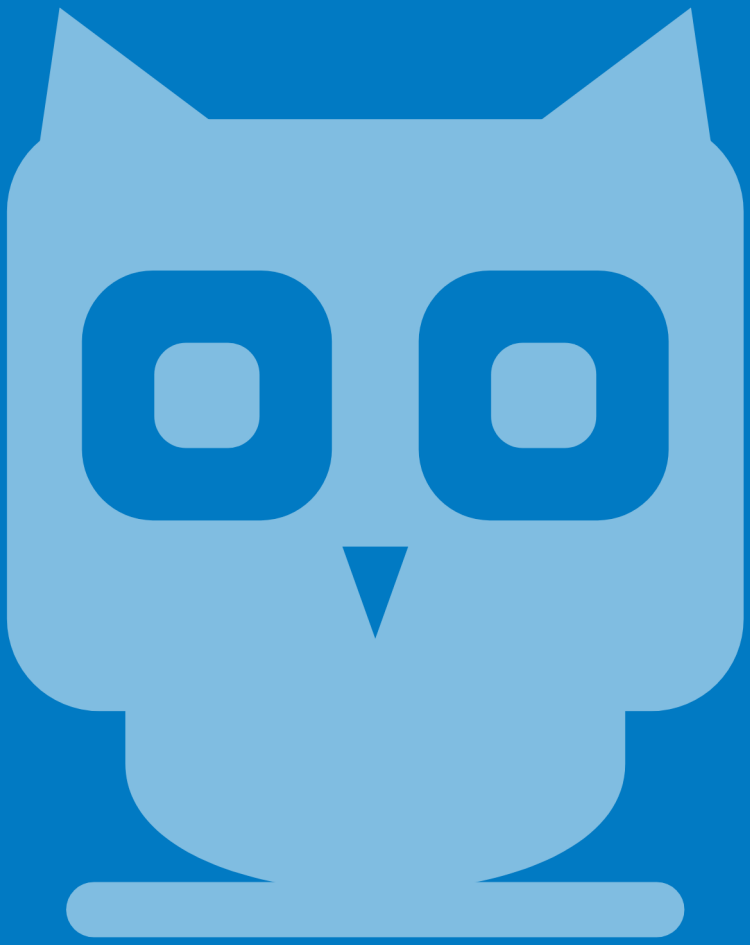
OWL 2.0

WHERE ODOO ENDS AND YOU BEGIN

Understand Odoo’s frontend well enough to argue with it



Simplify it!

Grover Menacho - Simplify IT S.R.L.

Edition v1 · August 2026 · simplifyit.com.bo

Contents

License

The Example Code Is Licensed Differently

The terms above cover the **book text only**. The companion example addons at github.com/simplifyitsrl/owl-book-examples are a separate work released under the **Apache License 2.0**.

That means the NonCommercial and NoDerivatives restrictions do **not** apply to the code: you are free to use, modify, and redistribute the example addons, including inside commercial Odoo projects, as long as you keep the license and attribution notices that ship with them. That is the whole point of putting them there — go build something with them.

This is not official Odoo documentation, and the author has no affiliation with Odoo S.A. beyond being, like the reader, someone who has to make their framework work in the real world.

Chapter 0: A Note From the Author

I learned Odoo back when it was still OpenERP, around 2013, when the documentation was a single PDF laid out like a brochure, when Odoo had barely a handful of modules and was Open Source and nothing else. From the start I loved playing with whatever was within reach: tweaking views, building my first POS synchronizer between computers so that a single register could charge everyone's orders. Basically, I have always liked taking the tools I have one level past where they ship. That is the intent underneath this book: to squeeze Odoo and get more out of it than what comes in the box.

With that out of the way, let me tell you a bit more about myself. I have been working with Odoo for over a decade. I started at Poiesis Consulting, in Bolivia, one of the best implementation companies in the country. Then that same restlessness I have never quite shaken pushed me into freelancing, and getting to know other cultures, other companies, and other ways of working opened up my head and led me to do far more than I would have imagined. For these last two years, through Simplify IT S.R.L., my work has centered on making Odoo do what businesses actually need it to do — implementations, migrations, custom modules, integrations with tax authorities and marketplaces across Bolivia, Latin America, Europe, and the United States. Most of that work happens below the surface: models, views, business logic, the plumbing that keeps an ERP honest. OWL was, for a long time, something I used without truly seeing.

That changed the day a client's upgrade from an older Odoo version broke every custom widget we had built for them. The old code did not just need patching — it needed rethinking. Going through that migration line by line, I found myself asking questions I had quietly avoided for years: Why does OWL exist at all, when Odoo could have used React or Vue like everyone else? What actually changed between OWL 1.0 and 2.0, and why did it matter enough to justify starting over instead of patching forward? What is the mental model I am supposed to have, instead of the one I accidentally built by copy-pasting from existing addons?

This book is the resource I wished existed when I was asking those questions.

It is written from the perspective of a working Odoo consultant, not a framework author. I am not going to pretend OWL is the only reasonable choice, or that every design decision in it is beyond question — I will tell you why the choices were made, what trade-offs they carry, and where they show up in the code you write every day. My goal is for you to stop treating OWL as "the way Odoo happens to do frontend" and start treating it as a tool you understand well enough to use with intention, debug with confidence, and occasionally disagree with.

Who This Book Is For

If you build custom addons, portal pages, or backend extensions for Odoo — whether you come from a Python background reaching into the frontend, or a JavaScript background trying to make sense of Odoo's conventions — this book is for you. I assume you can read JavaScript and that you have opened an Odoo addon's `static/src` folder before, even if it intimidated you. I do not assume you already know what a reactive proxy is, or why Odoo abandoned QWeb-only rendering, or what "OWL" even stands for the first time you hear it.

How to Use This Book

Each chapter builds on assumptions from the ones before it, but I have tried to make the early chapters — the "why" before the "how" — valuable even if you skip straight to a later chapter for a specific answer. Along the way you will find Reflection Questions and Common Pitfalls sections drawn directly from real project work, not hypothetical edge cases. When something broke a client's production instance, or cost me an afternoon I did not have to spare, you will find it here, explained plainly enough that it does not cost you the same afternoon.

A last note: this is not official Odoo documentation, and I have no affiliation with Odoo S.A. beyond being, like you, someone who has to make their framework work in the real world. Where my explanations diverge from or simplify the official source, that is a deliberate choice in service of a working mental model — not a claim to greater authority than the framework's own code.

A Note on How This Book Came Together

This book is built on my own experience and everything I learned along the way — but it was refined and sharpened with the help of Claude, from Anthropic. Working through explanations, structuring chapters, and pressure-testing my own mental models with Claude accelerated this project considerably and made writing this book far more achievable than it would have been otherwise. I want to give credit where it's due.

Let's get into it.

Chapter 1: The "Why": From Static Pages to Modern UI

Why this chapter? Every consultant eventually gets asked "why can't we just use jQuery for this?" — this chapter is the answer, backed by the actual history of the web and of Odoo's own frontend.

What is Odoo trying to solve with this? Odoo needed a UI layer that could keep thousands of existing QWeb templates and legacy widgets working while still giving developers a modern, declarative way to build interfaces — a general-purpose framework like React couldn't guarantee that.

Real-world application: When a client's list view feels sluggish or a custom widget breaks after an upgrade, the root cause is almost always a misunderstanding of declarative vs. imperative UI — the distinction this chapter builds from scratch.

Welcome! Before we write a single line of OWL code, we need to understand *why* it exists. Every tool is created to solve a problem, and UI frameworks like OWL are a solution to decades of challenges in building dynamic, interactive user experiences on the web. This chapter is a journey through time, showing how we got from simple, static pages to the powerful applications we use today. Understanding this history will make the "how" of OWL much clearer.

The Old Way: The Server-Rendered World

Imagine walking into a restaurant. You look at the menu, tell the waiter you want a steak with fries, and a few minutes later, a complete, finished plate is placed in front of you. This is exactly how the web worked for a long time, and it's how classic Odoo applications were built.

This is the **server-rendered** model.

1. Your browser sends a request to the Odoo server (e.g., "I want to see the contact page for 'John Smith' ").
2. The Odoo server does all the work. It runs Python code, fetches data from the database, and uses a QWeb template to assemble a complete HTML page.
3. It sends this finished HTML page back to your browser, which simply displays it.

This approach is simple and effective. The server delivers a finished "plate." But what happens if you just want to change one small thing, like adding a pinch of salt? In this model, you have to send your entire plate back to the kitchen and wait for a whole new one to be made.

For web applications, this meant that even a tiny change—like marking a task as complete or updating a customer's phone number—required a full page reload. The screen would go white, and you'd have to wait for the server to send a completely new version of the page. It was slow, clunky, and not a great user experience.

The jQuery Era: Sprinkling Interactivity

Developers knew the full-page-reload problem had to be solved. The solution that dominated the web for nearly a decade was **jQuery**.

jQuery was a brilliant JavaScript library that allowed developers to reach into the web page and manipulate it directly, without asking the server for a new one. This is called an **imperative** approach. You give the computer direct, step-by-step commands.

Think back to our restaurant analogy. Instead of ordering a new plate, you now give the waiter imperative commands:

> "Go to my table. Pick up the salt shaker. Move it three inches to the left. Now, pick up the pepper shaker..."

You are describing *how* to change things. In code, it looked like this:

```
// When the button with the id 'complete-task-btn' is clicked...
$('#complete-task-btn').on('click', function() {
    // 1. Find the element with the id 'task-status'.
    // 2. Change its text to 'Completed!'.
    $('#task-status').text('Completed!');

    // 3. Find the button itself.
    // 4. Add the 'btn-success' class to make it green.
    $(this).addClass('btn-success');
});
```

This was a huge leap forward! We could now build much more responsive interfaces. But as applications grew larger, the imperative approach created its own set of problems:

"Spaghetti Code": You ended up with hundreds of little instructions all tied to different buttons and inputs. It became incredibly difficult to follow the logic and understand what state the application was in.

Difficult to Maintain: If you changed the HTML structure, you had to find and update all the JavaScript commands that were looking for the old structure. It was fragile and time-consuming.

The Paradigm Shift: Declarative UIs

The complexity of the imperative model led to the next major evolution in web development: declarative frameworks. This is where OWL, React, and Vue come in.

Instead of telling the computer how to do something, you simply declare what you want the end result to be.

In our restaurant, you no longer give step-by-step instructions. You just declare the state you want:

> "I want the salt shaker to be on the left side of the fork."

You don't care how the waiter does it. You've described the final state, and you trust the waiter (the framework) to make it happen efficiently.

In a declarative framework like OWL, you link your UI to your application's data (its "state").

```
<!-- In our template (simplified - you'll learn the real syntax in Chapter 5) -->
<button t-att-class="state.isCompleted ? 'btn-success' : 'btn-primary'">
  <t t-esc="state.isCompleted ? 'Completed!' : 'Mark as Complete'"/>
</button>
```

When you want to change the UI, you don't touch the UI directly. You just change the data: `this.state.isCompleted = true;`.

OWL sees that the state has changed and automatically and efficiently figures out the minimum number of steps needed to update the button's text and color. You declared the "what" (the UI should look like this when isCompleted is true), and OWL handled the "how."

This declarative approach is the foundation of all modern web development. It leads to code that is:

- **Predictable**: The UI is always a direct representation of the state.
- **Maintainable**: You can change the underlying HTML and logic without breaking dozens of little commands.
- **Scalable**: It's much easier to build large, complex applications.

Why Odoo Created Its Own Framework: The Perfect Fit

At this point, you might be wondering: "If React, Vue, and Angular are so popular and powerful, why didn't Odoo just use one of them? Why create OWL from scratch?"

This is an excellent question, and the answer reveals the thoughtful engineering behind OWL.

The Odoo-Specific Challenges

Odoo is not just any web application—it's a comprehensive business management suite with unique requirements:

Deep Integration with QWeb: Odoo already had years of investment in QWeb templates for server-side rendering. The new framework needed to work seamlessly with existing QWeb syntax and conventions, not replace them entirely.

Backward Compatibility: Odoo has thousands of existing modules and customizations built with the older widget-based system. Any new framework had to coexist with this legacy code during a gradual transition period.

Performance at Scale: Odoo applications can have dozens of simultaneous components displaying hundreds of records. The framework needed to be optimized for this specific use case, not general web development.

Small Bundle Size: Adding React or Vue would significantly increase Odoo's JavaScript bundle size. OWL is lightweight and focused, including only what Odoo actually needs.

The Technical Advantages of OWL

By creating their own framework, the Odoo team could make specific design decisions that perfectly match their needs:

Native QWeb Integration: OWL templates are QWeb templates. There's no translation layer or syntax conversion needed—developers use the same template language they already know.

Optimized Rendering Engine: OWL's rendering engine is specifically tuned for Odoo's patterns of data display and manipulation, making it fast for typical ERP operations.

Built-in Services System: OWL comes with a service system that directly integrates with Odoo's RPC layer, database models, and business logic.

Gradual Migration Path: OWL was designed from day one to work alongside the existing Odoo JavaScript framework, allowing for smooth, incremental upgrades of legacy code.

The Strategic Vision

Creating OWL wasn't just about solving today's problems—it was about ensuring Odoo's long-term technological independence and innovation. By owning their UI framework, the Odoo team can:

- Evolve the framework in lockstep with Odoo's business requirements
- Optimize performance for ERP-specific use cases
- Maintain complete control over the developer experience
- Avoid external dependencies and potential breaking changes from third-party frameworks

This is the problem OWL was built to solve for Odoo: providing all the power and elegance of modern declarative UI development while being perfectly tailored to the unique needs of business applications. It's not just a framework—it's Odoo's strategic investment in the future of their platform.

The Evolution: From OWL 1.0 to OWL 2.0

If you're reading this book, you might have some experience with OWL 1.0, or you might have heard about the transition. Understanding what changed between versions will help you appreciate why OWL 2.0 is such a significant improvement and why it's worth learning from scratch.

What Was OWL 1.0?

OWL 1.0 was Odoo's first attempt at a modern JavaScript framework. It served its purpose well and introduced many developers to declarative UI concepts. However, as it was used in production across thousands of Odoo installations, certain limitations became clear:

Performance Bottlenecks: The virtual DOM implementation, while functional, wasn't optimized for the complex, data-heavy interfaces that Odoo applications require.

Complex State Management: Managing state across multiple components required verbose patterns and a lot of boilerplate code.

Template Limitations: While QWeb was supported, the integration felt somewhat forced, and some advanced template features were difficult to use.

Learning Curve: The API, while powerful, had many concepts that were difficult for new developers to grasp quickly.

The OWL 2.0 Revolution: What Changed

OWL 2.0 wasn't just an update—it was a complete redesign from the ground up, incorporating lessons learned from years of production use:

1. Simplified Component API

OWL 1.0 approach:

```
class MyComponent extends Component {
    constructor(parent, props) {
        super(parent, props);
        this.state = {
            counter: 0
        };
    }

    willStart() {
        return this.loadData();
    }

    async loadData() {
        // Complex async setup
    }
}
```

OWL 2.0 approach:

```
class MyComponent extends Component {
    setup() {
        this.state = useState({
            counter: 0
        });

        onWillStart(this.loadData);
    }

    async loadData() {
        // Same async logic, cleaner setup
    }
}
```

2. Revolutionary Hooks System

OWL 2.0 introduced a hooks-based architecture (inspired by React hooks) that makes components more modular and reusable:

Before (OWL 1.0): State and lifecycle management was tightly coupled to the component class.

After (OWL 2.0): Hooks like `useState`, `useService`, `useRef`, and `useEffect` allow you to compose functionality in clean, reusable ways.

```
// OWL 2.0: Clean, composable logic
setup() {
    const state = useState({ count: 0 });
    const orm = useService("orm");
    const notification = useService("notification");

    useEffect(
        () => {
            // Side effects are cleanly separated
        },
        () => [state.count]  // dependencies are returned by a function
    );
}
```

3. Significantly Improved Performance

Faster Rendering: OWL 2.0 replaced the classic virtual DOM with a "block-based" rendering engine that is significantly faster, especially when handling large lists and frequent updates.

Smarter Re-rendering: The new reactivity system is much better at detecting when components actually need to update, reducing unnecessary re-renders.

4. Enhanced Developer Experience

Better Error Messages: When something goes wrong, OWL 2.0 provides much clearer, more actionable error messages with precise line numbers and context.

Improved DevTools Integration: Browser debugging is much more intuitive, with better component inspection and state visualization.

TypeScript Support: OWL 2.0 was designed with TypeScript in mind, providing excellent type safety for large projects.

5. Streamlined Service System

OWL 1.0: Services were available but the integration pattern was verbose and sometimes confusing.

OWL 2.0: The `useService` hook makes accessing Odoo services (like `orm`, `notification`, `action`) incredibly clean and consistent.

```
// OWL 2.0: Simple service access
setup() {
    this.orm = useService("orm");
    this.notification = useService("notification");
    this.actionService = useService("action");
}
```

6. Better Integration with Modern JavaScript

OWL 2.0 embraces all the modern JavaScript features we covered in this chapter: - Native support for `async/await` patterns - Excellent integration with ES6 modules - Built-in support for destructuring in templates and components - Template literals work seamlessly with the template system

Migration Strategy: Why Start Fresh?

If you're familiar with OWL 1.0, you might wonder whether to migrate existing code or start fresh. Odoo's recommendation—and the approach of this book—is to **treat OWL 2.0 as a new framework** rather than an upgrade.

Why this approach makes sense:

Different Mental Models: The hooks-based architecture requires thinking about components differently than the class-based approach of OWL 1.0.

New Best Practices: Patterns that were optimal in OWL 1.0 might be anti-patterns in OWL 2.0.

Clean Slate Benefits: Starting fresh lets you take advantage of all the new features without being constrained by old patterns.

Future-Proofing: OWL 2.0 is the foundation for Odoo's frontend for years to come. Learning it properly now is an investment in your long-term productivity.

What This Means for You

Whether you're completely new to OWL or coming from OWL 1.0, this book will teach you OWL 2.0 from the ground up using modern best practices. If you have OWL 1.0 experience, some concepts will feel familiar, but approach each chapter with an open mind—the improvements in OWL 2.0 will likely change how you think about building user interfaces.

The journey from jQuery spaghetti code to OWL 1.0 was significant. The leap from OWL 1.0 to OWL 2.0 is equally transformative, but in terms of developer productivity, code maintainability, and application performance.

Common Pitfalls

Assuming OWL 1.0 syntax still applies. If you find older Odoo tutorials or modules online, you may see `constructor(parent, props)`, `willStart()`, or `this.trigger(...)`. This book only teaches OWL 2.0, where `setup()` replaces the constructor and callback props replace event triggering (Chapter 9). Don't mix the two styles.

Thinking "declarative" means "no logic." Declarative doesn't mean you stop writing code—it means you stop writing DOM-manipulation code. You still decide what the state should be; OWL decides how to reflect it on screen.

Skipping the "why" to get to the "how." It's tempting to jump straight to Chapter 5 for working code. But the mental model in this chapter—state changes, not DOM instructions—is what makes every later chapter make sense. If a later example confuses you, it often helps to come back and re-read this chapter.

Reflection Questions

1. Using the restaurant analogy from this chapter, describe in your own words the difference between the jQuery (imperative) approach and the OWL (declarative) approach.
2. Look at the jQuery snippet in "The jQuery Era" section. If ten different buttons on a page each needed similar logic, what problems would you expect to run into as the codebase grows?
3. Name two Odoo-specific constraints (mentioned in "Why Odoo Created Its Own Framework") that a general-purpose framework like React would not have solved out of the box.

TL;DR: OWL exists because Odoo needed a declarative UI framework tailored to its own QWeb templates, legacy widgets, and ERP-scale performance needs — OWL 2.0 rebuilt that framework around hooks for a simpler, faster, more maintainable developer experience.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch1_ex1`. Installation instructions are in the repository README.

What's Next?

In the next chapter, we'll set up the tools you need to work effectively with this powerful, purpose-built framework — your editor, the browser DevTools, and the Odoo module structure you'll build everything in from here on.

Chapter 2: Your Essential JavaScript Toolkit

> **Why this chapter?** Most of the debugging time I've billed clients over the years wasn't spent reading code — it was spent watching the DevTools Network and Console tabs. Knowing this tooling cold is what separates guessing from diagnosing.
>
> **What is Odoo trying to solve with this?** Odoo doesn't ship its own debugger — it relies entirely on your browser's DevTools and a properly configured editor/server loop (`--dev=all`), so this chapter is about the generic web tooling every Odoo frontend developer depends on.
>
> **Real-world application:** When a client reports "the button doesn't do anything," the Console and Network tabs from this chapter are almost always where you find the actual error — long before you go anywhere near the OWL source.

Before a chef can cook a masterpiece, they must know their knives. Before a carpenter can build a table, they must master their saw and hammer. For a developer, our tools are just as critical. Writing code is only half the battle; the other half is debugging, inspecting, and understanding what our code is doing.

This chapter introduces the most important tool in your arsenal: your web browser's **Developer Tools**. We'll also cover the essential setup you need in your code editor and provide hands-on exercises to get you comfortable with these tools. Mastering these tools is not optional—it's the fastest way to go from a beginner who is guessing what's wrong to a professional who knows how to find out.

The Browser is Your Best Friend

Every modern web browser (like Chrome, Firefox, or Edge) comes with a powerful suite of built-in "DevTools". You can usually open them by right-clicking anywhere on a webpage and selecting **"Inspect"** or by pressing the `F12` key.

For an Odoo developer, the DevTools are your window into the soul of your application. They let you see the HTML structure, watch communication with the Odoo server, and debug your JavaScript line by line. Let's explore the four most important tabs.

The Console: Your Code's Diary

The **Console** is the first place you should look when something goes wrong. It's a live log where JavaScript can print messages, warnings, and—most importantly—errors.

What it's for: * **Seeing Errors:** When your OWL component crashes, a detailed error message will appear here, often telling you the exact line of code that failed. * **Logging Variables:** You can print the value of any variable from your code using `console.log()`. This is the simplest way to check the

state of your component at any given moment. * **Interactive Testing:** You can type JavaScript directly into the console and execute it immediately—perfect for testing small pieces of code.

How to use it: Imagine you want to see what data is inside a variable called `this.state.tasks`. In your component's JavaScript, you would add:

```
console.log("The current tasks are:", this.state.tasks);
```

When your code runs, this message will appear in the Console, allowing you to inspect the data.

Pro tip: You can use different console methods for different types of messages:

```
console.log("General information");
console.warn("This is a warning");
console.error("This is an error");
console.table(arrayOfObjects); // Displays arrays/objects as neat tables
```

The Elements Tab: The Live HTML

The **Elements** tab shows you the complete, live HTML and CSS of the page you are looking at. This isn't the source code you wrote; it's the final product after your browser has rendered everything, including the HTML generated by your OWL components.

What it's for: * **Inspecting Structure:** You can see exactly how your OWL component's template was turned into HTML. Is that `div` inside another `div` like you expected? * **Testing CSS Changes:** You can select any element and add, remove, or change its CSS styles directly in the browser to see the effect instantly. This is perfect for quickly testing visual changes without having to modify your code and reload. * **Finding CSS Issues:** When something doesn't look right, you can inspect the element to see which CSS rules are being applied and which ones are being overridden.

Pro tip: Right-click on any element in the Elements tab and select "Copy selector" to get the exact CSS selector for that element.

The Network Tab: Spying on the Server

The **Network** tab is your mission control for all communication between your browser and the Odoo server. Every time your OWL component makes an RPC call to fetch data, you will see it here.

What it's for: * **Watching RPC Calls:** When you click a button that should save data, you can watch the Network tab to confirm that a call was made to the Odoo server. * **Inspecting Payloads:** You can click on any request to see exactly what data was sent to the server (the "payload") and what data the server sent back in its response. This is invaluable for debugging issues where the data seems incorrect. * **Performance Monitoring:** You can see how long each request takes and identify slow operations.

Pro tip: Use the filter buttons (XHR, JS, CSS, etc.) to show only the type of requests you're interested in. For Odoo development, you'll often want to filter by "XHR" to see only the data requests.

The Sources Tab: The Ultimate Debugger

The **Sources** tab is the most powerful, and perhaps most intimidating, tool. It allows you to pause your code mid-execution and inspect everything.

What it's for: * **Setting Breakpoints:** You can pick a line in your JavaScript code and set a "breakpoint." When the browser gets to that line, it will pause the entire application. * **Inspecting State:** While paused, you can hover over any variable to see its current value. You can see the entire call stack (which functions called which other functions) and step through your code one line at a

time. * **Live Code Editing:** In some cases, you can even edit the code directly in the browser and see the changes immediately.

Pro tip: You can set conditional breakpoints that only pause when certain conditions are met. Right-click on a line number and select "Add conditional breakpoint."

Your Code Editor & Odoo Setup

While the browser is for inspecting the *result* of your code, your code editor is where you write it.

Visual Studio Code: Your Development Command Center

Visual Studio Code (VS Code) is the industry standard for modern web development. It's free, powerful, and has excellent support for JavaScript, Python, and XML—all languages you'll use in Odoo development.

Essential Extensions for Odoo Development

Here are the must-have VS Code extensions that will make your Odoo/OWL development much more efficient:

For JavaScript/OWL: * **JavaScript (ES6) Code Snippets:** Quick shortcuts for common JavaScript patterns * **Auto Rename Tag:** Automatically renames paired HTML/XML tags

> **Note:** Bracket pair colorization is now built into VS Code (Settings → "Bracket Pair Colorization")—no extension needed.

For Python/Odoo: * **Python:** Official Python extension with syntax highlighting and debugging * **Pylance:** Advanced Python language server for better code intelligence * **autoDocstring:** Automatically generates Python docstrings

For XML/QWeb: * **XML Tools:** Formatting and validation for XML files * **Auto Close Tag:** Automatically closes XML/HTML tags * **XML Language Support:** Enhanced XML editing features

General Productivity: * **GitLens:** Supercharged Git capabilities * **Material Icon Theme:** Better file icons for different file types * **Thunder Client:** API testing directly in VS Code (great for testing Odoo RPC calls)

Odoo Server Configuration

To make Odoo automatically reload your JavaScript and CSS changes without needing a manual server restart, you should run Odoo with the `--dev=all` flag. This is crucial for an efficient development workflow.

```
./odoo-bin -c your_config_file.conf --dev=all
```

The `--dev=all` flag enables: * Automatic reloading of Python code * Automatic reloading of JavaScript and CSS assets
* Automatic reloading of XML views and templates * Enhanced error messages and debugging information

---

Hands-On Practice: Getting Comfortable with the Tools

Let's put these tools to work with some practical exercises. These will help you become comfortable with the DevTools before we start building OWL components.

Exercise 1: Console Exploration

1. **Open any Odoo instance** (or any website) in your browser
2. **Open the DevTools** by pressing `F12` or right-clicking and selecting "Inspect"
3. **Go to the Console tab**
4. **Try these commands** (type each one and press Enter):

```
console.log("Hello, OWL world!");
console.warn("This is a warning message");
console.error("This is an error message");
console.table([{name: "Alice", age: 25}, {name: "Bob", age: 30}]);
```

5. **Notice how different console methods** display information in different ways

Exercise 2: Element Inspection

1. **Navigate to an Odoo form view** (like editing a contact or product)
2. **Right-click on any button** and select "Inspect"
3. **Observe the HTML structure** in the Elements tab
4. **Try changing the button text** by double-clicking on the text in the HTML and typing something new
5. **Add a CSS style** by clicking on the "Styles" panel and adding a new property like `background-color: red;`
6. **Watch the button change** in real-time

Exercise 3: Network Monitoring

1. **Open the Network tab** before performing any actions
2. **Click "Save" on any Odoo form** (like editing a contact)
3. **Watch the requests appear** in the Network tab
4. **Click on one of the requests** to see the details
5. **Look at the "Payload" or "Request" tab** to see what data was sent
6. **Look at the "Response" tab** to see what the server sent back

Exercise 4: Setting Your First Breakpoint

1. **Find any JavaScript file** in the Sources tab (look for `.js` files)
2. **Click on a line number** to set a breakpoint (a red dot will appear)
3. **Perform an action** that would trigger that code
4. **When the code pauses**, hover over variables to see their values
5. **Use the controls** (play, step over, step into) to control execution

Exercise 5: VS Code Setup Check

1. **Install VS Code** if you haven't already
2. **Install at least 3 extensions** from the list above
3. **Open an Odoo addon folder** in VS Code

4. **Try opening a Python file** and verify syntax highlighting works
5. **Try opening an XML file** and verify the XML extension is working

---

Troubleshooting Common Setup Issues

DevTools won't open: Try different methods—F12, Ctrl+Shift+I (Windows/Linux), or Cmd+Option+I (Mac).

No syntax highlighting in VS Code: Make sure you've installed the appropriate language extensions and that VS Code recognizes the file type (check the bottom-right corner).

Odoo not reloading changes: Verify you're running with `--dev=all` and check the console for any error messages.

Can't find JavaScript files in DevTools: Look for your files under the "Sources" tab, often in a folder structure that mirrors your Odoo addon.

With your browser's DevTools and a properly configured editor and server, you have a professional-grade workshop. The hands-on exercises above will help you become comfortable with these tools before we dive into JavaScript fundamentals. Remember: the more comfortable you are with debugging and inspection, the more effective and efficient you will be as a developer.

TL;DR: Your browser's DevTools (Console, Sources, Network) plus a properly configured editor and `--dev=all` server are the non-negotiable baseline toolkit for any OWL debugging you'll do in this book — and in real projects.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch2_ex1`. Installation instructions are in the repository README.

What's Next?

In the next chapter, we'll start building the JavaScript knowledge you need to create powerful OWL components.

Chapter 3: Modern JavaScript Fundamentals, Part I

> **Why this chapter?** I still see junior developers submit code with `var` and manual `for` loops on OWL projects — it works, but it fights the language OWL was designed around, and it shows up in every code review.
>
> **What is Odoo trying to solve with this?** OWL 2.0 (and Odoo's own JS codebase) assumes you're comfortable with modern JavaScript — `let`/`const`, arrow functions, template literals, and array methods aren't optional extras, they're the baseline the framework's own source code is written in.
>
> **Real-world application:** Every `.map()`/`.filter()` you'll write to transform ORM results into something a template can render, and every template literal you'll use to build a dynamic class or message, comes straight from this chapter.

Now that you know your tools, it's time to learn the language. Modern JavaScript (often called ES6 or newer) introduced powerful features that are now the standard for web development. OWL is built entirely on these modern concepts.

This chapter and the next are your JavaScript boot camp. We will focus on the absolute essentials you'll use every day when writing OWL components. To keep things clear, all the examples here are plain JavaScript, with no Odoo or OWL code involved. Let's build the foundation.

Variables Re-imagined: `let` and `const`

For years, JavaScript had only one way to declare a variable: `var`. It was confusing and had strange behaviors. Modern JavaScript fixed this by giving us two new, much more predictable keywords: `let` and `const`.

`let` is for variables whose value might need to change later. **`const`** is for constants—variables whose value, once assigned, will never change.

Rule of thumb: Always use `const` by default. Only use `let` if you know you will need to reassign the variable. You will almost never need to use the old `var`.

```
// Use let for a variable that will change.
let score = 0;
console.log(score); // Outputs: 0

score = 10; // This is allowed.
console.log(score); // Outputs: 10

// Use const for a value that should not change.
```

```
const playerName = "Alice";
console.log(playerName); // Outputs: "Alice"

// If you try to change a const, you get an error.
// playerName = "Bob"; // This would cause an error!
```

The main advantage of `let` and `const` is that they have "block scope," meaning they only exist within the curly braces `{}` they are defined in. This makes your code much easier to reason about and prevents many common bugs.

```
if (true) {
    const message = "Inside the block";
    console.log(message); // Works fine
}

// console.log(message); // Error! message doesn't exist outside the block
```

Arrow Functions: A Cleaner Syntax

Functions are the building blocks of any application. Arrow functions (`=>`) provide a shorter, cleaner way to write them.

Here's a traditional function:

```
function add(a, b) {
    return a + b;
}
```

And here is the exact same function written as an arrow function:

```
const add = (a, b) => {
    return a + b;
};
```

Even better, if your function is just one line, you can make it even shorter. The `return` is implicit:

```
const subtract = (a, b) => a - b;

console.log(subtract(10, 4)); // Outputs: 6
```

For functions with just one parameter, you can omit the parentheses:

```
const double = x => x * 2;
const greet = name => `Hello, ${name}!`;

console.log(double(5)); // Outputs: 10
console.log(greet("Alice")); // Outputs: Hello, Alice!
```

You will see arrow functions used everywhere in modern JavaScript and OWL code because they are concise and help avoid some tricky issues with the `this` keyword.

Template Literals: Building Strings with Power

One of the most useful features of modern JavaScript is **template literals**, written with backticks (`` ` ``) instead of regular quotes. They make creating dynamic strings much easier and more readable.

The Old Way: String Concatenation

Before template literals, building dynamic strings was clunky:

```
const name = "John Smith";
const age = 35;
const department = "Sales";

// The old, painful way
const message = "Hello, " + name + "! You are " + age + " years old and work in " +
    ↪  department + ".";
console.log(message);
// Outputs: Hello, John Smith! You are 35 years old and work in Sales.
```

The Modern Way: Template Literals

With template literals, you use backticks and `${}` to embed expressions directly in the string:

```
const name = "John Smith";
const age = 35;
const department = "Sales";

// The modern, clean way
const message = `Hello, ${name}! You are ${age} years old and work in ${department}.`;
console.log(message);
// Outputs: Hello, John Smith! You are 35 years old and work in Sales.
```

Why Template Literals Are Perfect for OWL

Template literals are especially useful in OWL development for several reasons:

1. Dynamic CSS Classes:

```
const isActive = true;
const priority = "high";

const cssClass = `task-item ${isActive ? 'active' : 'inactive'} priority-${priority}`;
console.log(cssClass); // Outputs: task-item active priority-high
```

2. Building URLs for RPC calls:

```
const recordId = 123;
const model = "res.partner";

const url = `/web/dataset/call_kw/${model}/read`;
const searchUrl = `/web/dataset/search_read?model=${model}&ids=[${recordId}]`;
```

3. Multi-line strings (great for debugging):

```
const debugInfo = `
    Component State:
    - Name: ${this.state.name}
    - Active: ${this.state.isActive}
    - Records: ${this.state.records.length}
`;
console.log(debugInfo);
```

4. Dynamic HTML generation (though you'll use templates for this in OWL):

```
const createNotification = (type, message) => {
    return `<div class="alert alert-${type}">
        <strong>${type.toUpperCase()}:</strong> ${message}
    </div>`;
};
```

Template Literals vs. Regular Strings

Here's a comparison to show when to use each:

```
// Use regular strings for static text
const staticMessage = "Welcome to Odoo";
const errorCode = "ERR_404";

// Use template literals when you need dynamic content
const welcomeMessage = `Welcome back, ${userName}!`;
const apiEndpoint = `${baseUrl}/api/v1/${resourceType}/${id}`;
const debugOutput = `Processing record ${recordId} at ${new Date()}`;

// Use template literals for multi-line content
const emailTemplate = `
    Dear ${customerName},

    Your order #${orderNumber} has been ${status}.

    Best regards,
    The Odoo Team
`;
```

Understanding Objects: The Cornerstone of Data

An object is a collection of related data and functionality, stored as key-value pairs. This is the most common way you will structure your data in JavaScript. Think of it like a contact card.

```
const contact = {
    // key: value
    name: "John Smith",
    email: "john@example.com",
    age: 35,
    isClient: true,

    // Objects can also have functions (called methods)
    sayHello() {
        console.log("Hello!");
    }
};

// You can access data using dot notation.
console.log(contact.name); // Outputs: "John Smith"

// And call methods the same way.
contact.sayHello(); // Outputs: "Hello!"
```

You can also access properties using bracket notation, which is useful when the property name is

dynamic:

```
const propertyName = "email";
console.log(contact[propertyName]); // Outputs: "john@example.com"

// This is especially useful with template literals
const fieldName = "name";
const value = contact[fieldName];
const message = `The ${fieldName} field contains: ${value}`;
```

Objects are fundamental to OWL because a component's `state` is always an object.

Mastering Arrays: Working with Lists

An array is an ordered list of items. It could be a list of numbers, strings, or even other objects.

```
const tasks = [
    { id: 1, text: "Write chapter 3", completed: true },
    { id: 2, text: "Create code examples", completed: false },
    { id: 3, text: "Drink coffee", completed: false }
];
```

You will constantly need to transform or filter arrays to display them in your UI. Modern JavaScript gives us powerful methods to do this without writing complicated loops. Here are the three you will use most often:

`.map()` - To Transform an Array

The `.map()` method creates a new array by transforming every item in the original array. It's perfect for when you want to create a list of UI elements from a list of data.

```
// Let's get an array of just the task descriptions.
const taskTexts = tasks.map(task => task.text);

console.log(taskTexts);
// Outputs: ["Write chapter 3", "Create code examples", "Drink coffee"]

// Using template literals with map for more complex transformations
const taskSummaries = tasks.map(task => {
    const status = task.completed ? "Done" : "Pending";
    return `${task.text} - ${status}`;
});

console.log(taskSummaries);
// Outputs: ["Write chapter 3 - Done", "Create code examples - Pending", ...]
```

`.filter()` - To Select Items from an Array

The `.filter()` method creates a new array containing only the items that pass a certain test.

```
// Let's get only the tasks that are not yet completed.
const incompleteTasks = tasks.filter(task => task.completed === false);

console.log(incompleteTasks);
// Outputs: An array with the "Create code examples" and "Drink coffee" objects.
```

```
// More complex filtering with template literals for logging
const highPriorityTasks = tasks.filter(task => {
    const isIncomplete = !task.completed;
    const isImportant = task.text.includes("coffee"); // Very important!

    if (isIncomplete && isImportant) {
        console.log(`Found important task: ${task.text}`);
        return true;
    }
    return false;
});
```

.find() - To Get a Single Item from an Array

The .find() method returns the first item in an array that passes a test. It's useful for finding a specific object by its ID.

```
// Let's find the task with id 2.
const specificTask = tasks.find(task => task.id === 2);

console.log(specificTask);
// Outputs: { id: 2, text: "Create code examples", completed: false }

// Using find with more complex logic
const findTaskByPartialText = (searchText) => {
    const foundTask = tasks.find(task =>
        task.text.toLowerCase().includes(searchText.toLowerCase())
    );

    if (foundTask) {
        console.log(`Found task: ${foundTask.text}`);
        return foundTask;
    } else {
        console.log(`No task found containing "${searchText}"`);
        return null;
    }
};

findTaskByPartialText("coffee"); // Will find the "Drink coffee" task
```

Putting It All Together: A Real-World Example

Let's combine all these concepts in a practical example that you might encounter in OWL development:

```
// Sample data that might come from an Odoo model
const customers = [
    { id: 1, name: "Acme Corp", email: "info@acme.com", isActive: true, totalOrders: 25
    ↪ },
    { id: 2, name: "Tech Solutions", email: "hello@tech.com", isActive: false,
    ↪ totalOrders: 8 },
    { id: 3, name: "Global Industries", email: "contact@global.com", isActive: true,
    ↪ totalOrders: 42 }
];
```

```
// Find active customers with more than 10 orders
const vipCustomers = customers
    .filter(customer => customer.isActive && customer.totalOrders > 10)
    .map(customer => {
        // Create a summary using template literals
        const status = customer.isActive ? "Active" : "Inactive";
        const summary = `${customer.name} (${customer.email}) - ${status} -
            ↪ ${customer.totalOrders} orders`;

        return {
            ...customer, // The spread operator copies every property of the original
                ↪ object into this new one, so we don't have to list them all by hand
            summary: summary,
            displayName: `VIP: ${customer.name}`
        };
    });

console.log("VIP Customers:");
vipCustomers.forEach(customer => {
    console.log(customer.summary);
});

// This might output:
// VIP Customers:
// Acme Corp (info@acme.com) - Active - 25 orders
// Global Industries (contact@global.com) - Active - 42 orders
```

Common Pitfalls

Confusing `.map()`, `.filter()`, and `.forEach()`. `.map()` and `.filter()` both return a *new* array and are meant to be used with the result (assigned, rendered, chained). `.forEach()` returns `undefined`—it's only for running side effects (like `console.log`) on each item, never for building a new list.

Mutating the original array or object. `tasks.push(...)` or `contact.name = "..."` changes the original data in place. In OWL, mutating a plain object or array directly won't trigger a re-render and can cause subtle bugs elsewhere. Prefer creating new arrays/objects (as `.map()` and `.filter()` already do) or using `useState`, which we'll cover in Chapter 7.

Using `==` instead of `===`. `==` converts types before comparing (`"5" == 5` is `true`), which causes confusing bugs. Always use `===` (and `!==`) unless you have a specific reason not to.

Forgetting that `const` prevents *reassignment*, not mutation. `const contact = {...}` means you can't do `contact = anotherObject`, but you *can* still do `contact.name = "New Name"`. `const` only locks the variable binding, not the object's contents.

Exercises

1. Write an arrow function `formatPrice(amount, currency)` that uses a template literal to return a string like `"$42.50"` (or `"42.50 USD"` if you prefer a different format).
2. Given `const numbers = [4, 15, 8, 23, 16, 42];`, use `.filter()` to get only the numbers greater than 10, then use `.map()` to double each of them. Try writing it as one chained

expression.

3. Given the `tasks` array from this chapter, use `.find()` to locate the task with `id: 3`, and log a message that says whether it's completed or not.

TL;DR: `let`/`const`, arrow functions, template literals, and the `.map()`/`.filter()`/`.find()` array methods are the modern JavaScript vocabulary that OWL's own source code — and every example in this book — is written in.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch3_ex1`. Installation instructions are in the repository README.

What's Next?

Mastering `let`, `const`, arrow functions, template literals, objects, and these array methods will give you a solid foundation for building powerful and clean OWL components. In the next chapter, we'll explore more advanced concepts — classes, destructuring, modules, and `async`/`await` — that are crucial for understanding how OWL components work internally.

Chapter 4: Modern JavaScript Fundamentals, Part II

> **Why this chapter?** Every OWL component you'll ever write is a `class`, every prop and ORM result you'll handle benefits from destructuring, and every server call is asynchronous — this is the chapter where those four pieces stop being abstract syntax and start being how you'll write code daily.
>
> **What is Odoo trying to solve with this?** OWL's component model (`class ... extends Component`), its module system (`/** @odoo-module **/` and `import`/`export`), and its services (`orm`, `notification`, etc.) are all built on these four JavaScript features — there's no way to write an OWL component without them.
>
> **Real-world application:** The `async`/`await` and `try`/`catch` patterns here are exactly what you'll use every time a component talks to the Odoo server — get this wrong and you get silent failures or "unhandled promise rejection" errors in production.

In the last chapter, we covered the basic building blocks of modern JavaScript. Now, we'll explore four more advanced concepts that are not just important—they are the very structure upon which OWL is built. Understanding classes, destructuring, modules, and asynchronous code is the final step before you can truly understand how an OWL component works.

The `class` Keyword: A Blueprint for Components

An OWL component is, at its core, a JavaScript **class**. A class is a blueprint for creating objects. It bundles data (properties) and the functions that operate on that data (methods) into a single, reusable package.

Think of a `class` as the architectural plan for a house. You can use the same plan to build many identical houses. Each house you build is an "instance" of the class.

Let's define a simple `Dog` class:

```
class Dog {
    // The constructor is a special method that runs when a new instance is created.
    // It's used to set up initial properties.
    constructor(name, breed) {
        this.name = name;
        this.breed = breed;
        this.energy = 100;
    }

    // A method is a function that belongs to the class.
    bark() {
```

```
        console.log(`${this.name} says: Woof!`);
    }

    play() {
        this.energy -= 10;
        console.log(`${this.name} is playing. Energy is now ${this.energy}.`);
    }

    // Methods can return information about the instance
    getInfo() {
        return `${this.name} is a ${this.breed} with ${this.energy} energy.`;
    }
}

// Now, let's create two instances (two different dogs) from our blueprint.
const dog1 = new Dog("Rex", "German Shepherd");
const dog2 = new Dog("Buddy", "Golden Retriever");

dog1.bark();      // Outputs: Rex says: Woof!
dog2.play();      // Outputs: Buddy is playing. Energy is now 90.
console.log(dog1.getInfo()); // Outputs: Rex is a German Shepherd with 100 energy.
```

Extending a Class

The real power comes from the `extends` keyword. It allows you to create a new class that inherits all the properties and methods of an existing one. This is the direct prerequisite for understanding OWL components, which always start with `class MyComponent extends Component`.

Let's create a specialized `Puppy` class that inherits from `Dog`:

```
class Puppy extends Dog {
    constructor(name, breed) {
        // Call the parent constructor with super()
        super(name, breed);

        // Puppies start with less energy
        this.energy = 50;
        this.isTeething = true;
    }

    // We can add a new method that only puppies have.
    chewShoe() {
        if (this.isTeething) {
            console.log(`${this.name} is chewing on a shoe! Bad puppy!`);
        } else {
            console.log(`${this.name} is too old to chew shoes.`);
        }
    }

    // We can also override an existing method.
    bark() {
        console.log(`${this.name} says: Yip! Yip!`); // A puppy's bark is different.
    }

    // We can extend existing methods while still using the parent's functionality
    play() {
```

```
        super.play(); // Call the parent's play method
        if (this.energy < 20) {
            console.log(`${this.name} is getting tired and might nap soon.`);
        }
    }
}

const myPuppy = new Puppy("Leo", "Labrador");

myPuppy.play();      // Inherited from Dog, but with puppy-specific behavior
myPuppy.bark();      // Overridden. Outputs: Leo says: Yip! Yip!
myPuppy.chewShoe(); // New method. Outputs: Leo is chewing on a shoe! Bad puppy!
```

When you write `class MyComponent extends Component`, you are creating a new class that inherits all the powerful features of OWL's base `Component` class, just like our puppy inherited from Dog.

Destructuring Assignment: Extracting Data with Style

Destructuring is a powerful feature that allows you to extract values from arrays and objects into separate variables. It's one of the most elegant features of modern JavaScript and you'll see it everywhere in OWL code.

Object Destructuring: Unpacking Properties

Instead of accessing object properties the old way, destructuring lets you extract multiple values in one clean statement:

```
// The old, repetitive way
const contact = { name: "John Smith", email: "john@example.com", age: 35, department:
    ↪ "Sales" };

const name = contact.name;
const email = contact.email;
const age = contact.age;

// The modern, clean way with destructuring
const contact = { name: "John Smith", email: "john@example.com", age: 35, department:
    ↪ "Sales" };

// Extract multiple properties in one line
const { name, email, age } = contact;

console.log(name);  // "John Smith"
console.log(email); // "john@example.com"
console.log(age);   // 35
```

Advanced Object Destructuring Patterns

Renaming variables during destructuring:

```
const contact = { name: "John Smith", email: "john@example.com" };

// Rename variables during destructuring
const { name: fullName, email: emailAddress } = contact;
```

```
console.log(fullName);     // "John Smith"
console.log(emailAddress); // "john@example.com"
```

Setting default values:

```
const contact = { name: "John Smith" }; // No email property

// Provide defaults for missing properties
const { name, email = "no-email@example.com", age = 0 } = contact;

console.log(email); // "no-email@example.com"
console.log(age);   // 0
```

Nested destructuring:

```
const customer = {
    name: "Acme Corp",
    address: {
        street: "123 Business Ave",
        city: "Commerce City",
        country: "USA"
    },
    contact: {
        phone: "555-0123",
        email: "info@acme.com"
    }
};

// Extract nested properties directly
const {
    name,
    address: { city, country },
    contact: { email }
} = customer;

console.log(`${name} is located in ${city}, ${country}`);
console.log(`Contact them at ${email}`);
```

Array Destructuring: Extracting by Position

Array destructuring extracts values based on their position in the array:

```
const colors = ["red", "green", "blue", "yellow"];

// Extract first few elements
const [primary, secondary, tertiary] = colors;

console.log(primary);   // "red"
console.log(secondary); // "green"
console.log(tertiary);  // "blue"

// Skip elements you don't need
const [first, , third] = colors; // Notice the empty space to skip "green"

console.log(first); // "red"
console.log(third); // "blue"
```

The Rest Pattern: Gathering What's Left

The `...` syntax you'll see inside destructuring is called the **rest pattern**. It collects whatever properties (or array items) weren't already pulled out into their own variable, bundling them into a new object (or array):

```
const customer = { id: 1, name: "Acme Corp", email: "info@acme.com", isActive: true };

// Pull out "id" by itself, and gather everything else into "rest"
const { id, ...rest } = customer;

console.log(id);   // 1
console.log(rest); // { name: "Acme Corp", email: "info@acme.com", isActive: true }
```

This looks identical to the **spread operator** from Chapter 3 (`{ ...customer, summary }`), but the direction is reversed: spread *expands* an object's properties into a new one, while rest *collects* remaining properties into a new one. Which behavior you get depends only on where the `...` appears—on the left side of `=` (a pattern being destructured) it's rest; on the right side (building a new value) it's spread.

Destructuring in Function Parameters

This is where destructuring becomes incredibly powerful for OWL development:

```
// Instead of accessing properties inside the function
const createUserCard = (user) => {
    const name = user.name;
    const email = user.email;
    const isActive = user.isActive;

    return `<div class="user-card ${isActive ? 'active' : 'inactive'}">
        <h3>${name}</h3>
        <p>${email}</p>
    </div>`;
};

// Destructure directly in the function parameters
const createUserCard = ({ name, email, isActive = true }) => {
    return `<div class="user-card ${isActive ? 'active' : 'inactive'}">
        <h3>${name}</h3>
        <p>${email}</p>
    </div>`;
};

// Usage
const userData = { name: "Alice Johnson", email: "alice@example.com", isActive: true };
console.log(createUserCard(userData));
```

Why Destructuring is Perfect for OWL

Destructuring is especially common in OWL for several scenarios:

1. Extracting props in components:

```
// In an OWL component
setup() {
    const { recordId, model, readonly = false } = this.props;
```

```
    // Now you can use recordId, model, and readonly directly
}
```

2. Working with service responses:

```
// orm.read returns an array - grab the first record with array destructuring
const [partner] = await this.orm.read("res.partner", [partnerId], ["name", "email"]);
const { name, email } = partner;
```

3. Event handling:

```
// Extracting data from events
onButtonClick({ target, currentTarget }) {
    // Work directly with target and currentTarget
}
```

4. State management:

```
// Extracting state values
const { isLoading, errorMessage, data = [] } = this.state;
```

Modules: `import` and `export`

As your application grows, you can't keep all your code in one giant file. **Modules** allow you to split your code into separate, reusable files. You can `export` variables, functions, or classes from one file and `import` them into another where they are needed.

This is how the Odoo asset system works. Each JavaScript file is a module.

Basic Export and Import

Let's imagine we have a file called `utils.js` with some helper functions:

```
// File: utils.js

export const PI = 3.14159;

export const add = (a, b) => {
    return a + b;
};

export const formatCurrency = (amount, currency = "USD") => {
    return `${currency} ${amount.toFixed(2)}`;
};

// Default export (one per file)
const calculateTax = (amount, rate = 0.1) => {
    return amount * rate;
};

export default calculateTax;
```

Now, in another file, `main.js`, we can import and use that code:

```
// File: main.js

// Named imports - specify what you want inside curly braces
```

```
import { PI, add, formatCurrency } from "./utils.js";

// Default import - no curly braces needed
import calculateTax from "./utils.js";

// You can also import everything
import * as Utils from "./utils.js";

console.log("The value of PI is:", PI); // Outputs: The value of PI is: 3.14159
console.log("2 + 3 is:", add(2, 3));   // Outputs: 2 + 3 is: 5
console.log(formatCurrency(99.99));     // Outputs: USD 99.99
console.log("Tax on $100:", calculateTax(100)); // Outputs: Tax on $100: 10

// Using the namespace import
console.log("PI from namespace:", Utils.PI);
console.log("Addition:", Utils.add(5, 7));
```

Named Imports Look Like Destructuring

Named imports use curly braces, so they look just like destructuring (though technically they are their own syntax):

```
// Pick exactly the pieces of the library you need
import { useState, useEffect } from "@odoo/owl";
```

Real-World OWL Import Example

Here's what a typical OWL component file looks like:

```
// File: my_component.js

// Import the base Component class and hooks
import { Component, useState, useService } from "@odoo/owl";

// Import utility functions from other files
import { formatDate, validateEmail } from "./utils.js";

// Define and export our component
export class MyComponent extends Component {
    static template = "my.Component";

    setup() {
        // useState returns a reactive object (not a [value, setter] pair like React)
        this.state = useState({
            email: "",
            isValid: false
        });

        this.orm = useService("orm");

        // Use imported utility functions
        this.validateAndSetEmail = (email) => {
            this.state.email = email;
            this.state.isValid = validateEmail(email);
        };
    }
```

```
}

// Export for use in other modules
export default MyComponent;
```

When you see `import { Component } from "@odoo/owl";` at the top of an OWL file, you are simply importing the base `Component` class from the OWL library module so you can extend it.

Asynchronous JavaScript: Handling Delays

The toughest but most important topic is **asynchronous** code. Not all tasks are instant. When you ask the Odoo server for data (an RPC call), there's a delay while the request travels over the network and the server processes it.

If JavaScript waited for the response, your entire application—the whole user interface—would freeze. This is called "blocking" code, and it's a terrible user experience.

Asynchronous code allows JavaScript to start a long-running task (like a network request), and then continue with other work. When the task finally finishes, it notifies your code so you can handle the result.

The Old Way: Promises

The original solution to this was an object called a **Promise**. A Promise is an object that represents a future value. You attach `.then()` to handle a successful result and `.catch()` to handle an error.

```
// This is a simplified example of fetching data.
fetch('https://api.example.com/data')
    .then(response => {
        // This code runs when the server responds successfully.
        return response.json(); // Get the data from the response.
    })
    .then(data => {
        // This code runs after the data has been processed.
        console.log("Data received:", data);
    })
    .catch(error => {
        // This code runs if anything went wrong.
        console.error("Failed to fetch data:", error);
    });

console.log("Request sent! Waiting for response...");
```

The Modern Way: `async/await`

While Promises work, chaining `.then()` can get complicated. Modern JavaScript introduced a much cleaner syntax called `async/await`. It lets you write asynchronous code that *looks* like normal, synchronous code.

- `async`: You put this keyword before a function declaration to tell JavaScript it contains asynchronous operations.
- `await`: You use this keyword inside an `async` function to tell JavaScript to pause execution on that line until the Promise resolves, without freezing the UI.

Here is the same example using `async/await`. This is the syntax you will use for all RPC calls in OWL:

```
// We define an async function to hold our logic.
const fetchData = async () => {
    try {
        // The 'await' keyword pauses the function here until the fetch is complete.
        const response = await fetch('https://api.example.com/data');

        // Once the response is back, the function continues.
        const data = await response.json();

        console.log("Data received:", data);

        // You can return the data for use elsewhere
        return data;
    } catch (error) {
        // If any 'await' call fails, the catch block will run.
        console.error("Failed to fetch data:", error);
        throw error; // Re-throw if needed
    }
};

fetchData();
console.log("Request sent! Waiting for response...");
```

Real-World Async Patterns for OWL

Here are common asynchronous patterns you'll use in OWL development:

1. Loading data on component startup:

```
class MyComponent extends Component {
    setup() {
        this.state = useState({
            isLoading: true,
            records: [],
            error: null
        });

        this.loadData();
    }

    async loadData() {
        try {
            // searchRead returns an array of record objects
            const records = await this.orm.searchRead("res.partner", [], ["name",
                ↪  "email"]);
            this.state.isLoading = false;
            this.state.records = records;
        } catch (error) {
            this.state.isLoading = false;
            this.state.error = error.message;
        }
    }
}
```

2. Handling user actions asynchronously:

```
async onSaveClick() {
    const { name, email } = this.state.formData;

    try {
        // Show loading state
        this.state.isSaving = true;

        // Save to server (create returns an array of new ids)
        const [recordId] = await this.orm.create("res.partner", [{ name, email }]);

        // Update UI with success
        this.state.isSaving = false;
        this.state.lastSavedId = recordId;

        // Show notification
        this.notification.add("Record saved successfully!", { type: "success" });

    } catch (error) {
        this.state.isSaving = false;
        this.notification.add(`Error: ${error.message}`, { type: "danger" });
    }
}
```

3. Using destructuring with async functions:

```
async fetchCustomerData(customerId) {
    try {
        // Destructure the response directly
        const [customer] = await this.orm.read("res.partner", [customerId], ["name",
            ↪  "email", "phone"]);

        const { name, email, phone = "No phone provided" } = customer;

        return {
            displayName: `${name} (${email})`,
            contactInfo: phone,
            isComplete: email && phone !== "No phone provided"
        };

    } catch (error) {
        console.error(`Failed to fetch customer ${customerId}:`, error);
        return null;
    }
}
```

This `async/await` syntax with a `try...catch` block is the standard, modern way to handle any operation that involves a delay, and it's fundamental to communicating with the Odoo server from your OWL components.

Putting It All Together: A Complete Example

Let's combine all the concepts from this chapter in a realistic OWL-style example. We'll build it up method by method rather than all at once.

First, the imports and the `setup()` method, which prepares the component's services and initial state, then kicks off loading data from the server:

```
// File: customer-manager.js

// Module imports with destructuring
import { Component, useState, useService } from "@odoo/owl";
import { formatDate, validateEmail } from "./utils.js";

export class CustomerManager extends Component {
    static template = "CustomerManager";

    setup() {
        // Destructure services
        const orm = useService("orm");
        const notification = useService("notification");

        // Set up state with destructuring assignment
        this.state = useState({
            customers: [],
            selectedCustomer: null,
            isLoading: false,
            filters: { activeOnly: true, hasEmail: false }
        });

        // Store services for use in methods
        this.orm = orm;
        this.notification = notification;

        // Load initial data
        this.loadCustomers();
    }
```

Next, `loadCustomers()` shows the full `async/await` + `try/catch` pattern from this chapter, combined with destructuring to read the active filters and the rest/spread pattern to enrich each record with extra display fields:

```
    async loadCustomers() {
        try {
            // Destructure filters from state
            const { activeOnly, hasEmail } = this.state.filters;

            this.state.isLoading = true;

            // Build domain based on filters
            const domain = [];
            if (activeOnly) domain.push(["active", "=", true]);
            if (hasEmail) domain.push(["email", "!=", false]);

            // searchRead returns an array of record objects
            const records = await this.orm.searchRead(
                "res.partner",
                domain,
                ["name", "email", "phone", "active", "create_date"]
            );
            const totalCount = records.length;
```

```
            // Transform the data using array methods and destructuring
            const processedCustomers = records.map(customer => {
                const { name, email, phone, create_date } = customer;

                return {
                    ...customer, // Spread original properties
                    displayName: `${name}${email ? ` (${email})` : ''}`,
                    formattedDate: formatDate(create_date),
                    isValidEmail: email ? validateEmail(email) : false,
                    hasContact: !!(email || phone)
                };
            });

            // Update state
            this.state.customers = processedCustomers;
            this.state.isLoading = false;

            this.notification.add(
                `Loaded ${totalCount} customers successfully`,
                { type: "success" }
            );

        } catch (error) {
            this.state.isLoading = false;
            console.error("Failed to load customers:", error);
            this.notification.add(
                `Error loading customers: ${error.message}`,
                { type: "danger" }
            );
        }
    }
```

`onCustomerSelect()` is smaller: it just demonstrates using `.find()` with a destructured parameter (`({ id }) => ...`) to locate a record by id:

```
    async onCustomerSelect(customerId) {
        // Find customer using array method with destructuring
        const selectedCustomer = this.state.customers.find(({ id }) => id ===
            ↪ customerId);

        if (selectedCustomer) {
            this.state.selectedCustomer = selectedCustomer;

            // Destructure for logging
            const { name, email } = selectedCustomer;
            console.log(`Selected customer: ${name} (${email || 'no email'})`);
        }
    }
```

Finally, `updateCustomerEmail()` puts the rest pattern to use for real: it pulls `id` out on its own and keeps every other property in `otherProps`, so updating the email doesn't require re-listing every field:

```
    // Class method using async/await and destructuring
    async updateCustomerEmail(customerId, newEmail) {
        try {
            // Validate using imported utility
            if (!validateEmail(newEmail)) {
```

```
                throw new Error("Invalid email format");
            }

            // Update on server
            await this.orm.write("res.partner", [customerId], { email: newEmail });

            // Update local state using array map with destructuring
            this.state.customers = this.state.customers.map(customer => {
                const { id, ...otherProps } = customer;

                if (id === customerId) {
                    return {
                        id,
                        ...otherProps,
                        email: newEmail,
                        isValidEmail: true,
                        hasContact: true
                    };
                }

                return customer;
            });

            this.notification.add("Email updated successfully!", { type: "success" });

        } catch (error) {
            console.error("Failed to update email:", error);
            this.notification.add(`Error: ${error.message}`, { type: "danger" });
        }
    }
}

// Default export
export default CustomerManager;
```

This example demonstrates: - **Classes and inheritance** (extending Component) - **Destructuring** in multiple contexts (imports, state, function parameters, array methods) - **Modules** (import/export) - **Async/await** for server communication - **Template literals** for string formatting - **Modern array methods** (map, find) combined with destructuring

These are the fundamental building blocks that make OWL components powerful and elegant. In the next chapter, we'll start building your first actual OWL component!

Common Pitfalls

Confusing spread and rest. Both use `...`, but spread (on the right of `=`, building a new value: `{ ...customer, email }`) expands properties out, while rest (on the left of `=`, in a destructuring pattern: `const { id, ...rest } = customer`) gathers leftover properties in. If your code isn't doing what you expect, check which side of the `=` your `...` is on.

Calling `this` before `super()` in a subclass. In a class that `extends` another one, `super(...)` must be the first line of the constructor if you use `this` anywhere in it. JavaScript will throw a `ReferenceError` if you try to access `this` first.

Forgetting `await` on an async call. `const records = this.orm.searchRead(...)` (without `await`) gives you a pending `Promise` object, not the actual data. If you're trying to use `.map()` or `.length` on something and it's not working, check whether you forgot an `await`.

Not wrapping `await` in `try/catch`. If a promise rejects (e.g., the server returns an error) and there's no `catch`, you get an "unhandled promise rejection" and your component silently fails instead of showing useful feedback to the user.

Exercises

1. Extend the `Puppy` class from this chapter with a new subclass `ServiceDog` that adds a `task` property (e.g., `"guide"`, `"alert"`) and a method `performTask()` that logs `"${name} is performing: ${task}"`. Make sure it calls `super()` correctly.
2. Given `const order = { id: 5, product: "Widget", qty: 3, customer: { name: "Alice", email: "alice@example.com" } };`, destructure `id`, `qty`, and the nested `name` (renamed to `customerName`) in a single statement.
3. Write an async function `checkStock(productId)` that simulates a network call with `await new Promise(resolve => setTimeout(resolve, 500))`, then returns `{ productId, inStock: true }`. Call it from another async function using `try/catch` and log the result.

TL;DR: OWL components are built entirely on `class` syntax, destructuring, ES modules (`import`/`export`), and `async`/`await` — master these four and there is no OWL syntax left that will surprise you.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch4_ex1`. Installation instructions are in the repository README.

What's Next?

With modern JavaScript in hand, Chapter 5 is where it all pays off: you'll write your first real OWL component, registered as a client action, and see this book's first line of actual framework code run in the browser.

Chapter 5: Hello, OWL! Your First Component

Why this chapter? Every OWL feature you'll learn later builds on the same three-file skeleton you set up here — get this part solid and the rest of the book is additions, not surprises.

What is Odoo trying to solve with this? A predictable, consistent structure for every custom UI piece in the ecosystem, so any Odoo developer can open any addon's `static/src` folder and immediately know where to look.

Real-world application: This is the exact scaffolding you'll reuse on day one of any client project — a custom dashboard widget, a portal form, a kanban card override — before a single line of business logic gets written.

Congratulations on making it through the fundamentals! You now have all the JavaScript knowledge you need to start building with OWL. In this chapter, the theory ends and the practical application begins. We are going to build our very first, classic "Hello, World!" component.

This is the moment where everything you've learned—classes, modules, template literals, destructuring, and the declarative mindset—comes together. By the end of this chapter, you'll have a working OWL component running inside Odoo. Let's get started.

Anatomy of an OWL Component

An OWL component is not a single file. It's a small, self-contained ecosystem typically made of three files, each with a specific responsibility. This separation of concerns is a core principle that keeps your code clean, organized, and maintainable.

Think of it like building a house:

1. **The JavaScript File (`.js`)**: This is the **brain** of your component. It holds the logic, the state, the methods, and defines how the component behaves. This is where all your modern JavaScript skills come into play.
2. **The XML File (`.xml`)**: This is the **skeleton**. It defines the structure and layout of your component's UI using QWeb, Odoo's powerful template language. This determines what users see.
3. **The CSS/SCSS File (`.scss`)**: This is the **skin**. It contains all the styling rules to make your component look professional and match Odoo's design system. (We'll cover styling in detail in later chapters.)

For our first component, we'll focus on the first two: the JavaScript and the XML file. Don't worry about styling for now—let's get the functionality working first.

Setting Up Your Module Structure

Before we write any code, let's establish a proper file structure. Organization matters, especially as your components grow in complexity.

Create the following directory structure in your Odoo module:

```
my_odoo_module/
|-- __manifest__.py
|-- static/
|   |-- src/
|       |-- components/
|           |-- hello_world/
|               |-- hello_world.js
|               |-- hello_world.xml
|               |-- hello_world.scss (we'll add this later)
|-- views/
    |-- hello_world_views.xml
```

Why this structure? - **`static/src/components/`**: This is the standard location for OWL components in Odoo modules - **Component-specific folders**: Each component gets its own folder (`hello_world/`) to keep related files together - **Consistent naming**: Files are named after the component for easy identification

---

The JavaScript File: The Component's Logic

Let's create our first component. Create this file:

`my_odoo_module/static/src/components/hello_world/hello_world.js`

```javascript
/** @odoo-module **/

import { Component } from "@odoo/owl";
import { registry } from "@web/core/registry";

export class HelloWorld extends Component {
    static template = "my_odoo_module.HelloWorld";

    setup() {
        console.log("HelloWorld component is being set up!");
    }
}

// Register the component as a client action so Odoo can display it
registry.category("actions").add("hello_world_app", HelloWorld);
```

Let's break this down line by line:

`/** @odoo-module **/`: This is a special comment that tells the Odoo asset system to treat this file as a module. It's mandatory for all JavaScript files in Odoo modules. Without it, your component won't be recognized.

`import { Component } from "@odoo/owl";`: Here we're using the destructuring import syntax we learned about. We're extracting the `Component` class from the OWL library. This gives us access to all the powerful features of OWL components.

`export class HelloWorld extends Component { ... }`: This line does several important things: - `export`: Makes our class available to other parts of the system (remember modules?) - `class HelloWorld`: Defines our component class with a descriptive name - `extends Component`: Inherits all the OWL functionality (remember class inheritance?)

`static template = "my_odoo_module.HelloWorld";`: This is the crucial link between our JavaScript logic and our XML template. The name must be: - Unique across your entire Odoo instance - Follow the convention: `module_name.ComponentName` - Match exactly what we define in the XML file

`setup() { ... }`: This is OWL 2.0's modern approach to component initialization. It replaces the old constructor pattern and runs when the component is created. The `console.log` will help us see when our component is working.

`registry.category("actions").add("hello_world_app", HelloWorld);`: This registers our component in Odoo's **actions registry** under the tag `hello_world_app`. In a moment, we'll create a client action on the server side whose `tag` field matches this name—that's how Odoo knows which component to render when the action is triggered.

Understanding the Component Lifecycle

Even in this simple example, OWL is doing a lot behind the scenes:

1. **Component Creation**: OWL creates an instance of our `HelloWorld` class
2. **Setup Execution**: Our `setup()` method runs, allowing us to initialize state and services
3. **Template Rendering**: OWL finds the template we specified and renders it
4. **DOM Insertion**: The rendered HTML is inserted into the page

---

The XML File: The Component's Template

Now let's create the template that our JavaScript file references. Create this file:

`my_odoo_module/static/src/components/hello_world/hello_world.xml`

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">

  <t t-name="my_odoo_module.HelloWorld" owl="1">
    <div class="hello-world-component">
      <div class="alert alert-info">
        <h1 class="alert-heading">
          <i class="fa fa-rocket me-2"></i>
          Hello, OWL World!
        </h1>
        <p class="mb-0">
          This is my first OWL component, and it's working perfectly!
        </p>
        <hr class="my-3"/>
        <p class="mb-0">
          <strong>Component Name:</strong> HelloWorld<br/>
          <strong>Template:</strong> my_odoo_module.HelloWorld<br/>
```

```
          <strong>Framework:</strong> OWL 2.0
        </p>
      </div>
    </div>
  </t>

</templates>
```

Let's examine the important parts:

`<templates xml:space="preserve">`: This is the standard wrapper for all Odoo template files. The `xml:space="preserve"` ensures that whitespace in your templates is handled correctly.

`<t t-name="my_odoo_module.HelloWorld" owl="1">`: This is where the magic happens: - `<t>`: A QWeb template tag that doesn't render itself, just its contents - `t-name="my_odoo_module.HelloWorld"`: The unique identifier that **must exactly match** the `static template` property in our JavaScript class - `owl="1"`: This crucial attribute tells Odoo's QWeb engine to process this template using the OWL renderer instead of the legacy renderer

The HTML Content: Inside the `<t>` tag is standard HTML that will become the actual DOM structure of your component. I've used Bootstrap classes (which Odoo includes) to make it look professional right away.

Why This Template Structure?

- **Semantic HTML**: We use proper HTML5 elements and Bootstrap classes
- **Clear visual feedback**: The alert styling makes it obvious when the component renders
- **Debugging information**: We include the component and template names for easy identification
- **Professional appearance**: Even a "Hello World" component should look good in Odoo

---

Registering the Component with Odoo

We've created the component files, but Odoo doesn't know about them yet. We need to register them with the asset system.

Step 1: Update the Module Manifest

Open your module's `__manifest__.py` file and add the new files to the `assets` dictionary:

```
# In __manifest__.py
{
    'name': 'My OWL Hello World Module',
    'version': '1.0',
    'depends': ['base', 'web'],
    'data': [
        'views/hello_world_views.xml',
    ],
    'assets': {
        'web.assets_backend': [
            'my_odoo_module/static/src/components/hello_world/hello_world.js',
            'my_odoo_module/static/src/components/hello_world/hello_world.xml',
        ],
    },
    'installable': True,
```

```
    'auto_install': False,
}
```

Important points about assets: - **web.assets_backend**: This bundle loads on the Odoo backend (where most business applications live) - **Order matters**: JavaScript files should generally come before XML files - **Path accuracy**: Double-check your file paths—typos here will cause silent failures

Step 2: Create a View to Display the Component

Create a view file at `my_odoo_module/views/hello_world_views.xml`:

```
<?xml version="1.0" encoding="utf-8"?>
<odoo>
  <data>

    <!-- Define a client action to display our component.
         The "tag" must match the name used in registry.category("actions").add(...) -->
    <record id="action_hello_world" model="ir.actions.client">
      <field name="name">Hello OWL World</field>
      <field name="tag">hello_world_app</field>
      <field name="target">current</field>
    </record>

    <!-- Add a menu item so users can find your component -->
    <menuitem id="menu_hello_world"
              name="Hello OWL World"
              action="action_hello_world"
              parent="base.menu_administration"
              sequence="100"/>

  </data>
</odoo>
```

Understanding this structure:

Client Action: Unlike regular Odoo views (form, list, etc.), OWL components often use "client actions"—special actions that render custom JavaScript applications instead of standard views.

The `tag` field is the link: When the user triggers this action, Odoo looks up `hello_world_app` in the JavaScript actions registry and finds the `HelloWorld` component we registered there. No server-side template is needed—the component's own QWeb template does the rendering.

Menu Integration: The `<menuitem>` gives users a way to access your component through Odoo's standard menu system.

Testing Your Component

Now for the moment of truth! Let's get your component running:

Step 1: Install/Upgrade Your Module

```
# If this is a new module
./odoo-bin -d your_database -i my_odoo_module --dev=all
```

```
# If you're updating an existing module
./odoo-bin -d your_database -u my_odoo_module --dev=all
```

The `--dev=all` flag is crucial—it ensures that your JavaScript and XML changes are loaded immediately without needing to restart the server.

Step 2: Navigate to Your Component

1. Log into your Odoo instance
2. Go to **Settings** (or wherever you placed your menu item)
3. Click on **"Hello OWL World"**

If everything worked correctly, you should see your beautiful component rendered with the Bootstrap alert styling!

Step 3: Verify in Browser DevTools

Open your browser's DevTools (F12) and:

1. **Check the Console**: You should see the message "HelloWorld component is being set up!"
2. **Inspect the Elements**: You should see the HTML structure from your template
3. **Check the Network tab**: Verify that your `.js` and `.xml` files were loaded

---

Common Issues and Solutions

"Component not found" error: - Check that your `static template` property exactly matches the `t-name` in your XML - Verify that both files are listed in `__manifest__.py` - Make sure you've upgraded your module

"Template not found" error: - Confirm the `owl="1"` attribute is present in your template - Check for typos in the template name - Verify the XML file is valid (no syntax errors)

Component doesn't appear: - Check the browser console for JavaScript errors - Verify your client action and menu item are correctly defined - Make sure you're running with `--dev=all`

Styling looks wrong: - Odoo includes Bootstrap by default, so our classes should work - Try inspecting the elements to see what CSS is being applied - Remember that we haven't added custom CSS yet—that comes in later chapters

---

What You've Accomplished

Congratulations! You've just built your first OWL component and integrated it into Odoo. This seemingly simple example demonstrates several crucial concepts:

Component Architecture: You've seen how JavaScript logic and XML templates work together

Modern JavaScript in Practice: You've used `import`, `export`, `class`, and `extends` in a real application

Odoo Integration: You understand how OWL components fit into Odoo's module system

Development Workflow: You know how to create, register, and test components

Debugging Foundation: You have the basics for troubleshooting when things go wrong

TL;DR: An OWL component is a JS class (`static template` + `setup()`) paired with a QWeb XML template (`t-name`, `owl="1"`) whose names must match exactly; register it with `registry.category("actions").add(tag, Component)` and an `ir.actions.client` record sharing that `tag`.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch5_ex1`. Installation instructions are in the repository README.

Exercises

1. **Personalize the greeting.** Change the component so it greets a specific name (e.g., `"Hello, Ana!"`) by adding a plain JavaScript property in `setup()` and displaying it with `t-esc` in the template.
2. **Break it on purpose.** Rename the `t-name` in your XML file so it no longer matches `static template` in the JavaScript file, reload the page, and read the exact error Odoo shows you in the console. Then fix it. Recognizing this error message will save you time later.
3. **Add a second action.** Add a button that calls a new method logging a message to the console with `console.log`, and verify it in the browser DevTools Console tab from Chapter 2.

What's Next?

Right now, this component is static — it shows the same text no matter what. Chapter 6 makes it dynamic: you'll learn the QWeb directives (`t-esc`, `t-if`, `t-foreach`, and more) that let your template react to data coming from the JavaScript side.

Chapter 6: Rendering with QWeb Templates

> **Why this chapter?** Static HTML gets you a demo; QWeb directives get you a real application. This is the vocabulary you'll be reading and writing in every single template file from here on.
>
> **What is Odoo trying to solve with this?** A single, consistent templating language shared across the entire platform — the same `t-if`/`t-foreach`/`t-esc` you learn here work identically in every Odoo module, so knowledge transfers between projects instead of resetting each time.
>
> **Real-world application:** Every list view, kanban card, and dashboard widget you'll build for a client leans on these five or six directives — get comfortable here and 80% of "how do I show this data" questions answer themselves.

In the last chapter, we created our first component and linked it to a template file. That template contained static HTML, meaning it never changed. The real power of a UI framework, however, is its ability to render dynamic data that responds to user interactions and state changes.

This is where QWeb, Odoo's templating language, truly shines. QWeb templates are not just plain HTML; they are enhanced with special attributes (directives) that can display variables, run loops, make decisions, and handle user interactions. These directives are the bridge between your component's JavaScript logic and the final HTML that the user sees.

To illustrate these concepts comprehensively, let's enhance our `HelloWorld` component with a more realistic state structure that demonstrates all the key QWeb features.

Our Enhanced Component State (in `hello_world.js`):

```
import { Component, markup, useState } from "@odoo/owl";

export class HelloWorld extends Component {
  static template = "my_odoo_module.HelloWorld";

  setup() {
    this.state = useState({
      user: {
        name: "Alice Johnson",
        isAdmin: true,
        avatar: "/web/static/img/user_menu_avatar.png",
        lastLogin: "2024-03-15T10:30:00Z"
      },
      counter: 5,
      tasks: [
        { id: 1, text: "Learn QWeb templating", completed: true, priority: "high" },
        { id: 2, text: "Build a component", completed: false, priority: "medium" },
        { id: 3, text: "Add user interactions", completed: false, priority: "high" },
```

```
        { id: 4, text: "Deploy to production", completed: false, priority: "low" }
      ],
      notifications: [
        { type: "success", message: "Welcome back!" },
        { type: "warning", message: "System maintenance in 1 hour" }
      ],
      settings: {
        theme: "dark",
        showCompleted: true,
        maxItems: 10
      },
      rawHtml: markup("<span>This contains <strong>formatted</strong> HTML
          ↪  content.</span>")
    });
  }

  // Methods we'll use in our template examples
  getTaskColor(priority) {
    const colors = { high: "danger", medium: "warning", low: "info" };
    return colors[priority] || "secondary";
  }

  toggleTask(taskId) {
    const task = this.state.tasks.find(t => t.id === taskId);
    if (task) {
      task.completed = !task.completed;
    }
  }

  incrementCounter() {
    this.state.counter++;
  }
}
```

Now, let's explore how to use this rich `state` data in our XML template with all the essential QWeb features.

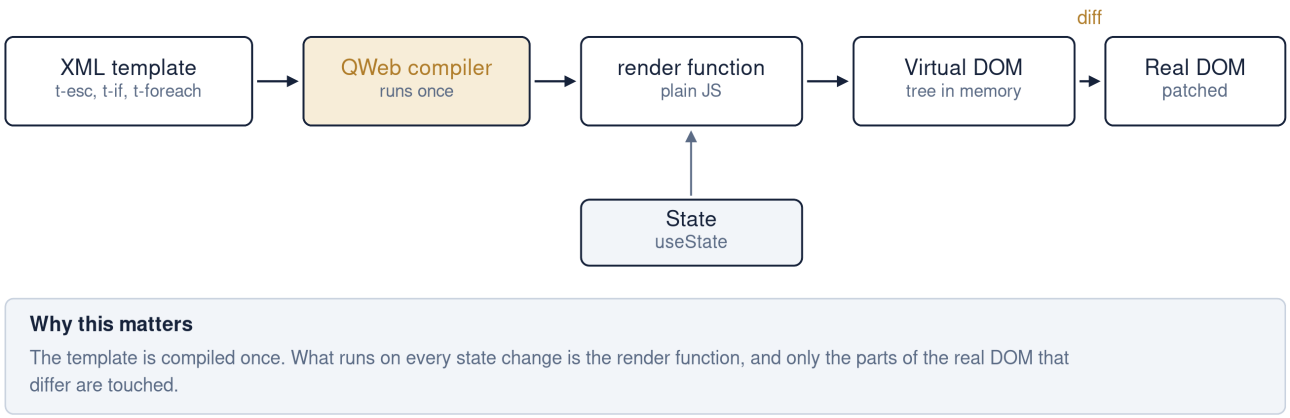


Figure 1: The rendering pipeline: the template is compiled once, the render function runs on every change.

Displaying Data: `t-esc` and `t-out`

The most fundamental task is displaying variables from your state. QWeb provides two primary ways to render data:

`t-esc`: The Safe Default Choice

The `t-esc` directive evaluates an expression and safely renders the result as text. This is your default choice for displaying data, as it protects against Cross-Site Scripting (XSS) attacks by automatically escaping any HTML characters.

```
<!-- Basic data display -->
<div class="user-profile">
  <h2>Welcome, <t t-esc="state.user.name"/>!</h2>
  <p>Current counter: <t t-esc="state.counter"/></p>
  <p>Tasks completed: <t t-esc="state.tasks.filter(t => t.completed).length"/> / <t
    ↪  t-esc="state.tasks.length"/></p>
  <small class="text-muted">Last login: <t t-esc="state.user.lastLogin"/></small>
</div>
```

Result:

```
<div class="user-profile">
    <h2>Welcome, Alice Johnson!</h2>
    <p>Current counter: 5</p>
    <p>Tasks completed: 1 / 4</p>
    <small class="text-muted">Last login: 2024-03-15T10:30:00Z</small>
</div>
```

Key Points about `t-esc`: - Can execute JavaScript expressions, not just simple variables - Automatically escapes HTML to prevent XSS attacks - Perfect for user-generated content or any untrusted data - Supports complex expressions like filtering arrays or calling methods

`t-out`: Rendering Trusted HTML Content

Sometimes you have a variable containing actual HTML that needs to be rendered as markup. Using `t-esc` would display the HTML tags as plain text. For these cases, OWL 2.0 provides `t-out`.

By default, `t-out` escapes its content exactly like `t-esc`. To render actual HTML, the value must be explicitly marked as safe with OWL's `markup()` helper—that's why we wrapped `rawHtml` in `markup()` in the component above. (The old `t-raw` directive from OWL 1.x was removed precisely because it made rendering untrusted HTML too easy.)

Security Warning: Only wrap content in `markup()` when you completely trust it (e.g., HTML generated by your own system or sanitized content). Never use it with user-provided data.

```
<!-- Rendering pre-formatted HTML content -->
<div class="content-area">
  <div class="formatted-content">
    <t t-out="state.rawHtml"/>
  </div>
</div>
```

Result:

```
<div class="content-area">
    <div class="formatted-content">
        <span>This contains <strong>formatted</strong> HTML content.</span>
```

```
    </div>
</div>
```

Dynamic Attributes: `t-att` and `t-attf`

Static attributes are straightforward, but dynamic attributes based on your component's state are where QWeb becomes powerful.

Simple Dynamic Attributes with `t-att`

The `t-att-` prefix followed by the attribute name lets you set attributes dynamically:

```
<!-- Dynamic classes and attributes -->
<div t-att-class="state.settings.theme === 'dark' ? 'theme-dark' : 'theme-light'">
  <img t-att-src="state.user.avatar"
       t-att-alt="state.user.name + ' avatar'"
       class="user-avatar"/>

  <button class="btn"
          t-att-disabled="state.counter >= 10"
          t-on-click="incrementCounter">
    Count: <t t-esc="state.counter"/>
  </button>
</div>
```

Complex Dynamic Attributes with `t-attf`

For more complex attribute values that need string interpolation, use `t-attf-` (attribute format) with `{{}}` placeholders:

```
<!-- String interpolation in attributes -->
<div t-attf-id="user-panel-{{state.user.name.replace(' ', '-').toLowerCase()}}"
     t-attf-data-user-id="{{state.user.id || 'anonymous'}}"
     t-attf-style="background-color: {{state.settings.theme === 'dark' ? '#2c3e50' :
     ↪  '#ecf0f1'}};">

  <div class="status-badge"
       t-attf-class="badge badge-{{state.user.isAdmin ? 'success' : 'secondary'}}">
    <t t-esc="state.user.isAdmin ? 'Administrator' : 'User'"/>
  </div>
</div>
```

Object-Based Class Binding

A powerful pattern for conditional classes:

```
<!-- Object syntax for multiple conditional classes -->
<div t-att-class="{
    'admin-panel': state.user.isAdmin,
    'user-panel': !state.user.isAdmin,
    'theme-dark': state.settings.theme === 'dark',
    'has-notifications': state.notifications.length > 0
}">
```

```
  <h3>Dashboard</h3>
</div>
```

Conditionals: `t-if`, `t-elif`, `t-else`

Control the visibility and rendering of template sections based on your state:

Basic Conditionals

```
<div class="user-status">
  <t t-if="state.user.isAdmin">
    <div class="alert alert-info">
      <i class="fa fa-crown"></i>
      You have administrator privileges.
    </div>
  </t>
  <t t-elif="state.tasks.filter(t => !t.completed).length > 5">
    <div class="alert alert-warning">
      <i class="fa fa-exclamation-triangle"></i>
      You have many pending tasks!
    </div>
  </t>
  <t t-else="">
    <div class="alert alert-success">
      <i class="fa fa-check-circle"></i>
      Everything looks good!
    </div>
  </t>
</div>
```

Nested Conditionals and Complex Logic

```
<!-- More complex conditional logic -->
<div class="task-summary">
  <t t-if="state.tasks.length > 0">
    <h4>Task Overview</h4>
    <t t-if="state.settings.showCompleted">
      <p>Showing all tasks (including completed)</p>
    </t>
    <t t-else="">
      <p>Showing only pending tasks</p>
    </t>

    <!-- Show warning for high-priority incomplete tasks -->
    <t t-if="state.tasks.filter(t => !t.completed and t.priority === 'high').length > 0">
      <div class="alert alert-danger">
        <strong>Attention:</strong> You have high-priority tasks pending!
      </div>
    </t>
  </t>
  <t t-else="">
    <div class="empty-state">
      <p>No tasks yet. Create your first task to get started!</p>
```

```
    </div>
  </t>
</div>
```

Loops: `t-foreach` and `t-as`

Render lists of items from arrays in your state:

Basic Loop Structure

- `t-foreach`: Specifies the array to iterate over
- `t-as`: Names the variable for each item
- `t-key`: Provides a unique key for optimal performance (crucial for OWL's rendering efficiency)

```
<div class="task-list">
  <h3>My Tasks (<t t-esc="state.tasks.length"/>)</h3>
  <div class="list-group">
    <t t-foreach="state.tasks" t-as="task" t-key="task.id">
      <div class="list-group-item d-flex justify-content-between align-items-center">
        <div class="task-content">
          <h5 t-att-class="{ 'text-decoration-line-through': task.completed }">
            <t t-esc="task.text"/>
          </h5>
          <small t-attf-class="badge bg-{{getTaskColor(task.priority)}}">
            <t t-esc="task.priority"/> priority
          </small>
        </div>
        <div class="task-actions">
          <button class="btn btn-sm btn-outline-primary"
                  t-on-click="() => this.toggleTask(task.id)">
            <i t-att-class="task.completed ? 'fa fa-undo' : 'fa fa-check'"></i>
            <t t-esc="task.completed ? 'Undo' : 'Complete'"/>
          </button>
        </div>
      </div>
    </t>
  </div>
</div>
```

Advanced Loop Patterns

Loop with Index

```
<!-- Access the current index with _index -->
<ol class="numbered-list">
  <t t-foreach="state.tasks" t-as="task" t-key="task.id">
    <li>
      <strong>Item #<t t-esc="task_index + 1"/>:</strong>
      <t t-esc="task.text"/>
    </li>
  </t>
</ol>
```

Filtered Loops

```
<!-- Show only incomplete tasks -->
<div class="pending-tasks">
  <h4>Pending Tasks</h4>
  <t t-foreach="state.tasks.filter(t => !t.completed)" t-as="task" t-key="task.id">
    <div class="task-item alert alert-light">
      <t t-esc="task.text"/>
    </div>
  </t>
</div>
```

Nested Loops

```
<!-- Group tasks by priority -->
<div class="tasks-by-priority">
  <t t-foreach="['high', 'medium', 'low']" t-as="priority" t-key="priority">
    <t t-set="priorityTasks" t-value="state.tasks.filter(t => t.priority === priority)"/>
    <t t-if="priorityTasks.length > 0">
      <div class="priority-section">
        <h4 t-attf-class="text-{{getTaskColor(priority)}}">
          <t t-esc="priority.toUpperCase()"/> Priority
          (<t t-esc="priorityTasks.length"/>)
        </h4>
        <t t-foreach="priorityTasks" t-as="task" t-key="task.id">
          <div class="task-item">
            <t t-esc="task.text"/>
          </div>
        </t>
      </div>
    </t>
  </t>
</div>
```

Advanced QWeb Features

Setting Variables with `t-set`

Create local variables within your template:

```
<!-- Calculate and store values for reuse -->
<div class="statistics">
  <t t-set="completedTasks" t-value="state.tasks.filter(t => t.completed)"/>
  <t t-set="completionRate" t-value="Math.round((completedTasks.length /
    ↪  state.tasks.length) * 100)"/>

  <div class="progress mb-3">
    <div class="progress-bar"
         t-attf-style="width: {{completionRate}}%"
         t-attf-aria-valuenow="{{completionRate}}">
      <t t-esc="completionRate"/>% Complete
    </div>
  </div>
```

```
  <p>You've completed <t t-esc="completedTasks.length"/> out of <t
    ↪  t-esc="state.tasks.length"/> tasks!</p>
</div>
```

Calling Component Methods

```
<!-- Methods can be called directly in templates -->
<div class="task-priority-colors">
  <t t-foreach="state.tasks" t-as="task" t-key="task.id">
    <span t-attf-class="badge bg-{{getTaskColor(task.priority)}}">
      <t t-esc="task.text"/>
    </span>
  </t>
</div>
```

Complete Example: Putting It All Together

Here's a comprehensive template that demonstrates all the concepts we've covered. We'll walk through it in three pieces so each part is easier to digest.

Part 1 — the header and notifications. This section renders a personalized greeting, toggling a dark-theme class with the object syntax of `t-att-class`, and switches the badge color depending on `state.user.isAdmin`. Notifications are rendered with `t-foreach`, each one dismissible:

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">

  <t t-name="my_odoo_module.HelloWorld" owl="1">
    <div class="hello-world-component p-4"
         t-att-class="{ 'theme-dark': state.settings.theme === 'dark' }">

      <!-- User Header with Dynamic Content -->
      <div class="user-header d-flex align-items-center mb-4">
        <img t-att-src="state.user.avatar"
             t-att-alt="state.user.name + ' avatar'"
             class="rounded-circle me-3" width="64" height="64"/>
        <div>
          <h2 class="mb-1">Welcome, <t t-esc="state.user.name"/>!</h2>
          <span t-att-class="{
              'badge': true,
              'bg-success': state.user.isAdmin,
              'bg-secondary': !state.user.isAdmin
          }">
            <t t-esc="state.user.isAdmin ? 'Administrator' : 'User'"/>
          </span>
        </div>
      </div>

      <!-- Notifications -->
      <t t-if="state.notifications.length > 0">
        <div class="notifications mb-4">
          <t t-foreach="state.notifications" t-as="notification"
    ↪  t-key="notification_index">
            <div t-attf-class="alert alert-{{notification.type}} alert-dismissible">
```

```
            <t t-esc="notification.message"/>
          </div>
        </t>
      </div>
    </t>
```

Part 2 — the counter and task list (same template, continued). The counter section disables its button once `state.settings.maxItems` is reached. The tasks section uses `t-set` to precompute a completed count once, nests a `t-if` inside `t-foreach` to respect the `showCompleted` setting, and falls back to a `t-else` empty state when there are no tasks:

```
<!-- Counter Section -->
<div class="counter-section mb-4">
  <div class="card">
    <div class="card-body text-center">
      <h3>Counter: <t t-esc="state.counter"/></h3>
      <button class="btn btn-primary"
              t-att-disabled="state.counter >= state.settings.maxItems"
              t-on-click="incrementCounter">
        <i class="fa fa-plus"></i>
        Increment
      </button>
      <t t-if="state.counter >= state.settings.maxItems">
        <p class="text-warning mt-2">Maximum limit reached!</p>
      </t>
    </div>
  </div>
</div>

<!-- Tasks Section -->
<div class="tasks-section">
  <div class="d-flex justify-content-between align-items-center mb-3">
    <h3>Tasks</h3>
    <t t-set="completedCount" t-value="state.tasks.filter(t => t.completed).length"/>
    <span class="badge bg-info">
      <t t-esc="completedCount"/> / <t t-esc="state.tasks.length"/> completed
    </span>
  </div>

  <t t-if="state.tasks.length > 0">
    <div class="task-list">
      <t t-foreach="state.tasks" t-as="task" t-key="task.id">
        <!-- Only show task if settings allow or it's incomplete -->
        <t t-if="state.settings.showCompleted or !task.completed">
          <div class="card mb-2"
               t-att-class="{ 'opacity-50': task.completed }">
            <div class="card-body d-flex justify-content-between align-items-center">
              <div>
                <h5 t-att-class="{ 'text-decoration-line-through': task.completed }">
                  <t t-esc="task.text"/>
                </h5>
                <small t-attf-class="badge bg-{{getTaskColor(task.priority)}}">
                  <t t-esc="task.priority"/> priority
                </small>
              </div>
              <button class="btn btn-sm btn-outline-primary"
```

```
                        t-on-click="() => this.toggleTask(task.id)">
                  <i t-att-class="task.completed ? 'fa fa-undo' : 'fa fa-check'"></i>
                  <t t-esc="task.completed ? 'Undo' : 'Complete'"/>
                </button>
              </div>
            </div>
          </t>
        </t>
      </div>
    </t>
    <t t-else="">
      <div class="empty-state text-center py-5">
        <i class="fa fa-tasks fa-3x text-muted mb-3"></i>
        <h4 class="text-muted">No tasks yet</h4>
        <p class="text-muted">Create your first task to get started!</p>
      </div>
    </t>
  </div>
```

Part 3 — trusted HTML, and closing the template. Finally, the pre-sanitized HTML content is rendered with `t-out`, and we close every tag we opened above:

```
  <!-- Raw HTML Example -->
  <div class="html-content mt-4">
    <h4>Formatted Content:</h4>
    <div class="border p-3 rounded">
      <t t-out="state.rawHtml"/>
    </div>
  </div>
  </div>
  </t>

</templates>
```

Best Practices and Tips

Performance Considerations: - Always use `t-key` in loops for optimal rendering performance - Prefer `t-esc` (or plain `t-out`) over `markup()` unless you specifically need HTML rendering - Complex calculations should be done in JavaScript methods, not in templates

Security: - Never wrap user-provided content in `markup()` - Always validate data before rendering - Use `t-esc` for any dynamic content that could contain HTML

Maintainability: - Keep template logic simple; move complex logic to component methods - Use descriptive variable names in `t-as` directives - Group related template sections with comments

Common Patterns: - Use `t-set` for complex calculations that are used multiple times - Combine `t-if` with `t-foreach` for conditional rendering of lists - Use object syntax for `t-att-class` when dealing with multiple conditional classes

These four fundamental concepts—displaying data, dynamic attributes, conditionals, and loops—are the building blocks of QWeb. By mastering them and understanding their advanced patterns, you can build sophisticated, data-driven user interfaces that react instantly and efficiently to changes in your component's state.

TL;DR: QWeb directives (`t-esc`/`t-out`, `t-att*`, `t-if`/`t-elif`/`t-else`, `t-foreach`+`t-key`) are how your template reads and reacts to your component's data — always pair loops with a stable `t-key`, and never put user content in `markup()`.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch6_ex1`. Installation instructions are in the repository README.

Common Pitfalls

Pitfall 1: Forgetting `t-key` in `t-foreach`

```
<!-- Wrong - no t-key: OWL can't track which item is which across re-renders -->
<t t-foreach="state.tasks" t-as="task">
  <div><t t-esc="task.text"/></div>
</t>

<!-- Correct - a stable, unique key per item -->
<t t-foreach="state.tasks" t-as="task" t-key="task.id">
  <div><t t-esc="task.text"/></div>
</t>
```

Without `t-key`, OWL may reuse the wrong DOM node for the wrong item, causing stale text, lost input focus, or flickering when the list changes.

Pitfall 2: Reaching for `t-raw`

```
<!-- Wrong - t-raw was removed in OWL 2 -->
<div t-raw="state.rawHtml"/>

<!-- Correct - explicitly mark the string as safe, then use t-out -->
<!-- in the component: this.state.rawHtml = markup("<b>Bold</b>"); -->
<div t-out="state.rawHtml"/>
```

`t-out` renders plain text safely by default; it only renders HTML when the value was explicitly wrapped in `markup()` in JavaScript. This two-step requirement exists specifically to prevent accidental XSS.

Pitfall 3: Confusing `t-att-class` with `t-attf-class`

```
<!-- t-att-class expects a JS expression: either a string OR an object of {className:
    ↪ boolean} -->
<div t-att-class="{ 'active': state.isActive }"/>

<!-- t-attf-class expects a plain string with {{ }} placeholders for interpolation -->
<div t-attf-class="badge bg-{{state.color}}"/>
```

Mixing them up—passing an object to `t-attf-class` or a `{{ }}` template to `t-att-class`—will either throw an error or silently render the wrong class.

Pitfall 4: Mutating State That Isn't Reactive Yet

```
<!-- The template only re-renders when a *reactive* property changes. -->
<!-- If state.tasks was never wrapped in useState() in the component's setup(), -->
<!-- pushing to it here has no visible effect on the page. -->
```

If a list or object stops updating the UI, the first thing to check is whether it was created with `useState(...)` in the component (covered in the next chapter)—not whether the template syntax is wrong.

Exercises

1. **Toggle a class by hand.** Add a new `<button>` to the counter section that calls a `toggleHighlight()` method, and use `t-att-class` (object syntax) to add a `bg-warning` class to the card only while `state.highlighted` is `true`.
2. **Fix the missing key.** Take the task list from the "Complete Example" above, remove `t-key="task.id"`, and reload the page after reordering `state.tasks` in the console. Notice what breaks, then restore `t-key` and confirm it's fixed.
3. **Build a filtered loop.** Add a new section that lists only tasks with `priority === "high"`, using a `t-foreach` combined with a `t-if` inside it (as shown in "Advanced Loop Patterns").

What's Next?

Templates can only display what your component gives them. Chapter 7 dives into where that data actually lives — `useState` and OWL's reactivity system — so you understand *why* changing a value automatically triggers everything you just learned about re-rendering.

Chapter 7: State and Reactivity: Making Components Dynamic

Why this chapter? This is the mental model everything else in OWL is built on top of — props, hooks, services, even OWL 3's signals (Chapter 18) are all variations on "change the data, let the framework handle the DOM."

What is Odoo trying to solve with this? Replacing manual DOM synchronization (find the element, update it, hope you didn't miss a spot) with a system where the UI is a guaranteed, automatic function of your data.

Real-world application: Any dashboard, wizard, or interactive form you build professionally lives or dies on getting this right — most "the UI didn't update" support tickets trace back to a misunderstanding of exactly what's covered in this chapter.

This chapter is the most important one in the entire book. If you understand this, you understand the core philosophy of all modern UI frameworks. We are about to uncover the "magic" that makes OWL so powerful: **reactivity**.

Reactivity is the mechanism by which the user interface automatically updates itself when the underlying data changes. You don't tell the UI *how* to change; you simply change the data, and the UI reacts. This is the heart of the declarative paradigm we discussed in Chapter 1.

Think of it like a spreadsheet: when you change a cell value, all the formulas that depend on that cell automatically recalculate and update. OWL brings this same automatic updating behavior to web interfaces.

The Component's Brain: The `setup()` Method

So far, our components have been simple blueprints. To give them memory, behavior, and the ability to change over time, we need to introduce a special method called `setup()`.

The `setup()` method is part of a component's lifecycle. It runs only once, right after the component is created but before it is rendered on the screen. It is the perfect place to:

- Initialize the component's internal data (state)
- Set up services and external connections
- Register event listeners
- Define computed values
- Configure the component's behavior

Let's start with a simple example and build up complexity:

```
// In hello_world.js
import { Component } from "@odoo/owl";

export class HelloWorld extends Component {
  static template = "my_odoo_module.HelloWorld";

  setup() {
    console.log("The component is being set up!");
    console.log("This runs once, before the first render");

    // We'll initialize our state here
    // Set up services here
    // Register event listeners here
  }
}
```

The useState Hook: Giving Your Component Memory

A component's **state** is an object that holds all the data the component needs to remember over its lifetime—things like form input values, loading states, user preferences, or a list of items to display.

But here's the crucial point: a plain JavaScript object isn't enough. If you change a property on a plain object, OWL has no way of knowing that it needs to re-render the UI. The interface would remain frozen, showing stale data.

To create a "reactive" state that OWL can monitor for changes, we use a special tool called the **`useState` hook**.

What is a Hook?

A "hook" in OWL is a function that lets you "hook into" OWL's powerful features like state management, lifecycle events, or services. Hooks can only be called inside the `setup()` method and they give your component superpowers.

Using `useState`

Here's how you create reactive state:

```
// In hello_world.js
import { Component, useState } from "@odoo/owl"; // 1. Import useState

export class HelloWorld extends Component {
  static template = "my_odoo_module.HelloWorld";

  setup() {
    // 2. Create a reactive state object
    this.state = useState({
      counter: 0,
      userName: "Guest",
      isVisible: true,
      items: []
    });

    console.log("Initial state:", this.state);
```

```
  }
}
```

Now, `this.state` is not just a regular object—it's a **reactive proxy**. OWL is actively monitoring it. Any change made to `this.state.counter`, `this.state.userName`, or any other property will automatically trigger a re-render of the component.

The Magic Behind `useState`

When you call `useState`, OWL:

1. **Wraps your object** in a Proxy that intercepts all property access and modifications
2. **Tracks dependencies** between your template and state properties
3. **Schedules re-renders** when state changes, but batches them for performance
4. **Updates only what changed** rather than re-rendering the entire component

This is why OWL apps are so fast and responsive!

The Magic Moment: Modifying State from User Events

Let's see reactivity in action. We'll create a component with multiple interactive elements to demonstrate how state changes drive UI updates.

1. Enhanced JavaScript (`hello_world.js`)

`setup()` seeds one reactive object with everything the template will need: a counter, an editable user name, a theme flag, and a todo list.

```
import { Component, useState } from "@odoo/owl";

export class HelloWorld extends Component {
  static template = "my_odoo_module.HelloWorld";

  setup() {
    this.state = useState({
      counter: 0,
      userName: "Anonymous User",
      theme: "light",
      todos: [
        { id: 1, text: "Learn OWL reactivity", completed: false },
        { id: 2, text: "Build awesome components", completed: false }
      ],
      isEditing: false,
      newTodoText: ""
    });

    console.log("Component initialized with state:", this.state);
  }

  // ... more methods below, still inside the same class
```

These methods only ever mutate `this.state`—never the DOM directly. The counter methods increment, decrement, or reset a single number; the theme method flips a string between two values; the user-name methods toggle an `isEditing` flag and read from an input event:

```
  // Counter methods
  increment() {
    this.state.counter++;
    console.log("Counter incremented to:", this.state.counter);
  }

  decrement() {
    this.state.counter--;
    console.log("Counter decremented to:", this.state.counter);
  }

  reset() {
    this.state.counter = 0;
    console.log("Counter reset to:", this.state.counter);
  }

  // Theme methods
  toggleTheme() {
    this.state.theme = this.state.theme === "light" ? "dark" : "light";
    console.log("Theme changed to:", this.state.theme);
  }

  // User name methods
  startEditing() {
    this.state.isEditing = true;
  }

  saveUserName() {
    if (this.state.userName.trim()) {
      this.state.isEditing = false;
      console.log("User name saved:", this.state.userName);
    }
  }

  onUserNameInput(event) {
    this.state.userName = event.target.value;
  }
```

The todo methods use array methods (`push`, `find`, `splice`) directly on the reactive `todos` array—OWL's reactivity wraps arrays too, so mutating them in place still triggers a re-render. The three `get` accessors at the end are computed properties: they aren't stored in state, they're recalculated from it every time the template reads them:

```
  // Todo methods
  addTodo() {
    if (this.state.newTodoText.trim()) {
      const newTodo = {
        id: Math.max(0, ...this.state.todos.map(t => t.id)) + 1,
        text: this.state.newTodoText.trim(),
        completed: false
      };
      this.state.todos.push(newTodo);
      this.state.newTodoText = "";
      console.log("Added new todo:", newTodo);
    }
  }
```

```
  toggleTodo(id) {
    const todo = this.state.todos.find(t => t.id === id);
    if (todo) {
      todo.completed = !todo.completed;
      console.log("Toggled todo:", todo);
    }
  }

  removeTodo(id) {
    const index = this.state.todos.findIndex(t => t.id === id);
    if (index !== -1) {
      const removed = this.state.todos.splice(index, 1)[0];
      console.log("Removed todo:", removed);
    }
  }

  onNewTodoInput(event) {
    this.state.newTodoText = event.target.value;
  }

  // Computed properties (calculated from state)
  get completedCount() {
    return this.state.todos.filter(todo => todo.completed).length;
  }

  get remainingCount() {
    return this.state.todos.length - this.completedCount;
  }

  get allCompleted() {
    return this.state.todos.length > 0 && this.completedCount ===
        ↪  this.state.todos.length;
  }
}
```

2. Comprehensive Template (`hello_world.xml`)

We'll look at this template in three pieces. First, the header: it swaps between a read-only greeting and an editable input based on `state.isEditing`, plus a theme-toggle button:

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">

  <t t-name="my_odoo_module.HelloWorld" owl="1">
    <div class="hello-world-app p-4"
         t-att-class="{ 'bg-dark text-white': state.theme === 'dark' }">

      <!-- Header with User Name -->
      <div class="header-section mb-4">
        <div class="d-flex justify-content-between align-items-center">
          <div>
            <t t-if="!state.isEditing">
              <h2 class="mb-1">
                Hello, <t t-esc="state.userName"/>!
                <button class="btn btn-sm btn-outline-secondary ms-2"
                        t-on-click="startEditing">
```

```
            <i class="fa fa-edit"></i>
          </button>
        </h2>
      </t>
      <t t-else="">
        <div class="input-group mb-2">
          <input type="text"
                 class="form-control"
                 t-att-value="state.userName"
                 t-on-input="onUserNameInput"
                 placeholder="Enter your name"/>
          <button class="btn btn-success" t-on-click="saveUserName">
            <i class="fa fa-check"></i> Save
          </button>
        </div>
      </t>
    </div>

    <!-- Theme Toggle -->
    <button class="btn"
            t-att-class="state.theme === 'dark' ? 'btn-light' : 'btn-dark'"
            t-on-click="toggleTheme">
      <i t-att-class="state.theme === 'dark' ? 'fa fa-sun' : 'fa fa-moon'"></i>
      <t t-esc="state.theme === 'dark' ? 'Light' : 'Dark'"/> Theme
    </button>
  </div>
</div>
```

Next, the counter (same template, continued). Notice the three-way `t-att-class` on the number itself, and the `t-if`/`t-elif`/`t-else` chain that picks a different message depending on whether the count is zero, positive, or negative:

```
<!-- Counter Section -->
<div class="counter-section mb-4">
  <div class="card" t-att-class="{ 'bg-secondary': state.theme === 'dark' }">
    <div class="card-body text-center">
      <h3 class="display-4 mb-3">
        <span t-att-class="{
            'text-success': state.counter > 0,
            'text-danger': state.counter < 0,
            'text-muted': state.counter === 0
        }">
          <t t-esc="state.counter"/>
        </span>
      </h3>

      <div class="btn-group" role="group">
        <button class="btn btn-danger" t-on-click="decrement">
          <i class="fa fa-minus"></i> Decrease
        </button>
        <button class="btn btn-secondary"
                t-att-disabled="state.counter === 0"
                t-on-click="reset">
          Reset
        </button>
        <button class="btn btn-success" t-on-click="increment">
```

```
            <i class="fa fa-plus"></i> Increase
          </button>
        </div>

        <!-- Dynamic messages based on counter value -->
        <div class="mt-3">
          <t t-if="state.counter === 0">
            <p class="text-muted mb-0">Start counting!</p>
          </t>
          <t t-elif="state.counter > 0">
            <p class="text-success mb-0">
              <t t-if="state.counter === 1">You've clicked once!</t>
              <t t-else="">You've clicked <t t-esc="state.counter"/> times!</t>
            </p>
          </t>
          <t t-else="">
            <p class="text-danger mb-0">Going negative: <t
    ↪  t-esc="Math.abs(state.counter)"/> below zero</p>
          </t>
        </div>
      </div>
    </div>
  </div>
```

Finally, the todo list and a small debug panel, then we close every tag opened above. The list combines `t-foreach` with a nested `t-if` (for the "all completed" banner) and a `t-else` for the empty state:

```
  <!-- Todo List Section -->
  <div class="todo-section">
    <div class="d-flex justify-content-between align-items-center mb-3">
      <h3>Todo List</h3>
      <div class="todo-stats">
        <span class="badge bg-primary me-2">
          <t t-esc="remainingCount"/> remaining
        </span>
        <span class="badge bg-success">
          <t t-esc="completedCount"/> completed
        </span>
      </div>
    </div>

    <!-- Add New Todo -->
    <div class="add-todo mb-3">
      <div class="input-group">
        <input type="text"
               class="form-control"
               placeholder="What needs to be done?"
               t-att-value="state.newTodoText"
               t-on-input="onNewTodoInput"
               t-on-keyup.enter="addTodo"/>
        <button class="btn btn-primary"
                t-att-disabled="!state.newTodoText.trim()"
                t-on-click="addTodo">
          <i class="fa fa-plus"></i> Add Todo
        </button>
      </div>
```

```
    </div>

    <!-- Todo List -->
    <t t-if="state.todos.length > 0">
      <!-- Success message when all completed -->
      <t t-if="allCompleted">
        <div class="alert alert-success">
          <i class="fa fa-trophy"></i>
          Congratulations! You've completed all your todos!
        </div>
      </t>

      <div class="todo-list">
        <t t-foreach="state.todos" t-as="todo" t-key="todo.id">
          <div class="todo-item card mb-2"
               t-att-class="{ 'opacity-75': todo.completed }">
            <div class="card-body d-flex align-items-center">
              <div class="form-check me-3">
                <input class="form-check-input"
                       type="checkbox"
                       t-att-checked="todo.completed"
                       t-on-change="() => this.toggleTodo(todo.id)"/>
              </div>

              <div class="todo-text flex-grow-1">
                <span t-att-class="{ 'text-decoration-line-through text-muted':
    ↪  todo.completed }">
                  <t t-esc="todo.text"/>
                </span>
              </div>

              <button class="btn btn-sm btn-outline-danger"
                      t-on-click="() => this.removeTodo(todo.id)">
                <i class="fa fa-trash"></i>
              </button>
            </div>
          </div>
        </t>
      </div>
    </t>
    <t t-else="">
      <div class="empty-state text-center py-5">
        <i class="fa fa-clipboard-list fa-3x text-muted mb-3"></i>
        <h4 class="text-muted">No todos yet</h4>
        <p class="text-muted">Add your first todo above to get started!</p>
      </div>
    </t>
  </div>

  <!-- Debug Information -->
  <div class="debug-info mt-4 p-3 border rounded" t-att-class="{ 'border-light':
    ↪  state.theme === 'dark' }">
    <h5>Debug Information</h5>
    <small class="text-muted">
      <strong>State Summary:</strong><br/>
```

```
      Counter: <t t-esc="state.counter"/>,
      Theme: <t t-esc="state.theme"/>,
      Todos: <t t-esc="state.todos.length"/>,
      Editing: <t t-esc="state.isEditing"/>
    </small>
  </div>
  </div>
  </t>

</templates>
```

3. Test the Magic!

Now, upgrade your module and view the component. Here's what you'll experience:

1. **Click the counter buttons**: Watch the number change instantly, along with the color and message
2. **Toggle the theme**: See the entire interface switch between light and dark modes immediately
3. **Edit your name**: Notice how the input field appears and the header updates in real-time
4. **Add todos**: Watch new items appear in the list instantly
5. **Check/uncheck todos**: See the completion stats update automatically
6. **Delete todos**: Watch items disappear and stats recalculate

The remarkable thing: We never wrote a single line of code to update the DOM directly. We only changed the data, and OWL handled all the visual updates automatically.

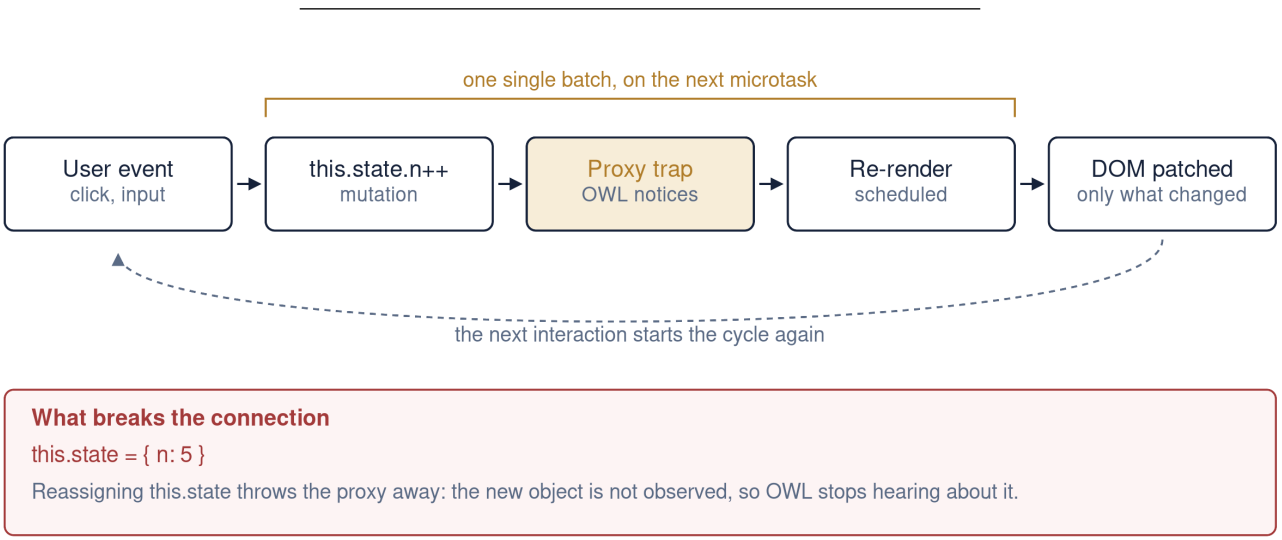


Figure 2: The reactivity loop, and the one assignment that breaks it.

Understanding Reactivity in Depth

What Triggers a Re-render?

OWL re-renders your component when:

1. **State properties change**: `this.state.counter++`
2. **Array mutations**: `this.state.items.push()`, `this.state.items.splice()`
3. **Object property updates**: `this.state.user.name = "New Name"`
4. **Nested property changes**: `this.state.settings.theme = "dark"`

What Doesn't Trigger a Re-render?

```
// This won't work - replacing the state object
this.state = { counter: 5 }; // OWL loses the reactive connection

// This won't work - modifying non-reactive objects
const plainObject = { count: 0 };
plainObject.count++; // Not reactive

// These work - modifying the reactive state
this.state.counter = 5;           // Direct assignment
this.state.items.push(newItem);   // Array mutation
delete this.state.temporaryData;  // Property deletion
```

Performance and Batching

OWL is smart about performance. It doesn't re-render after every single state change. Instead:

1. **Batching**: Multiple state changes in the same JavaScript execution cycle are batched into a single re-render
2. **Efficient Diffing**: Only the parts of the DOM that actually changed are updated
3. **Dependency Tracking**: OWL knows exactly which template parts depend on which state properties

```
// This will only cause ONE re-render, not three
updateMultipleValues() {
  this.state.counter++;        // Change 1
  this.state.userName = "Bob"; // Change 2
  this.state.theme = "dark";   // Change 3
  // OWL batches these and re-renders once
}
```

Advanced State Patterns

Computed Properties with Getters

Create derived values that automatically update when their dependencies change:

```
setup() {
  this.state = useState({
    firstName: "John",
    lastName: "Doe",
    items: [/* ... */]
  });
}

// These getters automatically "recompute" when state changes
get fullName() {
  return `${this.state.firstName} ${this.state.lastName}`;
}

get expensiveItems() {
  return this.state.items.filter(item => item.price > 100);
}
```

```
get totalValue() {
  return this.expensiveItems.reduce((sum, item) => sum + item.price, 0);
}
```

Use in template:

```
<h3>Welcome, <t t-esc="fullName"/>!</h3>
<p>You have <t t-esc="expensiveItems.length"/> expensive items worth $<t
    ↪  t-esc="totalValue"/></p>
```

Complex State Structures

```
setup() {
  this.state = useState({
    user: {
      profile: {
        name: "Alice",
        avatar: "/path/to/avatar.png",
        preferences: {
          theme: "dark",
          language: "en"
        }
      },
      permissions: ["read", "write"]
    },
    application: {
      currentView: "dashboard",
      loading: false,
      errors: []
    },
    data: {
      items: [],
      filters: {
        category: "all",
        sortBy: "name",
        ascending: true
      }
    }
  });
}

// All of these trigger re-renders:
updateUserName(newName) {
  this.state.user.profile.name = newName; // Nested property change
}

changeTheme(theme) {
  this.state.user.profile.preferences.theme = theme; // Deep nested change
}

addError(error) {
  this.state.application.errors.push(error); // Array mutation
}
```

State Validation and Constraints

```
setup() {
  this.state = useState({
    counter: 0,
    email: "",
    isValid: true
  });
}

setEmail(email) {
  this.state.email = email;
  this.state.isValid = this.validateEmail(email);
}

validateEmail(email) {
  const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
  return emailRegex.test(email);
}

setCounter(value) {
  // Constrain counter between 0 and 100
  this.state.counter = Math.max(0, Math.min(100, value));
}
```

Escaping Reactivity: `markRaw` and `toRaw`

`useState` wraps whatever object you give it in a reactive proxy, so OWL can notice when a property is read (to track it) and when it's written (to trigger a re-render). That's usually exactly what you want—but not always.

Sometimes an object shouldn't become reactive at all: a large dataset you'll never mutate in place, or an instance from a third-party library (a charting library, a map widget) that manages its own internal state and doesn't expect to be wrapped in a proxy. Wrapping it anyway wastes performance tracking keys nobody reads, and can even break libraries that do identity checks (`===`) internally, since a proxy is never `===` to the object it wraps.

`markRaw(object)` tells OWL's reactivity system to leave an object alone. If it's ever nested inside a reactive object, it's stored and returned as-is—untouched, unobserved:

```
import { Component, useState, markRaw } from "@odoo/owl";

setup() {
  // chartInstance manages its own internal state; we never want OWL
  // watching its properties or wrapping it in a proxy.
  const chartInstance = markRaw(new ThirdPartyChart());

  this.state = useState({
    counter: 0,
    chart: chartInstance,
  });

  // Mutating chart's own properties won't trigger a re-render-as intended.
  this.state.chart.zoomLevel = 2;
```

```
}
```

`toRaw(reactiveObject)` does the opposite job: given a reactive object (or a piece of one), it returns the original, non-proxied object underneath. This is handy for identity comparisons (`obj === toRaw(state.obj)` can be `true` even when `obj === state.obj` is `false`, since `state.obj` is a proxy) and for debugging—logging a raw object is easier to read than logging a proxy.

```
const original = { name: "Alice" };
const state = useState({ user: original });

console.log(original === state.user);          // false - state.user is a proxy
console.log(original === toRaw(state.user));    // true
```

You won't reach for either of these often, but they're worth knowing about the day you integrate a chart, a map, or any other library that owns its own state.

Common Pitfalls and Solutions

Pitfall 1: Losing Reactivity

```
// Wrong - This breaks reactivity
someMethod() {
  this.state = { items: [1, 2, 3] }; // Completely replaces the reactive object
}

// Correct - Modify properties, don't replace the object
someMethod() {
  this.state.items = [1, 2, 3]; // Modifies the reactive state
}
```

Pitfall 2: Forgetting `useState`

```
// Wrong - plain object: changes are invisible to OWL, the UI never updates
setup() {
  this.state = { counter: 0 };
}

// Correct - reactive object: changes trigger re-renders
setup() {
  this.state = useState({ counter: 0 });
}
```

Pitfall 3: Async State Updates

```
// Race conditions possible
async fetchData() {
  this.state.loading = true;
  const data = await this.service.getData();
  this.state.data = data;
  this.state.loading = false; // What if component was destroyed meanwhile?
}

// Better approach: OWL's status() helper tells you if the component is still alive
import { status } from "@odoo/owl";
```

```
async fetchData() {
  this.state.loading = true;
  try {
    const data = await this.service.getData();
    if (status(this) === "destroyed") return; // Component is gone, stop here
    this.state.data = data;
  } finally {
    if (status(this) !== "destroyed") {
      this.state.loading = false;
    }
  }
}
```

State vs Props: When to Use Which

Use `useState` when: - Data belongs to this component - Data changes based on user interactions in this component - Data is temporary or session-specific - You need to modify the data

Use Props when: - Data comes from a parent component - Data is configuration or settings - Multiple components need to display the same data - The data is read-only for this component

```
// State: Component-owned data
this.state = useState({
  inputValue: "",         // User typing in this component
  isLoading: false,       // This component's loading state
  showModal: false        // This component's UI state
});

// Props: Parent-provided data
static props = {
  userId: { type: Number },       // Configuration from parent
  readonly: { type: Boolean },    // Behavior setting
  initialData: { type: Array }    // Data to display
};
```

Debugging Reactive State

Browser DevTools Tips

1. **Owl DevTools**: Install the Owl browser extension for component and state inspection
2. **Console Logging**: Add strategic `console.log` statements in methods
3. **Breakpoints**: Set breakpoints in state-changing methods
4. **State Snapshots**: Log entire state at key moments

```
// Debugging helper method
debugState(action) {
  console.group(`State Change: ${action}`);
  console.log("Current state:", JSON.parse(JSON.stringify(this.state)));
  console.log("Component:", this);
  console.groupEnd();
}
```

```
// Use in methods
increment() {
  this.state.counter++;
  this.debugState("increment");
}
```

Common Debug Scenarios

```
<!-- Add temporary debugging to templates -->
<div class="debug-panel" style="position: fixed; top: 10px; right: 10px; background:
    ↪  yellow; padding: 10px;">
  <pre t-esc="JSON.stringify(state, null, 2)"/>
</div>

// Method to reset state for testing
resetToInitialState() {
  Object.assign(this.state, {
    counter: 0,
    userName: "Guest",
    todos: [],
    // ... other initial values
  });
}
```

The Declarative Mindset

This chapter introduced you to the fundamental shift in thinking that modern UI frameworks require:

Instead of thinking: "When the user clicks this button, I need to find the counter element in the DOM and update its text"

Think: "When the user clicks this button, I need to update the counter value in my state, and the UI will automatically reflect this change"

This declarative approach makes your code:

- **More predictable**: The UI is always a function of your state
- **Easier to debug**: You can inspect state to understand the UI
- **More maintainable**: No manual DOM manipulation to keep track of
- **More testable**: You can test state changes without involving the DOM

TL;DR: Wrap data in `useState()` to make it reactive; mutate its properties directly (`state.counter++`) rather than reassigning the whole object, and the template re-renders automatically — you change data, OWL updates the DOM.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch7_ex1`. Installation instructions are in the repository README.

Exercises

1. **Add a reset-all button.** Add a button that resets `counter` to `0`, `todos` to an empty array, and `userName` back to `"Anonymous User"`—all in one method—and confirm the whole UI updates at once.

2. **Trigger Pitfall 1 on purpose.** In `reset()`, temporarily replace `this.state.counter = 0` with `this.state = { counter: 0 }` (as shown in Pitfall 1 above). Click the button and observe that the UI silently stops updating. Undo the change and confirm it works again.
3. **Add a computed property.** Add a new getter, `oldestPendingTodo`, that returns the first todo in `state.todos` where `completed` is `false` (or `null` if there isn't one), and display it in the template.

What's Next?

Reactive state is great for data a component owns, but real applications are made of components talking to each other. Chapter 8 covers **props**, the mechanism for passing data from a parent component down to its children.

Chapter 8: Props: Parent-to-Child Communication

Why this chapter? Almost every bug report I get that starts with "the widget shows stale data" traces back to a component reading or mutating data it was never supposed to own. Props are the rule that prevents that class of bug entirely.

What is Odoo trying to solve with this? Odoo's own backend — list views feeding cell widgets, form views feeding field widgets — is built on exactly this parent-to-child contract. Understanding props is what lets you extend those views instead of fighting them.

Real-world application: Any custom dashboard where a filter panel controls what a chart or table displays is this pattern in production: the parent owns the filters, the children just render whatever they're handed.

So far, our components have lived in isolation, managing their own state and rendering their own content. But in a real application, components are nested inside each other, forming a tree-like structure. A `Dashboard` component might contain multiple `Widget` components. A `TodoList` might contain multiple `TodoItem` components. A `UserProfile` might contain `Avatar`, `ContactInfo`, and `PreferenceSettings` components.

This component composition raises a critical question: how does a parent component communicate with its children? How does the `TodoList` tell each `TodoItem` what text to display and whether it's completed? How does a `Dashboard` pass configuration data to its widgets?

The answer is **props** (short for properties).

Props are the primary mechanism for passing data from a parent component down to its children. Think of them as arguments you pass to a function—the parent provides the data, and the child receives and uses it. This creates a predictable, one-way data flow that makes your application easy to understand, debug, and maintain.

Understanding the Component Tree

Before diving into props, let's visualize how components form a hierarchy:

```
Dashboard (Parent)
|-- Header (Child of Dashboard)
|   |-- Logo (Child of Header)
|   |-- UserMenu (Child of Header)
|-- Sidebar (Child of Dashboard)
|   |-- Navigation (Child of Sidebar)
|   |-- QuickActions (Child of Sidebar)
|-- MainContent (Child of Dashboard)
    |-- TodoList (Child of MainContent)
```

```
|   |-- TodoItem (Child of TodoList)
|   |-- TodoItem (Child of TodoList)
|   |-- AddTodoForm (Child of TodoList)
|-- Statistics (Child of MainContent)
```

In this tree: - **Parents** need to pass data down to their **children** - **Children** receive this data as **props** - Data flows in one direction: down the tree

---

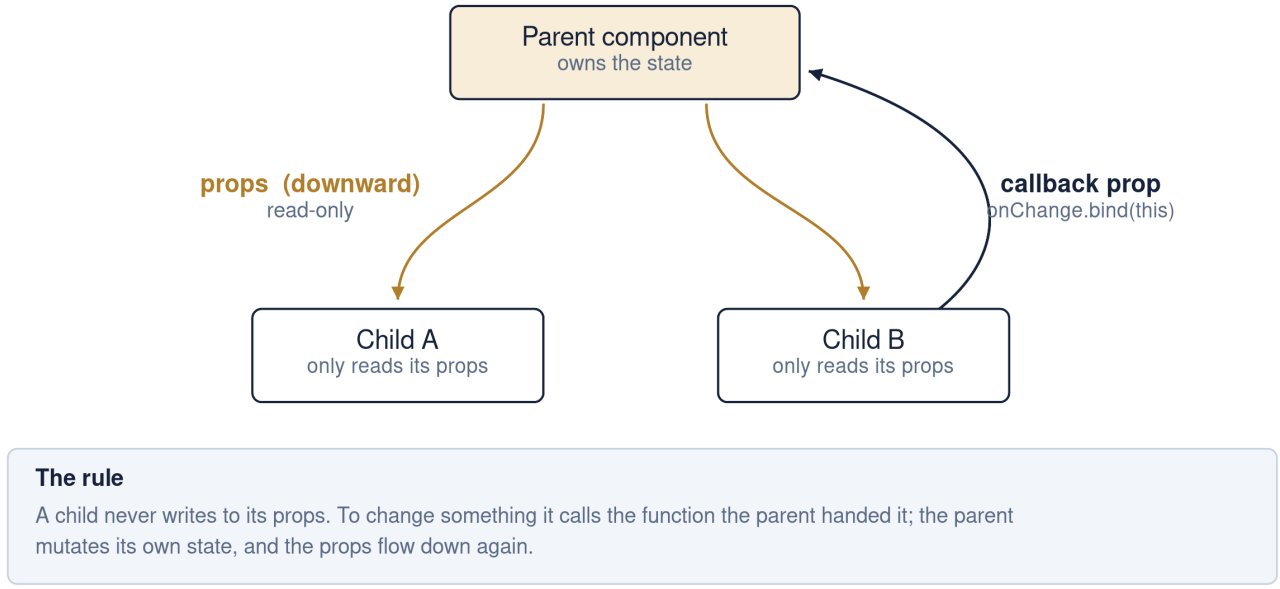


Figure 3: Props flow down, callbacks travel back up.

Defining Props in the Child Component

Before a child component can receive props, it must declare what it expects to receive. This is done using a `static props` definition in the child component's class. Defining props is considered a best practice because:

- **Documentation**: It clearly shows what data the component needs
- **Validation**: OWL can check that the right data types are being passed
- **Development Safety**: Helps catch bugs early in development
- **IDE Support**: Better autocomplete and error detection

Let's create a comprehensive `UserCard` component that demonstrates various prop types:

In `user_card.js`:

```
import { Component } from "@odoo/owl";

export class UserCard extends Component {
  static template = "my_module.UserCard";

  // Comprehensive props definition
  static props = {
    // Required props
    user: {
      type: Object,
```

```
    shape: {
      id: Number,
      name: String,
      email: String,
      avatar: { type: String, optional: true },
      role: String,
      isActive: Boolean,
      lastLogin: { type: String, optional: true }
    }
  },

  // Optional props with defaults
  showDetails: { type: Boolean, optional: true },
  size: { type: String, optional: true }, // 'small', 'medium', 'large'
  theme: { type: String, optional: true },

  // Function props (callbacks)
  onUserClick: { type: Function, optional: true },
  onEditUser: { type: Function, optional: true },
  onDeleteUser: { type: Function, optional: true },

  // Advanced props
  customActions: { type: Array, optional: true },
  permissions: { type: Array, optional: true },

  // Validation with custom validator
  priority: {
    type: String,
    optional: true,
    validate: (value) => ['low', 'medium', 'high'].includes(value)
  }
};

// Default values for optional props
static defaultProps = {
  showDetails: false,
  size: 'medium',
  theme: 'light',
  customActions: [],
  permissions: [],
  priority: 'medium'
};

setup() {
  // Props are available as this.props
  console.log("UserCard props:", this.props);
}

// Method to handle internal card click
handleCardClick() {
  if (this.props.onUserClick) {
    this.props.onUserClick(this.props.user);
  }
}
```

```
  // Method to handle edit action
  handleEdit() {
    if (this.props.onEditUser) {
      this.props.onEditUser(this.props.user.id);
    }
  }

  // Method to handle delete action
  handleDelete() {
    if (this.props.onDeleteUser) {
      this.props.onDeleteUser(this.props.user.id);
    }
  }

  // Computed properties based on props
  get cardSizeClass() {
    const sizeClasses = {
      small: 'user-card-sm',
      medium: 'user-card-md',
      large: 'user-card-lg'
    };
    return sizeClasses[this.props.size] || sizeClasses.medium;
  }

  get priorityBadgeClass() {
    const priorityClasses = {
      low: 'badge-secondary',
      medium: 'badge-warning',
      high: 'badge-danger'
    };
    return priorityClasses[this.props.priority] || priorityClasses.medium;
  }

  get canEdit() {
    return this.props.permissions.includes('edit') && this.props.onEditUser;
  }

  get canDelete() {
    return this.props.permissions.includes('delete') && this.props.onDeleteUser;
  }
}
```

The corresponding template (`user_card.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">

  <t t-name="my_module.UserCard" owl="1">
    <div class="user-card card"
         t-att-class="cardSizeClass"
         t-att-data-theme="props.theme"
         t-on-click="handleCardClick">

      <!-- Card Header -->
      <div class="card-header d-flex justify-content-between align-items-center">
        <div class="user-basic-info d-flex align-items-center">
```

```
      <!-- Avatar -->
      <div class="user-avatar me-3">
        <t t-if="props.user.avatar">
          <img t-att-src="props.user.avatar"
               t-att-alt="props.user.name + ' avatar'"
               class="rounded-circle"
               width="40" height="40"/>
        </t>
        <t t-else="">
          <div class="avatar-placeholder rounded-circle bg-secondary
                      d-flex align-items-center justify-content-center"
               style="width: 40px; height: 40px;">
            <i class="fa fa-user text-white"></i>
          </div>
        </t>
      </div>

      <!-- Name and Status -->
      <div>
        <h5 class="card-title mb-1">
          <t t-esc="props.user.name"/>
          <t t-if="props.priority !== 'medium'">
            <span t-att-class="'badge ms-2 ' + priorityBadgeClass">
              <t t-esc="props.priority"/>
            </span>
          </t>
        </h5>
        <small t-att-class="props.user.isActive ? 'text-success' : 'text-muted'">
          <i t-att-class="props.user.isActive ? 'fa fa-circle' : 'fa
↪ fa-circle-o'"></i>
          <t t-esc="props.user.isActive ? 'Active' : 'Inactive'"/>
        </small>
      </div>
    </div>

    <!-- Action Buttons -->
    <div class="card-actions">
      <t t-if="canEdit">
        <button class="btn btn-sm btn-outline-primary me-1"
                t-on-click.stop="handleEdit">
          <i class="fa fa-edit"></i>
        </button>
      </t>
      <t t-if="canDelete">
        <button class="btn btn-sm btn-outline-danger"
                t-on-click.stop="handleDelete">
          <i class="fa fa-trash"></i>
        </button>
      </t>
    </div>
  </div>

  <!-- Card Body (Optional Details) -->
  <t t-if="props.showDetails">
    <div class="card-body">
```

```
          <div class="user-details">
            <p class="mb-2">
              <strong>Email:</strong>
              <a t-attf-href="mailto:{{props.user.email}}">
                <t t-esc="props.user.email"/>
              </a>
            </p>
            <p class="mb-2">
              <strong>Role:</strong>
              <span class="badge bg-info">
                <t t-esc="props.user.role"/>
              </span>
            </p>
            <t t-if="props.user.lastLogin">
              <p class="mb-2">
                <strong>Last Login:</strong>
                <small class="text-muted">
                  <t t-esc="props.user.lastLogin"/>
                </small>
              </p>
            </t>
          </div>

          <!-- Custom Actions -->
          <t t-if="props.customActions.length > 0">
            <div class="custom-actions mt-3">
              <h6>Quick Actions:</h6>
              <t t-foreach="props.customActions" t-as="action" t-key="action.id">
                <button class="btn btn-sm btn-outline-secondary me-1 mb-1"
                        t-on-click="() => action.handler(props.user)">
                  <i t-att-class="action.icon"></i>
                  <t t-esc="action.label"/>
                </button>
              </t>
            </div>
          </t>
        </div>
      </t>
    </div>
  </t>

</templates>
```

Passing Props from the Parent Component

Now let's create a parent component that uses our `UserCard` and demonstrates various ways to pass props.

Parent Component's JavaScript (`user_dashboard.js`):

```
import { Component, useState } from "@odoo/owl";
import { UserCard } from "../user_card/user_card"; // Import the child component

export class UserDashboard extends Component {
```

```
  static template = "my_module.UserDashboard";
  static components = { UserCard }; // Register the child component

  setup() {
    this.state = useState({
      users: [
        {
          id: 1,
          name: "Alice Johnson",
          email: "alice@example.com",
          avatar: "/web/static/img/user_menu_avatar.png",
          role: "Administrator",
          isActive: true,
          lastLogin: "2024-03-15T10:30:00Z"
        },
        {
          id: 2,
          name: "Bob Smith",
          email: "bob@example.com",
          avatar: null,
          role: "Manager",
          isActive: true,
          lastLogin: "2024-03-14T15:45:00Z"
        },
        {
          id: 3,
          name: "Carol Brown",
          email: "carol@example.com",
          avatar: "/path/to/carol-avatar.jpg",
          role: "Employee",
          isActive: false,
          lastLogin: "2024-03-10T09:15:00Z"
        }
      ],

      viewSettings: {
        showDetails: false,
        cardSize: 'medium',
        theme: 'light'
      },

      userPermissions: ['view', 'edit', 'delete'],
      selectedUserId: null
    });

    // Define custom actions that will be passed as props
    this.customActions = [
      {
        id: 'message',
        label: 'Send Message',
        icon: 'fa fa-envelope',
        handler: this.sendMessageToUser.bind(this)
      },
      {
        id: 'schedule',
```

```
        label: 'Schedule Meeting',
        icon: 'fa fa-calendar',
        handler: this.scheduleMeetingWith.bind(this)
      }
    ];
  }
```

`setup()` only does two things: it puts the user list and view settings into reactive state, and it prepares a `customActions` array whose handlers are pre-bound with `.bind(this)` so they can be called safely from the child later.

Next come the methods that will be handed down to `UserCard` as callback props—this is the "communication flows up through function calls" half of the pattern:

```
  // Event handlers that will be passed as prop callbacks
  onUserClick(user) {
    console.log("User clicked:", user);
    this.state.selectedUserId = user.id;

    // Show user details in a modal or sidebar
    this.showUserDetails(user);
  }

  onEditUser(userId) {
    console.log("Edit user:", userId);
    const user = this.state.users.find(u => u.id === userId);
    if (user) {
      // Open edit modal or navigate to edit form
      this.openEditModal(user);
    }
  }

  onDeleteUser(userId) {
    console.log("Delete user:", userId);
    if (confirm("Are you sure you want to delete this user?")) {
      this.state.users = this.state.users.filter(u => u.id !== userId);
    }
  }

  // Custom action handlers
  sendMessageToUser(user) {
    console.log("Sending message to:", user.name);
    // Open messaging interface
  }

  scheduleMeetingWith(user) {
    console.log("Scheduling meeting with:", user.name);
    // Open calendar scheduling interface
  }
```

The rest is local UI state management—toggling view settings and computing derived lists—none of it involves props at all, which is the point: only the parent needs to know *how* users are stored and filtered.

```
  // Settings change handlers
  toggleDetails() {
    this.state.viewSettings.showDetails = !this.state.viewSettings.showDetails;
```

```
  }

  changeCardSize(size) {
    this.state.viewSettings.cardSize = size;
  }

  toggleTheme() {
    this.state.viewSettings.theme = this.state.viewSettings.theme === 'light' ? 'dark' :
        ↪  'light';
  }

  // Helper methods
  showUserDetails(user) {
    // Implementation for showing user details
    console.log("Showing details for:", user);
  }

  openEditModal(user) {
    // Implementation for opening edit modal
    console.log("Opening edit modal for:", user);
  }

  // Computed properties
  get activeUsers() {
    return this.state.users.filter(user => user.isActive);
  }

  get inactiveUsers() {
    return this.state.users.filter(user => !user.isActive);
  }
}
```

Parent Component's Template (`user_dashboard.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">

  <t t-name="my_module.UserDashboard" owl="1">
    <div class="user-dashboard p-4">

      <!-- Dashboard Header -->
      <div class="dashboard-header mb-4">
        <div class="d-flex justify-content-between align-items-center">
          <div>
            <h2>User Dashboard</h2>
            <p class="text-muted">
              <t t-esc="activeUsers.length"/> active,
              <t t-esc="inactiveUsers.length"/> inactive users
            </p>
          </div>

          <!-- View Controls -->
          <div class="view-controls">
            <div class="btn-group me-3" role="group">
              <button class="btn btn-sm"
                      t-att-class="state.viewSettings.showDetails
```

```
                                ? 'btn-primary' : 'btn-outline-primary'"
                        t-on-click="toggleDetails">
                  <i class="fa fa-info-circle"></i>
                  <t t-esc="state.viewSettings.showDetails ? 'Hide Details' : 'Show
↪  Details'"/>
                </button>
              </div>

              <div class="btn-group me-3" role="group">
                <button class="btn btn-sm"
                        t-att-class="state.viewSettings.cardSize === 'small'
                                ? 'btn-secondary' : 'btn-outline-secondary'"
                        t-on-click="() => this.changeCardSize('small')">
                  Small
                </button>
                <button class="btn btn-sm"
                        t-att-class="state.viewSettings.cardSize === 'medium'
                                ? 'btn-secondary' : 'btn-outline-secondary'"
                        t-on-click="() => this.changeCardSize('medium')">
                  Medium
                </button>
                <button class="btn btn-sm"
                        t-att-class="state.viewSettings.cardSize === 'large'
                                ? 'btn-secondary' : 'btn-outline-secondary'"
                        t-on-click="() => this.changeCardSize('large')">
                  Large
                </button>
              </div>

              <button class="btn btn-sm btn-outline-secondary" t-on-click="toggleTheme">
                <i t-att-class="state.viewSettings.theme === 'dark'
                            ? 'fa fa-sun' : 'fa fa-moon'"></i>
                <t t-esc="state.viewSettings.theme === 'dark' ? 'Light' : 'Dark'"/> Theme
              </button>
            </div>
          </div>
        </div>

        <!-- User Cards Grid -->
        <div class="users-grid">
          <div class="row">
            <t t-foreach="state.users" t-as="user" t-key="user.id">
              <div t-att-class="state.viewSettings.cardSize === 'small' ? 'col-md-4'
                              : state.viewSettings.cardSize === 'large' ? 'col-12'
                              : 'col-md-6'">
                <div class="mb-3">
                  <!-- This is where props are passed! -->
                  <UserCard
                      user="user"
                      showDetails="state.viewSettings.showDetails"
                      size="state.viewSettings.cardSize"
                      theme="state.viewSettings.theme"
                      onUserClick.bind="onUserClick"
                      onEditUser.bind="onEditUser"
                      onDeleteUser.bind="onDeleteUser"
```

```
                    customActions="customActions"
                    permissions="state.userPermissions"
                    priority="user.role === 'Administrator' ? 'high' : 'medium'"
                />
              </div>
            </div>
          </t>
        </div>
      </div>

      <!-- Empty State -->
      <t t-if="state.users.length === 0">
        <div class="empty-state text-center py-5">
          <i class="fa fa-users fa-3x text-muted mb-3"></i>
          <h4 class="text-muted">No users found</h4>
          <p class="text-muted">Add some users to see them here!</p>
        </div>
      </t>

      <!-- Selected User Info (if any) -->
      <t t-if="state.selectedUserId">
        <t t-set="selectedUser" t-value="state.users.find(u => u.id ===
    ↪ state.selectedUserId)"/>
        <div class="selected-user-info mt-4 p-3 bg-light rounded">
          <h5>Selected User: <t t-esc="selectedUser.name"/></h5>
          <p class="mb-0">Click on another user card to select them.</p>
        </div>
      </t>
    </div>
  </t>

</templates>
```

Understanding Prop Passing Syntax

Let's break down the different ways to pass props:

1. Static Values

```
<!-- Passing a string literal -->
<UserCard title="'User Profile'"/>

<!-- Passing a number -->
<UserCard maxItems="10"/>

<!-- Passing a boolean -->
<UserCard readonly="true"/>
```

Important: Notice the single quotes inside double quotes for strings. The attribute value is a JavaScript expression, so `"'Hello'"` evaluates to the string `"Hello"`.

2. Dynamic Values from State

```
<!-- Passing state properties -->
<UserCard user="state.currentUser"/>
<UserCard showDetails="state.viewSettings.showDetails"/>
<UserCard theme="state.theme"/>
```

3. Computed Values

```
<!-- Passing the result of an expression -->
<UserCard priority="user.role === 'Administrator' ? 'high' : 'medium'"/>
<UserCard isSelected="state.selectedUserId === user.id"/>
<UserCard canEdit="state.permissions.includes('edit') and user.isActive"/>
```

4. Function References

```
<!-- Passing method references (callbacks) -->
<!-- The .bind suffix binds the function to the parent component,
     so `this` works correctly when the child calls it -->
<UserCard onUserClick.bind="onUserClick"/>
<UserCard onEditUser.bind="onEditUser"/>
<UserCard onDeleteUser.bind="onDeleteUser"/>
```

5. Objects and Arrays

```
<!-- Passing complex objects -->
<UserCard user="user" customActions="customActions"/>
<UserCard settings="state.userSettings"/>
<UserCard permissions="['read', 'write', 'delete']"/>
```

Advanced Props Patterns

Conditional Props

```
<!-- Only pass certain props under conditions -->
<UserCard
    user="user"
    t-props="{
        showDetails: state.viewSettings.showDetails,
        ...(user.role === 'admin' ? { adminActions: adminActionList } : {}),
        ...(state.currentUser.id === user.id ? { highlight: true } : {})
    }"
/>
```

Spread Props Pattern

```
<!-- Spread an entire object as props -->
<UserCard t-props="user" additionalProp="someValue"/>

<!-- Combine multiple prop sources -->
<UserCard t-props="{
    ...user,
    ...state.cardSettings,
```

```
    onUserClick: onUserClick,
    customClass: 'special-user'
}"/>
```

Props with Defaults

```
// In the child component
static defaultProps = {
    size: 'medium',
    theme: 'light',
    showActions: true
};

// These props will have default values if not provided by parent
```

Validation and Error Handling

```
static props = {
    user: {
        type: Object,
        validate: (user) => {
            if (!user.id || !user.name) {
                throw new Error("User must have id and name");
            }
            return true;
        }
    },
    priority: {
        type: String,
        optional: true,
        validate: (value) => ['low', 'medium', 'high'].includes(value)
    }
};
```

The Golden Rule: One-Way Data Flow

This brings us to the most important principle about props: **A child component should never, ever modify its own props.**

Think of props as a **read-only contract** from the parent. The child can use the data, display it, and make decisions based on it, but it cannot change it.

Why This Rule Exists

1. **Predictability**: You always know where data changes come from (the parent)
2. **Debugging**: Easier to trace data flow and find bugs
3. **Reusability**: Child components remain pure and reusable
4. **Performance**: OWL can optimize rendering when data flow is predictable

What This Means in Practice

```
// NEVER do this in a child component
setup() {
```

```
  // DON'T modify props directly
  this.props.user.name = "Modified Name"; // This breaks the contract!
  this.props.showDetails = true; // This will cause issues!
}

// Instead, use props as read-only data
setup() {
  // Use props to initialize local state if needed
  this.state = useState({
    localShowDetails: this.props.showDetails,
    editedName: this.props.user.name
  });

  // Or create computed properties
  this.displayName = this.props.user.name.toUpperCase();
}
```

When a Child Needs to "Change" Props

If a child component needs to signal that something should change, it should **emit an event** up to the parent. The parent listens for the event and decides whether to update its own state, which then flows back down as new props.

```
// Child component method
requestNameChange(newName) {
  // Don't change props.user.name directly
  // Instead, ask the parent to change it
  this.props.onNameChangeRequest?.(this.props.user.id, newName);
}

// Parent component handles the request
onNameChangeRequest(userId, newName) {
  const user = this.state.users.find(u => u.id === userId);
  if (user) {
    user.name = newName; // Parent updates its own state
    // This will flow back down to the child as new props
  }
}
```

This pattern maintains the one-way data flow while still allowing children to influence parent state through well-defined communication channels.

Props vs State: Decision Framework

Understanding when to use props vs state is crucial for building maintainable components:

Use Props When:

- **Data comes from a parent** component
- **Data is configuration** or settings for the component
- **Multiple components need the same data**
- **Data represents the "inputs" to your component**
- **Data shouldn't be modified** by this component

```
// Props examples
static props = {
  userId: { type: Number },         // ID to display data for
  readonly: { type: Boolean },      // Configuration option
  theme: { type: String },          // App-wide setting
  onSave: { type: Function }        // Callback to parent
};
```

Use State When:

- **Data belongs to this component**
- **Data changes based on user interactions** in this component
- **Data is temporary or UI-specific** (like "is modal open?")
- **Data is derived from user input** in this component

```
// State examples
this.state = useState({
  inputValue: "",             // User typing in this component
  isLoading: false,           // This component's loading state
  showModal: false,           // This component's UI state
  validationErrors: []        // This component's validation state
});
```

Common Props Patterns and Best Practices

1. Callback Props Pattern

```
<!-- Parent provides callbacks for child to communicate back -->
<TodoItem
  todo="todo"
  onToggle="(id) => this.toggleTodo(id)"
  onEdit="(id, newText) => this.editTodo(id, newText)"
  onDelete="(id) => this.deleteTodo(id)"
/>
```

2. Configuration Props Pattern

```
<!-- Parent configures child behavior -->
<DataTable
  data="state.users"
  columns="tableColumns"
  sortable="true"
  filterable="true"
  pageSize="10"
  showPagination="true"
/>
```

3. Render Props Pattern

```
<!-- Parent provides rendering logic -->
<Modal
  isOpen="state.showModal"
  onClose="closeModal"
```

```
  renderContent="() => this.renderModalContent()"
  renderFooter="() => this.renderModalFooter()"
/>
```

4. Compound Component Pattern

```
<!-- Multiple related child components -->
<Card>
  <CardHeader title="'User Profile'" actions="headerActions"/>
  <CardBody content="userDetails"/>
  <CardFooter buttons="footerButtons"/>
</Card>
```

Debugging Props

Common Props Issues and Solutions

Issue 1: Props not updating

```
<!-- Problem: Passing a static value instead of reactive state -->
<UserCard user="staticUserObject"/>

<!-- Solution: Pass reactive state -->
<UserCard user="state.currentUser"/>
```

Issue 2: Props undefined or wrong type

```
// Check props in component setup
setup() {
  console.log("Received props:", this.props);

  // Validate critical props
  if (!this.props.user) {
    console.error("UserCard requires a user prop!");
  }
}
```

Issue 3: Function props not working

```
<!-- Problem: Calling function instead of passing reference -->
<UserCard onUserClick="onUserClick()"/>

<!-- Solution: Pass function reference (bound with .bind) -->
<UserCard onUserClick.bind="onUserClick"/>
```

Props Debugging Template

```
<!-- Temporary debugging section -->
<div class="props-debug"
     style="background: #f8f9fa; padding: 10px; margin: 10px 0; border-radius: 4px;">
  <strong>Props Debug:</strong>
  <pre t-esc="JSON.stringify(props, null, 2)"/>
</div>
```

Performance Considerations

Optimizing Props for Performance

1. **Avoid Creating Objects in Templates**:

```
<!-- Bad: Creates new object every render -->
<UserCard settings="{ theme: 'dark', size: 'large' }"/>

<!-- Good: Store object in component state or method -->
<UserCard settings="cardSettings"/>
```

2. **Memoize Complex Calculations** (*memoization* means caching the result of an expensive computation so it isn't recalculated when the inputs haven't changed):

```
// Calculate expensive props once
get expensiveComputedProp() {
  // Memoize or cache this calculation
  return this.state.data.reduce(...);
}
```

3. **Use Stable Function References**:

```
// Create bound methods once in setup
setup() {
  this.boundHandleClick = this.handleClick.bind(this);
}

<!-- Use the stable reference in template -->
<UserCard onUserClick="boundHandleClick"/>
```

Common Pitfalls

Pitfall 1: Mutating Props Directly

```
// DON'T: this breaks the one-way data flow contract
this.props.user.name = "New Name";
```

Props belong to the parent. If the child needs different data, copy it into local state (`useState`) or ask the parent to change it via a callback prop.

Pitfall 2: Skipping `static props` Validation

Without a `static props` declaration, OWL can't catch a missing or mistyped prop for you—the bug surfaces later as a confusing runtime error instead of a clear message at render time. Always declare `static props` on components that receive data from a parent.

Pitfall 3: Forgetting `.bind` on Callback Props

```
<!-- `this` inside onEditUser will be undefined -->
<UserCard onEditUser="onEditUser"/>

<!-- Correct -->
<UserCard onEditUser.bind="onEditUser"/>
```

Pitfall 4: No `defaultProps` for Optional Props

If a prop is `optional: true` but the component still assumes it has a value (e.g. `this.props.size.toUpperCase()`), a parent that omits it will crash the child. Pair every optional prop with a sensible entry in `static defaultProps`.

Exercises

Exercise 1: Validate and Default

Take the `UserCard` component from this chapter and add a new optional prop `badgeText` (`String`). Give it a default value of `"Member"` via `static defaultProps`, and render it next to the user's name only when `props.showDetails` is true.

Exercise 2: Read-Only Discipline

Write a small child component that receives a `counter` prop (`Number`). Try mutating `this.props.counter` inside a method and confirm (via the browser console) that OWL warns you or that the change doesn't persist. Then fix it by copying the prop into local state with `useState` on `setup()`.

Exercise 3: Callback Prop From Scratch

Build a `RatingWidget` component that receives a `value` prop and an `onRate` callback prop. When the user clicks one of 5 stars, the component should call `this.props.onRate(starIndex)`—it should never modify `this.props.value` itself. Wire it up from a parent that stores the current rating in `useState`.

Props are the fundamental building blocks of component communication in OWL. By mastering props patterns and understanding the one-way data flow principle, you'll build applications that are predictable, maintainable, and scalable.

TL;DR: Props flow one way, parent to child; a child never mutates its own props, and declares `static props`/`static defaultProps` so mistakes surface early instead of as confusing runtime bugs.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch8_ex1`. Installation instructions are in the repository README.

What's Next?

We've covered how data flows down, from parent to child. In the next chapter, we'll learn about the other half of parent-child communication: how children can communicate back to their parents using **callback props**.

Chapter 9: Handling Events: Child-to-Parent Communication

Why this chapter? I've inherited more than one addon where a child widget reached straight into its parent's state to "fix" something, and every one of them turned brittle the moment someone changed the parent. Callback props are the disciplined alternative.

What is Odoo trying to solve with this? OWL removed the old `trigger`/event-bus system from OWL 1 on purpose — Odoo needed child-to-parent communication that's traceable from the parent's own template, not scattered across event listeners you have to hunt down.

Real-world application: A `TodoItem` telling its `TodoList` "delete me" or a table row telling its parent "I was clicked" are the same shape as a form field widget telling the form view "my value changed" — the pattern you'll use constantly once you start building real Odoo UI.

In Chapter 8, we established the golden rule of props: **data flows one way, from parent down to child**. A child component should never modify its own props directly. This creates predictable, maintainable applications where you always know where data changes originate.

But this raises an important question: what happens when a child component needs to communicate back to its parent? What if a user clicks a "Delete" button inside a `TodoItem` component? The child can't delete the data itself because that data belongs to the parent and was passed down as a prop.

This is where the second half of the communication loop comes in: **callback props**.

While data flows down through props, **communication flows up through function calls**. The parent passes a function to the child as a prop; when something happens inside the child (a click, a submit, an error), the child simply calls that function. The parent decides what to do with the information. This completes the data flow cycle while keeping the parent firmly in control of the application state.

> **Coming from OWL 1.x?** The old framework had a `trigger()` mechanism that dispatched custom events up the component tree, which parents listened to with `t-on-event-name` on the component tag. **That mechanism was removed in OWL 2.0.** In OWL 2.0, `t-on-` works only on real DOM elements (for native events like `click` and `input`), and child-to-parent communication is done exclusively with callback props. If you see `this.trigger("some-event")` in old code, that's legacy OWL 1 (we'll cover migrating it in Chapter 17).

Understanding the Communication Flow

Let's visualize how this bidirectional communication works:

```
Parent Component
|-- State: { todos: [...], users: [...] }
|-- Data flows DOWN via props ↓
|-- Callbacks flow DOWN via props ↓
|
Child Component
|-- Receives: data props (read-only) + callback props
|-- User interaction occurs
|-- Child CALLS the callback ↑
|
Parent Component
|-- Callback runs in the parent
|-- Updates its own state
|-- New state flows DOWN as props ↓
```

This pattern ensures that: - **Parents control the data** and business logic - **Children remain pure and reusable** - **Data flow is predictable** and easy to debug - **Components are loosely coupled** and maintainable

---

Basic Callback Pattern

Let's start with a simple example. We'll create a `TodoItem` component that can notify its parent when it needs to be deleted.

The Child Component: TodoItem

TodoItem JavaScript (`todo_item.js`):

```
import { Component } from "@odoo/owl";

export class TodoItem extends Component {
  static template = "my_module.TodoItem";

  static props = {
    todo: {
      type: Object,
      shape: {
        id: Number,
        text: String,
        completed: Boolean,
        priority: { type: String, optional: true },
        dueDate: { type: String, optional: true }
      }
    },
    // Callback props: functions the parent gives us to "call home"
    onDelete: { type: Function },
    onToggle: { type: Function },
    onEdit: { type: Function, optional: true },
    canEdit: { type: Boolean, optional: true },
    canDelete: { type: Boolean, optional: true },
    showPriority: { type: Boolean, optional: true }
  };

  static defaultProps = {
```

```
    canEdit: true,
    canDelete: true,
    showPriority: false
  };

  // Event handler for delete button
  onDeleteClick() {
    console.log("TodoItem: Delete button clicked, calling onDelete callback");

    // Call the parent's function with a payload object
    this.props.onDelete({
      todoId: this.props.todo.id,
      todoText: this.props.todo.text
    });
  }

  // Event handler for toggle completion
  onToggleClick() {
    console.log("TodoItem: Toggle clicked, calling onToggle callback");

    this.props.onToggle({
      todoId: this.props.todo.id,
      currentStatus: this.props.todo.completed
    });
  }

  // Event handler for edit request
  onEditClick() {
    console.log("TodoItem: Edit clicked, calling onEdit callback");

    // Optional callbacks may be undefined - use optional chaining
    this.props.onEdit?.({
      todoId: this.props.todo.id,
      currentText: this.props.todo.text
    });
  }

  // Computed properties for styling
  get todoClasses() {
    const classes = ['todo-item'];
    if (this.props.todo.completed) {
      classes.push('todo-completed');
    }
    if (this.props.todo.priority === 'high') {
      classes.push('todo-high-priority');
    }
    return classes.join(' ');
  }

  get priorityBadgeClass() {
    const priorityClasses = {
      low: 'badge-secondary',
      medium: 'badge-warning',
      high: 'badge-danger'
    };
```

```
    return priorityClasses[this.props.todo.priority] || 'badge-secondary';
  }
}
```

TodoItem Template (`todo_item.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">

  <t t-name="my_module.TodoItem" owl="1">
    <div t-att-class="todoClasses" class="d-flex align-items-center p-3 border rounded
    ↪ mb-2">

      <!-- Toggle Completion Checkbox -->
      <div class="todo-toggle me-3">
        <input type="checkbox"
               class="form-check-input"
               t-att-checked="props.todo.completed"
               t-on-click="onToggleClick"/>
      </div>

      <!-- Todo Content -->
      <div class="todo-content flex-grow-1">
        <div class="d-flex justify-content-between align-items-start">
          <div class="todo-text">
            <span t-att-class="props.todo.completed ? 'text-decoration-line-through
    ↪ text-muted' : ''">
              <t t-esc="props.todo.text"/>
            </span>

            <!-- Priority Badge -->
            <t t-if="props.showPriority and props.todo.priority">
              <span t-att-class="'badge ms-2 ' + priorityBadgeClass">
                <t t-esc="props.todo.priority"/>
              </span>
            </t>
          </div>

          <!-- Due Date -->
          <t t-if="props.todo.dueDate">
            <small class="text-muted">
              Due: <t t-esc="props.todo.dueDate"/>
            </small>
          </t>
        </div>
      </div>

      <!-- Action Buttons -->
      <div class="todo-actions ms-3">
        <t t-if="props.canEdit">
          <button class="btn btn-sm btn-outline-primary me-1"
                  t-on-click="onEditClick">
            <i class="fa fa-edit"></i>
          </button>
        </t>
```

```
        <t t-if="props.canDelete">
          <button class="btn btn-sm btn-outline-danger"
                  t-on-click="onDeleteClick">
            <i class="fa fa-trash"></i>
          </button>
        </t>
      </div>
    </div>
  </t>

</templates>
```

Notice that `t-on-click` is used **on real DOM elements** (the checkbox and buttons)—that's exactly what `t-on-` is for in OWL 2.0: listening to native browser events. The bridge to the parent is the callback prop.

The Parent Component: TodoList

Now let's create the parent component that provides those callbacks:

TodoList JavaScript (`todo_list.js`):

```
import { Component, useState } from "@odoo/owl";
import { TodoItem } from "../todo_item/todo_item";

export class TodoList extends Component {
  static template = "my_module.TodoList";
  static components = { TodoItem };

  setup() {
    this.state = useState({
      todos: [
        {
          id: 1,
          text: "Learn OWL basics",
          completed: true,
          priority: "medium",
          dueDate: "2024-03-20"
        },
        {
          id: 2,
          text: "Build a todo app",
          completed: false,
          priority: "high",
          dueDate: "2024-03-25"
        },
        {
          id: 3,
          text: "Master component communication",
          completed: false,
          priority: "low",
          dueDate: "2024-03-30"
        }
      ],

      editingTodoId: null,
```

```
      newTodoText: "",
      showCompleted: true,
      showPriority: true
    });
  }

  // Callback: child asked us to delete a todo
  onTodoDelete({ todoId, todoText }) {
    console.log(`TodoList: onTodoDelete called for ID ${todoId}`);

    // Show confirmation (optional)
    if (confirm(`Are you sure you want to delete "${todoText}"?`)) {
      // Update parent state - this will cause re-render
      this.state.todos = this.state.todos.filter(todo => todo.id !== todoId);

      console.log(`TodoList: Todo ${todoId} deleted successfully`);
    }
  }

  // Callback: child asked us to toggle completion
  onTodoToggle({ todoId }) {
    console.log(`TodoList: onTodoToggle called for ID ${todoId}`);

    // Find and update the todo
    const todo = this.state.todos.find(t => t.id === todoId);
    if (todo) {
      todo.completed = !todo.completed;
      console.log(`TodoList: Todo ${todoId} completion status changed to
          ↪  ${todo.completed}`);
    }
  }

  // Callback: child asked us to start editing a todo
  onTodoEdit({ todoId, currentText }) {
    console.log(`TodoList: onTodoEdit called for ID ${todoId}`);

    // Enter edit mode
    this.state.editingTodoId = todoId;
    this.state.newTodoText = currentText;
  }

  // Local methods for managing todos
  addNewTodo() {
    if (!this.state.newTodoText.trim()) return;

    const newTodo = {
      id: Math.max(...this.state.todos.map(t => t.id), 0) + 1,
      text: this.state.newTodoText.trim(),
      completed: false,
      priority: "medium",
      dueDate: null
    };

    this.state.todos.push(newTodo);
    this.state.newTodoText = "";
```

```
  }

  saveEdit() {
    if (!this.state.newTodoText.trim()) return;

    const todo = this.state.todos.find(t => t.id === this.state.editingTodoId);
    if (todo) {
      todo.text = this.state.newTodoText.trim();
    }

    this.cancelEdit();
  }

  cancelEdit() {
    this.state.editingTodoId = null;
    this.state.newTodoText = "";
  }

  toggleShowCompleted() {
    this.state.showCompleted = !this.state.showCompleted;
  }

  toggleShowPriority() {
    this.state.showPriority = !this.state.showPriority;
  }

  // Computed properties
  get visibleTodos() {
    if (this.state.showCompleted) {
      return this.state.todos;
    }
    return this.state.todos.filter(todo => !todo.completed);
  }

  get completedCount() {
    return this.state.todos.filter(todo => todo.completed).length;
  }

  get totalCount() {
    return this.state.todos.length;
  }
}
```

TodoList Template (`todo_list.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">

  <t t-name="my_module.TodoList" owl="1">
    <div class="todo-list-container p-4">

      <!-- Header -->
      <div class="todo-header mb-4">
        <div class="d-flex justify-content-between align-items-center">
          <div>
            <h2>My Todo List</h2>
```

```
        <p class="text-muted mb-0">
          <t t-esc="completedCount"/> completed of <t t-esc="totalCount"/> todos
        </p>
      </div>

      <!-- View Controls -->
      <div class="view-controls">
        <button class="btn btn-sm me-2"
                t-att-class="state.showCompleted ? 'btn-primary' :
↪  'btn-outline-primary'"
                t-on-click="toggleShowCompleted">
          <i class="fa fa-eye"></i>
          <t t-esc="state.showCompleted ? 'Hide Completed' : 'Show All'"/>
        </button>

        <button class="btn btn-sm btn-outline-secondary"
                t-att-class="state.showPriority ? 'active' : ''"
                t-on-click="toggleShowPriority">
          <i class="fa fa-flag"></i>
          Priority
        </button>
      </div>
    </div>
  </div>

  <!-- Add New Todo Form -->
  <div class="add-todo-form mb-4 p-3 bg-light rounded">
    <div class="row">
      <div class="col">
        <input type="text"
               class="form-control"
               placeholder="Add a new todo..."
               t-model="state.newTodoText"
               t-on-keyup.enter="addNewTodo"/>
      </div>
      <div class="col-auto">
        <button class="btn btn-success" t-on-click="addNewTodo">
          <i class="fa fa-plus"></i>
          Add Todo
        </button>
      </div>
    </div>
  </div>

  <!-- Edit Form -->
  <t t-if="state.editingTodoId">
    <div class="edit-todo-form mb-4 p-3 bg-warning bg-opacity-10 rounded">
      <h5>Edit Todo</h5>
      <div class="row">
        <div class="col">
          <input type="text"
                 class="form-control"
                 t-model="state.newTodoText"
                 t-on-keyup.enter="saveEdit"/>
        </div>
```

```
            <div class="col-auto">
              <button class="btn btn-success me-2" t-on-click="saveEdit">
                <i class="fa fa-check"></i>
                Save
              </button>
              <button class="btn btn-secondary" t-on-click="cancelEdit">
                <i class="fa fa-times"></i>
                Cancel
              </button>
            </div>
          </div>
        </div>
      </t>

      <!-- Todo Items -->
      <div class="todo-items">
        <t t-if="visibleTodos.length > 0">
          <t t-foreach="visibleTodos" t-as="todo" t-key="todo.id">
            <!-- This is where we hand our callbacks to the child -->
            <TodoItem
                todo="todo"
                canEdit="true"
                canDelete="true"
                showPriority="state.showPriority"
                onDelete.bind="onTodoDelete"
                onToggle.bind="onTodoToggle"
                onEdit.bind="onTodoEdit"
            />
          </t>
        </t>

        <!-- Empty State -->
        <t t-else="">
          <div class="empty-state text-center py-5">
            <i class="fa fa-tasks fa-3x text-muted mb-3"></i>
            <h4 class="text-muted">No todos to show</h4>
            <p class="text-muted">
              <t t-if="!state.showCompleted">
                All todos are completed! Toggle "Show All" to see them.
              </t>
              <t t-else="">
                Add your first todo above to get started.
              </t>
            </p>
          </div>
        </t>
      </div>
    </div>
  </t>

</templates>
```

The `.bind` Suffix: Small Detail, Big Deal

Look closely at how the callbacks are passed:

```
<TodoItem onDelete.bind="onTodoDelete"/>
```

The `.bind` suffix tells OWL to bind the function to the parent component before passing it down. Without it, when the child calls `this.props.onDelete(...)`, the `this` inside `onTodoDelete` would be `undefined` and `this.state` would crash.

You have three equivalent options—pick one and be consistent:

```
<!-- Option 1 (recommended): the .bind suffix -->
<TodoItem onDelete.bind="onTodoDelete"/>

<!-- Option 2: an inline arrow function (binds `this` automatically) -->
<TodoItem onDelete="(payload) => this.onTodoDelete(payload)"/>
```

```
// Option 3: bind manually in setup()
setup() {
  this.onTodoDelete = this.onTodoDelete.bind(this);
}
```

The `.bind` suffix is the idiomatic OWL 2.0 way and what you'll see throughout the Odoo codebase.

Advanced Callback Patterns

1. Native Events and Stopping Propagation

`t-on-` on DOM elements supports useful modifiers:

```
<!-- .stop calls event.stopPropagation() for you -->
<button t-on-click.stop="onDeleteClick">Delete</button>

<!-- .prevent calls event.preventDefault() -->
<form t-on-submit.prevent="onSubmit">...</form>

<!-- .enter fires only for the Enter key -->
<input t-on-keyup.enter="addNewTodo"/>
```

Use these to control the native DOM event; then call your callback prop as usual.

2. Conditional Callback Invocation

```
// Only call the callback if certain conditions are met
onDeleteClick() {
  if (this.props.todo.completed) {
    this.props.onDelete({ todoId: this.props.todo.id });
  } else {
    this.props.onConfirmDelete?.({
      todoId: this.props.todo.id,
      message: "This todo is not completed. Are you sure?"
    });
  }
}
```

3. Async Callbacks

Callbacks can be `async`, which lets the child await the parent's work (for example, to show a spinner while the parent saves):

```
// Child
async onSaveClick() {
  this.state.saving = true;
  try {
    await this.props.onSave({ data: this.state.formData });
  } finally {
    this.state.saving = false;
  }
}
```

4. Validation in the Parent

```
// Parent validates the payload before processing
onTodoUpdate({ todoId, newText }) {
  // Validate the data
  if (!todoId || typeof newText !== 'string' || newText.trim().length === 0) {
    console.error("Invalid todo update:", { todoId, newText });
    return;
  }

  // Proceed with update
  const todo = this.state.todos.find(t => t.id === todoId);
  if (todo) {
    todo.text = newText.trim();
  }
}
```

Callback Naming Conventions

Following consistent naming conventions makes your code more maintainable:

```
// In the child's props definition: "on" + what happened, camelCase
static props = {
  onTodoCreated: { type: Function },
  onUserSelected: { type: Function },
  onFormSubmitted: { type: Function },
  onError: { type: Function, optional: true },
};
```

Best Practices:

1. **Prefix with `on`**: `onDelete`, `onSave`, `onUserSelected`—it instantly reads as "callback"
2. **Use camelCase**: callback props are regular props, so they follow prop naming (`onTodoDeleted`, not `on-todo-deleted`)
3. **Be descriptive**: `onUserProfileUpdated` not `onUpdate` when the context isn't obvious
4. **Include context in payloads**: pass an object (`{ todoId, todoText }`) rather than loose arguments—it's self-documenting and extensible
5. **Mark truly optional callbacks as `optional: true`** and call them with `?.()`

Complex Communication Pattern

Let's create a more sophisticated example with a `UserManager` parent and multiple child components:

UserManager Parent (`user_manager.js`):

```
import { Component, useState } from "@odoo/owl";
import { UserCard } from "../user_card/user_card";
import { UserForm } from "../user_form/user_form";
import { UserFilters } from "../user_filters/user_filters";

export class UserManager extends Component {
  static template = "my_module.UserManager";
  static components = { UserCard, UserForm, UserFilters };

  setup() {
    this.state = useState({
      users: [
        { id: 1, name: "Alice Johnson", role: "admin", active: true, department: "IT" },
        { id: 2, name: "Bob Smith", role: "user", active: true, department: "Sales" },
        { id: 3, name: "Carol Brown", role: "user", active: false, department: "HR" }
      ],
      filters: {
        role: "all",
        department: "all",
        active: true
      },
      editingUser: null,
      showForm: false,
      searchText: ""
    });
  }

  // Callbacks for UserCard
  onUserEdit({ user }) {
    console.log("UserManager: Edit user requested", user);
    this.state.editingUser = { ...user }; // Clone for editing
    this.state.showForm = true;
  }

  onUserDelete({ userId, userName }) {
    console.log("UserManager: Delete user requested", userId);

    if (confirm(`Delete user ${userName}?`)) {
      this.state.users = this.state.users.filter(u => u.id !== userId);
    }
  }

  onUserToggleStatus({ userId }) {
    console.log("UserManager: Toggle user status", userId);

    const user = this.state.users.find(u => u.id === userId);
    if (user) {
      user.active = !user.active;
    }
  }
```

```
  // Callbacks for UserForm
  onUserSave({ user }) {
    console.log("UserManager: Save user", user);

    if (user.id) {
      // Update existing user
      const index = this.state.users.findIndex(u => u.id === user.id);
      if (index !== -1) {
        this.state.users[index] = user;
      }
    } else {
      // Create new user
      user.id = Math.max(...this.state.users.map(u => u.id), 0) + 1;
      this.state.users.push(user);
    }

    this.onFormCancel();
  }

  onFormCancel() {
    console.log("UserManager: Form cancelled");
    this.state.showForm = false;
    this.state.editingUser = null;
  }
```

Three different children (`UserCard`, `UserForm`, `UserFilters`) each get their own dedicated callback methods above—`UserManager` never needs to guess which child called it, since every callback is passed explicitly to a single component in the template.

```
  // Callbacks for UserFilters
  onFiltersChanged({ filters }) {
    console.log("UserManager: Filters changed", filters);
    this.state.filters = { ...this.state.filters, ...filters };
  }

  onSearchChanged({ searchText }) {
    console.log("UserManager: Search changed", searchText);
    this.state.searchText = searchText;
  }

  // Local actions
  openNewUserForm() {
    this.state.editingUser = {
      id: null,
      name: "",
      role: "user",
      active: true,
      department: ""
    };
    this.state.showForm = true;
  }
```

Finally, `filteredUsers` is a plain computed getter—no props or callbacks involved—that combines the active filters and search text into the list actually shown in the template:

```
  // Computed properties
```

```
  get filteredUsers() {
    let filtered = this.state.users;

    // Apply role filter
    if (this.state.filters.role !== "all") {
      filtered = filtered.filter(u => u.role === this.state.filters.role);
    }

    // Apply department filter
    if (this.state.filters.department !== "all") {
      filtered = filtered.filter(u => u.department === this.state.filters.department);
    }

    // Apply active filter
    if (this.state.filters.active !== null) {
      filtered = filtered.filter(u => u.active === this.state.filters.active);
    }

    // Apply search
    if (this.state.searchText) {
      const search = this.state.searchText.toLowerCase();
      filtered = filtered.filter(u =>
        u.name.toLowerCase().includes(search) ||
        u.department.toLowerCase().includes(search)
      );
    }

    return filtered;
  }
}
```

And in the template, each child gets exactly the callbacks it needs:

```
<UserFilters
    filters="state.filters"
    onFiltersChanged.bind="onFiltersChanged"
    onSearchChanged.bind="onSearchChanged"
/>

<t t-foreach="filteredUsers" t-as="user" t-key="user.id">
  <UserCard
      user="user"
      onEdit.bind="onUserEdit"
      onDelete.bind="onUserDelete"
      onToggleStatus.bind="onUserToggleStatus"
  />
</t>

<t t-if="state.showForm">
  <UserForm
      user="state.editingUser"
      onSave.bind="onUserSave"
      onCancel.bind="onFormCancel"
  />
</t>
```

This example demonstrates how a single parent can coordinate multiple child components through

callbacks, maintaining centralized state management while allowing children to remain focused and reusable.

Data Props vs Callback Props: Decision Framework

Both travel down as props, but they serve opposite directions of communication:

Use Callback Props When:

- **Child needs to notify parent** of user actions
- **Child needs to request** parent to change data
- **Child encounters an error** that parent should handle
- **Child completes an async operation**

```
static props = {
  onDeleted: { type: Function },       // "Something happened"
  onSaveRequested: { type: Function }, // "Please do something"
  onError: { type: Function },         // "Something went wrong"
};
```

Use Data Props When:

- **Parent needs to configure** child behavior
- **Parent needs to pass data** to child
- **Parent controls** child's display state

```
static props = {
  user: { type: Object },          // Data prop
  readonly: { type: Boolean },     // Configuration prop
  showDetails: { type: Boolean }   // State prop
};
```

> **What about components that aren't parent and child?** Callbacks work through the component tree. For communication between distant components (siblings, or across the whole app), you'll use shared state and the environment bus—both covered in Chapter 15.

Debugging Callbacks

Console Logging Pattern

```
// In child component
onAction() {
  console.log("Child: Calling onAction callback", { payload: "data" });
  this.props.onAction({ payload: "data" });
}

// In parent component
onChildAction(payload) {
  console.log("Parent: onChildAction received", payload);
```

```
  // Handle it...
}
```

Common Issues and Solutions

Issue 1: "Cannot read properties of undefined (reading 'state')"

```
<!-- Problem: unbound function loses `this` -->
<TodoItem onDelete="onTodoDelete"/>

<!-- Solution: use the .bind suffix -->
<TodoItem onDelete.bind="onTodoDelete"/>
```

Issue 2: "props.onEdit is not a function"

```
// The parent didn't pass an optional callback.
// Declare it optional and call it safely:
static props = {
  onEdit: { type: Function, optional: true },
};

onEditClick() {
  this.props.onEdit?.({ todoId: this.props.todo.id });
}
```

Issue 3: Callback runs immediately on render

```
<!-- Problem: this CALLS the function during rendering -->
<TodoItem onDelete="onTodoDelete(todo.id)"/>

<!-- Solution: pass a function, don't call one -->
<TodoItem onDelete="() => this.onTodoDelete(todo.id)"/>
```

Performance Considerations

1. Debounce High-Frequency Callbacks

```
// Bad: calls the parent on every keystroke
onInputChange(event) {
  this.props.onTextChanged({ value: event.target.value });
}

// Good: debounced callback. Debouncing means delaying a function call until
// a burst of triggering events (e.g. keystrokes) stops for a given period,
// so you don't fire one call per keystroke. `debounce` is available in
// @web/core/utils/timing.
setup() {
  this.debouncedNotify = debounce((value) => {
    this.props.onTextChanged({ value });
  }, 300);
}

onInputChange(event) {
  this.debouncedNotify(event.target.value);
}
```

2. Minimize Payload Size

```
// Bad: large payload with unnecessary data
this.props.onUserSelected({
  fullUser: this.props.user,   // Entire user object
  allUsers: this.props.users   // Unnecessary data
});

// Good: minimal payload
this.props.onUserSelected({
  userId: this.props.user.id   // Just the ID
});
```

3. Use Stable Function References

The `.bind` suffix creates the bound function once, so children don't see a "new" prop on every render. Prefer it over defining fresh arrow functions in templates when the callback doesn't need loop variables.

Common Pitfalls

Pitfall 1: Forgetting `.bind`

```
<!-- `this` inside onTodoDelete will be undefined -->
<TodoItem onDelete="onTodoDelete"/>
```

Always use `.bind`, an inline arrow function, or a manual `.bind(this)` in `setup()`—see "The `.bind` Suffix" above.

Pitfall 2: Calling the Callback Instead of Passing It

```
<!-- Problem: this CALLS onTodoDelete during rendering, not on click -->
<TodoItem onDelete="onTodoDelete(todo.id)"/>

<!-- Solution -->
<TodoItem onDelete="() => this.onTodoDelete(todo.id)"/>
```

Pitfall 3: Treating Optional Callbacks as Always Present

If a callback prop is `optional: true`, calling it directly (`this.props.onEdit(...)`) throws when the parent didn't pass it. Use optional chaining: `this.props.onEdit?.(...)`.

Pitfall 4: Sending Bulky Payloads

Passing the entire component's state, or unrelated data, in a callback payload couples the child to internals it shouldn't know about. Send the minimum the parent needs (an id, a small object) instead of whole objects or arrays.

Exercises

Exercise 1: Add a Callback

Extend the `TodoItem`/`TodoList` example with a new callback prop `onPriorityChange` that the child calls when the user clicks a "bump priority" button. The parent should update the todo's priority in its state.

Exercise 2: Fix the Bug

Given `<TodoItem onDelete="onTodoDelete"/>` (no `.bind`, no arrow function), predict what happens when the delete button is clicked, then verify it in the browser console. Fix it using the `.bind` suffix.

Exercise 3: Debounced Search

Build a `SearchBox` component with a text input and an `onSearch` callback prop. Debounce the callback so it only fires 300ms after the user stops typing, using `debounce` from `@web/core/utils/timing`.

Callback props are the key to building maintainable OWL applications where components can communicate effectively while remaining loosely coupled. By mastering this pattern—data down, callbacks up—you create applications that are both powerful and predictable.

TL;DR: Children talk to parents by calling a function the parent passed down as a prop (always with `.bind`), never by mutating parent state or reaching for a legacy event bus.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch9_ex1`. Installation instructions are in the repository README.

What's Next?

With data down and callbacks up, the parent-child communication loop is complete. In the next chapter, we'll explore component lifecycle hooks that let you tap into key moments in a component's life to perform initialization, cleanup, and optimization.

Chapter 10: The Component Lifecycle Hooks

Why this chapter? More production incidents I've dealt with trace back to lifecycle mistakes than to almost anything else — a fetch that fires before the DOM exists, a listener that never gets cleaned up and slowly leaks memory across a long-running session.

What is Odoo trying to solve with this? A backend view can mount and unmount dozens of widgets as a user navigates — lifecycle hooks are how Odoo guarantees each one initializes and tears down cleanly, without leftover timers or listeners piling up.

Real-world application: Loading a partner's related records before first render (`onWillStart`), focusing an input when a form opens (`onMounted`), and clearing an interval when a kanban card is removed (`onWillUnmount`) are all lifecycle hooks doing exactly the job this chapter describes.

A component, much like a living organism, has a **lifecycle**. It is born (created and mounted to the DOM), it lives (updates in response to state and prop changes), and eventually, it dies (is unmounted and removed from the DOM).

OWL provides a powerful way to tap into these key moments in a component's life using **lifecycle hooks**. These hooks are special functions that you can call within your `setup` method to register callbacks that will execute at specific points in the lifecycle.

Understanding these hooks is essential for managing side effects, fetching data correctly, and cleaning up resources to prevent memory leaks. Let's explore the most important hooks you'll use in your day-to-day development.

The Lifecycle at a Glance

Before diving into individual hooks, let's visualize the complete journey of a component:

1. **Creation Phase:**
   - `setup()`: The component's main setup logic runs and registers the hooks.
   - `onWillStart`: Runs before the first render, perfect for initial data fetching.
   - Component instance created and prepared.
2. **Mounting Phase (Added to the page):**
   - The component renders its HTML template.
   - The HTML is inserted into the DOM.
   - `onMounted`: The component is now live on the page.
3. **Update Phase (During its life - repeats as needed):**
   - Parent passes new props → `onWillUpdateProps`.
   - State changes trigger re-render.
   - The component re-renders its HTML.

- DOM is updated with changes.

4. **Unmounting Phase (Removal from the page):**
   - `onWillUnmount`: The component is about to be destroyed.
   - The component is removed from the DOM.
   - Memory and resources are cleaned up.

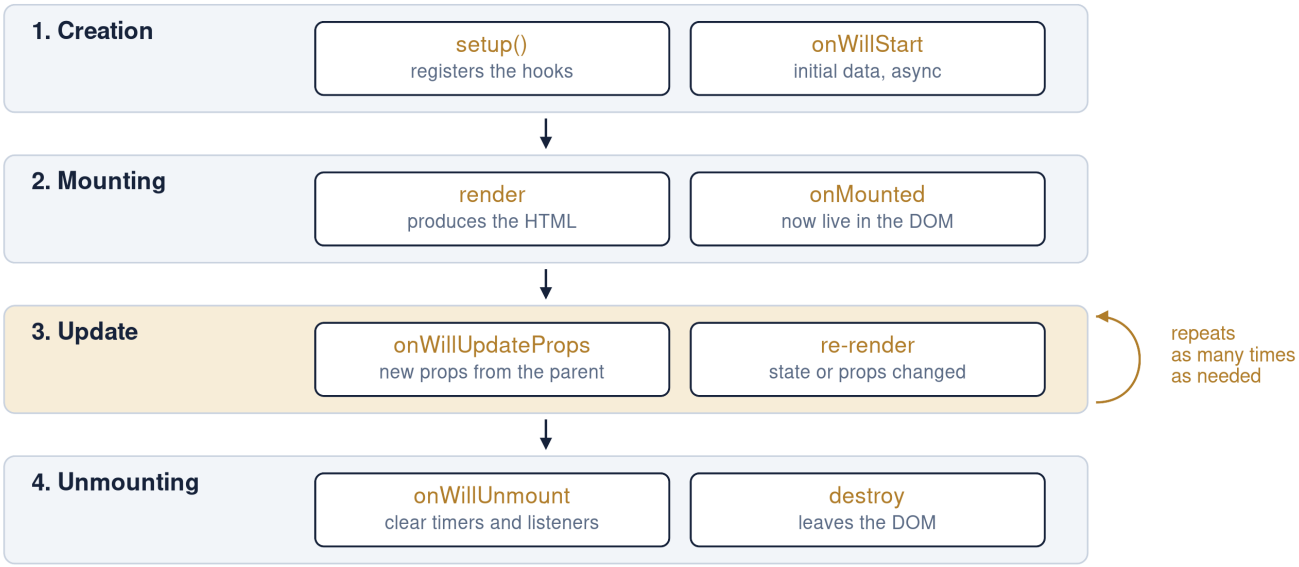


Figure 4: The four phases of a component's life, and which hook fires in each.

`onWillStart`: Fetching Initial Data

The `onWillStart` hook is unique because it's the **only hook that can be asynchronous**. It runs before the component's initial render, making it the perfect place to fetch data that the component needs for its very first appearance.

- **When it runs:** After `setup()` finishes, but before the initial render.
- **Key feature:** Can be async - the component waits for the promise to resolve.
- **Use case:** Fetching essential data that must be ready before first render.

Basic Example: Product List

Let's create a `ProductList` component that fetches products before rendering:

JavaScript (`product_list.js`):

```
import { Component, onWillStart, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ProductList extends Component {
  static template = "my_module.ProductList";

  setup() {
    this.state = useState({
      products: [],
      loading: true,
```

```
      error: null
    });

    this.orm = useService("orm");

    onWillStart(async () => {
      try {
        console.log("ProductList: Fetching products before first render...");
        const data = await this.orm.searchRead(
          "product.product",
          [],
          ["name", "list_price", "categ_id"]
        );
        this.state.products = data;
        console.log(`ProductList: Loaded ${data.length} products`);
      } catch (error) {
        console.error("Failed to load products:", error);
        this.state.error = "Failed to load products";
      } finally {
        this.state.loading = false;
      }
    });
  }

  get formattedProducts() {
    return this.state.products.map(product => ({
      ...product,
      formattedPrice: `$${product.list_price.toFixed(2)}`
    }));
  }
}
```

Template (`product_list.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.ProductList" owl="1">
    <div class="product-list">
      <!-- Loading State -->
      <t t-if="state.loading">
        <div class="text-center py-4">
          <div class="spinner-border text-primary" role="status">
            <span class="visually-hidden">Loading...</span>
          </div>
          <p class="mt-2 text-muted">Loading products...</p>
        </div>
      </t>

      <!-- Error State -->
      <t t-elif="state.error">
        <div class="alert alert-danger" t-esc="state.error"/>
      </t>

      <!-- Products Grid -->
      <t t-else="">
        <div class="row">
```

```
          <t t-foreach="formattedProducts" t-as="product" t-key="product.id">
            <div class="col-md-4 mb-3">
              <div class="card">
                <div class="card-body">
                  <h5 class="card-title" t-esc="product.name"/>
                  <p class="card-text">
                    <strong t-esc="product.formattedPrice"/>
                  </p>
                  <small class="text-muted" t-esc="product.categ_id[1]"/>
                </div>
              </div>
            </div>
          </t>
        </div>
      </t>
    </div>
  </t>
</templates>
```

By fetching data in `onWillStart`, you prevent the component from rendering an empty state and then "flickering" when the data arrives. The component will wait for the `onWillStart` promise to resolve before performing its initial render.

Advanced onWillStart Patterns

Multiple Async Operations:

```
onWillStart(async () => {
  // Run multiple operations in parallel
  const [products, categories, suppliers] = await Promise.all([
    this.orm.searchRead("product.product", [], ["name", "list_price"]),
    this.orm.searchRead("product.category", [], ["name"]),
    this.orm.searchRead("res.partner", [["supplier_rank", ">", 0]], ["name"])
  ]);

  this.state.products = products;
  this.state.categories = categories;
  this.state.suppliers = suppliers;
});
```

Error Recovery Pattern:

```
onWillStart(async () => {
  try {
    // Try primary data source
    this.state.data = await this.fetchFromPrimarySource();
  } catch (primaryError) {
    console.warn("Primary source failed, trying fallback:", primaryError);

    try {
      // Try fallback data source
      this.state.data = await this.fetchFromFallbackSource();
      this.state.usingFallback = true;
    } catch (fallbackError) {
      console.error("All data sources failed:", fallbackError);
      this.state.error = "Unable to load data from any source";
    }
```

```
  }
});
```

`onMounted`: Interacting with the DOM

The `onMounted` hook runs **after** the component has been rendered for the first time and its HTML has been added to the DOM. This is your gateway to DOM manipulation and third-party library integration.

- **When it runs:** After the initial render and DOM insertion.
- **Use case:** Any task that requires the component's HTML to exist in the DOM.

Key Use Cases:

- **Focus input elements**
- **Initialize third-party libraries** (charts, maps, calendars)
- **Measure element dimensions**
- **Set up DOM event listeners**

To get a reference to a specific element in your template, you use the `t-ref` directive and the `useRef` hook.

Example: Auto-Focus Search Component

JavaScript (`search_component.js`):

```
import { Component, onMounted, useRef, useState } from "@odoo/owl";

export class SearchComponent extends Component {
  static template = "my_module.SearchComponent";

  setup() {
    this.state = useState({
      searchTerm: "",
      results: []
    });

    // Create a reference to the search input
    this.searchInputRef = useRef("search-input");

    onMounted(() => {
      console.log("SearchComponent: Component mounted, focusing input");

      // The input element is now in the DOM and can be focused
      if (this.searchInputRef.el) {
        this.searchInputRef.el.focus();
        console.log("SearchComponent: Input focused successfully");
      }
    });
  }

  onSearchInput(event) {
    this.state.searchTerm = event.target.value;
    console.log("Search term changed:", this.state.searchTerm);
```

```
    // Here you could trigger a search API call
  }
}
```

Template (`search_component.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.SearchComponent" owl="1">
    <div class="search-component">
      <div class="search-container mb-3">
        <input
            type="text"
            class="form-control"
            t-ref="search-input"
            placeholder="Search products..."
            t-model="state.searchTerm"
            t-on-input="onSearchInput"
        />
      </div>

      <div class="search-results">
        <t t-if="state.searchTerm">
          <p class="text-muted">
            Searching for: "<t t-esc="state.searchTerm"/>"
          </p>
        </t>

        <!-- Results would go here -->
        <div class="results-list">
          <!-- Results implementation -->
        </div>
      </div>
    </div>
  </t>
</templates>
```

Advanced onMounted Example: Chart Integration

Here's a more complex example that integrates with Chart.js:

JavaScript (`sales_chart.js`):

```
import { Component, onMounted, onWillUnmount, useRef, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class SalesChart extends Component {
  static template = "my_module.SalesChart";

  setup() {
    this.state = useState({
      chartData: [],
      loading: true
    });

    this.chartCanvasRef = useRef("sales-chart");
    this.orm = useService("orm");
```

```
    this.chartInstance = null;

    onMounted(async () => {
      console.log("SalesChart: Component mounted, initializing chart");
      await this.loadSalesData();
      this.initializeChart();
    });

    onWillUnmount(() => {
      // Clean up the chart when component is destroyed
      if (this.chartInstance) {
        console.log("SalesChart: Destroying chart instance");
        this.chartInstance.destroy();
      }
    });
  }

  async loadSalesData() {
    try {
      const salesData = await this.orm.searchRead(
        "sale.order",
        [["state", "=", "sale"]],
        ["date_order", "amount_total"]
      );

      this.state.chartData = salesData;
      this.state.loading = false;
    } catch (error) {
      console.error("Failed to load sales data:", error);
      this.state.loading = false;
    }
  }

  initializeChart() {
    if (!this.chartCanvasRef.el || this.state.loading) {
      return;
    }

    // Assuming Chart.js is loaded globally
    const ctx = this.chartCanvasRef.el.getContext('2d');

    // Process data for chart
    const processedData = this.processChartData();

    this.chartInstance = new Chart(ctx, {
      type: 'line',
      data: {
        labels: processedData.labels,
        datasets: [{
          label: 'Sales Revenue',
          data: processedData.data,
          borderColor: 'rgb(75, 192, 192)',
          backgroundColor: 'rgba(75, 192, 192, 0.1)',
          tension: 0.1
        }]
```

```
      },
      options: {
        responsive: true,
        plugins: {
          title: {
            display: true,
            text: 'Sales Performance'
          }
        }
      }
    });

    console.log("SalesChart: Chart initialized successfully");
  }

  processChartData() {
    // Group sales by month and sum amounts
    const monthlyData = {};

    this.state.chartData.forEach(order => {
      const month = order.date_order.substring(0, 7); // YYYY-MM format
      if (!monthlyData[month]) {
        monthlyData[month] = 0;
      }
      monthlyData[month] += order.amount_total;
    });

    const sortedMonths = Object.keys(monthlyData).sort();

    return {
      labels: sortedMonths,
      data: sortedMonths.map(month => monthlyData[month])
    };
  }
}
```

Template (`sales_chart.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.SalesChart" owl="1">
    <div class="sales-chart">
      <t t-if="state.loading">
        <div class="text-center py-4">
          <div class="spinner-border text-primary" role="status">
            <span class="visually-hidden">Loading chart data...</span>
          </div>
        </div>
      </t>

      <t t-else="">
        <div class="chart-container">
          <canvas t-ref="sales-chart" width="400" height="200"></canvas>
        </div>
      </t>
    </div>
```

```
  </t>
</templates>
```

`onWillUpdateProps`: Reacting to Prop Changes

This hook is called when a parent component is about to pass new props to the child. It allows the child to react to the incoming changes *before* it re-renders itself.

- **When it runs:** When new props are received, before the component updates.
- **Use case:** When a child component needs to fetch new data based on changed props.

Example: User Profile Component

Imagine a `UserProfile` component that displays user details. When the parent changes the `userId` prop, the component needs to fetch the new user's data:

JavaScript (`user_profile.js`):

```
import { Component, onWillStart, onWillUpdateProps, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class UserProfile extends Component {
  static template = "my_module.UserProfile";
  static props = {
    userId: { type: Number }
  };

  setup() {
    this.state = useState({
      user: null,
      loading: true,
      error: null
    });

    this.orm = useService("orm");

    // Load initial user data
    onWillStart(async () => {
      await this.loadUser(this.props.userId);
    });

    // React to prop changes
    onWillUpdateProps(async (nextProps) => {
      console.log("UserProfile: Props will update", {
        current: this.props.userId,
        next: nextProps.userId
      });

      // Only reload if the user ID actually changed
      if (this.props.userId !== nextProps.userId) {
        console.log(`UserProfile: User ID changed from ${this.props.userId} to
            ↪ ${nextProps.userId}`);
        this.state.loading = true;
        await this.loadUser(nextProps.userId);
```

```
      }
    });
  }

  async loadUser(userId) {
    try {
      console.log(`UserProfile: Loading user data for ID ${userId}`);

      const users = await this.orm.read("res.users", [userId], [
        "name",
        "email",
        "phone",
        "partner_id"
      ]);

      if (users.length > 0) {
        this.state.user = users[0];
        this.state.error = null;
        console.log(`UserProfile: Successfully loaded user ${users[0].name}`);
      } else {
        throw new Error(`User with ID ${userId} not found`);
      }

    } catch (error) {
      console.error(`Failed to load user ${userId}:`, error);
      this.state.error = `Failed to load user: ${error.message}`;
      this.state.user = null;
    } finally {
      this.state.loading = false;
    }
  }
}
```

Template (`user_profile.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.UserProfile" owl="1">
    <div class="user-profile card">
      <div class="card-header">
        <h5>User Profile</h5>
      </div>
      <div class="card-body">
        <!-- Loading State -->
        <t t-if="state.loading">
          <div class="text-center py-3">
            <div class="spinner-border text-primary" role="status">
              <span class="visually-hidden">Loading user...</span>
            </div>
            <p class="mt-2 text-muted">Loading user profile...</p>
          </div>
        </t>

        <!-- Error State -->
        <t t-elif="state.error">
          <div class="alert alert-danger" t-esc="state.error"/>
```

```
        </t>

        <!-- User Data -->
        <t t-elif="state.user">
          <div class="user-info">
            <h6 class="mb-3" t-esc="state.user.name"/>
            <div class="user-details">
              <div class="mb-2">
                <strong>Email:</strong>
                <span t-esc="state.user.email"/>
              </div>
              <t t-if="state.user.phone">
                <div class="mb-2">
                  <strong>Phone:</strong>
                  <span t-esc="state.user.phone"/>
                </div>
              </t>
              <div class="mb-2">
                <strong>Partner ID:</strong>
                <span t-esc="state.user.partner_id[1]"/>
              </div>
            </div>
          </div>
        </t>
      </div>
    </div>
  </t>
</templates>
```

Parent Component Example

Here's how a parent component might use the `UserProfile`:

JavaScript (`user_manager.js`):

```
import { Component, useState } from "@odoo/owl";
import { UserProfile } from "../user_profile/user_profile";

export class UserManager extends Component {
  static template = "my_module.UserManager";
  static components = { UserProfile };

  setup() {
    this.state = useState({
      selectedUserId: 1,
      availableUsers: [
        { id: 1, name: "Alice Johnson" },
        { id: 2, name: "Bob Smith" },
        { id: 3, name: "Carol Brown" }
      ]
    });
  }

  selectUser(userId) {
    console.log("UserManager: Selecting user", userId);
    this.state.selectedUserId = userId;
```

```
  }
}
```

Template (`user_manager.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.UserManager" owl="1">
    <div class="user-manager">
      <div class="row">
        <!-- User Selection -->
        <div class="col-md-4">
          <div class="card">
            <div class="card-header">
              <h6>Select User</h6>
            </div>
            <div class="card-body">
              <t t-foreach="state.availableUsers" t-as="user" t-key="user.id">
                <button
                    class="btn btn-outline-primary btn-sm me-2 mb-2"
                    t-att-class="state.selectedUserId === user.id ? 'btn btn-primary
    ↪  btn-sm me-2 mb-2' : 'btn btn-outline-primary btn-sm me-2 mb-2'"
                    t-on-click="() => this.selectUser(user.id)">
                  <t t-esc="user.name"/>
                </button>
              </t>
            </div>
          </div>
        </div>

        <!-- User Profile Display -->
        <div class="col-md-8">
          <UserProfile userId="state.selectedUserId"/>
        </div>
      </div>
    </div>
  </t>
</templates>
```

This pattern ensures that the `UserProfile` component always shows data consistent with the `userId` prop it receives from its parent. When the user clicks a different user button in the parent, the `UserProfile` will automatically detect the prop change and fetch the new user's data.

`onWillUnmount`: Cleaning Up

The `onWillUnmount` hook is the component's last breath. It runs just before the component is removed from the DOM. This is **crucial** for cleanup to prevent memory leaks.

- **When it runs:** Right before the component is destroyed.
- **Critical use case:** Clean up resources that would otherwise persist after component destruction.

What to Clean Up:

- **Timers and intervals** (`setInterval`, `setTimeout`)

- **Global event listeners** (on `window`, `document`)
- **Third-party library instances** (charts, maps, etc.)
- **WebSocket connections**
- **Ongoing API requests** (if cancellable)

Example: Timer-Based Component

JavaScript (`live_clock.js`):

```
import { Component, onMounted, onWillUnmount, useState } from "@odoo/owl";

export class LiveClock extends Component {
  static template = "my_module.LiveClock";

  setup() {
    this.state = useState({
      currentTime: new Date().toLocaleTimeString(),
      isRunning: true
    });

    // Store timer reference for cleanup
    this.timer = null;

    onMounted(() => {
      console.log("LiveClock: Starting clock timer");
      this.startClock();
    });

    onWillUnmount(() => {
      console.log("LiveClock: Cleaning up clock timer");
      this.stopClock();
    });
  }

  startClock() {
    // Update time every second
    this.timer = setInterval(() => {
      if (this.state.isRunning) {
        this.state.currentTime = new Date().toLocaleTimeString();
      }
    }, 1000);
  }

  stopClock() {
    if (this.timer) {
      clearInterval(this.timer);
      this.timer = null;
      console.log("LiveClock: Timer cleared successfully");
    }
  }

  toggleClock() {
    this.state.isRunning = !this.state.isRunning;
    console.log("LiveClock: Clock", this.state.isRunning ? "resumed" : "paused");
  }
}
```

Template (`live_clock.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.LiveClock" owl="1">
    <div class="live-clock card">
      <div class="card-body text-center">
        <h3 class="display-4 mb-3" t-esc="state.currentTime"/>
        <button
            class="btn btn-primary"
            t-on-click="toggleClock">
          <t t-esc="state.isRunning ? 'Pause' : 'Resume'"/>
        </button>
      </div>
    </div>
  </t>
</templates>
```

Advanced Cleanup Example: Multiple Resources

JavaScript (`notification_system.js`):

```
import { Component, onMounted, onWillUnmount, useState } from "@odoo/owl";

export class NotificationSystem extends Component {
  static template = "my_module.NotificationSystem";

  setup() {
    this.state = useState({
      notifications: [],
      isConnected: false
    });

    // Store cleanup references
    this.cleanupTasks = [];

    onMounted(() => {
      this.initializeSystem();
    });

    onWillUnmount(() => {
      console.log("NotificationSystem: Performing comprehensive cleanup");
      this.performCleanup();
    });
  }

  initializeSystem() {
    // 1. Set up periodic health check
    const healthCheckTimer = setInterval(() => {
      this.checkSystemHealth();
    }, 5000);

    this.cleanupTasks.push(() => {
      clearInterval(healthCheckTimer);
      console.log("NotificationSystem: Health check timer cleared");
    });
```

```
    // 2. Set up window focus/blur listeners
    const handleWindowFocus = () => {
      console.log("NotificationSystem: Window focused, refreshing notifications");
      this.refreshNotifications();
    };

    const handleWindowBlur = () => {
      console.log("NotificationSystem: Window blurred");
    };

    window.addEventListener('focus', handleWindowFocus);
    window.addEventListener('blur', handleWindowBlur);

    this.cleanupTasks.push(() => {
      window.removeEventListener('focus', handleWindowFocus);
      window.removeEventListener('blur', handleWindowBlur);
      console.log("NotificationSystem: Window event listeners removed");
    });

    // 3. Set up online/offline listeners
    const handleOnline = () => {
      this.state.isConnected = true;
      this.refreshNotifications();
    };

    const handleOffline = () => {
      this.state.isConnected = false;
    };

    window.addEventListener('online', handleOnline);
    window.addEventListener('offline', handleOffline);

    this.cleanupTasks.push(() => {
      window.removeEventListener('online', handleOnline);
      window.removeEventListener('offline', handleOffline);
      console.log("NotificationSystem: Network event listeners removed");
    });

    // 4. Initialize connection state
    this.state.isConnected = navigator.onLine;
  }
```

Notice the pattern: every time `initializeSystem` sets something up (a timer, a pair of listeners), it immediately pushes a matching "undo" function onto `this.cleanupTasks`. That's what makes the single call to `performCleanup()` below able to reverse everything, no matter how many resources were started.

```
  checkSystemHealth() {
    // Simulate system health check
    console.log("NotificationSystem: Performing health check");
  }

  refreshNotifications() {
    // Simulate notification refresh
    console.log("NotificationSystem: Refreshing notifications");
```

```
  }

  performCleanup() {
    // Execute all cleanup tasks
    this.cleanupTasks.forEach((cleanup, index) => {
      try {
        cleanup();
      } catch (error) {
        console.error(`NotificationSystem: Cleanup task ${index} failed:`, error);
      }
    });

    // Clear the cleanup tasks array
    this.cleanupTasks = [];

    console.log("NotificationSystem: All cleanup tasks completed");
  }
}
```

If you forget to cleanup in `onWillUnmount`, resources like timers would keep running in the background forever, even after the component is gone, creating memory leaks. Using `onWillUnmount` for cleanup is a critical part of writing robust applications.

Lifecycle Best Practices

DO's

1. **Always cleanup in onWillUnmount** - Clear timers, remove event listeners, destroy third-party instances
2. **Use onWillStart for critical data** - Data that must be ready before first render
3. **Make onWillStart async** - Take advantage of its unique async capability
4. **Handle errors gracefully** - Wrap async operations in try-catch blocks
5. **Log lifecycle events** - Helps with debugging during development

DON'Ts

1. **Never mutate props** - Props are read-only in all lifecycle hooks
2. **Don't forget cleanup** - Memory leaks will hurt your application's performance
3. **Don't assume DOM exists** - Only `onMounted` and later hooks have DOM access
4. **Don't perform heavy computations** - Keep lifecycle hooks fast and focused

Common Patterns

Pattern 1: Conditional Data Loading

```
onWillUpdateProps(async (nextProps) => {
  // Only fetch if the important prop changed
  if (this.props.customerId !== nextProps.customerId) {
    await this.loadCustomerData(nextProps.customerId);
  }
});
```

Pattern 2: Cleanup Helper

```
setup() {
  this.cleanupFunctions = [];

  onMounted(() => {
    // Set up something that needs cleanup
    const timer = setInterval(this.updateData, 1000);
    this.cleanupFunctions.push(() => clearInterval(timer));
  });

  onWillUnmount(() => {
    this.cleanupFunctions.forEach(cleanup => cleanup());
  });
}
```

Pattern 3: Error Boundary

```
onWillStart(async () => {
  try {
    await this.loadCriticalData();
  } catch (error) {
    console.error("Failed to load critical data:", error);
    this.state.error = "System temporarily unavailable";
    // Could also dispatch to parent or show fallback UI
  }
});
```

Debugging Lifecycle Issues

When working with lifecycle hooks, common issues include:

Issue: Component doesn't update when props change **Solution**: Add `onWillUpdateProps` handler:

```
onWillUpdateProps(async (nextProps) => {
  if (this.props.dataId !== nextProps.dataId) {
    await this.loadNewData(nextProps.dataId);
  }
});
```

Issue: Memory leaks from forgotten timers **Solution**: Always clear in `onWillUnmount`:

```
onMounted(() => {
  this.timer = setInterval(this.doSomething, 1000);
});

onWillUnmount(() => {
  if (this.timer) {
    clearInterval(this.timer);
  }
});
```

Issue: Component renders empty then flickers when data loads **Solution**: Use `onWillStart` instead of `onMounted`:

```
// This causes flicker
onMounted(async () => {
  this.state.data = await this.fetchData();
```

```
});

// This prevents flicker
onWillStart(async () => {
  this.state.data = await this.fetchData();
});
```

Error Handling with `onError` and Error Boundaries

Everything so far assumes rendering goes smoothly. It doesn't always. If a child component throws while rendering—a `null` reference, a bad API response shape, anything—OWL's default behavior is to destroy the **entire application**, not just the component that failed. One bug in a small widget can take down the whole page.

An **error boundary** is a component that catches errors thrown anywhere in its subtree and shows a fallback UI instead of letting the crash propagate. OWL provides the `onError` hook for exactly this.

`onError(callback)` registers a function that OWL calls whenever a rendering or lifecycle error occurs in one of this component's descendants. It does **not** catch errors thrown inside event handlers (a `t-on-click` handler that throws is your own `try/catch` territory)—only errors from the render/lifecycle machinery.

```
import { Component, useState, onError, xml } from "@odoo/owl";

class ErrorBoundary extends Component {
    static template = xml`
        <t t-if="state.error" t-slot="fallback">
            <div class="alert alert-danger">Something went wrong.</div>
        </t>
        <t t-else="" t-slot="default"/>`;

    setup() {
        this.state = useState({ error: false });
        onError(() => {
            this.state.error = true;
        });
    }
}
```

Wrap any part of the tree you don't fully trust (a widget rendering user-generated content, a third-party integration) with this component, passing the risky content as its default slot and an optional `fallback` slot for a custom message. If your own `onError` handler can't recover, it can re-throw the error so a boundary further up the tree gets a chance to handle it. Just make sure the fallback template itself never throws—that would create an infinite loop of failures.

Other Lifecycle Hooks

`onWillStart`, `onMounted`, `onWillUpdateProps`, and `onWillUnmount` cover the hooks you'll reach for daily. OWL exposes a few more for less common, more advanced needs—know they exist so you recognize them in code you didn't write:

- **`onWillRender()`** — runs immediately before a component's template function executes (parents before children). Rarely needed directly.

- **`onRendered()`** — runs immediately after the template function executes, but before the result is applied to the real DOM. Also rarely needed directly.
- **`onWillPatch()`** — runs just before an *already-mounted* component's DOM is updated (parents before children). Useful for reading DOM state—like a scroll position—before it changes.
- **`onPatched()`** — runs just after an already-mounted component's DOM has been updated (children before parents). This is the one you'll actually use sometimes: for example, re-measuring an element's size after its content changed.

```
onPatched(() => {
    // the DOM has just been updated with the latest state/props
    this.scrollToBottom();
});
```

- **`onWillDestroy()`** — always called when a component is being torn down, even if it never finished mounting. Use it for cleanup that must happen no matter what (as opposed to `onWillUnmount`, which only fires for components that were actually mounted).

For a junior developer, the practical rule is: reach for the four hooks from earlier in this chapter first, and only look at this second group when you have a specific, DOM-update-related problem to solve.

Common Pitfalls

Pitfall 1: Fetching Data in `onMounted` Instead of `onWillStart`

Fetching in `onMounted` renders the component empty first, then flickers once data arrives. Use `onWillStart` for data the component needs for its very first render.

Pitfall 2: Forgetting to Clean Up in `onWillUnmount`

Timers, `window`/`document` listeners, and third-party library instances started in `onMounted` keep running after the component is destroyed unless you clear them in `onWillUnmount`—a classic source of memory leaks.

Pitfall 3: Assuming the DOM Exists Too Early

`this.someRef.el` is only guaranteed to exist from `onMounted` onward. Accessing a `useRef` inside `setup()` or `onWillStart` will give you `undefined`.

Pitfall 4: Reloading Data on Every `onWillUpdateProps` Call

`onWillUpdateProps` runs whenever *any* prop changes, not just the one you care about. Always compare the old and new values (`this.props.x !== nextProps.x`) before re-fetching.

Pitfall 5: Forgetting an Error Boundary Anywhere in the Tree

Without an `onError` handler somewhere above it, a single throwing child component crashes the *entire* application, not just itself. Wrap risky subtrees (third-party widgets, anything rendering unpredictable data) in an `ErrorBoundary` component.

Exercises

Exercise 1: Fix the Flicker

Take a component that loads its data inside `onMounted` and move the fetch into `onWillStart` instead. Confirm in the browser that the "loading" flash disappears on first render.

Exercise 2: Plug the Leak

Write a component that starts a `setInterval` in `onMounted` without cleaning it up. Open the console, mount and unmount the component a few times, and observe the timer keeps firing. Then fix it with `onWillUnmount`.

Exercise 3: Selective Reloading

Build a component with two props, `userId` and `theme`. In `onWillUpdateProps`, only re-fetch user data when `userId` changes—not when `theme` changes. Add `console.log` calls to prove the fetch is skipped for theme-only updates.

Exercise 4: Build an Error Boundary

Write a `Buggy` component whose `setup()` throws if a `crash` prop is `true`. Wrap it in the `ErrorBoundary` component from this chapter and confirm that when `crash` is `true`, the rest of the page keeps working and shows the fallback message instead of a blank screen.

Summary

Lifecycle hooks are the foundation of robust OWL components. They let you:

- **`onWillStart`**: Fetch critical data before rendering
- **`onMounted`**: Interact with the DOM and initialize third-party libraries

- **`onWillUpdateProps`**: React to prop changes and fetch new data
- **`onWillUnmount`**: Clean up resources to prevent memory leaks

Master these hooks, and you'll be able to create components that are not only functional, but also performant and reliable.

TL;DR: `onWillStart` loads data before the first render, `onMounted` touches the DOM, `onWillUpdateProps` reacts to new props, and `onWillUnmount` cleans up — miss the last one and you leak memory.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch10_ex1`. Installation instructions are in the repository README.

What's Next?

In the next chapter, we'll explore the power of custom hooks and how they can help you create reusable lifecycle patterns across your application.

Chapter 11: Modern Hooks - useEffect and useRef

Why this chapter? Lifecycle hooks alone push you toward writing code organized by "when," not "why" — a timer's setup ends up in `onMounted`, its cleanup in `onWillUnmount`, and the two drift apart as the component grows. Modern hooks let you keep everything about one concern together.

What is Odoo trying to solve with this? Odoo's own backend components juggle a lot of external resources — DOM measurements, subscriptions, timers, focus management — and `useEffect`/`useRef` give a single, predictable place to set them up and tear them down without scattering that logic across multiple lifecycle methods.

Real-world application: Debounced search boxes, auto-focus on a newly opened dialog, and syncing a chart library to component state are all things I've shipped in client addons using exactly the patterns in this chapter.

In Chapter 10, we explored the lifecycle hooks like `onMounted`, `onWillStart`, and `onWillUnmount`. These hooks are powerful and intuitive, but modern OWL development encourages a more functional and flexible approach using **modern hooks**, primarily `useEffect` and `useRef`.

Modern hooks represent a paradigm shift in how we think about component logic. Instead of organizing code by lifecycle events ("when the component mounts, do this"), we organize it by **features** or **concerns** ("everything related to this timer lives together"). This approach makes code more maintainable, testable, and easier to reason about.

Let's dive deep into these modern hooks and learn how they can make your components more powerful and elegant.

`useRef`: Beyond DOM References

We briefly encountered `useRef` in Chapter 10 for accessing DOM elements. While DOM access is its most common use case, `useRef` is actually a much more versatile hook for creating **stable references** to any value.

Understanding Refs

In OWL, a ref is an object with a single `.el` property that points to the DOM element marked with the matching `t-ref` directive in your template. The key characteristic of a ref is that it's **stable** across re-renders—it always points to the same object instance, and `.el` is kept up to date as the DOM changes.

> **Coming from React?** OWL has no "value refs" with a `.current` property. It doesn't need them: OWL components are class instances, so for mutable values that shouldn't

trigger re-renders (timer IDs, previous values, library instances) you simply use plain instance properties like `this.timer`. `useRef` in OWL is exclusively for DOM access.

DOM References: The Classic Use Case

Let's start with the familiar DOM reference pattern:

JavaScript (`focus_manager.js`):

```
import { Component, useRef, useState } from "@odoo/owl";

export class FocusManager extends Component {
  static template = "my_module.FocusManager";

  setup() {
    this.state = useState({
      inputValue: "",
      focusCount: 0
    });

    // Create refs for multiple elements
    this.nameInputRef = useRef("name-input");
    this.emailInputRef = useRef("email-input");
    this.submitButtonRef = useRef("submit-button");
  }

  focusName() {
    if (this.nameInputRef.el) {
      this.nameInputRef.el.focus();
      this.state.focusCount++;
      console.log("FocusManager: Focused name input");
    }
  }

  focusEmail() {
    if (this.emailInputRef.el) {
      this.emailInputRef.el.focus();
      this.state.focusCount++;
      console.log("FocusManager: Focused email input");
    }
  }

  focusSubmit() {
    if (this.submitButtonRef.el) {
      this.submitButtonRef.el.focus();
      this.state.focusCount++;
      console.log("FocusManager: Focused submit button");
    }
  }

  handleKeyDown(event) {
    // Navigate between fields with Tab
    if (event.key === "Tab" && event.shiftKey) {
      // Handle custom tab navigation if needed
      console.log("FocusManager: Shift+Tab pressed");
    }
```

```
  }

  onInputChange(event) {
    this.state.inputValue = event.target.value;

    // Auto-focus next field when current is filled
    if (event.target === this.nameInputRef.el && event.target.value.length > 2) {
      setTimeout(() => this.focusEmail(), 100);
    }
  }
}
```

Template (`focus_manager.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.FocusManager" owl="1">
    <div class="focus-manager card">
      <div class="card-header">
        <h5>Focus Manager Demo</h5>
        <small class="text-muted">Focus events: <t t-esc="state.focusCount"/></small>
      </div>

      <div class="card-body">
        <form t-on-keydown="handleKeyDown">
          <div class="mb-3">
            <label for="name" class="form-label">Name</label>
            <input
                type="text"
                class="form-control"
                t-ref="name-input"
                placeholder="Enter your name"
                t-model="state.inputValue"
                t-on-input="onInputChange"
            />
          </div>

          <div class="mb-3">
            <label for="email" class="form-label">Email</label>
            <input
                type="email"
                class="form-control"
                t-ref="email-input"
                placeholder="Enter your email"
            />
          </div>

          <button
              type="submit"
              class="btn btn-primary me-2"
              t-ref="submit-button">
            Submit
          </button>
        </form>

        <div class="mt-3">
```

```
          <small class="text-muted">Quick Focus:</small><br/>
          <button class="btn btn-sm btn-outline-secondary me-1" t-on-click="focusName">
            Focus Name
          </button>
          <button class="btn btn-sm btn-outline-secondary me-1" t-on-click="focusEmail">
            Focus Email
          </button>
          <button class="btn btn-sm btn-outline-secondary" t-on-click="focusSubmit">
            Focus Submit
          </button>
        </div>
      </div>
    </div>
  </t>
</templates>
```

Storing Mutable Data: Plain Instance Properties

What about mutable values that **should not trigger re-renders** when changed—timer IDs, previous values for comparison, third-party library instances? In OWL you don't need a hook for these at all. Because your component is a class instance, a plain property does the job:

Example: Timer with Instance Properties

```
import { Component, useState } from "@odoo/owl";

export class TimerComponent extends Component {
  static template = "my_module.TimerComponent";

  setup() {
    this.state = useState({
      seconds: 0,
      isRunning: false
    });

    // Plain instance properties - changing them never causes a re-render
    this.timer = null;             // timer ID
    this.previousSeconds = 0;      // previous value for comparison
    this.startCount = 0;           // how many times the timer was started
  }

  startTimer() {
    if (this.state.isRunning) return;

    console.log("TimerComponent: Starting timer");
    this.state.isRunning = true;
    this.startCount += 1;

    // Store timer ID on the instance - this won't trigger re-render
    this.timer = setInterval(() => {
      this.state.seconds++;

      // Log every 10 seconds using previous value comparison
      if (this.state.seconds % 10 === 0 && this.state.seconds !== this.previousSeconds) {
        console.log(`TimerComponent: ${this.state.seconds} seconds elapsed`);
        this.previousSeconds = this.state.seconds;
```

```
      }
    }, 1000);
  }

  stopTimer() {
    if (!this.state.isRunning) return;

    console.log("TimerComponent: Stopping timer");
    this.state.isRunning = false;

    // Clear timer using stored ID
    if (this.timer) {
      clearInterval(this.timer);
      this.timer = null;
    }
  }

  resetTimer() {
    this.stopTimer();
    this.state.seconds = 0;
    this.previousSeconds = 0;
    console.log("TimerComponent: Timer reset");
  }

  get timerStats() {
    return {
      currentTime: this.formatTime(this.state.seconds),
      startCount: this.startCount,
      isActive: this.timer !== null
    };
  }

  formatTime(seconds) {
    const mins = Math.floor(seconds / 60);
    const secs = seconds % 60;
    return `${mins.toString().padStart(2, '0')}:${secs.toString().padStart(2, '0')}`;
  }
}
```

The rule of thumb: **reactive data that the template shows goes in `useState`; everything else is a plain instance property; `useRef` is only for DOM elements.**

`useEffect`: The Swiss Army Knife of Side Effects

The `useEffect` hook is the most powerful and flexible hook in modern OWL. It replaces multiple lifecycle hooks with a single, unified API that groups related logic together.

> **What's a "side effect"?** It's anything a piece of code does that reaches outside of computing its return value — starting a timer, adding an event listener, calling the server, changing `document.title`. Rendering should be a pure calculation of "state in, DOM out"; `useEffect` is where you put the parts that aren't.

Understanding Dependencies

The magic of `useEffect` lies in its second parameter: a **dependency function** that returns the list of values the effect depends on. OWL calls this function on every render and re-runs the effect only when one of the returned values changed.

> **Careful if you know React:** in React the dependencies are a plain array (`[state.count]`). In OWL they are a *function that returns an array* (`() => [state.count]`). Passing a plain array is a common migration bug.

```
// Runs once after mount, cleanup on unmount (like onMounted + onWillUnmount)
useEffect(
  () => {
    console.log("Component mounted");
  },
  () => []
);

// Runs after every render (usually not what you want) - omit the dependency function
useEffect(() => {
  console.log("Component rendered");
});

// Runs when specific values change
useEffect(
  () => {
    console.log("Count or name changed");
  },
  () => [this.state.count, this.state.name]
);
```

Pattern 1: Mount-Only Effects (Replacing onMounted)

Let's create a component that sets up multiple things when it mounts:

JavaScript (`dashboard.js`):

```
import { Component, useEffect, useRef, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class Dashboard extends Component {
  static template = "my_module.Dashboard";

  setup() {
    this.state = useState({
      currentTime: new Date().toLocaleTimeString(),
      stats: {
        users: 0,
        orders: 0,
        revenue: 0
      },
      loading: true
    });

    this.orm = useService("orm");
    this.titleRef = useRef("page-title");
```

```
    // Mount-only effect - runs once after component mounts
    useEffect(() => {
      console.log("Dashboard: Component mounted, initializing...");

      // 1. Set up clock
      const clockInterval = setInterval(() => {
        this.state.currentTime = new Date().toLocaleTimeString();
      }, 1000);

      // 2. Focus the title
      if (this.titleRef.el) {
        this.titleRef.el.focus();
      }

      // 3. Load initial data
      this.loadDashboardData();

      // 4. Set up window focus listener
      const handleWindowFocus = () => {
        console.log("Dashboard: Window focused, refreshing data");
        this.loadDashboardData();
      };

      window.addEventListener('focus', handleWindowFocus);

      // Cleanup function - runs when component unmounts
      return () => {
        console.log("Dashboard: Cleaning up mount effects");
        clearInterval(clockInterval);
        window.removeEventListener('focus', handleWindowFocus);
      };
    }, () => []); // Empty dependency list = run once on mount
  }

  async loadDashboardData() {
    try {
      console.log("Dashboard: Loading dashboard statistics");

      // Simulate loading multiple stats in parallel
      const [users, orders] = await Promise.all([
        this.orm.searchCount("res.users", []),
        this.orm.searchCount("sale.order", [["state", "=", "sale"]])
      ]);

      this.state.stats = {
        users,
        orders,
        revenue: orders * 150 // Simulated revenue calculation
      };

      this.state.loading = false;
      console.log("Dashboard: Statistics loaded successfully");

    } catch (error) {
      console.error("Dashboard: Failed to load statistics:", error);
```

```
      this.state.loading = false;
    }
  }
}
```

Pattern 2: Reactive Effects (Replacing onWillUpdateProps)

Effects can react to specific state or prop changes:

JavaScript (`user_profile.js`):

```
import { Component, useEffect, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class UserProfile extends Component {
  static template = "my_module.UserProfile";
  static props = {
    userId: { type: Number }
  };

  setup() {
    this.state = useState({
      user: null,
      loading: false,
      error: null,
      loadCount: 0
    });

    this.orm = useService("orm");

    // Effect that reacts to userId prop changes
    useEffect(() => {
      if (!this.props.userId) {
        console.log("UserProfile: No user ID provided");
        this.state.user = null;
        return;
      }

      console.log(`UserProfile: Loading user data for ID ${this.props.userId}`);
      this.loadUserData(this.props.userId);

    }, () => [this.props.userId]); // Runs when userId changes

    // Separate effect for logging load attempts
    useEffect(() => {
      if (this.state.loadCount > 0) {
        console.log(`UserProfile: Load attempt #${this.state.loadCount}`);
      }
    }, () => [this.state.loadCount]);
  }

  async loadUserData(userId) {
    this.state.loading = true;
    this.state.error = null;
    this.state.loadCount += 1;
```

```
    try {
      const users = await this.orm.read("res.users", [userId], [
        "name",
        "email",
        "phone",
        "partner_id",
        "login_date"
      ]);

      if (users.length > 0) {
        this.state.user = users[0];
        console.log(`UserProfile: Successfully loaded ${users[0].name}`);
      } else {
        throw new Error(`User ${userId} not found`);
      }

    } catch (error) {
      console.error(`UserProfile: Failed to load user ${userId}:`, error);
      this.state.error = error.message;
      this.state.user = null;
    } finally {
      this.state.loading = false;
    }
  }
}
```

Pattern 3: Multiple Independent Effects

One of the biggest advantages of `useEffect` is that you can have multiple effects, each handling a specific concern:

JavaScript (`notification_center.js`):

First, the component sets up its state and its first effect, which only cares about whether the browser is online:

```
import { Component, useEffect, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class NotificationCenter extends Component {
  static template = "my_module.NotificationCenter";

  setup() {
    this.state = useState({
      notifications: [],
      connectionStatus: 'connecting',
      lastUpdate: null,
      unreadCount: 0
    });

    this.orm = useService("orm");

    // Effect 1: Connection management
    useEffect(() => {
      console.log("NotificationCenter: Setting up connection monitoring");
```

```
    const checkConnection = () => {
      this.state.connectionStatus = navigator.onLine ? 'connected' : 'disconnected';
    };

    // Initial check
    checkConnection();

    // Set up listeners
    window.addEventListener('online', checkConnection);
    window.addEventListener('offline', checkConnection);

    return () => {
      window.removeEventListener('online', checkConnection);
      window.removeEventListener('offline', checkConnection);
    };
  }, () => []);
```

Next, a second effect watches `connectionStatus` and only starts polling for new notifications once the browser is actually online — notice its dependency function returns `[this.state.connectionStatus]`, so it re-runs whenever that value flips:

```
  // Effect 2: Periodic notification refresh
  useEffect(() => {
    if (this.state.connectionStatus !== 'connected') {
      console.log("NotificationCenter: Skipping refresh - not connected");
      return;
    }

    console.log("NotificationCenter: Setting up periodic refresh");

    const refreshInterval = setInterval(() => {
      this.refreshNotifications();
    }, 30000); // Refresh every 30 seconds

    // Initial load
    this.refreshNotifications();

    return () => {
      clearInterval(refreshInterval);
    };
  }, () => [this.state.connectionStatus]); // Re-run when connection status changes
```

Finally, a third, completely independent effect recalculates the unread count and keeps the browser tab title in sync whenever the notification list changes — it doesn't know or care about the connection logic above:

```
  // Effect 3: Unread count calculation
  useEffect(() => {
    const unread = this.state.notifications.filter(n => !n.is_read).length;
    this.state.unreadCount = unread;

    console.log(`NotificationCenter: ${unread} unread notifications`);

    // Update browser title if there are unread notifications
    if (unread > 0) {
      document.title = `(${unread}) Odoo - Notifications`;
    } else {
```

```
        document.title = "Odoo";
      }

      return () => {
        // Cleanup title on unmount
        document.title = "Odoo";
      };
    }, () => [this.state.notifications.length]);
  }

  async refreshNotifications() {
    try {
      console.log("NotificationCenter: Refreshing notifications");

      const notifications = await this.orm.searchRead(
        "mail.notification",
        [["res_partner_id", "=", this.currentUserId]],
        ["id", "mail_message_id", "is_read", "create_date"],
        {
          order: "create_date desc",
          limit: 50
        }
      );

      this.state.notifications = notifications;
      this.state.lastUpdate = new Date();

    } catch (error) {
      console.error("NotificationCenter: Failed to refresh notifications:", error);
    }
  }

  get currentUserId() {
    // This would come from a user service in a real app
    return 1; // Simplified for example
  }
}
```

Three effects, three concerns, zero tangled logic — that's the payoff of organizing by feature instead of by lifecycle moment.

Advanced useEffect Patterns

Pattern: Effect with Async Logic

```
useEffect(() => {
  let cancelled = false;

  const fetchData = async () => {
    try {
      const result = await this.orm.searchRead("some.model", [], []);

      // Only update state if effect hasn't been cancelled
      if (!cancelled) {
        this.state.data = result;
      }
```

```
    } catch (error) {
      if (!cancelled) {
        console.error("Fetch failed:", error);
      }
    }
  };

  fetchData();

  // Cleanup: mark as cancelled to prevent state updates
  return () => {
    cancelled = true;
  };
}, () => [this.state.someDependency]);
```

Pattern: Debounced Effect

Debouncing means delaying a piece of work until a burst of triggering changes has paused for a set amount of time — instead of searching on every keystroke, you wait until the user stops typing for 300ms.

```
useEffect(() => {
  const timeoutId = setTimeout(() => {
    // Perform expensive operation only after value has been stable for 300ms
    this.performSearch(this.state.searchTerm);
  }, 300);

  return () => {
    clearTimeout(timeoutId);
  };
}, () => [this.state.searchTerm]);
```

Combining useRef and useEffect: Powerful Patterns

When you combine `useRef` and `useEffect`, you unlock some very powerful patterns:

Pattern: Previous Value Comparison

```
setup() {
  this.state = useState({ count: 0 });
  this.previousCount = undefined; // plain instance property

  useEffect(
    () => {
      console.log(`Count changed from ${this.previousCount} to ${this.state.count}`);

      // Store current value for next comparison
      this.previousCount = this.state.count;
    },
    () => [this.state.count]
  );
}
```

Pattern: Third-Party Library Integration

```
setup() {
  this.chartCanvasRef = useRef("chart-canvas"); // DOM ref (t-ref="chart-canvas")
  this.chart = null;                            // plain property for the library
      ↪ instance
  this.state = useState({ data: [] });

  // Set up chart on mount
  useEffect(
    () => {
      if (!this.chartCanvasRef.el) return;

      const ctx = this.chartCanvasRef.el.getContext('2d');
      this.chart = new Chart(ctx, {
        type: 'line',
        data: { datasets: [] }
      });

      return () => {
        if (this.chart) {
          this.chart.destroy();
        }
      };
    },
    () => []
  );

  // Update chart when data changes
  useEffect(
    () => {
      if (this.chart) {
        this.chart.data = this.processChartData();
        this.chart.update();
      }
    },
    () => [this.state.data.length]
  );
}
```

Migration Guide: From Lifecycle to Modern Hooks

If you have existing components using lifecycle hooks, here's how to migrate them:

Before: Lifecycle Hooks

```
setup() {
  this.state = useState({ data: null });
  this.orm = useService("orm");

  onWillStart(async () => {
    this.state.data = await this.orm.searchRead("model", [], []);
  });
```

```
  onMounted(() => {
    document.title = "My Component";
    this.timer = setInterval(this.refresh, 5000);
  });

  onWillUnmount(() => {
    document.title = "Odoo";
    if (this.timer) {
      clearInterval(this.timer);
    }
  });
}
```

After: Modern Hooks

```
setup() {
  this.state = useState({ data: null });
  this.orm = useService("orm");

  // Keep onWillStart for pre-render data: useEffect runs AFTER mounting,
  // so replacing it with an effect would bring back the empty-then-flicker render
  onWillStart(async () => {
    this.state.data = await this.orm.searchRead("model", [], []);
  });

  // Mount effects (replaces onMounted + onWillUnmount)
  useEffect(
    () => {
      document.title = "My Component";
      const timer = setInterval(this.refresh, 5000);

      return () => {
        document.title = "Odoo";
        clearInterval(timer);
      };
    },
    () => []
  );
}
```

Note that `onWillStart` stays: it's the only hook that can delay the **first** render, so it has no `useEffect` equivalent.

Environment Hooks: `useEnv`, `useSubEnv`, `useChildSubEnv`

Props are great for passing data one level down, from a parent to its direct child. But what about data that dozens of components scattered across the tree all need—the current user, a translation function, feature flags? Passing it down as a prop through every intermediate component (**prop drilling**, the same problem that motivates the global state store in Chapter 15) gets painful fast.

OWL's answer is the **environment** (`env`): a plain object shared by every component in the tree, set once when the application is mounted and available everywhere without threading it through props.

`useEnv()` reads the current component's environment:

```
import { Component, useEnv } from "@odoo/owl";

class UserBadge extends Component {
    static template = xml`<span t-esc="env.currentUser.name"/>`;

    setup() {
        this.env = useEnv();
    }
}
```

Sometimes a component wants to add or override something in the environment for its own subtree—a theme, a scoped configuration—without changing it for the whole app. Two hooks do this, and the difference between them matters:

- **`useSubEnv(newValues)`** — merges `newValues` into the environment for **this component and all of its children**.
- **`useChildSubEnv(newValues)`** — merges `newValues` into the environment for **only the children**, leaving this component's own `env` untouched.

```
setup() {
    // Every component below this one (children, grandchildren, ...) now
    // sees env.theme === "dark". This component itself does too, because
    // useSubEnv affects the caller as well.
    useSubEnv({ theme: "dark" });
}
```

Use `useChildSubEnv` when a component wants to configure its descendants without implying that the same configuration applies to itself—for example, a `<Section title="...">` component that sets a heading level for its children without becoming a heading itself.

`useExternalListener`: Listening Outside the Component

Chapter 1 attached a `click` listener by hand with `addEventListener`, and had to remember to call `removeEventListener` in `onWillUnmount` to avoid a leak. `useExternalListener` does both steps for you in one call, which is exactly what you want for listening on `window` or `document`—targets outside the component's own DOM that OWL doesn't manage.

```
import { Component, useExternalListener } from "@odoo/owl";

class DropdownMenu extends Component {
    setup() {
        // Automatically added on mount and removed on unmount-no
        // onMounted/onWillUnmount pair to write or forget.
        useExternalListener(window, "click", this.closeIfOutside.bind(this));
    }

    closeIfOutside(ev) {
        if (!this.el.contains(ev.target)) {
            this.state.open = false;
        }
    }
}
```

It takes the target (`window`, `document`, or any DOM node), the event name, and a handler, and forwards any extra arguments to `addEventListener`—so `useExternalListener(window, "keydown",`

`this.onKeyDown, { capture: true })` works too.

One more hook worth knowing about, even though you'll rarely call it directly: `useComponent()` returns the current component instance. It exists mainly as a building block for writing your *own* custom hooks that need to reach the calling component—application code almost never needs it.

Best Practices and Common Pitfalls

Do's

1. **Group related logic together** - Put setup and cleanup for the same feature in one effect
2. **Use multiple effects** - Separate concerns into different effects
3. **Include all dependencies** - The dependency function should return every reactive value the effect relies on
4. **Return cleanup functions** - Prevent memory leaks by cleaning up resources
5. **Use the right storage** - Reactive template data in `useState`, mutable non-reactive values (timer IDs, previous values, library instances) as plain instance properties, `useRef` only for DOM elements

Don'ts

1. **Don't pass a plain array as dependencies** - OWL expects a function returning an array (`() => [..]`), not the array itself
2. **Don't use effects for everything** - Some logic belongs in event handlers, not effects
3. **Don't replace `onWillStart` with an effect** - Effects run after mounting; only `onWillStart` can delay the first render
4. **Don't over-complicate** - Sometimes lifecycle hooks are clearer for simple cases

Common Pitfall: Array Instead of Function

```
// BAD: React-style plain array - OWL will not track these dependencies
useEffect(() => {
  this.syncWithServer(this.state.filter);
}, [this.state.filter]);

// GOOD: a function that returns the dependency array
useEffect(
  () => {
    this.syncWithServer(this.state.filter);
  },
  () => [this.state.filter]
);
```

Why a function? OWL calls it on **every render** to get fresh values and compare them with the previous ones. A plain array would be evaluated only once, when `setup()` ran.

Common Pitfall: Expecting `useSubEnv` Not to Touch the Caller

```
// This component's OWN env.theme also becomes "dark" here, not just
// its children's-useSubEnv always includes the caller.
useSubEnv({ theme: "dark" });
```

If you want to configure descendants without changing anything for the current component itself, use `useChildSubEnv` instead. Reaching for the wrong one of the two is an easy, easy-to-miss bug: everything below the component behaves correctly, but the component's own rendering unexpectedly changes too.

Summary

Modern hooks represent a significant evolution in how we write components:

- **`useRef`** creates stable references to DOM elements or any mutable value
- **`useEffect`** handles all side effects with precise control over when they run
- **Dependency arrays** make effects predictable and performant
- **Cleanup functions** prevent memory leaks and resource conflicts

The key insight is organizational: instead of thinking "what happens when the component mounts," think "what does this feature need to work properly?" Group all the logic for each feature—setup, updates, and cleanup—into focused effects.

This approach makes your components more maintainable, testable, and easier to understand. In the next chapter, we'll explore how to communicate with the server using services, building on the solid foundation of modern hooks we've established here.

TL;DR: `useRef` gives you a stable handle to a DOM element or a mutable value; `useEffect` runs (and cleans up) side effects based on an explicit dependency function, so you group setup and teardown for one concern in one place instead of spreading it across lifecycle hooks.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch11_ex1`. Installation instructions are in the repository README.

Exercises

1. **Focus-on-mount:** Build a small component with a text input and a `useRef`. Use `useEffect` with an empty dependency function (`() => []`) to focus the input as soon as the component mounts.
2. **Fix the dependency bug:** Take this broken snippet and fix it so the effect actually re-runs when `state.query` changes:

   ```
   useEffect(() => {
     console.log("Searching for", this.state.query);
   }, [this.state.query]);
   ```

3. **Debounced counter:** Add a `useEffect` that logs `"Stable!"` to the console only after `state.count` hasn't changed for 500ms (hint: `setTimeout` in the effect, `clearTimeout` in the cleanup function — the same shape as the debounced search example above).

What's Next?

You now have components that manage their own DOM references and side effects — but real Odoo addons rarely live in isolation from the server. Chapter 12 covers `orm` and `rpc`, the services that let your components read and write actual business data.

Chapter 12: Server Communication with useService

> **Why this chapter?** Every non-trivial Odoo addon eventually needs to read or write real records — a component that only manages local state is a toy. This chapter is where your components start talking to the actual database.
>
> **What is Odoo trying to solve with this?** Odoo needs a consistent, safe way for frontend code to reach the ORM and custom controllers without every developer hand-rolling their own AJAX calls, error handling, and security checks.
>
> **Real-world application:** Every dashboard widget, custom kanban action, and portal form I've built for a client eventually comes down to `orm.searchRead/create/write` calls exactly like the ones in this chapter.

In the previous chapters, we've built components that can manage state, respond to events, and handle complex lifecycle scenarios. But a modern web application isn't complete without the ability to communicate with the server. Your beautiful OWL components need to fetch real data, save user changes, and integrate seamlessly with Odoo's backend.

This is where the **service system** comes in. Odoo's service architecture provides a clean, consistent way for your frontend components to communicate with the Python backend, display notifications, trigger actions, and much more.

The gateway to this powerful system is the `useService` hook—your components' bridge to the entire Odoo ecosystem.

Understanding Odoo's Service Architecture

Services in Odoo are **singleton objects** that provide specific functionality across your entire application. They're designed to be:

- **Centralized**: One instance per service type
- **Consistent**: Same API everywhere in the app
- **Integrated**: Seamlessly connected to Odoo's backend
- **Testable**: Easy to mock and test

Think of services as your application's "utilities"—specialized tools that handle specific concerns so your components can focus on their primary job: rendering UI and handling user interactions.

The `useService` Hook

The `useService` hook is remarkably simple:

```
import { useService } from "@web/core/utils/hooks";

// Inside your component's setup()
const serviceName = useService("service_name");
```

That's it! You get a fully configured service instance ready to use. Let's explore the most important services you'll use daily.

The `orm` Service: Your Database Gateway

The `orm` service is your primary tool for database operations. It provides a clean, Promise-based API that mirrors Odoo's Python ORM methods, making server communication feel natural and intuitive.

Basic CRUD Operations

Let's build a comprehensive product management component that demonstrates all the core ORM operations:

JavaScript (`product_manager.js`):

```
import { Component, useEffect, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ProductManager extends Component {
  static template = "my_module.ProductManager";

  setup() {
    this.state = useState({
      products: [],
      categories: [],
      loading: true,
      error: null,
      selectedProduct: null,
      formData: {
        name: "",
        list_price: 0,
        categ_id: null,
        description: ""
      }
    });

    // Get the ORM service
    this.orm = useService("orm");

    // Load initial data when component mounts
    useEffect(
      () => {
        this.loadInitialData();
      },
      () => []
    );
  }

  async loadInitialData() {
```

```
    try {
      console.log("ProductManager: Loading initial data");
      this.state.loading = true;
      this.state.error = null;

      // Load products and categories in parallel for better performance
      const [products, categories] = await Promise.all([
        this.loadProducts(),
        this.loadCategories()
      ]);

      this.state.products = products;
      this.state.categories = categories;

      console.log(`ProductManager: Loaded ${products.length} products and
          ↪  ${categories.length} categories`);

    } catch (error) {
      console.error("ProductManager: Failed to load initial data:", error);
      this.state.error = "Failed to load data. Please refresh the page.";
    } finally {
      this.state.loading = false;
    }
  }

  async loadProducts() {
    // searchRead: Search for records and read their fields in one call
    return await this.orm.searchRead(
      "product.product",                    // Model name
      [["sale_ok", "=", true]],             // Domain (search filters)
      ["id", "name", "list_price", "categ_id", "qty_available"], // Fields to read
      {
        order: "name asc",                  // Sort order
        limit: 100                          // Maximum records
      }
    );
  }

  async loadCategories() {
    return await this.orm.searchRead(
      "product.category",
      [],                                   // Empty domain = all records
      ["id", "name"],
      { order: "name asc" }
    );
  }

  async createProduct() {
    if (!this.state.formData.name.trim()) {
      this.state.error = "Product name is required";
      return;
    }

    try {
      console.log("ProductManager: Creating new product", this.state.formData);
```

```
    // create: Create new records
    const newProductIds = await this.orm.create(
      "product.product",
      [{
        name: this.state.formData.name,
        list_price: this.state.formData.list_price,
        categ_id: this.state.formData.categ_id,
        sale_ok: true,
        purchase_ok: true
      }]
    );

    console.log(`ProductManager: Created product with ID ${newProductIds[0]}`);

    // Reload products to show the new one
    await this.refreshProducts();

    // Reset form
    this.resetForm();

    this.state.error = null;

  } catch (error) {
    console.error("ProductManager: Failed to create product:", error);
    this.state.error = `Failed to create product: ${error.message}`;
  }
}

async updateProduct(productId, updates) {
  try {
    console.log(`ProductManager: Updating product ${productId}`, updates);

    // write: Update existing records
    await this.orm.write(
      "product.product",
      [productId],                    // List of record IDs to update
      updates                         // Dictionary of field updates
    );

    console.log(`ProductManager: Successfully updated product ${productId}`);

    // Refresh the product list
    await this.refreshProducts();

  } catch (error) {
    console.error(`ProductManager: Failed to update product ${productId}:`, error);
    this.state.error = `Failed to update product: ${error.message}`;
  }
}

async deleteProduct(productId) {
  try {
    console.log(`ProductManager: Deleting product ${productId}`);
```

```
      // unlink: Delete records
      await this.orm.unlink("product.product", [productId]);

      console.log(`ProductManager: Successfully deleted product ${productId}`);

      // Remove from local state immediately for better UX
      this.state.products = this.state.products.filter(p => p.id !== productId);

      // Clear selection if deleted product was selected
      if (this.state.selectedProduct?.id === productId) {
        this.state.selectedProduct = null;
      }

    } catch (error) {
      console.error(`ProductManager: Failed to delete product ${productId}:`, error);
      this.state.error = `Failed to delete product: ${error.message}`;
    }
  }

  async refreshProducts() {
    try {
      const products = await this.loadProducts();
      this.state.products = products;
      console.log("ProductManager: Products refreshed successfully");
    } catch (error) {
      console.error("ProductManager: Failed to refresh products:", error);
    }
  }

  // Helper methods for form handling
  selectProduct(product) {
    this.state.selectedProduct = product;
    this.state.formData = {
      name: product.name,
      list_price: product.list_price,
      categ_id: product.categ_id[0],
      description: ""
    };
  }

  resetForm() {
    this.state.formData = {
      name: "",
      list_price: 0,
      categ_id: null,
      description: ""
    };
    this.state.selectedProduct = null;
  }

  onFormChange(field, value) {
    this.state.formData[field] = value;
  }

  // Quick actions for product management
```

```
  async quickUpdatePrice(productId, newPrice) {
    await this.updateProduct(productId, { list_price: newPrice });
  }

  async toggleProductAvailability(productId, currentStatus) {
    await this.updateProduct(productId, { sale_ok: !currentStatus });
  }

  // Computed properties for template
  get formattedProducts() {
    return this.state.products.map(product => ({
      ...product,
      formattedPrice: `$${product.list_price.toFixed(2)}`,
      categoryName: product.categ_id ? product.categ_id[1] : "No Category",
      stockStatus: product.qty_available > 0 ? "In Stock" : "Out of Stock"
    }));
  }

  get isFormValid() {
    return this.state.formData.name.trim().length > 0;
  }
}
```

Template (`product_manager.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.ProductManager" owl="1">
    <div class="product-manager">
      <!-- Header -->
      <div class="d-flex justify-content-between align-items-center mb-4">
        <h2>Product Manager</h2>
        <button
            class="btn btn-outline-secondary btn-sm"
            t-on-click="refreshProducts"
            t-att-disabled="state.loading">
          <i class="fa fa-refresh"></i> Refresh
        </button>
      </div>

      <!-- Error Display -->
      <t t-if="state.error">
        <div class="alert alert-danger alert-dismissible">
          <t t-esc="state.error"/>
          <button
              type="button"
              class="btn-close"
              t-on-click="() => this.state.error = null">
          </button>
        </div>
      </t>

      <!-- Loading State -->
      <t t-if="state.loading">
        <div class="text-center py-5">
          <div class="spinner-border text-primary" role="status">
```

```
          <span class="visually-hidden">Loading products...</span>
        </div>
        <p class="mt-2 text-muted">Loading products and categories...</p>
      </div>
    </t>

    <!-- Main Content -->
    <t t-else="">
      <div class="row">
        <!-- Product Form -->
        <div class="col-md-4">
          <div class="card">
            <div class="card-header">
              <h5 class="mb-0">
                <t t-if="state.selectedProduct">Edit Product</t>
                <t t-else="">New Product</t>
              </h5>
            </div>
            <div class="card-body">
              <form t-on-submit.prevent="createProduct">
                <!-- Product Name -->
                <div class="mb-3">
                  <label class="form-label">Product Name *</label>
                  <input
                      type="text"
                      class="form-control"
                      t-model="state.formData.name"
                      placeholder="Enter product name"
                      required
                  />
                </div>

                <!-- Price -->
                <div class="mb-3">
                  <label class="form-label">Price</label>
                  <div class="input-group">
                    <span class="input-group-text">$</span>
                    <input
                        type="number"
                        class="form-control"
                        t-model="state.formData.list_price"
                        min="0"
                        step="0.01"
                    />
                  </div>
                </div>

                <!-- Category -->
                <div class="mb-3">
                  <label class="form-label">Category</label>
                  <select
                      class="form-select"
                      t-model="state.formData.categ_id">
                    <option value="">Select Category</option>
                    <t t-foreach="state.categories" t-as="category"
↪ t-key="category.id">
```

```
                <option t-att-value="category.id" t-esc="category.name"/>
              </t>
            </select>
          </div>

          <!-- Action Buttons -->
          <div class="d-flex gap-2">
            <button
                type="submit"
                class="btn btn-primary flex-fill"
                t-att-disabled="!isFormValid">
              <t t-if="state.selectedProduct">Update Product</t>
              <t t-else="">Create Product</t>
            </button>

            <button
                type="button"
                class="btn btn-outline-secondary"
                t-on-click="resetForm">
              Clear
            </button>
          </div>
        </form>
      </div>
    </div>
  </div>

  <!-- Product List -->
  <div class="col-md-8">
    <div class="card">
      <div class="card-header">
        <h5 class="mb-0">
          Products (<t t-esc="state.products.length"/>)
        </h5>
      </div>
      <div class="card-body p-0">
        <t t-if="state.products.length === 0">
          <div class="text-center py-4 text-muted">
            <i class="fa fa-box-open fa-2x mb-2"></i>
            <p>No products found. Create your first product!</p>
          </div>
        </t>
        <t t-else="">
          <div class="table-responsive">
            <table class="table table-hover mb-0">
              <thead class="table-light">
                <tr>
                  <th>Name</th>
                  <th>Category</th>
                  <th>Price</th>
                  <th>Stock</th>
                  <th class="text-end">Actions</th>
                </tr>
              </thead>
              <tbody>
```

```
                    <t t-foreach="formattedProducts" t-as="product"
    ↪  t-key="product.id">
                      <tr t-att-class="state.selectedProduct?.id === product.id ?
    ↪  'table-active' : ''">
                        <td>
                          <strong t-esc="product.name"/>
                        </td>
                        <td>
                          <small class="text-muted" t-esc="product.categoryName"/>
                        </td>
                        <td>
                          <span t-esc="product.formattedPrice"/>
                        </td>
                        <td>
                          <span
                              class="badge"
                              t-att-class="product.qty_available > 0 ? 'bg-success' :
    ↪  'bg-warning'">
                            <t t-esc="product.stockStatus"/>
                          </span>
                        </td>
                        <td class="text-end">
                          <div class="btn-group btn-group-sm">
                            <button
                                class="btn btn-outline-primary"
                                t-on-click="() => this.selectProduct(product)"
                                title="Edit">
                              <i class="fa fa-edit"></i>
                            </button>
                            <button
                                class="btn btn-outline-danger"
                                t-on-click="() => this.deleteProduct(product.id)"
                                title="Delete">
                              <i class="fa fa-trash"></i>
                            </button>
                          </div>
                        </td>
                      </tr>
                    </t>
                  </tbody>
                </table>
              </div>
            </t>
          </div>
        </div>
      </div>
    </t>
  </div>
</t>
</templates>
```

This example demonstrates all the key ORM operations:

- **searchRead**: Search and read records in one operation
- **create**: Create new records

- **write**: Update existing records
- **unlink**: Delete records

Advanced ORM Operations

The orm service provides more advanced methods for complex scenarios:

Using read for Specific Records:

```
// Read specific records by ID
const products = await this.orm.read(
  "product.product",
  [1, 2, 3],                              // Specific record IDs
  ["name", "list_price", "description"] // Fields to read
);
```

Using search and searchCount:

```
// Get only record IDs matching criteria
const productIds = await this.orm.search(
  "product.product",
  [["sale_ok", "=", true]],
  { limit: 5, offset: 10 }
);

// Count records without fetching them
const totalProducts = await this.orm.searchCount(
  "product.product",
  [["sale_ok", "=", true]]
);
```

Calling Custom Model Methods:

```
// Call any method on your Python model
const result = await this.orm.call(
  "product.product",      // Model name
  "my_custom_method",     // Method name
  [recordId],            // Args (usually record IDs)
  {                      // Kwargs
    param1: "value1",
    param2: "value2"
  }
);
```

The rpc Service: Direct Controller Communication

While the orm service is perfect for model operations, sometimes you need to call controller routes directly. The rpc service (short for **Remote Procedure Call** — invoking a function that actually runs on the server, as if it were local) provides this capability:

JavaScript Example:

```
import { Component, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";
```

```
export class CustomReportGenerator extends Component {
  static template = "my_module.CustomReportGenerator";

  setup() {
    this.state = useState({
      reportData: null,
      loading: false,
      filters: {
        start_date: "",
        end_date: "",
        partner_ids: []
      }
    });

    this.rpc = useService("rpc");
  }

  async generateReport() {
    try {
      this.state.loading = true;

      console.log("CustomReportGenerator: Generating custom report");

      // Call your custom controller route
      const reportData = await this.rpc("/my_module/generate_report", {
        filters: this.state.filters,
        format: "json",
        include_details: true
      });

      this.state.reportData = reportData;
      console.log("CustomReportGenerator: Report generated successfully");

    } catch (error) {
      console.error("CustomReportGenerator: Report generation failed:", error);
      // Handle error appropriately
    } finally {
      this.state.loading = false;
    }
  }

  async downloadReportPDF() {
    try {
      console.log("CustomReportGenerator: Downloading PDF report");

      // This might return a blob or URL for download
      const pdfData = await this.rpc("/my_module/generate_report", {
        filters: this.state.filters,
        format: "pdf"
      });

      // Handle PDF download logic here

    } catch (error) {
      console.error("CustomReportGenerator: PDF download failed:", error);
```

```
    }
  }
}
```

Error Handling and User Feedback

Professional applications need robust error handling. Here's how to handle common scenarios:

Comprehensive Error Handling Pattern:

```
async performDatabaseOperation() {
  try {
    this.state.loading = true;
    this.state.error = null;

    const result = await this.orm.searchRead(
      "some.model",
      this.buildDomain(),
      this.getRequiredFields()
    );

    this.state.data = result;

  } catch (error) {
    console.error("Database operation failed:", error);

    // Different error types need different handling
    if (error.code === 403) {
      this.state.error = "You don't have permission to access this data.";
    } else if (error.code === 404) {
      this.state.error = "The requested data was not found.";
    } else if (error.message.includes("network")) {
      this.state.error = "Network error. Please check your connection.";
    } else {
      this.state.error = `Operation failed: ${error.message}`;
    }

  } finally {
    this.state.loading = false;
  }
}
```

Optimistic Updates Pattern:

```
async quickUpdate(recordId, field, value) {
  // Store original value for rollback
  const originalData = [...this.state.records];

  try {
    // Update UI immediately (optimistic)
    const record = this.state.records.find(r => r.id === recordId);
    if (record) {
      record[field] = value;
    }
```

```
    // Then update server
    await this.orm.write("model.name", [recordId], { [field]: value });

    console.log("Quick update successful");

  } catch (error) {
    // Rollback UI on error
    this.state.records = originalData;
    console.error("Quick update failed, rolled back:", error);

    // Show user-friendly error message
    this.showError(`Failed to update ${field}. Please try again.`);
  }
}
```

Service Integration Patterns

Pattern 1: Service Composition

```
setup() {
  // Multiple services working together
  this.orm = useService("orm");
  this.rpc = useService("rpc");
  this.notification = useService("notification");
  this.action = useService("action");

  // Composed operation using multiple services
  this.performComplexOperation = async () => {
    try {
      // 1. Save data via ORM
      const [recordId] = await this.orm.create("model.name", [this.state.formData]);

      // 2. Trigger server-side processing via RPC
      await this.rpc("/custom/process_record", { record_id: recordId });

      // 3. Show success notification
      this.notification.add("Record created and processed successfully!", {
        type: "success"
      });

      // 4. Navigate to the new record
      this.action.doAction({
        type: "ir.actions.act_window",
        res_model: "model.name",
        res_id: recordId,
        views: [[false, "form"]],
        target: "current"
      });

    } catch (error) {
      this.notification.add("Operation failed. Please try again.", {
        type: "danger"
      });
```

```
    }
  };
}
```

Pattern 2: Service Abstraction

```
setup() {
  this.orm = useService("orm");

  // Create abstraction layer for your specific domain
  this.customerService = {
    async getCustomers(filters = {}) {
      return await this.orm.searchRead(
        "res.partner",
        [["is_company", "=", true], ...this.buildDomain(filters)],
        ["name", "email", "phone", "city"],
        { order: "name asc" }
      );
    },

    async createCustomer(customerData) {
      return await this.orm.create("res.partner", [customerData]);
    },

    async updateCustomer(customerId, updates) {
      return await this.orm.write("res.partner", [customerId], updates);
    }
  };
}
```

Performance Considerations

Batch Operations

```
// BAD: Multiple individual calls
for (const productId of productIds) {
  await this.orm.write("product.product", [productId], { active: false });
}

// GOOD: Single batch call
await this.orm.write("product.product", productIds, { active: false });
```

Efficient Field Selection

```
// BAD: Loading unnecessary data
const products = await this.orm.searchRead("product.product", [], []);

// GOOD: Only load what you need
const products = await this.orm.searchRead(
  "product.product",
  [],
  ["id", "name", "list_price"] // Only the fields you actually use
);
```

Smart Caching Pattern

```
setup() {
  this.orm = useService("orm");
  this.cache = new Map();

  this.getCachedData = async (cacheKey, fetcher) => {
    if (this.cache.has(cacheKey)) {
      console.log(`Using cached data for ${cacheKey}`);
      return this.cache.get(cacheKey);
    }

    const data = await fetcher();
    this.cache.set(cacheKey, data);
    return data;
  };

  this.getCategories = () => {
    return this.getCachedData('categories', () =>
      this.orm.searchRead("product.category", [], ["id", "name"])
    );
  };
}
```

Common Pitfalls and Solutions

Pitfall 1: Assuming `searchRead` Returns a Wrapper Object

```
// Wrong - some ORMs wrap results in an object; Odoo's doesn't
const { records } = await this.orm.searchRead("res.partner", [], ["name"]);

// Correct - searchRead resolves directly to an array of records
const records = await this.orm.searchRead("res.partner", [], ["name"]);
```

Pitfall 2: Forgetting `await`

```
// Wrong - every orm method returns a Promise; without await you store the Promise
    ↪ itself
loadProducts() {
  this.state.products = this.orm.searchRead("product.product", [], ["name"]);
}

// Correct
async loadProducts() {
  this.state.products = await this.orm.searchRead("product.product", [], ["name"]);
}
```

Pitfall 3: Not Handling Server Errors

```
// Wrong - an unhandled rejection can crash the whole action
async saveRecord() {
  await this.orm.create("res.partner", [this.state.formData]);
}
```

```
// Correct - wrap in try/catch and give the user feedback
async saveRecord() {
  try {
    await this.orm.create("res.partner", [this.state.formData]);
  } catch (error) {
    this.state.error = "Could not save the record. Please try again.";
  }
}
```

Pitfall 4: Forgetting that `create` Returns an Array of IDs

```
// Wrong - assuming create resolves to a single id
const id = await this.orm.create("res.partner", [{ name: "Test" }]);

// Correct - create always resolves to an array, even for a single record
const [id] = await this.orm.create("res.partner", [{ name: "Test" }]);
```

Testing Service Integration

Because components receive services through `useService`, tests can swap in mocks. The exact helpers depend on your Odoo version (we'll cover Odoo's real test framework in Chapter 16 Part 1), but the idea always looks like this:

```
// Pseudo-code: the shape of a service-mocking test

describe("ProductManager", () => {
  test("should load products on mount", async () => {
    const mockOrm = {
      searchRead: jest.fn().mockResolvedValue([
        { id: 1, name: "Test Product", list_price: 10.0 }
      ])
    };

    const component = await makeTestComponent(ProductManager, {
      services: {
        orm: mockOrm
      }
    });

    await nextTick(); // Wait for effects to complete

    expect(mockOrm.searchRead).toHaveBeenCalledWith(
      "product.product",
      [["sale_ok", "=", true]],
      expect.any(Array),
      expect.any(Object)
    );

    expect(component.state.products).toHaveLength(1);
  });
});
```

Summary

The `useService` hook and Odoo's service system provide:

- **`orm` service**: Complete CRUD operations for Odoo models
- **`rpc` service**: Direct communication with custom controllers
- **Consistent API**: Same patterns across your entire application
- **Error handling**: Built-in error management and recovery
- **Performance**: Optimized communication with the backend

Key principles for effective service usage:

1. **Use the right service** for each task (ORM for models, RPC for controllers)
2. **Handle errors gracefully** with user-friendly messages
3. **Batch operations** when possible for better performance
4. **Cache strategically** to reduce server requests
5. **Test with mocks** to ensure reliability

In the next chapter, we'll explore more of Odoo's built-in services like notifications and actions, which will help you create truly integrated user experiences.

TL;DR: Use `useService("orm")` for CRUD against Odoo models (`searchRead`, `create`, `write`, `unlink`) and `useService("rpc")` for custom controllers — always `await` them, always wrap calls in `try/catch`, and remember `create` returns an array of ids, not a bare id.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch12_ex1`. Installation instructions are in the repository README.

Exercises

1. **Basic CRUD:** Build a tiny component that lists `res.partner` records (`searchRead` with just `name` and `email`), and add a button that creates a new partner named "Test Contact" using `orm.create`. Remember `create` returns an array of ids.
2. **Handle the error:** Wrap a `orm.write` call in a `try/catch` and show a Bootstrap alert with a friendly message if it fails. Test it by passing an invalid model name and see what happens without the try/catch first.
3. **Count before you fetch:** Use `orm.searchCount` to show "X contacts found" before actually loading the full list with `searchRead`, so the user sees a number immediately while the data streams in.

What's Next?

With data flowing between your components and the server, Chapter 13 turns to the rest of Odoo's built-in services — notifications, dialogs, actions, and permissions — the pieces that make a component feel like a native part of Odoo instead of a bolted-on widget.

Chapter 13 Part 1: Odoo's Built-in Services

Why this chapter? Because "it works, but it doesn't feel like Odoo" is the most common complaint about a custom addon, and it almost never comes from the logic — it comes from rolling your own toast, your own modal, and your own redirect instead of using the ones Odoo already ships.

What is Odoo trying to solve with this? Odoo has thousands of screens that all have to notify, navigate, and confirm the same way. Exposing `notification`, `action`, and `dialog` as injectable services is how the platform holds that consistency together without every addon author reimplementing it — and reimplementing it slightly differently.

Real-world application: Every client addon I've shipped ends up calling `notification.add()` after a save, `action.doAction()` to drop the user on the right record, and `dialog.add()` to confirm something destructive. Those three account for most of the UI plumbing in day-to-day work.

In Chapter 12, we mastered server communication with the `orm` and `rpc` services. But Odoo's service ecosystem extends far beyond data operations. The platform provides a rich collection of built-in services that handle everything from user notifications to navigation, dialog management, and system integration.

These services are the secret to creating components that feel like native parts of the Odoo experience. Instead of building custom solutions for common UI patterns, you can leverage Odoo's battle-tested services to provide consistent, polished user interactions.

This chapter is split into two parts. Part 1 (this one) covers the services you'll reach for on almost every project: `notification`, `action`, `dialog`, and a handful of smaller essentials. Part 2 covers advanced composition patterns, performance optimization, and error recovery for more complex scenarios — skip ahead once you're comfortable with the basics here.

The `notification` Service: User Feedback Done Right

User feedback is crucial for good UX. Users need to know when operations succeed, when errors occur, and when important events happen. The `notification` service provides Odoo's signature toast notifications that appear in the top-right corner of the screen.

Basic Notification Usage

Let's build a comprehensive example that demonstrates all notification types and advanced features. We'll build the `NotificationDemo` class one group of methods at a time — each code block below continues directly inside the same class body.

JavaScript (`notification_demo.js`):

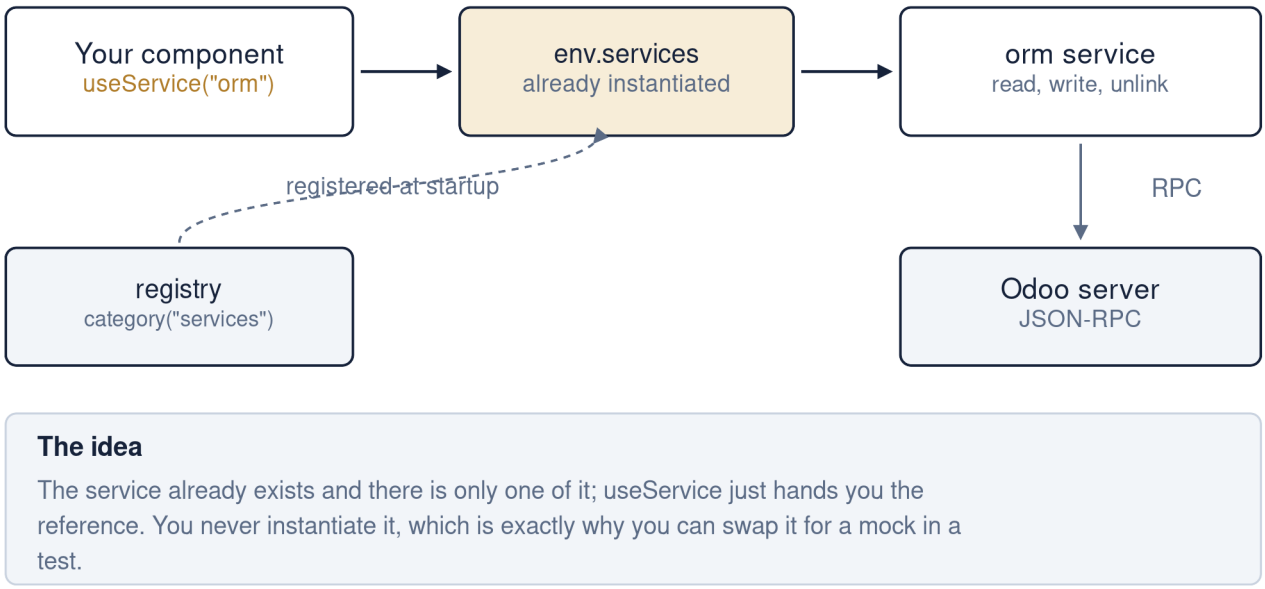


Figure 5: From useService to the server: the service already exists, you only get a reference.

First, the imports, the class declaration, and `setup()`, where we grab the two services we'll need:

```
import { Component, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class NotificationDemo extends Component {
  static template = "my_module.NotificationDemo";

  setup() {
    this.state = useState({
      messageText: "Operation completed successfully!",
      notificationCount: 0
    });

    this.notification = useService("notification");
    this.orm = useService("orm");
  }
```

Next, the four basic notification types. Notice the pattern: call `this.notification.add(message, options)` and pass a `type` that matches the situation (`success`, `info`, `warning`, or `danger`):

```
  // Basic notification types
  showSuccessNotification() {
    this.notification.add(this.state.messageText, {
      title: "Success",
      type: "success",
      sticky: false // Auto-dismiss after a few seconds
    });

    this.incrementNotificationCount();
    console.log("NotificationDemo: Success notification shown");
  }

  showInfoNotification() {
```

```
    this.notification.add("Here's some useful information for you.", {
      title: "Information",
      type: "info"
    });

    this.incrementNotificationCount();
  }

  showWarningNotification() {
    this.notification.add("Please review this carefully before proceeding.", {
      title: "Warning",
      type: "warning",
      sticky: true // Stays until manually dismissed
    });

    this.incrementNotificationCount();
  }

  showErrorNotification() {
    this.notification.add("Something went wrong. Please try again.", {
      title: "Error",
      type: "danger",
      sticky: true
    });

    this.incrementNotificationCount();
  }
```

Now two methods that show notifications doing real work: one adds a custom CSS class for styling, the other wraps an async operation that can fail, showing feedback before, on success, and on error:

```
  // Advanced notification with custom content
  showRichNotification() {
    this.notification.add("Your document has been processed.", {
      title: "Document Processing Complete",
      type: "success",
      sticky: false,
      className: "custom-notification-style" // Custom CSS class
    });

    this.incrementNotificationCount();
  }

  // Real-world example: Save operation with feedback
  async saveDataWithFeedback() {
    try {
      // Show immediate feedback
      this.notification.add("Saving your changes...", {
        title: "Saving",
        type: "info"
      });

      // Simulate save operation
      await this.simulateAsyncOperation();

      // Show success feedback
```

```
      this.notification.add("Your changes have been saved successfully.", {
        title: "Saved",
        type: "success"
      });

      console.log("NotificationDemo: Data saved successfully");

    } catch (error) {
      // Show error feedback
      this.notification.add(`Failed to save: ${error.message}`, {
        title: "Save Failed",
        type: "danger",
        sticky: true // Keep error visible until user dismisses
      });

      console.error("NotificationDemo: Save failed:", error);
    }

    this.incrementNotificationCount();
  }
```

The last two methods handle a multi-step bulk operation with progress updates, and a form validation flow that gives feedback either way:

```
  // Bulk operation with progress feedback
  async performBulkOperation() {
    const items = ['Item A', 'Item B', 'Item C', 'Item D'];
    let processed = 0;

    try {
      // Initial notification
      this.notification.add(`Starting bulk operation on ${items.length} items...`, {
        title: "Bulk Operation",
        type: "info"
      });

      for (const item of items) {
        await this.simulateAsyncOperation(500); // Simulate processing time
        processed++;

        // Progress notification
        this.notification.add(`Processed ${processed}/${items.length} items`, {
          title: "Progress Update",
          type: "info"
        });
      }

      // Completion notification
      this.notification.add(`Successfully processed all ${items.length} items!`, {
        title: "Bulk Operation Complete",
        type: "success",
        sticky: false
      });

    } catch (error) {
```

```
      this.notification.add(`Bulk operation failed after processing ${processed}
          ↪ items.`, {
        title: "Operation Failed",
        type: "danger",
        sticky: true
      });
    }
  }

  // Validation feedback example
  validateAndNotify() {
    if (!this.state.messageText.trim()) {
      this.notification.add("Message text cannot be empty.", {
        title: "Validation Error",
        type: "warning"
      });
      return false;
    }

    if (this.state.messageText.length > 100) {
      this.notification.add("Message text is too long (maximum 100 characters).", {
        title: "Validation Error",
        type: "warning"
      });
      return false;
    }

    this.notification.add("Message validation passed!", {
      title: "Valid Input",
      type: "success"
    });

    return true;
  }
```

Finally, the small helper methods that keep the demo tidy, and the closing brace for the class:

```
  // Helper methods
  incrementNotificationCount() {
    this.state.notificationCount++;
  }

  async simulateAsyncOperation(delay = 1000) {
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        // Simulate occasional failures
        if (Math.random() < 0.1) {
          reject(new Error("Simulated network error"));
        } else {
          resolve();
        }
      }, delay);
    });
  }

  onMessageChange(event) {
```

```
    this.state.messageText = event.target.value;
  }

  resetDemo() {
    this.state.messageText = "Operation completed successfully!";
    this.state.notificationCount = 0;
  }
}
```

Template (`notification_demo.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.NotificationDemo" owl="1">
    <div class="notification-demo card">
      <div class="card-header">
        <h4>Notification Service Demo</h4>
        <small class="text-muted">
          Notifications shown: <span class="badge bg-secondary"
    ↪  t-esc="state.notificationCount"/>
        </small>
      </div>

      <div class="card-body">
        <!-- Custom Message Input -->
        <div class="mb-4">
          <label class="form-label">Custom Message</label>
          <div class="input-group">
            <input
                type="text"
                class="form-control"
                t-model="state.messageText"
                placeholder="Enter notification message"
                maxlength="100"
            />
            <button
                class="btn btn-outline-secondary"
                t-on-click="validateAndNotify">
              Validate
            </button>
          </div>
          <div class="form-text">
            Message length: <t t-esc="state.messageText.length"/>/100 characters
          </div>
        </div>

        <!-- Basic Notification Types -->
        <div class="mb-4">
          <h6>Basic Notification Types</h6>
          <div class="btn-group me-2 mb-2" role="group">
            <button
                class="btn btn-success btn-sm"
                t-on-click="showSuccessNotification">
              <i class="fa fa-check"></i> Success
            </button>
            <button
```

```
                class="btn btn-info btn-sm"
                t-on-click="showInfoNotification">
              <i class="fa fa-info-circle"></i> Info
            </button>
            <button
                class="btn btn-warning btn-sm"
                t-on-click="showWarningNotification">
              <i class="fa fa-exclamation-triangle"></i> Warning
            </button>
            <button
                class="btn btn-danger btn-sm"
                t-on-click="showErrorNotification">
              <i class="fa fa-times"></i> Error
            </button>
          </div>
        </div>

        <!-- Advanced Examples -->
        <div class="mb-4">
          <h6>Real-World Examples</h6>
          <div class="d-flex gap-2 flex-wrap">
            <button
                class="btn btn-primary btn-sm"
                t-on-click="showRichNotification">
              <i class="fa fa-star"></i> Rich Notification
            </button>

            <button
                class="btn btn-outline-primary btn-sm"
                t-on-click="saveDataWithFeedback">
              <i class="fa fa-save"></i> Save with Feedback
            </button>

            <button
                class="btn btn-outline-secondary btn-sm"
                t-on-click="performBulkOperation">
              <i class="fa fa-tasks"></i> Bulk Operation
            </button>
          </div>
        </div>

        <!-- Demo Controls -->
        <div class="border-top pt-3">
          <button
              class="btn btn-outline-secondary btn-sm"
              t-on-click="resetDemo">
            <i class="fa fa-refresh"></i> Reset Demo
          </button>
        </div>
      </div>
    </div>
  </t>
</templates>
```

Notification Best Practices

Do's: - Use appropriate notification types (`success` for completed actions, `warning` for important info, `danger` for errors) - Make error notifications sticky so users can read them fully - Provide clear, actionable messages - Use notifications for feedback on user actions

Don'ts: - Don't spam users with too many notifications - Don't use notifications for critical system errors (use dialogs instead) - Avoid generic messages like "Error occurred" - Don't use notifications for information that needs user response

---

The `action` Service: Navigation and Integration

The `action` service is your gateway to Odoo's action system. It can trigger any action in the system—open forms, show lists, run reports, execute server actions, and much more.

Understanding Action Types

Odoo supports several action types: - **Window Actions**: Open views (form, list, kanban, etc.) - **Server Actions**: Execute Python code on the server - **Report Actions**: Generate and display reports - **URL Actions**: Navigate to external URLs - **Client Actions**: Open custom client-side components

Let's build a comprehensive example, again method group by method group.

JavaScript (`action_navigator.js`):

First, the setup and the data loader that fills the two select boxes we'll use to pick a record:

```
import { Component, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ActionNavigator extends Component {
  static template = "my_module.ActionNavigator";

  setup() {
    this.state = useState({
      selectedPartnerId: null,
      selectedProductId: null,
      partners: [],
      products: []
    });

    this.action = useService("action");
    this.orm = useService("orm");
    this.notification = useService("notification");

    // Load sample data
    this.loadSampleData();
  }

  async loadSampleData() {
    try {
      const [partners, products] = await Promise.all([
        this.orm.searchRead("res.partner", [], ["name"], { limit: 10 }),
        this.orm.searchRead("product.product", [], ["name"], { limit: 10 })
```

```
      ]);

      this.state.partners = partners;
      this.state.products = products;
    } catch (error) {
      console.error("ActionNavigator: Failed to load sample data:", error);
    }
  }
```

With sample data loaded, let's add the basic window-action methods — opening a form, a list, a kanban, plus the variations for opening in a new window, in a modal, or waiting on the result with a callback:

```
  // Basic window actions
  openCustomerForm(partnerId = null) {
    console.log(`ActionNavigator: Opening customer form for ID: ${partnerId || 'new'}`);

    this.action.doAction({
      type: "ir.actions.act_window",
      name: partnerId ? "Edit Customer" : "New Customer",
      res_model: "res.partner",
      res_id: partnerId || false,
      views: [[false, "form"]],
      target: "current", // Opens in current view
      context: {
        default_is_company: true,
        default_customer_rank: 1
      }
    });
  }

  openCustomerList() {
    console.log("ActionNavigator: Opening customer list view");

    this.action.doAction({
      type: "ir.actions.act_window",
      name: "Customers",
      res_model: "res.partner",
      views: [
        [false, "list"],
        [false, "form"]
      ],
      domain: [["is_company", "=", true]],
      target: "current"
    });
  }

  openCustomerKanban() {
    this.action.doAction({
      type: "ir.actions.act_window",
      name: "Customer Pipeline",
      res_model: "res.partner",
      views: [
        [false, "kanban"],
        [false, "form"]
      ],
```

```
      domain: [["is_company", "=", true]],
      target: "current"
    });
  }

  // Open in new tab/window
  openInNewWindow(partnerId) {
    console.log(`ActionNavigator: Opening customer ${partnerId} in new window`);

    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "res.partner",
      res_id: partnerId,
      views: [[false, "form"]],
      target: "new" // Opens in new window/tab
    });
  }

  // Modal dialogs
  openInModal(partnerId) {
    console.log(`ActionNavigator: Opening customer ${partnerId} in modal`);

    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "res.partner",
      res_id: partnerId,
      views: [[false, "form"]],
      target: "new",
      flags: {
        mode: "readonly" // Optional: open in read-only mode
      }
    });
  }

  // Action with callback
  async openWithCallback(partnerId) {
    console.log(`ActionNavigator: Opening customer ${partnerId} with callback`);

    try {
      const result = await this.action.doAction({
        type: "ir.actions.act_window",
        res_model: "res.partner",
        res_id: partnerId,
        views: [[false, "form"]],
        target: "new"
      });

      console.log("ActionNavigator: Action completed with result:", result);

      this.notification.add("Customer form was opened successfully!", {
        type: "success"
      });

    } catch (error) {
      console.error("ActionNavigator: Action failed:", error);
```

```
      this.notification.add("Failed to open customer form.", {
        type: "danger"
      });
    }
  }
```

Beyond opening forms, the `action` service can also open related records, run server actions and reports, navigate to external URLs, and open your own custom client actions:

```
  // Open related records
  openCustomerSales(partnerId) {
    console.log(`ActionNavigator: Opening sales for customer ${partnerId}`);

    this.action.doAction({
      type: "ir.actions.act_window",
      name: "Customer Sales",
      res_model: "sale.order",
      views: [
        [false, "list"],
        [false, "form"]
      ],
      domain: [["partner_id", "=", partnerId]],
      context: {
        default_partner_id: partnerId,
        search_default_partner_id: partnerId
      },
      target: "current"
    });
  }

  // Server actions
  async executeServerAction(actionXmlId) {
    console.log(`ActionNavigator: Executing server action: ${actionXmlId}`);

    try {
      const result = await this.action.doAction(actionXmlId, {
        // Additional context for the server action
        active_ids: [this.state.selectedPartnerId],
        active_id: this.state.selectedPartnerId,
        active_model: "res.partner"
      });

      this.notification.add("Server action executed successfully!", {
        type: "success"
      });

      console.log("ActionNavigator: Server action result:", result);

    } catch (error) {
      console.error("ActionNavigator: Server action failed:", error);

      this.notification.add(`Server action failed: ${error.message}`, {
        type: "danger"
      });
    }
```

```
}

// Report actions
openCustomerReport(partnerId) {
  console.log(`ActionNavigator: Opening report for customer ${partnerId}`);

  this.action.doAction({
    type: "ir.actions.report",
    report_name: "base.report_partner_ledger", // Example report
    context: {
      active_ids: [partnerId],
      active_model: "res.partner"
    }
  });
}

// URL actions
openExternalUrl(url) {
  console.log(`ActionNavigator: Opening external URL: ${url}`);

  this.action.doAction({
    type: "ir.actions.act_url",
    url: url,
    target: "new"
  });
}

// Custom client actions
openCustomClientAction() {
  console.log("ActionNavigator: Opening custom client action");

  this.action.doAction({
    type: "ir.actions.client",
    tag: "my_custom_action", // Your custom action tag
    name: "Custom Dashboard",
    params: {
      // Custom parameters for your action
      customer_id: this.state.selectedPartnerId
    }
  });
}

// Navigation methods
goToAppsMenu() {
  this.action.doAction("base.open_module_tree");
}

goToSettings() {
  this.action.doAction("base.action_res_config_settings");
}
```

Finally, a few small helpers the template needs to read and react to the current selection:

```
// Helper methods for template
onPartnerSelect(event) {
  this.state.selectedPartnerId = parseInt(event.target.value) || null;
```

```
  }

  onProductSelect(event) {
    this.state.selectedProductId = parseInt(event.target.value) || null;
  }

  get selectedPartner() {
    return this.state.partners.find(p => p.id === this.state.selectedPartnerId);
  }

  get hasSelectedPartner() {
    return this.state.selectedPartnerId !== null;
  }
}
```

Template (`action_navigator.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="my_module.ActionNavigator" owl="1">
    <div class="action-navigator">
      <div class="row">
        <!-- Selection Panel -->
        <div class="col-md-4">
          <div class="card">
            <div class="card-header">
              <h5>Select Records</h5>
            </div>
            <div class="card-body">
              <!-- Partner Selection -->
              <div class="mb-3">
                <label class="form-label">Customer</label>
                <select
                    class="form-select"
                    t-on-change="onPartnerSelect">
                  <option value="">Select a customer...</option>
                  <t t-foreach="state.partners" t-as="partner" t-key="partner.id">
                    <option
                        t-att-value="partner.id"
                        t-att-selected="state.selectedPartnerId === partner.id"
                        t-esc="partner.name"/>
                  </t>
                </select>
              </div>

              <!-- Product Selection -->
              <div class="mb-3">
                <label class="form-label">Product</label>
                <select
                    class="form-select"
                    t-on-change="onProductSelect">
                  <option value="">Select a product...</option>
                  <t t-foreach="state.products" t-as="product" t-key="product.id">
                    <option
                        t-att-value="product.id"
                        t-att-selected="state.selectedProductId === product.id"
```

```
                    t-esc="product.name"/>
              </t>
            </select>
          </div>

          <!-- Selection Info -->
          <t t-if="hasSelectedPartner">
            <div class="alert alert-info">
              <small>
                Selected: <strong t-esc="selectedPartner.name"/>
              </small>
            </div>
          </t>
        </div>
      </div>
    </div>

    <!-- Action Panel -->
    <div class="col-md-8">
      <div class="card">
        <div class="card-header">
          <h5>Action Examples</h5>
        </div>
        <div class="card-body">
          <!-- Window Actions -->
          <div class="mb-4">
            <h6>Window Actions</h6>
            <div class="btn-group mb-2 me-2" role="group">
              <button
                  class="btn btn-primary btn-sm"
                  t-on-click="() => this.openCustomerForm()">
                <i class="fa fa-plus"></i> New Customer
              </button>
              <button
                  class="btn btn-outline-primary btn-sm"
                  t-on-click="() => this.openCustomerForm(state.selectedPartnerId)"
                  t-att-disabled="!hasSelectedPartner">
                <i class="fa fa-edit"></i> Edit Selected
              </button>
            </div>

            <div class="btn-group mb-2" role="group">
              <button
                  class="btn btn-info btn-sm"
                  t-on-click="openCustomerList">
                <i class="fa fa-list"></i> Customer List
              </button>
              <button
                  class="btn btn-info btn-sm"
                  t-on-click="openCustomerKanban">
                <i class="fa fa-columns"></i> Customer Kanban
              </button>
            </div>
          </div>
```

```
<!-- Target Options -->
<div class="mb-4">
  <h6>Open Targets</h6>
  <div class="btn-group mb-2" role="group">
    <button
        class="btn btn-outline-secondary btn-sm"
        t-on-click="() => this.openInNewWindow(state.selectedPartnerId)"
        t-att-disabled="!hasSelectedPartner">
      <i class="fa fa-external-link"></i> New Window
    </button>
    <button
        class="btn btn-outline-secondary btn-sm"
        t-on-click="() => this.openInModal(state.selectedPartnerId)"
        t-att-disabled="!hasSelectedPartner">
      <i class="fa fa-window-restore"></i> Modal
    </button>
    <button
        class="btn btn-outline-secondary btn-sm"
        t-on-click="() => this.openWithCallback(state.selectedPartnerId)"
        t-att-disabled="!hasSelectedPartner">
      <i class="fa fa-reply"></i> With Callback
    </button>
  </div>
</div>

<!-- Related Records -->
<div class="mb-4">
  <h6>Related Records</h6>
  <button
      class="btn btn-success btn-sm"
      t-on-click="() => this.openCustomerSales(state.selectedPartnerId)"
      t-att-disabled="!hasSelectedPartner">
    <i class="fa fa-shopping-cart"></i> Customer Sales
  </button>
</div>

<!-- Reports and External -->
<div class="mb-4">
  <h6>Reports &amp;amp; External</h6>
  <div class="btn-group mb-2 me-2" role="group">
    <button
        class="btn btn-warning btn-sm"
        t-on-click="() => this.openCustomerReport(state.selectedPartnerId)"
        t-att-disabled="!hasSelectedPartner">
      <i class="fa fa-file-pdf-o"></i> Customer Report
    </button>
  </div>

  <div class="btn-group mb-2" role="group">
    <button
        class="btn btn-outline-info btn-sm"
        t-on-click="() => this.openExternalUrl('https://www.odoo.com')">
      <i class="fa fa-globe"></i> Open Odoo.com
    </button>
  </div>
```

```
            </div>

            <!-- System Navigation -->
            <div class="mb-4">
              <h6>System Navigation</h6>
              <div class="btn-group" role="group">
                <button
                    class="btn btn-secondary btn-sm"
                    t-on-click="goToAppsMenu">
                  <i class="fa fa-th"></i> Apps Menu
                </button>
                <button
                    class="btn btn-secondary btn-sm"
                    t-on-click="goToSettings">
                  <i class="fa fa-cog"></i> Settings
                </button>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </t>
</templates>
```

The `dialog` Service: User Interactions

The `dialog` service provides sophisticated modal dialogs for user interactions:

JavaScript Example:

```
import { Component } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";
import { ConfirmationDialog } from "@web/core/confirmation_dialog/confirmation_dialog";

export class DialogDemo extends Component {
  static template = "my_module.DialogDemo";

  setup() {
    this.dialog = useService("dialog");
    this.notification = useService("notification");
  }

  // Confirmation dialog
  showConfirmationDialog() {
    this.dialog.add(ConfirmationDialog, {
      title: "Confirm Delete",
      body: "Are you sure you want to delete this record? This action cannot be undone.",
      confirm: () => {
        this.notification.add("Record deleted successfully!", { type: "success" });
        console.log("User confirmed deletion");
      },
      cancel: () => {
```

```
        this.notification.add("Deletion cancelled.", { type: "info" });
        console.log("User cancelled deletion");
      }
    });
  }

  // Custom dialog with form
  showCustomDialog() {
    this.dialog.add(CustomFormDialog, {
      title: "Create New Item",
      onSave: (data) => {
        console.log("Dialog saved with data:", data);
        this.notification.add("Item created successfully!", { type: "success" });
      }
    });
  }
}
```

Additional Essential Services

The user Service

Access current user information:

```
setup() {
  this.user = useService("user");

  console.log("Current user:", this.user.name);
  console.log("Is admin:", this.user.isAdmin);
  console.log("User context:", this.user.context);

  // Group membership checks are asynchronous
  onWillStart(async () => {
    this.isSaleManager = await this.user.hasGroup("sales_team.group_sale_manager");
  });
}
```

The router Service

Handle URL routing:

```
setup() {
  this.router = useService("router");

  // Navigate to a specific route
  this.navigateToCustomers = () => {
    this.router.pushState({ action: "customers" });
  };
}
```

The company Service

Access current company information:

```
setup() {
  this.company = useService("company");

  console.log("Current company:", this.company.currentCompany.name);
  console.log("Allowed companies:", this.company.allowedCompanies);
}
```

Service Integration Patterns

Pattern 1: Coordinated Service Usage

```
async createCustomerWithFeedback(customerData) {
  try {
    // 1. Show progress notification
    this.notification.add("Creating customer...", { type: "info" });

    // 2. Create the customer (create returns an array of new ids)
    const [customerId] = await this.orm.create("res.partner", [customerData]);

    // 3. Show success notification
    this.notification.add("Customer created successfully!", { type: "success" });

    // 4. Navigate to the new customer
    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "res.partner",
      res_id: customerId,
      views: [[false, "form"]],
      target: "current"
    });

  } catch (error) {
    this.notification.add(`Failed to create customer: ${error.message}`, {
      type: "danger",
      sticky: true
    });
  }
}
```

Pattern 2: Conditional Actions Based on User Permissions

```
async deleteRecord(recordId) {
  // Check if user has delete permissions (hasGroup is async)
  const canDelete = this.user.isAdmin || (await this.user.hasGroup("base.group_system"));
  if (!canDelete) {
    this.notification.add("You don't have permission to delete records.", {
      type: "warning"
    });
    return;
  }

  // Show confirmation dialog
  this.dialog.add(ConfirmationDialog, {
```

```
    title: "Confirm Deletion",
    body: "This will permanently delete the record.",
    confirm: async () => {
      try {
        await this.orm.unlink("model.name", [recordId]);
        this.notification.add("Record deleted successfully!", { type: "success" });
      } catch (error) {
        this.notification.add("Failed to delete record.", { type: "danger" });
      }
    }
  });
}
```

Testing Service Integration

Services can be mocked for testing:

```
describe("ServiceIntegration", () => {
  test("should show notification on success", async () => {
    const mockNotification = {
      add: jest.fn()
    };

    const mockAction = {
      doAction: jest.fn().mockResolvedValue(true)
    };

    const component = await makeTestComponent(MyComponent, {
      services: {
        notification: mockNotification,
        action: mockAction,
        orm: mockOrm
      }
    });

    await component.performAction();

    expect(mockNotification.add).toHaveBeenCalledWith(
      "Operation completed successfully!",
      { type: "success" }
    );

    expect(mockAction.doAction).toHaveBeenCalledWith({
      type: "ir.actions.act_window",
      res_model: "test.model",
      views: [[false, "form"]],
      target: "current"
    });
  });

  test("should handle service errors gracefully", async () => {
    const mockNotification = {
      add: jest.fn()
    };
```

```
    const mockOrm = {
      create: jest.fn().mockRejectedValue(new Error("Database error"))
    };

    const component = await makeTestComponent(MyComponent, {
      services: {
        notification: mockNotification,
        orm: mockOrm
      }
    });

    await component.createRecord();

    expect(mockNotification.add).toHaveBeenCalledWith(
      "Failed to create record: Database error",
      { type: "danger", sticky: true }
    );
  });
});
```

Service Best Practices

Do's

1. **Use appropriate services** for each task
2. **Handle errors gracefully** with user-friendly messages
3. **Provide feedback** for all user actions
4. **Cache expensive operations** when possible
5. **Test service interactions** with proper mocks
6. **Use confirmation dialogs** for destructive actions
7. **Coordinate services** for complex workflows

Don'ts

1. **Don't ignore service errors** - always handle exceptions
2. **Don't spam notifications** - be selective about what requires user attention
3. **Don't block UI** - use loading states for long operations
4. **Don't hardcode action IDs** - use XML IDs when possible
5. **Don't forget to cleanup** - clear timers and subscriptions in service callbacks

---

Common Pitfalls

Pitfall 1: Treating `hasGroup()` as Synchronous

```
// Wrong - hasGroup() returns a Promise; the `if` always takes this branch,
// because a Promise object is truthy regardless of what it resolves to
setup() {
  this.user = useService("user");
  if (this.user.hasGroup("base.group_system")) {
```

```
    // Runs unconditionally, before the group check has even resolved!
  }
}

// Correct - await it, typically inside onWillStart or another async method
onWillStart(async () => {
  this.isSystemAdmin = await this.user.hasGroup("base.group_system");
});
```

Pitfall 2: Requesting a Service Outside `setup()`

```
// Wrong - useService only works while a component is being set up;
// calling it later (e.g. from an event handler) throws an error
someButtonHandler() {
  const notification = useService("notification"); // Error!
  notification.add("Done!");
}

// Correct - request the service once in setup(), then reuse the reference
setup() {
  this.notification = useService("notification");
}

someButtonHandler() {
  this.notification.add("Done!");
}
```

Pitfall 3: Swallowing ORM Errors Silently

```
// Wrong - if orm.create() rejects, the promise is unhandled and the user
// sees nothing happen at all
async saveRecord(data) {
  await this.orm.create("res.partner", [data]);
}

// Correct - catch the error and tell the user what happened
async saveRecord(data) {
  try {
    await this.orm.create("res.partner", [data]);
    this.notification.add("Saved!", { type: "success" });
  } catch (error) {
    this.notification.add(`Save failed: ${error.message}`, {
      type: "danger",
      sticky: true
    });
  }
}
```

Pitfall 4: Reaching for a Notification When You Need a Dialog

```
// Wrong - a toast is easy to miss, disappears on its own, and can't
// block the next line of code from running
this.notification.add("This will permanently delete the record. Continue?", {
  type: "warning"
});
```

```
this.orm.unlink("res.partner", [id]); // Runs immediately, no matter what!

// Correct - a destructive action needs a confirmation dialog, which pauses
// the flow until the user actually decides
this.dialog.add(ConfirmationDialog, {
  title: "Confirm Deletion",
  body: "This will permanently delete the record. Continue?",
  confirm: () => this.orm.unlink("res.partner", [id]),
});
```

Exercises

1. **Notification variety**: Add a button that shows a `warning` notification with `sticky: true`, and another that shows an `info` notification that auto-dismisses after a few seconds. Confirm you can see the behavioral difference.
2. **Guarded delete**: Write a `deleteRecord(id)` method that first checks `await this.user.hasGroup("base.group_system")`. If the user doesn't belong to that group, show a `warning` notification instead of deleting.
3. **Confirm, then act**: Use the `dialog` service to show a `ConfirmationDialog` before calling `this.orm.unlink(...)`. On cancel, show an `info` notification saying the deletion was cancelled instead.

TL;DR: Reach for `notification` to give feedback, `dialog` for confirmations and custom dialogs, `action` to navigate, and remember `user.hasGroup()` is async — always inject services through `useService`, never build your own.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch13_1_ex1`. Installation instructions are in the repository README.

What's Next?

In Part 2, we'll build on these fundamentals with service composition patterns for multi-step workflows, performance techniques like debouncing and caching, and strategies for recovering gracefully from failures — the patterns you'll reach for once your components start doing more than a single API call.

Chapter 13 Part 2: Advanced Service Patterns and Optimization

Why this chapter? A single `notification.add()` call is easy; coordinating three services in one workflow, keeping a search box from hammering the server on every keystroke, and recovering from a flaky network call are the things that actually eat a workday once an addon grows past its first version.

What is Odoo trying to solve with this? Odoo's UI has to stay responsive and resilient under real usage — spotty connections, slow endpoints, users who click twice — so the framework leans on patterns like debouncing, caching, and structured retries instead of leaving every developer to reinvent them.

Real-world application: Debounced search, retry-with-backoff on a flaky external API call, and multi-step workflows spanning several services are patterns I reach for constantly once a client addon moves past its first proof-of-concept.

In Part 1, we covered the services you'll use every day — `notification`, `action`, `dialog`, and a few essentials — along with the common mistakes junior developers run into with them. Now we'll tackle more demanding scenarios: composing several services into a single workflow, keeping expensive service calls fast, and recovering gracefully when a call fails.

Advanced Service Patterns

Pattern 3: Service Composition for Complex Workflows

```
async processCustomerOrder(orderData) {
  const steps = [
    { name: "Validating order data", action: () => this.validateOrderData(orderData) },
    { name: "Creating customer", action: () => this.createCustomer(orderData.customer) },
    { name: "Creating order", action: () => this.createOrder(orderData) },
    { name: "Sending confirmation", action: () => this.sendConfirmationEmail(orderData) }
  ];

  let completedSteps = 0;

  try {
    // Initial progress notification
    this.notification.add(`Starting order processing (${steps.length} steps)...`, {
      type: "info"
    });

    for (const step of steps) {
      console.log(`Processing step: ${step.name}`);
```

```
      // Execute step
      await step.action();
      completedSteps++;

      // Progress notification
      this.notification.add(
        `${step.name} completed (${completedSteps}/${steps.length})`,
        { type: "info" }
      );
    }

    // Success notification and navigation
    this.notification.add("Order processed successfully!", {
      type: "success",
      sticky: false
    });

    // Navigate to the order
    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "sale.order",
      res_id: orderData.orderId,
      views: [[false, "form"]],
      target: "current"
    });

  } catch (error) {
    console.error(`Order processing failed at step ${completedSteps + 1}:`, error);

    this.notification.add(
      `Order processing failed at step: ${steps[completedSteps]?.name || 'Unknown'}.
          ↪  ${error.message}`,
      { type: "danger", sticky: true }
    );

    // Optionally show rollback dialog
    this.dialog.add(ConfirmationDialog, {
      title: "Processing Failed",
      body: "Would you like to rollback the changes made so far?",
      confirm: () => this.rollbackChanges(completedSteps),
      cancel: () => console.log("User chose not to rollback")
    });
  }
}
```

Pattern 4: Service-Based State Management

This pattern combines several services inside one component: `useState` for local reactive data, and `orm/action/notification/dialog/user` coordinated together. We'll build `CustomerDashboard` in three pieces.

First, the constructor-like `setup()`: state, service references, and a small `dataLoader` object that groups the read-only queries the dashboard needs:

```
export class CustomerDashboard extends Component {
```

```
  static template = "my_module.CustomerDashboard";

  setup() {
    this.state = useState({
      customers: [],
      selectedCustomer: null,
      loading: false,
      filters: {
        active: true,
        country: null,
        category: null
      }
    });

    // Initialize services
    this.orm = useService("orm");
    this.action = useService("action");
    this.notification = useService("notification");
    this.dialog = useService("dialog");
    this.user = useService("user");

    // Service-based data loader
    this.dataLoader = {
      customers: () => this.orm.searchRead(
        "res.partner",
        this.buildCustomerDomain(),
        ["name", "email", "phone", "country_id", "category_id"],
        { order: "name asc" }
      ),

      countries: () => this.orm.searchRead(
        "res.country",
        [],
        ["name", "code"],
        { order: "name asc" }
      ),

      categories: () => this.orm.searchRead(
        "res.partner.category",
        [],
        ["name", "color"],
        { order: "name asc" }
      )
    };

    // Initialize dashboard
    useEffect(
      () => {
        this.initializeDashboard();
      },
      () => []
    );

    // Reactive filter updates
    useEffect(
```

```
      () => {
        this.refreshCustomers();
      },
      () => [this.state.filters.active, this.state.filters.country,
    ↪  this.state.filters.category]
    );
  }
```

Next, the method that runs once on mount, loading all three datasets in parallel and reporting success or failure:

```
  async initializeDashboard() {
    try {
      this.state.loading = true;

      // Load all required data in parallel
      const [customers, countries, categories] = await Promise.all([
        this.dataLoader.customers(),
        this.dataLoader.countries(),
        this.dataLoader.categories()
      ]);

      this.state.customers = customers;
      this.state.countries = countries;
      this.state.categories = categories;

      this.notification.add("Dashboard loaded successfully!", {
        type: "success"
      });

    } catch (error) {
      console.error("Dashboard initialization failed:", error);

      this.notification.add("Failed to load dashboard data.", {
        type: "danger",
        sticky: true
      });
    } finally {
      this.state.loading = false;
    }
  }
```

Finally, the bulk-action methods: a lookup table of possible actions (archive, delete, export), a confirmation step for the dangerous ones, and the method that actually runs the chosen action:

```
  // Service-orchestrated actions
  async performBulkAction(action, customerIds) {
    const actionMap = {
      archive: {
        title: "Archive Customers",
        method: () => this.orm.write("res.partner", customerIds, { active: false }),
        successMessage: `${customerIds.length} customers archived successfully`
      },

      delete: {
        title: "Delete Customers",
        method: () => this.orm.unlink("res.partner", customerIds),
```

```
        successMessage: `${customerIds.length} customers deleted successfully`,
        dangerous: true
      },

      export: {
        title: "Export Customers",
        method: () => this.action.doAction({
          type: "ir.actions.report",
          report_name: "base.report_partner_list",
          context: { active_ids: customerIds }
        }),
        successMessage: "Customer export initiated"
      }
    };

    const actionConfig = actionMap[action];
    if (!actionConfig) return;

    // Show confirmation for dangerous actions
    if (actionConfig.dangerous) {
      this.dialog.add(ConfirmationDialog, {
        title: `Confirm ${actionConfig.title}`,
        body: `This will permanently ${action} ${customerIds.length} customers. This
            ↪  action cannot be undone.`,
        confirm: () => this.executeBulkAction(actionConfig)
      });
    } else {
      await this.executeBulkAction(actionConfig);
    }
  }

  async executeBulkAction(actionConfig) {
    try {
      this.notification.add(`${actionConfig.title} in progress...`, {
        type: "info"
      });

      await actionConfig.method();

      this.notification.add(actionConfig.successMessage, {
        type: "success"
      });

      // Refresh data after bulk action
      await this.refreshCustomers();

    } catch (error) {
      console.error(`${actionConfig.title} failed:`, error);

      this.notification.add(`${actionConfig.title} failed: ${error.message}`, {
        type: "danger",
        sticky: true
      });
    }
  }
```

```
}
```

Service Performance Optimization

Debounced Service Calls

Debouncing means delaying a function call until a burst of triggering events has stopped for a set amount of time — for example, waiting until the user pauses typing before firing a search request, instead of sending one request per keystroke. It's a common technique for keeping expensive service calls (like an ORM search) from overwhelming the server.

```
setup() {
  this.orm = useService("orm");

  // Debounce expensive searches
  this.debouncedSearch = this.debounce(async (searchTerm) => {
    if (!searchTerm.trim()) return;

    try {
      const results = await this.orm.searchRead(
        "res.partner",
        [["name", "ilike", searchTerm]],
        ["name", "email"],
        { limit: 10 }
      );

      this.state.searchResults = results;
    } catch (error) {
      console.error("Search failed:", error);
    }
  }, 300); // Wait 300ms after user stops typing
}

debounce(func, wait) {
  let timeout;
  return (...args) => {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), wait);
  };
}

onSearchInput(event) {
  const searchTerm = event.target.value;
  this.state.searchTerm = searchTerm;
  this.debouncedSearch(searchTerm);
}
```

Odoo already ships a ready-made debounce helper, so in real addons you rarely need to write your own: `import { debounce } from "@web/core/utils/timing";`.

Service Response Caching

```
setup() {
  this.orm = useService("orm");
```

```
  this.cache = new Map();
  this.cacheExpiry = new Map();

  this.getCachedData = async (cacheKey, fetcher, ttl = 5 * 60 * 1000) => {
    // Check if we have valid cached data
    if (this.cache.has(cacheKey)) {
      const expiry = this.cacheExpiry.get(cacheKey);
      if (Date.now() < expiry) {
        console.log(`Using cached data for: ${cacheKey}`);
        return this.cache.get(cacheKey);
      }
    }

    // Fetch fresh data
    console.log(`Fetching fresh data for: ${cacheKey}`);
    const data = await fetcher();

    // Cache the result
    this.cache.set(cacheKey, data);
    this.cacheExpiry.set(cacheKey, Date.now() + ttl);

    return data;
  };
}

async loadCountries() {
  return this.getCachedData(
    'countries',
    () => this.orm.searchRead("res.country", [], ["name", "code"]),
    10 * 60 * 1000 // Cache for 10 minutes
  );
}
```

Service Error Recovery Strategies

Automatic Retry with Exponential Backoff

Exponential backoff means waiting longer between each retry attempt (1s, then 2s, then 4s, and so on) instead of retrying immediately — this gives a struggling server room to recover instead of hammering it with instant retries.

```
async performResilientOperation(operation, maxRetries = 3) {
  let lastError;

  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      const result = await operation();

      if (attempt > 1) {
        this.notification.add("Operation succeeded after retry.", {
          type: "success"
        });
      }
```

```
      return result;

    } catch (error) {
      lastError = error;
      console.warn(`Attempt ${attempt} failed:`, error);

      if (attempt < maxRetries) {
        const delay = Math.pow(2, attempt - 1) * 1000; // Exponential backoff

        this.notification.add(`Operation failed, retrying in ${delay/1000}s...
            ↪ (${attempt}/${maxRetries})`, {
          type: "warning"
        });

        await new Promise(resolve => setTimeout(resolve, delay));
      }
    }
  }

  // All retries failed
  this.notification.add(`Operation failed after ${maxRetries} attempts:
      ↪ ${lastError.message}`, {
    type: "danger",
    sticky: true
  });

  throw lastError;
}

// Usage
async saveData() {
  await this.performResilientOperation(async () => {
    return await this.orm.create("model.name", [this.state.formData]);
  });
}
```

Real-World Integration Example

Here's a complete example showing how multiple services work together in a production scenario. We'll look at it in two pieces: the setup and data loading, then the action that processes selected orders.

```
export class OrderProcessingCenter extends Component {
  static template = "my_module.OrderProcessingCenter";

  setup() {
    this.state = useState({
      orders: [],
      processing: false,
      selectedOrders: new Set(),
      statistics: {
        total: 0,
        pending: 0,
```

```
        processing: 0,
        completed: 0
      }
    });

    // Service initialization
    this.orm = useService("orm");
    this.action = useService("action");
    this.notification = useService("notification");
    this.dialog = useService("dialog");
    this.user = useService("user");

    // Load data on mount
    useEffect(
      () => {
        this.loadOrders();
        this.loadStatistics();
      },
      () => []
    );
  }

  async loadOrders() {
    try {
      const orders = await this.orm.searchRead(
        "sale.order",
        [["state", "in", ["draft", "sent", "sale"]]],
        ["name", "partner_id", "amount_total", "state", "date_order"],
        { order: "date_order desc", limit: 100 }
      );

      this.state.orders = orders;
    } catch (error) {
      this.notification.add("Failed to load orders.", { type: "danger" });
    }
  }

  async loadStatistics() {
    try {
      const stats = await this.orm.call("sale.order", "get_order_statistics");
      this.state.statistics = stats;
    } catch (error) {
      console.error("Failed to load statistics:", error);
    }
  }
```

The interesting part is `processSelectedOrders`: it validates the selection, confirms with the user, then coordinates a batch ORM call, a success notification, a data refresh, and finally opens a report — all five services working together around one user action:

```
  async processSelectedOrders() {
    if (this.state.selectedOrders.size === 0) {
      this.notification.add("Please select orders to process.", {
        type: "warning"
      });
      return;
```

```
    }

    const orderIds = Array.from(this.state.selectedOrders);

    // Confirmation dialog
    this.dialog.add(ConfirmationDialog, {
      title: "Process Orders",
      body: `Process ${orderIds.length} selected orders?`,
      confirm: async () => {
        try {
          this.state.processing = true;

          // Progress notification
          this.notification.add(`Processing ${orderIds.length} orders...`, {
            type: "info"
          });

          // Batch process orders
          await this.orm.call("sale.order", "action_confirm", orderIds);

          // Success feedback
          this.notification.add(`Successfully processed ${orderIds.length} orders!`, {
            type: "success"
          });

          // Clear selection and refresh
          this.state.selectedOrders.clear();
          await this.loadOrders();
          await this.loadStatistics();

          // Open processing report
          this.action.doAction({
            type: "ir.actions.report",
            report_name: "sale.action_report_saleorder",
            context: { active_ids: orderIds }
          });

        } catch (error) {
          console.error("Order processing failed:", error);

          this.notification.add(`Processing failed: ${error.message}`, {
            type: "danger",
            sticky: true
          });
        } finally {
          this.state.processing = false;
        }
      }
    });
  }
}
```

Summary

Odoo's built-in services provide a comprehensive toolkit for creating professional, integrated applications:

- **`notification`**: User feedback and status updates
- **`action`**: Navigation and Odoo integration
- **`dialog`**: User interactions and confirmations
- **`user`**, **`company`**, **`router`**: System information and navigation

By mastering these services, your components will: - Feel native to the Odoo experience - Provide consistent user interactions - Handle errors gracefully - Integrate seamlessly with Odoo's workflows

The key to success is understanding when and how to use each service, combining them effectively for complex workflows, and always prioritizing the user experience with clear feedback and intuitive interactions.

In the next chapter, we'll explore advanced component patterns and architectural considerations for building large-scale OWL applications.

TL;DR: Compose services deliberately (one notification per step of a multi-step workflow), debounce anything that fires on every keystroke with `@web/core/utils/timing`, and wrap unreliable calls in a retry-with-backoff strategy instead of failing silently.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch13_2_ex1`. Installation instructions are in the repository README.

Exercises

1. **Composed workflow**: Write a method that creates a partner, then immediately creates a related contact for it, showing one progress notification per step (similar to Pattern 3). Make sure a failure at either step shows which step failed.
2. **Debounce it yourself**: Implement a 300ms debounce over a live search input backed by `orm.searchRead`, without using Odoo's built-in `debounce` helper — then swap in `import { debounce } from "@web/core/utils/timing"` and confirm the behavior is the same.
3. **Retry with backoff**: Take `performResilientOperation` and adapt it to retry an `orm.call` up to 3 times, then write a quick test (real or on paper) describing what notification the user should see after each failed attempt.

What's Next?

You've now covered the full service layer, from first notification to resilient multi-step workflows. Chapter 14 moves to composition — slots, scoped slots, and `t-portal` — the tools for building reusable, flexible components instead of one-off ones.

Chapter 14: Advanced Composition with Slots

> **Why this chapter?** In real client work, you rarely get to build a component once and never touch it again — the next project asks for "the same modal, but with a different footer" or "the same list, but each row needs a custom action." Slots are what let you say yes without duplicating the component.
>
> **What is Odoo trying to solve with this?** Odoo's own UI (kanban cards, dialogs, list rows) has to stay flexible across wildly different models and use cases without an explosion of near-identical one-off widgets. Slots let a single component own the structure while callers own the content.
>
> **Real-world application:** A reusable confirmation-dialog wrapper used across a dozen addons for the same client, or a generic table component whose cell rendering changes per view without forking the table itself.

So far, our components have been self-contained units. A `Card` component always has a title and content. A `PartnerList` always renders a list of partners in a predefined format. But what if we want to create flexible, reusable components that can adapt to different use cases?

Imagine you're building a generic `Modal` component. The modal itself provides the background overlay, the white container, the positioning, and the close button functionality. But the content *inside* the modal—whether it's a confirmation message, a complex form, a list of images, or a video player—should be completely up to the parent component that uses it.

This is where **composition** becomes essential, and OWL's mechanism for achieving this is **slots**. Slots are placeholders in a child component's template that parent components can fill with their own content. Think of them as "content holes" that parents can plug with custom markup, creating infinite possibilities from a single, well-designed component.

This pattern is fundamental to building scalable component libraries and is used extensively throughout Odoo's interface.

Understanding the Problem: Why We Need Composition

Before diving into slots, let's understand the problem they solve with a concrete example.

The Rigid Approach (Without Slots)

Imagine we create a `ProductCard` component that displays product information:

```
// ProductCard.js
import { Component, xml } from "@odoo/owl";

export class ProductCard extends Component {
```

```
    static template = xml`
        <div class="product-card">
            <h3 t-esc="props.product.name"/>
            <p t-esc="props.product.description"/>
            <div class="price">$<t t-esc="props.product.price"/></div>
            <button class="btn btn-primary">Add to Cart</button>
        </div>
    `;
}
```

This works fine for a basic product listing. But what happens when you need: - A version with an image thumbnail? - A version with customer reviews? - A version with quantity selectors? - A version with discount badges?

Without composition, you'd end up creating multiple similar components (`ProductCardWithImage`, `ProductCardWithReviews`, etc.) or adding dozens of conditional props (`showImage`, `showReviews`, `showQuantity`). Both approaches lead to rigid, hard-to-maintain code.

The Flexible Approach (With Slots)

With slots, you create a single, flexible `ProductCard` component that provides the structure and styling, while allowing parents to inject custom content:

```
// FlexibleProductCard.js
export class FlexibleProductCard extends Component {
    static template = xml`
        <div class="product-card">
            <t t-slot="header"/>
            <h3 t-esc="props.product.name"/>
            <t t-slot="content"/>
            <div class="price">$<t t-esc="props.product.price"/></div>
            <t t-slot="actions"/>
        </div>
    `;
}
```

Now parents can customize each section:

```
<!-- In parent template -->
<FlexibleProductCard product="product">
  <t t-set-slot="header">
    <img t-att-src="product.imageUrl" class="product-thumbnail"/>
    <span class="badge badge-sale" t-if="product.onSale">SALE!</span>
  </t>

  <t t-set-slot="content">
    <p t-esc="product.description"/>
    <div class="reviews">
      <t t-foreach="product.reviews.slice(0, 3)" t-as="review" t-key="review.id">
        <div class="review"><t t-esc="review.text"/></div>
      </t>
    </div>
  </t>

  <t t-set-slot="actions">
    <input type="number" t-model="state.quantity" min="1" max="10"/>
    <button class="btn btn-primary" t-on-click="addToCart">
```

```
      Add <t t-esc="state.quantity"/> to Cart
    </button>
  </t>
</FlexibleProductCard>
```

This single component can now handle countless variations without becoming bloated or complex.

The Default Slot: Your First Step into Composition

The most common use case for slots is when you need a simple content placeholder. This is handled by the **default slot**—a single slot that doesn't need to be named.

Creating a Modal Component

Let's build a reusable `Modal` component step by step:

1. Define the Modal Structure (`modal.xml`)

```
<t t-name="my_module.Modal" owl="1">
  <div class="modal-backdrop" t-on-click="closeIfClickOutside">
    <div class="modal-dialog" t-on-click.stop="">
      <div class="modal-content">
        <!-- Modal Header -->
        <div class="modal-header">
          <h5 class="modal-title" t-esc="props.title or 'Modal'"/>
          <button type="button"
                  class="btn-close"
                  t-on-click="close"
                  aria-label="Close">
            <span aria-hidden="true">&times;</span>
          </button>
        </div>

        <!-- Modal Body - This is where parent content goes -->
        <div class="modal-body">
          <t t-slot="default"/>
        </div>

        <!-- Modal Footer (if provided) -->
        <div class="modal-footer" t-if="props.slots.footer">
          <t t-slot="footer"/>
        </div>
      </div>
    </div>
  </div>
</t>
```

2. The Modal Component Class (`modal.js`)

```
import { Component, onWillUnmount } from "@odoo/owl";

export class Modal extends Component {
    static template = "my_module.Modal";

    setup() {
```

```
        // Close modal when Escape key is pressed
        this.onKeydown = this.onKeydown.bind(this);
        document.addEventListener('keydown', this.onKeydown);

        onWillUnmount(() => {
            document.removeEventListener('keydown', this.onKeydown);
        });
    }

    onKeydown(event) {
        if (event.key === 'Escape') {
            this.close();
        }
    }

    close() {
        if (this.props.onClose) {
            this.props.onClose();
        }
    }

    closeIfClickOutside(event) {
        // Only close if clicking the backdrop, not the modal content
        if (event.target === event.currentTarget) {
            this.close();
        }
    }
}
```

3. Using the Modal from a Parent Component

```
<t t-name="my_module.Dashboard" owl="1">
  <div class="dashboard">
    <h1>My Dashboard</h1>
    <button class="btn btn-primary" t-on-click="openConfirmModal">
      Delete All Data
    </button>

    <!-- Confirmation Modal -->
    <t t-if="state.showConfirmModal">
      <Modal title="Confirm Deletion" onClose="closeConfirmModal">
        <!-- Everything between these tags goes into the default slot -->
        <div class="alert alert-danger">
          <h4>Warning</h4>
          <p>This action will permanently delete all your data. This cannot be
    ↪  undone.</p>
          <p>Type <strong>DELETE</strong> to confirm:</p>
          <input type="text"
                 class="form-control"
                 t-model="state.confirmText"
                 placeholder="Type DELETE here"/>
        </div>

        <!-- Footer slot for custom buttons -->
        <t t-set-slot="footer">
          <button class="btn btn-secondary" t-on-click="closeConfirmModal">
```

```
            Cancel
          </button>
          <button class="btn btn-danger"
                  t-att-disabled="state.confirmText !== 'DELETE'"
                  t-on-click="performDeletion">
            Delete Everything
          </button>
        </t>
      </Modal>
    </t>
  </div>
</t>
```

4. The Dashboard Component Logic

```
import { Component, useState } from "@odoo/owl";

export class Dashboard extends Component {
    static template = "my_module.Dashboard";
    static components = { Modal };

    setup() {
        this.state = useState({
            showConfirmModal: false,
            confirmText: ""
        });
    }

    openConfirmModal() {
        this.state.showConfirmModal = true;
        this.state.confirmText = "";
    }

    closeConfirmModal() {
        this.state.showConfirmModal = false;
    }

    performDeletion() {
        if (this.state.confirmText === 'DELETE') {
            // Perform the actual deletion
            console.log("Deleting all data...");
            this.closeConfirmModal();
        }
    }
}
```

Key Benefits of This Approach

1. Reusability: The same `Modal` component can display confirmation dialogs, forms, image galleries, or any other content.

2. Separation of Concerns: The modal handles positioning, backdrop, keyboard events, and styling. The parent handles the specific content and business logic.

3. Maintainability: Modal behavior is centralized. Fixing a bug or adding a feature (like animation) benefits all modals across your application.

Named Slots: Multiple Content Areas

When you need more control over where different pieces of content go, **named slots** allow you to define multiple, distinct areas for content injection.

Building a Flexible Page Layout

Let's create a `PageLayout` component that provides a standard page structure:

1. The PageLayout Template (`page_layout.xml`)

```
<t t-name="my_module.PageLayout" owl="1">
  <div class="page-container">
    <!-- Navigation Area -->
    <nav class="page-navigation" t-if="props.slots.navigation">
      <t t-slot="navigation"/>
    </nav>

    <!-- Header Section -->
    <header class="page-header">
      <t t-slot="header">
        <!-- Default header content if none provided -->
        <h1 t-esc="props.title or 'Page Title'"/>
      </t>
    </header>

    <!-- Sidebar (if provided) -->
    <aside class="page-sidebar" t-if="props.slots.sidebar">
      <t t-slot="sidebar"/>
    </aside>

    <!-- Main Content Area -->
    <main class="page-content" t-att-class="{ 'with-sidebar': props.slots.sidebar }">
      <t t-slot="default">
        <!-- Default content if none provided -->
        <p>No content provided.</p>
      </t>
    </main>

    <!-- Action Bar -->
    <div class="page-actions" t-if="props.slots.actions">
      <t t-slot="actions"/>
    </div>

    <!-- Footer -->
    <footer class="page-footer" t-if="props.slots.footer">
      <t t-slot="footer"/>
    </footer>
  </div>
</t>
```

2. Using Named Slots from Multiple Parents

Example 1: A Settings Page

```
<PageLayout title="User Settings">
  <t t-set-slot="navigation">
    <ul class="nav-menu">
      <li><a href="#profile">Profile</a></li>
      <li><a href="#security">Security</a></li>
      <li><a href="#notifications">Notifications</a></li>
    </ul>
  </t>

  <t t-set-slot="sidebar">
    <div class="settings-sidebar">
      <h3>Quick Settings</h3>
      <label>
        <input type="checkbox" t-model="state.darkMode"/>
        Dark Mode
      </label>
      <label>
        <input type="checkbox" t-model="state.notifications"/>
        Email Notifications
      </label>
    </div>
  </t>

  <!-- Main content goes to default slot -->
  <div class="settings-content">
    <h2>Profile Settings</h2>
    <form t-on-submit.prevent="saveSettings">
      <div class="form-group">
        <label>Full Name</label>
        <input type="text" t-model="state.user.name" class="form-control"/>
      </div>
      <div class="form-group">
        <label>Email</label>
        <input type="email" t-model="state.user.email" class="form-control"/>
      </div>
    </form>
  </div>

  <t t-set-slot="actions">
    <button class="btn btn-secondary" t-on-click="resetSettings">Reset</button>
    <button class="btn btn-primary" t-on-click="saveSettings">Save Changes</button>
  </t>
</PageLayout>
```

Example 2: A Product Catalog Page

```
<PageLayout title="Product Catalog">
  <t t-set-slot="header">
    <div class="catalog-header">
      <h1>Our Products</h1>
      <div class="search-bar">
        <input type="text"
               placeholder="Search products..."
               t-model="state.searchQuery"
               class="form-control"/>
      </div>
```

```
    </div>
  </t>

  <t t-set-slot="sidebar">
    <div class="filters">
      <h3>Filters</h3>
      <div class="filter-group">
        <label>Category</label>
        <select t-model="state.selectedCategory" class="form-control">
          <option value="">All Categories</option>
          <t t-foreach="state.categories" t-as="category" t-key="category.id">
            <option t-att-value="category.id" t-esc="category.name"/>
          </t>
        </select>
      </div>
      <div class="filter-group">
        <label>Price Range</label>
        <input type="range" t-model="state.maxPrice" min="0" max="1000"/>
        <span>Up to $<t t-esc="state.maxPrice"/></span>
      </div>
    </div>
  </t>

  <!-- Product grid in main content -->
  <div class="product-grid">
    <t t-foreach="filteredProducts" t-as="product" t-key="product.id">
      <ProductCard product="product"/>
    </t>
  </div>

  <t t-set-slot="footer">
    <div class="pagination">
      <button t-att-disabled="state.currentPage === 1"
              t-on-click="previousPage">
        Previous
      </button>
      <span>Page <t t-esc="state.currentPage"/> of <t t-esc="state.totalPages"/></span>
      <button t-att-disabled="state.currentPage === state.totalPages"
              t-on-click="nextPage">
        Next
      </button>
    </div>
  </t>
</PageLayout>
```

Understanding Slot Detection

Notice the `t-if="props.slots.sidebar"` pattern in our template. OWL automatically provides a `props.slots` object that tells you which slots the parent has filled:

```
<!-- Only render the sidebar container if parent provided sidebar content -->
<aside class="page-sidebar" t-if="props.slots.sidebar">
  <t t-slot="sidebar"/>
</aside>

<!-- Apply different CSS class based on whether sidebar exists -->
```

```
<main class="page-content" t-att-class="{ 'with-sidebar': props.slots.sidebar }">
  <t t-slot="default"/>
</main>
```

This allows your component to adapt its layout dynamically based on what content the parent provides.

Scoped Slots: Sharing Data Between Child and Parent

The most advanced slot pattern is **scoped slots**, where the child component passes data up to the parent's slot content. This enables incredibly flexible components that handle data logic while giving parents complete control over presentation.

Building a Generic Data Table

Let's create a `DataTable` component that handles sorting, filtering, and pagination, but allows parents to completely customize how each row is rendered:

1. The DataTable Component (`data_table.js`)

```
import { Component, useState } from "@odoo/owl";

export class DataTable extends Component {
    static template = "my_module.DataTable";

    setup() {
        this.state = useState({
            sortField: null,
            sortDirection: 'asc',
            currentPage: 1,
            pageSize: this.props.pageSize || 10,
            searchQuery: ""
        });
    }

    get filteredData() {
        let data = this.props.data || [];

        // Apply search filter
        if (this.state.searchQuery) {
            const query = this.state.searchQuery.toLowerCase();
            data = data.filter(item =>
                Object.values(item).some(value =>
                    String(value).toLowerCase().includes(query)
                )
            );
        }

        // Apply sorting
        if (this.state.sortField) {
            data = [...data].sort((a, b) => {
                const aVal = a[this.state.sortField];
                const bVal = b[this.state.sortField];
```

```
                if (aVal < bVal) return this.state.sortDirection === 'asc' ? -1 : 1;
                if (aVal > bVal) return this.state.sortDirection === 'asc' ? 1 : -1;
                return 0;
            });
        }

        return data;
    }

    get paginatedData() {
        const start = (this.state.currentPage - 1) * this.state.pageSize;
        const end = start + this.state.pageSize;
        return this.filteredData.slice(start, end);
    }

    get totalPages() {
        return Math.ceil(this.filteredData.length / this.state.pageSize);
    }

    sortBy(field) {
        if (this.state.sortField === field) {
            this.state.sortDirection = this.state.sortDirection === 'asc' ? 'desc' :
                ↪  'asc';
        } else {
            this.state.sortField = field;
            this.state.sortDirection = 'asc';
        }
        this.state.currentPage = 1; // Reset to first page when sorting
    }

    nextPage() {
        if (this.state.currentPage < this.totalPages) {
            this.state.currentPage++;
        }
    }

    previousPage() {
        if (this.state.currentPage > 1) {
            this.state.currentPage--;
        }
    }
}
```

2. The DataTable Template (`data_table.xml`)

```
<t t-name="my_module.DataTable" owl="1">
  <div class="data-table-container">
    <!-- Search Bar -->
    <div class="table-controls">
      <input type="text"
             class="form-control search-input"
             placeholder="Search..."
             t-model="state.searchQuery"/>
    </div>

    <!-- Table Header (if provided) -->
```

```
    <div class="table-header" t-if="props.slots.header">
      <t t-slot="header"
         sortBy="sortBy"
         sortField="state.sortField"
         sortDirection="state.sortDirection"/>
    </div>

    <!-- Table Body -->
    <div class="table-body">
      <t t-if="paginatedData.length === 0">
        <div class="empty-state">
          <t t-slot="empty">
            <p>No data available</p>
          </t>
        </div>
      </t>
      <t t-else="">
        <t t-foreach="paginatedData" t-as="item" t-key="item.id">
          <!-- Pass item data and utilities to parent slot -->
          <t t-slot="row"
             item="item"
             index="item_index"
             isEven="item_index % 2 === 0"/>
        </t>
      </t>
    </div>

    <!-- Pagination -->
    <div class="table-pagination" t-if="totalPages > 1">
      <button class="btn btn-sm btn-secondary"
              t-att-disabled="state.currentPage === 1"
              t-on-click="previousPage">
        Previous
      </button>

      <span class="pagination-info">
        Page <t t-esc="state.currentPage"/> of <t t-esc="totalPages"/>
        (<t t-esc="filteredData.length"/> total items)
      </span>

      <button class="btn btn-sm btn-secondary"
              t-att-disabled="state.currentPage === totalPages"
              t-on-click="nextPage">
        Next
      </button>
    </div>
  </div>
</t>
```

3. Using the Scoped Data Table

Now parents can use this powerful table while having complete control over how data is displayed:

```
<!-- In a parent component -->
<DataTable data="state.customers" pageSize="5">
  <!-- Custom sortable header -->
  <t t-set-slot="header" t-slot-scope="headerProps">
```

```
    <div class="table-header-row">
      <div class="header-cell sortable"
           t-on-click="() => headerProps.sortBy('name')"
           t-att-class="{
               'sort-asc': headerProps.sortField === 'name' && headerProps.sortDirection
    ↪  === 'asc',
               'sort-desc': headerProps.sortField === 'name' && headerProps.sortDirection
    ↪  === 'desc'
           }">
        Customer Name
      </div>
      <div class="header-cell sortable"
           t-on-click="() => headerProps.sortBy('email')">
        Email
      </div>
      <div class="header-cell sortable"
           t-on-click="() => headerProps.sortBy('totalOrders')">
        Total Orders
      </div>
      <div class="header-cell">Actions</div>
    </div>
  </t>

  <!-- Custom row rendering -->
  <t t-set-slot="row" t-slot-scope="rowProps">
    <div class="table-row" t-att-class="{ 'even': rowProps.isEven }">
      <div class="cell">
        <img t-att-src="rowProps.item.avatar" class="avatar-sm"/>
        <strong t-esc="rowProps.item.name"/>
        <span t-if="rowProps.item.isVip" class="badge badge-gold">VIP</span>
      </div>

      <div class="cell">
        <a t-att-href="`mailto:${rowProps.item.email}`" t-esc="rowProps.item.email"/>
      </div>

      <div class="cell">
        <span class="order-count" t-esc="rowProps.item.totalOrders"/>
        <small t-if="rowProps.item.lastOrderDate">
          (Last: <t t-esc="formatDate(rowProps.item.lastOrderDate)"/>)
        </small>
      </div>

      <div class="cell actions">
        <button class="btn btn-sm btn-primary"
                t-on-click="() => this.editCustomer(rowProps.item)">
          Edit
        </button>
        <button class="btn btn-sm btn-danger"
                t-on-click="() => this.deleteCustomer(rowProps.item)">
          Delete
        </button>
      </div>
    </div>
  </t>
```

```
  <!-- Custom empty state -->
  <t t-set-slot="empty">
    <div class="text-center p-4">
      <h3>No customers found</h3>
      <p>Try adjusting your search criteria or add your first customer.</p>
      <button class="btn btn-primary" t-on-click="openAddCustomerModal">
        Add First Customer
      </button>
    </div>
  </t>
</DataTable>
```

Understanding Scoped Slot Props

The key to scoped slots is the `t-slot-scope` directive on `<t t-set-slot>`:

```
<!-- Child passes data as attributes on the t-slot call -->
<t t-slot="row"
   item="item"
   index="item_index"
   isEven="item_index % 2 === 0"/>

<!-- Parent names the scope object with t-slot-scope and receives everything in it -->
<t t-set-slot="row" t-slot-scope="rowProps">
  <!-- rowProps contains: { item: {...}, index: 0, isEven: true } -->
  <div>Item name: <t t-esc="rowProps.item.name"/></div>
</t>
```

This pattern allows the child component to provide both data and utility functions to the parent's rendering logic.

Rendering Outside the Tree: `t-portal`

Slots let a parent control *what* a child renders. `t-portal` solves a different problem: it lets a component render *where* in the actual DOM its content appears, independent of where the component sits in OWL's component tree.

Why would you want that? Think of a modal dialog or a tooltip triggered from deep inside a page—say, from a table row inside a card inside a sidebar. If the modal renders normally, it inherits whatever `overflow: hidden` or `z-index` stacking context its ancestors set up, which can clip it or bury it behind other content. The fix web developers have used for years is to render that modal's DOM as a direct child of `<body>`, sidestepping every ancestor's styling—while still keeping it a normal, fully reactive OWL component logically owned by its parent.

`t-portal` does exactly that: the component's logical position in the tree (props, state, lifecycle) doesn't change, only *where its DOM ends up*.

```
class Tooltip extends Component {
    static template = xml`
        <span t-ref="anchor" t-on-mouseenter="() => state.visible = true">
            <t t-esc="props.label"/>
        </span>
        <div t-if="state.visible" t-portal="'body'" class="tooltip-content">
```

```
            <t t-esc="props.text"/>
        </div>`;

    setup() {
        this.state = useState({ visible: false });
    }
}
```

The `t-portal` attribute takes a CSS selector, as a string, for where the content should be attached—here, `'body'`. OWL inserts an empty placeholder node at the tooltip's original location to keep track of things internally, but the actual `.tooltip-content` element is appended to `<body>`, free of any clipping or stacking issues from its logical ancestors.

Reach for `t-portal` specifically for modals, tooltips, and dropdown menus—content that needs to visually escape its parent's layout while remaining, from OWL's point of view, an ordinary part of the component that created it.

Best Practices for Slot-Based Architecture

1. Design Slots with Purpose

Each slot should have a clear, single responsibility: - **`header`**: Page titles, navigation, search bars - **`actions`**: Buttons, controls, action menus
- **`footer`**: Pagination, totals, help text - **`default`**: Main content area

Avoid generic slots like `slot1`, `slot2` that don't convey meaning.

2. Provide Sensible Defaults

Always provide default content for slots when it makes sense:

```
<t t-slot="header">
  <!-- Default header if parent doesn't provide one -->
  <h1 t-esc="props.title or 'Untitled Page'"/>
</t>
```

This makes your component work out-of-the-box while still being customizable.

3. Document Your Slot API

When creating reusable components, document what slots are available and what props scoped slots provide:

```
/**
 * DataTable Component
 *
 * Slots:
 * - header: Table column headers (props: { sortBy, sortField, sortDirection })
 * - row: Individual row rendering (props: { item, index, isEven })
 * - empty: Shown when no data available
 *
 * Props:
 * - data: Array of objects to display
 * - pageSize: Number of items per page (default: 10)
 */
```

```
export class DataTable extends Component {
    // ...
}
```

4. Use Conditional Slot Rendering

Adapt your component's layout based on which slots are provided:

```
<!-- Adjust main content width based on sidebar presence -->
<main t-att-class="{
    'col-12': !props.slots.sidebar,
    'col-9': props.slots.sidebar
}">
  <t t-slot="default"/>
</main>
```

5. Combine Slots with Props for Maximum Flexibility

Some content areas might be simple enough for props, while others need the full power of slots:

```
// Simple text can be a prop
static props = {
    title: { type: String, optional: true },
    subtitle: { type: String, optional: true },
    // Complex content uses slots
};
```

```
<div class="card">
  <!-- Simple content via props -->
  <h3 t-if="props.title" t-esc="props.title"/>
  <p t-if="props.subtitle" t-esc="props.subtitle"/>

  <!-- Complex content via slots -->
  <t t-slot="default"/>
</div>
```

Common Pitfalls and How to Avoid Them

1. Overusing Slots

Not everything needs to be a slot. If content rarely changes, a prop might be simpler:

```
// Instead of this...
<Modal>
    <t t-set-slot="title">Save Changes?</t>
</Modal>

// Consider this...
<Modal title="Save Changes?">
```

2. Forgetting to Handle Empty Slots

Always check if a slot has content before rendering its container:

```
<!-- Good: Only render footer if content provided -->
<footer t-if="props.slots.footer" class="modal-footer">
  <t t-slot="footer"/>
</footer>

<!-- Bad: Empty footer div if no content -->
<footer class="modal-footer">
  <t t-slot="footer"/>
</footer>
```

3. Complex Logic in Templates

Keep slot logic simple. Move complex computations to getters or methods:

```
// Good: Logic in component
get shouldShowSidebar() {
    return this.props.slots.sidebar && !this.props.hideSidebar;
}

<!-- Simple template -->
<aside t-if="shouldShowSidebar">
  <t t-slot="sidebar"/>
</aside>
```

Mastering slots transforms how you think about component design. Instead of creating rigid, single-purpose components, you build flexible foundations that adapt to countless use cases. This approach is essential for building maintainable, scalable Odoo applications and is used throughout the Odoo codebase for maximum reusability and customization.

In the next chapter, we'll explore how to manage state that needs to be shared across components that aren't directly related in the component tree.

TL;DR: Slots let a parent inject content (default, named, or scoped) into a child's structure, so you build one flexible component instead of many rigid, near-identical ones.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch14_ex1`. Installation instructions are in the repository README.

Exercises

1. **Modal with two slots.** Build a `Modal` component with a default slot for the body and a named `footer` slot. Render it once with only a body, and once with both body and footer, confirming the footer area disappears cleanly when unused (see "Forgetting to Handle Empty Slots" above).
2. **Scoped slot practice.** Extend the `DataTable` from this chapter so the parent's slot receives not just the row `record`, but also the row's `index`. Use it in the parent template to render alternating row colors.
3. **Slot vs. prop.** Take a component you've built with a slot and ask: could this content instead be a simple prop? Rewrite it as a prop if so, and write one sentence on which version is easier to use from the parent's side.

What's Next?

We've covered how content flows between components that are directly related as parent and child. Chapter 15 tackles the harder case: state shared between components that aren't related at all in the tree, using a global store.

Chapter 15: Global State Management with useStore

> **Why this chapter?** Sooner or later, two components that have nothing to do with each other in the tree need to react to the same change — a notification badge and an unrelated list both needing to reflect a new record. Props alone can't reach across the tree, and reaching for a global mutable object or DOM events is how you get bugs nobody can trace.
>
> **What is Odoo trying to solve with this?** Coordinating shared UI state without abandoning OWL's reactivity model — a store should feel like `useState`, just visible from more than one place, registered the same way any other service is.
>
> **Real-world application:** A shared "unsaved changes" flag surfaced by several widgets on the same form view, or shared draft/cart state across a custom multi-step screen where the steps aren't in a direct parent-child relationship.

We've mastered the fundamentals of data flow in OWL: props flow down from parent to child, and events bubble up from child to parent. This pattern works beautifully for components that are directly related—a parent form and its input fields, a list and its items, a card and its buttons.

But what happens when components are far apart in the component tree, or when multiple unrelated components need to share the same data?

Imagine you have a `UserMenu` component in the main header that displays the current user's name and avatar. Somewhere else in your application, you have a `ProfileSettings` page where users can update their information. When a user changes their name on the `ProfileSettings` page, how does the `UserMenu` in the header—possibly rendered by a completely different part of the application—find out about it?

Passing props down through dozens of intermediate components (a painful anti-pattern known as "prop drilling") is inefficient and creates a maintenance nightmare. Emitting an event that has to bubble up through the entire application tree is equally messy and fragile.

The solution is a **global state store**—a centralized, single source of truth for data that needs to be shared across your entire application. Any component can access this data or trigger changes to it, and any component using that data will automatically update when it changes.

Understanding the Problem: When Local State Isn't Enough

Before diving into global state management, let's understand exactly when and why you need it.

The Prop Drilling Problem

Consider this component hierarchy in an Odoo application:

```
App
|-- Header
|   |-- Navigation
|   |-- UserMenu (needs user data)
|-- Main
|   |-- Sidebar
|   |   |-- QuickActions (needs user permissions)
|   |-- Content
|       |-- Dashboard (needs user preferences)
|       |-- ProfilePage
|           |-- ProfileForm (updates user data)
|-- Footer
    |-- UserInfo (needs user data)
```

Without global state management, sharing user data would require:

1. **Storing user data at the top level** (App component)
2. **Passing it down** through Header → UserMenu
3. **Passing it down** through Main → Sidebar → QuickActions

4. **Passing it down** through Main → Content → Dashboard
5. **Passing update functions up** from ProfileForm through Content → Main → App
6. **Passing data down** to Footer → UserInfo

This creates a web of props that have nothing to do with the intermediate components—they're just data couriers. It's fragile, verbose, and makes refactoring difficult.

The Callback Chain Problem

You might think, "I'll just use callback props!" But over long distances this approach has its own issues:

```
// In ProfileForm, deep in the component tree
this.props.onUserUpdated(newUserData);

// Every component in the chain needs to receive the callback and pass it down
// App -> Main -> Content -> ProfilePage -> ProfileForm...
// and the updated data still has to travel back down to UserMenu, QuickActions, etc.
```

This creates tight coupling between components that shouldn't know about each other, and it's easy to break the chain.

When You Need Global State

You should consider global state management when:

- **Cross-cutting concerns**: User authentication, theme settings, language preferences
- **Shared data**: Current user info, shopping cart contents, notification count
- **Remote data caching**: API responses that multiple components need
- **Application-wide settings**: Feature flags, configuration, permissions
- **Real-time updates**: WebSocket data that affects multiple components

---

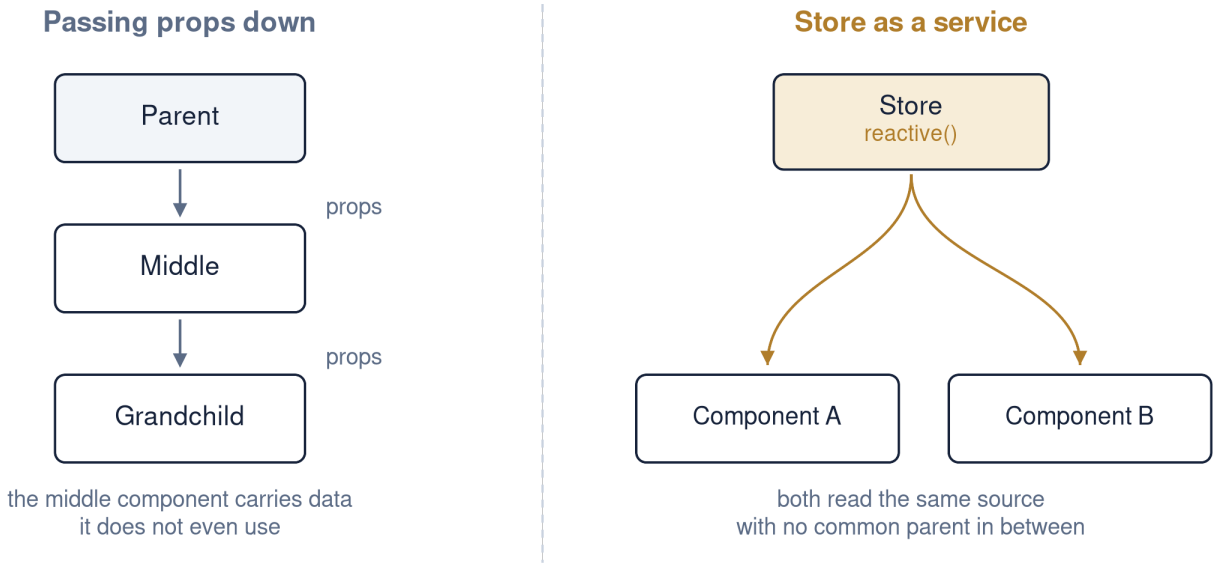


Figure 6: Prop drilling versus a store shared as a service.

Understanding Stores in Odoo

In Odoo's architecture, a **store** is a specialized service that manages reactive state. Unlike regular services that provide functionality, stores are specifically designed to hold data that components can subscribe to.

The Anatomy of a Store

A store in Odoo typically has these characteristics:

1. **Reactive State**: Uses OWL's `reactive` function to make data changes trigger component updates
2. **Centralized Logic**: All operations that modify the state are contained within the store
3. **Service Integration**: Registered as an Odoo service, making it available throughout the application
4. **Event System**: Can emit events when significant changes occur

Built-in Stores in Odoo

Before creating custom stores, it's important to know what Odoo already provides:

User Service: Information about the current user

```
const user = useService("user");
console.log(user.name, user.isAdmin, user.partnerId);
```

Company Service: Current company information

```
const company = useService("company");
console.log(company.currentCompany.name, company.allowedCompanies);
```

Notification Service: For displaying messages

```
const notification = useService("notification");
notification.add("Save successful!", { type: "success" });
```

Let's now create our own store to understand the pattern completely.

Creating Your First Store: Theme Management

Let's build a comprehensive theme store that manages dark/light mode, font size preferences, and color customization across an entire Odoo application.

1. Building the Theme Store Service

File: `my_module/static/src/services/theme_store.js`

```
/** @odoo-module **/

import { reactive } from "@odoo/owl";
import { registry } from "@web/core/registry";
import { browser } from "@web/core/browser/browser";

export class ThemeStore {
    constructor() {
        // Load saved preferences from localStorage
        const savedPreferences = this.loadPreferences();

        // Create reactive state
        this.state = reactive({
            mode: savedPreferences.mode || "light",
            fontSize: savedPreferences.fontSize || "medium",
            primaryColor: savedPreferences.primaryColor || "#007bff",
            borderRadius: savedPreferences.borderRadius || "medium",
            animations: savedPreferences.animations !== false, // default true
        });

        // Apply theme immediately when store is created
        this.applyTheme();
    }

    /**
     * Load preferences from localStorage
     */
    loadPreferences() {
        try {
            const saved = browser.localStorage.getItem('odoo_theme_preferences');
            return saved ? JSON.parse(saved) : {};
        } catch (error) {
            console.warn("Failed to load theme preferences:", error);
            return {};
        }
    }

    /**
     * Save current preferences to localStorage
     */
    savePreferences() {
        try {
            const preferences = {
                mode: this.state.mode,
```

```
                fontSize: this.state.fontSize,
                primaryColor: this.state.primaryColor,
                borderRadius: this.state.borderRadius,
                animations: this.state.animations,
            };
            browser.localStorage.setItem('odoo_theme_preferences',
↪   JSON.stringify(preferences));
        } catch (error) {
            console.warn("Failed to save theme preferences:", error);
        }
    }

    /**
     * Toggle between light and dark mode
     */
    toggleMode() {
        this.state.mode = this.state.mode === "light" ? "dark" : "light";
        this.applyTheme();
        this.savePreferences();
    }

    /**
     * Set a specific theme mode
     */
    setMode(mode) {
        if (["light", "dark", "auto"].includes(mode)) {
            this.state.mode = mode;
            this.applyTheme();
            this.savePreferences();
        }
    }

    /**
     * Set font size
     */
    setFontSize(size) {
        if (["small", "medium", "large", "xl"].includes(size)) {
            this.state.fontSize = size;
            this.applyTheme();
            this.savePreferences();
        }
    }

    /**
     * Set primary color
     */
    setPrimaryColor(color) {
        // Validate color format (hex)
        if (/^#[0-9A-F]{6}$/i.test(color)) {
            this.state.primaryColor = color;
            this.applyTheme();
            this.savePreferences();
        }
    }
```

```
    /**
     * Set border radius preference
     */
    setBorderRadius(radius) {
        if (["none", "small", "medium", "large"].includes(radius)) {
            this.state.borderRadius = radius;
            this.applyTheme();
            this.savePreferences();
        }
    }

    /**
     * Toggle animations on/off
     */
    toggleAnimations() {
        this.state.animations = !this.state.animations;
        this.applyTheme();
        this.savePreferences();
    }

    /**
     * Reset to default theme
     */
    resetToDefaults() {
        this.state.mode = "light";
        this.state.fontSize = "medium";
        this.state.primaryColor = "#007bff";
        this.state.borderRadius = "medium";
        this.state.animations = true;

        this.applyTheme();
        this.savePreferences();
    }

    /**
     * Apply current theme to document
     */
    applyTheme() {
        const root = document.documentElement;

        // Apply CSS custom properties
        root.style.setProperty('--primary-color', this.state.primaryColor);

        // Font size mapping
        const fontSizes = {
            small: '14px',
            medium: '16px',
            large: '18px',
            xl: '20px'
        };
        root.style.setProperty('--base-font-size', fontSizes[this.state.fontSize]);

        // Border radius mapping
        const borderRadiuses = {
            none: '0',
```

```
            small: '4px',
            medium: '8px',
            large: '16px'
        };
        root.style.setProperty('--border-radius',
            ↪ borderRadiuses[this.state.borderRadius]);

        // Apply theme mode class
        root.className = root.className.replace(/theme-\w+/g, '');
        root.classList.add(`theme-${this.state.mode}`);

        // Handle animations
        if (!this.state.animations) {
            root.classList.add('no-animations');
        } else {
            root.classList.remove('no-animations');
        }

        // Auto mode: detect system preference
        if (this.state.mode === 'auto') {
            const prefersDark = window.matchMedia('(prefers-color-scheme:
                ↪ dark)').matches;
            root.classList.add(`theme-${prefersDark ? 'dark' : 'light'}`);
        }
    }

    /**
     * Get computed theme values
     */
    get computedTheme() {
        return {
            isDark: this.state.mode === 'dark' ||
                   (this.state.mode === 'auto' &&
                    window.matchMedia('(prefers-color-scheme: dark)').matches),
            isLight: this.state.mode === 'light' ||
                    (this.state.mode === 'auto' &&
                     !window.matchMedia('(prefers-color-scheme: dark)').matches),
            ...this.state
        };
    }
}

// Register as an Odoo service
export const themeStoreService = {
    dependencies: [],
    start(env) {
        const store = new ThemeStore();

        // Listen for system theme changes when in auto mode
        window.matchMedia('(prefers-color-scheme: dark)').addEventListener('change', ()
            ↪ => {
            if (store.state.mode === 'auto') {
                store.applyTheme();
            }
        });
```

```
        return store;
    },
};

registry.category("services").add("themeStore", themeStoreService);
```

2. Accessing the Store with useService

Now any component can access this theme store:

Component: theme_display.js

```
/** @odoo-module **/

import { Component } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ThemeDisplay extends Component {
    static template = xml`
        <div class="theme-display">
            <h3>Current Theme</h3>
            <div class="theme-info">
                <p><strong>Mode:</strong> <span t-esc="themeStore.state.mode"/></p>
                <p><strong>Font Size:</strong> <span
    ↪  t-esc="themeStore.state.fontSize"/></p>
                <p><strong>Primary Color:</strong>
                   <span t-esc="themeStore.state.primaryColor"/>
                   <div class="color-preview"
                        t-att-style="\`background-color:
    ↪  \${themeStore.state.primaryColor}\`"/>
                </p>
                <p><strong>Border Radius:</strong>
                   <span t-esc="themeStore.state.borderRadius"/></p>
                <p><strong>Animations:</strong>
                   <span t-esc="themeStore.state.animations ? 'Enabled' :
    ↪  'Disabled'"/></p>
            </div>
        </div>
    `;

    setup() {
        this.themeStore = useService("themeStore");
    }
}
```

Component: theme_controls.js

```
/** @odoo-module **/

import { Component } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ThemeControls extends Component {
    static template = xml`
        <div class="theme-controls">
            <h3>Theme Settings</h3>
```

```
        <!-- Mode Selection -->
        <div class="control-group">
            <label>Theme Mode:</label>
            <select t-model="themeStore.state.mode" t-on-change="onModeChange">
                <option value="light">Light</option>
                <option value="dark">Dark</option>
                <option value="auto">Auto (System)</option>
            </select>
        </div>

        <!-- Font Size -->
        <div class="control-group">
            <label>Font Size:</label>
            <select t-model="themeStore.state.fontSize"
↪   t-on-change="onFontSizeChange">
                <option value="small">Small</option>
                <option value="medium">Medium</option>
                <option value="large">Large</option>
                <option value="xl">Extra Large</option>
            </select>
        </div>

        <!-- Primary Color -->
        <div class="control-group">
            <label>Primary Color:</label>
            <input type="color"
                   t-att-value="themeStore.state.primaryColor"
                   t-on-change="onColorChange"/>
        </div>

        <!-- Border Radius -->
        <div class="control-group">
            <label>Border Radius:</label>
            <select t-model="themeStore.state.borderRadius"
↪   t-on-change="onRadiusChange">
                <option value="none">None</option>
                <option value="small">Small</option>
                <option value="medium">Medium</option>
                <option value="large">Large</option>
            </select>
        </div>

        <!-- Animations Toggle -->
        <div class="control-group">
            <label>
                <input type="checkbox"
                       t-att-checked="themeStore.state.animations"
                       t-on-change="onAnimationToggle"/>
                Enable Animations
            </label>
        </div>

        <!-- Quick Actions -->
        <div class="quick-actions">
```

```
                <button class="btn btn-secondary" t-on-click="themeStore.toggleMode">
                    Quick Toggle Mode
                </button>
                <button class="btn btn-outline-secondary"
                        t-on-click="themeStore.resetToDefaults">
                    Reset to Defaults
                </button>
            </div>
        </div>
    `;

    setup() {
        this.themeStore = useService("themeStore");
    }

    onModeChange(event) {
        this.themeStore.setMode(event.target.value);
    }

    onFontSizeChange(event) {
        this.themeStore.setFontSize(event.target.value);
    }

    onColorChange(event) {
        this.themeStore.setPrimaryColor(event.target.value);
    }

    onRadiusChange(event) {
        this.themeStore.setBorderRadius(event.target.value);
    }

    onAnimationToggle(event) {
        this.themeStore.toggleAnimations();
    }
}
```

3. CSS Integration

Add CSS that responds to your theme variables:

File: `my_module/static/src/css/theme.css`

```
/* CSS Custom Properties that the store controls */
:root {
    --primary-color: #007bff;
    --base-font-size: 16px;
    --border-radius: 8px;
}

/* Base typography */
body {
    font-size: var(--base-font-size);
    transition: font-size 0.3s ease;
}

/* Theme mode styles */
```

```
.theme-light {
    --bg-color: #ffffff;
    --text-color: #333333;
    --border-color: #dee2e6;
}

.theme-dark {
    --bg-color: #1a1a1a;
    --text-color: #ffffff;
    --border-color: #444444;
}

body {
    background-color: var(--bg-color);
    color: var(--text-color);
    border-color: var(--border-color);
}

/* Component styles that use theme variables */
.btn-primary {
    background-color: var(--primary-color);
    border-radius: var(--border-radius);
}

.card {
    border-radius: var(--border-radius);
    border-color: var(--border-color);
}

/* Disable animations when requested */
.no-animations * {
    animation-duration: 0s !important;
    transition-duration: 0s !important;
}

/* Theme controls styling */
.theme-controls .control-group {
    margin-bottom: 1rem;
}

.theme-controls label {
    display: block;
    margin-bottom: 0.5rem;
    font-weight: 500;
}

.color-preview {
    width: 20px;
    height: 20px;
    border-radius: var(--border-radius);
    display: inline-block;
    margin-left: 8px;
    border: 1px solid var(--border-color);
}
```

Advanced Store Patterns

1. Store with Async Operations

Many stores need to interact with APIs. Here's a pattern for a shopping cart store:

```
export class ShoppingCartStore {
    constructor(orm, notification) {
        this.orm = orm;
        this.notification = notification;

        this.state = reactive({
            items: [],
            loading: false,
            total: 0,
        });

        this.loadCart();
    }

    async loadCart() {
        this.state.loading = true;
        try {
            const cartData = await this.orm.call("sale.order", "get_current_cart", []);
            this.state.items = cartData.items;
            this.state.total = cartData.total;
        } catch (error) {
            this.notification.add("Failed to load cart", { type: "danger" });
        } finally {
            this.state.loading = false;
        }
    }

    async addItem(productId, quantity = 1) {
        this.state.loading = true;
        try {
            await this.orm.call("sale.order", "add_to_cart", [productId, quantity]);
            await this.loadCart(); // Refresh cart data
            this.notification.add("Item added to cart", { type: "success" });
        } catch (error) {
            this.notification.add("Failed to add item", { type: "danger" });
        } finally {
            this.state.loading = false;
        }
    }

    async removeItem(itemId) {
        this.state.loading = true;
        try {
            await this.orm.call("sale.order", "remove_from_cart", [itemId]);
            await this.loadCart();
            this.notification.add("Item removed", { type: "info" });
        } catch (error) {
            this.notification.add("Failed to remove item", { type: "danger" });
        } finally {
            this.state.loading = false;
```

```
        }
    }

    get itemCount() {
        return this.state.items.reduce((total, item) => total + item.quantity, 0);
    }

    get isEmpty() {
        return this.state.items.length === 0;
    }
}

export const shoppingCartService = {
    dependencies: ["orm", "notification"],
    start(env, { orm, notification }) {
        return new ShoppingCartStore(orm, notification);
    },
};
```

2. Store with Computed Properties

For complex derived state, use getters:

```
export class NotificationStore {
    constructor() {
        this.state = reactive({
            notifications: [],
            settings: {
                email: true,
                push: true,
                sms: false,
            }
        });
    }

    get unreadCount() {
        return this.state.notifications.filter(n => !n.read).length;
    }

    get urgentNotifications() {
        return this.state.notifications.filter(n => n.priority === 'urgent' && !n.read);
    }

    get hasUrgentNotifications() {
        return this.urgentNotifications.length > 0;
    }

    get notificationsByType() {
        return this.state.notifications.reduce((acc, notification) => {
            if (!acc[notification.type]) {
                acc[notification.type] = [];
            }
            acc[notification.type].push(notification);
            return acc;
        }, {});
    }
```

```
}
```

3. Store with Event Emitting

For complex applications, stores can emit events:

```
import { EventBus } from "@odoo/owl";

export class UserActivityStore extends EventBus {
    constructor() {
        super();
        this.state = reactive({
            isOnline: navigator.onLine,
            lastActivity: Date.now(),
            idleTime: 0,
        });

        this.setupActivityTracking();
    }

    setupActivityTracking() {
        // Track online/offline status
        window.addEventListener('online', () => {
            this.state.isOnline = true;
            this.trigger('connectivity-changed', { online: true });
        });

        window.addEventListener('offline', () => {
            this.state.isOnline = false;
            this.trigger('connectivity-changed', { online: false });
        });

        // Track user activity
        const resetActivity = () => {
            this.state.lastActivity = Date.now();
            this.state.idleTime = 0;
        };

        ['mousedown', 'mousemove', 'keypress', 'scroll', 'touchstart'].forEach(event => {
            document.addEventListener(event, resetActivity, true);
        });

        // Check for idle users every minute
        setInterval(() => {
            this.state.idleTime = Date.now() - this.state.lastActivity;

            if (this.state.idleTime > 5 * 60 * 1000) { // 5 minutes
                this.trigger('user-idle', { idleTime: this.state.idleTime });
            }
        }, 60000);
    }
}
```

Best Practices for Store Management

1. Keep Stores Focused

Each store should have a single, clear responsibility:

```
// Good: Focused stores
- UserStore: Current user info and authentication
- ThemeStore: UI preferences and theming
- CartStore: Shopping cart state and operations
- NotificationStore: In-app notifications

// Bad: God object store
- AppStore: Everything mixed together
```

2. Use Getters for Computed Values

Don't store derived values in state—compute them with getters:

```
// Good: Computed property
get totalPrice() {
    return this.state.items.reduce((sum, item) => sum + item.price * item.quantity, 0);
}

// Bad: Stored derived value (can get out of sync)
this.state.totalPrice = /* computed value */;
```

3. Make State Changes Explicit

Always provide methods for modifying state, don't allow direct mutations:

```
// Good: Controlled mutations
addNotification(message, type = 'info') {
    this.state.notifications.push({
        id: Date.now(),
        message,
        type,
        timestamp: new Date(),
        read: false
    });
}

// Bad: Direct state mutation from components
// notificationStore.state.notifications.push(...);
```

4. Handle Loading States

For async operations, always provide loading indicators:

```
async loadUserData() {
    this.state.loading = true;
    this.state.error = null;

    try {
        const userData = await this.orm.call("res.users", "get_current_user", []);
        this.state.user = userData;
    } catch (error) {
```

```
        this.state.error = error.message;
    } finally {
        this.state.loading = false;
    }
}
```

5. Provide Clear APIs

Document your store's public interface:

```
/**
 * Theme Store
 *
 * State:
 * - mode: 'light' | 'dark' | 'auto'
 * - fontSize: 'small' | 'medium' | 'large' | 'xl'
 * - primaryColor: hex color string
 *
 * Methods:
 * - toggleMode(): Toggle between light/dark
 * - setMode(mode): Set specific mode
 * - setFontSize(size): Set font size
 * - setPrimaryColor(color): Set primary color
 * - resetToDefaults(): Reset all settings
 *
 * Computed:
 * - computedTheme: Object with resolved theme values
 */
```

When NOT to Use Global State

Global state is powerful, but it's not always the right solution:

Use Local State When:

- Data is only used by one component and its children
- Data doesn't need to persist between component mounts/unmounts
- The relationship between components is clear and direct

Use Props/Events When:

- Components have a clear parent-child relationship
- Data flow is simple and unidirectional
- You want to keep components loosely coupled

Use Global State When:

- Multiple unrelated components need the same data
- Data needs to persist across navigation
- You're dealing with user preferences or application settings
- You need to coordinate state across different parts of the app

---

Debugging Store Issues

1. Add Logging to Store Methods

```
setMode(mode) {
    console.log(`Theme mode changing from ${this.state.mode} to ${mode}`);
    this.state.mode = mode;
    this.applyTheme();
    this.savePreferences();
}
```

2. Use Browser DevTools

Stores are regular JavaScript objects, so you can inspect them in the console:

```
// In browser console
const themeStore = odoo.__SERVICES__.themeStore;
console.log(themeStore.state);
```

3. Add Validation

```
setFontSize(size) {
    const validSizes = ["small", "medium", "large", "xl"];
    if (!validSizes.includes(size)) {
        console.error(`Invalid font size: ${size}. Valid options: ${validSizes.join(',
            ↪  ')}`);
        return;
    }

    this.state.fontSize = size;
    this.applyTheme();
    this.savePreferences();
}
```

Real-World Integration Example

Here is a complete example showing how multiple services work together in a production scenario:

```
export class DashboardStore {
    constructor(orm, notification, user) {
        this.orm = orm;
        this.notification = notification;
        this.user = user;

        this.state = reactive({
            // Dashboard data
            metrics: {
                totalSales: 0,
                activeCustomers: 0,
                pendingOrders: 0,
                monthlyRevenue: 0
            },

            // Loading states
```

```
            loadingMetrics: false,
            loadingCharts: false,

            // Dashboard configuration
            layout: this.user.dashboardLayout || 'grid',
            visibleWidgets: this.user.visibleWidgets || ['sales', 'customers', 'orders'],

            // Chart data
            salesData: [],
            customerData: [],

            // Filters
            dateRange: 'last-month',
            selectedTeam: null,

            // Errors
            errors: {}
        });

        // Load initial data
        this.initialize();
    }

    async initialize() {
        try {
            await Promise.all([
                this.loadMetrics(),
                this.loadChartData()
            ]);

            this.notification.add("Dashboard loaded successfully", {
                type: "success"
            });
        } catch (error) {
            this.notification.add("Failed to load dashboard", {
                type: "danger",
                sticky: true
            });

            console.error("Dashboard initialization error:", error);
        }
    }
```

The constructor builds the reactive `state` shape and immediately kicks off `initialize()`, which loads everything the dashboard needs in parallel and shows a notification either way. Next come the two loader methods that `initialize()` calls — each follows the same `loading` flag / `try/catch/finally` pattern you saw earlier in this chapter:

```
    async loadMetrics() {
        this.state.loadingMetrics = true;
        this.state.errors.metrics = null;

        try {
            const metrics = await this.orm.call(
                "dashboard.analytics",
                "get_metrics",
```

```
                [],
                {
                    date_from: this.getDateFrom(),
                    date_to: this.getDateTo(),
                    team_id: this.state.selectedTeam
                }
            );

            this.state.metrics = {
                totalSales: metrics.total_sales,
                activeCustomers: metrics.active_customers,
                pendingOrders: metrics.pending_orders,
                monthlyRevenue: metrics.monthly_revenue
            };
        } catch (error) {
            this.state.errors.metrics = error.message;
            console.error("Failed to load metrics:", error);
        } finally {
            this.state.loadingMetrics = false;
        }
    }

    async loadChartData() {
        this.state.loadingCharts = true;
        this.state.errors.charts = null;

        try {
            const [salesData, customerData] = await Promise.all([
                this.orm.call("dashboard.analytics", "get_sales_chart_data", [], {
                    period: this.state.dateRange
                }),
                this.orm.call("dashboard.analytics", "get_customer_chart_data", [], {
                    period: this.state.dateRange
                })
            ]);

            this.state.salesData = salesData;
            this.state.customerData = customerData;
        } catch (error) {
            this.state.errors.charts = error.message;
            console.error("Failed to load chart data:", error);
        } finally {
            this.state.loadingCharts = false;
        }
    }
```

With loading handled, the rest of the store is the public API other components actually call: small methods that change one piece of state and, when needed, trigger a reload or a save. Notice none of these methods reach into `this.state` from outside the store — every mutation goes through a named method, which is what keeps a growing store debuggable:

```
    // Configuration change methods
    changeDateRange(newRange) {
        if (this.state.dateRange !== newRange) {
            this.state.dateRange = newRange;
```

```
            this.refreshData();
        }
    }

    selectTeam(teamId) {
        if (this.state.selectedTeam !== teamId) {
            this.state.selectedTeam = teamId;
            this.refreshData();
        }
    }

    toggleWidget(widgetId) {
        const widgets = [...this.state.visibleWidgets];
        const index = widgets.indexOf(widgetId);

        if (index > -1) {
            widgets.splice(index, 1);
        } else {
            widgets.push(widgetId);
        }

        this.state.visibleWidgets = widgets;
        this.saveUserConfig();
    }

    changeLayout(newLayout) {
        this.state.layout = newLayout;
        this.saveUserConfig();
    }

    async refreshData() {
        await Promise.all([
            this.loadMetrics(),
            this.loadChartData()
        ]);
    }

    async saveUserConfig() {
        try {
            await this.orm.call("res.users", "save_dashboard_config", [], {
                layout: this.state.layout,
                visible_widgets: this.state.visibleWidgets
            });

            this.notification.add("Configuration saved", { type: "success" });
        } catch (error) {
            this.notification.add("Failed to save configuration", { type: "warning" });
        }
    }
```

Finally, a couple of plain helper methods and three computed getters expose derived data (is anything loading? are there errors? what's the current full configuration?) without duplicating it in `state`, followed by the service registration that turns this class into a **singleton** — a single shared instance created once and reused everywhere — that every component reaches via `useService("dashboardStore")`:

```
    getDateFrom() {
        const now = new Date();
        switch (this.state.dateRange) {
            case 'last-week':
                return new Date(now.getTime() - 7 * 24 * 60 * 60 * 1000);
            case 'last-month':
                return new Date(now.getFullYear(), now.getMonth() - 1, now.getDate());
            case 'last-quarter':
                return new Date(now.getFullYear(), now.getMonth() - 3, now.getDate());
            case 'last-year':
                return new Date(now.getFullYear() - 1, now.getMonth(), now.getDate());
            default:
                return new Date(now.getFullYear(), now.getMonth() - 1, now.getDate());
        }
    }

    getDateTo() {
        return new Date();
    }

    // Computed getters
    get hasErrors() {
        return Object.keys(this.state.errors).some(key => this.state.errors[key]);
    }

    get isLoading() {
        return this.state.loadingMetrics || this.state.loadingCharts;
    }

    get fullConfiguration() {
        return {
            layout: this.state.layout,
            widgets: this.state.visibleWidgets,
            filters: {
                dateRange: this.state.dateRange,
                team: this.state.selectedTeam
            }
        };
    }
}

// Dashboard Store service
export const dashboardStoreService = {
    dependencies: ["orm", "notification", "user"],
    start(env, { orm, notification, user }) {
        return new DashboardStore(orm, notification, user);
    },
};

registry.category("services").add("dashboardStore", dashboardStoreService);
```

Common Pitfalls

1. Forgetting to Subscribe with `useState`

Creating the store with `reactive({...})` only makes the data reactive at the *source*. A component that reads the store directly (`useService("dashboardStore").state.metrics`) without wrapping it in `useState` won't re-render when that data changes:

```
// BAD: reads the reactive state once, component never re-renders on changes
this.store = useService("dashboardStore");

// GOOD: subscribes this component to the store's reactivity
this.store = useState(useService("dashboardStore").state);
```

2. Mutating the Store from Anywhere

If any component can do `this.store.state.selectedTeam = 5` directly, you lose track of who changes what and why. Keep every mutation behind a named method on the store (`selectTeam(id)`), even when the method is a one-liner — it gives you a single place to add logging, validation, or a side effect later.

3. One Store Instance per Component

Registering the store as a service (as shown above) guarantees a single shared instance. Instantiating `new DashboardStore(...)` directly inside a component's `setup()` creates a private copy that no other component can see — defeating the entire purpose of global state.

4. Reaching for Global State Too Early

Not everything needs to live in a store. If only one component (and its direct children, via props) ever reads a piece of data, local `useState` is simpler and easier to reason about. Promote state to a store only when two or more unrelated components genuinely need to share it.

Global state management with stores is essential for building complex, interconnected Odoo applications. By centralizing shared state and providing clear APIs for accessing and modifying it, you create applications that are more maintainable, more predictable, and easier to debug.

The key is knowing when to use global state versus local state, and designing your stores with clear responsibilities and clean interfaces. Master this pattern, and you'll be able to build sophisticated applications that scale gracefully as they grow in complexity.

In the next chapter, we'll explore how to ensure your components work correctly through testing and effective debugging techniques.

TL;DR: Register a `reactive()` store as a service and consume it with `useState(useService(...).state)` wherever it's needed, so unrelated components can share reactive state without prop drilling.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch15_ex1`. Installation instructions are in the repository README.

Exercises

1. **Shared counter store.** Create a `counterStore` service using `reactive({ count: 0 })` with `increment()` and `decrement()` methods. Consume it from two sibling components with `useState(useService("counterStore").state)` and confirm that clicking a button in one updates the count shown in the other.
2. **Add a computed getter.** Extend the store from Exercise 1 with a `get isPositive()` getter, and use it in a template to conditionally show a warning when the count goes negative.
3. **Local vs. global.** Take a component you built in an earlier chapter's exercises (or the `DataTable` from Chapter 14) and decide: does any of its state genuinely need to be global? Write one sentence justifying your answer before writing any code.

What's Next?

State that's correct isn't the same as state you can trust in front of a client — Chapter 16 Part 1 covers how to test and debug OWL components so you catch regressions before they do.

Chapter 16 Part 1: Testing and Debugging Fundamentals

> **Why this chapter?** "It works on my machine" isn't good enough when you're the one on the hook for a client's production instance — tests are what let you touch code with confidence after the first release, instead of dreading every change.
>
> **What is Odoo trying to solve with this?** A JS test framework that can boot a mock Odoo environment with the right services quickly, so testing a small addon doesn't require a full running server and database.
>
> **Real-world application:** Catching a regression in code review, before a routine update breaks a dashboard a client's sales team relies on every morning.

Writing code that works today is good. Writing code that you can confidently change tomorrow without breaking everything is exceptional. The difference lies in having a comprehensive testing strategy and mastering debugging techniques.

Testing is not just about catching bugs—it's about designing better components, documenting expected behavior, and creating a safety net that allows you to refactor and improve your code with confidence. When combined with effective debugging skills, testing transforms development from a series of hopeful experiments into a methodical, predictable process.

Odoo provides a powerful JavaScript testing framework built on `@odoo/hoot` (the modern unit-test runner used since Odoo 17.4, replacing the older QUnit-based framework), specifically designed for testing OWL components in realistic conditions. This part of the chapter takes you from writing your first simple test to testing components that depend on services, plus the debugging tools and best practices you'll use every day.

Why Testing Matters in OWL Development

Before diving into the "how," let's understand the "why" of testing OWL components.

The Reality of Component Interdependence

OWL components don't exist in isolation. They: - Receive props from parents and pass props to children - Emit events that parents listen to - Use services to interact with Odoo's backend - Manipulate the DOM through templates - Manage local state that affects rendering

Each of these interactions is a potential point of failure. Without tests, you're essentially flying blind—making changes and hoping nothing breaks.

The Cost of Manual Testing

Consider a typical development workflow without automated tests:

1. **Make a change** to a component
2. **Start the Odoo server** (30-60 seconds)
3. **Navigate to the page** where the component is used
4. **Manually interact** with the component to verify it works
5. **Check edge cases** by manually creating different scenarios
6. **Repeat** for every component that might be affected

For a single small change, this process might take 5-10 minutes. Multiply that by dozens of changes per day, and you're spending hours on repetitive manual testing.

The Benefits of Automated Testing

Automated tests change this completely:

- **Speed**: Tests run in seconds, not minutes
- **Consistency**: Tests check the same things every time, in the same way
- **Coverage**: Tests can check edge cases you might forget manually
- **Confidence**: Green tests mean your changes haven't broken existing functionality
- **Documentation**: Tests serve as executable examples of how components should behave

---

Setting Up Your Testing Environment

1. Test File Organization

Odoo follows a clear convention for organizing test files:

```
my_module/
|-- static/src/
|   |-- components/
|   |   |-- counter/
|   |   |   |-- counter.js
|   |   |   |-- counter.xml
|   |   |   |-- tests/
|   |   |       |-- counter.test.js
|   |   |-- task_list/
|   |       |-- task_list.js
|   |       |-- task_list.xml
|   |       |-- tests/
|   |           |-- task_list.test.js
|   |-- services/
|       |-- theme_store.js
|       |-- tests/
|           |-- theme_store.test.js
```

This organization keeps tests close to the code they're testing, making them easy to find and maintain.

2. Basic Test File Structure

Every Odoo JavaScript test file follows this pattern:

```
/** @odoo-module **/

import { describe, test, beforeEach } from "@odoo/hoot";
import { mountWithCleanup } from "@web/../tests/web_test_helpers";

// Import the component you're testing
import { Counter } from "../counter";

describe("Counter", () => {
    beforeEach(() => {
        // Runs before every test in this describe() block
    });

    test("basic rendering", async () => {
        // Test implementation goes here
    });
});
```

Key helpers explained: - **`describe()`**: Groups related tests together, similar to a test suite - **`test()`**: Defines a single test case - **`beforeEach()`**: Runs setup code before each test in the surrounding `describe()` - **`mountWithCleanup()`**: Mounts a component into the test page and automatically unmounts it when the test ends — no manual `target`/`env`/`.destroy()` bookkeeping needed, unlike the older QUnit-based framework

3. Adding Tests to Your Module

Don't forget to include your test files in `__manifest__.py`:

```
{
    'name': 'My Module',
    # ... other manifest data ...
    'assets': {
        'web.assets_backend': [
            'my_module/static/src/components/**/*.js',
            'my_module/static/src/components/**/*.xml',
        ],
        'web.assets_unit_tests': [
            'my_module/static/src/components/**/tests/*.test.js',
        ],
    },
}
```

Writing Your First Component Test

Let's start with a simple but realistic example: testing a `Counter` component.

The Counter Component

First, here's the component we'll be testing:

File: `counter.js`

```
/** @odoo-module **/
```

```
import { Component, useState } from "@odoo/owl";

export class Counter extends Component {
    static template = xml`
        <div class="counter-widget">
            <h3>Counter: <span class="counter-value" t-esc="state.count"/></h3>
            <div class="counter-controls">
                <button class="btn btn-secondary decrement-btn"
                        t-on-click="decrement"
                        t-att-disabled="state.count <= props.min">
                    -
                </button>
                <button class="btn btn-primary reset-btn" t-on-click="reset">
                    Reset
                </button>
                <button class="btn btn-secondary increment-btn"
                        t-on-click="increment"
                        t-att-disabled="state.count >= props.max">
                    +
                </button>
            </div>
            <div class="counter-info" t-if="showInfo">
                <small>Min: <t t-esc="props.min"/>, Max: <t t-esc="props.max"/></small>
            </div>
        </div>
    `;

    static props = {
        initialValue: { type: Number, optional: true },
        min: { type: Number, optional: true },
        max: { type: Number, optional: true },
        onValueChange: { type: Function, optional: true },
    };

    static defaultProps = {
        initialValue: 0,
        min: 0,
        max: 100,
    };

    setup() {
        this.state = useState({
            count: this.props.initialValue,
        });
    }

    get showInfo() {
        return this.props.min !== 0 || this.props.max !== 100;
    }

    increment() {
        if (this.state.count < this.props.max) {
            this.state.count++;
            this._notifyChange();
        }
```

```
    }

    decrement() {
        if (this.state.count > this.props.min) {
            this.state.count--;
            this._notifyChange();
        }
    }

    reset() {
        this.state.count = this.props.initialValue;
        this._notifyChange();
    }

    _notifyChange() {
        if (this.props.onValueChange) {
            this.props.onValueChange(this.state.count);
        }
    }
}
```

Comprehensive Test Suite

Now let's write thorough tests for this component. We'll build the suite incrementally, a few tests at a time, so it's easier to see what each group is checking.

File: `counter.test.js`

First, the rendering tests — they check that the component displays the right thing given different props:

```
/** @odoo-module **/

import { describe, test, expect } from "@odoo/hoot";
import { mountWithCleanup } from "@web/../tests/web_test_helpers";

import { Counter } from "../counter";

describe("Counter", () => {
    test("renders with default props", async () => {
        // Mount component with no props (should use defaults)
        await mountWithCleanup(Counter);

        // Check initial display
        expect(".counter-value").toHaveText("0");

        // Check that buttons exist
        expect(".increment-btn").toHaveCount(1);
        expect(".decrement-btn").toHaveCount(1);
        expect(".reset-btn").toHaveCount(1);

        // Check that info is not shown (default min/max)
        expect(".counter-info").toHaveCount(0);
    });

    test("renders with custom props", async () => {
```

```
        await mountWithCleanup(Counter, {
            props: {
                initialValue: 5,
                min: 2,
                max: 8,
            },
        });

        // Check custom initial value
        expect(".counter-value").toHaveText("5");

        // Check that info is shown, with the custom min/max values
        expect(".counter-info").toHaveCount(1);
        expect(".counter-info").toHaveText("Min: 2, Max: 8");
    });
});
```

Key matchers explained: - **`expect(selector).toHaveText(text)`**: Asserts the element matching `selector` has exactly this text content - **`expect(selector).toHaveCount(n)`**: Asserts exactly `n` elements match `selector` (use `0` to assert something is absent)

Next, the interaction tests — clicking the increment, decrement, and reset buttons and checking that the `min`/`max` boundaries are respected. These are added inside the same `describe("Counter", ...)` block shown above, and use `click()` from `@odoo/hoot-dom`, which simulates a real user click and waits for OWL to finish re-rendering before resolving:

```
    test("increment functionality", async () => {
        await mountWithCleanup(Counter, {
            props: { initialValue: 0, max: 3 },
        });

        // Test normal increment
        await click(".increment-btn");
        expect(".counter-value").toHaveText("1");

        await click(".increment-btn");
        expect(".counter-value").toHaveText("2");

        await click(".increment-btn");
        expect(".counter-value").toHaveText("3");

        // Test that button becomes disabled at max
        expect(".increment-btn").toHaveProperty("disabled", true);

        // Clicking a disabled button does nothing
        await click(".increment-btn");
        expect(".counter-value").toHaveText("3");
    });

    test("decrement functionality", async () => {
        await mountWithCleanup(Counter, {
            props: { initialValue: 3, min: 1 },
        });

        // Test normal decrement
        await click(".decrement-btn");
```

```
        expect(".counter-value").toHaveText("2");

        await click(".decrement-btn");
        expect(".counter-value").toHaveText("1");

        // Test that button becomes disabled at min
        expect(".decrement-btn").toHaveProperty("disabled", true);

        // Clicking a disabled button does nothing
        await click(".decrement-btn");
        expect(".counter-value").toHaveText("1");
    });

    test("reset functionality", async () => {
        await mountWithCleanup(Counter, {
            props: { initialValue: 5, min: 0, max: 10 },
        });

        // Change the value
        await click(".increment-btn");
        await click(".increment-btn");
        expect(".counter-value").toHaveText("7");

        // Test reset
        await click(".reset-btn");
        expect(".counter-value").toHaveText("5");
    });
```

Remember to add `import { click } from "@odoo/hoot-dom";` alongside the `@odoo/hoot` import at the top of the file.

Finally, the callback and edge-case tests — verifying that `onValueChange` fires with the right values, and that unusual prop combinations (like `min === max`) don't crash the component. Same `describe("Counter", ...)` block as above, closed at the end. Notice each scenario gets its own `test()` — following the "keep tests independent" practice covered later in this chapter, instead of cramming multiple mounts into one test:

```
    test("onValueChange callback", async () => {
        const valueChanges = [];
        const onValueChange = (newValue) => {
            valueChanges.push(newValue);
        };

        await mountWithCleanup(Counter, {
            props: {
                initialValue: 2,
                onValueChange,
            },
        });

        // Test increment callback
        await click(".increment-btn");
        // Test decrement callback
        await click(".decrement-btn");
        // Test reset callback
        await click(".reset-btn");
```

```
        // increment to 3, decrement to 2, reset to 2
        expect(valueChanges).toEqual([3, 2, 2]);
    });

    test("increment and decrement are both disabled when min === max", async () => {
        await mountWithCleanup(Counter, {
            props: { initialValue: 5, min: 5, max: 5 },
        });

        expect(".increment-btn").toHaveProperty("disabled", true);
        expect(".decrement-btn").toHaveProperty("disabled", true);
    });

    test("initial value outside bounds is preserved", async () => {
        // Note: In a real implementation, you might want to clamp the initial value.
        // This test documents current behavior.
        await mountWithCleanup(Counter, {
            props: { initialValue: 100, min: 1, max: 10 },
        });

        expect(".counter-value").toHaveText("100");
    });
});
```

Key matchers explained: - `expect(value).toEqual(other)`: Asserts deep equality between two values (arrays, objects) — use `.toBe()` instead for primitives like strings, numbers, and booleans - `expect(selector).toHaveProperty(name, value)`: Asserts a DOM property (like `disabled`) on the matched element equals `value`

Testing Components with Services

Many real-world components depend on Odoo services. Here's how to test them.

Component Using Services

This `TaskManager` component loads tasks with the `orm` service on mount, and shows feedback with the `notification` service. Let's look at the data-loading half first:

```
/** @odoo-module **/

import { Component, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class TaskManager extends Component {
    static template = xml`
        <div class="task-manager">
            <h3>My Tasks</h3>
            <div class="task-input">
                <input type="text"
                       t-model="state.newTaskText"
                       placeholder="Add a new task..."
                       t-on-keydown="onKeydown"/>
                <button class="btn btn-primary"
```

```
                    t-on-click="addTask"
                    t-att-disabled="!state.newTaskText">
                Add Task
            </button>
        </div>

        <div class="task-list" t-if="state.loading">
            <p>Loading tasks...</p>
        </div>

        <div class="task-list" t-else="">
            <div t-foreach="state.tasks" t-as="task" t-key="task.id"
                 class="task-item"
                 t-att-class="{ 'completed': task.completed }">
                <input type="checkbox"
                       t-att-checked="task.completed"
                       t-on-change="(ev) => this.toggleTask(task.id)"/>
                <span class="task-text" t-esc="task.text"/>
                <button class="btn btn-sm btn-danger delete-btn"
                        t-on-click="() => this.deleteTask(task.id)">
                    Delete
                </button>
            </div>
        </div>
    </div>
`;

setup() {
    this.orm = useService("orm");
    this.notification = useService("notification");

    this.state = useState({
        tasks: [],
        newTaskText: "",
        loading: true,
    });

    this.loadTasks();
}

async loadTasks() {
    this.state.loading = true;
    try {
        const tasks = await this.orm.searchRead(
            "project.task",
            [["user_id", "=", this.orm.user.userId]],
            ["id", "name", "stage_id"]
        );

        this.state.tasks = tasks.map(task => ({
            id: task.id,
            text: task.name,
            completed: task.stage_id[1] === "Done"
        }));
    } catch (error) {
```

```
            this.notification.add("Failed to load tasks", { type: "danger" });
        } finally {
            this.state.loading = false;
        }
    }
```

And the rest of the class — adding, toggling, and deleting tasks, plus the Enter-key shortcut. This is still the same `TaskManager` class, continued:

```
    async addTask() {
        if (!this.state.newTaskText.trim()) return;

        try {
            const [taskId] = await this.orm.create("project.task", {
                name: this.state.newTaskText,
                user_id: this.orm.user.userId,
            });

            this.state.tasks.push({
                id: taskId,
                text: this.state.newTaskText,
                completed: false
            });

            this.state.newTaskText = "";
            this.notification.add("Task added successfully", { type: "success" });
        } catch (error) {
            this.notification.add("Failed to add task", { type: "danger" });
        }
    }

    async toggleTask(taskId) {
        const task = this.state.tasks.find(t => t.id === taskId);
        if (!task) return;

        try {
            // In a real implementation, you'd update the stage_id
            task.completed = !task.completed;
            this.notification.add(
                task.completed ? "Task completed!" : "Task reopened",
                { type: "info" }
            );
        } catch (error) {
            // Revert on error
            task.completed = !task.completed;
            this.notification.add("Failed to update task", { type: "danger" });
        }
    }

    async deleteTask(taskId) {
        try {
            await this.orm.unlink("project.task", [taskId]);
            this.state.tasks = this.state.tasks.filter(t => t.id !== taskId);
            this.notification.add("Task deleted", { type: "info" });
        } catch (error) {
            this.notification.add("Failed to delete task", { type: "danger" });
```

```
        }
    }

    onKeydown(event) {
        if (event.key === "Enter") {
            this.addTask();
        }
    }
}
```

Testing with Mock Services

To test `TaskManager` without a real Odoo server, we mock the RPC layer instead of hitting the database, using the `onRpc()` helper to intercept specific ORM calls. First, the tests that cover loading data and adding a new task:

```
/** @odoo-module **/

import { describe, test, expect } from "@odoo/hoot";
import { click, edit, queryAllTexts } from "@odoo/hoot-dom";
import { mountWithCleanup, onRpc } from "@web/../tests/web_test_helpers";

import { TaskManager } from "../task_manager";

describe("TaskManager", () => {
    test("loads and displays tasks", async () => {
        // Mock the ORM's search_read for the project.task model
        const mockTasks = [
            { id: 1, name: "Buy groceries", stage_id: [1, "To Do"] },
            { id: 2, name: "Walk the dog", stage_id: [2, "Done"] },
            { id: 3, name: "Write tests", stage_id: [1, "To Do"] },
        ];
        onRpc("project.task", "search_read", () => mockTasks);

        await mountWithCleanup(TaskManager);

        // Check that loading state is gone
        expect(".task-list p").toHaveCount(0);

        // Check that tasks are displayed
        expect(".task-item").toHaveCount(3);
        expect(queryAllTexts(".task-text")).toEqual([
            "Buy groceries",
            "Walk the dog",
            "Write tests",
        ]);

        // Check completed state (second task has stage_id "Done")
        expect(".task-item:nth-child(2)").toHaveClass("completed");
    });

    test("adds a new task", async () => {
        let createdTask = null;
        onRpc("project.task", "search_read", () => []); // Start with empty task list
        onRpc("project.task", "create", ({ args }) => {
            createdTask = args[0]; // Capture created task data
```

```
            return 123; // Mock id
        });

        await mountWithCleanup(TaskManager);

        // Type in new task and click add
        await edit(".task-input input", "New test task");
        await click(".btn-primary");

        // Verify ORM call was made
        expect(createdTask).not.toBe(null);
        expect(createdTask.name).toBe("New test task");

        // Verify task appears in UI
        expect(".task-item").toHaveCount(1);
        expect(queryAllTexts(".task-text")).toEqual(["New test task"]);

        // Verify input is cleared
        expect(".task-input input").toHaveValue("");
    });
});
```

Key helpers explained: - **`onRpc(model, method, callback)`**: Intercepts a specific ORM call instead of hitting the real server; the callback receives `{ args, kwargs }` (the arguments the component passed to `orm.create`/`orm.write`/etc.) and its return value becomes the RPC result - **`edit(selector, value)`**: Types `value` into the matched input and fires the events a real user's typing would (from `@odoo/hoot-dom`) - **`queryAllTexts(selector)`**: Returns the trimmed text content of every element matching `selector`, as an array — convenient for comparing a whole list at once

The remaining two tests check error handling and the keyboard shortcut. They're added inside the same `describe("TaskManager", ...)` block shown above, closed at the end:

```
    test("handles RPC errors gracefully", async () => {
        const notificationMessages = [];
        onRpc("project.task", "search_read", () => {
            throw new Error("Network error");
        });
        mockService("notification", {
            add: (message, options) => {
                notificationMessages.push({ message, options });
            },
        });

        await mountWithCleanup(TaskManager);

        // Check that error notification was shown
        expect(notificationMessages.length).toBe(1);
        expect(notificationMessages[0].message).toBe("Failed to load tasks");
        expect(notificationMessages[0].options.type).toBe("danger");

        // Check that loading state is cleared even on error
        expect(".task-list p").toHaveCount(0);
    });

    test("keyboard shortcuts work", async () => {
```

```
        let taskCreated = false;
        onRpc("project.task", "search_read", () => []);
        onRpc("project.task", "create", () => {
            taskCreated = true;
            return 456;
        });

        await mountWithCleanup(TaskManager);

        // Type task text and press Enter
        await edit(".task-input input", "Task via Enter key");
        await press("Enter");

        expect(taskCreated).toBe(true);
    });
});
```

Add `import { mockService } from "@web/../tests/web_test_helpers";` and `import { press } from "@odoo/hoot-dom";` alongside the other imports at the top of the file. **`mockService()`** replaces a real Odoo service with a fake implementation for the duration of the test — here it swaps out `notification` so we can capture the messages it receives instead of showing real toasts. **`press(key)`** simulates pressing a keyboard key on the currently focused element.

Effective Debugging Techniques

While tests catch many issues, you'll still need to debug problems. Here are professional debugging strategies:

1. Using Browser DevTools Effectively

Setting Strategic Breakpoints:

```
// In your component
increment() {
    debugger; // Execution will pause here
    if (this.state.count < this.props.max) {
        this.state.count++;
        this._notifyChange();
    }
}
```

Conditional Breakpoints: Right-click on a line number in DevTools Sources tab and select "Add conditional breakpoint":

```
this.state.count > 5  // Only pause when count exceeds 5
```

Watching Variables: In the DevTools debugger, add variables to the "Watch" panel to see how they change as you step through code.

2. Strategic Console Logging

Component Lifecycle Logging:

```
setup() {
    console.log("Counter setup", this.props);
```

```
    this.state = useState({ count: this.props.initialValue });
}

increment() {
    console.log("Before increment:", this.state.count);
    if (this.state.count < this.props.max) {
        this.state.count++;
        console.log("After increment:", this.state.count);
        this._notifyChange();
    } else {
        console.log("Increment blocked - at max:", this.props.max);
    }
}
```

State Change Tracking:

```
setup() {
    this.state = useState({ count: this.props.initialValue });

    // Log all state changes
    const originalState = this.state;
    Object.defineProperty(this, 'state', {
        get: () => originalState,
        set: (newState) => {
            console.log("State changing from", originalState, "to", newState);
            originalState = newState;
        }
    });
}
```

3. Component Inspector Tools

OWL DevTools: Install the OWL DevTools browser extension to: - See the component tree structure - Inspect component props and state - Track component updates and re-renders

Accessing Components from Console: OWL does not expose a supported, public way to grab a mounted component instance from a plain DOM element — that's exactly what the OWL DevTools extension above is for. If you need programmatic access while developing, expose the instance yourself, deliberately, as a dev-only escape hatch:

```
// In the component, during development only:
setup() {
    onMounted(() => {
        // Intentional debug hook - never rely on this in production code.
        this.el.__debugComponent = this;
    });
}

// Then, in the browser console:
const counterComponent = document.querySelector('.counter-widget').__debugComponent;
console.log(counterComponent.state);
console.log(counterComponent.props);
```

4. Debugging Common Issues

Props Not Updating:

```
// Check if parent is actually passing new props
setup() {
    onWillUpdateProps((nextProps) => {
        console.log("Current props:", this.props);
        console.log("Next props:", nextProps);
        console.log("Props changed:", JSON.stringify(this.props) !==
            ↪ JSON.stringify(nextProps));
    });
}
```

State Not Triggering Re-render:

```
// Ensure you're using useState correctly
setup() {
    // Good - reactive state
    this.state = useState({ count: 0 });

    // Bad - not reactive
    this.state = { count: 0 };
}
```

Event Handlers Not Working:

```
// Check event binding in template
static template = xml`
    <!-- Good - proper binding -->
    <button t-on-click="increment">+</button>

    <!-- Bad - calling function immediately -->
    <button t-on-click="increment()">+</button>

    <!-- Good - arrow function for parameters -->
    <button t-on-click="() => this.incrementBy(5)">+5</button>
`;
```

Testing Best Practices

1. Write Descriptive Test Names

```
// Bad - vague test names
test("test counter", async () => { ... });
test("button test", async () => { ... });

// Good - descriptive test names
test("counter displays initial value from props", async () => { ... });
test("increment button becomes disabled at maximum value", async () => { ... });
```

2. Test Behavior, Not Implementation

```
// Bad - testing implementation details
test("state.count increases by 1", async () => {
    const counter = await mountWithCleanup(Counter);
    counter.state.count = 5; // Direct state manipulation
    expect(counter.state.count).toBe(5);
});
```

```
// Good - testing user-visible behavior
test("clicking increment button increases displayed value", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
    expect(".counter-value").toHaveText("1");
});
```

3. Use Arrange-Act-Assert Pattern

```
test("reset button restores initial value", async () => {
    // Arrange
    await mountWithCleanup(Counter, { props: { initialValue: 5 } });
    await click(".increment-btn"); // Change value

    // Act
    await click(".reset-btn");

    // Assert
    expect(".counter-value").toHaveText("5");
});
```

4. Keep Tests Independent

```
// Bad - tests depend on each other
let sharedCounter;

test("setup counter", async () => {
    sharedCounter = await mountWithCleanup(Counter);
    expect(sharedCounter).toBeTruthy();
});

test("increment counter", async () => {
    await click(".increment-btn"); // Depends on the previous test still being mounted
    // ...
});

// Good - each test is independent
test("counter can be incremented", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
    expect(".counter-value").toHaveText("1");
});
```

5. Test Edge Cases

```
// One test per scenario keeps failures easy to pinpoint
test("handles extreme values", async () => {
    await mountWithCleanup(Counter, {
        props: { initialValue: Number.MAX_SAFE_INTEGER, max: Number.MAX_SAFE_INTEGER },
    });
    expect(".counter-value").toHaveCount(1);
});

test("handles negative ranges", async () => {
```

```
    await mountWithCleanup(Counter, {
        props: { initialValue: -5, min: -10, max: 0 },
    });
    expect(".counter-value").toHaveText("-5");
});

test("handles min equal to max", async () => {
    await mountWithCleanup(Counter, {
        props: { initialValue: 5, min: 5, max: 5 },
    });
    expect(".increment-btn").toHaveProperty("disabled", true);
});
```

Common Pitfalls and Solutions

Pitfall 1: Forgetting to `await` Async Interactions

```
// Wrong - the assertion runs before the DOM has re-rendered
test("flaky", async () => {
    await mountWithCleanup(Counter);
    click(".increment-btn"); // not awaited!
    expect(".counter-value").toHaveText("1");
});

// Correct - await the click; hoot's click() only resolves after OWL re-renders
test("reliable", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
    expect(".counter-value").toHaveText("1");
});
```

Pitfall 2: Leftover Components Between Tests

The old QUnit-based framework required calling `counter.destroy()` yourself at the end of every test, and forgetting it left a mounted component behind for the next test to trip over. `mountWithCleanup()` fixes this footgun by design — it always unmounts the component automatically once the test ends. The pitfall now is reaching for OWL's plain `mount()` out of habit instead:

```
// Wrong - plain mount() from @odoo/owl is never cleaned up automatically
import { mount } from "@odoo/owl";

test("increment works", async () => {
    await mount(Counter, document.body);
    await click(".increment-btn");
    // this component is still mounted when the next test starts
});

// Correct - mountWithCleanup() always unmounts after the test, pass or fail
test("increment works", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
});
```

Pitfall 3: Reaching for `console.log` Instead of the Component Inspector

Sprinkling `console.log` everywhere works, but it's slow to iterate and easy to forget to remove afterward. For anything beyond a quick one-off check, install the OWL DevTools extension (see above) and inspect `state`/`props` directly on the live component tree — it's faster and never accidentally ships to production.

Pitfall 4: Debugging Without Reading the Full Stack Trace

It's tempting to jump straight to a `debugger` statement or a `console.log` when something breaks. Read the full stack trace in the console first — it usually names the exact component, method, and line, which tells you where to add a breakpoint instead of guessing.

Exercises

Exercise 1: Test a New Behavior

Add a `step` prop to `Counter` (defaulting to `1`) that controls how much `increment`/`decrement` change the count by. Write a test that mounts the component with `props: { step: 5 }` and verifies that one click on the increment button moves the value from `0` to `5`.

Exercise 2: Mock a Service Failure

Using `TaskManager`, write a test where `onRpc("project.task", "create", ...)` throws (not just `search_read`) and verify that the component shows a "Failed to add task" notification instead of adding the task to the list.

Exercise 3: Debug a Broken Handler

Take `onKeydown` and intentionally break it by comparing `event.key === "enter"` (lowercase) instead of `"Enter"`. Use a breakpoint or console logging to find the bug before looking at the fix, then correct it.

You now have the fundamentals: why testing matters, how to set up your environment, how to write and run tests for components with and without services, how to debug issues with browser tools, and the best practices that keep a test suite trustworthy. In Part 2, we'll build on this foundation with advanced testing patterns, Test-Driven Development, performance and integration testing, continuous integration, and the techniques professional teams use to debug issues that only show up in production.

TL;DR: Write `@odoo/hoot` tests with `mountWithCleanup`, mock the services a component depends on, and use the browser's OWL DevTools instead of scattering `console.log` around — that combination is what makes changing code later safe instead of scary.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch16_1_ex1`. Installation instructions are in the repository README.

What's Next?

You can now write and trust a basic test suite. Chapter 16 Part 2 goes further — advanced testing patterns, Test-Driven Development, and the debugging techniques you need once a bug only shows up in production.

Chapter 16 Part 2: Advanced Testing, TDD, and Production Debugging

> **Why this chapter?** Real projects eventually hit the issues that don't show up in a quick manual check — a memory leak that only appears after weeks of runtime, a flaky test that erodes trust in the whole suite, a bug that only reproduces under production load. This is the tooling for that stage of a project's life.
>
> **What is Odoo trying to solve with this?** Giving teams a way to test integration across services, wire up CI so regressions are caught before deploy, and debug production issues safely — instead of ad hoc `console.log` sessions on a client's live server.
>
> **Real-world application:** Diagnosing a memory leak a client reports after a month of daily use, or setting up CI so every merge is tested automatically before it reaches their instance.

In Part 1, we covered the fundamentals of testing and debugging OWL components: writing your first tests, testing components that depend on services, using browser DevTools effectively, and following testing best practices. Now we'll go further: advanced testing patterns, Test-Driven Development, performance and integration testing, continuous integration, pitfalls that trip up even experienced developers, and how to debug issues once your code is running in production.

Advanced Testing Patterns

1. Testing Component Communication

When testing parent-child component communication:

```
/** @odoo-module **/

import { Component, useState, xml } from "@odoo/owl";
import { describe, test, expect } from "@odoo/hoot";
import { click } from "@odoo/hoot-dom";
import { mountWithCleanup } from "@web/../tests/web_test_helpers";
import { Counter } from "../counter";

describe("Counter integration", () => {
    test("parent-child communication", async () => {
        const receivedEvents = [];

        // Create a parent component that uses our tested component
        class TestParent extends Component {
            static template = xml`
                <div>
```

```
                    <Counter initialValue="5" onValueChange.bind="onCounterChange"/>
                    <p class="parent-display">Parent sees: <t
    ↪ t-esc="state.lastValue"/></p>
                </div>
            `;
            static components = { Counter };

            setup() {
                this.state = useState({ lastValue: 5 });
            }

            onCounterChange(newValue) {
                receivedEvents.push(newValue);
                this.state.lastValue = newValue;
            }
        }

        await mountWithCleanup(TestParent);

        // Interact with the child component
        await click(".increment-btn");

        // Verify parent received the event
        expect(receivedEvents).toEqual([6]);
        expect(".parent-display").toHaveText("Parent sees: 6");
    });
});
```

2. Testing with Props Changes

Test how components react to prop changes. Since props always flow down from a parent in OWL, the cleanest way to test this is to wrap the component in a small test parent and change what it passes down — the same pattern used above:

```
describe("Counter integration", () => {
    test("reacts to prop changes", async () => {
        class TestParent extends Component {
            static template = xml`<Counter max="state.max"/>`;
            static components = { Counter };
            setup() {
                this.state = useState({ max: 5 });
            }
        }

        await mountWithCleanup(TestParent);

        // Increment to max
        for (let i = 0; i < 5; i++) {
            await click(".increment-btn");
        }
        expect(".increment-btn").toHaveProperty("disabled", true);

        // Raise the max on the parent's state; OWL re-renders the child with new props
        // Note: this snippet only illustrates the idea - TestParent's state isn't
        // reachable from outside, so in a real test you'd expose a way to update it
        // (e.g. a button in TestParent's template) and click that instead.
```

```
    });
});
```

3. Testing Async Operations

For components with async operations:

```
describe("TaskManager integration", () => {
    test("handles async operations", async () => {
        let resolvePromise;
        const asyncPromise = new Promise((resolve) => {
            resolvePromise = resolve;
        });
        onRpc("project.task", "search_read", () => asyncPromise);

        await mountWithCleanup(TaskManager);

        // Check loading state
        expect(".task-list p").toHaveCount(1);

        // Resolve the async operation
        resolvePromise([]);
        await animationFrame(); // let OWL flush the re-render after the promise
            ↪ resolves

        // Check that loading is gone
        expect(".task-list p").toHaveCount(0);
    });
});
```

Add `import { animationFrame } from "@odoo/hoot-dom";` at the top — it's the general-purpose "wait a tick for the DOM to catch up" helper in hoot, used whenever you resolve a promise manually instead of going through `click()` or `edit()` (which already wait internally).

Test-Driven Development (TDD) with OWL

Test-Driven Development is a powerful approach where you write tests before implementing functionality:

1. Red-Green-Refactor Cycle

Red: Write a failing test

```
test("counter displays correctly", async () => {
    await mountWithCleanup(Counter);
    expect(".counter-value").toHaveText("0");
});
```

Green: Write minimal code to make it pass

```
export class Counter extends Component {
    static template = xml`
        <div>
            <span class="counter-value">0</span>
        </div>
```

```
    `;
}
```

Refactor: Improve the code while keeping tests green

```
export class Counter extends Component {
    static template = xml`
        <div class="counter-widget">
            <span class="counter-value" t-esc="state.count"/>
        </div>
    `;

    setup() {
        this.state = useState({ count: 0 });
    }
}
```

2. Benefits of TDD

- **Clear Requirements:** Tests serve as specifications
- **Better Design:** Writing tests first leads to more testable, modular code
- **Regression Protection:** Comprehensive test suite catches future breakages
- **Confidence:** Green tests mean working software

Performance Testing and Debugging

1. Measuring Component Performance

```
test("counter performance with many updates", async () => {
    await mountWithCleanup(Counter, { props: { max: 1000 } });

    const startTime = performance.now();

    // Simulate rapid clicking
    for (let i = 0; i < 100; i++) {
        await click(".increment-btn");
    }

    const duration = performance.now() - startTime;

    expect(duration).toBeLessThan(1000);
    expect(".counter-value").toHaveText("100");
});
```

2. Memory Leak Detection

This one deliberately manages mount/destroy by hand instead of `mountWithCleanup()`, since the point of the test is to verify that manual cleanup doesn't leave anything behind — so it imports `mount` straight from `@odoo/owl`:

```
import { mount } from "@odoo/owl";

test("counter cleans up properly", async () => {
    const initialComponents = document.querySelectorAll(".counter-widget").length;
```

```
    const container = document.createElement("div");

    // Create and destroy multiple components
    for (let i = 0; i < 10; i++) {
        const counter = await mount(Counter, container);
        counter.destroy();
        container.innerHTML = ""; // Clear the container
    }

    // Force garbage collection if available
    if (window.gc) {
        window.gc();
    }

    const finalComponents = document.querySelectorAll(".counter-widget").length;
    expect(finalComponents).toBe(initialComponents);
});
```

Integration Testing

Testing Components in Real Scenarios

Sometimes you need to test how components work together in realistic conditions:

```
test("counter works in form context", async () => {
    // Create a realistic parent component
    class TestForm extends Component {
        static template = xml`
            <form t-on-submit.prevent="onSubmit">
                <div class="form-group">
                    <label>Quantity:</label>
                    <Counter initialValue="1"
                             min="1"
                             max="10"
                             onValueChange.bind="onQuantityChange"/>
                </div>
                <div class="form-group">
                    <span>Total: $<t t-esc="state.total"/></span>
                </div>
                <button type="submit" class="btn btn-primary">Order</button>
            </form>
        `;

        static components = { Counter };

        setup() {
            this.state = useState({
                quantity: 1,
                price: 10,
                total: 10,
            });
        }

        onQuantityChange(newQuantity) {
```

```
            this.state.quantity = newQuantity;
            this.state.total = this.state.quantity * this.state.price;
        }

        onSubmit() {
            // Form submission logic
        }
    }

    await mountWithCleanup(TestForm);

    // Test integrated behavior
    await click(".increment-btn");

    expect(".counter-value").toHaveText("2");
    expect(".form-group:nth-child(2)").toHaveText("Total: $20");
});
```

Continuous Integration and Testing

1. Running Tests in CI/CD

Create a test script for your module:

File: `my_module/tests/test_js.py`

```
import odoo.tests


@odoo.tests.tagged('post_install', '-at_install')
class TestJavaScript(odoo.tests.HttpCase):

    def test_js_modules(self):
        """Test that JavaScript modules load and hoot unit tests pass"""
        self.browser_js(
            "/web/tests?module=my_module&failfast",
            code="",
            timeout=60,
        )
```

2. Test Coverage Reporting

Coverage for hoot tests is enabled from the test runner itself (e.g. a `coverage=1` URL parameter on `/web/tests`), not from inside the test file. Once a coverage-enabled run finishes, the standard Istanbul coverage object is available on the page for inspection or export:

```
if (window.__coverage__) {
    console.log("Coverage data:", window.__coverage__);
}
```

Common Testing Pitfalls and Solutions

1. Flaky Tests

Problem: Tests that sometimes pass and sometimes fail

Solution: Ensure proper async handling

```
// Bad - not waiting for async operations
test("flaky test", async () => {
    await mountWithCleanup(Counter);
    click(".increment-btn"); // Not awaited!
    expect(".counter-value").toHaveText("1");
});

// Good - properly awaiting async operations
test("reliable test", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn"); // hoot's click() waits for the re-render itself
    expect(".counter-value").toHaveText("1");
});
```

2. Testing Private Methods

Problem: Wanting to test internal component methods

Solution: Test through public interface

```
// Bad - testing private methods
test("_calculateTotal works", async () => {
    const component = await mountWithCleanup(MyComponent);
    const result = component._calculateTotal(5, 10);
    expect(result).toBe(50);
});

// Good - testing through public behavior
test("displays correct total when quantity changes", async () => {
    await mountWithCleanup(MyComponent);
    await edit(".quantity-input", "5");
    expect(".total").toHaveText("Total: 50");
});
```

3. Over-mocking

Problem: Mocking too many dependencies

Solution: Mock only what's necessary

```
// Bad - over-mocking every service the component happens to touch
mockService("orm", mockOrm);
mockService("notification", mockNotification);
mockService("user", mockUser);
mockService("company", mockCompany);
// ... mocking everything makes the test brittle and hides real integration bugs

// Good - mock only the one RPC call this test actually cares about
onRpc("my.model", "my_specific_method", () => mockData);
// every other request goes through normal handling
```

Debugging Production Issues

1. Error Boundaries and Logging

```
export class ErrorBoundary extends Component {
    static template = xml`
        <div>
            <t t-if="state.hasError">
                <div class="alert alert-danger">
                    <h4>Something went wrong</h4>
                    <p t-esc="state.error.message"/>
                    <button t-on-click="retry">Try Again</button>
                </div>
            </t>
            <t t-else="">
                <t t-slot="default"/>
            </t>
        </div>
    `;

    setup() {
        this.state = useState({
            hasError: false,
            error: null
        });
    }

    catchError(error) {
        console.error("Component error caught:", error);

        // Send to error reporting service
        if (window.errorReporting) {
            window.errorReporting.captureException(error);
        }

        this.state.hasError = true;
        this.state.error = error;
    }

    retry() {
        this.state.hasError = false;
        this.state.error = null;
    }
}
```

2. Feature Flags for Safe Deployments

```
export class Counter extends Component {
    setup() {
        this.featureFlags = useService("feature_flags");
        this.state = useState({
            count: this.props.initialValue || 0
        });
```

```
    }

    increment() {
        if (this.featureFlags.isEnabled("advanced_counter")) {
            // New implementation
            this.state.count += this.props.step || 1;
        } else {
            // Safe fallback
            this.state.count++;
        }
    }
}
```

Exercises

Exercise 1: Write a TDD Cycle

Following the Red-Green-Refactor cycle, write a failing test for a new `doubled` getter on `Counter` that returns `state.count * 2`, implement the minimal code to make it pass, then refactor if you see room for improvement.

Exercise 2: Mock a Service and Assert on Its Arguments

Write a test for `TaskManager.deleteTask` that mocks the `orm` service directly with `mockService("orm", ...)` (instead of going through `onRpc`) and asserts that `orm.unlink` was called with `["project.task", [taskId]]`.

Exercise 3: Sketch a CI Check

Sketch (in comments — no need to run it) what you would add to `test_js.py` so CI fails the build if any hoot test in your module fails. In 2-3 sentences, explain why running JS tests in CI matters even when they already pass on your machine.

Summary: Building Robust OWL Applications

Testing and debugging are not afterthoughts—they're integral parts of building professional OWL applications. Here's your roadmap to excellence:

1. Testing Strategy

- **Unit Tests**: Test individual components in isolation
- **Integration Tests**: Test component interactions
- **End-to-End Tests**: Test complete user workflows
- **Performance Tests**: Ensure acceptable response times
- **Error Handling Tests**: Verify graceful failure modes

2. Debugging Toolkit

- **Browser DevTools**: Your primary debugging interface

- **Strategic Logging**: Capture important state changes
- **Error Boundaries**: Graceful error handling
- **Performance Monitoring**: Track and optimize slow operations

3. Professional Practices

- **Test-Driven Development**: Write tests before implementation
- **Continuous Integration**: Automated test running
- **Code Coverage**: Ensure adequate test coverage
- **Documentation**: Clear, executable specifications

4. Maintenance Mindset

- **Refactor with Confidence**: Good tests enable safe changes
- **Regression Prevention**: Tests catch unexpected breakages
- **Living Documentation**: Tests show how components should work
- **Team Collaboration**: Shared understanding through tests

By mastering these testing and debugging techniques, you transform from someone who writes code that works today into someone who builds robust, maintainable applications that continue working as they evolve. This is the hallmark of professional OWL development.

Remember: every bug you catch in a test is a bug your users will never see. Every debugging technique you master is time saved in future investigations. Invest in these skills, and they'll pay dividends throughout your entire development career.

TL;DR: TDD, service mocking, CI, and production-safe debugging are what separate "it works" from "it keeps working" — invest in them once a component is doing anything a client actually depends on.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch16_2_ex1`. Installation instructions are in the repository README.

What's Next?

Testing and debugging assume you're still writing pure OWL 2.0. Chapter 17 Part 1 covers the reality of most Odoo projects: bridging new OWL components with the legacy widget system still running in production.

Chapter 17 Part 1: The Legacy-OWL Bridge - Understanding and Basic Implementation

Why this chapter? Because almost no real Odoo project is a blank slate — you will spend far more time working inside a codebase that already has years of legacy widgets than building something brand new in pure OWL.

What is Odoo trying to solve with this? Odoo needs a way to let old and new frontend code coexist during the multi-year migration from the legacy widget system to OWL, without forcing a risky, all-at-once rewrite of every client's customizations.

Real-world application: This is exactly the situation that pushed me to write this book: a client upgrade broke every custom widget we had built for them, and the bridge techniques in this chapter are what let you modernize a module piece by piece instead of freezing a project until a full rewrite is done.

Congratulations on reaching the final core chapter! You've mastered modern JavaScript, built sophisticated OWL components, implemented global state management, and learned professional testing practices. You're fully equipped to build cutting-edge applications in Odoo from scratch.

However, the reality of professional Odoo development is more nuanced. Most of your work won't involve green-field projects where you can use pure OWL throughout. Instead, you'll be working with existing Odoo databases filled with years of customizations, third-party modules, and user interfaces built with Odoo's older JavaScript framework—the **legacy widget system**.

This creates a critical challenge: How do you modernize applications incrementally? How do you introduce powerful OWL components into existing legacy interfaces without breaking everything? How do you leverage existing legacy widgets in new OWL applications when rebuilding them would take months?

The answer lies in Odoo's **legacy-OWL bridge**—a sophisticated compatibility layer that enables seamless interoperability between old and new code. This isn't just a technical curiosity; it's an essential tool for any professional Odoo developer working on real-world projects.

Understanding the Legacy Landscape

Before diving into bridge techniques, let's understand what we're bridging between.

The Legacy Widget System

Odoo's legacy JavaScript framework, used from version 8 through parts of version 16, was built around:

Widget-Based Architecture: Components were classes extending `Widget`

```
const MyWidget = Widget.extend({
    template: 'my_module.MyTemplate',

    start: function() {
        // Initialization logic
        return this._super.apply(this, arguments);
    },

    destroy: function() {
        // Cleanup logic
        this._super.apply(this, arguments);
    }
});
```

jQuery-Heavy: Direct DOM manipulation and event handling

```
events: {
    'click .my-button': '_onButtonClick',
},

_onButtonClick: function(event) {
    this.$('.result').text('Button clicked!');
}
```

QWeb Templates: Server-side template compilation

```
<t t-name="my_module.MyTemplate">
  <div class="my-widget">
    <button class="my-button">Click me</button>
    <div class="result"></div>
  </div>
</t>
```

Why Bridge Instead of Rewrite?

Practical Reality: Large Odoo installations have hundreds of custom widgets representing years of business logic and user interface refinements.

Risk Management: A complete rewrite introduces significant risk of breaking existing functionality that users depend on daily.

Resource Constraints: Rewriting everything at once would require massive development resources and extended project timelines.

Business Continuity: Users need new features while existing functionality continues to work reliably.

The bridge approach allows for **gradual modernization**—introducing OWL components incrementally while maintaining system stability.

Bridge Architecture Overview

The legacy-OWL bridge works in both directions:

Figure 7: Direction 1 mounts OWL inside the widget; direction 2 wraps the widget so it can be used from OWL.

Direction 1: OWL in Legacy - Mount modern OWL components inside existing legacy widgets - Add new features without rewriting entire screens - Gradually modernize user interfaces

Direction 2: Legacy in OWL - Use existing legacy widgets within new OWL applications - Leverage complex legacy functionality without rebuilding - Maintain compatibility with third-party widgets

---

Scenario 1: Embedding OWL Components in Legacy Widgets

This is the most common modernization pattern. You have an existing legacy form or dashboard, and you want to add a sophisticated new feature built with OWL.

Example: Adding a Modern Chart to a Legacy Dashboard

Let's say you have a legacy sales dashboard widget, and you want to add an interactive revenue chart built with OWL.

1. The Legacy Dashboard Widget

```
odoo.define('sales_dashboard.LegacyDashboard', function (require) {
    "use strict";

    const Widget = require('web.Widget');
    const core = require('web.core');
    const { mount } = require("@odoo/owl");

    // Import our modern OWL component
    const { RevenueChart } = require('sales_dashboard.RevenueChart');

    const LegacyDashboard = Widget.extend({
```

```
    template: 'sales_dashboard.LegacyDashboardTemplate',

    events: {
        'click .refresh-data': '_onRefreshData',
        'change .date-filter': '_onDateFilterChange',
    },

    init: function(parent, options) {
        this._super.apply(this, arguments);
        this.salesData = options.salesData || [];
        this.owlComponents = {}; // Track mounted OWL components
    },

    async start() {
        await this._super(...arguments);

        // Initialize legacy functionality
        this._setupLegacyFeatures();

        // Mount OWL components
        await this._mountOWLComponents();

        // Load initial data
        await this._loadDashboardData();
    },

    _setupLegacyFeatures: function() {
        // Legacy jQuery-based functionality
        this.$('.legacy-counter').each(function(index, element) {
            $(element).data('value', 0);
        });

        // Initialize legacy date picker
        this.$('.date-filter').datepicker({
            format: 'yyyy-mm-dd',
            onSelect: this._onDateFilterChange.bind(this)
        });
    },

    async _mountOWLComponents() {
        try {
            // Mount the revenue chart in its designated container
            const chartTarget = this.el.querySelector('.revenue-chart-container');
            if (chartTarget) {
                this.owlComponents.revenueChart = await mount(RevenueChart,
                    ↪ chartTarget, {
                    props: {
                        initialData: this.salesData,
                        onDataPointClick: this._onChartDataPointClick.bind(this),
                        onDateRangeChange: this._onChartDateRangeChange.bind(this),
                        // Pass legacy widget reference for complex interactions
                        legacyWidget: this,
                    }
                });
            }
```

```
            // Mount additional OWL components as needed
            const metricsTarget = this.el.querySelector('.metrics-widget-container');
            if (metricsTarget) {
                const { SalesMetrics } = require('sales_dashboard.SalesMetrics');
                this.owlComponents.salesMetrics = await mount(SalesMetrics,
                    ↪ metricsTarget, {
                    props: {
                        data: this.salesData,
                        onMetricClick: this._onMetricClick.bind(this)
                    }
                });
            }

        } catch (error) {
            console.error('Failed to mount OWL components:', error);
            // Graceful degradation - show error message or fallback UI
            this._showComponentError(error);
        }
    },
```

Once the OWL components are mounted, the widget loads its data and pushes it into both the legacy counters and the OWL components' props:

```
    async _loadDashboardData() {
        try {
            const data = await this._rpc({
                model: 'sale.order',
                method: 'get_dashboard_data',
                args: [this._getDateRange()],
            });

            // Update legacy widgets
            this._updateLegacyCounters(data.counters);

            // Update OWL components
            await this._updateOWLComponents(data);

        } catch (error) {
            console.error('Failed to load dashboard data:', error);
            this._showError('Failed to load dashboard data');
        }
    },

    async _updateOWLComponents(data) {
        // Update OWL component props
        if (this.owlComponents.revenueChart) {
            this.owlComponents.revenueChart.props.data = data.revenue;
            // Trigger re-render by updating props
            await this.owlComponents.revenueChart.render();
        }

        if (this.owlComponents.salesMetrics) {
            this.owlComponents.salesMetrics.props.data = data.metrics;
            await this.owlComponents.salesMetrics.render();
        }
```

```
    },

    _updateLegacyCounters: function(counters) {
        // Legacy jQuery-based updates
        Object.keys(counters).forEach(key => {
            const $counter = this.$(`.counter[data-metric="${key}"]`);
            this._animateCounter($counter, counters[key]);
        });
    },

    _animateCounter: function($element, targetValue) {
        const currentValue = $element.data('value') || 0;
        $({ value: currentValue }).animate({ value: targetValue }, {
            duration: 1000,
            step: function() {
                $element.text(Math.floor(this.value));
            },
            complete: function() {
                $element.data('value', targetValue);
            }
        });
    },

    // Event handlers for legacy functionality
    _onRefreshData: function(event) {
        event.preventDefault();
        this._loadDashboardData();
    },

    _onDateFilterChange: function(event) {
        const newDateRange = this._getDateRange();
        this._loadDashboardData();
    },

    // Event handlers for OWL component interactions
    _onChartDataPointClick: function(dataPoint) {
        // Handle clicks from OWL chart - maybe show legacy drill-down dialog
        this._showDrillDownDialog(dataPoint);
    },

    _onChartDateRangeChange: function(dateRange) {
        // Update legacy date filters when OWL chart changes date range
        this.$('.date-filter').datepicker('setDate', dateRange.start);
        this._loadDashboardData();
    },

    _onMetricClick: function(metric) {
        // Navigate to legacy list view
        this.do_action({
            type: 'ir.actions.act_window',
            res_model: 'sale.order',
            views: [[false, 'list']],
            domain: metric.domain,
            context: metric.context,
        });
```

```
        },

        // Utility methods
        _getDateRange: function() {
            return {
                start: this.$('.date-start').val(),
                end: this.$('.date-end').val(),
            };
        },

        _showDrillDownDialog: function(dataPoint) {
            // Legacy dialog implementation
            const dialog = new Dialog(this, {
                title: `Details for ${dataPoint.label}`,
                size: 'medium',
                $content: $(`<div>Revenue: ${dataPoint.value}</div>`),
            });
            dialog.open();
        },

        _showComponentError: function(error) {
            this.$('.owl-error-container').show().find('.error-message').text(
                'Failed to load interactive components. Please refresh the page.'
            );
        },

        _showError: function(message) {
            // Legacy error display
            this.$('.error-container').show().find('.error-text').text(message);
        },
```

The last piece is the most important one for this chapter: cleaning up the mounted OWL components. Because the legacy `Widget` class has no automatic lifecycle hook for this, you must do it by hand inside `destroy`, or every mounted OWL component leaks memory every time the legacy widget is closed and reopened.

```
        // Critical: Cleanup OWL components to prevent memory leaks
        destroy: function() {
            // Unmount all OWL components
            Object.values(this.owlComponents).forEach(component => {
                if (component && component.destroy) {
                    component.destroy();
                }
            });
            this.owlComponents = {};

            // Call parent destroy
            this._super.apply(this, arguments);
        },
    });

    return LegacyDashboard;
});
```

2. The Legacy Template

```
<t t-name="sales_dashboard.LegacyDashboardTemplate">
```

```
<div class="legacy-sales-dashboard">
  <!-- Legacy header with controls -->
  <div class="dashboard-header">
    <h2>Sales Dashboard (Legacy)</h2>
    <div class="dashboard-controls">
      <input type="text" class="date-filter date-start" placeholder="Start Date"/>
      <input type="text" class="date-filter date-end" placeholder="End Date"/>
      <button class="btn btn-primary refresh-data">Refresh</button>
    </div>
  </div>

  <!-- Legacy counters -->
  <div class="legacy-counters row">
    <div class="col-md-3">
      <div class="counter-widget">
        <h4>Total Sales</h4>
        <div class="counter" data-metric="total_sales">0</div>
      </div>
    </div>
    <div class="col-md-3">
      <div class="counter-widget">
        <h4>New Customers</h4>
        <div class="counter" data-metric="new_customers">0</div>
      </div>
    </div>
    <div class="col-md-3">
      <div class="counter-widget">
        <h4>Active Deals</h4>
        <div class="counter" data-metric="active_deals">0</div>
      </div>
    </div>
    <div class="col-md-3">
      <div class="counter-widget">
        <h4>Conversion Rate</h4>
        <div class="counter" data-metric="conversion_rate">0%</div>
      </div>
    </div>
  </div>

  <!-- Modern OWL components mounted here -->
  <div class="modern-components">
    <div class="row">
      <div class="col-md-8">
        <div class="card">
          <div class="card-header">
            <h5>Revenue Trends (Modern OWL Component)</h5>
          </div>
          <div class="card-body">
            <!-- OWL component will be mounted here -->
            <div class="revenue-chart-container"></div>
            <!-- Fallback for when OWL component fails -->
            <div class="owl-error-container" style="display: none;">
              <div class="alert alert-warning">
                <span class="error-message"></span>
              </div>
```

```
            </div>
          </div>
        </div>
      </div>
      <div class="col-md-4">
        <div class="card">
          <div class="card-header">
            <h5>Key Metrics (Modern OWL Component)</h5>
          </div>
          <div class="card-body">
            <!-- Another OWL component mounted here -->
            <div class="metrics-widget-container"></div>
          </div>
        </div>
      </div>
    </div>
  </div>

  <!-- Legacy table -->
  <div class="legacy-table-section">
    <h4>Recent Orders (Legacy Table)</h4>
    <table class="table table-striped">
      <thead>
        <tr>
          <th>Order #</th>
          <th>Customer</th>
          <th>Amount</th>
          <th>Date</th>
        </tr>
      </thead>
      <tbody class="recent-orders-tbody">
        <!-- Populated by legacy JavaScript -->
      </tbody>
    </table>
  </div>

  <!-- Error container -->
  <div class="error-container" style="display: none;">
    <div class="alert alert-danger">
      <span class="error-text"></span>
    </div>
  </div>
 </div>
</t>
```

3. The Modern OWL Revenue Chart Component

```
/** @odoo-module **/

import { Component, useState, onMounted, onWillUpdateProps } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class RevenueChart extends Component {
    static template = xml`
        <div class="revenue-chart-owl">
            <div class="chart-controls">
```

```
                <select t-model="state.chartType" t-on-change="onChartTypeChange">
                    <option value="line">Line Chart</option>
                    <option value="bar">Bar Chart</option>
                    <option value="area">Area Chart</option>
                </select>
                <button class="btn btn-sm btn-secondary" t-on-click="exportChart">
                    Export
                </button>
            </div>

            <div class="chart-container" t-ref="chartContainer">
                <canvas t-ref="chartCanvas"></canvas>
            </div>

            <div class="chart-summary" t-if="chartSummary">
                <small class="text-muted">
                    Showing <t t-esc="chartSummary.totalDataPoints"/> data points.
                    Highest: <t t-esc="chartSummary.highest"/>
                    Average: <t t-esc="chartSummary.average"/>
                </small>
            </div>
        </div>
    `;

    static props = {
        initialData: { type: Array, optional: true },
        onDataPointClick: { type: Function, optional: true },
        onDateRangeChange: { type: Function, optional: true },
        legacyWidget: { type: Object, optional: true }, // Reference to legacy widget
    };

    setup() {
        this.notification = useService("notification");

        this.state = useState({
            chartType: 'line',
            data: this.props.initialData || [],
            loading: false,
        });

        this.chart = null;
        this.chartCanvasRef = useRef("chartCanvas");

        onMounted(() => {
            this.initializeChart();
        });

        onWillUpdateProps((nextProps) => {
            if (JSON.stringify(nextProps.initialData) !==
                ↪ JSON.stringify(this.props.initialData)) {
                this.updateChartData(nextProps.initialData);
            }
        });
    }
```

```
    async initializeChart() {
        // Initialize Chart.js or similar charting library
        const ctx = this.chartCanvas.getContext('2d');

        this.chart = new Chart(ctx, {
            type: this.state.chartType,
            data: this.getChartData(),
            options: {
                responsive: true,
                maintainAspectRatio: false,
                interaction: {
                    intersect: false,
                    mode: 'index',
                },
                onClick: (event, elements) => {
                    if (elements.length > 0 && this.props.onDataPointClick) {
                        const dataPoint = this.state.data[elements[0].index];
                        this.props.onDataPointClick(dataPoint);
                    }
                },
                plugins: {
                    legend: {
                        display: true,
                        position: 'top',
                    },
                    tooltip: {
                        callbacks: {
                            label: (context) => {
                                return `Revenue: $${context.parsed.y.toLocaleString()}`;
                            }
                        }
                    }
                },
                scales: {
                    x: {
                        type: 'time',
                        time: {
                            unit: 'day'
                        }
                    },
                    y: {
                        beginAtZero: true,
                        ticks: {
                            callback: function(value) {
                                return '$' + value.toLocaleString();
                            }
                        }
                    }
                }
            }
        });
    }

    getChartData() {
        return {
```

```
            labels: this.state.data.map(point => point.date),
            datasets: [{
                label: 'Revenue',
                data: this.state.data.map(point => ({
                    x: point.date,
                    y: point.revenue
                })),
                borderColor: 'rgb(75, 192, 192)',
                backgroundColor: 'rgba(75, 192, 192, 0.2)',
                tension: 0.1
            }]
        };
    }

    updateChartData(newData) {
        if (this.chart && newData) {
            this.state.data = newData;
            this.chart.data = this.getChartData();
            this.chart.update();
        }
    }

    onChartTypeChange() {
        if (this.chart) {
            this.chart.config.type = this.state.chartType;
            this.chart.update();
        }
    }

    exportChart() {
        if (this.chart) {
            const url = this.chart.toBase64Image();
            const link = document.createElement('a');
            link.download = 'revenue-chart.png';
            link.href = url;
            link.click();

            this.notification.add("Chart exported successfully", { type: "success" });
        }
    }

    get chartSummary() {
        if (!this.state.data.length) return null;

        const revenues = this.state.data.map(d => d.revenue);
        return {
            totalDataPoints: this.state.data.length,
            highest: Math.max(...revenues).toLocaleString(),
            average: (revenues.reduce((a, b) => a + b, 0) / revenues.length).toFixed(0),
        };
    }

    get chartCanvas() {
        return this.chartCanvasRef.el;
    }
```

```
}
```

This example demonstrates several key principles:

1. **Clear Separation**: Legacy and OWL code remain separate but communicate through well-defined interfaces
2. **Graceful Degradation**: If OWL components fail to load, the legacy functionality continues working
3. **Bidirectional Communication**: OWL components can notify legacy widgets of events, and vice versa
4. **Memory Management**: Proper cleanup prevents memory leaks
5. **Error Handling**: Robust error handling ensures system stability

---

Key Benefits of This Approach

1. Incremental Modernization: We replaced the simple chart with a sophisticated OWL component while keeping complex legacy features

2. Risk Management: Legacy export and dialog functionality continues working

3. Enhanced User Experience: Modern, interactive charts and tables

4. Maintainability: New features can be built in OWL while legacy features remain stable

5. Performance: Only load modern components when needed

Best Practices for Scenario 1

1. Component Isolation

Keep OWL components self-contained:

```
// Good: OWL component is independent
const owlComponent = await mount(ChartComponent, target, {
    props: {
        data: this.salesData,
        onEvent: this.handleEvent.bind(this)
    }
});

// Bad: OWL component depends on legacy widget internals
const owlComponent = await mount(ChartComponent, target, {
    props: {
        legacyWidget: this, // Too much coupling
        domElement: this.$('.some-element') // Direct DOM dependency
    }
});
```

2. Error Boundaries

Always implement graceful fallbacks:

```
async _mountOWLComponents() {
    const targets = [
        { selector: '.chart-container', component: ChartComponent },
        { selector: '.metrics-container', component: MetricsComponent }
    ];

    for (const { selector, component } of targets) {
        try {
            const target = this.el.querySelector(selector);
            if (target) {
                this.owlComponents[selector] = await mount(component, target, {
                    props: this.getComponentProps(component)
                });
            }
        } catch (error) {
            console.error(`Failed to mount ${component.name}:`, error);
            this._showFallbackUI(selector, error);
        }
    }
}

_showFallbackUI(selector, error) {
    const target = this.el.querySelector(selector);
    if (target) {
        target.innerHTML = `
            <div class="alert alert-warning">
                <h6>Component Unavailable</h6>
                <p>This feature is temporarily unavailable. Please refresh the page.</p>
                <button class="btn btn-sm btn-secondary" onclick="location.reload()">
                    Refresh Page
                </button>
            </div>
        `;
    }
}
```

3. Memory Management

Critical for preventing leaks:

```
destroy: function() {
    // Clean up OWL components first
    this._destroyOWLComponents();

    // Then call parent destroy
    this._super.apply(this, arguments);
},

_destroyOWLComponents: function() {
    Object.entries(this.owlComponents).forEach(([key, component]) => {
        try {
            if (component && typeof component.destroy === 'function') {
                component.destroy();
            }
        } catch (error) {
            console.error(`Error destroying component ${key}:`, error);
```

```
        }
    });
    this.owlComponents = {};
}
```

TL;DR: Wrap legacy widgets from inside an OWL component with `useRef` + `onMounted`/`onWillUnmount`, or mount OWL components inside a legacy widget's `start()`/`destroy()` — either direction works as long as you keep the interface between the two explicit.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch17_1_ex1`. Installation instructions are in the repository README.

Common Pitfalls

Forgetting to destroy mounted OWL components. If you mount an OWL component inside `_mountOWLComponents` but don't unmount it in `destroy`, the component (and everything it holds onto — event listeners, timers, service subscriptions) leaks every time the legacy widget is closed and reopened.

Interacting with the legacy widget's DOM before it's ready. The legacy widget's `this.el` only exists once `start()` has run. Mounting an OWL component or querying `this.el.querySelector(...)` from `init()` will fail silently or throw — always do this from inside (or after) `start()`.

Mixing the two event systems without a clear boundary. It's tempting to have OWL components call legacy `this.trigger_up(...)` directly, or have the legacy widget reach into OWL internals. Keep the interface between them explicit — callback props going one way, the legacy widget's public methods going the other — so you don't end up with two objects that both think they own the same piece of state.

Assuming the OWL component's props are "live". Legacy widgets mutate plain objects; OWL only re-renders when it detects a change through its reactivity system. Directly poking a value into `this.owlComponents.revenueChart.props.data` (like the example above does) does *not* automatically trigger a re-render — that's why the example calls `.render()` explicitly afterward.

Exercises

1. Take the `LegacyDashboard` widget and add a third OWL component (e.g., a simple `TopCustomersList`) mounted into a new container. Make sure it's properly destroyed alongside the other two.
2. Write the `_destroyOWLComponents` cleanup by hand (without looking at the "Memory Management" example) for a widget that tracks two mounted OWL components in `this.owlComponents`.
3. Modify `_onChartDataPointClick` so that instead of opening a legacy dialog, it calls a callback prop passed down from an OWL parent — describe in a sentence why that's not possible in this direction (legacy widgets aren't OWL components and don't receive props).

In Part 2, we'll explore Scenario 2 (using legacy widgets in OWL components), advanced bridge patterns, testing strategies, and migration planning. This foundational understanding of Scenario 1 will prepare you for the more complex integration challenges ahead.

What's Next?

Part 2 flips the direction you just learned — instead of mounting OWL inside a legacy widget, you'll wrap a legacy widget inside an OWL component, plus cover advanced bridge patterns and a practical migration timeline.

Chapter 17 Part 2: Advanced Bridge Patterns and Migration Strategies

Why this chapter? Because bridging in only one direction isn't enough on a real project — you'll just as often need to reuse a battle-tested legacy widget (a signature pad, a barcode scanner integration) from inside a brand-new OWL screen, and you'll need a plan for eventually retiring the bridge altogether.

What is Odoo trying to solve with this? Beyond basic interoperability, Odoo needs patterns robust enough for two-way communication and testing between old and new code, plus a realistic, incremental path off the legacy widget system rather than an indefinite dependency on it.

Real-world application: Migration timelines like the one in this chapter are what I actually hand to clients — not "rewrite everything," but a phased plan that keeps their system running while legacy widgets get replaced module by module.

In Part 1, we explored how to embed modern OWL components within legacy widgets. Now we'll tackle the reverse scenario, advanced communication patterns, and strategic approaches to complete modernization.

Scenario 2: Using Legacy Widgets in OWL Components

Sometimes you need to include complex legacy functionality in new OWL applications. This might be a specialized field widget, a complex third-party component, or legacy business logic that would take months to rewrite.

Example: Including a Legacy Field Widget in an OWL Form

Let's say you're building a modern OWL customer management interface, but you need to include a complex legacy signature widget that handles electronic signatures.

1. The Modern OWL Customer Form

```
/** @odoo-module **/

import { Component, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";
import { LegacyComponent } from "@web/legacy/legacy_component";

export class CustomerForm extends Component {
    static template = "customer_management.CustomerForm";
    static components = { LegacyComponent };

    setup() {
```

```
        this.orm = useService("orm");
        this.notification = useService("notification");

        this.state = useState({
            customer: {
                name: "",
                email: "",
                phone: "",
                signature: null,
            },
            loading: false,
            signatureRequired: false,
        });

        // Configuration for legacy signature widget
        this.legacySignatureConfig = {
            Widget: "web_digital_sign.DigitalSignWidget", // Legacy widget class name
            widgetOptions: {
                signatureMode: "draw",
                required: true,
                width: 400,
                height: 200,
                onSignatureChange: this.onSignatureChange.bind(this),
                onSignatureClear: this.onSignatureClear.bind(this),
            }
        };

        // Configuration for legacy address widget (another example)
        this.legacyAddressConfig = {
            Widget: "base_address.AddressWidget",
            widgetOptions: {
                countryField: "country_id",
                stateField: "state_id",
                onAddressValidated: this.onAddressValidated.bind(this),
            }
        };
    }
```

With the configuration in place, the component loads the customer record through the `orm` service, exactly like any regular OWL component — the legacy widgets don't change how you fetch data, only how part of the form is rendered:

```
    async loadCustomer(customerId) {
        this.state.loading = true;
        try {
            const customer = await this.orm.read("res.partner", [customerId], [
                "name", "email", "phone", "signature", "street", "city", "country_id",
                    ↪ "state_id"
            ]);

            if (customer.length > 0) {
                this.state.customer = customer[0];
                // Update legacy widget data if needed
                this._updateLegacyWidgets();
            }
        } catch (error) {
```

```
            this.notification.add("Failed to load customer data", { type: "danger" });
        } finally {
            this.state.loading = false;
        }
    }

    async saveCustomer() {
        if (!this._validateForm()) {
            return;
        }

        this.state.loading = true;
        try {
            // Get data from legacy widgets before saving
            const signatureData = this._getLegacyWidgetData('signature');
            const addressData = this._getLegacyWidgetData('address');

            const customerData = {
                ...this.state.customer,
                signature: signatureData,
                ...addressData,
            };

            if (customerData.id) {
                await this.orm.write("res.partner", [customerData.id], customerData);
            } else {
                customerData.id = await this.orm.create("res.partner", customerData);
                this.state.customer.id = customerData.id;
            }

            this.notification.add("Customer saved successfully", { type: "success" });

        } catch (error) {
            this.notification.add("Failed to save customer", { type: "danger" });
        } finally {
            this.state.loading = false;
        }
    }

    // Callbacks for legacy widget interactions
    onSignatureChange(signatureData) {
        this.state.customer.signature = signatureData;
        this.state.signatureRequired = false;
        console.log("Signature updated:", signatureData);
    }

    onSignatureClear() {
        this.state.customer.signature = null;
        this.state.signatureRequired = true;
        console.log("Signature cleared");
    }

    onAddressValidated(addressData) {
        // Update customer data with validated address
        Object.assign(this.state.customer, addressData);
```

```
        console.log("Address validated:", addressData);
    }

    // Form validation including legacy widget validation
    _validateForm() {
        if (!this.state.customer.name.trim()) {
            this.notification.add("Customer name is required", { type: "warning" });
            return false;
        }

        if (!this.state.customer.email.trim()) {
            this.notification.add("Email is required", { type: "warning" });
            return false;
        }

        // Validate legacy widgets
        if (!this._validateLegacyWidget('signature')) {
            this.notification.add("Valid signature is required", { type: "warning" });
            return false;
        }

        if (!this._validateLegacyWidget('address')) {
            this.notification.add("Valid address is required", { type: "warning" });
            return false;
        }

        return true;
    }

    // Helper methods for legacy widget interaction
    _updateLegacyWidgets() {
        // Trigger updates in legacy widgets when OWL state changes
        // This would depend on the specific legacy widget APIs
    }

    _getLegacyWidgetData(widgetType) {
        // Extract data from legacy widgets
        // Implementation depends on how legacy widgets expose their data
        return {};
    }

    _validateLegacyWidget(widgetType) {
        // Validate legacy widget data
        // Implementation depends on specific widget validation APIs
        return true;
    }

    // Event handlers for OWL form elements
    onNameChange(event) {
        this.state.customer.name = event.target.value;
    }

    onEmailChange(event) {
        this.state.customer.email = event.target.value;
    }
```

```
    onPhoneChange(event) {
        this.state.customer.phone = event.target.value;
    }

    clearSignature() {
        // Programmatically clear legacy signature widget
        // This would call methods on the legacy widget instance
    }
}
```

Notice that `saveCustomer` pulls data *out of* the legacy widgets (via `_getLegacyWidgetData`) right before saving, rather than keeping that data in OWL state at all times — the legacy signature and address widgets are the source of truth for their own fields until the moment you need to persist them.

2. The OWL Template

```
<t t-name="customer_management.CustomerForm" owl="1">
  <div class="customer-form-container">
    <div class="form-header">
      <h2>Customer Information</h2>
      <div class="form-actions">
        <button class="btn btn-primary"
                t-on-click="saveCustomer"
                t-att-disabled="state.loading">
          <t t-if="state.loading">Saving...</t>
          <t t-else="">Save Customer</t>
        </button>
      </div>
    </div>

    <!-- Modern OWL form fields -->
    <div class="row">
      <div class="col-md-6">
        <div class="card modern-form-section">
          <div class="card-header">
            <h5>Basic Information (Modern OWL)</h5>
          </div>
          <div class="card-body">
            <div class="form-group">
              <label for="customer-name">Customer Name *</label>
              <input type="text"
                     id="customer-name"
                     class="form-control"
                     t-att-value="state.customer.name"
                     t-on-input="onNameChange"
                     placeholder="Enter customer name"/>
            </div>

            <div class="form-group">
              <label for="customer-email">Email *</label>
              <input type="email"
                     id="customer-email"
                     class="form-control"
                     t-att-value="state.customer.email"
                     t-on-input="onEmailChange"
                     placeholder="customer@example.com"/>
```

```
          </div>

          <div class="form-group">
            <label for="customer-phone">Phone</label>
            <input type="tel"
                   id="customer-phone"
                   class="form-control"
                   t-att-value="state.customer.phone"
                   t-on-input="onPhoneChange"
                   placeholder="+1 (555) 123-4567"/>
          </div>
        </div>
      </div>
    </div>

    <div class="col-md-6">
      <!-- Legacy address widget -->
      <div class="card legacy-widget-section">
        <div class="card-header">
          <h5>Address Information (Legacy Widget)</h5>
        </div>
        <div class="card-body">
          <LegacyComponent
              widget="legacyAddressConfig.Widget"
              widgetOptions="legacyAddressConfig.widgetOptions"
              record="state.customer"
          />
        </div>
      </div>
    </div>
  </div>

  <!-- Legacy signature widget -->
  <div class="row">
    <div class="col-12">
      <div class="card legacy-widget-section">
        <div class="card-header">
          <h5>Digital Signature (Legacy Widget)</h5>
          <div class="card-actions">
            <button class="btn btn-sm btn-secondary"
                    t-on-click="clearSignature">
              Clear Signature
            </button>
          </div>
        </div>
        <div class="card-body">
          <div t-if="state.signatureRequired" class="alert alert-warning">
            Please provide a digital signature
          </div>

          <!-- Legacy signature widget mounted here -->
          <LegacyComponent
              widget="legacySignatureConfig.Widget"
              widgetOptions="legacySignatureConfig.widgetOptions"
              record="state.customer"
```

```
            />
          </div>
        </div>
      </div>
    </div>

    <!-- Loading overlay -->
    <div t-if="state.loading" class="loading-overlay">
      <div class="spinner-border" role="status">
        <span class="sr-only">Loading...</span>
      </div>
    </div>
  </div>
</t>
```

3. Supporting CSS for Mixed Architecture

```
/* Customer form specific styles */
.customer-form-container {
    max-width: 1200px;
    margin: 0 auto;
    padding: 20px;
}

.form-header {
    display: flex;
    justify-content: space-between;
    align-items: center;
    margin-bottom: 30px;
    padding-bottom: 15px;
    border-bottom: 2px solid #e9ecef;
}

/* Differentiate modern vs legacy sections */
.modern-form-section {
    border-left: 4px solid #007bff;
}

.modern-form-section .card-header {
    background-color: #f8f9fa;
    color: #007bff;
}

.legacy-widget-section {
    border-left: 4px solid #ffc107;
}

.legacy-widget-section .card-header {
    background-color: #fff8e1;
    color: #856404;
}

.legacy-widget-section .card-header h5::after {
    content: " (Legacy)";
    font-size: 0.8em;
    opacity: 0.7;
```

```
}

/* Loading overlay */
.loading-overlay {
    position: absolute;
    top: 0;
    left: 0;
    right: 0;
    bottom: 0;
    background: rgba(255, 255, 255, 0.8);
    display: flex;
    align-items: center;
    justify-content: center;
    z-index: 1000;
}

/* Form validation states */
.form-group.has-error .form-control {
    border-color: #dc3545;
}

.form-group.has-success .form-control {
    border-color: #28a745;
}
```

Understanding the LegacyComponent Wrapper

The `LegacyComponent` is a special OWL component provided by Odoo that acts as an adapter. Here's how it works internally:

The LegacyComponent Implementation (Simplified)

```
// This is a simplified version of how Odoo's LegacyComponent works
import { Component, onMounted, onWillUnmount, useRef, xml } from "@odoo/owl";

export class LegacyComponent extends Component {
    static template = xml`<div t-ref="legacyContainer"/>`;

    static props = {
        widget: String,
        widgetOptions: { type: Object, optional: true },
        record: { type: Object, optional: true },
    };

    setup() {
        this.legacyWidget = null;
        this.legacyContainerRef = useRef("legacyContainer");

        onMounted(() => {
            this.mountLegacyWidget();
        });

        onWillUnmount(() => {
            this.unmountLegacyWidget();
        });
    }
```

```
async mountLegacyWidget() {
    try {
        // Get the legacy widget class by name
        const WidgetClass =
            ↪  this.env.services.legacy_widget_registry.get(this.props.widget);

        if (!WidgetClass) {
            throw new Error(`Legacy widget '${this.props.widget}' not found`);
        }

        // Create instance of legacy widget
        this.legacyWidget = new WidgetClass(this, this.props.widgetOptions || {});

        // Set up data binding if record provided
        if (this.props.record) {
            this.legacyWidget.set('record', this.props.record);
        }

        // Mount the legacy widget
        await this.legacyWidget.appendTo(this.legacyContainerRef.el);

        // Set up bidirectional communication
        this.setupLegacyEvents();

    } catch (error) {
        console.error('Failed to mount legacy widget:', error);
        this.showError(error.message);
    }
}
```

Once the widget is mounted, `setupLegacyEvents` wires up the two directions of communication: legacy events flow into OWL callbacks, and (if the caller provided one) an `onValueChange` prop is forwarded to the legacy widget's own event.

```
setupLegacyEvents() {
    if (!this.legacyWidget) return;

    // Listen for legacy widget events
    this.legacyWidget.on('value_changed', this, this.onLegacyValueChange);
    this.legacyWidget.on('validation_error', this, this.onLegacyValidationError);

    // Forward OWL events to legacy widget
    if (this.props.widgetOptions.onValueChange) {
        this.legacyWidget.on('value_changed', this, (newValue) => {
            this.props.widgetOptions.onValueChange(newValue);
        });
    }
}

onLegacyValueChange(newValue) {
    // Update the record if provided
    if (this.props.record && this.legacyWidget.field_name) {
        this.props.record[this.legacyWidget.field_name] = newValue;
    }
```

```
        // Trigger OWL re-render if needed
        this.render();
    }

    onLegacyValidationError(error) {
        console.warn('Legacy widget validation error:', error);
        // Could trigger OWL component to show error state
    }

    unmountLegacyWidget() {
        if (this.legacyWidget) {
            // Remove event listeners
            this.legacyWidget.off('value_changed', this);
            this.legacyWidget.off('validation_error', this);

            // Destroy the legacy widget
            this.legacyWidget.destroy();
            this.legacyWidget = null;
        }
    }

    showError(message) {
        // Render error state in OWL component
        this.legacyContainerRef.el.innerHTML = `
            <div class="alert alert-danger">
                <strong>Widget Error:</strong> ${message}
            </div>
        `;
    }
}
```

Advanced Bridge Patterns

1. Event Communication Between Legacy and OWL

Often, you need sophisticated communication between legacy widgets and OWL components:

```
// In the legacy widget
const LegacyWidget = Widget.extend({
    start: function() {
        this._super(...arguments);

        // Listen for events from OWL components
        this.eventBus = core.bus;
        this.eventBus.on('owl_component_event', this, this._onOWLEvent);
    },

    _onOWLEvent: function(data) {
        console.log('Legacy widget received OWL event:', data);
        // React to OWL component events
        this._updateLegacyUI(data);
    },

    _triggerEventForOWL: function(data) {
```

```
        // Send events to OWL components
        this.eventBus.trigger('legacy_widget_event', data);
    },

    destroy: function() {
        this.eventBus.off('owl_component_event', this);
        this._super(...arguments);
    }
});

// In the OWL component
export class ModernComponent extends Component {
    setup() {
        this.eventBus = useService("bus");

        // Listen for events from legacy widgets
        this.eventBus.addEventListener('legacy_widget_event',
            ↪ this.onLegacyEvent.bind(this));
    }

    onLegacyEvent(event) {
        console.log('OWL component received legacy event:', event.detail);
        // React to legacy widget events
    }

    sendEventToLegacy(data) {
        // Send events to legacy widgets
        this.eventBus.trigger('owl_component_event', data);
    }
}
```

2. Shared State Management

For complex scenarios where legacy and OWL components need to share state:

```
// Shared state service that both legacy and OWL can use
odoo.define('shared_state.StateManager', function (require) {
    "use strict";

    const core = require('web.core');

    class SharedStateManager {
        constructor() {
            this.state = {};
            this.listeners = {};
        }

        setState(key, value) {
            const oldValue = this.state[key];
            this.state[key] = value;

            // Notify listeners
            if (this.listeners[key]) {
                this.listeners[key].forEach(callback => {
                    callback(value, oldValue);
                });
```

```
            }

            // Trigger global event
            core.bus.trigger('shared_state_changed', { key, value, oldValue });
        }

        getState(key) {
            return this.state[key];
        }

        subscribe(key, callback) {
            if (!this.listeners[key]) {
                this.listeners[key] = [];
            }
            this.listeners[key].push(callback);

            // Return unsubscribe function
            return () => {
                const index = this.listeners[key].indexOf(callback);
                if (index > -1) {
                    this.listeners[key].splice(index, 1);
                }
            };
        }
    }

    // Create singleton instance
    const sharedState = new SharedStateManager();

    return sharedState;
});

// Usage in legacy widget
const LegacyWidget = Widget.extend({
    start: function() {
        this._super(...arguments);
        this.sharedState = require('shared_state.StateManager');

        // Subscribe to state changes
        this.unsubscribe = this.sharedState.subscribe('currentUser', (newUser) => {
            this._updateUserDisplay(newUser);
        });
    },

    _updateCurrentUser: function(userData) {
        this.sharedState.setState('currentUser', userData);
    },

    destroy: function() {
        if (this.unsubscribe) {
            this.unsubscribe();
        }
        this._super(...arguments);
    }
});
```

```
// Usage in OWL component
export class ModernComponent extends Component {
    setup() {
        this.sharedState = useService("shared_state");

        this.state = useState({
            currentUser: this.sharedState.getState('currentUser')
        });

        // Subscribe to shared state changes
        this.unsubscribe = this.sharedState.subscribe('currentUser', (newUser) => {
            this.state.currentUser = newUser;
        });
    }

    updateUser(userData) {
        this.sharedState.setState('currentUser', userData);
    }

    willUnmount() {
        if (this.unsubscribe) {
            this.unsubscribe();
        }
    }
}
```

3. Progressive Migration Strategy

Here's a systematic approach to migrating from legacy to OWL:

```
// Migration wrapper that can switch between legacy and OWL implementations
odoo.define('migration_wrapper.ComponentSwitcher', function (require) {
    "use strict";

    const Widget = require('web.Widget');
    const { mount } = require("@odoo/owl");

    const ComponentSwitcher = Widget.extend({
        init: function(parent, options) {
            this._super(...arguments);
            this.options = options;
            this.useOWL = this._shouldUseOWL();
        },

        _shouldUseOWL: function() {
            // Logic to determine whether to use OWL or legacy
            // Could be based on feature flags, user preferences, etc.
            return this.options.forceOWL ||
                   localStorage.getItem('use_owl_components') === 'true' ||
                   this.options.migrationPhase >= 2;
        },

        async start() {
            await this._super(...arguments);
```

```
            if (this.useOWL) {
                await this._mountOWLComponent();
            } else {
                await this._mountLegacyComponent();
            }
        },

        async _mountOWLComponent() {
            const { ModernComponent } = require('my_module.ModernComponent');
            this.owlComponent = await mount(ModernComponent, this.el, {
                props: this.options.componentProps
            });
        },

        async _mountLegacyComponent() {
            const LegacyComponent = require('my_module.LegacyComponent');
            this.legacyComponent = new LegacyComponent(this,
                ↪  this.options.componentProps);
            await this.legacyComponent.appendTo(this.el);
        },

        destroy: function() {
            if (this.owlComponent) {
                this.owlComponent.destroy();
            }
            if (this.legacyComponent) {
                this.legacyComponent.destroy();
            }
            this._super(...arguments);
        }
    });

    return ComponentSwitcher;
});
```

Testing Bridge Components

Testing components that use the legacy-OWL bridge requires special considerations:

1. Testing OWL Components with Legacy Dependencies

Odoo's modern test framework is `@odoo/hoot` (Chapter 16), not the legacy QUnit runner — the same conventions apply here: `describe`/`test`/`expect` instead of `QUnit.module`/`QUnit.test`/`assert`, and `mountWithCleanup` instead of a manual `mount` + `destroy()`.

```
import { describe, test, expect, beforeEach } from "@odoo/hoot";
import { mountWithCleanup } from "@web/../tests/web_test_helpers";

describe("Bridge Components", () => {
    let mockLegacyWidget;
    let widgetWasCreated;
    let dataWasFetched;
```

```
    beforeEach(() => {
        widgetWasCreated = false;
        dataWasFetched = false;

        // Plain mock object standing in for the legacy widget instance
        mockLegacyWidget = {
            start: () => Promise.resolve(),
            destroy: () => {},
            getData: () => {
                dataWasFetched = true;
                return { signature: "mock_signature" };
            },
            validate: () => true,
        };

        // Mock the legacy widget constructor referenced by name
        window.MockLegacyWidget = function () {
            widgetWasCreated = true;
            return mockLegacyWidget;
        };
    });

    test("OWL component integrates with legacy widget", async () => {
        const customerForm = await mountWithCleanup(CustomerForm, {
            props: {
                legacyWidgetConfig: {
                    Widget: "MockLegacyWidget",
                    widgetOptions: { required: true },
                },
            },
        });

        // The legacy widget should have been instantiated
        expect(widgetWasCreated).toBe(true);

        // Data exchange with the legacy widget should have happened
        await customerForm.saveCustomer();
        expect(dataWasFetched).toBe(true);

        // No manual destroy() needed - mountWithCleanup unmounts automatically
        // after the test.
    });
});
```

Migration Best Practices

1. Start Small and Iterative

```
// Phase 1: Add new OWL components to legacy screens
// - Add modern charts to existing dashboards
// - Introduce new interactive widgets
// - Keep legacy functionality intact
```

```
// Phase 2: Replace non-critical legacy components
// - Migrate simple widgets first
// - Replace form controls and basic UI elements
// - Use bridge for complex interactions

// Phase 3: Migrate core functionality
// - Replace main form widgets
// - Migrate list views and kanban views
// - Maintain bridge only for edge cases

// Phase 4: Full modernization
// - Complete migration to OWL
// - Remove bridge code
// - Optimize performance
```

2. Migration Timeline Example

Quarter 1: Foundation - Set up bridge infrastructure - Migrate 2-3 simple components - Train team on bridge patterns - Establish testing procedures

Quarter 2: Acceleration - Migrate 5-10 medium complexity components - Build reusable bridge utilities - Optimize performance bottlenecks - Document best practices

Quarter 3: Major Features - Migrate core business logic components - Replace main form and list interfaces - Minimize bridge usage - Performance optimization

Quarter 4: Completion - Migrate remaining edge cases - Remove bridge code - Full OWL implementation - Performance validation

3. Success Metrics

Track your migration progress:

```
const MigrationMetrics = {
    totalComponents: 50,
    migratedToOWL: 35,
    usingBridge: 10,
    remainingLegacy: 5,

    get migrationProgress() {
        return (this.migratedToOWL / this.totalComponents) * 100;
    },

    get bridgeUsage() {
        return (this.usingBridge / this.totalComponents) * 100;
    },

    generateReport() {
        return {
            migrationProgress: `${this.migrationProgress.toFixed(1)}%`,
            bridgeUsage: `${this.bridgeUsage.toFixed(1)}%`,
            componentsRemaining: this.remainingLegacy,
            estimatedCompletionTime: `${Math.ceil(this.remainingLegacy / 5)} sprints`
        };
    }
};
```

```
console.table(MigrationMetrics.generateReport());
```

Performance Considerations

1. Bundle Size Management

When bridging legacy and OWL, be mindful of JavaScript bundle size:

```
// Lazy loading to avoid loading unnecessary code
const loadLegacyWidget = async (widgetName) => {
    // Only load legacy widget when needed
    const { [widgetName]: Widget } = await import(`./legacy_widgets/${widgetName}`);
    return Widget;
};

const loadOWLComponent = async (componentName) => {
    // Only load OWL component when needed
    const { [componentName]: Component } = await
        ↪ import(`./owl_components/${componentName}`);
    return Component;
};
```

2. Memory Management

Proper cleanup is crucial when mixing legacy and OWL:

```
const BridgeManager = {
    components: new Map(),

    async mountOWLInLegacy(owlComponent, target, props) {
        const component = await mount(owlComponent, target, { props });
        this.components.set(target, { type: 'owl', instance: component });
        return component;
    },

    mountLegacyInOWL(legacyWidget, target, options) {
        const widget = new legacyWidget(null, options);
        widget.appendTo(target);
        this.components.set(target, { type: 'legacy', instance: widget });
        return widget;
    },

    cleanup(target) {
        const component = this.components.get(target);
        if (component) {
            if (component.type === 'owl') {
                component.instance.destroy();
            } else if (component.type === 'legacy') {
                component.instance.destroy();
            }
            this.components.delete(target);
        }
    },
```

```
    cleanupAll() {
        this.components.forEach((component, target) => {
            this.cleanup(target);
        });
    }
};

// Ensure cleanup on page unload
window.addEventListener('beforeunload', () => {
    BridgeManager.cleanupAll();
});
```

Common Pitfalls

Forgetting `onWillUnmount` on the `LegacyComponent` wrapper. If `unmountLegacyWidget` isn't called when the OWL component is destroyed, the legacy widget instance and its DOM event listeners stay alive forever — this is the single most common source of memory leaks in bridge code.

Reading legacy widget state as if it were reactive. `this.legacyWidget.field_name` or similar values are plain properties on a legacy object; OWL has no way to know when they change. Any time the legacy widget's data needs to affect OWL rendering, it must flow through an explicit event (like `value_changed`) that updates `this.state`.

Building a new "shared state" mechanism per bridge instead of reusing one. The `SharedStateManager` pattern shown above is meant to be a single registered service, not something you re-invent for every legacy/OWL pair — otherwise legacy widgets and OWL components end up talking to different, disconnected state objects.

Skipping the migration plan. It's tempting to leave a bridge component in place indefinitely because "it works." Every bridge component should have a documented target date or trigger condition for full migration — otherwise the bridge layer itself becomes permanent technical debt.

Exercises

1. Extend the `CustomerForm` example so the legacy signature widget's `onSignatureChange` callback also calls a new `_markFormDirty()` method, and use it to disable the Save button until the user provides a signature.
2. Using the `LegacyComponent` implementation as a reference, write (on paper or in code) the `onWillUnmount` hook you'd need if `LegacyComponent` didn't already provide one — what exactly must be cleaned up, and in what order?
3. Look at the `SharedStateManager` example: rewrite `subscribe` so a component that forgets to call the returned unsubscribe function doesn't cause a memory leak (hint: consider what happens if `subscribe` is called every time a component mounts, but `unsubscribe` is only called manually).

Conclusion: Bridging the Past and Future

The legacy-OWL bridge represents more than just a technical solution—it's a strategic approach to software evolution. It acknowledges the reality that in professional software development, you rarely

have the luxury of starting from scratch. Instead, you must build the future while maintaining the present.

Throughout this book, you've learned:

1. **Modern JavaScript fundamentals** that power contemporary web development
2. **OWL component architecture** for building maintainable, scalable interfaces
3. **Advanced patterns** like state management, composition, and testing
4. **Real-world integration** techniques for working with existing systems

The bridge patterns in this chapter complete your toolkit, giving you the skills to:

- **Modernize incrementally** without breaking existing functionality
- **Integrate smoothly** between different architectural paradigms
- **Manage complexity** in mixed-technology environments
- **Plan migrations** strategically with measurable progress

Your Next Steps

As you apply these concepts in your own projects:

1. **Start small**: Choose low-risk components for your first bridge implementations
2. **Document everything**: Clear documentation helps your team understand the migration strategy
3. **Test thoroughly**: Bridge components require testing both sides of the integration
4. **Plan the future**: Every bridge component should have a migration path to pure OWL
5. **Share knowledge**: Help your team understand both legacy and modern patterns

The Professional Developer's Mindset

Professional Odoo developers understand that mastery isn't just about knowing the latest technology—it's about knowing when and how to apply the right tool for each situation. Sometimes that's cutting-edge OWL components. Sometimes it's reliable legacy widgets. Often, it's the bridge between them.

The skills you've learned in this book will serve you well as Odoo continues to evolve. You now have the foundation to adapt to new frameworks, integrate with future technologies, and most importantly, deliver value to users while maintaining system stability.

Final Challenge

As you close this book, here's a challenge to cement your learning:

Build a complete Odoo module that demonstrates everything you've learned: 1. Modern OWL components with hooks and state management 2. Integration with Odoo services and backend models 3. Comprehensive test suite 4. Bridge integration with at least one legacy component 5. Professional documentation and code organization

This project will serve as your portfolio piece and proof of mastery. More importantly, it will give you the confidence to tackle any OWL development challenge that comes your way.

Welcome to the ranks of professional Odoo developers. You're ready to build the future, one component at a time.

Happy coding, and may your bridges be strong and your migrations smooth!

TL;DR: Legacy widgets can also live inside OWL components (via `LegacyComponent` + `onWillUnmount` cleanup), two-way state sharing needs one reusable service rather than a bridge-

specific hack, and every bridge component should carry an explicit migration deadline so it doesn't become permanent.

Try it yourself: This chapter's example is available as an installable Odoo 19 addon: `simplifyit_owl_book_ch17_2_ex1`. Installation instructions are in the repository README.

What's Next?

You now have OWL 2.0 covered end to end — the final chapter is a look at where the framework is heading next, so you're not caught off guard when OWL 3 eventually arrives.

End of Chapter 17

Chapter 18: OWL 3 — What's Next (in Alpha)

> **WARNING — Alpha software, read for awareness, not for production.** Everything in this chapter describes **OWL 3**, a version currently published as `3.0.0-alpha.45` in the odoo/owl repository. The official design document says outright: *"the design is still a work in progress."* There is no stable release date. Nothing here should be used in a real project today — the APIs you learned in the rest of this book (OWL 2.0) are what ships in current Odoo versions, including Odoo 19. This chapter exists so that, when OWL 3 does arrive, none of it surprises you.

Why this chapter? Because the OWL you've just spent seventeen chapters mastering is not the end state — a major rewrite is already underway, and knowing its shape now means you won't have to relearn everything from scratch later.

What is Odoo trying to solve with this? OWL 2.0's reactivity is tied to component instances in a way that limits how state can be shared, derived, or observed outside a render cycle — OWL 3 is a ground-up rework meant to remove that limitation.

Real-world application: None yet, and that's the point — this is preparation, not a technique to bring into a client project. The realistic application today is recognizing the vocabulary (signals, Plugins, `useProps()`) when it starts showing up in release notes, so a future migration isn't a surprise.

Every framework eventually rewrites the parts of itself that turned out to be limiting. React did it going from class components to hooks. Vue did it going from Options API to Composition API. OWL is now doing something similar with its reactivity system — and the ripple effects touch props, services, lifecycle hooks, and even the template compiler.

This chapter is a guided tour of the biggest changes, based directly on OWL's own design document and draft migration guide in the odoo/owl repository. Think of it as a preview, not a tutorial — you won't build anything here, you'll just learn to recognize what's coming.

Why a New Major Version?

In OWL 2.0, reactivity is built on `useState`, which wraps an object in a `Proxy` (Chapter 7). That proxy is tied to the component that created it: OWL tracks "this component read this property, so re-render it when the property changes." It works well, but it has a structural limit — reactive values are hard to share, compute, or observe *outside* of a component's render cycle.

OWL 3 rebuilds reactivity around **signals**: small reactive containers that are not tied to any particular component. Anything that *reads* a signal — a component's render, a computed value, an effect — automatically subscribes to it, regardless of where that code lives. That single change is the root of almost everything else in this chapter.

The Reactivity Model: From Proxies to Signals

Where OWL 2.0 gives you `useState()` and `reactive()`, OWL 3 gives you four building blocks:

- **`signal(value)`** — a single reactive value, read/write.
- **`proxy(object)`** — the closest equivalent to today's `useState()`/`reactive()`: an object whose properties are each backed by a signal.
- **`computed(() => ...)`** — a *derived* value that recalculates automatically when the signals it reads change, and is cached until then.
- **`effect(() => ...)`** — runs a callback whenever the signals it reads change (similar in spirit to `useEffect`, but not tied to a component).

```
// OWL 2.0 (this book)
setup() {
    this.state = useState({ count: 0, step: 1 });
}

// OWL 3 (alpha)
setup() {
    this.state = proxy({ count: 0, step: 1 });
    this.doubled = computed(() => this.state.count * 2);
}
```

What this replaces: the pattern from Chapter 7 where you'd compute a derived value inline in the template, or recompute it on every render inside a getter. `computed()` caches the result and only recalculates when its actual dependencies change — closer to how `memoization` (Chapter 8) works, but automatic.

Props and Environment: A New Injection Model

This is the change with the widest blast radius. In OWL 2.0, every component automatically receives `this.props` and `this.env` (Chapters 8 and 11). In OWL 3, **both are removed as implicit component members.**

Props are declared explicitly with `useProps()` and a validator built from the `t` helper:

```
// OWL 2.0 (this book)
static props = { name: String, age: { type: Number, optional: true } };

// OWL 3 (alpha)
setup() {
    this.props = useProps({
        name: t.string(),
        age: t.number().optional(0),
    });
}
```

`this.env` and the whole services layer (`useService`, Chapter 13) are replaced by a **Plugin system**: instead of a global environment object that any component can pull services out of, you define a `Plugin` class and explicitly declare which components depend on it.

```
// OWL 3 (alpha) - sketch, API still moving
class OrmPlugin extends Plugin { /* ... */ }

setup() {
    this.orm = usePlugin(OrmPlugin);
}
```

Why this matters for you: everything Chapter 13 taught about `useService("orm")`, `useService("notification")`, and so on maps conceptually to "inject a Plugin" in OWL 3 — the *pattern* (declare a dependency, don't reach for a global) survives, but the API is completely different.

Lifecycle Changes

Three changes here directly affect hooks you already know from Chapter 10:

- **`onWillUpdateProps` is removed, with no direct replacement.** Depending on what you were doing with it, the migration guide points you toward `computed()` (if you were deriving state from props), `useEffect()` (if you were running a one-off side effect), or a new `asyncComputed()` hook (if you were loading data based on a prop — the prop has to be a signal for this to work).
- **`onPatched`/`onWillPatch` become stricter.** In OWL 2.0, these fire when *any* descendant re-renders, including through a slot. In OWL 3, they only fire when the component itself re-renders — a descendant's re-render no longer bubbles up to it.
- **`this.render()`, `onWillRender`, and `onRendered` are removed outright**, with no direct replacement listed yet.

Template Compiler Changes

A few changes affect how you write QWeb templates:

- **No more implicit free variables.** In OWL 2.0 you can write `t-on-click="onClick"` and OWL resolves `onClick` to `this.onClick`. OWL 3 requires the explicit `this.`: `t-on-click="this.onClick"`.
- **`t-esc` is removed.** Use `t-out` for everything — in OWL 3, `t-out` has been extended to safely handle plain values as well as markup, so it covers what `t-esc` used to do (Chapter 6).
- **`t-ref` and `t-model` now take a signal, not a string.** In OWL 2.0 you write `t-ref="myRef"` and read `this.myRef.el`; in OWL 3 the ref itself is a signal you create and pass in.
- **`t-slot` is renamed to `t-call-slot`.** The rename is meant to make it clearer that this directive *inserts* a slot's content, as opposed to `t-set-slot`, which still *defines* what a slot receives (Chapter 14).

Events

`useExternalListener` (the hook you'd use to listen on `window` or `document` with automatic cleanup) is renamed to **`useListener`**. Functionally it's the same idea: attach a listener that gets removed for you when the component unmounts.

`t-on-*` directives also gain a new `.passive` modifier for performance-sensitive listeners like `scroll` or `touchmove` — it tells the browser the handler will never call `preventDefault()`, which lets the browser optimize scrolling. It's mutually exclusive with `.prevent` for the same reason.

What's Being Removed Without a Direct Replacement

A short list of things this book covers (or that exist in OWL 2.0) that the migration guide currently lists as gone, with no 1:1 replacement yet documented:

- **`t-portal`** (rendering outside the component tree)
- **`useComponent()`** (the low-level hook for accessing the current component instance)
- **`loadFile`**

If you build something with these today, know that a future OWL 3 migration will require rethinking that part of the code, not just search-and-replacing an API name.

Current Status and Timeline

As of this writing, `odoo/owl` is publishing alpha pre-releases (the latest at research time was `3.0.0-alpha.45`) from a `master` branch that has already been restructured into separate packages (`owl-core`, `owl-compiler`, `owl-runtime`, `owl`). The `owl3_design.md` document itself states the design is still evolving, and there is no announced date for a stable `3.0.0`. Odoo's own codebase (the `saas-19.4` branch) already has commits vendoring OWL 3 alpha builds — a sign that internal testing is underway — but that is not the same as OWL 3 being ready for addon authors.

Should You Learn OWL 3 Now?

Not for production work — not yet. Everything you built in this book's examples and the companion example addons is OWL 2.0, and that's what every currently supported Odoo version runs, including Odoo 19. Treat this chapter as a map of *where the concepts are heading*, so that terms like "signals," "Plugins," and `useProps()` won't be unfamiliar when they show up in release notes or blog posts.

If you want to go deeper once you're comfortable with everything in Chapters 1–17, the primary sources are: - `doc/v3/owl/owl3_design.md` in the odoo/owl repository (the design rationale) - `doc/v3/owl/migration_owl2_to_owl3.md` in the same repository (the practical, breaking-change-by-breaking-change migration guide)

Both are living documents — expect them to change before OWL 3 stabilizes.

Summary

OWL 3 is a ground-up rework of OWL's reactivity system, moving from component-bound proxies (`useState`) to standalone **signals** (`signal`, `proxy`, `computed`, `effect`). That single shift cascades into how props are declared (`useProps()` instead of `static props`), how services are injected (a **Plugin** system instead of `env`/`useService`), which lifecycle hooks exist (`onWillUpdateProps` gone, `onPatched`/`onWillPatch` stricter), and even small template syntax rules (explicit `this.`, `t-esc` gone, `t-slot` renamed to `t-call-slot`). All of it is still **alpha** and explicitly a work in progress — read this chapter to recognize the shape of what's coming, not to start building with it.

TL;DR: OWL 3 is still alpha (`3.0.0-alpha.45`, no stable date) and rebuilds reactivity around signals, replacing `useState`/props/services/several hooks along the way — read this chapter for awareness, not to start building with it.

What's Next?

There's no next chapter — this is the end of the book. The best next step is to go back and put Chapters 1 through 17 to work: build something real with the example addons, break it, fix it, and let that be how OWL 2.0 actually sticks. When you're curious about OWL 3 again, `doc/v3/owl/` in the odoo/owl repository is where the design keeps evolving.

End of Chapter 18 End of "OWL 2.0: Where Odoo Ends and You Begin"

Keep Building With OWL

You've made it through the whole book — modern JavaScript, every core OWL 2.0 concept, a preview of what's coming in OWL 3, and 20 installable example addons to go with it. Here's where to go from here.

Keep the Examples Handy

Every chapter's example lives as an installable Odoo 19 addon in the companion examples repository (`simplifyit_owl_book_ch1_ex1` through `simplifyit_owl_book_ch17_2_ex1`) — install the ones you need and use them as a working reference the next time you're stuck on a real project. See the repository README for installation instructions.

Follow Along

New chapters and OWL/Odoo write-ups are published on the blog: **https://www.simplifyit.com.bo/blog**

Need Help on a Real Project?

If you're migrating a legacy Odoo instance, building a custom addon, or just stuck on an OWL problem this book didn't cover, Simplify IT S.R.L. does exactly this kind of work.

- Website: **https://simplifyit.com.bo**
- Email: **gm@simplifyit.com.bo**

Thanks for reading — now go build something.