Markus Oberlehner

# Writing Good Tests for Vue Applications

Craft tests that enable you to
build better applications faster

# Contents

## MASTERING COMPONENT TESTING

...

## UNIT TESTING FUNDAMENTALS

...

## TESTING IN THE AGILE WORLD

...

## DEEP DIVE INTO TEST-DRIVEN DEVELOPMENT

...

# Preface

Over the past few years of my career, I spent countless hours dealing with testing. I wrote a lot of tests and thought a lot about how to write *good* tests. But unfortunately, I also wrote a lot of *bad* tests and made many mistakes typical for people new to the art of testing.

This book is for **Vue developers** who want to up their testing game and write *good* tests for Vue applications. Maybe, like me, you already wrote your fair share of *bad* tests and wonder how to reach the promised land of fast feedback loops, rapid release cycles, and refactoring with confidence testing advocates keep talking about. Over the next couple of chapters, I'll teach you how.

But if your primary goal is to learn just the basics of how to use `vue-test-utils` and Jest to write unit tests for Vue components, this is not the perfect book for you. Whereas if, instead, you want to know the secrets of how to write tests that enable **rapid feature development** and **refactoring with confidence** while **introducing as few bugs as possible,** look no further!

The book on your screen is not mainly about learning to use specific frameworks and libraries. Quite the opposite! There is a separate chapter on decoupling tests from a particular test framework. Instead of going into the details of how to use this or that tool, this book is about principles and best practices that enable us to write highly valuable and maintainable tests. That said, we will also examine the pros and cons of popular frameworks and which to choose. And, spoiler alert, we will settle on **Vitest** and **Playwright** as test frameworks. In addition, we will use the **Testing Library** to decouple tests from implementation details and the **Mock Service Worker** library to mock API requests.

Although this book primarily aims at Vue developers (we use Vue.js in all examples of components and application code), the basic principles apply to all kinds of web applications, no matter what framework we use or if it's a multi-page application (MPA) or single-page application (SPA). So the knowledge in this book is a solid foundation for your future software developer career, even if your path leads you to a different technology stack.

While I'm confident that the tips and principles in this book will help you avoid the mistakes I've made in the past, there's always room for improvement, and what works for me might not work for you. Additionally, new tools open up new opportunities and challenge established best practices. Over time, new insights can radically change how and what we test. Still, I am certain: after reading this book, you will write better tests and, by extension, better code!

> Testing can only prove the presence of bugs, not their absence.
>
> — Edsger W. Dijkstra

## Standing on the Shoulders of Giants

Throughout my 15-year journey in programming, I've been fortunate to have access to countless blog articles, talks and videos, and StackOverflow threads (isn't the internet amazing?). The collective wisdom of this community has been instrumental in shaping my knowledge and skills. While it's impossible to thank everyone individually, I'd like to acknowledge a few individuals who have profoundly influenced my approach to testing.

Firstly, **Anthony Fu,** the creator of Vitest, deserves special mention. Vitest quickly became my favorite test runner. It has

made testing more efficient and enjoyable for me.

Next, I want to express my gratitude to **Kent C. Dodds.** As the creator of the Testing Library and author of numerous insightful blog articles about testing practices, Kent's contributions to the field are nothing short of remarkable.

I must also express my admiration for **Debbie O'Brien.** Her passion and enthusiasm for testing are genuinely infectious. Debbie's dedication to demystifying testing and making it accessible to all developers, regardless of their experience level, is inspirational.

Lastly, I'd like to thank **Dave Farley.** His insightful videos on the Continuous Delivery YouTube channel have been a constant source of inspiration. Dave's teachings on effective development practices have influenced many of the concepts discussed in this book.

As I already said, there is no way I can mention everybody. Still, I also want to name **Lachlan Miller, Mark Noonan,** and **Jessica Sachs,** whom I'm incredibly thankful for, for their contributions to the testing space.

Standing on the shoulders of these and so many more giants, I've seen further and understand deeper. I hope this book helps you in your journey and perhaps, one day, inspires you to become a giant on whose shoulders others may stand.

## Setting the Stage

This book starts with a good deal of theory. Why, you ask? Because I believe in giving you a solid foundation you can build upon. The principles and concepts I introduce will equip you with the knowledge to devise your own solutions. After all, every project is unique, and what works in one situation may not work in another.

Although we'll learn a lot about principles and testing theory, I'll provide plenty of practical examples and techniques for writing tests per these principles throughout the book. But remember, these are just examples. There are countless ways to write tests that adhere to the principles we'll discuss. The techniques and methods I present are just one of the paths to effective testing. My goal is to provide you with the knowledge to navigate the often murky testing waters, regardless of the tools you use or the specific challenges you face.

And let's be honest: sometimes, you might not adhere as strictly to these principles as I describe in this book. And that's okay! The real world of software development is a complex, ever-changing landscape. There will be times when you need to adapt and adjust based on your circumstances and goals.

Remember, the ultimate aim here is not to follow a rigid set of rules but to understand the principles that underpin effective testing. Once you grasp these, you can apply them to your unique context and challenges.

In the final chapter, we'll roll our sleeves and dive deep into Test-Driven Development (TDD). We'll build a real-world application using TDD and all the knowledge you've soaked up from the previous chapters. It's where theory meets practice; trust me, it will be fun!

I want to write the book I wish I had in my hands when I started my journey into the world of TDD many years ago. Let's do this!

# Breakdown of Testing Approaches

As we embark on this journey to improve our testing skills and enhance the quality of our Vue applications, it's crucial, to begin with a solid understanding of the various testing approaches available. This chapter serves as a foundation for building our test strategy in the next chapter, ensuring that we have the tools and techniques to create effective, maintainable, and reliable tests.

We will explore different testing approaches, from manual testing to automated End-to-End (E2E) System, Application, Component, and Unit Tests. Each technique has unique advantages and limitations, and understanding their nuances will enable us to select the most suitable methods for our specific project requirements. Furthermore, by combining these approaches, we can create a comprehensive testing strategy that ensures our applications are robust, resilient, and ready to face the challenges of the real world.

As we progress through this chapter, remember that the goal is not to advocate for one testing approach over another but to provide the knowledge and insights to make informed decisions about which methods best serve our needs. By the end of this chapter, you will clearly understand the various testing approaches and will be well-equipped to craft a tailored testing strategy for your application.

## The Role of Manual Testing in TDD

As we delve into the various testing approaches, let's first examine the role of manual testing in Test-Driven Development. While

TDD primarily focuses on automated testing, manual testing still has a place in our overall testing strategy. It can be instrumental in uncovering errors, especially those that occur in specific edge cases and can complement automated tests by focusing on an application's user experience and usability.

By understanding the role and value of manual testing in TDD, we will be better equipped to make informed decisions about when and how to incorporate it into our Vue application testing strategy. In addition, this knowledge will enable you to create a well-rounded testing approach that ensures not only the functionality of your application but also its usability and overall user experience.

## The Value of Manual Testing

Systematic manual testing can lead to excellent results in preventing the deployment of errors that only occur in specific edge cases. Quality Assurance (QA) professionals excel at finding various types of errors, including those that often only happen in particularly exceptional circumstances. But manual testing isn't just about bug hunting. It's also an invaluable tool for assessing the quality of the user experience. It allows us to answer not just the question of 'does it work?' but also 'does it work well?' So, manual testing serves a dual purpose: finding bugs that might remain undiscovered for a long time and ensuring the overall quality of the user experience.

## Testing on Production!?

Manual testing can take place directly on the production system or, ideally, on a near-perfect copy of the production system. Testing on a reproduction of the production system, even before

changes are published, is preferable to testing directly on the live system if stakes are high and the cost of deploying buggy code can be fatal. However, remember that setting up and operating a staging environment as close to the production system as possible further increases the already high costs incurred by the additional personnel.

Ideally, you don't only have a single staging environment, but you can quickly spin up new environments as needed. But each environment must be 100% independent, which means that changes (to the data) in one environment don't affect other environments in any way.

## Avoid Striving for Perfection

A common question arises: how close should our testing approach be to reality? Should we perform manual tests on the production system? Or should our staging system mirror the production system as accurately as possible? Is it necessary for our test framework to control a real browser or even be capable of simulating all browsers and devices we want to support? Striving for extreme accuracy in testing can be very time-consuming, resource-intensive, and expensive.

For the maximum possible amount of confidence, we have to use the production system itself and observe a representative set of users. In this perfect world scenario, the users do not know we are keeping taps on them. In practice, however, we must respect the privacy of our users.

Although we can use this approach to find errors, typically, however, we mostly go this route to uncover usability issues. For this purpose, such tests can be instrumental and thus justify the high costs in many cases. However, because of the significant effort required, this type of testing is unsuitable as a systematic

approach to finding errors in our system. Therefore, user tests will no longer play a significant role in the rest of this book.

## Manual Test Plans and Exploratory Testing

The exact procedure for manual testing can vary greatly. For example, we can take a very structured approach, with meticulously worked-out test plans that we process step by step. Or, we try to provoke errors that can occur with improper or unforeseen usage patterns. While executing fixed test plans often can (and should) be automated, fully automating exploratory testing is much more challenging. Manual, exploratory testing, therefore, represents a valuable building block in a successful testing strategy.

When we use manual testing in conjunction with automated tests, our primary goal is not to prevent regressions–automated tests are better suited to clarify *whether* our application works–instead, with manual testing, we can focus on checking *how well* the application performs its tasks. Machines cannot answer this question. At least not yet. On the other hand, we humans are well suited to empathize with other people and thus find out whether interacting with our application is intuitive. And whether the application can cope with users using it differently than we planned. In contrast to automated testing, manual testing is costly. Therefore, we should try to get the most out of it when we choose this approach.

Remember that this type of testing is not only expensive because of the high personnel effort but also because developers receive feedback in their daily work very slowly. Feedback loops should be as short as possible. A few milliseconds are perfect, seconds good, minutes may be acceptable in some cases, and anything beyond hours is unsuitable as a tool to support the

development process. **From the development cycle perspective, manual testing can only ever supplement automated forms of testing.**

## Manual Testing in High-Stakes Industries

However, we can consider manual testing as our primary testing strategy if we have a lot of money available or work in a business sector with low error tolerance (e.g., aviation, healthcare, or finance). Manual testing can help us to discover errors that we would otherwise only find when it is too late. In most cases, however, I recommend using this type of testing only in addition to the following strategies. And even if we decide to rely heavily on manual testing, it's not an excuse for doing less automated testing. Quite the opposite! **In high-stakes industries, automated testing is even more essential.**

## Balancing Manual and Automated Testing

Finding the right balance between manual and automated testing is crucial for optimizing the effectiveness of our testing strategy. While manual testing can provide valuable insights into usability and user experience, automated testing should be the foundation of our approach, ensuring our application's functionality, performance, and reliability.

By implementing a manual testing regime that is well-dosed, does not block deployments (no gatekeeping!), and focuses on finding issues with the usability of our application instead of detecting regressions, we can create a well-rounded and efficient testing strategy that ensures the quality of your applications while minimizing costs and maximizing the value of our testing efforts.

## Four Types of Automated Tests

Automated tests are the gold standard for ensuring that applications and whole systems perform as expected. By automating tests, we save a significant amount of manual work, and once we have written the tests, we can run them as often as we want, almost for free. **Every test we want to perform regularly and *can* be automated *should* be automated.** This reduces the need for manual testing, and we can use the freed-up resources to build new features or find errors that automated tests cannot detect.

In the realm of automated testing, we often encounter a plethora of terms for various test types: E2E tests, acceptance tests, integration tests, unit tests, and many more. Since there is no official authority to define and standardize these terms, it can be challenging to establish clear distinctions. In this book, we will differentiate between the following four types of automated tests, providing a consistent framework for understanding their unique purposes and characteristics:

- **E2E System Tests:** These tests aim to evaluate a complete system, including its entire infrastructure (data persistence, third-party services, etc.). They provide a comprehensive assessment of how well the system functions as a whole, ensuring that all pieces work together seamlessly.

- **Application Tests:** Focused on verifying the fulfillment of acceptance criteria from the user's perspective, Application Tests examine both the user interface and the application's business logic. Opposed to E2E System Tests, these tests isolate the application from other system components, using mocks to replace external dependencies and allowing for a more targeted assessment of the application's functionality.
- **Component Tests:** These tests concentrate on individual Vue components within the application, ensuring they function correctly. By isolating components and testing them independently, Component Tests help surface issues quickly.
- **Unit Tests:** Designed to validate the functionality of smaller units within the application, Unit Tests evaluate the correct working of specific elements from the perspective of a developer using a piece of code (unit) in their own code.

## E2E System Tests

The way I define E2E System Tests is that we test a whole system without mocking any of its parts. While we use mocks in other forms of automated testing to run tests independently of external services, we avoid them in E2E testing altogether. As a result, E2E System Tests provide a high degree of confidence by testing the entire system *end-to-end*, including the surrounding infrastructure, allowing us to identify specific kinds of errors which otherwise would go undetected.

```
// Test with no mocking whatsoever
it('should be possible to buy a bike', async ({ page }) => {
  await page.goto('https://my.bikestore');

  await page.findByText('Best Bike Ever').click();
  await page.findByRole('button', { name: 'Add to cart' }).click();
  await page.findByRole('button', { name: 'Checkout' }).click();

  // ...

  const successMessage = page.findByText('Thank you for your order!');
  expect(await successMessage.isVisible()).toBe(true);
});
```

However, even having E2E System Tests in place does not mean our system is error-free! Even the best and most elaborate tests can only prove that there *are* errors, not that there are *no* errors. Therefore, if we discover a bug through manual testing or because our users point it out, we should add a new test as part of the bug-fixing process to ensure the error does not recur. Remember that the last part is also true for the following types of tests, not only E2E System Tests!

## Challenges and Costs of E2E System Tests

Automated E2E System Tests provide high confidence but have associated challenges and costs. To ensure high confidence, we must create a test environment that closely resembles the production environment. Establishing and maintaining such a replica can be challenging, especially when considering the need

to prevent interference between test scenarios. We need the flexibility to run our tests in any order and even in parallel. The thoughtful design of our test infrastructure is crucial to achieving a non-interfering test environment. Succumbing to the temptation of taking shortcuts may lead to long-term issues, such as false positives and flaky tests.

We also must consider that running one or multiple exact copies of the production system necessitates considerable resources. Therefore, a classic problem in the real world is that many companies resort to using underpowered hardware for their testing and staging environments to cut costs. Consequently, E2E test suites may take several hours to complete, undermining the goal of leveraging testing for fast feedback loops. Ideally, tests should be stable and swift, as unstable tests with lengthy runtimes often prove too costly due to their moderate benefit for developers and high maintenance requirements.

## Technical Implementation of E2E System Tests

Tailoring the architecture and infrastructure of our testing environment to the requirements of our system is crucial for running E2E System Tests independently. This process involves configuring the system to allow for isolated, parallel execution of tests and ensuring the necessary resources and services are available for each test scenario.

Consider a microservices architecture: to set up an effective E2E test suite, we must be able to bring each microservice into a particular state. E.g., if we want to test if a web shop user can cancel an order, at the very least, we have to set up a fresh user and assign a pending order to it. Designing and maintaining such an environment can be a complex task, deserving of a book of its own. However, investing in a well-structured architecture and

infrastructure will ultimately enhance the efficiency, reliability, and value of your E2E System Tests.

The technical implementation of E2E System Tests varies based on the system under test and the technologies employed. As a result, I will not delve into the details of writing E2E System Tests in this book. Nonetheless, many principles and techniques discussed in the chapter on Application Tests also apply to E2E System Tests.

## Balancing High Confidence and Costs

E2E System Tests offer a high level of confidence at a moderate cost. However, given the complexity and time-consuming nature of implementing and maintaining E2E System Tests, it is crucial to carefully weigh the pros against the many cons to decide which parts of the system warrant E2E testing. A possible approach to significantly reduce the run times of E2E System Tests is to rely on so-called smoke tests.

Two factors make E2E System Tests particularly expensive: the effort required to set up and continuously maintain an environment for conducting E2E System Tests and the comparatively long run times. One way to deal with these challenges is to adopt a smoke testing approach.

Smoke tests focus on the most critical and delicate parts of the system, particularly those areas where integration with other system components is essential. The role of smoke tests is to quickly alert us about fundamental problems with our software through a few select tests rather than achieving very high test coverage. By smoke testing our entire system, we can promptly uncover issues in the most crucial areas of our application.

We strictly focus on the core functions of our system, such as adding a product to the shopping cart or going through the

checkout process in an e-commerce application. In those cases, we are interested in whether the integration with the payment provider functions correctly or whether our system sends the order to the correct order-tracking application and not so much in verifying every tiny detail of the process.

Using smoke tests to integrate E2E System Testing into our overall testing strategy allows us to focus on testing those functionalities whose failure would result in significant financial loss and where there are delicate dependencies with other parts of the system. This approach, combined with other testing methodologies, is an excellent way to keep the effort required for E2E System Tests in check while benefiting from their advantages, such as ensuring that the individual components of our system communicate correctly with each other.

## Monitoring and Reporting

Monitoring and reporting the results of E2E System Tests are crucial for maintaining visibility into the application's performance and reliability. Addressing any issues timely can help the team stay proactive in identifying and resolving potential problems before they escalate. **Remember: whenever our pipeline fails, it becomes the top priority of the whole team to fix it.** Clear and concise reporting helps stakeholders stay informed about the application's status and whether it is in a deployable state.

## Continuous Improvement

Reviewing and improving the E2E System Testing process is essential as applications evolve and grow. Regularly updating test cases to cover new features, addressing any false positives or flaky

tests, and refining the testing strategy to focus on the most critical system components will help maintain a high level of confidence in the application's performance and reliability.

Automated E2E System Tests can provide significant value in ensuring the reliability of our system, but they come at a cost. However, we can achieve high confidence while keeping the costs in check by addressing the challenges associated with E2E System Testing by incorporating smoke tests.

## Application Tests

The term *Application Test*, as defined in this book, is not universally accepted. The word *integration tests* is often used for this type of test, while other resources use the term *E2E Test* to describe tests with a similar purpose. Even the phrase *Application Test* can be found with varying meanings online. However, for clarity, this book will use the term *Application Test* and differentiate between *E2E System Tests* and *Application Tests*.

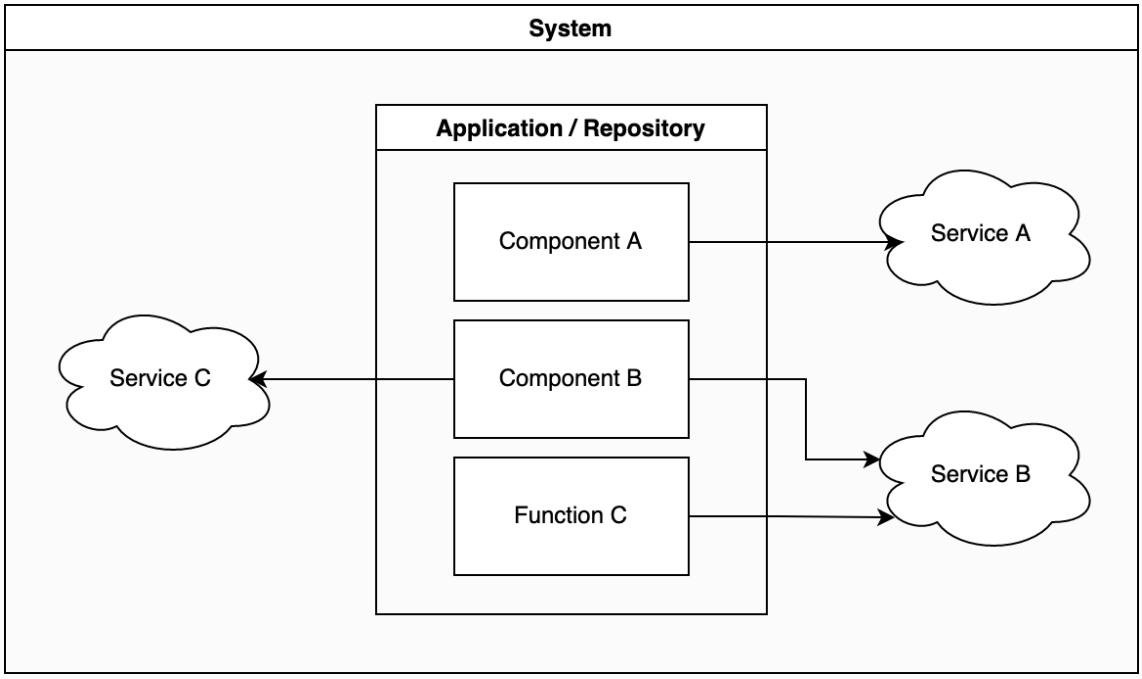
You might disagree with the names I decided to use. And that's ok. Consider adopting a naming scheme that fits your or your team's preferences. But one thing is important: Due to the ambiguity of the terminology, everyone within a team or a company must have a shared understanding of what you call each test type.

```
// Application Test mocking an external service
it('should be possible to buy a bike', async ({ page }) => {
  await page.route('/api/product/list', async route => {
    const json = [
      bestBikeEver,
      notSoGoodBike,
      badBike,
    ];
    await route.fulfill({ json });
  });
  await page.route('/api/checkout', async route => {
    const json = { success: true };
    await route.fulfill({ json });
  });
  await page.goto('https://my.bikestore');

  await page.findByText('Best Bike Ever').click();
  await page.findByRole('button', { name: 'Add to cart' }).click();
  await page.findByRole('button', { name: 'Checkout' }).click();

  // ...

  const successMessage = page.findByText('Thank you for your order!');
  expect(await successMessage.isVisible()).toBe(true);
});
```

## The Purpose of Application Tests

With Application Tests, we do not aim to test the whole system with all its services and components and ensure that their

integration works as expected; that is the purpose of E2E System Tests or other means, like contract tests. Instead, the primary goal of Application Tests is to verify whether our *application* behaves according to the users' expectations based on predefined acceptance criteria. Of course, ensuring this with absolute certainty requires validating the integration of all parts of our system. Yet with Application Tests, we try to strike a balance between accuracy and efficiency, favoring fast feedback loops over absolute certainty.




A critical aspect of my definition of *Application Tests* is that they are designed to test the acceptance criteria of a **specific application** (part of a more extensive system). I define an *application* as everything within a single repository. And to be even more specific, in this book, we focus on testing Vue.js-based Single Page Applications (SPAs). Users interact with our system through the Vue application, meaning that all acceptance criteria, formulated from the users' perspective, go through the Vue SPA first. In contrast, parts of the system outside our Vue SPA

(database, backend services, etc.) are invisible to our users and, therefore, safe to ignore for this type of testing.

In Application Tests, we presume all other modules of our system function as expected, which is a fair assumption considering that we may not even have access to the code of all the services our application communicates with. When assuming that all modules of the system we rely on work as expected, we can safely replace every external module with mocks as long as we ensure we adhere to the contract provided by those external services.

## Application Tests vs. E2E System Tests

While E2E System Tests evaluate the entire system, including all its individual pieces like API services and databases, the primary goal of automated Application Tests is to ensure that one particular part, the application that our users directly interact with, functions correctly. We test the behavior of **our application** and not the implementation of the system as a whole. So, for example, whether data is fetched from or sent to a particular service is irrelevant to us in the context of Application Testing.

The type of application we test determines which factors we consider relevant for evaluating whether we meet the acceptance criteria. For instance, the exact data persistence method is not our concern when testing a web application's user interface. However, verifying whether data is stored correctly is crucial when testing an API service.

The primary distinction between Application Tests and E2E System Tests lies in the clear boundary to other system parts, with tests focusing on a single application. Therefore, we ignore or *mock* all aspects of the *system* that are not part of our *application*.

Although omitting large parts of our system in our tests may make some people nervous and question the validity of our tests, the advantages of this compromise generally outweigh the disadvantages. Comprehensive E2E System Tests offer *diminishing returns* over comparatively simpler and faster Application Tests. The minimal confidence gain they provide is usually not worth the additional cost.

### Advantages of Application Tests

Application Tests are fast, stable, and easy to implement when done correctly. As a result, they are generally more cost-effective due to less maintenance and shorter feedback loops, positively impacting the team's velocity. Of course, automated Application Tests don't guarantee that our entire system, including all services and databases, will work. However, they offer a high confidence level at a comparatively low price.

## Component Tests

Before diving into the details, let's clarify what I mean by Component Tests. In the context of this book, when we refer to Component Tests, we're talking about tests that focus on a single Vue.js component. Components are the building blocks of our application. Each one has a specific job and must perform it flawlessly. Component Tests allow us to ensure this by testing the functionality of each component in isolation.

The most significant benefit of Component Tests over Application Tests is their speed. Creating Component Tests is a quick process, and they provide rapid feedback. By following certain best practices, maintenance is also straightforward

compared to more extensive ways of testing. However, Component Tests can not tell us whether our whole application functions as expected. As the name suggests, Component Tests evaluate the functionality of a single component, so only a tiny part of our application, in isolation.

```
// In this component test we only test a single
// `PostForm` component in isolation.
it('should inform the user when invalid data is entered', async () => {
  render(PostForm);

  const inputField = await screen.findByLabel('Title');
  await userEvent.type(inputField, '');
  await userEvent.click(screen.getByRole('button', { name: 'Save' }));

  expect(await screen.findByText('Please enter a valid
title')).toBeInTheDocument();
});
```

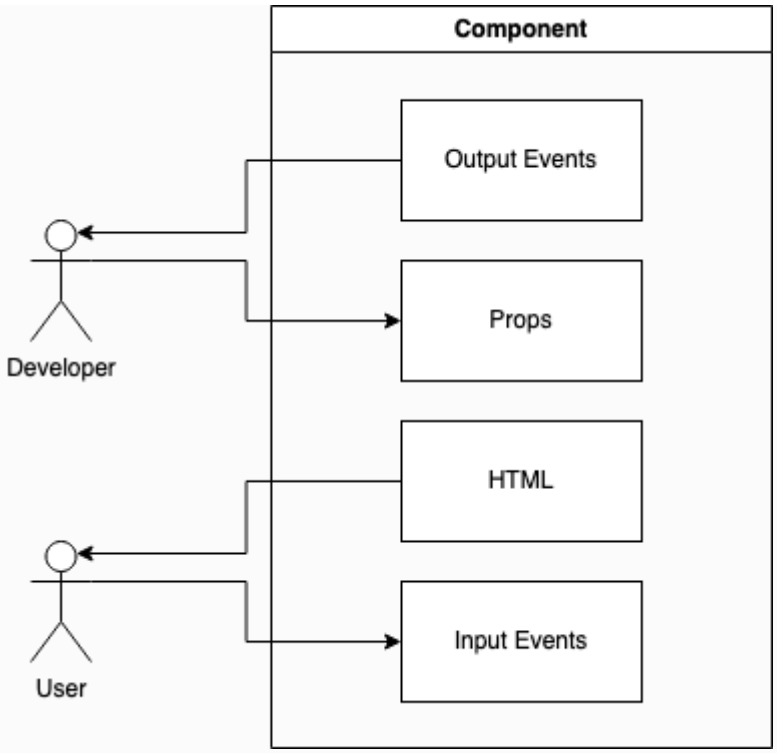
## Test Frameworks and Test Types

It's common to associate certain tools with specific types of tests. For example, people often identify Playwright with E2E and Application Tests due to its capabilities for simulating user interactions in a real browser environment. On the other hand, we frequently see Jest and Vitest as the tools of choice for Unit and Component Tests. However, all the tools above can execute Application and Component Tests. The choice depends on your specific needs and the trade-offs that best suit your circumstances.

The crucial point here is that the type of test—be it Component or Application—is not determined by the tool you use but by what and how you test. You can run Component Tests using Jest or Vitest with `vue-test-utils`, just as you can conduct Application or E2E Tests with Playwright or Cypress. These tools are versatile enough to test both individual components and complete applications.

## Two Perspectives: User And Developer

We use Component Tests to test a specific Vue component in isolation. While with E2E System Tests and Application Tests, we write all of our tests from the perspective of a user, when it comes to Component Tests, we have to consider two perspectives: again, the view of a *user* but also how a *developer* interacts with a component when using it to build a new feature. Taking a developer's perspective is what we do with Unit Tests, too. So the main differentiator between Unit and Component Tests is that we must also consider a user's perspective with the latter in addition to the developer's view. So always consider the two points of view when working on Vue.js Component Test.




## What If Writing Tests Feels Like a Chore?

Sometimes, writing Component Tests might feel like a chore. If this is the case, we probably write code that is hard to test, often indicating that our code could be better quality. However, the more we adhere to best practices when writing code, the easier it will be to write *good* Component Tests. For example, pure functions (and components) without side effects are much easier to test than functions that trigger side effects (e.g., fetching data from or sending data to an API endpoint) or depend on globals. I, therefore, advise against registering components globally or using Vue plug-ins excessively and avoiding fetching data at every level of your application.

Furthermore, most developers who have difficulty writing new and maintaining tests tend to write tests after finishing the code of a new component. On the other hand, if we flip things around and practice TDD, writing the test before the actual

implementation, we will find that writing code that is easy to test comes naturally.

### Benefits of Component Tests

Component Tests can be written swiftly, have low maintenance overhead, and provide feedback almost instantaneously on whether individual components of our application work correctly in isolation. However, they can't tell us if we've wired all our Components correctly. Consequently, the confidence we gain from Component Tests regarding the functionality of our application as a whole is limited. Nevertheless, Component Tests are a valuable building block in our overall testing strategy because they provide speedy feedback at minimal costs.

## Unit Tests

Unit Tests play a vital role in ensuring the quality and reliability of an application. In contrast to Component Tests, which focus on testing whole Vue components, Unit Tests target the most atomic pieces of our application, such as modules, classes, and functions. As a result, we can quickly identify issues and potential improvements within our codebase by testing these elements in isolation with Unit Tests.

```
it('should correctly add two numbers', () => {
  expect(add(1, 2)).toBe(3);
});
```

Unit Tests are essential for several reasons:

1. They provide almost instantaneous feedback, allowing us to identify and fix issues while working on the code.
2. Unit Tests can serve as living documentation, illustrating how to use individual functions, classes, or modules in our code.
3. Well-written Unit Tests can improve the overall maintainability of our codebase by encouraging the use of best practices and modular, thus testable code.

## Writing Effective Unit Tests

When writing Unit Tests, it's crucial to keep the following principles in mind:

1. **Test in isolation:** Unit Tests should focus on a single function, class, or module, ensuring that each piece of our application does exactly what it's supposed to do. This approach allows us to pinpoint issues quickly and accurately.
2. **Keep tests simple:** Unit Tests should be easy to read, understand, and maintain. We must avoid complex test setups and test a single behavior or functionality per test.
3. **Use appropriate test data:** Choose test data representative of real-world scenarios and edge cases, ensuring that our tests cover various possible inputs and outcomes.

4. **Write tests from a developer's perspective:** Unlike Component Tests, which also consider the user's perspective, we write Unit Tests primarily from the developer's point of view. This approach helps ensure that our tests accurately reflect how we use a piece of code in practice.

5. **Practice TDD:** Writing Unit Tests *before* implementing the actual code leads to more testable and maintainable code. In addition, TDD encourages using best practices and modular design, making writing *good* Unit Tests easier.

## The Relationship Between Unit Tests and Component Tests

As mentioned in the previous chapter, Component Tests focus on testing individual Vue components from *both* the user's *and* the developer's perspectives. While there is some conceptual overlap between Component Tests and Unit Tests, considering two perspectives clearly distinguishes those two types of tests.

Furthermore, Unit Tests are even more granular, targeting specific functions, classes, or modules in isolation. In contrast, Component Tests evaluate the functionality of a Vue component as a whole. Both tests are crucial for ensuring the quality and reliability of our application, but they serve different roles within our overall testing strategy.

## The Value of Unit Tests

Unit Tests provide several benefits to your application development process. First, they offer rapid feedback on the functionality of individual code elements, helping us catch issues early and maintain a high-quality codebase. Additionally, Unit

Tests can serve as a form of documentation, guiding developers on how to use and interact with various parts of our application.

While Unit Tests cannot provide complete confidence in the functionality of our application as a whole, they are a valuable building block in our overall testing strategy. By combining Unit Tests with other testing types, such as Component Tests and Application Tests, we can create a comprehensive and effective testing strategy that ensures the reliability and maintainability of our application in the long run.

## Summary

This chapter taught us about various testing approaches, including manual testing, automated E2E System, Application, Component, and Unit Tests. Each testing method has unique advantages and limitations, and understanding these nuances helps select the most suitable techniques for specific project requirements.

### Key Learnings

1. Exploratory, manual testing has its place in TDD, uncovering edge cases and assessing the user experience.
2. Balancing manual and automated testing with a focus on automation is crucial for optimizing the effectiveness of a testing strategy.
3. E2E System Tests comprehensively assess the whole system, including its infrastructure.

4. Application Tests focus on verifying acceptance criteria from the user's perspective, examining the user interface and the application's business logic.
5. Component Tests concentrate on individual Vue components within the application, ensuring they function correctly from the perspective of users and developers.
6. Unit Tests validate the functionality of smaller units within the application, evaluating specific elements from a developer's perspective.
7. We can use tools like Cypress, Playwright, Vitest, and Jest for both, Application and Component Tests–what and how we test is the main differentiator, not the tool we use.
8. Component and Unit Tests provide almost instantaneous feedback and can help us pinpoint particular errors, while other forms of testing provide less precise feedback.