

# DESIGN PATTERNS FOR VUE.JS

**A test driven approach  
to maintainable applications**

Lachlan Miller



# Contents

<b>1</b>	<b>About the Book</b>	<b>4</b>
1.1	About the Author . . . . .	4
<b>2</b>	<b>Design Patterns for Vue.js - a Test Driven Approach to Main- tainable Applications</b>	<b>5</b>
2.1	Introduction . . . . .	5
2.2	The Book . . . . .	5
2.3	What To Expect . . . . .	6
<b>3</b>	<b>Patterns for Testing Props</b>	<b>8</b>
3.1	The Basics . . . . .	9
3.2	Adding a Validator . . . . .	11
3.3	Key Concept: Separation of Concerns . . . . .	13
3.4	Separation of Concerns - Case Study . . . . .	14
3.5	Another Example . . . . .	16
3.6	The real test: Does it refactor? . . . . .	19
3.7	Conclusion . . . . .	20
<b>4</b>	<b>Emitting Events</b>	<b>21</b>
4.1	Starting Simple . . . . .	21
4.2	Clean Templates . . . . .	23
4.3	Declaring emits . . . . .	24
4.4	More Robust Event Validation . . . . .	25
4.5	With the Composition API . . . . .	27
4.6	Conclusion . . . . .	28
<b>5</b>	<b>Writing Testable Forms</b>	<b>29</b>
5.1	The Patient Form . . . . .	30
5.2	A Mini Form Validation Framework . . . . .	33
5.3	The <code>required</code> validator . . . . .	34
5.4	The <code>isBetween</code> validator . . . . .	35
5.5	Building <code>validateMeasurement</code> with <code>isBetween</code> . . . . .	37
5.6	The Form Object and Full Form Validation . . . . .	39
5.7	Writing the UI Layer . . . . .	44
5.8	Some Basic UI Tests . . . . .	48
5.9	Improvements and Conclusion . . . . .	49
5.10	Exercises . . . . .	49
<b>6</b>	<b>HTTP and API Requests</b>	<b>50</b>
6.1	The Login Component . . . . .	50
6.2	Starting Simple . . . . .	50
6.3	Refactoring to a store . . . . .	52
6.4	To mock or not to mock? . . . . .	55
6.5	Mock Less - mock the lowest dependency in the chain . . . . .	57
6.6	Mock Service Worker . . . . .	58

6.7	Conclusion . . . . .	61
6.8	Exercises . . . . .	62
<b>7</b>	<b>Renderless Components</b>	<b>63</b>
7.1	Rendering without Markup . . . . .	64
7.2	Adding Password and Confirmation Inputs . . . . .	66
7.3	Adding Password Complexity . . . . .	69
7.4	Computing Form Validity . . . . .	74
7.5	Exercises . . . . .	78
<b>8</b>	<b>The Power of Render Functions</b>	<b>79</b>
8.1	Why Render Functions? . . . . .	81
8.2	Creating the Components . . . . .	81
8.3	Filtering Slots by Component . . . . .	83
8.4	Filtering default slots . . . . .	89
8.5	Adding Attributes to Render Functions . . . . .	91
8.6	What is h? A Crash Course . . . . .	93
8.7	Adding a Dynamic Class Attribute . . . . .	95
8.8	Event Listeners in Render Functions . . . . .	98
8.9	Filtering Content . . . . .	100
8.10	Testing Render Function Components . . . . .	102
8.11	Exercises . . . . .	102
<b>9</b>	<b>Dependency Injection with Provide and Inject</b>	<b>104</b>
9.1	A Simple Store . . . . .	106
9.2	Usage via import . . . . .	107
9.3	Adding a users forms . . . . .	109
9.4	Provide/Inject to Avoid Cross Test Contamination . . . . .	112
9.5	Provide in Testing Library . . . . .	115
9.6	A useStore composable . . . . .	116
9.7	Exercises . . . . .	118
<b>10</b>	<b>Modular Components, v-model, and the Strategy Pattern</b>	<b>119</b>
10.1	Foundations of v-model . . . . .	123
10.2	Deserializing for modelValue . . . . .	125
10.3	Deserializing modelValue . . . . .	126
10.4	Serializing modelValue . . . . .	130
10.5	Error Handling . . . . .	134
10.6	Deploying . . . . .	135
10.7	Exercises . . . . .	135
<b>11</b>	<b>Grouping Features with Composables</b>	<b>136</b>
11.1	Defining the Initial Board . . . . .	138
11.2	Computing the Current State . . . . .	140
11.3	Tests . . . . .	142
11.4	Setting an Initial State . . . . .	142

11.5 Making a Move . . . . .	143
11.6 Conclusion . . . . .	148
11.7 Exercises . . . . .	148
<b>12 Functional Core, Imperative Shell - Immutable Logic, Mutable Vue</b>	<b>149</b>
12.1 Functional Core, Imperative Shell . . . . .	150
12.2 Business Logic - The Functional Core . . . . .	151
12.3 Immutable <code>makeMove</code> . . . . .	152
12.4 Vue Integration - Imperative Shell . . . . .	154
12.5 Integrating <code>makeMove</code> . . . . .	157
12.6 Pushing Business Logic into the Functional Core . . . . .	159
12.7 Reflections and Philosophy . . . . .	161
12.8 Exercises . . . . .	162

# 1 About the Book

This book is aimed at developers who are comfortable with JavaScript and Vue.js. It focuses on ideas and patterns rather than specific APIs. Separation of concerns, pure functions, writing for testability and re-usability are some of the primary motifs.

The examples are written with Vue.js 3, the latest version of Vue.js. Both the classic Options API and new Composition API are covered. Tests are written with Vue Testing Library: <https://github.com/testing-library/vue-testing-library>. If you prefer Vue Test Utils, no problem - the source code contains the same tests written using both Testing Library and Vue Test Utils.

The final source code including the solutions to the exercises is available here: <https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>

## 1.1 About the Author

Lachlan Miller is a full stack software developer based in Brisbane, Australia. He is passionate about open source and mentoring. His primary areas of interest is testing, software quality and design patterns.

He picked up Vue.js in 2016 and was immediately hooked. He has been contributing to the Vue.js ecosystem since, and is the primary maintainer of several popular libraries, Vue Test Utils as the most notable.

He also has a YouTube channel where he posts advanced content, similar to that found in this book. He has made two full length courses, enjoyed by over 13000 students.

Lachlan's website: <https://lachlan-miller.me>.

Lachlan's YouTube channel: <https://www.youtube.com/c/LachlanMiller>.

Lachlan's Udemy Profile: <https://www.udemy.com/user/lachlan-miller-4/>

## 2 Design Patterns for Vue.js - a Test Driven Approach to Maintainable Applications

### 2.1 Introduction

Chances are, if you are interested in something as abstract as *design patterns*, you have been coding for a while now, likely with an interest in front-end development. You might have written a few interfaces using jQuery, that seemed pretty good. As time passed, requirements changed, and your once maintainable jQuery code-base has become a mess.

At this point, you might have looked around for an alternative. Things like React and Vue are often the next step - Components! Encapsulation! Unit Tests! You **vue create app** and things are great!

Again, time passes - requirements change. Things get messy - again. It's a more organized, less crazy kind of messy, but it still doesn't feel *right*. You end up with giant components ("god components"), that are nearing a thousand lines, doing everything from fetching data, validation and everything else you can imagine. Passing tens of props down and emitting tens of events back up becomes normal. A "single source of truth" starts to become "several sources of truth". The lines between your business logic, the problem you are solving and the presentation components starts to blur. Eventually velocity slows, and subtle bugs start to creep in.

A few years later, all the original developers are gone and no-one really knows how things work, or what exactly `// TODO: fix this` actually refers to. The business opts to do a major rewrite, and the cycle continues.

*This isn't unusual.* Maybe jQuery wasn't the problem after all?

It doesn't have to be like this! Vue is a powerful and flexible UI layer, JavaScript (and TypeScript) get improvements every year and Jest is a first class test runner. All the tools needed to write reliable, maintainable and bug free applications are available. What's often missing is the design patterns, best practices, proper separation of concerns, and a reliable test suite. These are the fundamental ideas that underpin all software development, not just front-end applications.

### 2.2 The Book

This is a book about design patterns and testing. But it's also more. Thinking in design patterns is not about memorizing a lot of fancy names and diagrams. Knowing how to test is not really about learning a test runner or reading documentation.

Thinking in patterns, consider how data flows between different parts of a system and writing for testability starts *before* writing any code.

Good software design is a philosophy. It's a way of life. Finally, as engineer, writing good software *is your job*. So is writing testable code - even if HR forgot to put it in your job description.

My goal is to get you in the habit of writing testable code, and how to choose the right abstraction for the problem at hand. The first things you think when you hear a new business requirement or request should be:

- What design pattern will give me the most flexibility moving forward?
- What new requirements could come up, and how will this decision deal with them?
- How am I going to write my code in a testable, loosely coupled fashion?

The lessons and patterns I'll be sharing are not Vue-specific at all; they are framework agnostic. I'd even say that they are language agnostic; they are fundamental concepts you can take with you and apply them to any software design problem. Good developers focus on tools and frameworks, great developers focus on data structures and how they interact with each other, testability and maintainability.

All of the content is, of course, based on my opinion. Like most best practices and design patterns, there is a time and place for everything, and not every recommendation will apply to every use case. The best way to get value from this book is to read the examples, think about the concepts and compare it to what you are currently doing.

If you think it solves a problem you have, try it out and see where it leads. There will always be exceptions, but the most important thing is you think about your concerns, testability and reliability. If you disagree with something you see, I'm always open to discussion and improvements; please reach out. I'm always interested in new perspectives and ideas.

## 2.3 What To Expect

Most books that teach you frameworks, languages or testing will be an app or two, incrementally adding new features. This works great for learning a new language or framework, but isn't ideal for focusing on concepts or ideas. In this book, each section will be focused on a single idea, and we will build a small component or application to illustrate it. This approach has a few benefits; you can read the content in any order, and use it as a kind of reference.

We start with some patterns for **props**, as well as a discussion around one of the most fundamental ideas in this book, *separation of concerns*. We proceed to cover a wide variety of design patterns for events, forms, components, renderless components, feature separation with the Composition API, and everything else you'll need to know to create well engineered Vue.js applications.

Most sections end with some exercises to help you improve and practice what you learned. The source code, including all the solutions for the exercises are included in the source code: (<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>), so you can check your solutions.

Each section is independent; you don't need to read it in order, so if there is a particular section you are interested in, feel free to skip to it. Try to think of this book as a reference tool; I hope it is something you can come back to for years to come and learn something useful each time.

I hope this has given you a good idea of what to expect. If you have any feedback, questions or comments, or just want to chat about Vue and testing, feel free to reach out via email or Twitter (find my most up to date contact details on the website you got this book).

See you in the next section!



### 3 Patterns for Testing Props

You can find the completed source code in the GitHub repository under examples/props:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

---

In this section we explore **props**, and the kind of tests you might want to consider writing. This leads into a much more fundamental and important topic; drawing a clear line between business logic and UI, also known as *separation of concerns*, and how your tests can help make this distinction clear.

Consider one of the big ideas behind frameworks like Vue and React:

Your user interface is a function of your data.

This idea comes in many forms; another is “data driven interfaces”. Basically, your user interface (UI) should be determined by the data present. Given X data, your UI should be Y. In computer science, this is referred to as *determinism*. Take this **sum** function for example:

```
function sum(a, b) {  
  return a + b  
}
```

A simple sum function. It’s a pure function.

When called with the same value for **a** and **b**, you always get same result. The result is pre-determined. It’s *deterministic*. An example of an impure function would be this:

```
async function fetchData(userId) {  
  return axios.get(`/api/users/${userId}`)  
}
```

A impure function - it has a side effect. Not ideal, but necessary for most systems to do anything useful.

This is *not* a pure function because it relies on an external resource - in this case an API and a database. Depending on what is in the database when it is called, we might get a different result. It’s *unpredictable*.

How does this relate to **props**? Think of a component that decides what to render based on it’s **props** (don’t worry about **data**, **computed** or **setup** for now - but the same ideas apply, as you’ll see throughout the book). If you think of a component as a function and the **props** as the arguments, you’ll realize that given the same **props**, the component will always render the same thing. It’s output is deterministic. Since you decide what **props** are passed to the component, it’s easy to test, since we know all the possible states the component can be in.

### 3.1 The Basics

You can declare props in a few ways. We will work with the `<message>` component for this example. You can find it under `examples/props/message.vue`.

```
<template>
  <div :class="variant">Message</div>
</template>

<script>
export default {
  // can be 'success', 'warning', 'error'
  props: ['variant']
}
</script>
```

Declaring a variant prop with the inferior array syntax.

In this example we declare props using the array syntax: `props: ['variant']`. I recommend avoiding the array syntax. Using the object syntax gives the reader more insight into the type of values `variant` can take:

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

Declaring a variant prop with the superior object syntax.

If you are using TypeScript, even better - create a type:

```
type Variant = 'success' | 'warning' | 'error'

export default {
  props: {
    variant: {
      type: String as () => Variant,
      required: true
    }
  }
}
```

A strongly typed variant prop using TypeScript.

In our `<message>` example, we are working with regular JavaScript, so we cannot

specify specific strings for the **variant** props like you can in TypeScript. There are some other patterns we can use, though.

We have specified the **variant** prop is **required**, and we would like to enforce a specific subset of string values that it can receive. Vue allows us to validate props using a **validator** key. It works like this:

```
export default {
  props: {
    variant: {
      validator: (val) => {
        // if we return true, the prop is valid.
        // if we return false, a runtime warning will be shown.
      }
    }
  }
}
```

Prop validators are functions. If they return false, Vue will show a warning in the console.

Prop validators are like the **sum** function we talked about earlier in that they are pure functions! That means they are easy to test - given X prop, the validator should return Y result.

Before we add a validator, let's write a simple test for the `<message>` component. We want to test *inputs* and *outputs*. In the case of `<message>`, the **variant** prop is the input, and what is rendered is the output. We can write a test to assert the correct class is applied using Testing Library and the `classList` attribute:

```
import { render, screen } from '@testing-library/vue'
import Message, { validateVariant } from './message.vue'

describe('Message', () => {
  it('renders variant correctly when passed', () => {
    const { container } = render(Message, {
      props: {
        variant: 'success'
      }
    })

    expect(container.firstChild.classList).toContain('success')
  })
})
```

Testing the prop is applied to the class.

This verifies everything works as expected when a valid **variant** prop is passed to `<message>`. What about when an invalid **variant** is passed? We want to

prohibit using the `<message>` component with a valid `variant`. This is a good use case for a `validator`.

## 3.2 Adding a Validator

Let's update the `variant` prop to have a simple validator:

```
export default {
  props: {
    variant: {
      type: String,
      required: true,
      validator: (variant) => {
        if (!['success', 'warning', 'error'].includes(variant)) {
          throw Error(
            `variant is required and must` +
            `be either 'success', 'warning' or 'error'.` +
            `You passed: ${variant}`
          )
        }

        return true
      }
    }
  }
}
```

If the variant is not valid, we throw an error.

Now we will get an error if an invalid prop is passed. An alternative would be just to return `false` instead of throwing an error - this will just give you a warning in the console via `console.warn`. Personally, I like loud and clear errors when a component isn't used correctly, so I chose to throw an error.

How do we test the validator? If we want to cover all the possibilities, we need 4 tests; one for each `variant` type, and one for an invalid type.

I prefer to test prop validators in isolation. Since validators are generally pure functions, they are easy to test. There is another reason I test prop validators in isolation too, which we will talk about after writing the test.

To allow testing the validator in isolation, we need to refactor `<message>` a little to separate the validator from the component:

```
<template>
  <div :class="variant">Message</div>
</template>
```

```

<script>
export function validateVariant(variant) {
  if (!['success', 'warning', 'error'].includes(variant)) {
    throw Error(
      `variant is required and must` +
      `be either 'success', 'warning' or 'error'.` +
      `You passed: ${variant}`
    )
  }

  return true
}

export default {
  props: {
    variant: {
      type: String,
      required: true,
      validator: validateVariant
    }
  }
}
</script>

```

Exporting the validator separately to the component.

Great, `validateVariant` is now exported separately and easy to test:

```

import { render, screen } from '@testing-library/vue'
import Message, { validateVariant } from './message.vue'

describe('Message', () => {
  it('renders variant correctly when passed', () => {
    // omitted for brevity ...
  })

  it('validates valid variant prop', () => {
    ;['success', 'warning', 'error'].forEach(variant => {
      expect(() => validateVariant(variant)).not.toThrow()
    })
  })

  it('throws error for invalid variant prop', () => {
    expect(() => validateVariant('invalid')).toThrow()
  })
})

```

Testing all the cases for the validator.

Simply making the `validateVariant` a separate function that is exported might seem like a small change, but it's actually a big improvement. By doing so, we were able to write tests for `validateVariant` with ease. We can be confident the `<message>` component can only be used with valid a `variant`.

If the developer passes an invalid prop, they get a nice clear message in the console:

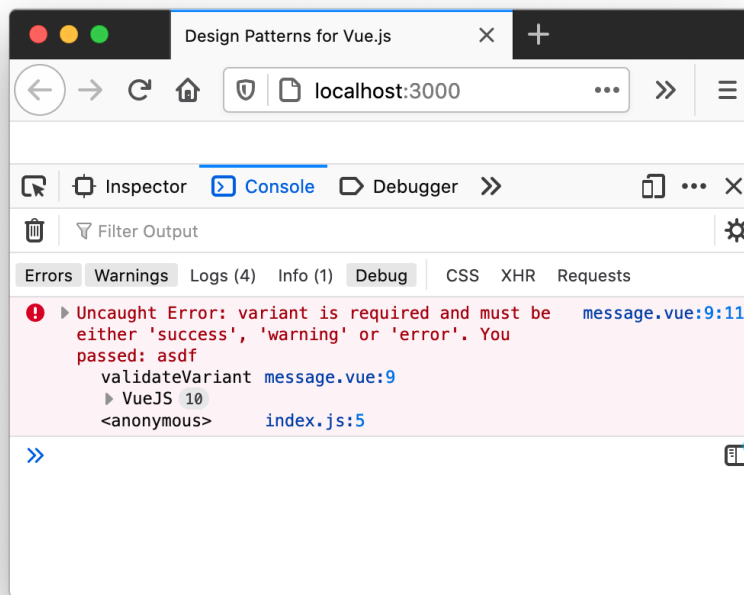


Figure 1: Error! Passed variant is invalid.

### 3.3 Key Concept: Separation of Concerns

We have written two different types of tests. The first is a UI test - that's the one where we make an assertions against `classList`. The second is for the validator. It tests business logic.

To make this more clear, imagine your company specializes in design systems. You have some designers who probably use Figma or Sketch to design things like buttons and messages.

They have decided to support for three message variants: success, warning and error. You are a front-end developer. In this example, you are working on the Vue integration - you will write Vue components that apply specific classes, which use the CSS you provided by the designers.

In the future, you also need to build React and Angular components using the same CSS and guidelines. All three of the integrations could make use of the `validateVariant` function and test. It's the core business logic.

This distinction is important. When we use Testing Library methods (such as `render`) and DOM APIs (like `classList`) we are verifying that the Vue UI layer is working correctly. The test for `validateVariant` is for our business logic. These differences are sometimes called *concerns*. One piece of code is concerned with the UI. The other is concerned with the business logic.

Separating them is good. It makes your code easier to test and maintain. This concept is known as *separation of concerns*. We will revisit this throughout the book.

If you want to know if something is part of the UI or business logic, ask yourself this: "if I switched to React, would I be able to re-use this code and test?"

In this case, you could reuse the validator and it's test when you write the React integration. The validator is concerned with the business logic, and doesn't know anything about the UI framework. Vue or React, we will only support three message variants: success, warning and error. The component and component test (where we assert using `classes()`) would have to be rewritten using a React component and React testing library.

Ideally, you don't want your business logic to be coupled to your framework of choice; frameworks come and go, but it's unlikely the problems your business is solving will change significantly.

I have seen poor separation of concerns costs companies tens of thousands of dollars; they get to a point where adding new features is risky and slow, because their core business problem is too tightly coupled to the UI. Rewriting the UI means rewriting the business logic.

### 3.4 Separation of Concerns - Case Study

One example of poor separation of concerns costing an organization was an application I worked on for an electrical components supplier. They had an application customers would use to get an approximate quote for the price of components. The ordering process was quite complex - you would go through a form with several steps, and the values from the previous step would impact the fields on the next step.

The application was written using jQuery (which is not bad. No framework is bad - only if they are used incorrectly). All of the business logic was mixed in

with the UI logic (this is the bad part). They had a quantity based discount model - “If purchasing more than 50 resistors, then apply X discount, otherwise Y” - this kind of thing. They decided to move to something a bit more modern - the UI was very dated, and wasn’t mobile friendly at all. The complexity of the jQuery code was high and the code was a mess.

Not only did I need to rewrite the entire UI layer (which was what I was paid to do), but I ended up having to either rewrite or extract the vast majority of the business logic from within the jQuery code, too. This search and extract mission made the task much more difficult and risky than it should have been - instead of just updating the UI layer, I had to dive in and learn their business and pricing model as well (which ended up taking a lot more time and costing a lot more than it probably should have).

Here is a concrete example using the above real-world scenario. Let’s say a resistor (a kind of electrical component) costs \$0.60. If you buy over 50, you get a 20% discount. The jQuery code-base looked something like this:

```
const $resistorCount = $('#resistors-count')

$resistorCount.change((event) => {
  const amount = parseInt(event.target.value)
  const totalCost = 0.6 * amount
  const $price = $('#price')
  if (amount > 50) {
    $price.value(totalCost * 0.8)
  } else {
    $price.value(totalCost)
  }
})
```

You need to look really carefully to figure out where the UI ends and the business starts. In this scenario, I wanted to move to Vue - the perfect tool for a highly dynamic, reactive form. I had to dig through the code base and figure out this core piece of business logic, extract it, and rewrite it with some tests (of course the previous code base had no tests, like many code bases from the early 2000s). This search-extract-isolate-rewrite journey is full of risk and the chance of making a mistake or missing something is very high! What would have been much better is if the business logic and UI had been separated:

```
const resistorPrice = 0.6
function resistorCost(price, amount) {
  if (amount > 50) {
    return price * amount * 0.8
  } else {
    return price * amount
  }
}
```



```
$resistorCount.change((event) => {
  const amount = parseInt(event.target.value)
  $("#price").value = resistorCost(resistorPrice, amount)
})
```

The second is far superior. You can see where the business logic ends and the UI begins - they are literally separated in two different functions. The pricing strategy is clear - a discount for any amount greater than 50. It's also very easy to test the business logic in isolation. If the day comes you decide your framework of choice is no longer appropriate, it's trivial to move to another framework - your business logic unit tests can remain unchanged and untouched, and hopefully you have some end-to-end browser tests as well to keep you safe.

Moving to Vue is trivial - no need to touch the business logic, either:

```
<template>
  <input v-model="amount" />
  <div>Price: {{ totalCost }}</div>
</template>

<script>
import { resistorCost, resistorPrice } from './logic.js'

export default {
  data() {
    return {
      amount: 0
    }
  },
  computed: {
    totalCost() {
      return resistorCost(resistorPrice, this.amount)
    }
  }
}
</script>
```

Understanding and identifying the different concerns in a system and correctly structuring applications is the difference good engineers and great engineers.

### 3.5 Another Example

Enough design philosophy for now. Let's see another example related to props. This examples uses the `<navbar>` component. You can find it in `examples/props/navbar.vue`. It looks like this:

```

<template>
  <button v-if="authenticated">Logout</button>
  <button v-if="!authenticated">Login</button>
</template>

<script>
export default {
  props: {
    authenticated: {
      type: Boolean,
      default: false
    }
  }
}
</script>

```

The navbar component. It has one prop, `authenticated`. It is false by default. Before even seeing the test, it is clear we need *two* tests to cover all the use cases. The reason this is immediately clear is the `authenticated` prop is a `Boolean`, which only has two possible values. The test is not especially interesting (but the discussion that follows is!):

```

import { render, screen } from '@testing-library/vue'
import Navbar from './navbar.vue'

describe('navbar', () => {
  it('shows logout when authenticated is true', () => {
    render(Navbar, {
      props: {
        authenticated: true
      }
    })

    // getByText will throw an error if it cannot find the element.
    screen.getByText('Logout')
  })

  it('shows login by default', () => {
    render(Navbar)

    screen.getByText('Login')
  })
})

```

Testing the navbar behavior for all values of `authenticated`.

The only thing that changes based on the value of `authenticated` is the button text. Since the `default` value is `false`, we don't need to pass it as in `props` in the second test.

We can refactor a little with a `renderNavbar` function:

```
describe('Navbar', () => {
  function renderNavbar(props) {
    render(Navbar, {
      props
    })
  }

  it('shows login authenticated is true', () => {
    renderNavbar({ authenticated: true })
    screen.getByText('Logout')
  })

  it('shows logout by default', () => {
    renderNavbar()
    screen.getByText('Login')
  })
})
```

More concise tests.

I like this version of the test better. It might seem a little superficial for such a simple test, but as your components become more complex, having a function to abstract away some of the complexity can make your tests more readable.

I also removed the new line between the rendering the component and making the assertion. I usually don't leave any new lines in my tests when they are this simple. When they get more complex, I like to leave some space - I think it makes it more readable. This is just my personal approach. The important thing is not your code style, but that you are writing tests.

Although we technically have covered all the cases, I like to add the third case: where `authenticated` is explicitly set to `false`.

```
describe('navbar', () => {
  function renderNavbar(props) {
    render(Navbar, {
      props
    })
  }

  it('shows login authenticated is true', () => {
    // ...
  })
})
```

```

it('shows logout by default', () => {
  // ...
})

it('shows login when authenticated is false', () => {
  renderNavbar({ authenticated: false })
  screen.getByText('Login')
})
})

```

Adding a third test to be explicit.

This, of course, passes. I really like the symmetry the three tests exhibit, showing all three cases in such a concise manner.

Let's revisit the idea of separation of concerns; is this a UI test or business logic test? If we moved framework, could we re-use this test?

The answer is *no* - we'd need to write a new test (to work with React and it's Testing Library integration). This is fine - it just means this part of our codebase is part of the UI layer, not our core business logic. Nothing to extract.

### 3.6 The real test: Does it refactor?

We can do a little sanity check and make sure our tests are not testing implementation details. Implementation details refers to *how* something works. When testing, we don't care about the specifics of how something works. Instead, we care about *what it does*, and if it does it correctly. Remember, we should be testing that we get the expected output based on given inputs. In this case, we want to test that the correct text is rendered based on the data, and not caring too much about how the logic is actually implemented.

We can validate this by refactoring the `<navbar>` component. As long as the tests continue to pass, we can be confident they are resilient to refactors and are testing behaviors, not implementation details.

Let's refactor `<navbar>`:

```

<template>
  <button>
    {{ `${authenticated ? 'Logout' : 'Login'}` }}
  </button>
</template>

<script>
export default {
  props: {

```

```

    authenticated: {
      type: Boolean,
      default: false
    }
  }
}
</script>

```

Refactoring navbar. The behavior is still the same!

Everything still passes! Our tests are doing what they are supposed to be. Or are they? What if we decide we would like to use a `<a>` tag instead of a `<button>`?

```

<template>
  <a>{{ `${authenticated ? 'Logout' : 'Login'}` }}</a>
</template>

<script>
export default {
  props: {
    authenticated: {
      type: Boolean,
      default: false
    }
  }
}
</script>

```

Using an anchor tag instead of a button.

Obviously in a real system a `href` property would be required and change depending on `authenticated`, but that isn't what we are focusing on here. It still passes. Great news! Our tests survived two refactors - this means we are testing the behavior, not the implementation details, which is good.

### 3.7 Conclusion

This chapter discussed some techniques for testing props. We also saw how to use Testing Library's `render` method to test components. We touched on the concept of *separation of concerns*, and how it can make your business logic more testable and your applications more maintainable. Finally, we saw how tests can let us refactoring code with confidence.

You can find the completed source code in the GitHub repository under `examples/props`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

## 4 Emitting Events

You can find the completed source code in the GitHub repository under examples/events:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

---

Vue's primary mechanic for passing data *down* to components is **props**. In contrast, when components needs to communicate with another component higher in the hierarchy, you do so by *emitting events*. This is done by calling `this.$emit()` (Options API) or `ctx.emit()` (Composition API).

Let's see some examples on how this works, and some guidelines we can set to keep things clean and understandable.

### 4.1 Starting Simple

Here is a very minimal yet perfectly working `<counter>` component. It is not ideal; we will work on improving it during this section.

This example starts with the Options API; we will eventually refactor it to use the Composition API (using the tests we write to ensure we don't break anything).

```
<template>
  <button role="increment" @click="count += 1" />
  <button role="submit" @click="$emit('submit', count)" />
</template>

<script>
export default {
  data() {
    return {
      count: 0
    }
  }
}
</script>
```

A simple counter component.

There are two buttons. One increments the `count` value by 1. The other emits a `submit` event with the current count. Let's write a simple test that will let us refactor with the confidence.

As with the other examples, this one uses Testing Library, but you could really use any testing framework - the important part is that we have a mechanism to

let us know if we break something.

```
import { render, screen, fireEvent } from '@testing-library/vue'
import Counter from './counter.vue'

describe('Counter', () => {
  it('emits an event with the current count', async () => {
    const { emitted } = render(Counter)
    await fireEvent.click(screen.getByRole('increment'))
    await fireEvent.click(screen.getByRole('submit'))
    console.log(emitted())
  })
})
```

Observing the emitted events with `emitted()`.

I did a `console.log(emitted())` to illustrate how `emitted` works in Testing Library. If you run the test, the console output is as follows:

```
{
  submit: [
    [ 1 ]
  ]
}
```

A submit event was emitted with one argument: the number 1.

`emitted` is an object - each event is a key, and it maps to an array with an entry for each time the event was emitted. `emit` can have any amount of arguments; if I had written `$emit('submit', 1, 2, 3,)` the output would be:

```
{
  submit: [
    [ 1, 2, 3 ]
  ]
}
```

A submit event was emitted with three arguments, 1, 2, 3.

Let's add an assertion, before we get onto the main topic: patterns and practices for emitting events.

```
import { render, screen, fireEvent } from '@testing-library/vue'
import Counter from './counter.vue'

describe('Counter', () => {
  it('emits an event with the current count', async () => {
    const { emitted } = render(Counter)

    await fireEvent.click(screen.getByRole('increment'))
    await fireEvent.click(screen.getByRole('submit'))

    expect(emitted().submit[0]).toEqual([1])
  })
})
```

Making an assertion against the emitted events.

## 4.2 Clean Templates

Templates can often get chaotic among passing props, listening for events and using directives. For this reason, wherever possible, we want to keep our templates simple by moving logic into the `<script>` tag. One way we can do this is to avoid writing `count += 1` and `$emit()` in `<template>`. Let's make this change in the `<counter>` component, moving the logic from `<template>` into the `<script>` tag by creating two new methods:

```
<template>
  <button role="increment" @click="increment" />
  <button role="submit" @click="submit" />
</template>

<script>
export default {
  data() {
    return {
      count: 0
    }
  },
  methods: {
    submit() {
      this.$emit('submit', this.count)
    },
    increment() {
      this.count += 1
    }
  }
}
```



```
}  
</script>
```

Moving the emit logic from the template to the script.

Running the test confirms that everything is still working. This is good. Good tests are resilient to refactors, since they test inputs and outputs, not implementation details.

I recommend you avoid putting any logic into `<template>`. Move everything into `<script>`. `count += 1` might seem simple enough to inline in `<template>`. That said, I personally value consistency over saving a few key strokes, and for this reason I put all the logic inside `<script>` - no matter how simple it is.

Another thing you may have noticed is the *name* of the method we created - `submit`. This is another personal preference, but I recommend having a good convention around naming methods. Here are two I've found useful.

1. Name the method that emits the event the same as the event name. If you are doing `$emit('submit')`, you could name the method that calls this `submit`, too.
2. Name methods that call `$this.emit()` or `ctx.emit()` using the convention `handleXXX`. In this example, we could name the function `handleSubmit`. The idea is those methods *handle* the interactions and emit the corresponding event.

Which of these you choose isn't really important; you could even pick another convention you like better. Having a convention is generally a good thing, though. Consistency is king!

### 4.3 Declaring emits

As of Vue 3, you are able to (and encouraged to) declare the events your component will emit, much like you declare props. It's a good way to communicate to the reader what the component does. Also, if you are using TypeScript, you will get better autocompletion and type safety.

Failing to do so will give you a warning in the browser console: *"Component emitted event" but it is neither declared in the emits option nor as an "prop"*.

By declaring the events a component emits, it can make it easier for other developers (or yourself in six months time) to understand what your component does and how to use it.

You can declare events in the same way you declare props; using the array syntax:

```
export default {
  emits: ['submit']
}
```

Declaring emits with the inferior array syntax.

Or the more verbose but explicit object syntax:

```
export default {
  emits: {
    submit: (count) => {}
  }
}
```

Declaring emits with the verbose but explicit object syntax.

If you are using TypeScript, you will get even better type safety with this syntax - including the types in the payload!

The object syntax also supports *validation*. As an example, we could validate the payload for an imaginary `submit` event is a number:

```
export default {
  emits: {
    submit: (count) => {
      return typeof count !== 'string' && !isNaN(count)
    },
  }
}
```

Validating the emitted event.

If the validator returns `false`, the event will not be emitted.

## 4.4 More Robust Event Validation

Depending on your application, you may want to have more thorough validation. I tend to favor defensive programming; I don't like taking chances, not matter how unlikely the scenario might seem.

Getting burned by a lack of defensive programming and making assumptions like “this will never happen in production” is something everyone has experienced. It's almost a rite of passage. There is a reason more experienced developers tend to be more cautious, write defensive code, and write lots of tests.

I also have a strong emphasis on testing, separation of concerns, and keeping things simple and modular. With these philosophies in mind, let's extract this validator, make it more robust, and add some tests.

The first step is to move the validation out of the component definition. For brevity, I am just going to export it from the component file, but you could move it to another module entirely (for example, a `validators` module).

```
<script>
export function submitValidator(count) {
  return typeof count !== 'string' && !isNaN(count)
}

export default {
  emits: {
    submit: submitValidator
  },
  data() {
    return {
      count: 0
    }
  },
  methods: {
    increment() {
      this.count += 1
    }
  }
}
</script>
```

A more robust validator with a custom validator function.

Another convention is emerging: I like to call event validators `xxxValidator`.

I am also going to make a change to `submitValidator`; the argument *must* be a number; if not, bad things will happen. So instead of waiting for bad things to happen, I am going to throw an error:

```
export function submitValidator(count) {
  if (typeof count === 'string' || isNaN(count)) {
    throw Error(`
      Count should be a number.
      Got: ${count}
    `)
  }
  return true
}
```

Defensive programming; failing loudly is good.

`submitValidator` is just a plain old JavaScript function. It's also a pure function - it's output is solely dependant on it's inputs. This means writing tests is trivial:

```
describe('submitValidator', () => {
  it('throws and error when count is NaN', () => {
    const actual = () => submitValidator('1')
    expect(actual).toThrow()
  })

  it('returns true when count is a number', () => {
    const actual = () => submitValidator(1)
    expect(actual).not.toThrow()
  })
})
```

Testing submitValidator in isolation.

A lot of these type specific validations can be partially mitigated with TypeScript. TypeScript won't give you runtime validation, though. If you are using an error logging service (like Sentry), throwing an error like this can give you valuable information for debugging.

## 4.5 With the Composition API

The <counter> example used the Options API. All the topics discussed here translate to the Composition API, too.

A good way to see if you are testing inputs and outputs, as opposed to implementation details, is to refactor your component from the Options API to the Composition API, or vice versa; good tests are resilient to refactor.

Let's see the refactor:

```
<template>
  <button role="increment" @click="increment" />
  <button role="submit" @click="submit" />
</template>

<script>
export function submitValidator(count) {
  if (typeof count === 'string' || isNaN(count)) {
    throw Error(`
      Count should be a number.
      Got: ${count}
    `)
  }
  return true
}

import { ref } from 'vue'
```

```

export default {
  emits: {
    submit: submitValidator
  },
  setup(props, { emit }) {
    const count = ref(0)

    const increment = () => {
      count.value += 1
    }
    const submit = () => {
      emit('submit', count.value)
    }

    return {
      count,
      increment,
      submit
    }
  }
}
</script>

```

The completed counter component with validation.

Everything still passes - great news!

## 4.6 Conclusion

We discussed emitting events, and the various features Vue provides to keep our components clean and testable. We also covered some of my favorite conventions and best practices to keep things maintainable in the long run, as well as bring consistency to your code base.

Finally, we saw how our tests was focused on inputs and outputs (in this case, the input is the user interaction via the buttons, and the output is the emitted `submit` event).

We touch on events again later on, in the `v-model` chapter - stay tuned.

You can find the completed source code in the GitHub repository under `examples/events`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

## 5 Writing Testable Forms

You can find the completed source code (including exercises) in the GitHub repository under `examples/form-validation`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

---

Forms are the primary way a user enters information into any web-based system, so getting them right is important. The focus of this section will be on forms, specifically *writing good forms*.

What exactly is a *good* form?

We want to ensure the form logic is decoupled from the Vue components - this will let us test in isolation. We also need to think about validation.

In traditional server-rendered apps, you would only get validation after submitting the form - not a great user experience. Vue allows us to deliver a great user experience by implementing highly dynamic, client-side validation. We will make use of this and implement two levels of validation:

1. Field validation - if a user enters incorrect or invalid data in a single field, we will show an error immediately.
2. Form validation - the submit button should only be enabled when the entire form is correctly filled out.

Finally, we need two types of tests. The first is around the business logic; given some form, which fields are invalid, and when is the form considered complete? The second is around interactions - ensuring that the UI layer is working correctly and that the user can enter data, see error messages, and submit the form if all the fields are valid.

## 5.1 The Patient Form

For this example, we are building a form to enter patient data for a hospital application. The form will look like this when filled out without any errors:

The screenshot shows a web browser window with the title 'Design Patterns for Vue.js'. The address bar indicates the page is at 'localhost:3000'. The form, titled 'Patient Data', contains two input fields: 'Name' with the value 'lachlan' and 'Weight' with the value '150'. The 'Weight' field has a unit selector set to 'lb'. Below the inputs is a 'Submit' button. At the bottom of the page, a JSON object displays the current state of the form data and validation status.

**Patient Data**

Name

Weight  lb

```
Patient Data
{
  "name": "lachlan",
  "weight": {
    "value": 150,
    "units": "lb"
  }
}

Form State
{
  "name": {
    "valid": true
  },
  "weight": {
    "valid": true
  }
}
```

Figure 2: Valid form with debug info

There are two inputs. The first is the patient name, which is required and can be any text. The second is the patient weight, which can be in imperial or metric units. The constraints are as follows:

Constraint	Imperial	Metric
min	66	30
max	440	200

We will need to validate both the name and the weight. The form with errors looks like this:



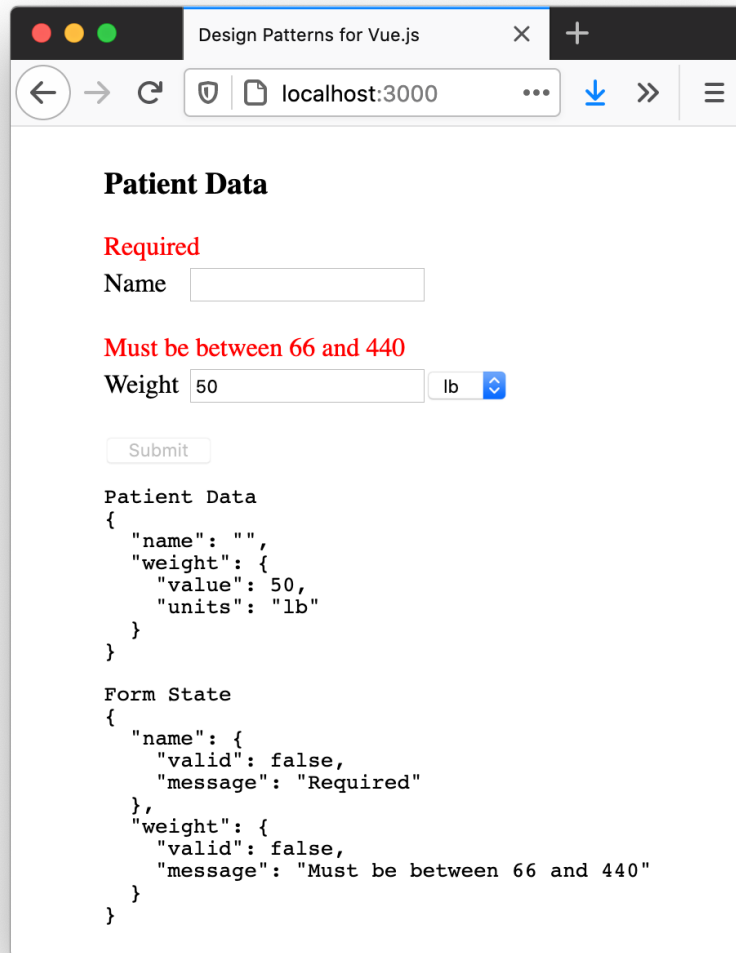


Figure 3: Invalid form with debug info

We will define the constraints using an object:

```
const limits = {  
  kg: { min: 30, max: 200 },  
  lb: { min: 66, max: 440 }  
}
```

The submit button should only be enabled if both inputs are valid. Finally, we should show validation for each field.

## 5.2 A Mini Form Validation Framework

There are plenty of full-featured Vue (and non-Vue) form validation frameworks out there. For this simple example, we will write our own - this will let us discuss some ideas, as well as avoid learning a specific API or library.

We need two types of validations:

1. A required field. Both the patient's name and weight are required fields.
2. Minimum and maximum constraints. This is for the weight field - it has to be within a specific range. It also needs to support metric and imperial units.

As well as validating the fields, our form validation framework should also return an error messages for each invalid input.

We will write two validation functions: **required** and **isBetween**. While test driven development (abbreviated to TDD - where you write the tests first, and let the failing tests guide the implementation) isn't always the right tool, for writing these two functions I believe it is. This is because we know the inputs and outputs, and all the possible states of the system, it's just a matter of writing the tests and then making them pass.

Let's do that - starting with the tests for the **required** validator. Each validator will return an object with the validation status, and a message if there is an error. A validated input should have this shape:

```
interface ValidationResult {  
  valid: boolean  
  message?: string  
}
```

This will be the format our two validators (and any future ones) will need to conform to. Now we've settled on our validation API, we can write the tests for **required**.

### 5.3 The required validator

```
import {
  required,
} from './form.js'

describe('required', () => {
  it('is invalid when undefined', () => {
    expect(required(undefined)).toEqual({
      valid: false,
      message: 'Required'
    })
  })

  it('is invalid when empty string', () => {
    expect(required('')).toEqual({
      valid: false,
      message: 'Required'
    })
  })

  it('returns true false value is present', () => {
    expect(required('some value')).toEqual({ valid: true })
  })
})
```

Tests for the required validator.

Basically, anything that does not evaluated to `true` is invalid; anything else is considered valid. We can get all the tests passing with this implementation:

```
export function required(value) {
  if (!value) {
    return {
      valid: false,
      message: 'Required'
    }
  }

  return { valid: true }
}
```

required validator implementation.

I like to check for the null case first, when `value` is not defined. That's just a personal preference.

## 5.4 The `isBetween` validator

`isBetween` is a bit more interesting. We need to support imperial and metric; we will build another function on top of `isBetween` that will pass in the correct constraints.

Let's start by identifying all the edge cases. If the minimum weight is 66 lb and the maximum weight is 440 lb, obviously 65 lb and 441 lb are invalid. 66 lb and 440 lb are valid, however, so we should make sure we add tests for those cases.

This means we need 5 tests:

1. The “happy” path, where the input is valid.
2. Value is above the maximum value.
3. Value is below the minimum value.
4. Value is equal to the maximum value.
5. Value is equal to the minimum value.

For this function, it is safe to assume that only numbers can be passed as the input value; this validation is something we will handle at a higher level.

```
import {
  required,
  isBetween
} from './form.js'

describe('required' () => {
  // ...
})

describe('isBetween', () => {
  it('returns true when value is equal to min', () => {
    expect(isBetween(5, { min: 5, max: 10 }))
      .toEqual({ valid: true })
  })

  it('returns true when value is between min/max', () => {
    expect(isBetween(7, { min: 5, max: 10 }))
      .toEqual({ valid: true })
  })

  it('returns true when value is equal to max', () => {
    expect(isBetween(10, { min: 5, max: 10 }))
      .toEqual({ valid: true })
  })

  it('returns false when value is less than min', () => {
    expect(isBetween(4, { min: 5, max: 10 }))
```

```

        .toEqual({
          valid: false,
          message: 'Must be between 5 and 10'
        })
      })

it('returns false when value is greater than max', () => {
  expect(isBetween(11, { min: 5, max: 10 }))
    .toEqual({
      valid: false,
      message: 'Must be between 5 and 10'
    })
})
})

```

Tests for the isBetween validator.

I think the tests are simple enough to have everything in a single `expect` statement. If the tests were more complex, I'd probably assign the result of `isBetween()` to a variable (I like to call it `actual`) and pass that to the `expect` assertion. More on structuring larger, more complex tests later.

The implementation is much less code than the tests; this is not unusual.

```

export function isBetween(value, { min, max }) {
  if (value < min || value > max) {
    return {
      valid: false,
      message: `Must be between ${min} and ${max}`
    }
  }

  return { valid: true }
}

```

isBetween validator implementation.

Again, I like to have the validation at the start of the function.

## 5.5 Building validateMeasurement with isBetween

Now we have written our little validation framework (well, two functions), it's time to validate the patient weight. We will build a `validateMeasurement` function using `isBetween` and `required`.

Since we are supporting imperial and metric, we will be passing the constraints as an argument. Dealing with which one is selected will be done later on, in the UI layer.

There are three scenarios to consider:

1. The happy path when the value is valid.
2. The value is null/undefined.
3. The value is defined, but outside the constraints.

I don't feel the need to add tests for all the cases as we did with `isBetween`, since we already tested that thoroughly.

```
import {
  required,
  isBetween,
  validateMeasurement
} from './form.js'

describe('required' () => {
  // ...
})

describe('isBetween', () => {
  // ...
})

describe('validateMeasurement', () => {
  it('returns invalid for input', () => {
    const constraints = { min: 10, max: 30 }
    const actual = validateMeasurement(undefined, { constraints })

    expect(actual).toEqual({ valid: false, message: 'Required' })
  })

  it('returns invalid when outside range', () => {
    const constraints = { min: 10, max: 30 }
    const actual = validateMeasurement(40, { constraints })

    expect(actual).toEqual({
      valid: false,
      message: 'Must be between 10 and 30'
    })
  })
})
```

```

    })
  })
})

```

Tests for the `validateMeasurement` validator.

Since the test is a bit more complex, I decided to assign the result to `actual`, and assert against that. I think this makes it more clear.

We don't need to use the specific constraints for pounds and kilograms outlined in the table earlier. As long as the tests pass with the constraints we pass in here, we can be confident `validateMeasurement` will work correctly for any given set of min/max constraints.

I also left a blank line between the body of the test and the assertion. This is a personal preference, loosely inspired by the three phases of a test: *arrange*, *act* and *assert*. We will talk about those later.

You don't have to write your tests like this. I find it useful to think in terms of "doing things" (eg, creating some variables, calling some functions) and asserting (where we say "given this scenario, this should happen").

Personal philosophy aside - the implementation, again, is much shorter than the test code. Notice a pattern? It's common for the test code to be longer than the implementation. It might feel a little strange at first, but it's not a problem and expected for complex logic.

```

export function validateMeasurement(value, { constraints }) {
  const result = required(value)
  if (!result.valid) {
    return result
  }

  return isBetween(value, constraints)
}

```

Composing `validateMeasurement` with `required` and `isBetween`.

Nice! We were able to reuse `required` and `isBetween`. We "composed" a validator using two small ones. Re-usability is good. Composability is good.

## 5.6 The Form Object and Full Form Validation

We have completed all the validations for each field. Let's think about the structure of the form now.

We have two fields: `name` and `weight`.

1. `name` is a string.
2. `weight` is a number with associated units.

These are the *inputs*. It should have this shape:

```
// definition
interface PatientFormState {
  name: string
  weight: {
    value: number
    units: 'kg' | 'lb'
  }
}

// usage
const patientForm: PatientFormState = {
  name: 'John',
  weight: {
    value: 445,
    units: 'lb'
  }
}
```

Object describing the patient.

Given an input (a `patientForm`), we can valid each field. Fields when validated are either `{ valid: true }` or `{ valid: false, message: '...' }`. So the form and validity interfaces could look like this:

```
interface ValidationResult {
  valid: boolean
  message?: string
}

interface PatientFormValidity {
  name: ValidationResult
  weight: ValidationResult
}

const patientForm: PatientFormState = {
  name: 'John',
  weight: {
```



```

    value: 445,
    units: 'lb'
  }
}

const validState = validateForm(patientForm)
// Return value should be:
// {
//   name: { valid: true }
//   weight: {
//     valid: false,
//     message: 'Must be between 66 and 440'
//   }
// }

```

Example usage of the `validateForm` function we will be writing.

We will need two functions:

1. `isFormValid`, to tell us if the form is valid or not.
2. `patientForm`, which handles figuring out the correct weight units, and calling all the validators.

Let's start with the tests for `isFormValid`. The form is considered valid when all fields are valid, so we only need two tests: the case where all fields are valid, and the case where at least one field is not:

```

import {
  required,
  isBetween,
  validateMeasurement,
  isFormValid
} from './form.js'

describe('required' () => {
  // ...
})

describe('isBetween', () => {
  // ...
})

describe('validateMeasurement', () => {
  // ...
})

describe('isFormValid', () => {

```

```

it('returns true when name and weight field are valid', () => {
  const form = {
    name: { valid: true },
    weight: { valid: true }
  }

  expect(isFormValid(form)).toBe(true)
})

it('returns false when any field is invalid', () => {
  const form = {
    name: { valid: false },
    weight: { valid: true }
  }

  expect(isFormValid(form)).toBe(false)
})
})

```

Testing isFormValid.

The implementation is simple:

```

export function isFormValid(form) {
  return form.name.valid && form.weight.valid
}

```

isFormValid implementation.

You could get fancy and iterate over the `form` using `Object.keys` or `Object.entries` if you were building a more generic form validation library. This would be a more general solution. In this case, I am keeping it as simple as possible.

The last test we need to complete the business logic is `patientForm`. This function takes an object with the `PatientFormState` interface we defined earlier. It returns the validation result of each field.

We will want to have quite a few tests here, to make sure we don't miss anything. The cases I can think of are:

1. Happy path: all inputs are valid
2. Patient name is null
3. Patient weight is outside constraints (imperial)
4. Patient weight is outside constraints (metric)

```

import {
  required,
  isBetween,
  validateMeasurement,
  isFormValid,
  patientForm
} from './form.js'

describe('required' () => {
  // ...
})

describe('isBetween', () => {
  // ...
})

describe('validateMeasurement', () => {
  // ...
})

describe('isFormValid', () => {
  // ...
})

describe('patientForm', () => {
  const validPatient = {
    name: 'test patient',
    weight: { value: 100, units: 'kg' }
  }

  it('is valid when form is filled out correctly', () => {
    const form = patientForm(validPatient)
    expect(form.name).toEqual({ valid: true })
    expect(form.weight).toEqual({ valid: true })
  })

  it('is invalid when name is null', () => {
    const form = patientForm({ ...validPatient, name: '' })
    expect(form.name).toEqual({ valid: false, message: 'Required' })
  })

  it('validates weight in imperial', () => {
    const form = patientForm({
      ...validPatient,
      weight: {

```

```

        value: 65,
        units: 'lb'
      }
    })

    expect(form.weight).toEqual({
      valid: false,
      message: 'Must be between 66 and 440'
    })
  })

  it('validates weight in metric', () => {
    const form = patientForm({
      ...validPatient,
      weight: {
        value: 29,
        units: 'kg'
      }
    })

    expect(form.weight).toEqual({
      valid: false,
      message: 'Must be between 30 and 200'
    })
  })
})

```

Testing patientForm.

The test code is quite long! The implementation is trivial, however. In this example, I am just hard-coding the weight constraints in an object called `limits`. In a real-world system, you would likely get these from an API and pass them down to the `patientForm` function.

```

const limits = {
  kg: { min: 30, max: 200 },
  lb: { min: 66, max: 440 },
}

export function patientForm(patient) {
  const name = required(patient.name)

  const weight = validateMeasurement(patient.weight.value, {
    nullable: false,
    constraints: limits[patient.weight.units]
  })
}

```

```
    return {  
      name,  
      weight  
    }  
  }  
}
```

Implementing patientForm.

This completes the business logic for the patient form - noticed we haven't written and Vue components yet? That's because we are adhering to one of our goals; *separation of concerns*, and isolating the business logic entirely.

## 5.7 Writing the UI Layer

Now the fun part - writing the UI layer with Vue. Although I think TDD is a great fit for business logic, I generally do not use TDD for my component tests.

I like to start by thinking about how I will manage the state of my component. Let's use the Composition API; I think works great for forms.

```
<script>  
import { reactive, computed, ref } from 'vue'  
import { patientForm, isValidForm } from './form.js'  
  
export default {  
  setup() {  
    const form = reactive({  
      name: '',  
      weight: {  
        value: '',  
        units: 'kg'  
      }  
    })  
  
    const validatedForm = computed(() => patientForm(form))  
    const valid = computed(() => isValidForm(validatedForm.value))  
  
    return {  
      form,  
      validatedForm,  
      valid  
    }  
  }  
}
```

Integrating the form business logic and the Vue UI layer.

I decided to keep the state in a **reactive** object. Both the **valid** state and **validateForm** are **computed** values - we want the validation and form state to update reactively when any value in the form changes.

Let's add the **<template>** part now - it's very simple, just good old HTML.

```

<template>
  <h3>Patient Data</h3>
  <form>
    <div class="field">
      <div v-if="!validatedForm.name.valid" class="error">
        {{ validatedForm.name.message }}
      </div>
      <label for="name">Name</label>
      <input id="name" name="name" v-model="form.name" />
    </div>
    <div class="field">
      <div v-if="!validatedForm.weight.valid" class="error">
        {{ validatedForm.weight.message }}
      </div>
      <label for="weight">Weight</label>
      <input
        id="weight"
        name="weight"
        v-model.number="form.weight.value"
      />
      <select name="weightUnits" v-model="form.weight.units">
        <option value="kg">kg</option>
        <option value="lb">lb</option>
      </select>
    </div>
    <div class="field">
      <button :disabled="!valid">Submit</button>
    </div>
  </form>
  <pre>
  Patient Data
  {{ form }}
  </pre>
  <br />
  <pre>
  Form State
  {{ validatedForm }}
  </pre>
</template>

```

A simple template with form v-model bindings.

I added the `<pre>` block for some debugging. Everything works!

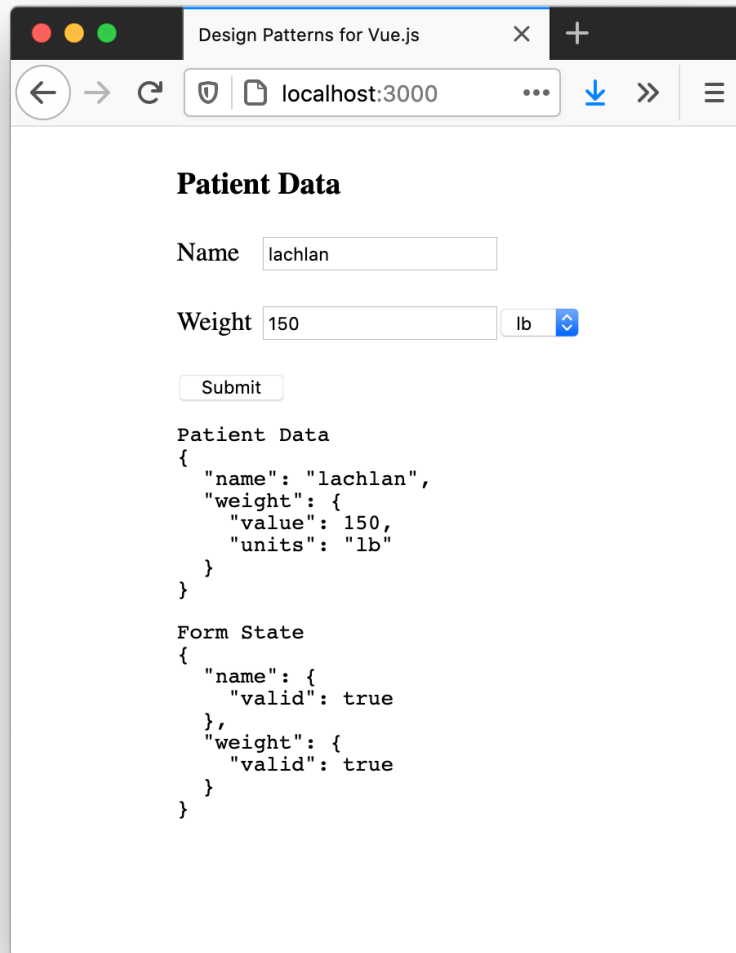


Figure 4: Validation debug info



## 5.8 Some Basic UI Tests

We can add some basic UI tests using Testing Library, too. Here are two fairly simple ones that cover most of the functionality:

```
import { render, screen, fireEvent } from '@testing-library/vue'
import FormValidation from './form-validation.vue'

describe('FormValidation', () => {
  it('fills out form correctly', async () => {
    render(FormValidation)

    await fireEvent.update(screen.getByLabelText('Name'), 'lachlan')
    await fireEvent.update(screen.getByDisplayValue('kg'), '1b')
    await fireEvent.update(screen.getByLabelText('Weight'), '150')

    expect(screen.queryByRole('error')).toBe(null)
  })

  it('shows errors for invalid inputs', async () => {
    render(FormValidation)

    await fireEvent.update(screen.getByLabelText('Name'), '')
    await fireEvent.update(screen.getByLabelText('Weight'), '5')
    await fireEvent.update(screen.getByDisplayValue('kg'), '1b')

    expect(screen.getAllByRole('error')).toHaveLength(2)
  })
})
```

Testing the UI layer with Testing Library.

Since these tests are a little larger, I am making the separation between each step clear. I like to write my tests like this:

```
it('...', async () => {
  // Arrange - this is where we set everything up
  render(FormValidation)

  // Act - do things!
  // Call functions
  // Assign values
  // Simulate interactions
  await fireEvent.update(screen.getByLabelText('Name'), 'lachlan')

  // Assert
  expect(...).toEqual(...)
```

```
} )
```

Anatomy of a test - arrange, act, assert.

We don't have any tests to ensure the `<button>` is correctly disabled - see below for more.

## 5.9 Improvements and Conclusion

The goal here wasn't to build the *perfect* form but illustrate how to separate your form validation and business logic from the UI layer.

As it stands, you can enter any string into the weight field and it will be considered valid - not ideal, but also trivial to fix. A good exercise would be to write some tests to ensure the input is a number, and if not, return a useful error message. We also haven't got any tests to ensure the `<button>` is correctly disabled.

## 5.10 Exercises

- Add a test to ensure that any non-numeric values entered into the `weight` field cause the field to become invalid and show a "Weight must be a number" error.
- Add a `@submit.prevent` listener to `<form>`. When the form is submitted, emit an event with the `patientForm`.
- Submit the form using Testing Library and assert the correct event and payload are emitted.

You can find the completed source code (including exercises) in the GitHub repository under `examples/form-validation`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

## 6 HTTP and API Requests

Something almost every Vue.js application is going to do is make HTTP requests to an API of some sort. This could be for authentication, loading data, or something else. Many patterns have emerged to manage HTTP requests, and even more to test them.

This chapter looks at various ways to architecture your HTTP requests, different ways to test them, and discusses the pros and cons of each approach.

### 6.1 The Login Component

The example I will use is the `<login>` component. It lets the user enter their username and password and attempt to authenticate. We want to think about:

- where should the HTTP request be made from? The component, another module, in a store (like Vuex?)
- how can we test each of these approaches?

There is no one size fits all solution here. I'll share how I currently like to structure things, but also provide my opinion on other architectures.

### 6.2 Starting Simple

If your application is simple, you probably won't need something like Vuex or an isolated HTTP request service. You can just inline everything in your component:

```
<template>
  <h1 v-if="user">
    Hello, {{ user.name }}
  </h1>
  <form @submit.prevent="handleAuth">
    <input v-model="formData.username" role="username" />
    <input v-model="formData.password" role="password" />
    <button>Click here to sign in</button>
  </form>
  <span v-if="error">{{ error }}</span>
</template>

<script>
import axios from 'axios'

export default {
  data() {
    return {
```

```

    username: '',
    password: '',
    user: undefined,
    error: ''
  }
},

methods: {
  async handleAuth() {
    try {
      const response = await axios.post('/login')
      this.user = response.data
    } catch (e) {
      this.error = e.response.data.error
    }
  }
}
}
</script>

```

A simple login form component, It makes a request using axios.

This example uses the axios HTTP library, but the same ideas apply if you are using fetch.

We don't want to make a request to a real server when testing this component - unit tests should run in isolation. One option here is to mock the `axios` module with `jest.mock`.

We probably want to test:

- is the correct endpoint used?
- is the correct payload included?
- does the DOM update accordingly based on the response?

A test where the user successfully authenticates might look like this:

```

import { render, fireEvent, screen } from '@testing-library/vue'
import App from './app.vue'

let mockPost = jest.fn()
jest.mock('axios', () => ({
  post: (url, data) => {
    mockPost(url, data)
    return Promise.resolve({
      data: { name: 'Lachlan' }
    })
  }
}))

```

```

}))

describe('login', () => {
  it('successfully authenticates', async () => {
    render(App)
    await fireEvent.update(
      screen.getByRole('username'), 'Lachlan')
    await fireEvent.update(
      screen.getByRole('password'), 'secret-password')
    await fireEvent.click(screen.getByText('Click here to sign in'))

    expect(mockPost).toHaveBeenCalledWith('/login', {
      username: 'Lachlan',
      password: 'secret-password'
    })
    await screen.findByText('Hello, Lachlan')
  })
})

```

Using a mock implementation of axios to test the login workflow.

Testing a failed request is straight forward as well - you would just throw an error in the mock implementation.

### 6.3 Refactoring to a store

If you are working on anything other than a trivial application, you probably don't want to store the response in component local state. The most common way to scale a Vue app has traditionally been Vuex. More often than not, you end up with a Vuex store that looks like this:

```

import axios from 'axios'

export const store = {
  state() {
    return {
      user: undefined
    }
  },
  mutations: {
    updateUser(state, user) {
      state.user = user
    }
  },
  actions: {
    login: async ({ commit }, { username, password }) => {

```

```

    const response = await axios.post('/login', {
      username,
      password
    })
    commit('updateUser', response.data)
  }
}
}

```

A simple Vuex store.

There are many strategies for error handling in this set up. You can have a local `try/catch` in the component. Other developers store the error in the Vuex state, as well.

Either way, the `<login>` component using a Vuex store would look something like this:

```

<template>
  <!-- no change -->
</template>

<script>
import axios from 'axios'

export default {
  data() {
    return {
      username: '',
      password: '',
      error: ''
    }
  },
  computed: {
    user() {
      return this.$store.state.user
    }
  },
  methods: {
    async handleAuth() {
      try {
        await this.$store.dispatch('login', {
          username: this.username,
          password: this.password
        })
      } catch (e) {
        this.error = e.response.data.error
      }
    }
  }
}

```

```

    }
  }
}
</script>

```

Using Vuex in the login component.

You now need a Vuex store in your test, too. You have a few options. The two most common are:

- use a real Vuex store - continue mocking axios
- use a mock Vuex store

The first option would look something like this:

```

import { store } from './store.js'

describe('login', () => {
  it('successfully authenticates', async () => {
    // add
    render(App, { store })
  })
})

```

Updating the test to use Vuex.

I like this option. We continue to mock `axios`. The only change we made to the test is passing a `store`. The actual user facing behavior has not changed, so the test should not need significant changes either - in fact, the actual test code is the same (entering the username and password and submitting the form). It also shows we are not testing implementation details - we were able to make a significant refactor without changing the test (except for providing the Vuex store - we added this dependency, so this change is expected).

To further illustrate this is a good test, I am going to make another refactor and convert the component to use the Composition API. Everything *should* still pass:

```

<template>
  <!-- no changes -->
</template>

<script>
import { reactive, ref, computed } from 'vue'
import { useStore } from 'vuex'

export default {
  setup () {

```

```

const store = useStore()
const formData = reactive({
  username: '',
  password: '',
})
const error = ref('')
const user = computed(() => store.state.user)

const handleAuth = async () => {
  try {
    await store.dispatch('login', {
      username: formData.username,
      password: formData.password
    })
  } catch (e) {
    error.value = e.response.data.error
  }
}

return {
  user,
  formData,
  error,
  handleAuth
}
}
</script>

```

Converting the component to use the Component API.

Everything still passes - another indication we are testing the behavior of the component, as opposed to the implementation details.

I've used the real store + axios mock strategy for quite a long time in both Vue and React apps and had a good experience. The only downside is you need to mock `axios` a lot - you often end up with a lot of copy-pasting between tests. You can make some utilities methods to avoid this, but it's still a little boilerplate heavy.

## 6.4 To mock or not to mock?

As your application gets larger and larger, though, using a real store can become complex. Some developers opt to mock the entire store in this scenario. It leads to less boilerplate, for sure, especially if you are using Vue Test Utils, which



has a `mocks` mounting option designed for mocking values on `this`, for example `this.$store`.

Testing Library does not support mocking things so easily - intentionally. They want your tests to be as production-like as possible, which means using real dependencies whenever possible. I like this philosophy. To see why I prefer to use a real Vuex store in my tests, let's see what happens if we mock Vuex using `jest.mock`.

```
let mockDispatch = jest.fn()
jest.mock('vuex', () => ({
  useStore: () => ({
    dispatch: mockDispatch,
    state: {
      user: { name: 'Lachlan' }
    }
  })
}))

describe('login', () => {
  it('successfully authenticates', async () => {
    render(App)
    await fireEvent.update(
      screen.getByRole('username'), 'Lachlan')
    await fireEvent.update(
      screen.getByRole('password'), 'secret-password')
    await fireEvent.click(screen.getByText('Click here to sign in'))

    expect(mockDispatch).toHaveBeenCalledWith('login', {
      username: 'Lachlan',
      password: 'secret-password'
    })
    await screen.findByText('Hello, Lachlan')
  })
})
```

#### Mocking Vuex.

Since we are mocking the Vuex store now, we have bypassed `axios` entirely. This style of test is tempting at first. There is less code to write. It's very easy to write. You can also directly set the `state` however you like - in the snippet above, `dispatch` doesn't actually update the state.

Again, the actual test code didn't change much - we are no longer passing a `store` to `render` (since we are not even using a real store in the test, we mocked it out entirely). We don't have `mockPost` any more - instead we have `mockDispatch`. The assertion against `mockDispatch` became an assertion that a `login` action was dispatched with the correct payload, as opposed to a HTTP

call to the correct endpoint.

There is a big problem. Even if you delete the `login` action from the store, the test will *continue to pass*. This is scary! The tests are all green, which should give you confidence everything is working correctly. In reality, your entire application is completely broken.

This is not the case with the test using a real Vuex store - breaking the store correctly breaks the tests. There is only one thing worse than a code-base with no tests - a code-base with *bad* tests. At least if you have not tests, you have no confidence, which generally means you spend more time testing by hand. Tests that give false confidence are actually worse - they lure you into a false sense of security. Everything seems okay, when really it is not.

## 6.5 Mock Less - mock the lowest dependency in the chain

The problem with the above example is we are mocking too far up the chain. Good tests are as production like as possible. This is the best way to have confidence in your test suite. This diagram shows the dependency chain in the `<login>` component:

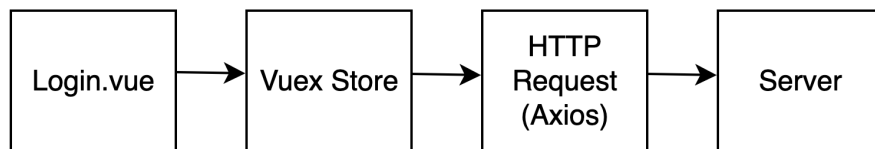


Figure 5: Authentication dependency chain

The previous test, where we mocked Vuex, mocks the dependency chain here:

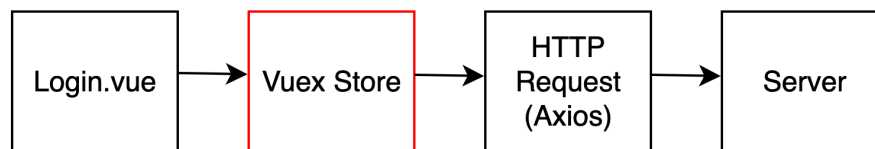


Figure 6: Mocking Vuex

This means if anything breaks in Vuex, the HTTP call, or the server, our test will not fail.

The axios test is slightly better - it mocks one layer lower:

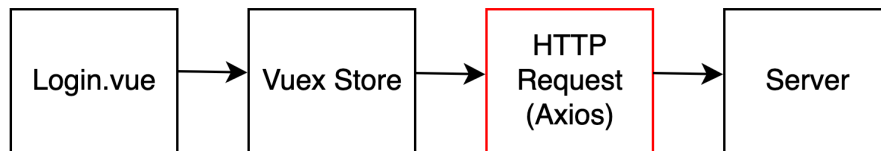


Figure 7: Mocking Axios

This is better. If something breaks in either the `<login>` or Vuex, the test will fail.

Wouldn't it be great to avoid mocking `axios`, too? This way, we could not need to do:

```
let mockPost = jest.fn()
jest.mock('axios', () => ({
  post: (url, data) => {
    mockPost(url, data)
    return Promise.resolve({
      data: { name: 'Lachlan' }
    })
  }
}))
```

Boilerplate code to mock axios.

... in every test. And we'd have more confidence, further down the dependency chain.

## 6.6 Mock Service Worker

A new library has come into the scene relatively recently - Mock Service Worker, or `msw` for short. This does exactly what is discussed above - it operates one level lower than `axios`, mocking the actual network request! How `msw` works will not be explained here, but you can learn more on the website: <https://mswjs.io/>. One of the cool features is that you can use it both for tests in a Node.js environment and in a browser for local development.

Let's try it out. Basic usage is like this:

```
import { rest } from 'msw'
import { setupServer } from 'msw/node'

const server = setupServer(
  rest.post('/login', (req, res, ctx) => {
    return res(
      ctx.json({
        name: 'Lachlan'
      })
    )
  })
)

```

A basic server with msw.

The nice thing is we are not mocking `axios` anymore. You could change your application to use `fetch` instead - and you wouldn't need to change your tests at all, because we are now mocking at a layer lower than before.

A full test using msw looks like this:

```
import { render, fireEvent, screen } from '@testing-library/vue'
import { rest } from 'msw'
import { setupServer } from 'msw/node'
import App from './app.vue'
import { store } from './store.js'

const server = setupServer(
  rest.post('/login', (req, res, ctx) => {
    return res(
      ctx.json({
        name: 'Lachlan'
      })
    )
  })
)

describe('login', () => {
  beforeAll(() => server.listen())
  afterAll(() => server.close())

  it('successfully authenticates', async () => {
    render(App, { store })
    await fireEvent.update(
      screen.getByRole('username'), 'Lachlan'
    )
    await fireEvent.update(
      screen.getByRole('password'), 'secret-password'
    )
  })
})

```

```

    await fireEvent.click(screen.getByText('Click here to sign in'))

    await screen.findByText('Hello, Lachlan')
  })
})

```

Using msw instead of mocking axios.

You can have even less boilerplate by setting up the server in another file and importing it automatically, as suggested in the documentation: <https://mswjs.io/docs/getting-started/integrate/node>. Then you won't need to copy this code into all your tests - you just test as if you are in production with a real server that responds how you expect it to.

One thing we are not doing in this test that we were doing previously is asserting the expected payload is sent to the server. If you want to do that, you can just keep track of the posted data with an array, for example:

```

const postedData = []
const server = setupServer(
  rest.post('/login', (req, res, ctx) => {
    postedData.push(req.body)
    return res(
      ctx.json({
        name: 'Lachlan'
      })
    )
  })
)

```

Keeping track of posted data.

Now you can just assert that `postedData[0]` contains the expected payload. You could reset it in the `beforeEach` hook, if testing the body of the post request is something that is valuable to you.

`msw` can do a lot of other things, like respond with specific HTTP codes, so you can easily simulated a failed request, too. This is where `msw` really shines compared to the using `jest.mock` to mock `axios`. Let's add another test for this exact case:

```

describe('login', () => {
  beforeAll(() => server.listen())
  afterAll(() => server.close())

  it('successfully authenticates', async () => {
    // ...
  })
})

```

```

it('handles incorrect credentials', async () => {
  const error = 'Error: please check the details and try again'
  server.use(
    rest.post('/login', (req, res, ctx) => {
      return res(
        ctx.status(403),
        ctx.json({ error })
      )
    })
  )

  render(App, { store })
  await fireEvent.update(
    screen.getByRole('username'), 'Lachlan')
  await fireEvent.update(
    screen.getByRole('password'), 'secret-password')
  await fireEvent.click(screen.getByText('Click here to sign in'))

  await screen.findByText(error)
})
})

```

A test for a failed request.

It's easy to extend the mock server on a test by test basis. Writing these two tests using `jest.mock` to mock `axios` would be very messy!

Another very cool feature about `msw` is you can use it in a browser during development. It isn't showcased here, but a good exercise would be to try it out and experiment. Can you use the same endpoint handlers for both tests and development?

## 6.7 Conclusion

This chapter introduces various strategies for testing HTTP requests in your components. We saw the advantage of mocking `axios` and using a real Vuex store, as opposed to mocking the Vuex store. We then moved one layer lower, mocking the actual server with `msw`. This can be generalized - the lower the mock in the dependency chain, the more confidence you can be in your test suite.

Tests `msw` is not enough - you still need to test your application against a real server to verify everything is working as expected. Tests like the ones described in this chapter are still very useful - they run fast and are very easy to write. I tend to use `testing-library` and `msw` as a development tool - it's definitely faster than opening a browser and refreshing the page every time you make a change to your code.

## 6.8 Exercises

- Trying using `msw` in a browser. You can use the same mock endpoint handlers for both your tests and development.
- Explore `msw` more and see what other interesting features it offers.

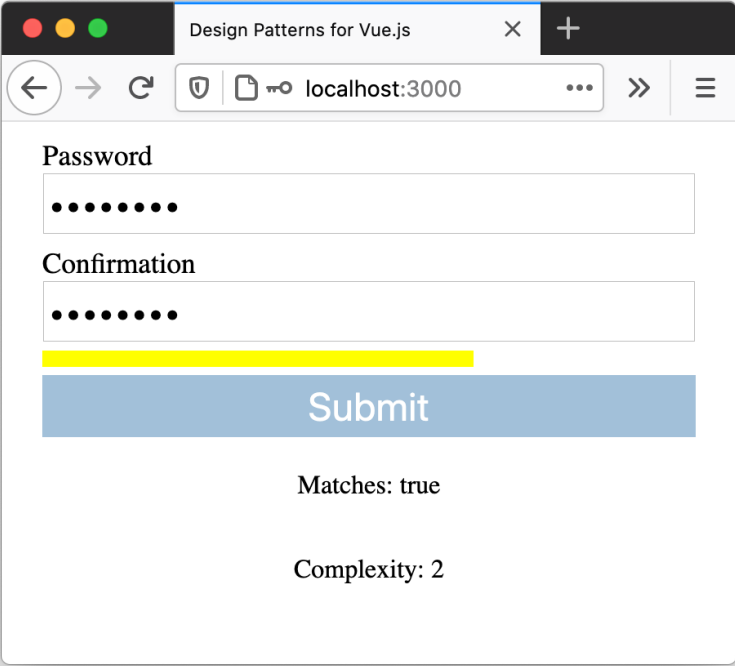
## 7 Renderless Components

You can find the completed source code in the GitHub repository under `examples/renderless-password`:  
<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

The primary way you reuse components in Vue is *slots*. This works great for a lot of cases, but sometimes you need *more* flexibility.

One example is you have some complex logic that needs to be reused in two different interfaces. One way to reuse complex logic with several different interfaces is the *renderless* component pattern.

In this section we will build the following component, a password strength form:



The screenshot shows a web browser window with the title "Design Patterns for Vue.js". The address bar displays "localhost:3000". The form contains two input fields labeled "Password" and "Confirmation", both filled with dots. Below these fields is a yellow progress bar that is approximately 50% full. A blue "Submit" button is positioned below the progress bar. At the bottom of the form, the text "Matches: true" and "Complexity: 2" are displayed.

Figure 8: Completed Password Complexity Component



There are a few requirements. We'd like to publish this on npm; to make it as flexible as possible, we will use a technique known as a "renderless" component. This means we will not ship any specific markup. Instead, the developer will need to provide their own.

This means we will work with a **render** function, the low-level JavaScript that `<template>` is compiled to. This will allow developers to fully customize the markup and style as they see fit.

We would like to support the following features:

- A **matching** variable that returns true if the password and confirmation match.
- Support a **minComplexity** prop; by default, the minimum complexity is 0 and the maximum complexity is 3. This is represented by the yellow bar above the submit button in the screenshot above.
- support a custom complexity algorithm (eg, require specific characters or numbers in the password).
- Expose a **valid** value which is true when the password and confirmation match and the password meets the minimum complexity.

Let's get started.

## 7.1 Rendering without Markup

I will work out of a file called **renderless-password.js**. That's right - not a **vue** file. No need - we won't be shipping a `<template>`.

```
export default {
  setup(props, { slots }) {
    return () => slots.default({
      complexity: 5
    })
  }
}
```

Renderless functions return `slots.default()` in `setup` or `render`.

This is how renderless components work; calling **slots** with an object will expose whatever properties are passed to the object via the **v-slot** directive.

Let's see this in action but using the component in a regular **vue** file. Mine is called **app.vue**; find the completed version in the source code.

```
<template>
  <renderless-password
    v-slot="{
      complexity
    }">
```

```

    >
      {{ complexity }}
    </renderless-password>
  </template>

<script>
import RenderlessPassword from './renderless-password.js'

export default {
  components: {
    RenderlessPassword
  }
}
</script>

```

Trying out the renderless-password.

We can destructure the object passed to `slots.default()` in `v-slot`, and are free to use them however we like in the `<template>`. Great! This currently just renders a 5; not very interesting, but it illustrates the idea of exposing properties via `v-slot`.

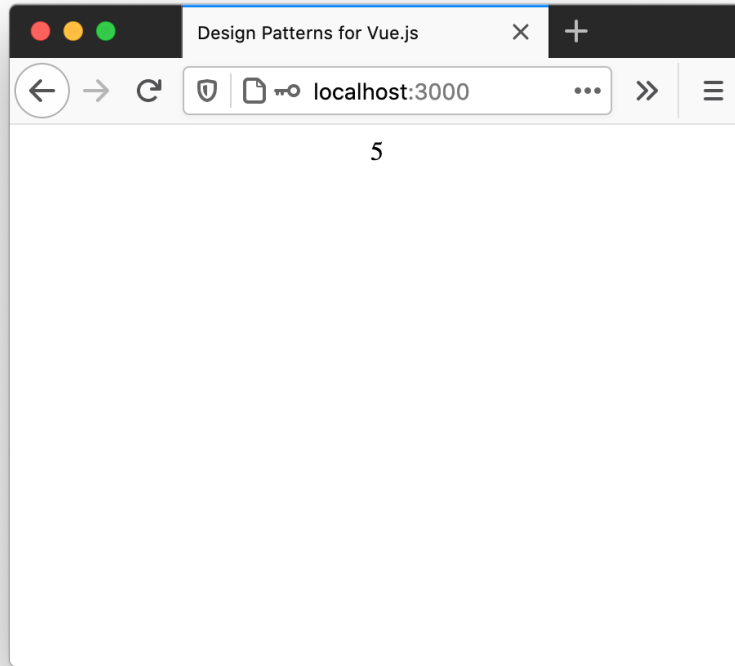


Figure 9: Rendering with `slots.default()` and `v-slot`

## 7.2 Adding Password and Confirmation Inputs

The next feature we will add is the password and confirmation fields. We will also expose a `matching` property, to see if the password and confirmation are the same.

First, update `renderless-password.js` to receive a `password` and `confirmation` prop. We also add the logic to see if the passwords match:

```
import { computed } from 'vue'

export function isMatching(password, confirmation) {
  if (!password || !confirmation) {
    return false
  }
}
```

```

    return password === confirmation
  }

  export default {
    props: {
      password: {
        type: String
      },
      confirmation: {
        type: String
      }
    },

    setup(props, { slots }) {
      const matching = computed(() => isMatching(
        props.password, props.confirmation))

      return () => slots.default({
        matching: matching.value
      })
    }
  }
}

```

Checking if password and confirmation match.

You may notice I implemented `isMatching` as a separate function, which I've exported. I consider this part of the *business logic*, not the UI, so I like to keep it separate. This makes it super easy to test, and also keeps my `setup` function nice and simple. You could declare it inside of `setup` if you prefer that style.

I also removed `complexity: 5` from `slots.default()`; we will come back to this, but we aren't using it right now.

One thing that might be a little surprising if you need to pass `matching.value` to `slots.default()`. This is because I would like to let the developer destructure `matching` by doing `v-slot="{ matching }"` as opposed to `v-slot="{ matching: matching.value }"`; the former feels cleaner to me.

Let's try it out:

```

<template>
  <renderless-password
    :password="input.password"
    :confirmation="input.confirmation"
    v-slot="{
      matching
    }"
  >

```

```

    <div class="wrapper">
      <div class="field">
        <label for="password">Password</label>
        <input v-model="input.password" id="password" />
      </div>
      <div class="field">
        <label for="confirmation">Confirmation</label>
        <input v-model="input.confirmation" id="confirmation" />
      </div>
    </div>

    <p>Matches: {{ matching }}</p>

  </renderless-password>
</template>

<script>
import { reactive } from 'vue'
import RenderlessPassword from './renderless-password.js'

export default {
  components: {
    RenderlessPassword
  },

  setup(props) {
    const input = reactive({
      password: '',
      confirmation: ''
    })

    return {
      input
    }
  }
}
</script>

```

password and confirmation are saved in a reactive object.

The main change is we now have a **reactive** input that has **password** and **confirmation** properties. You could have used two **refs**; one for **password** and one for **confirmation**. I like to group related properties using **reactive**, so that's why I am using **reactive** here.

I also added some extra **<div>** elements and classes - those are mainly for styling.

You can grab the final styles from the source code. It looks like this:

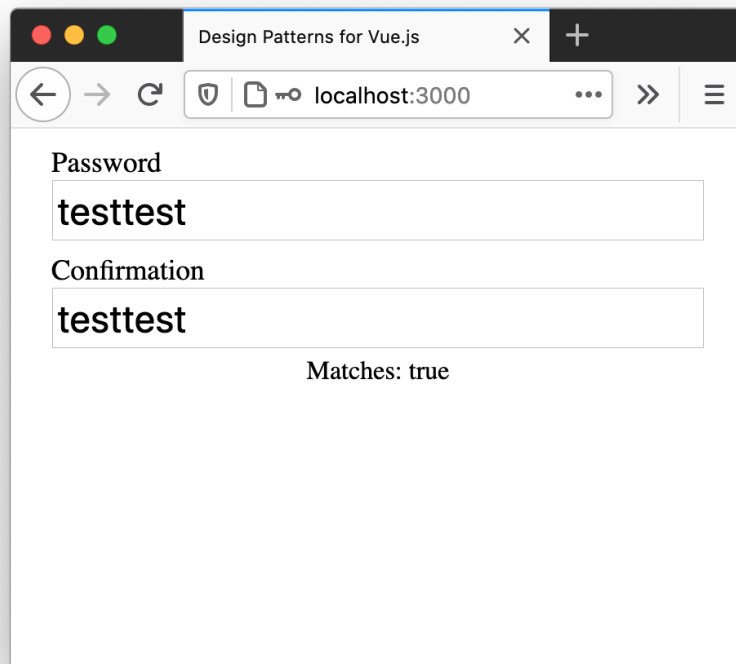


Figure 10: Rendering Inputs and Debug Info

This works great! The complexity and business logic is nicely abstracted away in `renderless-password`. The developer can use the logic to style the component to suit their application and use case.

Let's keep going and add a customizable `complexity` feature, to rate whether a password is sufficiently complex.

### 7.3 Adding Password Complexity

For now, we will implement a very naive complexity check. Most developers will want to customize this. For this example, we will keep it simple and choose an algorithm that will rate complexity based on the length of the password:

- high: length  $\geq 10$
- mid: length  $\geq 7$
- low: length  $\geq 5$

As with `isMatching`, we will make a `calcComplexity` a pure function. Decoupled, deterministic, and easily testable.

```
import { computed } from 'vue'

export function isMatching() {
  // ...
}

export function calcComplexity(val) {
  if (!val) {
    return 0
  }

  if (val.length >= 10) {
    return 3
  }
  if (val.length >= 7) {
    return 2
  }
  if (val.length >= 5) {
    return 1
  }

  return 0
}

export default {
  props: {
    // ...
  },

  setup(props, { slots }) {
    const matching = computed(() => isMatching(
      props.password, props.confirmation))
    const complexity = computed(() => calcComplexity(
      props.password))

    return () => slots.default({
      matching: matching.value,
      complexity: complexity.value
    })
  }
}
```

```

  }
}

```

Adding a simple calcComplexity function.

Everything is very similar to what we did with the `isMatching` function and `matching` computed property. We will add support for a custom complexity function in the future passed via a prop.

Let's try it out:

```

<template>
  <renderless-password
    :password="input.password"
    :confirmation="input.confirmation"
    v-slot="{
      matching,
      complexity
    }"
  >
    <div class="wrapper">
      <div class="field">
        <label for="password">Password</label>
        <input v-model="input.password" id="password" />
      </div>
      <div class="field">
        <label for="confirmation">Confirmation</label>
        <input v-model="input.confirmation" id="confirmation" />
      </div>
      <div class="complexity-field">
        <div
          class="complexity"
          :class="complexityStyle(complexity)"
        />
      </div>
    </div>

    <p>Matches: {{ matching }}</p>
    <p>Complexity: {{ complexity }}</p>

  </renderless-password>
</template>

<script>
import { reactive } from 'vue'
import RenderlessPassword from './renderless-password.js'

```



```

export default {
  components: {
    RenderlessPassword
  },

  setup(props) {
    const input = reactive({
      password: '',
      confirmation: ''
    })

    const complexityStyle = (complexity) => {
      if (complexity >= 3) {
        return 'high'
      }
      if (complexity >= 2) {
        return 'mid'
      }
      if (complexity >= 1) {
        return 'low'
      }
    }

    return {
      input,
      complexityStyle
    }
  }
}
</script>

<style>
/**
  some styles excluded for brevity
  see source code for full styling
*/
.complexity {
  transition: 0.2s;
  height: 10px;
}

.high {
  width: 100%;
  background: lime;
}

```

```

.mid {
  width: 66%;
  background: yellow;
}

.low {
  width: 33%;
  background: red;
}
</style>

```

Styling the form based using a computed style.

I also added a `complexityStyle` function to apply a different CSS class depending on the complexity. I have consciously chosen *not* to define and export this function outside of `setup` - instead, I defined it *inside* of `setup`.

The reason for this is I see no value in testing `complexityStyle` separately to the component - knowing that the correct class (`high`, `mid`, or `low`) is returned is not enough. To fully test this component, I'll need to assert against the DOM.

You could still export `complexityStyle` and test it individually, but you still need to test that the correct class is applied (eg, you could forget to code `:class="complexityStyle(complexity)"`, for example, and the `complexityStyle` test would still pass).

By writing a test and asserting against the DOM, you test `complexityStyle` implicitly. The test would look something like this (see the source code for the full working example):

```

it('meets default requirements', async () => {
  render(TestComponent)

  await fireEvent.update(
    screen.getByLabelText('Password'), 'this is a long password')
  await fireEvent.update(
    screen.getByLabelText('Confirmation'), 'this is a long password')

  expect(screen.getByRole('password-complexity').classList)
    .toContain('high')
  expect(screen.getByText('Submit').disabled).toBeFalsy()
})

```

Testing the correct complexity class is included.

The application now looks like this:

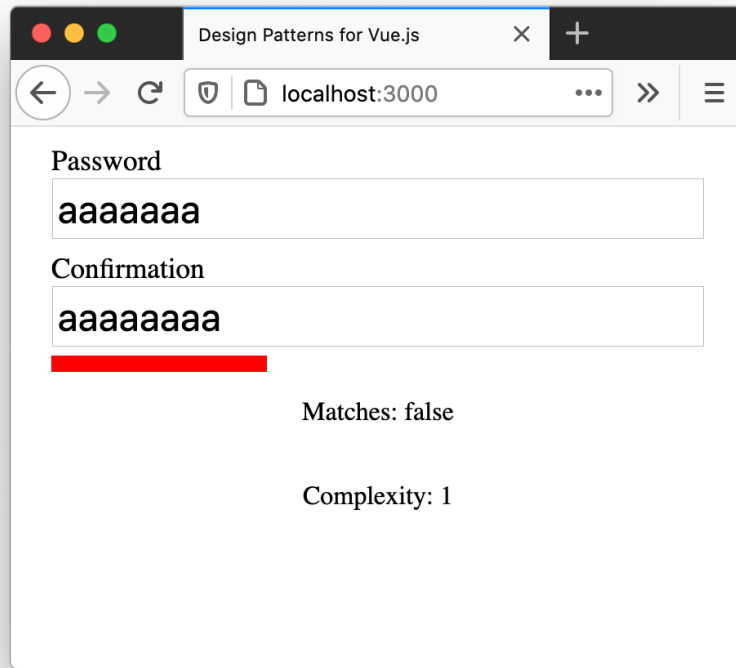


Figure 11: Complexity Indicator

## 7.4 Computing Form Validity

Let's add the final feature: a button that is only enabled when a `valid` property is `true`. The `valid` property is exposed by the `<renderless-password>` and accessed via `v-slot`.

```
import { computed } from 'vue'

export isMatching() {
  // ...
}

export calcComplexity() {
  // ...
}
```

```

}

export default {
  props: {
    minComplexity: {
      type: Number,
      default: 3
    },
    // ... other props ...
  },

  setup(props, { slots }) {
    const matching = computed(() => isMatching(
      props.password, props.confirmation))
    const complexity = computed(() => calcComplexity(
      props.password))
    const valid = computed(() =>
      complexity.value >= props.minComplexity &&
      matching.value)

    return () => slots.default({
      matching: matching.value,
      complexity: complexity.value,
      valid: valid.value
    })
  }
}

```

Validating the form with a valid computed property, derived from matching and complexity.

I added a **valid** computed property, based on the result of **complexity** and **matching**. You could make a separate function for this if you wanted to test it in isolation. If I was going to distribute this npm, I probably would; alternatively, we can test this implicitly by binding **valid** to a button's **disabled** attribute, like we are about to do, and then assert against the DOM that the attribute is set correctly.

Update the usage to include a `<button>` that binds to **valid**:

```

<template>
  <renderless-password
    :password="input.password"
    :confirmation="input.confirmation"
    v-slot="{
      matching,
      complexity,

```

```

        valid
      }"
    >
    <div class="wrapper">
      <!-- ... omitted for brevity ... -->
      <div class="field">
        <button :disabled="!valid">Submit</button>
      </div>
    </div>

    <p>Matches: {{ matching }}</p>
    <p>Complexity: {{ complexity }}</p>

  </renderless-password>
</template>

```

Destructuring `valid` and binding to it.

Everything works! And we can easily move elements around to change the look and feel of `<renderless-password>`.

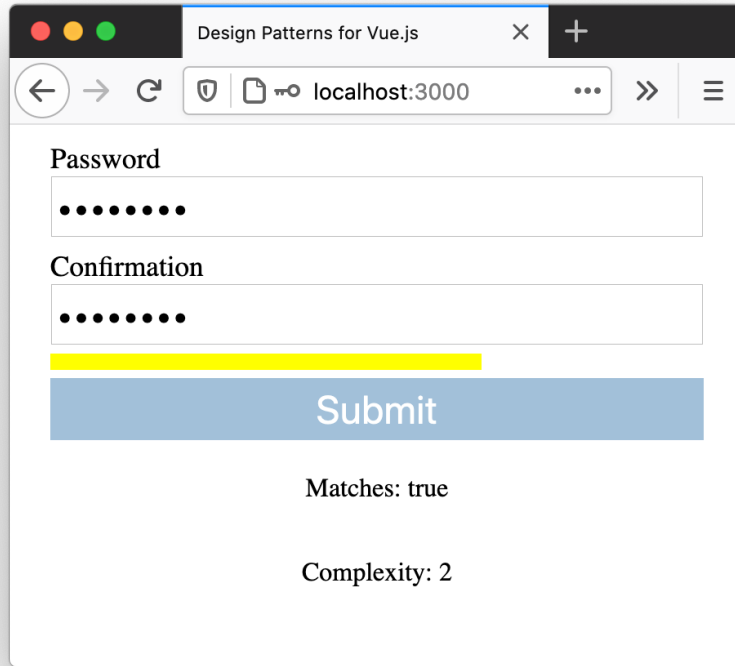


Figure 12: Completed Password Complexity Component

Just for fun, I tried making an alternative UI. All I had to do was move around some markup:

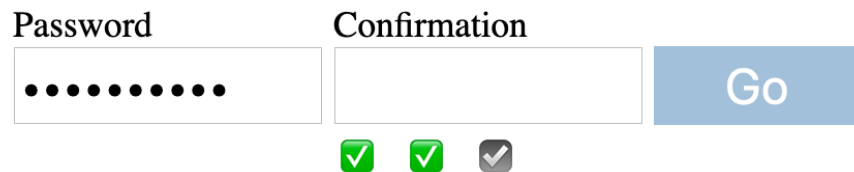


Figure 13: Alternative Password Complexity Component

See what else you can come up with. I think there is a lot of room for innovation with the renderless component pattern. There is at least one project using this pattern, Headless UI - check it out for more inspiration: <https://headlessui.dev/>.

## 7.5 Exercises

This section intentionally omitted writing tests to focus on the concepts. Several techniques regarding tests were mentioned. For practice, try to write the following tests (find the solutions in the source code):

- Some tests using Testing Library to assert the correct complexity class is assigned.
- Test that the button is appropriately disabled.

You could also write some tests for the business logic, to make sure we didn't miss any edge cases:

- Test the `calcComplexity` and `isMatching` functions in isolation.

There are also some improvements you could try making:

- Allow the developer to pass their own `calcComplexity` function as a prop. Use this if it's provided.
- Support passing a custom `isValid` function, that receives `password`, `confirmation`, `isMatching` and `complexity` as arguments.

You can find the completed source code in the GitHub repository under `examples/renderless-password`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

## 8 The Power of Render Functions

You can find the completed source code in the GitHub repository under `examples/renderless-password`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

---

So far, all the examples in this book have used a `<template>` to structure the components. In reality, Vue does a ton of heavy lifting in the background between writing markup in `<template>` and rendering content in a browser. This is primarily handled by one of Vue's core packages, `@vue/compiler-sfc`.

Code in `<template>` is compiled to something called *render functions*. Several things happen during this compilation step. Some of these are:

- Directives such as `v-if` and `v-for` are converted to regular JavaScript (`if` and `for` or `map`, for example).
- Optimizations.
- CSS is scoped (if you are using `<style scoped>`).

While it is generally more ergonomic to write your components with `<template>`, there are some situations where it can be beneficial to write the render functions yourself. One such situation is when writing a very generic UI library. It's also good to understand how things work under the hood.

In this section we will build a tab component. The usage will look something like this:

```
<template>
  <tab-container v-model:activeTabId="activeTabId">
    <tab tabId="1">Tab #1</tab>
    <tab tabId="2">Tab #2</tab>
    <tab tabId="3">Tab #3</tab>

    <tab-content tabId="1">Content #1</tab-content>
    <tab-content tabId="2">Content #2</tab-content>
    <tab-content tabId="3">Content #3</tab-content>
  </tab-container>
</template>
```

Final markup for the tabs component.



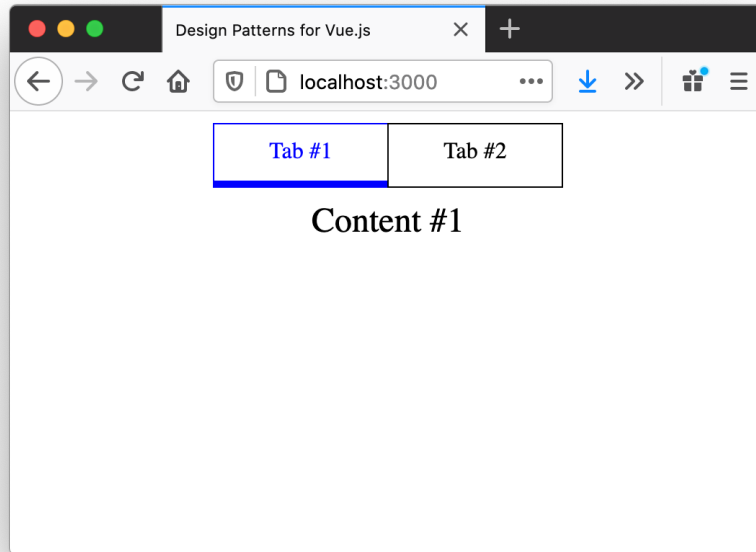


Figure 14: Completed Tabs Component

The `<tab-container>` component works by taking a `<tab>` component with a `tabId` prop. This is paired with a `<tab-content>` component with the same `tabId`. Only the `<tab-content>` where the `tabId` prop matches the `activeTabId` value will be shown. We will dynamically update `activeTabId` when a `<tab>` is clicked.

## 8.1 Why Render Functions?

This example shows a great use case for render functions. Without them, you might need to write something like this:

```
<template>
  <tab-container v-model:activeTabId="activeTabId">
    <tab @click="activeTabId = '1'">Tab #1</tab>
    <tab @click="activeTabId = '2'">Tab #2</tab>
    <tab @click="activeTabId = '3'">Tab #3</tab>

    <tab-content v-if="activeTabId === '1'">
      Content #1
    </tab-content>
    <tab-content v-if="activeTabId === '2'">
      Content #2
    </tab-content>
    <tab-content v-if="activeTabId === '3'">
      Content #3
    </tab-content>
  </tab-container>
</template>
```

Alternative, less flexible syntax.

As far as general development goes, I think the former is much cleaner and lends itself to a better development experience.

Another common use case for render functions is when you are writing a general component library (such as Vuetify). In these cases, you will not know how many tabs the user is going to use, so using `v-if` like above isn't an option. You will need something more generic and generalizable. There are other alternatives, but I've found render functions really useful for writing reusable components.

## 8.2 Creating the Components

One of the nice things about render function components is you can create multiple in the same file. Although I generally like to have one component per file, in this particular case I have no problem putting `<tab-container>`, `<tab-content>` and `<tab>` in the same file. The main reason for this is both

`<tab>` and `<tab-content>` are very simple, and I don't see any use case where you would want to use them outside of nesting them in `<tab-container>`.

Start by creating those two components. We won't be using a `vue` file, but just a plain old `js` file:

```
import { h } from 'vue'

export const TabContent = {
  props: {
    tabId: {
      type: String,
      required: true
    }
  },

  render() {
    return h(this.$slots.default)
  }
}

export const Tab = {
  props: {
    tabId: {
      type: String,
      required: true
    }
  },

  render() {
    return h('div', h(this.$slots.default))
  }
}
```

`Tab` and `TabContent` components using a render function instead of a template.

We do a deep dive on `h` soon - don't worry if you don't understand that fully right now.

Before we go any further, the fact we are working with render functions, which are *just JavaScript*, allows us to make a sneaky refactor and save some boilerplate. Both components have the same props: a `tabId`. We can generalize this with a `withTabId` function and the spread `(...)` operator:

```
const withTabId = (content) => ({
  props: {
    tabId: {
      type: String,
```

```

        required: true
      }
    },
    ...content
  })

export const TabContent = withTabId({
  render() {
    return h(this.$slots.default)
  }
})

export const Tab = withTabId({
  render() {
    return h('div', h(this.$slots.default))
  }
})

```

The withTabId function reduces duplication.

This technique is very useful when making component libraries where many components use similar props.

### 8.3 Filtering Slots by Component

Now we get to the exciting part - the render function for the <tab-container> component. It has one prop - activeTabId:

```

export const TabContainer = {
  props: {
    activeTabId: String
  },

  render() {
    console.log(this.$slots.default())
  }
}

```

Creating the TabContainer component and logging the default slot.

If you prefer Composition API, you could also do this with setup:

```

export const TabContainer = {
  props: {
    activeTabId: String
  },

```

```

    setup(props, { slots }) {
      console.log(slots.default())
    }
  }
}

```

Accessing slots with the Composition API.

I will be using the Options API and a **render** function for this example.

The first thing we need to do is separate the slots. I will use the following example for development:

```

<template>
  <tab-container v-model:activeTabId="activeTabId">
    <tab tabId="1" />
    <tab tabId="2" />

    <tab-content tabId="1" />
    <tab-content tabId="2" />
  </tab-container>
</template>

<script>
import { ref } from 'vue'

import {
  Tab,
  TabContent,
  TabContainer
} from './tab-container.js'

export default {
  components: {
    Tab,
    TabContainer,
    TabContent
  },

  setup() {
    return {
      activeTabId: ref('1')
    }
  }
}
</script>

```

Combining the render function components in a template.

In this example, `this.$slots.default()` would contain *four* slots (technically, we can say four `VNodes`). Two `<tab>` components and two `<tab-content>` components. To make this more clear we will do some “console driven” development.

Create a new app using the above component as the root component. Open a browser and open up the console. You should see something like this:

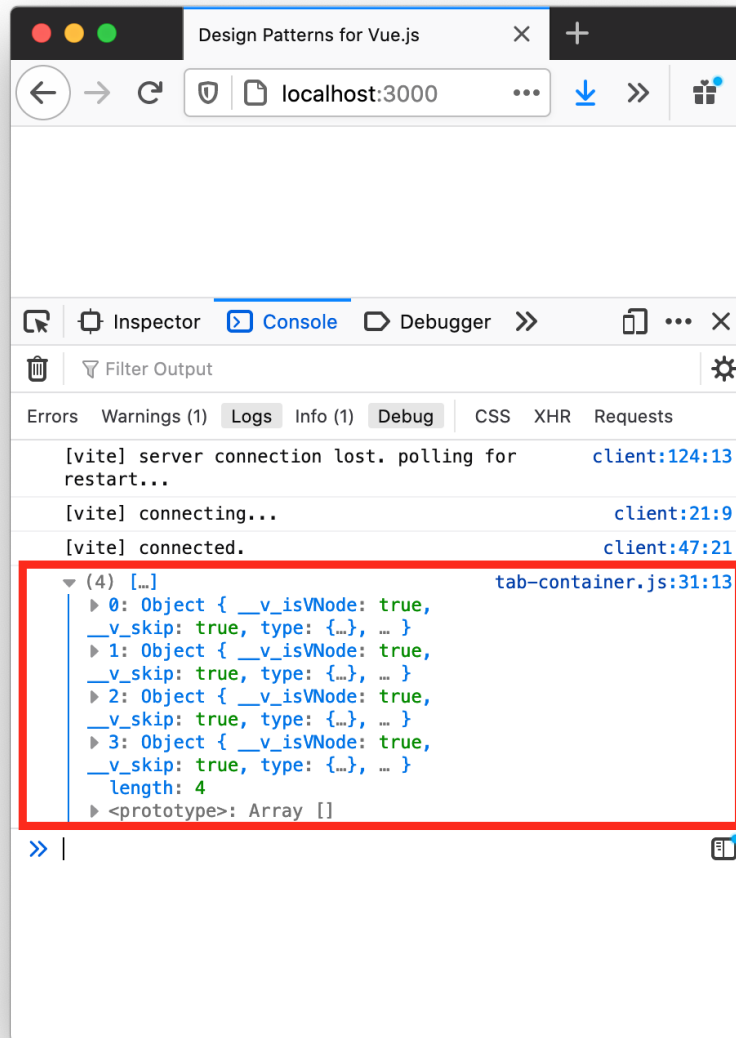


Figure 15: Logging Slots (Array of VNodes)

An array of four complex objects. These are **VNodes** - how Vue internally represents nodes in it's virtual DOM. I expanded the first one and marked some of the relevant properties for this section:





The first one is **children**. This is where the slots go. For example in:

```
<tab tabId="1">Tab #1</tab>
```

There is one child, **Tab #1**. In this case, it is a *text node* - just some text. It could be another **VNode**, which in turn could contain more **VNodes** - a tree like structure.

The next marked property is **props** - this one is pretty obvious, it's the props we passed. In this case, there is just one - **tabId**.

Finally we have **type**. Type can be a few things - for a regular HTML element, such as `<div>`, it would just be `div`. For a component, it contains the entire component. In this case, you can see the component we defined - `<tab>` - which has **props** and **render** attributes.

Now we know how to identify which component a **VNode** is using - the **type** property. Let's use this knowledge to filter the **slots**.

## 8.4 Filtering default slots

The **type** property is a *direct reference* to the component the **VNode** is using. This means we can match using an object and strict equality. If this sounds a bit abstract, let's see it in action and sort the slots into **tabs** and **contents**:

```
export const TabContainer = {
  props: {
    activeTabId: String
  },
  render() {
    const $slots = this.$slots.default()
    const tabs = $slots
      .filter(slot => slot.type === Tab)
    const contents = $slots
      .filter(slot => slot.type === TabContent)

    console.log(
      tabs,
      contents
    )
  }
}
```

Separating the different slots using filter.

Since **type** is a direct reference to the original component (eg, not a copy), we can use `===` (strict equality) to filter the slots.

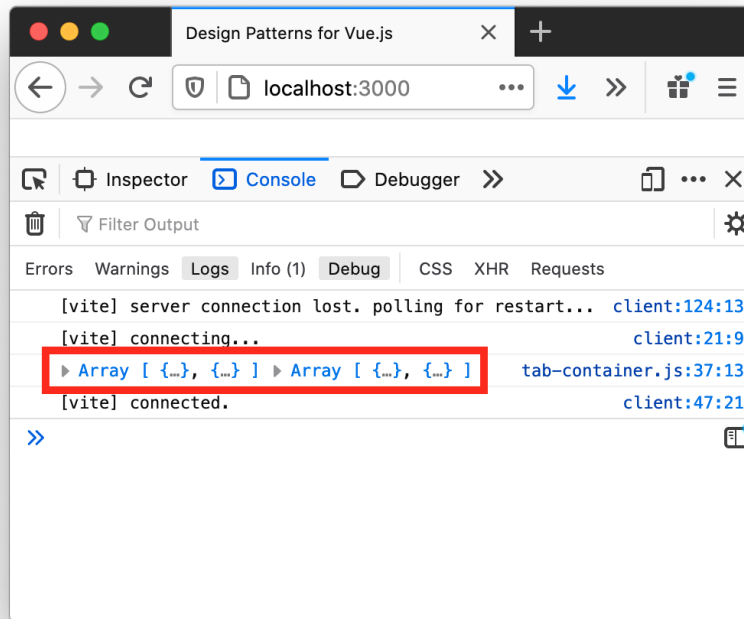


Figure 17: Filtered VNodes

The next goal will be to render the tabs. We will also add some classes to get some nice styling, as well as show which tab is currently selected.

## 8.5 Adding Attributes to Render Functions

First things first, let's render something! Enough console driven development. Import `h` from `vue`, and then `map` over the filtered tabs - I will explain the crazy (amazing?) `h` function afterwards:

```
import { h } from 'vue'

export const TabContainer = {
  props: {
    activeTabId: String
  },

  render() {
    const $slots = this.$slots.default()
    const tabs = $slots
      .filter(slot => slot.type === Tab)
      .map(tab => {
        return h(tab)
      })

    const contents = $slots
      .filter(slot => slot.type === TabContent)

    return h(() => tabs)
  }
}
```

Rendering the tabs using `h`.

Finally, we have something rendering:

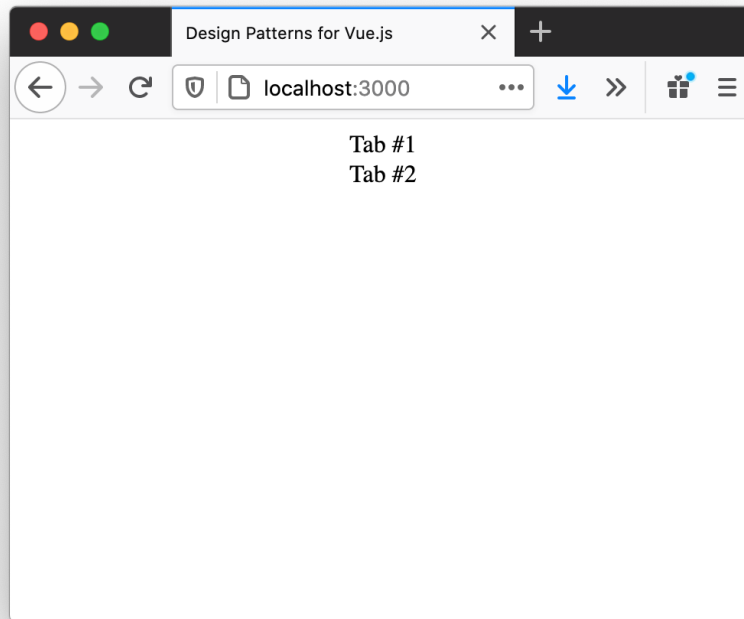


Figure 18: Rendered Tabs

You may have noticed I did `h(() => tabs)` instead of just `return tabs`. `h` also accepts a callback - in which case, it will evaluate the callback function when it renders. I recommend always returning `h(() => /* render function */)` for the final value in `render` - if you don't, you may run into subtle caching issues.

You can also return an array from `render` - this is known as a *fragment*, where there is no root node.

If this looks confusing, don't worry - here comes the `h` crash course.

## 8.6 What is `h`? A Crash Course

A more complex example of a component with a `render` function that returns an array of render functions, consisting of both regular HTML elements and a custom component.

```
const Comp = {
  render() {
    const e1 = h('div')
    const e2 = h('span')
    const e3 = h({
      render() {
        return h('p', {}, ['Some Content'])
      }
    })

    return [
      h(() => e1),
      h(() => e2),
      h(() => e3)
    ]
  }
}
```

A more complex example of a render function with `h`.

We are using `h` to render our tabs - `h(tab)` - where `tab` is a `VNode`, which in turn has a render function that returns `h`. What is `h`? It is derived from the term “hyperscript”, which in turn owes its roots to HTML - specifically the `H`, which stands for *hyper*. `h` is shorter, and easier to type. It can be thought of as “a JavaScript function that creates HTML structures”.

It has quite a few overloads. For example, a minimal usage would be:

```
const e1 = h('div')
```

A minimal `VNode` representing a `div`.

This will create a single `<div>` - not very useful. The second argument can be attributes, represented by an object.

```
const el = h('div', { class: 'tab', foo: 'bar' })`
```

The second argument to `h` is an object containing attributes.

The attributes object can take an attribute - standard or not. This would render:

```
<div class="tab" foo="bar" />
```

The third and final argument is children, usually an array:

```
const el = h('div', { class: 'tab', foo: 'bar' }, ['Content'])`
```

The third argument is the children.

Which renders:

```
<div class="tab" foo="bar">
  Content
</div>
```

You can also pass more `VNodes`, created with nested calls to `h`:

```
const el = h(
  'div',
  {
    class: 'tab',
    foo: 'bar'
  },
  [
    h(
      'span',
      {},
      ['Hello world!']
    )
  ]
)
```

Children can be plain text or `VNodes`.

I spread it out to make it more readable. `render` functions using `h` can get messy - you need to be disciplined. Some tips will follow relating to this. The above call to `h` gives us:

```
<div class="tab" foo="bar">
  <span>Hello world!</span>
</div>
```

As shown above, you are not just limited to standard HTML elements. You can pass a custom component to `h`, too:

```
const Tab = {
  render() {
    return h('span')
  }
}

const el = h('div', {}, [h(Tab), {}, ['Tab #1']])
```

Passing a custom component, `Tab`, as a child.

This can get difficult to read quickly. The main strategy I use to work around this is creating a separate variable for each `VNode`, and returning them all at the end of the `render` function (keep reading to see this in action).

## 8.7 Adding a Dynamic Class Attribute

Now we have a better understanding of `h`, we can add some classes to the `<tab>` components. Each `<tab>` will have a `tab` class, and the active tab will have an `active` class. Update the `render` function:

```
export const TabContainer = {
  props: {
    activeTabId: String
  },

  render() {
    const $slots = this.$slots.default()
    const tabs = $slots
      .filter(slot => slot.type === Tab)
      .map(tab => {
        return h(
          tab,
          {
            class: {
              tab: true,
              active: tab.props.tabId === this.activeTabId
            }
          }
        )
      })

    const contents = $slots
      .filter(slot => slot.type === TabContent)

    return h(() => h('div', { class: 'tabs' }, tabs))
```



```
}  
}
```

Passing an dynamic "active" prop.

Does this look familiar?

```
{  
  class: {  
    tab: true,  
    active: tab.props.tabId === this.activeTabId  
  }  
}
```

A dynamic class binding.

It's v-bind:class syntax! This is how you write v-bind:class="{ tab: true, active: tabId === activeTabId }" in a render function. Here's how it looks in a browser (I added some CSS - grab the CSS from [examples/render-functions/app.vue](#)):

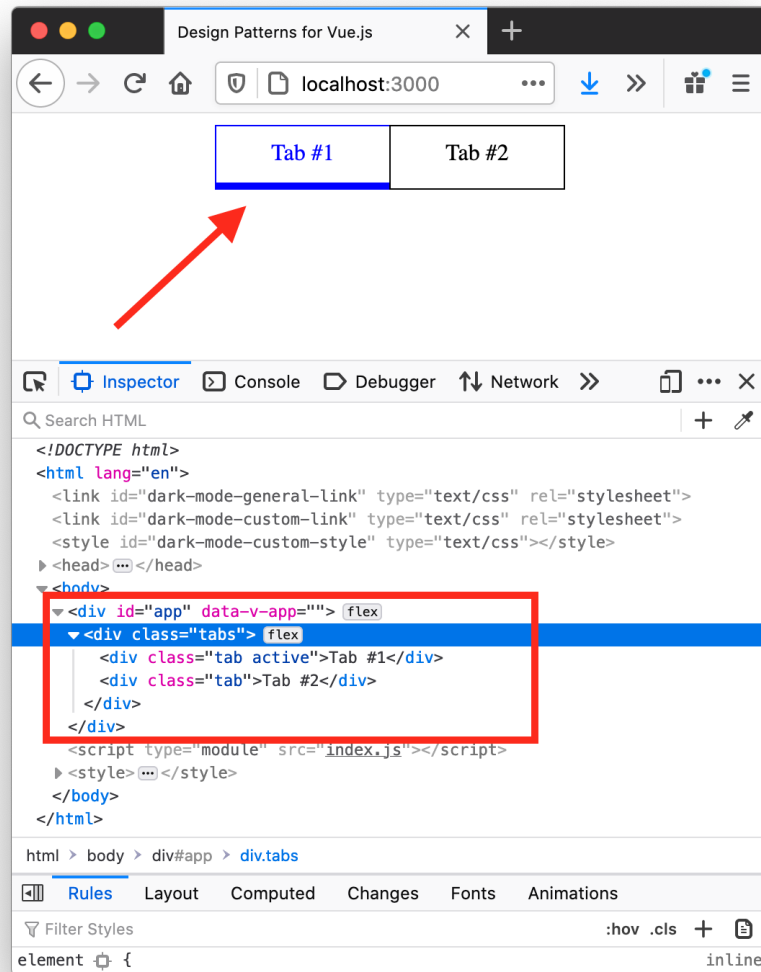


Figure 19: Dynamic Classes

## 8.8 Event Listeners in Render Functions

The active tab needs to update when the user clicks a tab. Let's implement that. Event listeners are much the same as attributes like `class`.

```
{
  class: {
    tab: true,
    active: tab.props.tabId === this.activeTabId
  },
  onClick: () => {
    this.$emit('update:activeTabId', tab.props.tabId)
  }
}
```

`onClick` listener implemented a the render function

This is the render function version of `<tab v-on:click="update:activeTabId(tabId)" />`. `on:click` becomes `onClick`. Events need to be prepended with `on`. This is enough to update the active tab (I added some debugging information):

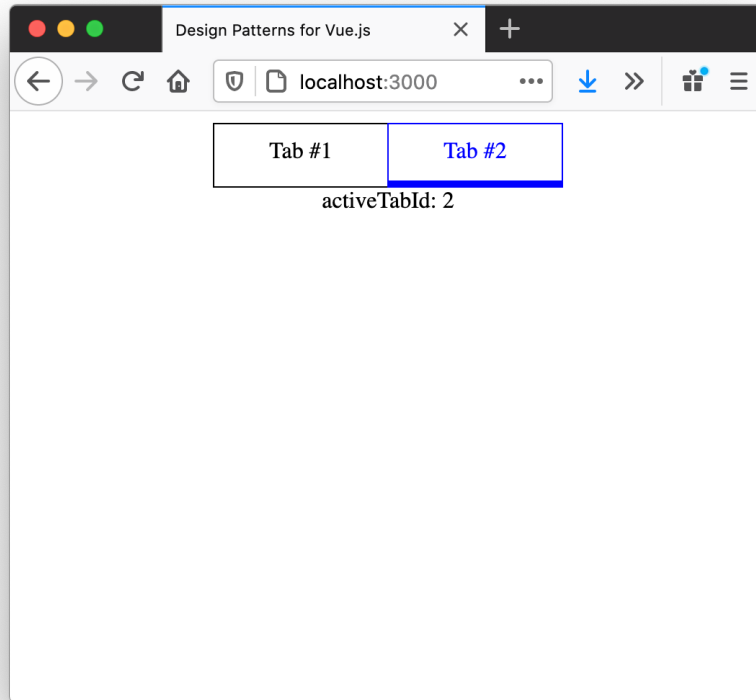


Figure 20: Emitting Events in Render Functions

## 8.9 Filtering Content

The last feature we need to implement is rendering the content - but only the content that matches the `activeTabId`. Instead of using `filter` to get the `contents` `VNodes`, we should use `find` - there will only ever be one tab selected at any given time. Use `find` instead of `filter` in the `render` function:

```
const content = $slots.find(slot =>
  slot.type === TabContent &&
  slot.props.tabId === this.activeTabId
)
```

Finding the active content among the slots.

Finally, we need to change what is returned. Instead of just rendering the tabs, we will render the content as well. Here is the completed `render` function:

```
export const TabContainer = {
  props: {
    activeTabId: String
  },

  render() {
    const $slots = this.$slots.default()
    const tabs = $slots
      .filter(slot => slot.type === Tab)
      .map(tab => {
        return h(
          tab,
          {
            class: {
              tab: true,
              active: tab.props.tabId === this.activeTabId
            },
            onClick: () => {
              this.$emit('update:activeTabId', tab.props.tabId)
            }
          }
        )
      })

    const content = $slots.find(slot =>
      slot.type === TabContent &&
      slot.props.tabId === this.activeTabId
    )

    return [
```

```

    h(() => h('div', { class: 'tabs' }, tabs)),
    h(() => h('div', { class: 'content' }, content)),
  ]
}
}

```

Completed render function for TabContainer.

It's possible to return an array of **VNodes** from **render**, which is what we do here. We kept everything nice and readable by creating separate variables for each of the different elements we are rendering - in this case, **tabs** and **content**.

It works!

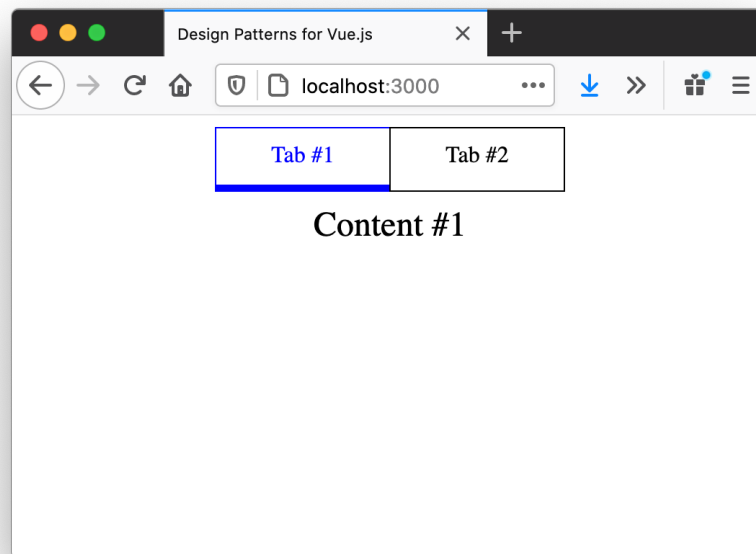


Figure 21: Completed Tabs Component

## 8.10 Testing Render Function Components

Now that we finished the implementation, we should write a test to make sure everything continues working correctly. Writing a test is pretty straight forward - the `render` function from Testing Library works fine with render functions (vue files are compiled into render functions, so all the tests we've been writing have been using `render` functions under the hood).

```
import { render, screen, fireEvent } from '@testing-library/vue'
import App from './app.vue'

test('tabs', async () => {
  render(App)
  expect(screen.queryByText('Content #2')).toBeFalsy()

  fireEvent.click(screen.getByText('Tab #2'))
  await screen.findByText('Content #2')
})
```

Testing render function components is the same as template components.

## 8.11 Exercises

- Try refactoring the components to use a `setup` function. This means instead of using a `render` function, you will return a function from `setup` that handles the rendering.
- Rewrite this example using TypeScript. You will want to use `defineComponent` and the Composition API for maximum type safety. This screenshot illustrates some of the benefits of TypeScript. Combined with declaring `emits`, you can get type safety for both emitted events and props.
- Attempt to refactor the other examples throughout this book to use render functions instead of vue files (these are not included in the solutions - you can email me if you want help writing a specific example using TypeScript and the Composition API).

You can find the completed source code in the GitHub repository under `examples/renderless-password`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

```

import { createApp, defineComponent } from 'vue'

const TabContainer = defineComponent({
  props: {
    activeTabId: {
      type: String
    }
  },

  emits: {
    'update:activeTabId': (tabId: string) => {}
  },

  setup(props, { emit, slots }) {
    // not a valid emit
    emit('update:foo')

    // wrong payload
    emit('update:activeTabId', {})

    // wrong args
    emit('update:activeTabId', '1')

    props.
  }
})

```

activeTabId? (property) act...

Figure 22: Typesafe Component with Render Function



## 9 Dependency Injection with Provide and Inject

You can find the completed source code in the GitHub repository under `examples/provide-inject`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

---

In this section we discuss a pair of functions, `provide` and `inject`. These facilitate *dependency injection* in Vue. This feature was available in Vue 2. In Vue 2, it was common to attach global variables to this Vue prototype and access them via the `this.$`. A common example of this is `this.$router` or `this.$store`. For this reason, `provide` and `inject` were not as commonly used. With Vue 3 and the Composition API discouraging mutating the global Vue prototype, dependency injection with `provide` and `inject` is more common.

Instead of providing a toy example, we will see a real use case by building a simple store (like Vuex) and making it available via a `useStore` composable. This will use `provide` and `inject` under the hood. There are other ways to implement a `useStore` function, for example simply importing and exporting a global singleton. We will see why `provide` and `inject` are a better way of sharing a global variable.

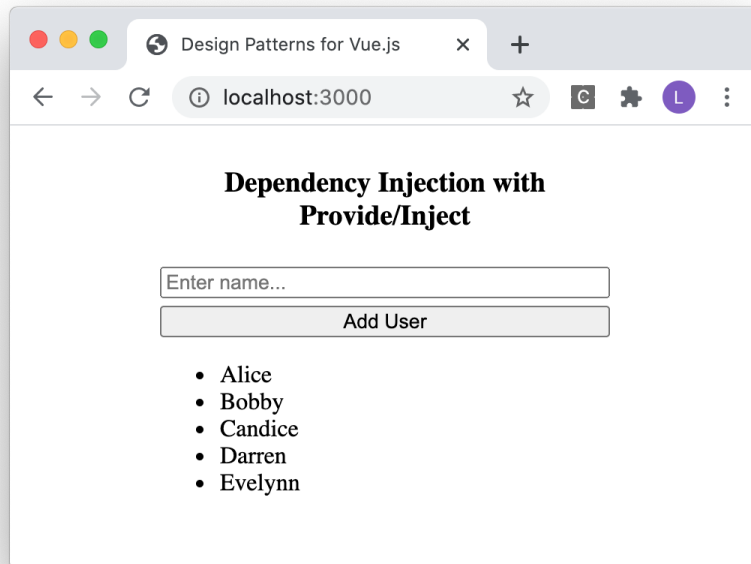


Figure 23: Completed demo app

## 9.1 A Simple Store

Let's quickly define a dead simple store. We won't have a complex API like Vuex - just a class with some methods. Let's start with a reactive state, and expose it in readonly fashion via a `getState` function.

```
import { reactive, readonly } from 'vue'

export class Store {
  #state = {}

  constructor(state) {
    this.#state = reactive(state)
  }

  getState() {
    return readonly(this.#state)
  }
}
```

A simple store with private state and a readonly accessor.

If you haven't seen the `#state` syntax before, this is a private property - one of the newer features to classes in JavaScript. At the time of writing this, it only works in Chrome natively. You can omit the `#` if you like - it will still work just fine.

The `#` means that the property can only be accessed inside the class instance. So `this.#state` works for methods declared inside the `Store` class, but `new Store({ count: 1 }).#state.count` is not allowed. Instead, we will access the state in a readonly manner using `getState()`.

We pass `state` to the constructor to let the user seed the initial state. We will take the disciplined approach and write a test.

```
import { Store } from './store.js'

describe('store', () => {
  it('seeds the initial state', () => {
    const store = new Store({
      users: []
    })

    expect(store.getState()).toEqual({ users: [] })
  })
})
```

The tests verifies everything is working correctly.

## 9.2 Usage via import

Let's get something rendering before we go. Export a new instance of the store:

```
import { reactive, readonly } from 'vue'

export class Store {
  // ...
}

export const store = new Store({
  users: [{ name: 'Alice' }]
})
```

Exporting the store as a global singleton with some initial state.

Next, import it into your component and iterate over the users:

```
<template>
  <ul>
    <li
      v-for="user in users"
      :key="user"
    >
      {{ user.name }}
    </li>
  </ul>
</template>

<script>
import { computed } from 'vue'
import { store } from './store.js'

export default {
  setup() {
    return {
      users: computed(() => store.getState().users)
    }
  }
}
</script>
```

accessing the state via the the imported store.

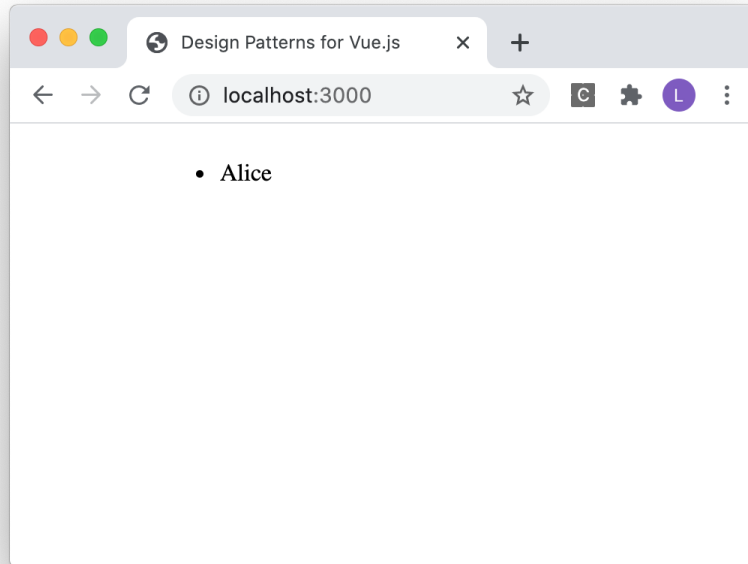


Figure 24: Displaying a user from the store state.

It works! Good progress - I added a tiny bit of CSS as well, grab that from the source code.

This single shared **store** is known as a *global singleton*.

We will allow adding more users via a form - but first let's add a UI test using Testing Library.

```
import { render, screen, fireEvent } from '@testing-library/vue'
import { Store } from './store.js'
import Users from './users.vue'

describe('store', () => {
  it('seeds the initial state', () => {
    // ...
  })

  it('renders a user', async () => {
    render(Users, {
      global: {
```

```

        provide: {
          store: new Store({
            users: []
          })
        }
      })

      await fireEvent.update(screen.getByRole('username'), 'Alice')
      await fireEvent.click(screen.getByRole('submit'))
      await screen.findByText('Alice')
    })
  })
})

```

### UI test with Testing Library

Working great! We do not want to hard code any users in the store, though. This highlights one of the downsides of a global singleton - no easy way to initialize or update the state for testing purposes. Let's add a feature to create new users via a form, and then that way.

## 9.3 Adding a users forms

To add a user, we will first create a `addUser` function to the store:

```

import { reactive, readonly } from 'vue'

export class Store {
  #state = {}

  // ...

  addUser(user) {
    this.#state.users.push(user)
  }
}

export const store = new Store({
  users: []
})

```

`addUser` can access the private state because it is declared in the `Store` class.

I also removed the initial user, Alice, from the store. Update the tests - we can test `addUser` in isolation.

```

describe('store', () => {
  it('seeds the initial state', () => {
    // ...
  })

  it('renders a user', async () => {
    // ...
  })

  it('adds a user', () => {
    const store = new Store({
      users: []
    })

    store.addUser({ name: 'Alice' })

    expect(store.getState()).toEqual({
      users: [{ name: 'Alice' }]
    })
  })
})

```

Testing addUser in isolation - no component, no mounting.

The UI test is now failing. We need to implement a form that calls addUser:

```

<template>
  <form @submit.prevent="handleSubmit">
    <input v-model="username" />
    <button>Add User</button>
  </form>
  <ul>
    <li
      v-for="user in users"
      :key="user"
    >
      {{ user.name }}
    </li>
  </ul>
</template>

<script>
import { ref, computed } from 'vue'
import { store } from './store.js'

export default {

```

```
setup() {  
  const username = ref('')  
  const handleSubmit = () => {  
    store.addUser({ name: username.value })  
    username.value = ''  
  }  
  
  return {  
    username,  
    handleSubmit,  
    users: computed(() => store.getState().users)  
  }  
}  
}  
</script>
```

A form to create new users.

Great! The test now passes - again, I added a tiny bit of CSS and a nice title, which you can get in the source code if you like.



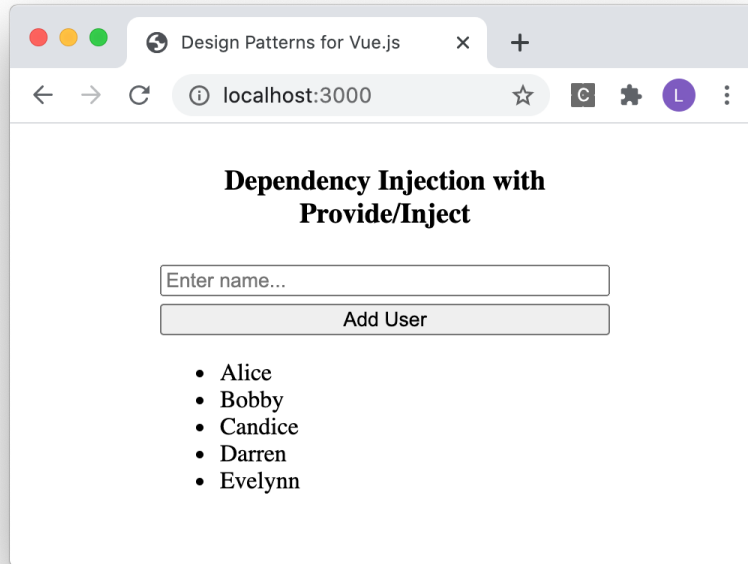


Figure 25: Completed app

## 9.4 Provide/Inject to Avoid Cross Test Contamination

Everything looks to be working on the surface, but we will eventually run into a problem as our application grows: shared state across tests. We have a *single* store instance for all of our tests - when we mutate the state, this change will impact all the other tests, too.

Ideally each test should run in isolation. We can't isolate the store if we are importing the same global singleton into each of our tests. This is where **provide** and **inject** come in handy.

This diagram, taken from this official documentation, explains it well:

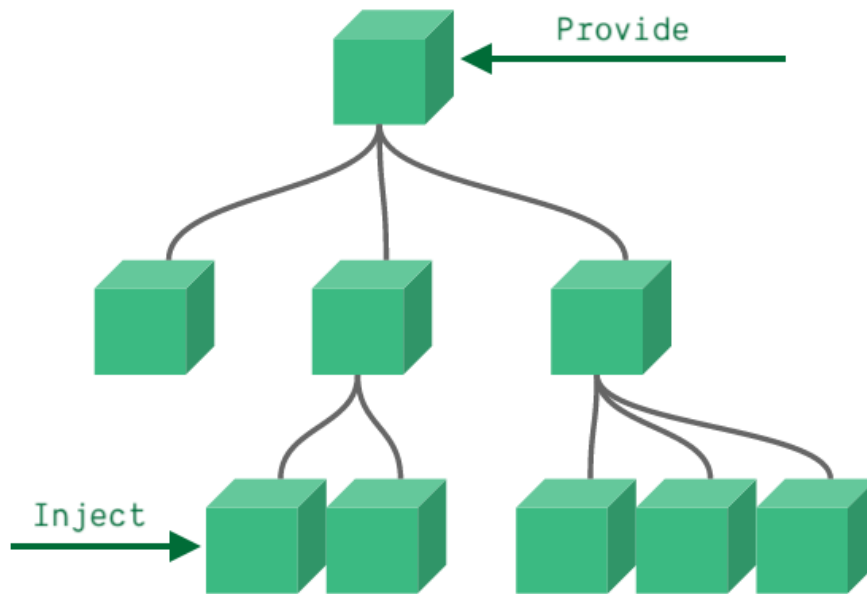


Figure 26: Provide/Inject diagram. Credit: Vue documentation.

Let's say you have a component, `Parent.vue`, that looks like something this:

```
<template>
  <child />
</template>

<script>
import { provide } from 'vue'

export default {
  setup() {
    const theColor = 'blue'
    provide('color', theColor)
  }
}
</script>
```

We are making a `color` variable available to *any* child component that might want access to it, no matter how deep in the component hierarchy it appears.

Child.vue might look like this:

```
<template>
  <!-- renders Color is: blue -->
  Color is: {{ color }}
</template>

<script>
import { inject } from 'vue'

export default {
  setup() {
    const color = inject('color')
    return {
      color
    }
  }
}
</script>
```

You can pass anything to **provide** - including a reactive store. Let's do that. Head to the top level file where you create your app (mine is `index.js`; see the source code for a complete example):

```
import { createApp } from 'vue'
import { store } from './examples/provide-inject/store.js'
import Users from './examples/provide-inject/users.vue'

const app = createApp(Users)
app.provide('store', store)
app.mount('#app')
```

Using **provide** to make the store available in all the components.

You can call **provide** in a component's **setup** function. This makes the provided value available to all that component's children (and their children, etc). You can also call **provide** on **app**. This will make your value available to all the components, which is what we want to do in this example.

Instead of importing the store, we can now just call `const store = inject('store')`:

```
<template>
  <!-- ... -->
</templat>

<script>
import { ref, inject, computed } from 'vue'
```

```

export default {
  setup() {
    const store = inject('store')
    const username = ref('')

    const handleSubmit = () => {
      // ...
    }

    return {
      username,
      handleSubmit,
      users: computed(() => store.getState().users)
    }
  }
}
</script>

```

Using inject to access the store.

## 9.5 Provide in Testing Library

The final UI test is failing. We did `provide('store', store)` when we created our app, but we didn't do it in the test. Testing Library has a mounting option specifically for `provide` and `inject`: `global.provide`:

```

import { render, screen, fireEvent } from '@testing-library/vue'
import { Store } from './store.js'
import Users from './users.vue'

describe('store', () => {
  it('seeds the initial state', () => {
    // ...
  })

  it('adds a user', () => {
    // ...
  })

  it('renders a user', async () => {
    render(Users, {
      global: {
        provide: {
          store: new Store({
            users: []

```

```

    })
  }
}
})

    await fireEvent.update(screen.getByRole('username'), 'Alice')
    await fireEvent.click(screen.getByRole('submit'))
    await screen.findByText('Alice')
  })
})

```

Using the `global.provide` mounting option.

Everything is passing again. We now can avoid cross test contamination - it's easy to provide a new store instance using `global.provide`.

## 9.6 A `useStore` composable

We can write a little abstraction to make using our store a bit more ergonomic. Instead of typing `const store = inject('store')` everywhere, it would be nice to just type `const store = useStore()`.

Update the store:

```

import { reactive, readonly, inject } from 'vue'

export class Store {
  // ...
}

export const store = new Store({
  users: []
})

export function useStore() {
  return inject('store')
}

```

A `useStore` composable.

Now update the component:

```

<template>
  <!-- ... -->
</template>

<script>

```

```

import { ref, computed } from 'vue'
import { useStore } from './store.js'

export default {
  setup() {
    const store = useStore()
    const username = ref('')

    const handleSubmit = () => {
      store.addUser({ name: username.value })
      username.value = ''
    }

    return {
      username,
      handleSubmit,
      users: computed(() => store.getState().users)
    }
  }
}
</script>

```

Using the useStore composable.

All the test are still passing, so we can be confident everything still works.

Now anywhere you need access to the store, just call `useStore`. This is the same API Vuex uses. It's a common practice to make global singletons available via a `useXXX` function, which uses `provide` and `inject` under the hood.

## 9.7 Exercises

1. Update the store to have a `removeUser` function. Test it in isolation.
2. Add a button next to each user - clicking the button should remove them from the store. Use the `removeUser` function here.
3. Write a UI test to verify this works using Testing Library. You can set up the store with a user by using `globals.provide` and passing in a store with a user already created.

You can find the completed source code in the GitHub repository under `examples/provide-inject`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

## 10 Modular Components, v-model, and the Strategy Pattern

You can find the completed source code in the GitHub repository under `examples/reusable-date-time`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

---

In this section we will author a reusable date component. Usage will be like this:

```
<date-time
  v-model="date"
  :serialize="..."
  :deserialize="..."
/>
```

The goal - a `<datetime>` component that works with any `DateTime` library via the strategy pattern.

The finished component will look like this:



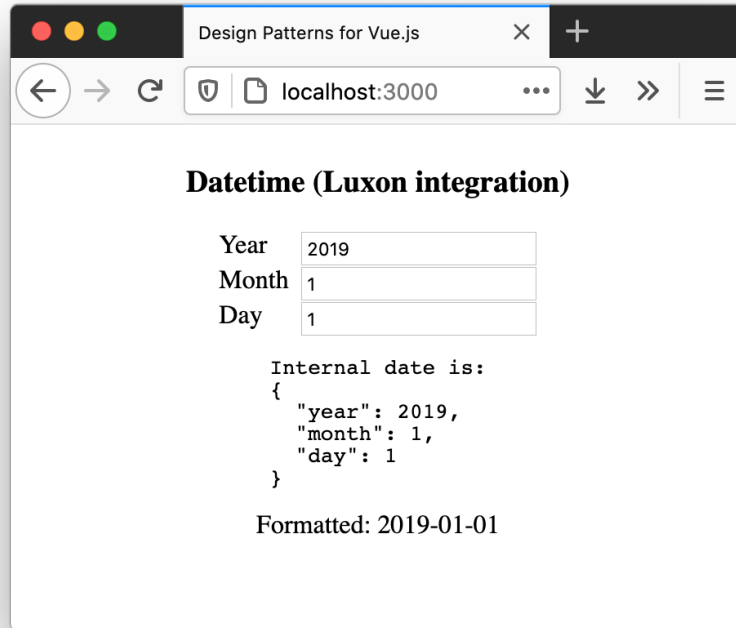


Figure 27: Completed DateTime Component

There are three props: `v-model`, `serialize` and `deserialize`. More on what `serialize` and `deserialize` are soon.

The idea is that the `date` value passed to `v-model` can use whichever DateTime library the developer wants to use. We want to allow developers to choose their a DateTime library, instead of mandating a specific one.

Some applications use the native JavaScript `Date` object (don't do this; it's not a very good experience). Older applications will often use Moment and newer ones common opt for Luxon.

I'd like to support both - and any other library the user might choose to use! In other words, we want the component to be agnostic - it should not be coupled to a specific date time library.

One way to handle this would be to pick a simple format of our own, for example `YYYY-MM-DD`, and then have the user wrap the component and provide

a custom integration layer. For example a user wanting to use Luxon might wrap `<date-time>` in their own `<date-time-luxon>` component:

```
<template>
  <date-time
    :modelValue="date"
    @update:modelValue="updateDate"
  />
</template>

<script>
import { ref } from 'vue'
import { DateTime } from 'luxon'

export default {
  setup() {
    return {
      date: ref(DateTime.local()),
      updateDate: (value) => {
        // some logic to turn value which is
        // YYYY-MM-DD into Luxon DateTime
      }
    }
  }
}
```

Wrapping ‘`<datetime>`’ to provide Luxon integration.

This might work okay - now you can put your `<luxon-date-time>` on npm to share, listing `luxon` as a `peerDependency` in `package.json`. But other people may have different ways they’d like to validate the date from `v-model` before calling `updateValue`, or have a different opinion on the API `<date-time-luxon>` should expose. Can we be more flexible? What about moment? Do we need to make a `<moment-date-time>` component too?

The core problem of the “wrapper” solution is you are adding another abstraction - another layer. Not ideal. The problem that needs solving is *serializing* and *deserializing* `v-model` in a library agnostic way. The `<date-time>` component doesn’t need to know the specifics of the `DateTime` object it is dealing with.

Here is the API I am proposing to make `<date-time>` truly agnostic, not needing to know the implementation details of the date library:

```
<date-time
  v-model="date"
  :serialize="serialize"
  :deserialize="deserialize"
```

```
</>
```

`<datetime>` with `serialize` and `deserialize` props.

`date` can be whatever you want - `serialize` and `deserialize` will be the functions that tell `<date-time>` how to handle the value, which will be some kind of `DateTime` object. This pattern is generalized as the “strategy” pattern. A textbook definition is as follows:

In computer programming, the strategy pattern (also known as the policy pattern) is a behavioral software design pattern that enables selecting an algorithm at runtime. Instead of implementing a single algorithm directly, code receives runtime instructions as to which in a family of algorithms to use ([https://en.wikipedia.org/wiki/Strategy\\_pattern](https://en.wikipedia.org/wiki/Strategy_pattern)). The strategy lets the algorithm vary independently from clients that use it.

The key part is the last sentence. We push the onus of selecting the algorithm onto the developer.

A diagram might make this more clear:

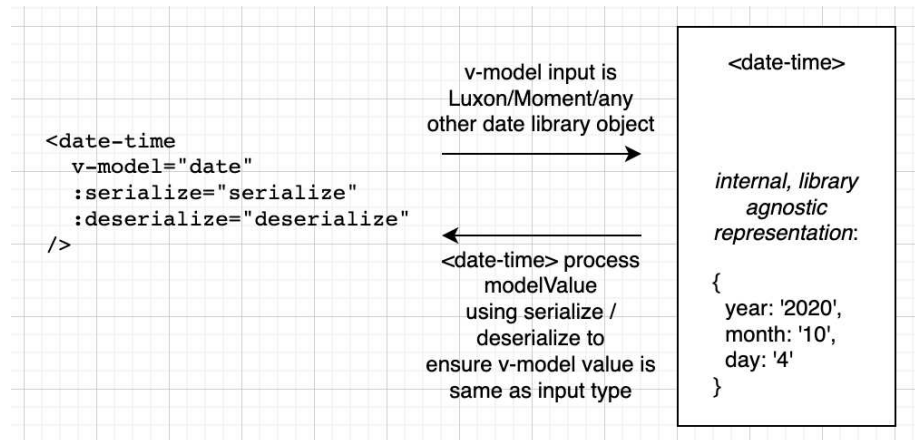


Figure 28: DateTime data flow

In this diagram, the internal implementation of `<date-time>` is on the right. Regardless of what the developer passes to `v-model`, we will convert it to a framework agnostic representation. In this case, it's `{ year: '', month: '', day: '' }`. We then transform it *back* to the desired value when it is updated.

If the developer was using Luxon, the workflow would be something like `luxon.DateTime() -> { year: '', month: '', day: '' } ->`

`luxon.DateTime()`. The input *and* output is a Luxon `DateTime` - the developer doesn't need to know or care about the internal representation.

## 10.1 Foundations of v-model

Before implementing the strategy pattern (in this example, the `serialize` and `deserialize` functions), let's write the base for `<date-time>`. It will use `v-model`. This means we receive a `modelValue` prop and update the value by emitting a `update:modelValue` event. To keep things simple, I will just use 3 `<input>` elements for the year, month and day.

```
<template>
  <input :value="modelValue.year" @input="update($event, 'year')" />
  <input :value="modelValue.month" @input="update($event, 'month')" />
  <input :value="modelValue.day" @input="update($event, 'day')" />
</pre>
<pre>
Internal date is:
{{ modelValue }}
</pre>
</template>

<script>
import { reactive, watch, computed } from 'vue'
import { DateTime } from 'luxon'

export default {
  props: {
    modelValue: {
      type: Object
    },
  },

  setup(props, { emit }) {
    const update = ($event, field) => {
      const { year, month, day } = props.modelValue
      let newValue
      if (field === 'year') {
        newValue = { year: $event.target.value, month, day }
      }
      if (field === 'month') {
        newValue = { year, month: $event.target.value, day }
      }
      if (field === 'day') {
        newValue = { year, month, day: $event.target.value }
      }
    }
  }
}
```

```

        emit('update:modelValue', newValue)
    }

    return {
        update
    }
}
}
</script>

```

Implementing v-model for the datetime.

Usage is like this:

```

<template>
  <date-time v-model="dateLuxon" />
  {{ dateLuxon }}
</template>

<script>
import { ref } from 'vue'
import dateTime from './date-time.vue'

export default {
  components: { dateTime },
  setup() {
    const dateLuxon = ref({
      year: '2020',
      month: '01',
      day: '01',
    })

    return {
      dateLuxon
    }
  }
}
</script>

```

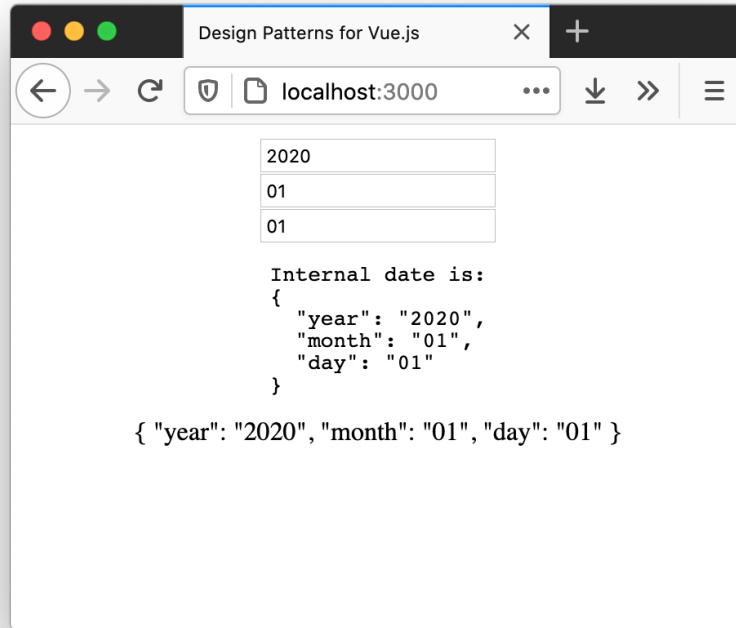


Figure 29: Rendering the Date Inputs

I called the variable `dateLuxon` since we will eventually change it to be a Luxon `DateTime`. For now it is just a plain JavaScript object, made reactive via `ref`. This is all standard - we made our custom component work with `v-model` by binding to `:value` with `modelValue`, and update the original value in the parent component with `emit('update:modelValue')`.

## 10.2 Deserializing for `modelValue`

We have established the internal API. This is how the `<date-time>` component will manage the value. For notation purposes, if we were to write an interface in TypeScript, it would look like this:

```
interface InternalDateTime {
  year?: string
  month?: string
```

```
    day?: string
  }
```

We will now work on the `deserialize` prop, which is a function that will convert any object (so a Luxon `DateTime` object, or Moment `Moment` object) into an `InternalDateTime`. This is the representation the `<date-time>` component uses internally.

### 10.3 Deserializing `modelValue`

The next goal is to write a `deserialize` function. In pseudocode:

```
export function deserialize(inputDateTime) {
  // do whatever needs to be done to convert
  // the inputDateTime to a JS object with
  // { year, month, day }
  return yearMonthDateObject
}
```

I will use Luxon's `DateTime` to demonstrate. You can create a new `DateTime` like this:

```
import { DateTime } from 'luxon'

const date = DateTime.fromObject({
  year: '2020',
  month: '10',
  day: '02'
})
```

The goal is to get from our input from `v-model`, in this case a Luxon `DateTime`, to our internal representation, `InternalDateTime`. This conversion is trivial in the case of Luxon's `DateTime`. You can just do `date.get()` passing in `year`, `month` or `day`. So our `deserialize` function looks like this:

```
// value is what is passed to `v-model`
// in this example a Luxon DateTime
// we need to return an InternalDateTime
export function deserialize(value) {
  return {
    year: value.get('year'),
    month: value.get('month'),
    day: value.get('day')
  }
}
```

Let's update the usage:

```

<template>
  <date-time
    v-model="dateLuxon"
    :deserialize="deserialize"
  />
  {{ dateLuxon.toISODate() }}
</template>

<script>
import { ref } from 'vue'
import dateTime from './date-time.vue'
import { DateTime } from 'luxon'

export function deserialize(value) {
  return {
    year: value.get('year'),
    month: value.get('month'),
    day: value.get('day')
  }
}

export default {
  components: { dateTime },

  setup() {
    const dateLuxon = ref(DateTime.fromObject({
      year: '2019',
      month: '01',
      day: '01',
    })))

    return {
      dateLuxon,
      deserialize
    }
  }
}
</script>

```

Next, update `<date-time>` to use the new `deserialize` prop:

```

<template>
  <input :value="date.year" @input="update($event, 'year')" />
  <input :value="date.month" @input="update($event, 'month')" />
  <input :value="date.day" @input="update($event, 'day')" />
</pre>

```



```

Internal date is:
{{ date }}
</pre>
</template>

<script>
import { reactive, computed } from 'vue'

export default {
  props: {
    modelValue: {
      type: Object
    },
    deserialize: {
      type: Function
    }
  },

  setup(props, { emit }) {
    const date = computed(() => {
      return props.deserialize(props.modelValue)
    })

    const update = ($event, field) => {
      const { year, month, day } = props.modelValue
      let newValue
      if (field === 'year') {
        newValue = { year: $event.target.value, month, day }
      }
      if (field === 'month') {
        newValue = { year, month: $event.target.value, day }
      }
      if (field === 'day') {
        newValue = { year, month, day: $event.target.value }
      }
      emit('update:modelValue', newValue)
    }

    return {
      update,
      date
    }
  }
}
</script>

```

The main changes are:

1. We now need to use a `computed` property for `modelValue`, to ensure it is correctly transformed into our `InternalDateTime` representation.
2. We use `deserialize` on the `modelValue` in the `update` function when preparing to update `modelValue`.

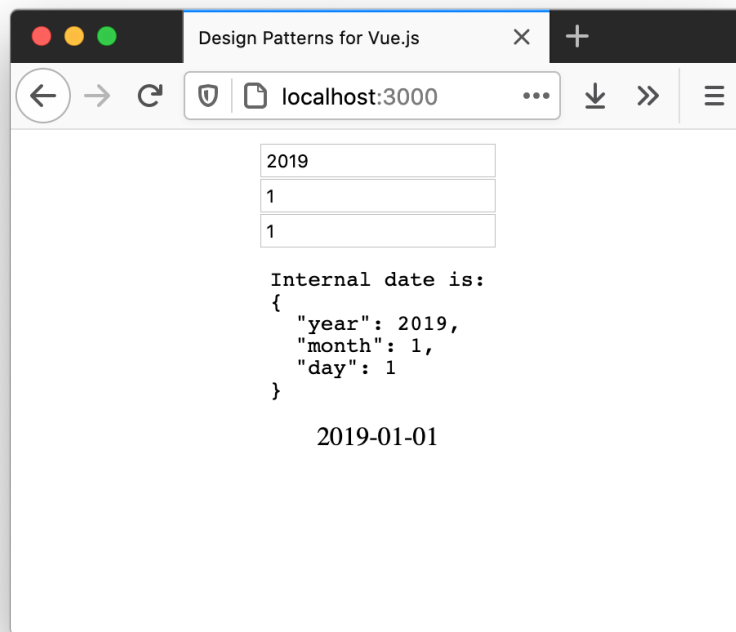


Figure 30: Using the serialize prop.

Now would generally be a good time to write a test for the `deserialize` function. Notice I exported it independently of the Vue component, and it does not use the Vue reactivity system. This is intentional. It's a pure function, so it's very easy to test. For brevity, the tests are not shown, but you can find them in the GitHub repository.

This implementation currently works - kind of - it displays the correct values in the `<input>` elements, but you cannot update the value. We need the opposite of `deserialize` - `serialize`.

## 10.4 Serializing modelValue

We need to ensure are calling `emit('update:modelValue')` with a Luxon `DateTime` now, not an `InternalDateTime` object, since that is what the developer expects. Remember, the input and output value needs to be of whichever `DateTime` library the developer has provided.

Let's write a `serialize` function to transform the value. It's simple. Luxon's `DateTime.fromObject` happens to take an object with the same shape as our `InternalDateTime` - `{ year, month, day }`. We will see a more complex example with the Moment integration.

```
export function serialize(value) {  
  return DateTime.fromObject(value)  
}
```

Again, update the usage.

```
<template>  
  <date-time  
    v-model="dateLuxon"  
    :deserialize="deserialize"  
    :serialize="serialize"  
  />  
  {{ dateLuxon.toISODate() }}  
</template>  
  
<script>  
import { ref } from 'vue'  
import dateTime from './date-time.vue'  
import { DateTime } from 'luxon'  
  
// ...  
  
export function serialize(value) {  
  return DateTime.fromObject(value)  
}
```

```

export default {

  // ...

  return {
    dateLuxon,
    deserialize,
    serialize
  }
}
}
</script>

```

I added a `:serialize` prop and returned `serialize` from the `setup` function. Next, we need to call `serialize` every time we try to update `modelValue`. Update `<date-time>`:

```

<template>
  <!--
    Omitted for brevity.
    Nothing to change here right now.
  -->
</template>

<script>
import { computed } from 'vue'
import { DateTime } from 'luxon'

export default {
  props: {
    modelValue: {
      type: Object
    },
    serialize: {
      type: Function
    },
    deserialize: {
      type: Function
    }
  },

  setup(props, { emit }) {

    // ...

```

```

const update = ($event, field) => {
  const { year, month, day } = props.deserialize(props.modelValue)
  let newValue

  // ...

  emit('update:modelValue', props.serialize(newValue))
}

return {
  update,
  date
}
}
</script>

```

All that changed was declaring the `serialize` prop and calling `props.serialize` when emitting the new value.

It works! Kind of - as long as you only enter value numbers. If you enter a 0 for the day, all the inputs show `NaN`. We need some error handling.

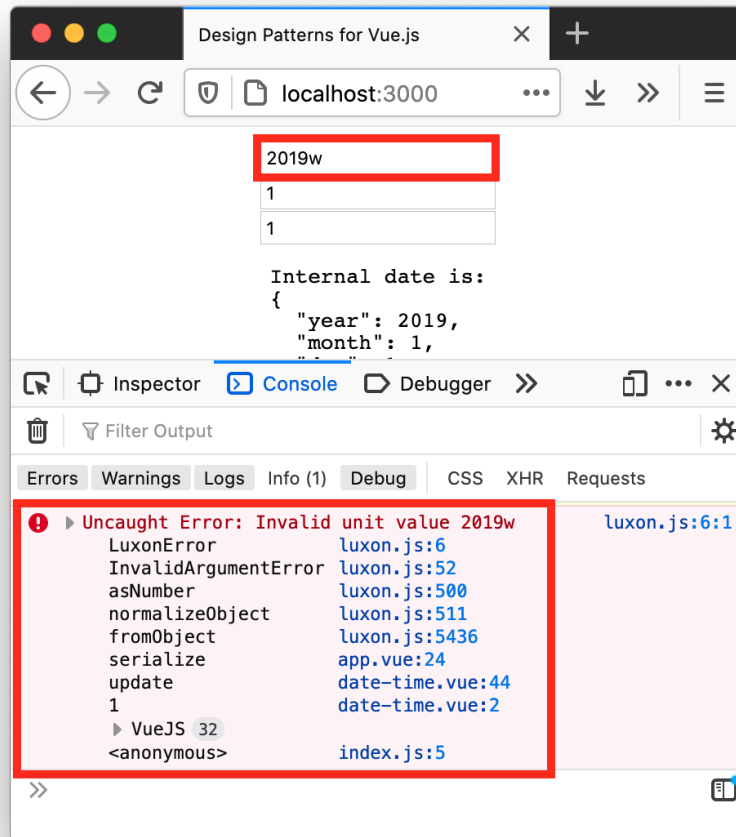


Figure 31: Serializing/Deserializing without error handling.

## 10.5 Error Handling

In the case of an error - either we could not serialize or deserialize the value - we will just return the current input value, and give the user a chance to fix things.

Let's update `serialize` to be more defensive:

```
export function serialize(value) {
  try {
    const obj = DateTime.fromObject(value)
    if (obj.invalid) {
      return
    }
  } catch {
    return
  }

  return DateTime.fromObject(value)
}
```

In the case that we failed to serialize the value, we just return `undefined`. Update the `emit` in `<date-time>` to use this new logic; if the value is invalid, we simply do not update `modelValue`:

```
export default {
  props: {
    // ...
  },

  setup(props, { emit }) {
    // ...
    const update = ($event, field) => {
      const { year, month, day } = props.modelValue
      let newValue

      // ...

      const asObject = props.serialize(newValue)
      if (!asObject) {
        return
      }
      emit('update:modelValue', asObject)
    }

    return {
      update,
      date
    }
  }
}
```

```
    }  
  }  
}
```

I just added a check - `if (!isObject)` and return early if the `props.serialize` did not return a value.

Now everything works correctly, and `<date-time>` will only update `modelValue` if the date is valid. This behavior is a design decision I made; you could do something different depending on how you would like your `<date-time>` to work.

Adding support for Moment is not especially difficult or interesting - it is left as an exercise, and the solution included in the source code.

## 10.6 Deploying

The goal here was to create a highly reusable `<date-time>` component. If I was going to release this on npm, there is a few things I'd do.

1. Remove `serialize` and `deserialize` from the `<date-time>` component and put them into another file. Perhaps one called `strategies.js`.
2. Write a number of strategies for popular `DateTime` libraries (Luxon, Moment etc).
3. Build and bundle the component and strategies separately.

This will allow developers using tools like webpack or rollup to take advantage of “tree shaking”. When they build their final bundle for production, it will only include the `<date-time>` component and the strategy they are using. It will also allow the developer to provide their own more opinionated strategy.

To make the component even more reusable, we could consider writing it as a renderless component, like the one described in the renderless components section.

## 10.7 Exercises

- We did not add any tests for `serialize` or `deserialize`; they are pure functions, so adding some is trivial. See the source code for some tests.
- Add support for another date library, like Moment. Support for Moment is implemented in the source code.
- Add hours, minutes, seconds, and AM/PM support.
- Write some tests with Testing Library; you can use `fireEvent.update` to update the value of the `<input>` elements.

You can find the completed source code in the GitHub repository under `examples/reusable-date-time`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.



## 11 Grouping Features with Composables

You can find the completed source code in the GitHub repository under examples/composition:  
<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

Vue 3's flagship feature is The Composition API; it's main selling point is to easily group and reuse code by *feature*. In this section we will see some techniques to write testable composables by building a tic tac toe game, including undo and redo.

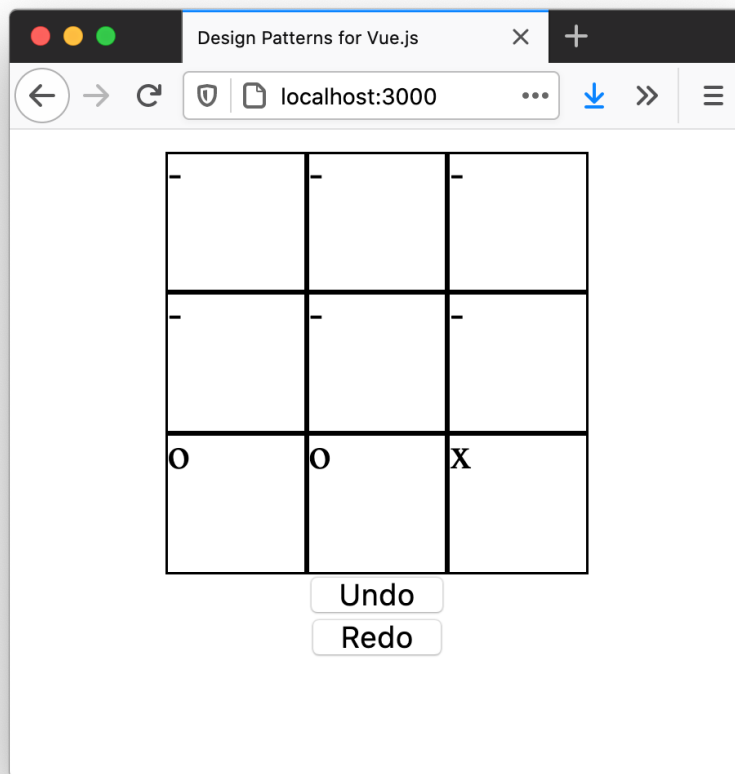


Figure 32: Completed Game

The API we will end with looks like this:

```
export default {
  setup() {
    const {
      currentBoard,
      makeMove,
      undo,
      redo
    } = useTicTacToe()

    return {
      makeMove,
      currentBoard
    }
  }
}
```

Final API.

`currentBoard` is a computed property that looks like this:

```
[
  ['x', 'o', '-'],
  ['x', 'o', 'x'],
  ['- ', 'o', '-']
]
```

Example game board represented as multi-dimensional array.

`makeMove` is a function that takes two arguments: `col` and `row`. Given this board:

```
[
  ['- ', '- ', '- '],
  ['- ', '- ', '- '],
  ['- ', '- ', '- ']
```

Calling `makeMove({ row: 0, col: 1 })` would yield the following board (where `o` goes first)

```
[
  ['- ', 'o', '- '],
  ['- ', '- ', '- '],
  ['- ', '- ', '- ']
```

We will also support `undo` and `redo`, so you can go back and see how the game progressed. Implementing this will be an exercise, and the solution is

included in the final source code.

## 11.1 Defining the Initial Board

Let's start with some way to maintain the game state. I will call this variable `initialBoard`:

```
const initialBoard = [
  ['-', '-', '-'],
  ['-', '-', '-'],
  ['-', '-', '-']
]
```

Initial board.

Before diving too far into the game logic, let's get something rendering. Remember we want to keep a history of the game for undo/redo? This means instead of overriding the current game state each move, we should just create a new game state and push it into an array. Each entry will represent a move in the game. We also need the board to be reactive, so Vue will update the UI. We can use `ref` for this. Update the code:

```
import { ref, readonly } from 'vue'

export function useTicTacToe() {
  const initialBoard = [
    ['-', '-', '-'],
    ['-', '-', '-'],
    ['-', '-', '-']
  ]

  const boards = ref([initialBoard])

  return {
    boards: readonly(boards)
  }
}
```

The start of a `useTicTacToe` composable.

I made the board `readonly`; I don't want to update the game state direct in the component, but via a method we will write soon in the composable.

Let's try it out! Create a new component and use the `useTicTacToe` function:

```
<template>
  <div v-for="(row, rowIdx) in boards[0]" class="row">
    <div
```

```

      v-for="(col, colIdx) in row"
      class="col"
    >
      {{ col }}
    </div>
  </div>
</template>

<script>
import { useTicTacToe } from './tic-tac-toe.js'

export default {
  setup() {
    const { boards } = useTicTacToe()

    return {
      boards
    }
  }
}
</script>

<style>
.row {
  display: flex;
}

.col {
  border: 1px solid black;
  height: 50px;
  width: 50px;
}
</style>

```

Testing out the new useTicTacToe composable.

Great! It works:

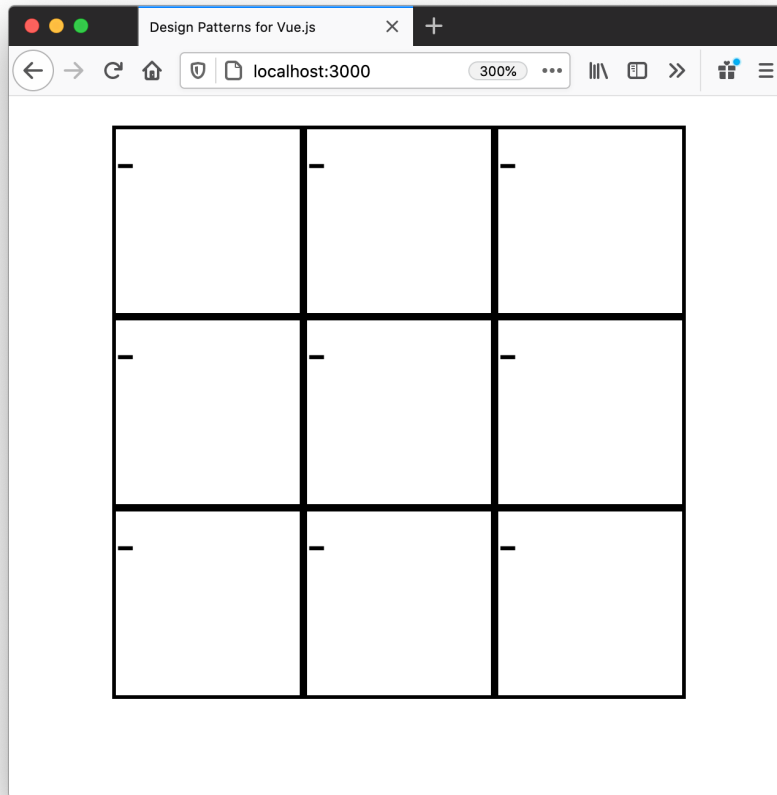


Figure 33: Rendered game board

## 11.2 Computing the Current State

Currently the component is hard coded to use `boards[0]`. What we want to do is use the last element, which is the latest game state. We can use a `computed` property for this. Update the composable:

```
import { ref, readonly, computed } from 'vue'

export function useTicTacToe() {
  const initialBoard = [
    ['-', '-', '-'],
    ['-', '-', '-'],
  ]
```

```

    ['-', '-', '-']
  ]

  const boards = ref([initialBoard])

  return {
    boards: readonly(boards),
    currentBoard: computed(() => boards.value[boards.value.length - 1])
  }
}

```

Getting the latest game state with a computed property.

Update the component to use the new `currentBoard` computed property:

```

<template>
  <div v-for="(row, rowIdx) in currentBoard" class="row">
    <div
      v-for="(col, colIdx) in row"
      class="col"
    >
      {{ col }}
    </div>
  </div>
</template>

<script>
import { useTicTacToe } from './tic-tac-toe.js'

export default {
  setup() {
    const { boards, currentBoard } = useTicTacToe()

    return {
      boards,
      currentBoard
    }
  }
}
</script>

```

Using the `currentBoard` computed property.

Everything is still working correctly. Let's make sure everything continues to work correctly by writing some tests.

## 11.3 Tests

We've written a little too much code without any tests for my liking. Now is a good time to write some, which will reveal some (potential) problems with our design.

```
import { useTicTacToe } from './tic-tac-toe.js'

describe('useTicTacToe', () => {
  it('initializes state to an empty board', () => {
    const initialBoard = [
      ['-', '-', '-'],
      ['-', '-', '-'],
      ['-', '-', '-']
    ]
    const { currentBoard } = useTicTacToe()

    expect(currentBoard.value).toEqual(initialBoard)
  })
})
```

Testing the initial game state.

It passes! Great. As it stands, there is no easy way to pre-set the game state - we currently cannot test a scenario where many moves have been played, without actually playing through the game. This means we need to implement `makeMove` before writing tests to see if the game has been won, since there is no way to update the board as it stands to test winning or losing scenarios. This is not ideal. Instead, let's pass in an initial state to `useTicTacToe`, for example `useTicTacToe(initialState)`.

## 11.4 Setting an Initial State

Update `useTicTacToe` to receive an `initialState` argument to facilitate easier testing:

```
import { ref, readonly, computed } from 'vue'

export function useTicTacToe(initialState) {
  const initialBoard = [
    ['-', '-', '-'],
    ['-', '-', '-'],
    ['-', '-', '-']
  ]

  const boards = ref(initialState || [initialBoard])
```

```

return {
  boards: readonly(boards),
  currentBoard: computed(() => boards.value[boards.value.length - 1])
}
}

```

Accept an initialState to facilitate testing.

Add a test to ensure we can seed an initial state:

```

describe('useTicTacToe', () => {

  it('initializes state to an empty board', () => {
    // ...
  })

  it('supports seeding an initial state', () => {
    const initialState = [
      ['o', 'o', 'o'],
      ['- ', '- ', '- '],
      ['- ', '- ', '- ']
    ]
    const { currentBoard } = useTicTacToe([initialState])

    expect(currentBoard.value).toEqual(initialState)
  })
})

```

A test for initial state.

Notice we pass in [initialState] as an array - we are representing the state as an array to preserve the history. This allows us to seed a fully completed game, which will be useful when writing the logic to see if a player has won.

## 11.5 Making a Move

The final feature we will add is the ability for a player to make a move. We need to keep track of the current player, and then update the board by pushing the next game state into boards. Let's start with a test:

```

describe('makeMove', () => {
  it('updates the board and adds the new state', () => {
    const game = useTicTacToe()
    game.makeMove({ row: 0, col: 0 })

    expect(game.boards.value).toHaveLength(2)
  })
})

```



```

    expect(game.currentPlayer.value).toBe('x')
    expect(game.currentBoard.value).toEqual([
      ['o', '-', '-'],
      ['-', '-', '-'],
      ['-', '-', '-']
    ])
  })
})
})

```

### Testing makeMove.

There isn't anything too surprising here. After making a move, we have two game states (initial and the current one). The current player is now **x** (since **o** goes first). Finally, the **currentBoard** should be updated.

One thing you should look out for is code like this:

```
game.makeMove({ row: 0, col: 0 })
```

When a function is called without returning anything, it usually means it has a side-effect - for example, mutating some global state. In this case, that is exactly what is happening - **makeMove** mutates the global **board** variable. It's considered global because it is not passed into **makeMove** as an argument. This means the function is not pure - there is no way to know the new state of the game after **makeMove** is called without knowing the previous state.

Another thing I'd like to highlight is that we are accessing **.value** three times - **game.boards.value**, **game.currentPlayer.value** and **game.currentBoard.value**. **.value** is part of Vue's reactivity system. Our tests have revealed we've coupled our business logic (the tic tac toe logic) to our UI layer (in this case, Vue). This is not necessarily bad, but it's something you should always be conscious of doing. The next chapter discusses this topic in more depth and suggests an alternative structure to avoid this coupling.

Back to the **makeMove** - now we have a test, let's see the implementation. The implementation is quite simple. We are using **JSON.parse(JSON.stringify())**, which feels pretty dirty - see below to find out why.

```

export function useTicTacToe(initialState) {
  const initialBoard = [
    ['-', '-', '-'],
    ['-', '-', '-'],
    ['-', '-', '-']
  ]

  const boards = ref(initialState || [initialBoard])
  const currentPlayer = ref('o')

  function makeMove({ row, col }) {

```

```

const newBoard = JSON.parse(
  JSON.stringify(boards.value)
)[boards.value.length - 1]
newBoard[row][col] = currentPlayer.value
currentPlayer.value = currentPlayer.value === 'o' ? 'x' : 'o'
boards.value.push(newBoard)
}

return {
  makeMove,
  boards: readonly(boards),
  currentPlayer: readonly(currentPlayer),
  currentBoard: computed(() => boards.value[boards.value.length - 1])
}
}

```

#### Implementing makeMove.

This gets the test to pass. As mentioned above we are using the somewhat dirty `JSON.parse(JSON.stringify(...))` to clone the board and lose reactivity. I want to get *non reactive* copy of the board - just a plain JavaScript array. Somewhat surprisingly, `[...boards.value[boards.value.length - 1]]` does not work - the new object is still reactive and updates when the source array is mutated. This means we are mutating the game history in `boards`! Not ideal.

What you would need to do is this:

```

const newState = [...boards.value[boards.value.length - 1]]
const newRow = [...newState[row]];

```

This works - `newRow` is now a plain, non-reactive JavaScript array. I don't think it's immediately obvious what is going on, however - you need to know Vue and the reactivity system really well to understand why it's necessary. On the other hand, I think the `JSON.parse(JSON.stringify(...))` technique is actually a little more obvious - most developers have seen this classic way to clone an object at some point or another.

You can pick whichever you like best. Let's continue by updating the usage:

```

<template>
  <div v-for="(row, rowIdx) in currentBoard" class="row">
    <div
      v-for="(col, colIdx) in row"
      class="col"
      @click="makeMove({ row: rowIdx, col: colIdx })"
    >
      {{ col }}
    </div>
  </div>

```

```
    </div>
  </div>
</template>

<script>
import { useTicTacToe } from './tic-tac-toe.js'

export default {
  setup() {
    const { boards, currentBoard, makeMove } = useTicTacToe()

    return {
      boards,
      currentBoard,
      makeMove
    }
  }
}
</script>
```

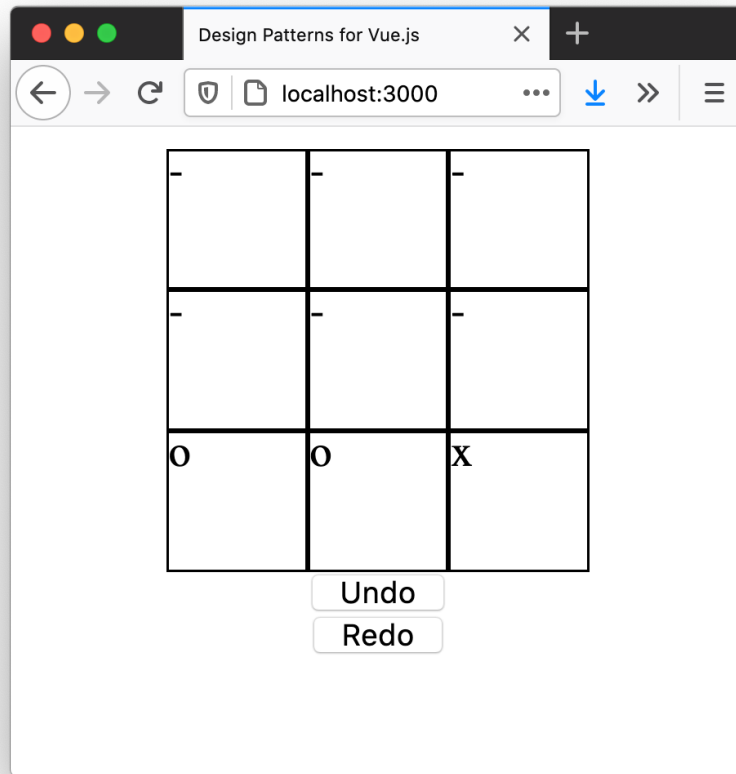


Figure 34: Completed Game

That's it! Everything now works. The game is now playable - well, you can make moves. There are several problems:

1. No way to know if a player has won.
2. You can make an invalid move (for example, going on a square that is already taken).
3. Did not implement undo/redo.

Fixing/implementing these is not very difficult and will be left as an exercise. You can find the solutions in the source code. Undo/redo is probably the most interesting one - you should try and implement this yourself before looking at the solutions.

## 11.6 Conclusion

We saw how you can isolate business logic in a composable, making it testable and reusable. We also discussed some trade-offs of our approach - namely, coupling the business logic to Vue's reactivity system. This concept will be further explored in the next section.

## 11.7 Exercises

1. Write some tests with Testing Library to ensure the UI is working correctly. See the GitHub repository for the solutions.
2. Do not allow moving on a square that is already taken.
3. Add a check after each move to see if a player has won. Display this somewhere in the UI.
4. Implement **undo** and **redo**.

You can find the completed source code in the GitHub repository under examples/composition:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

## 12 Functional Core, Imperative Shell - Immutable Logic, Mutable Vue

You can find the completed source code in the GitHub repository under `examples/composition-functional`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.

---

In the previous chapter, we build a Tic Tac Toe game, encapsulating the logic in a composable. We consciously decided to couple our implementation to Vue, when we used reactivity APIs like `computed` and `ref` in the business logic.

In this chapter, we will explore an paradigm best characterized as “functional core, imperative shell”. We will come back to this name and explain what it means soon.

The goal is to refactor the Tic Tac Toe logic to be more in line with the functional programming paradigm - this means pure functions and no mutation. Since we are avoiding mutation, this mean we will decoupled the logic from Vue’s reactivity system, which relies on mutation and side effects.

Let’s start with `makeMove`, which is full of mutation. In our previous implementation, `makeMove` looks like this:

```
function makeMove({ row, col }) {  
  const newBoard = [...boards.value[boards.value.length - 1]]  
  newBoard[row][col] = currentPlayer.value  
  currentPlayer.value = currentPlayer.value === 'o' ? 'x' : 'o'  
  boards.value.push(newBoard)  
}
```

Original `makeMove` implemented using mutation.

On line 3, we mutation the `newBoard` variable. We then mutate `boards` on line 4, by pushing a new value in. We are also using two global variables: `boards` and `currentPlayer`. If we want to approach this in a functional manner, the function needs include all the required data as arguments, and not rely on global variables. If we rely on global variables, the function will no longer be deterministic (we won’t be able to know the return value without knowing the value of the global variables. This means `makeMove` needs to have the following signature:

```
type Board = string[][]  
type Options {  
  col: number  
  row: number  
  counter: 'o' | 'x'  
}
```

```
function makeMove(board: Board, { col, row, counter }: Options): Board
```

The new `makeMove` will return an updated board based on its arguments.

In other words, `makeMove` needs to receive all required arguments to create a new board, and should return a new board. This makes it pure; the return value is determined exclusively by the inputs.

You may be wondering: if we cannot mutate anything, how do we get anything done? How will we update the UI?

## 12.1 Functional Core, Imperative Shell

The answer is that while we only avoid mutation in the business logic. This is the “functional core” part of the paradigm. All side effects, mutation and unpredictable actions, such as updating the DOM and listening for user input will be handled in a thin layer. This thin layer is the *imperative shell* part of the paradigm. The imperative shell wraps the functional core (the business logic) with Vue’s reactivity APIs. All mutation will occur in the imperative shell.

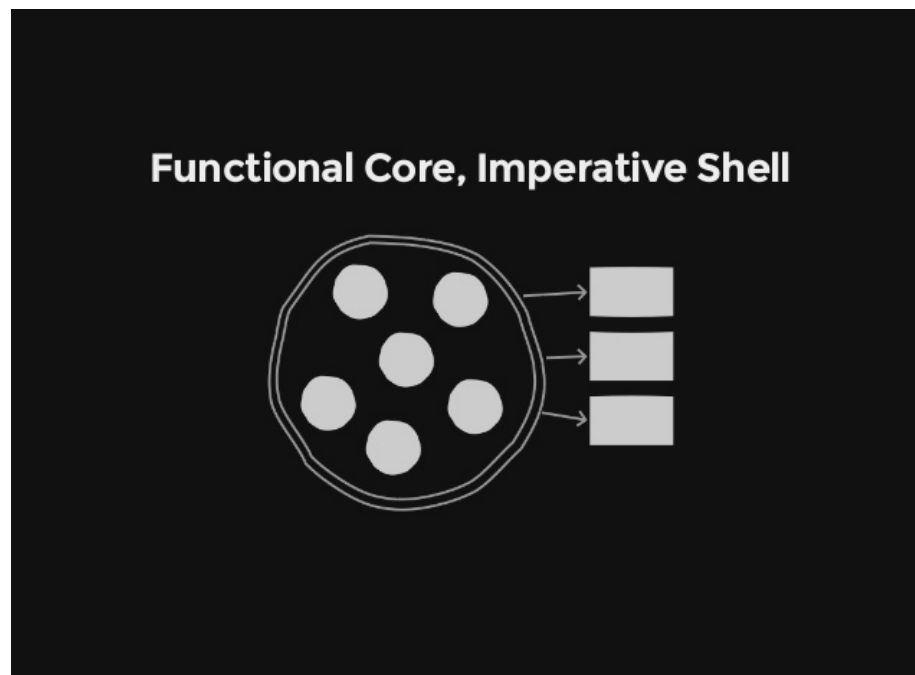


Figure 35: Functional Core, Imperative Shell. Image Credit: mokagio (Twitter)

In this diagram the solid white circles represents the “functional core”. These are a collection of pure functions that are written in plain JavaScript - no reactivity and no global variables. This includes methods like the new `makeMove` function we are about to write.

The thin layer surrounding the solid circles represents the “imperative shell”. In this system, it is the `useTicTacToe` composable - a thin layer written using Vue’s reactivity system, marrying the functional business logic and the UI layer.

The solid rectangles on the right represent interactions between the system and the outside world - things like user input, updating the DOM, the response to a HTTP request to a third party system or a push notification.

By making the business logic mutation free, it’s very easy to test. We will then test the imperative shell, or the Vue integration, using Testing Library - a library designed for this very purpose - to test Vue components. We won’t need too many tests, since all the complexity and edge cases will be covered in the functional core tests.

The final API is going to be the same:

```
import { useTicTacToe } from './tic-tac-toe.js'

export default {
  setup() {
    const { currentBoard, makeMove } = useTicTacToe()

    return {
      currentBoard,
      makeMove
    }
  }
}
```

Final API does not change - only the implementation details.

## 12.2 Business Logic - The Functional Core

Let’s start with the functional core, starting with a `createGame` function:

```
/**
 * Core Logic
 * Framework agnostic
 */
export const initialBoard = [
  ['-', '-', '-'],
  ['-', '-', '-'],
  ['-', '-', '-']
]
```



```

]

export function createGame(initialState) {
  return [...initialState]
}

```

So far, no mutation.

While we could have just done `createGame` without passing any arguments, this makes it easy to seed an initial state for testing. Also, we avoid relying on a global variable.

A test is so trivial it's almost pointless to write, but let's do it anyway:

```

describe('useTicTacToe', () => {
  it('initializes state to an empty board', () => {
    const expected = [
      ['-', '-', '-'],
      ['-', '-', '-'],
      ['-', '-', '-']
    ]
    expect(createGame(initialBoard)).toEqual(expected)
  })
})

```

A simple for the initial game state.

## 12.3 Immutable `makeMove`

Then bulk of the logic is in the `makeMove` function. To update the board, we need the current game state, the column and row to update, and the counter (x or o). So those will be the arguments we pass to the function.

```

export function makeMove(board, { col, row, counter }) {
  // copy current board
  // return copy with updated cell
}

```

The new `makeMove` function (without implementation).

I decided to have two arguments: the first is the `board`, which I consider the “main” argument. I decided to implement `col`, `row` and `counter` as an object, since I consider those to be “options”, which will change depending on the move the player makes.

Before going any further, a test will be useful. I'm going to write a verbose implementation of `makeMove` and then refactor it; the test will help ensure nothing breaks during the refactor.

```

describe('makeMove', () => {
  it('returns a new updated board', () => {
    const board = createGame()
    const updatedBoard = makeMove(board, {
      row: 0,
      col: 0,
      counter: 'o'
    })

    expect(updatedBoard).toEqual([
      ['o', '-', '-'],
      ['- ', '- ', '-'],
      ['- ', '- ', '-']
    ])
  })
})

```

A test to guide us.

Let's start with a verbose implementation. We will use `map` to iterate over each row. For each row, we will `map` each column. If we encounter the row and column the user has chosen, we will update the cell. Otherwise, we just return the current cell.

```

export function makeMove(board, { col, row, counter }) {
  // loop each row with map.
  return board.map((theRow, rowIdx) => {

    // for each row, loop each column with map.
    return theRow.map((cell, colIdx) => {

      // if we are on the row and column the user
      // has chosen, return the counter (o or x).
      if (rowIdx === row && colIdx === col) {
        return counter
      }
      // otherwise just return the current cell.
      return cell
    })
  })
}

```

A verbose and heavily commented `makeMove`.

The test passes! I left some comments to make it clear what's going on. If you haven't seen this type of code before, it can be a little difficult to understand - it was for me. Once I got used to using tools like `map` and `reduce` instead of

a for loop and mutation, I started to find this style of code more concise, and more importantly, less prone to bugs.

We can make this a lot more concise! This is optional; there is some merit to verbose, explicit code too. Let's see the concise version. You can make a decision which one you think is more readable.

```
export function makeMove(board, { col, row, counter }) {
  return board.map((theRow, rowIdx) =>
    theRow.map((cell, colIdx) =>
      rowIdx === row && colIdx === col
        ? counter
        : cell
    )
  )
}
```

Functional code can be very concise. Careful - readability can suffer.

We avoided making a new variable by returning the result of `board.map`. We also removed the `if` statements by using a ternary operator, and the `return` keyword from the `map` functions. The test still passes, so we can be confident the refactor was successfully. I think both implementations are fine; pick the one that you like best.

## 12.4 Vue Integration - Imperative Shell

Most of the business logic is encapsulated in the `createGame()` and `makeMove()` functions. They are stateless. All the values required are received as arguments. We do need some persisted state somewhere, as well as some mutation to update the DOM; that comes in the form of Vue's reactivity - the *imperative shell*.

Let's start with the composable, `useTicTacToe()`, and get something rendering:

```
/**
 * Vue integration layer
 * State here is mutable
 */
export function useTicTacToe() {
  const boards = ref([initialBoard])
  const counter = ref('o')

  const move = ({ col, row }) => {}

  const currentBoard = computed(() => {
    return boards.value[boards.value.length - 1]
  })
}
```

```

    return {
      currentBoard,
      makeMove: move
    }
  }
}

```

The composable integrates the functional core with Vue's reactivity system - the "imperative shell" around the functional core.

I added an empty `move` function, assigning it to `makeMove` in the return value of `useTicTacToe`. We will be implementing that soon.

Let's get something rendering:

```

<template>
  <div v-for="(row, rowIdx) in currentBoard" class="row">
    <div
      v-for="(col, colIdx) in row"
      class="col"
      :data-test="`row-${rowIdx}-col-${colIdx}`"
      @click="makeMove({ row: rowIdx, col: colIdx })"
    >
      {{ col }}
    </div>
  </div>
</template>

<script>
import { useTicTacToe } from './tic-tac-toe.js'

export default {
  setup(props) {
    const { currentBoard, makeMove } = useTicTacToe()

    return {
      currentBoard,
      makeMove
    }
  }
}
</script>

<style>
.row {
  display: flex;
}

```

```
.col {  
  border: 1px solid black;  
  height: 50px;  
  width: 50px;  
}  
</style>
```

Testing out the implementation.

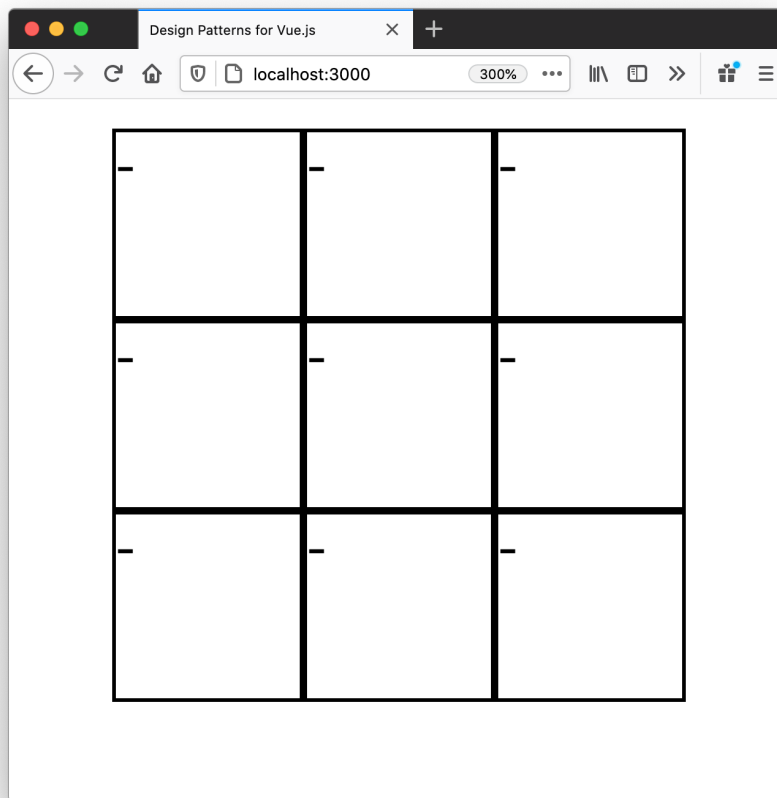


Figure 36: Rendered game board

## 12.5 Integrating makeMove

The last thing we need to do is wrap the functional, stateless `makeMove` function from the functional core. This is easy:

```
const move = ({ col, row }) => {
  const newBoard = makeMove(
    currentBoard.value,
    {
      col,
      row,
      counter: counter.value
    }
  )
  boards.value.push(newBoard)
  counter.value = counter.value === 'o' ? 'x' : 'o'
}
```

`move` is just a wrapper around the functional `makeMove`.

Everything now works in it's functional, loosely coupled, immutable glory.

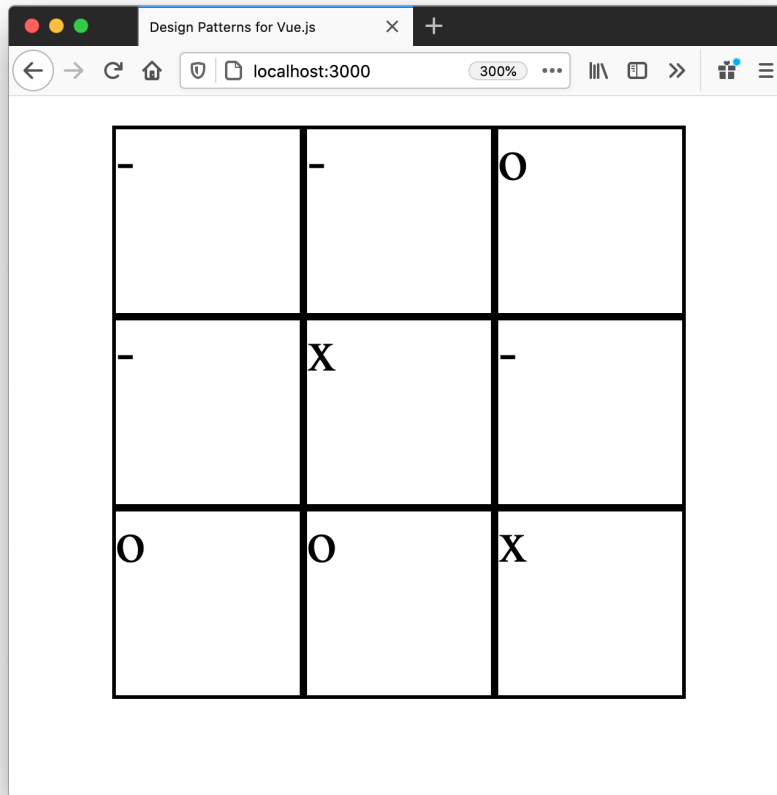


Figure 37: Rendered game board

From a user point of view, nothing has changed, and we can verify this by reusing the UI test (first exercise from the previous section):

```
import { render, fireEvent, screen } from '@testing-library/vue'
import TicTacToeApp from './tic-tac-toe-app.vue'

describe('TicTacToeApp', () => {
  it('plays a game', async () => {
    render(TicTacToeApp)

    await fireEvent.click(screen.getByTestId('row-0-col-0'))
    await fireEvent.click(screen.getByTestId('row-0-col-1'))
```

```

    await fireEvent.click(screen.getByTestId('row-0-col-2'))

    expect(screen.getByTestId('row-0-col-0').textContent).toContain('o')
    expect(screen.getByTestId('row-0-col-1').textContent).toContain('x')
    expect(screen.getByTestId('row-0-col-2').textContent).toContain('o')
  })
})

```

The UI test from previous section, ensuring the behavior has not changed.

## 12.6 Pushing Business Logic into the Functional Core

There is one last improvement we can make. We currently wrap the stateless `makeMove` function:

```

const move = ({ col, row }) => {
  const newBoard = makeMove(
    currentBoard.value,
    {
      col,
      row,
      counter: counter.value
    }
  )
  boards.value.push(newBoard)
  counter.value = counter.value === 'o' ? 'x' : 'o'
}

```

Ideally all the business logic should be in the functional core. This includes changing the counter after each move. I think this is part of the core gameplay - not the UI. For this reason I would like to move `counter.value === 'o' ? 'x' : 'o'` into the functional core.

Update `makeMove` to change the counter after updating the board, and return an object representing the new board as well as the updated counter:

```

export function makeMove(board, { col, row, counter }) {
  const newBoard = board.map((theRow, rowIdx) =>
    // ...
  )
  const newCounter = counter === 'o' ? 'x' : 'o'

  return {
    newBoard,
    newCounter
  }
}

```



```
}
```

Now `makeMove` handles updating the counter, as well as the board. Update `move` to use the new return value:

```
const move = ({ col, row }) => {
  const { newBoard, newCounter } = makeMove(
    currentBoard.value,
    {
      col,
      row,
      counter: counter.value
    }
  )
  boards.value.push(newBoard)
  counter.value = newCounter
}
```

Finally, since we changed the return value, the `makeMove` test needs to be updated (the UI test using Testing Library still passes, since the actual behavior from the user's point of view has not changed):

```
describe('makeMove', () => {
  it('returns a new updated board and counter', () => {
    const board = createGame(initialBoard)
    const { newBoard, newCounter } = makeMove(board, {
      row: 0,
      col: 0,
      counter: 'o'
    })

    expect(newCounter).toBe('x')
    expect(newBoard).toEqual([
      ['o', '-', '-'],
      ['-', '-', '-'],
      ['-', '-', '-']
    ])
  })
})
```

All the tests are now passing. I think this refactor is a good one; we pushed the business logic into the functional core, where it belongs.

## 12.7 Reflections and Philosophy

This section explores concepts that I think separates great developers from everyone else. Separation of concerns is really about understanding what a function should do, and where to draw the lines between the different parts of a system.

There are some easy ways to see if you are separating your Vue UI logic from your business logic, or in a more general sense, your imperative shell from your functional core:

- are you accessing Vue reactivity APIs in your business logic? This usually comes in the form of `.value` for accessing the values of `computed` and `ref`.
- are you relying on global or pre-defined state?

This also prompts another question: what and how should we be testing in our functional core and imperative shell? In the previous section, we tested both in one go - they were so tightly coupled together, so this was the natural way to test them. This worked out fine for that very simple composable, but can quickly become complex. I like to have lots of tests around my business logic. If you write them like we did here - pure functions - they are very easy to test, and the tests run really quickly.

When testing the imperative shell (in this case the Vue UI layer using Testing Library) I like to focus on more high level tests from a user point of view - clicking on buttons and asserting the correct text and DOM elements are rendered. The imperative shell doesn't (and shouldn't) know about how the functional core works - these tests focus on asserting the behavior of the application from the user's perspective.

There is no one true way to write applications. It is also very hard to transition an application from a mutation heavy paradigm to the style discussed in this chapter.. I am more and more of the opinion that coupling Vue's reactivity to your composables and business logic is generally not a good idea - this simple separation makes things a whole lot more easy to reason about, test, and has very little downside (maybe a bit more code, but I don't see this is a big deal).

I think you should extract your logic into a functional core that is immutable and does not rely on shared state. Test this in isolation. Next, you write and test your imperative shell - in this case the `useTicTacToe` composable, in the context of this chapter - a test is using something like Testing Library (or a similar UI testing framework). These tests are not testing business logic as such, but that your integration layer (the composable and Vue's reactivity) is correctly hooked up to your functional core.

## 12.8 Exercises

Repeat the exercises from the last chapter - undo/redo, defensive checks to prevent illegal moves, check if a player has won the game and display it on the UI.

You can find the completed source code in the GitHub repository under `examples/composition-functional`:

<https://github.com/lmiller1990/design-patterns-for-vuejs-source-code>.