Markus Oberlehner

Writing Good Tests for Vue Applications

Level up your Vue testing skills and build better applications faster

Contents

Preface

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

Over the past few years of my career, I spent countless hours dealing with testing. I wrote numerous tests and devoted tons of time thinking about how to write *good* tests. However, unfortunately, I also wrote a great deal of *bad* tests and made many mistakes typical for people new to the art of testing.

I wrote this book for **Vue developers** like me who want to up their testing game and write *good* tests for Vue applications. Maybe you have already written your fair share of *bad* tests and wonder how to get to the promised land of **fast feedback loops, rapid release cycles,** and **refactoring with confidence** testing advocates keep talking about. Over the following chapters, we'll explore how to reach this magical place.

This book is not mainly about learning to use specific frameworks and libraries. Quite the opposite! There is even a dedicated chapter on decoupling tests from particular test frameworks.

If your primary goal is to learn just the basics of using tools like `vue-test-utils` and Jest to write Unit Tests for Vue components, this is not the perfect book for you. Yet, if you want to learn how to build better applications faster, look no further!

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** package 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, after reading this book, you will write better tests and code and ultimately build better products faster.

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 thinking about coding in general and my approach to testing in particular.

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. And still, Vitest is only a tiny piece of Anthony's contributions to the Nuxt, Vue, and Web development community.

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.

Furthermore, I also want to 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.

Additionally, I'd like to thank **Dave Farley.** His insightful videos on the Continuous Delivery YouTube channel offer profound insights. 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,** for whom I'm incredibly thankful 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.

One Perspective Among Many

While researching for this book, reading other books, blog articles, and tweets, and watching countless videos, I encountered a wealth of material supporting my thoughts about testing. Some viewpoints, though slightly divergent from mine, still sparked inspiration.

However, many brilliant individuals also voice contrasting perspectives, often contradicting my views. After repeatedly discovering contradictory information, it quickly became clear to me: I can't write the *definitive* book on testing. This book can't provide a one-size-fits-all solution or dictate practices that will magically align seamlessly with your specific needs. No single book can.

I base everything I write on *my* experiences and personal opinions. What else can I do? On the one hand, this means I base it on the experiences of somebody who has written code almost daily for the past 15 years. On the other hand, it also means that I ground my writing on the opinions of a human being who can't know it all and has barely scratched the surface of what there is to explore in the vast world of software.

This text primarily addresses testing standalone SPAs that fetch their data from one or more independently developed APIs or microservices. We assume these services are independently tested and are not the main focus of the testing strategies discussed here. Suppose your application directly accesses a dedicated database rather than querying data over HTTP APIs. In that case, you may seek other resources or adjust the techniques presented in this book to fit your needs.

Although I believe we can apply many of the principles we'll explore in this book universally, I urge you to challenge everything I write. Remember that what works for me and others might be counterproductive in your particular case.

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 writing *good* tests. 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. Sometimes, 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 testing journey many years ago. Let's do this!

Breakdown of Testing Approaches

To create a robust and effective testing strategy for our Vue applications, we need a firm understanding of the various testing approaches available. In this chapter, we examine different test types and their use cases, laying the groundwork for our overall testing strategy.

In this chapter, we will:

- Learn about the spectrum of testing approaches, ranging from manual testing to fully automated tests
- Explore the four main types of automated tests: End-to-end (E2E) System Tests, Application Tests, Component Tests, and Unit Tests
- Discover the advantages and limitations of each test type and when to apply them
- Recognize the role of manual testing and its limitations, such as identifying UX issues or exploring edge cases
- Highlight the importance of automated tests in ensuring the deployability and quality of our applications
- Gain insights on how to integrate different testing approaches to form a comprehensive and well-rounded testing strategy

We're not pitting different testing methodologies against each other. Instead, we are focusing on understanding how different testing methodologies can complement each other to suit our project's requirements best. By the end of this chapter, we will clearly understand the various testing approaches and be well-prepared to create a tailored testing strategy for our Vue projects.

The Role of Manual Testing in Test-Driven Development

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.

Knowing when and how to incorporate manual testing into our testing strategy enables us to create a well-rounded testing approach that ensures not only the functionality of our application but also its usability and overall user experience.

Benefits and Limitations of Manual Testing

Systematic manual testing can lead to excellent results in preventing the deployment of errors that only occur in particular edge cases. Quality Assurance (QA) professionals excel at finding various types of errors, including those that often only happen in exceptional circumstances. But manual testing isn't just about bug hunting. It's also an invaluable tool for assessing the user experience of our product.

Manual testing serves a dual purpose: It helps us answer not merely the question of *'Does it work?'* but also *'Does it work well?'*

Testing in Production

Testing directly on the production system can be risky, as it may negatively impact user experience, data integrity, and system stability. Therefore, testing on a staging environment closely mimicking the production system is preferable.

A staging environment allows for thorough testing without the risks associated with testing on production. It should be fully independent of the production system to ensure that any changes or tests performed do not affect the live application.

However, be aware that maintaining a staging environment that perfectly mirrors the production system can be a complex and resource-intensive task. It necessitates maintaining parity between the two environments. Make sure to have automated processes to synchronize configurations, data (remember to anonymize sensitive information), and code updates.

The decision to test on production or use a staging environment depends on the specific needs and risks associated with the application and the tests we perform. In some cases, specific tests still need to be conducted on the production system, especially when performing performance testing or real-world user behavior simulations.

Don't Strive for Perfection

When starting out writing tests, a couple of common questions arise:

- How close should our testing approach be to the reality of our users?
- 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 available browsers and devices we want to support?

Suppose our goal was to guarantee absolute certainty our application works under real-world circumstances. In that case, we'd have to closely observe a representative set of users on the production system performing all possible actions within our application. In this *perfect* world scenario, the users do not know we are keeping taps on them so as not to distort the results. In practice, however, we must respect the privacy of our users, and, even more importantly, we want to know about any errors *before* deploying our application to production.

Although it's possible to use tools that track actual user behavior to find errors, we typically only go this route to uncover usability issues. Such tests can be instrumental in finding problems with our application's UX and thus justify the high costs and delayed feedback. However, due to the extended time gap between deployment and data collection, monitoring our users is unsuitable as a systematic approach to finding errors in our system.

But what about performing tests in production(-like) environments in various available browsers and devices? Here, the answer is more nuanced. You have to weigh costs against potential benefits and your risk appetite.

- How big of a deal is it if our service stops working in a particular niche browser after deploying a new feature?
- How fast can we recover after deploying faulty code?
- Is our application prone to bugs we can only replicate in production-like scenarios?

Manual testing, performed on production-like systems with various browsers on real devices, maybe even by actual users, might seem like the only bullet-proof way to be absolutely certain our application works in all circumstances. But going that route is not sustainable.

Remember that striving for maximum thoroughness in testing can be very time-consuming, resource-intensive, and expensive and offers diminishing returns.

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, humans are well suited to empathize with other humans and thus determine 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: Manual testing is expensive not only because of the high personnel effort but also because developers receive feedback very slowly during their daily work. 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

My typical experience as a developer revolves around building important but ultimately low-stakes applications like job boards and CV builders. However, I want to acknowledge that other types of applications have significantly higher stakes.

Imagine working on software for medical devices, financial systems, or aviation control systems. In these fields, the margin for error is incredibly slim. In such environments, we can justify the cost of complementary manual testing by the potential consequences of software failure.

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 entirely automated techniques. 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 important!**

Balancing Manual and Automated Testing

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

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

Four Types of Automated Tests

Automated tests are crucial for ensuring that applications and even whole systems perform as expected. They significantly reduce manual testing efforts, empowering us to run tests frequently with minimal additional cost. The principle is straightforward: every test that can be automated should be automated. This approach allows us to allocate resources more efficiently, focusing our manual QA activities on looking for errors that automated tests cannot detect.

> *Every test that can be automated should be automated.*

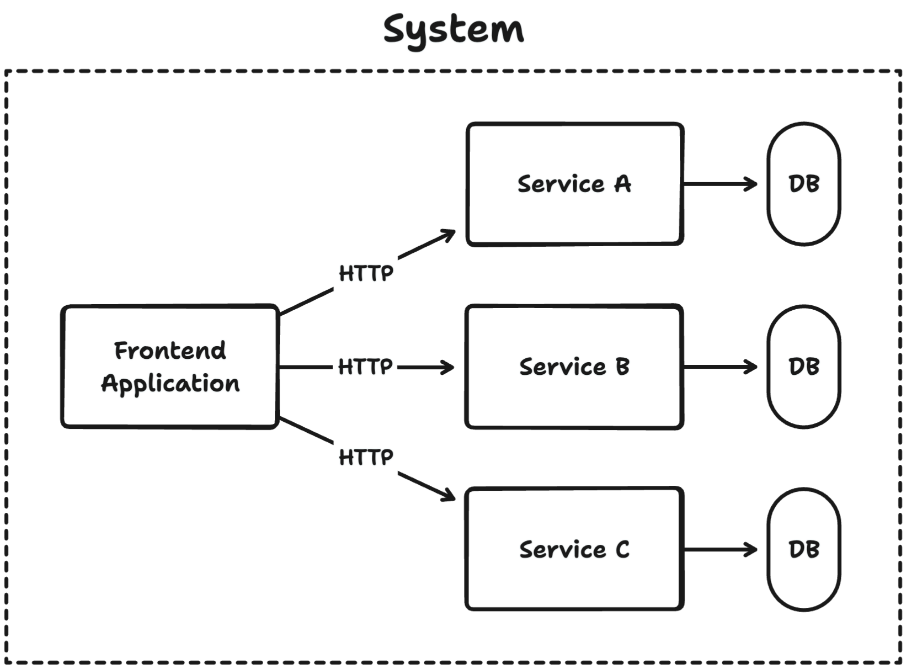
In the realm of automated testing, we often encounter various terms for different test types, such as E2E tests, acceptance tests, integration tests, and unit tests. As there is no official authority to standardize these terms, we will define and differentiate between four types of automated tests:

1. **E2E System Tests:** These tests evaluate the entire system, including the infrastructure layer (data persistence, third-party services, etc.). They ensure that all parts work together seamlessly, comprehensively assessing the system's overall functionality.
2. **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. Unlike E2E System Tests, these tests isolate the application from external dependencies, using mocks to allow for a more targeted assessment of the application's functionality.
3. **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 quickly identify and resolve issues at a granular level.
4. **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 code.

By understanding the distinct purposes and characteristics of these four types of automated tests, we can create a comprehensive testing strategy that ensures the reliability of our Vue applications while iterating quickly on new features.

E2E System Tests

I define E2E System Tests as tests that test a whole system without mocking or stubbing any dependencies of the frontend application. While we use mocks in other forms of automated testing to run tests independently of services external to our Vue application, we avoid them altogether in E2E testing. On one *end,* a user interacts with our application's UI; on the other *end,* we typically write data to a database or trigger an action. An E2E System Test simulates actual user behavior and triggers the same backend processes that would occur when performing the same actions for real.



View of an entire system with one user-facing frontend application and multiple (micro)services.

Because we test the entire system, including all (micro)services and the infrastructure, *end-to-end,* E2E System Tests instill maximum confidence. Testing all the possible interactions between different system parts allows us to identify errors that would otherwise 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);
});
```

Yet, despite the comprehensiveness of E2E System Tests, we cannot guarantee our system is error-free! Even the best and most elaborate tests can only prove the presence of errors, not their absence. Keep that in mind when later deciding between more and less thorough testing approaches.

Challenges and Costs of E2E System Tests

While automated E2E System Tests instill high confidence, they come with challenges and costs. For example, to ensure our tests provide reliable results, we must create a test environment that closely resembles the production environment. Establishing and maintaining such a replica is a complex task. We need to prevent interference between test scenarios. Our test infrastructure must allow us to run tests in any order and even in parallel without affecting each other.

Careful planning is essential for creating a test environment that enables consistent and reliable results. Taking shortcuts may lead to long-term issues, such as false positives and flaky tests, which can significantly impact the reliability of our tests.

Another factor to consider is the significant resources required to run one or multiple exact copies of the production system. To cut costs, many companies use underpowered hardware for their testing and staging environments. As a result, E2E test suites may take several hours to complete, undermining the goal of leveraging testing for fast feedback loops.

We should aim for stable and swift tests. Flaky tests with lengthy runtimes often prove too costly due to their moderate benefit for developers and high maintenance requirements. Striking the right balance between cost and confidence is crucial when designing and implementing E2E System Tests.

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. We must configure the system to allow for isolated, parallel test execution and ensure the necessary resources and services are available for each test scenario.

Let's consider a microservices architecture as an example. To set up an effective E2E test suite, we must bring each microservice into a particular state. For instance, if we want to test if a web shop user can cancel an order, at the very least, we must set up a fresh user and assign a pending order to them. Designing and maintaining such an environment is a complex task that could fill another book. However, investing in a well-structured architecture and infrastructure will ultimately enhance the efficiency, reliability, and value of our E2E System Tests.

The technical implementation of E2E System Tests varies based on the system under test and the technologies used. As a result, I won't 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.

Using Smoke Tests to Balance Confidence and Costs

E2E System Tests offer a high level of confidence but come at a steep cost. Given the complexity and time-consuming nature of implementing and maintaining E2E System Tests, we must carefully weigh the pros against the cons to decide which parts of the system warrant E2E testing. One approach to significantly reduce the run times of E2E System Tests is to rely on smoke tests.

Two factors make E2E System Tests particularly expensive: the effort required to set up and continuously maintain an environment for conducting these tests and the comparatively long run times. Adopting a smoke testing approach can help address these challenges.

Smoke tests focus on the most critical and delicate parts of the system, particularly 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 our system's core functions. For example, in an e-commerce application, we would test adding a product to the shopping cart or going through the checkout process. 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 service. We are less concerned with 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 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 our E2E System Test pipeline is crucial for staying informed about whether our application is currently deployable. By addressing any issues promptly, the team can proactively identify and resolve problems before they accumulate.

Remember: whenever our pipeline fails, fixing it becomes the top priority of the whole team. If we allow our tests to fail without taking immediate action, they lose their purpose and value. For example, if a team starts ignoring a pipeline because it's flaky and constantly failing with false positives, they may lose trust in the test suite. Over time, this can lead to a dangerous situation where the team becomes desensitized to failing tests and starts to assume that all failures are false positives. At some point, the time will miss a critical error. Clear and concise reporting helps stakeholders stay informed about the application's status and whether it is deployable.

Neglecting to address failing tests or ignoring the results of our E2E System Tests can lead to a false sense of security and potentially allow critical issues to slip through to production unnoticed. By prioritizing the maintenance and reliability of our test suite, we can ensure that our E2E System Tests continue to provide the intended value and help us deliver a high-quality application.

Continuous Improvement

As our application evolves and grows, we must review and improve our E2E System Testing process. We should regularly update test cases to cover new features and functionality. Addressing false positives and flaky tests is crucial for maintaining the reliability and effectiveness of our test suite.

Over time, we should refine our testing strategy to focus on the most critical components of our system. By prioritizing the areas with the most significant business impact, we can ensure that our E2E system tests provide the highest value.

While automated E2E System Tests can provide considerable benefits in ensuring the reliability of our system, they come with costs and challenges. However, by incorporating smoke tests and continuously improving our testing process, we can achieve high confidence while keeping costs in check.

Application Tests

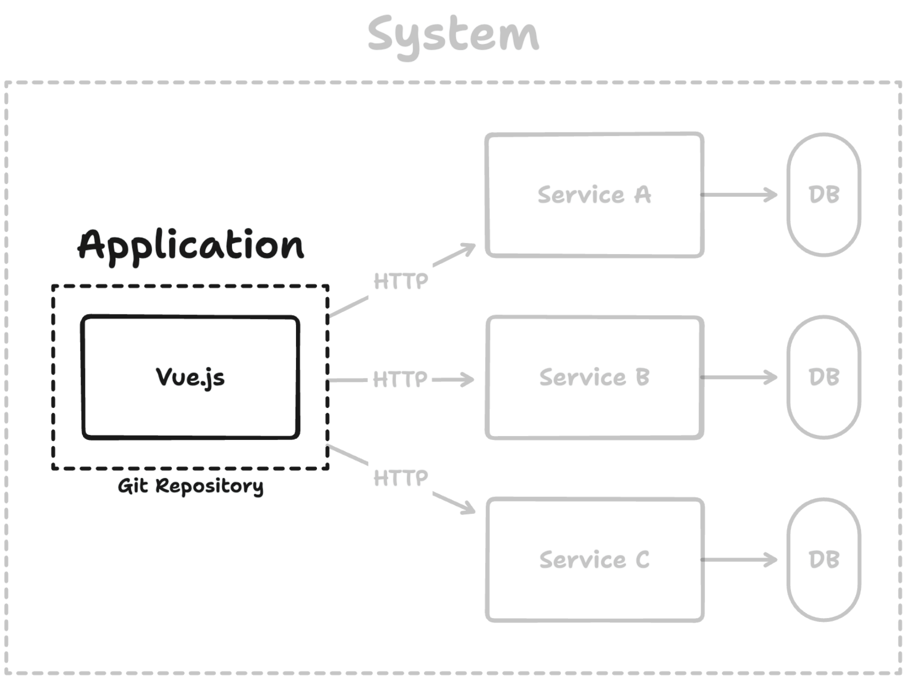
The term *Application Test*, as I use it in this book, doesn't have a universally agreed-upon meaning. Many resources use the term *integration test* for this type of test, while others use *E2E test* to describe tests with a similar purpose. Even the phrase *application test* has varying meanings online. For clarity, I will use the term *Application Test* throughout this book and differentiate between *E2E System Tests* and *Application Tests.*

You might disagree with the names I've chosen, and that's okay. Feel free to adopt a naming scheme that fits your or your team's preferences. However, one thing is crucial: Due to the ambiguity of the terminology, everyone within a team or company must have a shared understanding of each test type. Establishing a common language helps avoid confusion and ensures effective communication when discussing and working with different types of tests. Decide on the terminology you want to use in your company and make it a priority that everyone is on board.

```
// 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

Although both categories of tests ensure we fulfill a set of acceptance criteria, Application Tests serve a slightly different purpose than E2E System Tests. While E2E System Tests aim to verify the integration of our application with the entire system, including all services, Application Tests focus on validating only the behavior of *our application,* ignoring the rest of the system. The goal is to prioritize fast feedback loops over chasing absolute certainty.



We focus on one system piece: the Vue.js-based Single-Page Application.

In the context of this book, I define an *application* as a Vue.js-based Single Page Application (SPA) within a single repository, without any services and APIs it might rely on to get its data. Users interact with our system through the Vue application, meaning all acceptance criteria, formulated from the users' perspective, go through the Vue SPA first. Parts of the system outside our Vue SPA (database, backend services, etc.) are invisible to our users, and we can safely ignore them for this type of testing.

When writing Application Tests, we assume that all other parts of our system function as expected. This assumption is reasonable, considering that we may not have access to the code of all the services our application communicates with. By presuming that the external pieces we rely on work as expected, we can safely replace them with mocks as long as we ensure adherence to the contracts provided by those external services.

Application Tests vs. E2E System Tests

E2E System Tests cover the entire system, including all its components, like API services and databases. In contrast, the primary goal of automated Application Tests is to ensure that the application our users directly interact with, functions correctly. We test the behavior of our *application* rather than the implementation of the *system* as a whole.

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 does not concern us when testing a web application's user interface. However, verifying whether data is stored correctly is crucial when testing a backend API service.

The key difference between Application Tests and E2E System Tests is the clear boundary to other system parts, with Application Tests focusing on a single application while ignoring or *mocking* all aspects of the *system* that are not part of our *application.*

While some may question the validity of tests that omit large parts of our system, the advantages of this compromise generally outweigh the disadvantages. Comprehensive E2E System Tests offer diminishing returns compared to simpler and faster Application Tests. The minimal confidence gain they provide is usually not worth the additional cost.

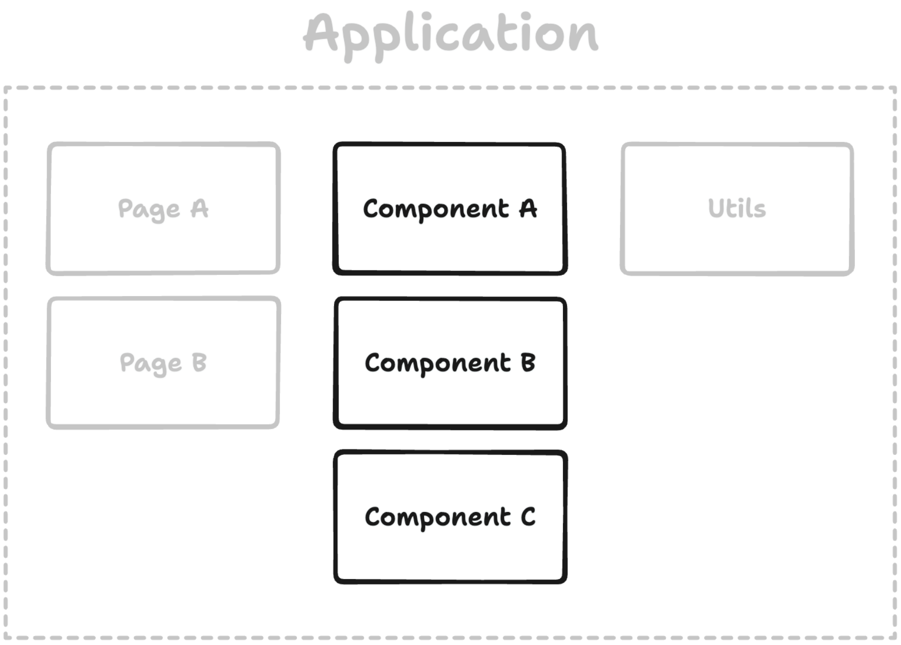
When deciding between E2E System Tests and Application Tests, consider the following:

- Use E2E System Tests sparingly, primarily as smoke tests, to ensure the system's critical functionality works as expected.
- Consider the project's specific needs and available resources.
- When in doubt, rely more heavily on Application Tests and prioritize rapid feedback loops.

When done correctly, Application Tests are fast, stable, and easy to implement. They are generally more cost-effective due to less maintenance and shorter feedback loops, positively impacting the team's velocity. While they don't guarantee that our entire system will work, 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 I refer to Component Tests, I'm 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.



An application consists of multiple pieces and components.

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 cannot tell us whether our entire 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 they enter invalid data', 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

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

The crucial point to remember is that the type of test—whether a Component or Application Test—is not determined by the tool we use but by what and how we test. We can run Application or E2E System Tests using Jest or Vitest, just as we can conduct Component Tests with Playwright or Cypress. These tools are versatile enough to test both individual components and complete applications.

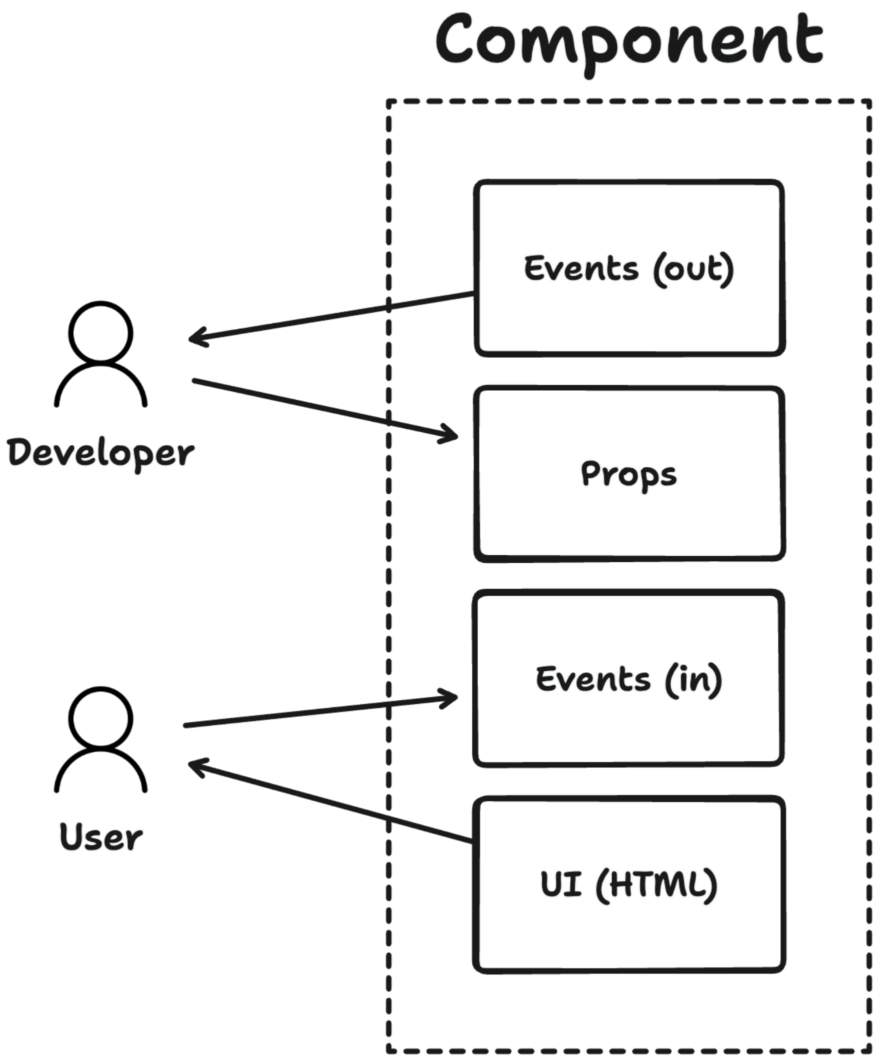
Ultimately, we should decide to use a particular tool based on factors such as the level of confidence we require, the speed of execution, and the ease of setup and maintenance. By understanding each tool's capabilities and trade-offs, we can make informed decisions that align with our testing goals and project requirements.

Two Perspectives: User And Developer

Unlike E2E System Tests and Application Tests, which we write from a user's point of view, Component Tests require us to consider two angles: the perspective of a *user* and how a *developer* interacts with a component when using it to build a new feature.

While we also take a developer's perspective with Unit Tests, the main differentiator between Unit and Component Tests is that with Component Tests, we must consider a user's perspective in addition to the developer's view.

As developers, we interact with a component through its API, which includes props, events, and slots. It's crucial to ensure that the component behaves as expected when receiving different combinations of inputs through its API. Equally important is to test how the component renders and behaves in response to user interactions, such as clicking buttons or filling out forms.



Two perspectives: Developer and user

By considering both the user's and developer's perspectives, we can create comprehensive Component Tests that ensure our Vue components work as intended and integrate seamlessly with other parts of our application. Always keep these two points of view in mind when writing Component Tests.

What If Writing Tests Feels Like a Chore?

If writing Component Tests feels like a chore, it might indicate that our code is hard to test, suggesting room for improvement in code quality. The more we adhere to best practices when writing code, the easier it becomes 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. To make our code more testable, we should avoid registering components globally, using Vue plug-ins excessively, and fetching data at every level of our component tree as we see fit.

Many developers who struggle with writing and maintaining tests tend to write tests *after* finishing the code for a new component. However, if we flip the script and practice TDD, writing the test *before* the actual implementation, we will find that writing testable code comes naturally.

Focusing on writing testable code and embracing TDD can make writing Component Tests more enjoyable and efficient.

Benefits of Component Tests

Component Tests offer several advantages in our testing strategy. We can write them quickly, and they have low maintenance overhead. These tests provide nearly instant feedback on whether individual components of our application work correctly in isolation.

However, it's important to note that Component Tests cannot tell us if we've wired all our components together correctly. As a result, the confidence we gain from Component Tests regarding the functionality of our entire application is limited.

Despite this limitation, Component Tests remain a valuable building block in our overall testing strategy. They provide fast feedback at minimal costs, allowing us to catch issues early in the development process and iterate quickly.

Unit Tests

Unit Tests play a crucial role in ensuring the quality and reliability of an application. Unlike Component Tests, which focus on testing entire Vue components, Unit Tests target the most atomic pieces of our application, such as modules, classes, and functions. By testing these elements in isolation, we can quickly identify issues and potential improvements within our codebase.

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

Unit Tests are essential for several reasons:

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

Writing Effective Unit Tests

To write effective Unit Tests, 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 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. Avoid complex test setups and test a single behavior or functionality per test. Simple tests make it easier to identify and fix problems when tests fail.
3. **Use appropriate test data:** Choose test data representing real-world scenarios and edge cases. Mimicking the real world ensures that our tests cover various possible inputs and outcomes, increasing the reliability and robustness of our code.
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 and that it behaves as expected when integrated with other application parts.
5. **Practice TDD:** Writing Unit Tests *before* implementing the actual code leads to more testable and maintainable code. TDD encourages using best practices and modular design, making it easier to write *good* Unit Tests. By focusing on testability from the start, we can create code that is more reliable, easier to understand, and simpler to modify.

The Distinction Between Unit Tests and Component Tests

Despite their differences, there is some overlap between Component and Unit Tests. Both types of tests focus on testing a single *unit* of code in isolation, ensuring it functions as expected.

However, as mentioned previously, Component Tests focus on testing individual Vue components from *both* the user's *and* the developer's perspectives. This dual perspective clearly distinguishes Component Tests from Unit Tests, despite some conceptual overlap between the two.

Unit Tests are even more granular than Component Tests, targeting specific functions, classes, or modules in isolation. They ensure that those individual *units* of code work as expected, disregarding the larger whole they are part of.

Both Unit Tests and Component Tests are crucial for ensuring the quality and reliability of our application, but they serve different purposes within our overall testing strategy. Component Tests focus more on the UI and user experience, verifying that they behave as intended when users interact with them. With Unit Tests, we aim to validate the correctness of the underlying logic driving the application.

The Value of Unit Tests

Unit Tests provide several benefits to our 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. By testing small, isolated units of code, we can quickly identify and fix bugs before they propagate to other parts of the application.

Additionally, Unit Tests can serve as a form of documentation, guiding developers on how to use and interact with various parts of our application. Well-written Unit Tests demonstrate how to call functions or methods, what inputs they expect, and what outputs they produce. This type of documentation is particularly helpful for new team members or when revisiting code after a long time.

While Unit Tests alone 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 approach that ensures the reliability and maintainability of our application in the long run.

Investing time in writing Unit Tests may seem like an additional effort upfront, but it pays off in the form of a more stable, maintainable, and easier-to-understand codebase. Embracing Unit Testing as part of our development process can lead to better code quality and faster development cycles.

Summary

In this chapter, we explored the various testing approaches, from manual testing to 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 crafting a testing strategy that caters to our application's specific requirements.

Key Learnings

1. Exploratory, manual testing has its place in TDD, uncovering weird edge cases and identifying UX problems.

2. We should do manual testing in a non-blocking manner (except in high-risk environments).
3. We must ensure the deployability of our applications through fully automated tests.
4. E2E System Tests comprehensively assess the entire system, including its infrastructure.
5. Application Tests focus on verifying acceptance criteria from the user's perspective, examining the user interface and the application's business logic.
6. Component Tests concentrate on individual Vue components within the application, ensuring they function correctly from the perspective of users and developers.
7. Unit Tests validate the functionality of smaller units within the application, evaluating the correctness of our code from a developer's perspective.
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. 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.