
      directives: [{
        name: "model",
        rawName: "v-model",
        value: (bar),
        expression: "bar"
      }],
      attrs: {
        "type": "radio"
      },
      domProps: {
        "checked": _q(bar, null)
      },
      on: {
```

Don’t worry, we don’t need to go over every line of code. Just know that it basically has to prepare for every type of possible scenario.

In Vue 3, outputting this amount of code is no longer necessary because `v-model` for input elements behaves almost the same way it does for custom components — with an extra module that helps Vue decide which prop/event to apply in each case.

The compiled result in comparison is incredibly smaller.

```
h('input', {
  modelValue: myValue,
  'onUpdate:modelValue': value => {
    myValue = value
  }
})
```

The new defaults

In Vue 3, when creating a component that has `v-model` capabilities we need to use a new set of defaults for creating the `v-model` binding.

In Vue 2, no matter what type of native input you were binding to inside the component you would always bind the value of your data to a `value` property, and you would listen to an `input` event.

Of course there was a way to modify this default behaviour by declaring a `model` property in our Vue component, but that’s the Vue 2 API and we’re not going to look in depth into it.

In Vue 3, we now have two new defaults. For the `prop` that binds the input of the value, we use `modelValue`, and for the emitted event we use `update:modelValue`.

I want you to play close attention to the names of the events before you panic about the verboseness of these new defaults. Did you notice how `modelValue` is actually present in both of them?

The new emit default `update:modelValue` can be extracted into two different parts. The declaration that something is being updated `update:` and the model that is being updated `modelValue`.

This is going to play a very important role later on in the course, when we look at how to create multiple `v-model` bindings into a single component!

Now, let’s take a look at how the code of a simple `input` wrapper component might look with the new Vue 3 `v-model` syntax.

```
<template>
  <input
    :value="modelValue"
    @input="$emit(
      'update:modelValue',
      $event.target.value
    )"
  >
</template>

<script>
export default {
  props: {
    modelValue: {
      type: [String, Number],
      default: ''
    }
  }
}
</script>
```

Using the new v-model in component instances

Now that we have the base for a `v-model` capable component, let’s take a look at how we would use it in an application.

As with any other component, we need to import it and declare it in the `components` object for our parent.

```
<template>
  <div>

  </div>
</template>

<script>
import BaseInput from './BaseInput'

export default {
  components: { BaseInput }
}
</script>
```

Next, we add the component to our `template` so that we can declare the `v-model` on top of it. After, we are going to use the object syntax (no composition API this time!) and use a `data` object to create a simple reactive state.

```
<template>
  <div>
    <BaseInput
      v-model="myInput"
    />
  </div>
</template>

<script>
import BaseInput from './BaseInput'

export default {
  components: { BaseInput },
  data() {
    return {
      myInput: ''
    }
  }
}
</script>
```

At this point you’re wondering, we’ll that’s all right and nice, but I already know this!

Fair enough, but I wanted to show you one last thing before we wrapped up our lesson. Using `v-model` like this is now actually a shorthand! The binding has been modified to now accept an intermediate parameter before the binding.

So `v-model="myInput"` is actually now a shorthand for `v-model:modelValue="myInput"`

You’re probably wondering at this point why this is important or actually useful. In the next lesson when we dive into more advanced parts of the new `v-model` system, we’re going to talk about *multi* `v-model` bindings, and this new syntax is going to play a very important role.

Coming up next

Now that we understand the basics of the new `v-model` system and its improved bindings, let’s go into the next lesson and look at a couple other new tools that it provides us for component development.

We will finally look at the multi `v-model` capabilities that I’ve been hinting at, and a cool new feature to build our own modifiers.

Can you think of any components in your current Vue 2 applications that will benefit or be able to be enhanced already by these new features?

See you in the next lesson!