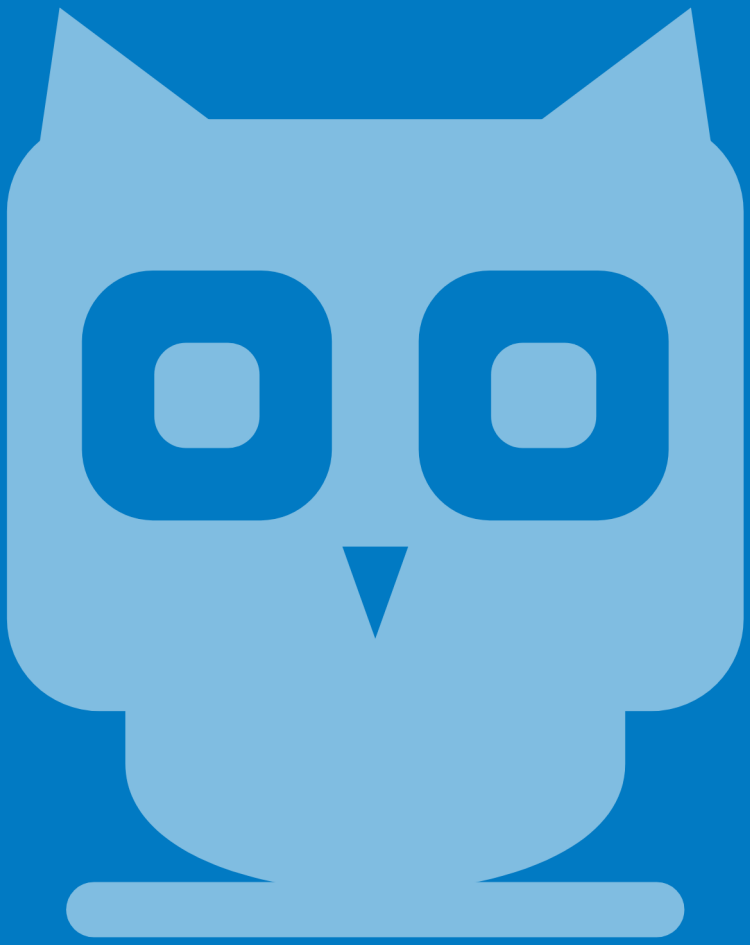
OWL 2.0

DONDE TERMINA ODOO Y EMPIEZAS TÚ

Entiende el frontend de Odoo lo suficiente como para discutirlo



Simplify it!

Grover Menacho - Simplify IT S.R.L.

Edición v1 · Agosto 2026 · simplifyit.com.bo

Índice general

Licencia

El Código de Ejemplo Tiene Otra Licencia

Los términos de arriba cubren **solo el texto del libro**. Los addons de ejemplo que lo acompañan, en github.com/simplifyitsrl/owl-book-examples, son una obra aparte publicada bajo la **Licencia Apache 2.0**.

Eso significa que las restricciones NoComercial y SinDerivadas **no** aplican al código: puedes usar, modificar y redistribuir los addons de ejemplo, incluso dentro de proyectos comerciales de Odoo, siempre que conserves los avisos de licencia y atribución que vienen con ellos. Para eso están ahí — anda y construye algo con ellos.

Esto no es documentación oficial de Odoo, y el autor no tiene ninguna afiliación con Odoo S.A. más allá de ser, como el lector, alguien que tiene que hacer que su framework funcione en el mundo real.

Capítulo 0: Una Nota del Autor

Aprendí Odoo cuando todavía era OpenERP, allá por el 2013, cuando la documentación era un PDF con formato de folleto, cuando Odoo tenía apenas un puñado de módulos y era solamente Open Source. Desde entonces me encantó jugar con lo que tenía a la mano: modificar vistas, armar mi primer sincronizador de POS entre computadoras para que una sola caja pudiera cobrar las órdenes de todas. Básicamente, siempre me gustó llevar las herramientas que tengo un nivel más allá de donde vienen. Esa es la intención de fondo de este libro: exprimir Odoo y sacarle más de lo que trae de fábrica.

Ya con eso en mente, te comento un poco más sobre mí. Llevo más de una década trabajando con Odoo. Empecé en Poiesis Consulting, en Bolivia, una de las mejores empresas de implementación que existen en el país. Después, esa misma inquietud que nunca se me quita me lanzó al mundo freelance, y ahí conocer otras culturas, otras empresas y otras formas de trabajar me abrió la cabeza y me llevó a hacer bastante más de lo que hubiera imaginado. Estos últimos dos años, a través de Simplify IT S.R.L., mi trabajo se ha centrado en hacer que Odoo haga lo que los negocios realmente necesitan — implementaciones, migraciones, módulos a medida, integraciones con autoridades tributarias y marketplaces en Bolivia, Latinoamérica, Europa y Estados Unidos. La mayor parte de ese trabajo ocurre por debajo de la superficie: modelos, vistas, lógica de negocio, la plomería que mantiene honesto a un ERP. OWL fue, durante mucho tiempo, algo que usaba sin verlo realmente.

Eso cambió el día en que la actualización de un cliente desde una versión anterior de Odoo rompió cada widget personalizado que habíamos construido para ellos. El código viejo no necesitaba parches — necesitaba repensarse. Al recorrer esa migración línea por línea, me encontré haciéndome preguntas que había evitado en silencio durante años: ¿Por qué existe OWL siquiera, cuando Odoo podría haber usado React o Vue como todos los demás? ¿Qué cambió realmente entre OWL 1.0 y 2.0, y por qué importó lo suficiente como para justificar empezar de nuevo en vez de seguir parchando? ¿Cuál es el modelo mental que se supone que debo tener, en vez del que construí sin querer copiando y pegando de addons existentes?

Este libro es el recurso que hubiera deseado que existiera cuando me hacía esas preguntas.

Está escrito desde la perspectiva de un consultor de Odoo en ejercicio, no de un autor de frameworks. No voy a fingir que OWL es la única opción razonable, ni que cada decisión de diseño en él está más allá de toda duda — te voy a contar por qué se tomaron esas decisiones, qué compromisos conllevan, y dónde aparecen en el código que escribes todos los días. Mi objetivo es que dejes de tratar a OWL como "la forma en que Odoo hace el frontend, porque sí" y empieces a tratarlo como una herramienta que entiendes lo suficientemente bien como para usarla con intención, depurarla con confianza y, de vez en cuando, estar en desacuerdo con ella.

Para Quién es Este Libro

Si construyes addons personalizados, páginas de portal o extensiones de backend para Odoo — ya vengas de un trasfondo en Python que se extiende hacia el frontend, o de un trasfondo en JavaScript tratando de entender las convenciones de Odoo — este libro es para ti. Asumo que puedes leer

JavaScript y que alguna vez abriste la carpeta `static/src` de un addon de Odoo, aunque te haya intimidado. No asumo que ya sepas qué es un proxy reactivo, o por qué Odoo abandonó el renderizado basado únicamente en QWeb, o siquiera qué significan las siglas "OWL" la primera vez que las escuchas.

Cómo Usar Este Libro

Cada capítulo se construye sobre lo asumido en los anteriores, pero traté de que los capítulos tempranos — el "por qué" antes del "cómo" — sean valiosos incluso si saltas directo a un capítulo posterior buscando una respuesta específica. En el camino encontrarás secciones de Preguntas de Reflexión y Errores Comunes extraídas directamente de trabajo real en proyectos, no de casos hipotéticos. Cuando algo rompió la instancia de producción de un cliente, o me costó una tarde que no me sobraba, lo vas a encontrar aquí, explicado con la claridad suficiente como para que a ti no te cueste esa misma tarde.

Una última nota: esto no es documentación oficial de Odoo, y no tengo ninguna afiliación con Odoo S.A. más allá de ser, como tú, alguien que tiene que hacer que su framework funcione en el mundo real. Donde mis explicaciones se apartan de la fuente oficial o la simplifican, es una decisión deliberada al servicio de un modelo mental que funcione — no una pretensión de mayor autoridad que el propio código del framework.

Una Nota Sobre Cómo se Hizo Este Libro

Este libro está construido sobre mi propia experiencia y todo lo que aprendí en el camino — pero fue refinado y afilado con la ayuda de Claude, de Anthropic. Trabajar las explicaciones, estructurar los capítulos y poner a prueba mis propios modelos mentales junto con Claude aceleró considerablemente este proyecto e hizo que escribir este libro fuera mucho más alcanzable de lo que hubiera sido de otra forma. Quiero dar el crédito donde corresponde.

Empecemos.

Capítulo 1: El "Por Qué": De Páginas Estáticas a UI Moderna

¿Por qué este capítulo? Todo consultor tarde o temprano recibe la pregunta "¿por qué no usamos jQuery para esto?" — este capítulo es la respuesta, respaldada por la historia real de la web y del propio frontend de Odoo.

¿Qué trata de resolver Odoo con esto? Odoo necesitaba una capa de UI que mantuviera funcionando miles de templates QWeb y widgets legados existentes, mientras le daba a los desarrolladores una forma moderna y declarativa de construir interfaces — algo que un framework de propósito general como React no podía garantizar.

Aplicación en la vida real: Cuando una vista de lista de un cliente se siente lenta, o un widget personalizado se rompe tras una actualización, la causa raíz casi siempre es un malentendido entre UI declarativa vs. imperativa — la distinción que este capítulo construye desde cero.

¡Bienvenido! Antes de escribir una sola línea de código OWL, necesitamos entender *por qué* existe. Cada herramienta se crea para resolver un problema, y los frameworks de UI como OWL son una solución a décadas de desafíos en la construcción de experiencias dinámicas e interactivas en la web. Este capítulo es un viaje a través del tiempo, mostrando cómo pasamos de páginas simples y estáticas a las poderosas aplicaciones que usamos hoy. Entender esta historia hará que el "cómo" de OWL sea mucho más claro.

La Forma Antigua: El Mundo del Renderizado en el Servidor

Imagina entrar a un restaurante. Miras el menú, le dices al mesero que quieres un bistec con papas fritas, y unos minutos después, un plato completo y terminado se coloca frente a ti. Esto es exactamente como funcionó la web durante mucho tiempo, y es como se construían las aplicaciones clásicas de Odoo.

Este es el modelo de **renderizado en el servidor**.

1. Tu navegador envía una solicitud al servidor de Odoo (ej., "Quiero ver la página de contacto de 'Juan Pérez' ").
2. El servidor de Odoo hace todo el trabajo. Ejecuta código Python, obtiene datos de la base de datos y usa una plantilla QWeb para ensamblar una página HTML completa.
3. Envía esta página HTML terminada de vuelta a tu navegador, que simplemente la muestra.

Este enfoque es simple y efectivo. El servidor entrega un "plato" terminado. Pero ¿qué pasa si solo quieres cambiar una pequeña cosa, como agregar una pizca de sal? En este modelo, tienes que enviar todo tu plato de vuelta a la cocina y esperar a que se haga uno completamente nuevo.

Para las aplicaciones web, esto significaba que incluso un cambio pequeño—como marcar una tarea

como completada o actualizar el número de teléfono de un cliente—requería una recarga completa de la página. La pantalla se ponía en blanco, y tenías que esperar a que el servidor enviara una versión completamente nueva de la página. Era lento, torpe y no era una gran experiencia de usuario.

La Era de jQuery: Añadiendo Interactividad

Los desarrolladores sabían que el problema de la recarga completa de página tenía que resolverse. La solución que dominó la web durante casi una década fue **jQuery**.

jQuery era una biblioteca JavaScript brillante que permitía a los desarrolladores alcanzar directamente la página web y manipularla directamente, sin pedirle al servidor una nueva. Esto se llama un enfoque **imperativo**. Le das a la computadora comandos directos, paso a paso.

Volvamos a nuestra analogía del restaurante. En lugar de pedir un plato nuevo, ahora le das al mesero comandos imperativos:

> "Ve a mi mesa. Toma el salero. Muévelo tres pulgadas a la izquierda. Ahora, toma el pimentero..."

Estás describiendo *cómo* cambiar las cosas. En código, se veía así:

```
// Cuando se hace clic en el botón con el id 'complete-task-btn'...
$('#complete-task-btn').on('click', function() {
    // 1. Encuentra el elemento con el id 'task-status'.
    // 2. Cambia su texto a '¡Completado!'.
    $('#task-status').text('¡Completado!');

    // 3. Encuentra el botón mismo.
    // 4. Añade la clase 'btn-success' para hacerlo verde.
    $(this).addClass('btn-success');
});
```

¡Esto fue un gran salto adelante! Ahora podíamos construir interfaces mucho más responsivas. Pero a medida que las aplicaciones crecían, el enfoque imperativo creó su propio conjunto de problemas:

"Código Espagueti": Terminabas con cientos de pequeñas instrucciones todas conectadas a diferentes botones y entradas. Se volvió increíblemente difícil seguir la lógica y entender en qué estado estaba la aplicación.

Difícil de Mantener: Si cambiabas la estructura HTML, tenías que encontrar y actualizar todos los comandos JavaScript que estaban buscando la estructura antigua. Era frágil y consumía mucho tiempo.

El Cambio de Paradigma: UIs Declarativas

La complejidad del modelo imperativo llevó a la siguiente evolución mayor en el desarrollo web: frameworks declarativos. Aquí es donde entran OWL, React y Vue.

En lugar de decirle a la computadora cómo hacer algo, simplemente declaras qué quieres que sea el resultado final.

En nuestro restaurante, ya no das instrucciones paso a paso. Solo declaras el estado que quieres:

> "Quiero que el salero esté en el lado izquierdo del tenedor."

No te importa cómo lo haga el mesero. Has descrito el estado final, y confías en el mesero (el framework) para que lo haga de manera eficiente.

En un framework declarativo como OWL, vinculas tu UI a los datos de tu aplicación (su "estado").

```
<!-- En nuestra plantilla (simplificada - aprenderás la sintaxis real en el Capítulo 5)
    ↪ -->
<button t-att-class="state.isCompleted ? 'btn-success' : 'btn-primary'">
  <t t-esc="state.isCompleted ? '¡Completado!' : 'Marcar como Completo'"/>
</button>
```

Cuando quieres cambiar la UI, no tocas la UI directamente. Solo cambias los datos: `this.state.isCompleted = true;`.

OWL ve que el estado ha cambiado y automática y eficientemente calcula el número mínimo de pasos necesarios para actualizar el texto y color del botón. Declaraste el "qué" (la UI debería verse así cuando isCompleted es verdadero), y OWL manejó el "cómo."

Este enfoque declarativo es la base de todo el desarrollo web moderno. Lleva a código que es:

- **Predecible**: La UI es siempre una representación directa del estado.
- **Mantenible**: Puedes cambiar el HTML subyacente y la lógica sin romper docenas de pequeños comandos.
- **Escalable**: Es mucho más fácil construir aplicaciones grandes y complejas.

Por Qué Odoo Creó Su Propio Framework: El Ajuste Perfecto

En este punto, podrías estar preguntándote: "Si React, Vue y Angular son tan populares y poderosos, ¿por qué Odoo no simplemente usó uno de ellos? ¿Por qué crear OWL desde cero?"

Esta es una excelente pregunta, y la respuesta revela la ingeniería reflexiva detrás de OWL.

Los Desafíos Específicos de Odoo

Odoo no es solo cualquier aplicación web—es una suite integral de gestión empresarial con requisitos únicos:

Integración Profunda con QWeb: Odoo ya tenía años de inversión en plantillas QWeb para renderizado del lado del servidor. El nuevo framework necesitaba trabajar sin problemas con la sintaxis y convenciones QWeb existentes, no reemplazarlas completamente.

Compatibilidad Hacia Atrás: Odoo tiene miles de módulos existentes y personalizaciones construidas con el sistema de widgets más antiguo. Cualquier nuevo framework tenía que coexistir con este código heredado durante un período de transición gradual.

Rendimiento a Escala: Las aplicaciones de Odoo pueden tener docenas de componentes simultáneos mostrando cientos de registros. El framework necesitaba estar optimizado para este caso de uso específico, no para desarrollo web general.

Tamaño de Bundle Pequeño: Agregar React o Vue aumentaría significativamente el tamaño del bundle JavaScript de Odoo. OWL es liviano y enfocado, incluyendo solo lo que Odoo realmente necesita.

Las Ventajas Técnicas de OWL

Al crear su propio framework, el equipo de Odoo pudo tomar decisiones de diseño específicas que coinciden perfectamente con sus necesidades:

Integración Nativa con QWeb: Las plantillas OWL son plantillas QWeb. No se necesita capa de traducción o conversión de sintaxis—los desarrolladores usan el mismo lenguaje de plantillas que ya conocen.

Motor de Renderizado Optimizado: El motor de renderizado de OWL está específicamente ajustado para los patrones de Odoo de visualización y manipulación de datos, haciéndolo rápido para operaciones típicas de ERP.

Sistema de Servicios Incorporado: OWL viene con un sistema de servicios que se integra directamente con la capa RPC de Odoo, modelos de base de datos y lógica de negocio.

Ruta de Migración Gradual: OWL fue diseñado desde el primer día para trabajar junto con el framework JavaScript existente de Odoo, permitiendo actualizaciones suaves e incrementales del código heredado.

La Visión Estratégica

Crear OWL no se trataba solo de resolver los problemas de hoy—se trataba de asegurar la independencia tecnológica e innovación a largo plazo de Odoo. Al poseer su framework de UI, el equipo de Odoo puede:

- Evolucionar el framework en sincronía con los requisitos de negocio de Odoo
- Optimizar el rendimiento para casos de uso específicos de ERP
- Mantener control completo sobre la experiencia del desarrollador
- Evitar dependencias externas y cambios potencialmente disruptivos de frameworks de terceros

Este es el problema que OWL fue construido para resolver para Odoo: proporcionar todo el poder y elegancia del desarrollo moderno de UI declarativa mientras está perfectamente adaptado a las necesidades únicas de aplicaciones de negocio. No es solo un framework—es la inversión estratégica de Odoo en el futuro de su plataforma.

La Evolución: De OWL 1.0 a OWL 2.0

Si estás leyendo este libro, es posible que tengas alguna experiencia con OWL 1.0, o que hayas oído hablar de la transición. Entender qué cambió entre versiones te ayudará a apreciar por qué OWL 2.0 es una mejora tan significativa y por qué vale la pena aprenderlo desde cero.

¿Qué Era OWL 1.0?

OWL 1.0 fue el primer intento de Odoo de crear un framework JavaScript moderno. Cumplió bien su propósito e introdujo a muchos desarrolladores a los conceptos de UI declarativa. Sin embargo, a medida que se usó en producción a través de miles de instalaciones de Odoo, ciertas limitaciones se hicieron evidentes:

Cuellos de Botella de Rendimiento: La implementación del virtual DOM, aunque funcional, no estaba optimizada para las interfaces complejas y con muchos datos que requieren las aplicaciones de Odoo.

Gestión de Estado Compleja: Gestionar el estado a través de múltiples componentes requería patrones verbosos y mucho código repetitivo.

Limitaciones de Plantillas: Aunque QWeb era soportado, la integración se sentía algo forzada, y algunas características avanzadas de plantillas eran difíciles de usar.

Curva de Aprendizaje: La API, aunque poderosa, tenía muchos conceptos que eran difíciles de entender rápidamente para nuevos desarrolladores.

La Revolución OWL 2.0: Qué Cambió

OWL 2.0 no fue solo una actualización—fue un rediseño completo desde cero, incorporando las lecciones aprendidas de años de uso en producción:

1. API de Componente Simplificada

Enfoque de OWL 1.0:

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

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

    async loadData() {
        // Configuración async compleja
    }
}
```

Enfoque de OWL 2.0:

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

        onWillStart(this.loadData);
    }

    async loadData() {
        // Misma lógica async, configuración más limpia
    }
}
```

2. Sistema Revolucionario de Hooks

OWL 2.0 introdujo una arquitectura basada en hooks (inspirada en React hooks) que hace que los componentes sean más modulares y reutilizables:

Antes (OWL 1.0): La gestión de estado y ciclo de vida estaba fuertemente acoplada a la clase del componente.

Después (OWL 2.0): Hooks como `useState`, `useService`, `useRef`, y `useEffect` te permiten componer funcionalidad de maneras limpias y reutilizables.

```
// OWL 2.0: Lógica limpia y componible
setup() {
    const state = useState({ count: 0 });
    const orm = useService("orm");
    const notification = useService("notification");

    useEffect(
        () => {
            // Los efectos secundarios están claramente separados
        },
        () => [state.count]  // las dependencias las devuelve una función
    );
}
```

3. Rendimiento Significativamente Mejorado

Renderizado Más Rápido: OWL 2.0 reemplazó el virtual DOM clásico por un motor de renderizado "basado en bloques" que es significativamente más rápido, especialmente al manejar listas grandes y actualizaciones frecuentes.

Re-renderizado Más Inteligente: El nuevo sistema de reactividad es mucho mejor detectando cuándo los componentes realmente necesitan actualizarse, reduciendo re-renders innecesarios.

4. Experiencia de Desarrollador Mejorada

Mejores Mensajes de Error: Cuando algo sale mal, OWL 2.0 proporciona mensajes de error mucho más claros y accionables con números de línea precisos y contexto.

Integración Mejorada con DevTools: La depuración en el navegador es mucho más intuitiva, con mejor inspección de componentes y visualización de estado.

Soporte para TypeScript: OWL 2.0 fue diseñado con TypeScript en mente, proporcionando excelente seguridad de tipos para proyectos grandes.

5. Sistema de Servicios Simplificado

OWL 1.0: Los servicios estaban disponibles pero el patrón de integración era verboso y a veces confuso.

OWL 2.0: El hook `useService` hace que acceder a servicios de Odoo (como `orm`, `notification`, `action`) sea increíblemente limpio y consistente.

```
// OWL 2.0: Acceso simple a servicios
setup() {
    this.orm = useService("orm");
    this.notification = useService("notification");
    this.actionService = useService("action");
}
```

6. Mejor Integración con JavaScript Moderno

OWL 2.0 abraza todas las características modernas de JavaScript que cubrimos en este capítulo: - Soporte nativo para patrones `async/await` - Excelente integración con módulos ES6 - Soporte incorporado para destructuring en plantillas y componentes - Template literals funcionan perfectamente con el sistema de plantillas

Estrategia de Migración: ¿Por Qué Empezar de Nuevo?

Si estás familiarizado con OWL 1.0, podrías preguntarte si migrar el código existente o empezar de nuevo. La recomendación de Odoo—y el enfoque de este libro—es **tratar OWL 2.0 como un nuevo framework** en lugar de una actualización.

Por qué este enfoque tiene sentido:

Modelos Mentales Diferentes: La arquitectura basada en hooks requiere pensar sobre componentes de manera diferente que el enfoque basado en clases de OWL 1.0.

Nuevas Mejores Prácticas: Patrones que eran óptimos en OWL 1.0 podrían ser anti-patrones en OWL 2.0.

Beneficios de Pizarra Limpia: Empezar de nuevo te permite aprovechar todas las nuevas características sin estar limitado por patrones antiguos.

A Prueba de Futuro: OWL 2.0 es la base para el frontend de Odoo por años venideros. Aprenderlo apropiadamente ahora es una inversión en tu productividad a largo plazo.

Qué Significa Esto Para Ti

Ya seas completamente nuevo a OWL o vengas de OWL 1.0, este libro te enseñará OWL 2.0 desde cero usando mejores prácticas modernas. Si tienes experiencia con OWL 1.0, algunos conceptos se sentirán familiares, pero aborda cada capítulo con mente abierta—las mejoras en OWL 2.0 probablemente cambiarán cómo piensas sobre construir interfaces de usuario.

El viaje del código espagueti de jQuery a OWL 1.0 fue significativo. El salto de OWL 1.0 a OWL 2.0 es igualmente transformador, pero en términos de productividad del desarrollador, mantenibilidad del código, y rendimiento de la aplicación.

Errores Comunes

Asumir que la sintaxis de OWL 1.0 sigue aplicando. Si encuentras tutoriales o módulos antiguos de Odoo en internet, puede que veas `constructor(parent, props)`, `willStart()`, o `this.trigger(...)`. Este libro enseña únicamente OWL 2.0, donde `setup()` reemplaza al constructor y las callback props reemplazan al disparo de eventos (Capítulo 9). No mezcles ambos estilos.

Pensar que "declarativo" significa "sin lógica". Declarativo no significa que dejes de escribir código—significa que dejas de escribir código que manipula el DOM directamente. Tú sigues decidiendo cuál debe ser el estado; OWL decide cómo reflejarlo en pantalla.

Saltarte el "por qué" para llegar rápido al "cómo". Es tentador ir directo al Capítulo 5 por código funcional. Pero el modelo mental de este capítulo—cambios de estado, no instrucciones al DOM—es lo que hace que cada capítulo posterior tenga sentido. Si un ejemplo más adelante te confunde, suele ayudar volver a leer este capítulo.

Preguntas de Reflexión

1. Usando la analogía del restaurante de este capítulo, describe con tus propias palabras la diferencia entre el enfoque de jQuery (imperativo) y el de OWL (declarativo).
2. Mira el fragmento de jQuery en la sección "La Era de jQuery". Si diez botones distintos en una página necesitaran una lógica similar, ¿qué problemas esperarías al crecer el código?

3. Nombra dos restricciones específicas de Odoo (mencionadas en "Por Qué Odoo Creó Su Propio Framework") que un framework de propósito general como React no habría resuelto de fábrica.

TL;DR: OWL existe porque Odoo necesitaba un framework de UI declarativo hecho a medida de sus propios templates QWeb, sus widgets legados y sus necesidades de rendimiento a escala de ERP — OWL 2.0 reconstruyó ese framework alrededor de hooks para una experiencia de desarrollo más simple, rápida y mantenible.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch1_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

En el siguiente capítulo, configuraremos las herramientas que necesitas para trabajar efectivamente con este poderoso framework construido a propósito — tu editor, las DevTools del navegador y la estructura de módulos de Odoo en la que construirás todo de aquí en adelante.

Capítulo 2: Tu Kit de Herramientas Esenciales de JavaScript

> **¿Por qué este capítulo?** La mayor parte del tiempo de depuración que le he facturado a clientes a lo largo de los años no lo pasé leyendo código — lo pasé mirando las pestañas de Network y Console de las DevTools. Dominar estas herramientas es lo que separa a quien adivina de quien diagnostica.
>
> **¿Qué trata de resolver Odoo con esto?** Odoo no trae su propio debugger — depende por completo de las DevTools de tu navegador y de un ciclo editor/servidor bien configurado (`--dev=all`), así que este capítulo cubre las herramientas web genéricas de las que depende todo desarrollador frontend de Odoo.
>
> **Aplicación en la vida real:** Cuando un cliente reporta que "el botón no hace nada", las pestañas Console y Network de este capítulo casi siempre son donde encuentras el error real — mucho antes de acercarte siquiera al código fuente de OWL.

Antes de que un chef pueda cocinar una obra maestra, debe conocer sus cuchillos. Antes de que un carpintero pueda construir una mesa, debe dominar su sierra y martillo. Para un desarrollador, nuestras herramientas son igual de críticas. Escribir código es solo la mitad de la batalla; la otra mitad es depurar, inspeccionar y entender qué está haciendo nuestro código.

Este capítulo presenta la herramienta más importante en tu arsenal: las **Herramientas de Desarrollador** de tu navegador web. También cubriremos la configuración esencial que necesitas en tu editor de código y proporcionaremos ejercicios prácticos para que te sientas cómodo con estas herramientas. Dominar estas herramientas no es opcional—es la forma más rápida de pasar de ser un principiante que está adivinando qué está mal a un profesional que sabe cómo descubrirlo.

El Navegador es tu Mejor Amigo

Cada navegador web moderno (como Chrome, Firefox o Edge) viene con un poderoso conjunto de "DevTools" integradas. Usualmente puedes abrirlas haciendo clic derecho en cualquier lugar de una página web y seleccionando **"Inspeccionar"** o presionando la tecla `F12`.

Para un desarrollador de Odoo, las DevTools son tu ventana al alma de tu aplicación. Te permiten ver la estructura HTML, observar la comunicación con el servidor de Odoo, y depurar tu JavaScript línea por línea. Exploremos las cuatro pestañas más importantes.

La Consola: El Diario de tu Código

La **Consola** es el primer lugar donde deberías mirar cuando algo sale mal. Es un registro en vivo donde JavaScript puede imprimir mensajes, advertencias y—lo más importante—errores.

Para qué sirve: * **Ver Errores:** Cuando tu componente OWL se bloquea, un mensaje de error

detallado aparecerá aquí, a menudo diciéndote la línea exacta de código que falló. * **Registrar Variables:** Puedes imprimir el valor de cualquier variable desde tu código usando `console.log()`. Esta es la forma más simple de verificar el estado de tu componente en cualquier momento dado. * **Pruebas Interactivas:** Puedes escribir JavaScript directamente en la consola y ejecutarlo inmediatamente—perfecto para probar pequeños pedazos de código.

Cómo usarla: Imagina que quieres ver qué datos están dentro de una variable llamada `this.state.tasks`. En el JavaScript de tu componente, añadirías:

```
console.log("Las tareas actuales son:", this.state.tasks);
```

Cuando tu código se ejecute, este mensaje aparecerá en la Consola, permitiéndote inspeccionar los datos.

Consejo profesional: Puedes usar diferentes métodos de consola para diferentes tipos de mensajes:

```
console.log("Información general");
console.warn("Esto es una advertencia");
console.error("Esto es un error");
console.table(arrayOfObjects); // Muestra arrays/objetos como tablas ordenadas
```

La Pestaña Elements: El HTML en Vivo

La pestaña **Elements** te muestra el HTML y CSS completo y en vivo de la página que estás viendo. Este no es el código fuente que escribiste; es el producto final después de que tu navegador haya renderizado todo, incluyendo el HTML generado por tus componentes OWL.

Para qué sirve: * **Inspeccionar Estructura:** Puedes ver exactamente cómo la plantilla de tu componente OWL se convirtió en HTML. ¿Está ese `div` dentro de otro `div` como esperabas? * **Probar Cambios de CSS:** Puedes seleccionar cualquier elemento y añadir, quitar o cambiar sus estilos CSS directamente en el navegador para ver el efecto instantáneamente. Esto es perfecto para probar rápidamente cambios visuales sin tener que modificar tu código y recargar. * **Encontrar Problemas de CSS:** Cuando algo no se ve bien, puedes inspeccionar el elemento para ver qué reglas CSS se están aplicando y cuáles están siendo sobrescritas.

Consejo profesional: Haz clic derecho en cualquier elemento en la pestaña Elements y selecciona "Copy selector" para obtener el selector CSS exacto para ese elemento.

La Pestaña Network: Espiando el Servidor

La pestaña **Network** es tu centro de control para toda la comunicación entre tu navegador y el servidor de Odoo. Cada vez que tu componente OWL hace una llamada RPC para obtener datos, la verás aquí.

Para qué sirve: * **Observar Llamadas RPC:** Cuando haces clic en un botón que debería guardar datos, puedes observar la pestaña Network para confirmar que se hizo una llamada al servidor de Odoo. * **Inspeccionar Cargas Útiles:** Puedes hacer clic en cualquier solicitud para ver exactamente qué datos se enviaron al servidor (la "carga útil") y qué datos envió el servidor de vuelta en su respuesta. Esto es invaluable para depurar problemas donde los datos parecen incorrectos. * **Monitoreo de Rendimiento:** Puedes ver cuánto tiempo toma cada solicitud e identificar operaciones lentas.

Consejo profesional: Usa los botones de filtro (XHR, JS, CSS, etc.) para mostrar solo el tipo de solicitudes que te interesan. Para el desarrollo de Odoo, a menudo querrás filtrar por "XHR" para ver solo las solicitudes de datos.

La Pestaña Sources: El Depurador Definitivo

La pestaña **Sources** es la herramienta más poderosa, y tal vez más intimidante. Te permite pausar tu código en medio de la ejecución e inspeccionar todo.

Para qué sirve: * **Establecer Breakpoints:** Puedes elegir una línea en tu código JavaScript y establecer un "breakpoint". Cuando el navegador llegue a esa línea, pausará toda la aplicación. * **Inspeccionar Estado:** Mientras está pausado, puedes pasar el cursor sobre cualquier variable para ver su valor actual. Puedes ver toda la pila de llamadas (qué funciones llamaron a qué otras funciones) y avanzar por tu código una línea a la vez. * **Edición de Código en Vivo:** En algunos casos, incluso puedes editar el código directamente en el navegador y ver los cambios inmediatamente.

Consejo profesional: Puedes establecer breakpoints condicionales que solo se pausan cuando se cumplen ciertas condiciones. Haz clic derecho en un número de línea y selecciona "Add conditional breakpoint."

Tu Editor de Código y Configuración de Odoo

Mientras que el navegador es para inspeccionar el *resultado* de tu código, tu editor de código es donde lo escribes.

Visual Studio Code: Tu Centro de Comando de Desarrollo

Visual Studio Code (VS Code) es el estándar de la industria para el desarrollo web moderno. Es gratuito, poderoso y tiene excelente soporte para JavaScript, Python y XML—todos los lenguajes que usarás en el desarrollo de Odoo.

Extensiones Esenciales para el Desarrollo de Odoo

Aquí están las extensiones imprescindibles de VS Code que harán tu desarrollo de Odoo/OWL mucho más eficiente:

Para JavaScript/OWL: * **JavaScript (ES6) Code Snippets:** Atajos rápidos para patrones comunes de JavaScript * **Auto Rename Tag:** Renombra automáticamente etiquetas HTML/XML emparejadas

> **Nota:** El coloreado de pares de paréntesis ya viene integrado en VS Code (Configuración → "Bracket Pair Colorization")—no necesitas ninguna extensión.

Para Python/Odoo: * **Python:** Extensión oficial de Python con resaltado de sintaxis y depuración * **Pylance:** Servidor de lenguaje Python avanzado para mejor inteligencia de código * **autoDocstring:** Genera automáticamente docstrings de Python

Para XML/QWeb: * **XML Tools:** Formateo y validación para archivos XML * **Auto Close Tag:** Cierra automáticamente etiquetas XML/HTML * **XML Language Support:** Características mejoradas de edición XML

Productividad General: * **GitLens:** Capacidades Git supercargadas * **Material Icon Theme:** Mejores iconos de archivo para diferentes tipos de archivos * **Thunder Client:** Pruebas de API directamente en VS Code (genial para probar llamadas RPC de Odoo)

Configuración del Servidor Odoo

Para hacer que Odoo recargue automáticamente tus cambios de JavaScript y CSS sin necesidad de un reinicio manual del servidor, deberías ejecutar Odoo con la bandera `--dev=all`. Esto es crucial para un flujo de trabajo de desarrollo eficiente.

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

La bandera `--dev=all` habilita: * Recarga automática del código Python * Recarga automática de assets JavaScript y CSS * Recarga automática de vistas XML y plantillas * Mensajes de error mejorados e información de depuración

---

Práctica Práctica: Sintiéndote Cómodo con las Herramientas

Pongamos estas herramientas a trabajar con algunos ejercicios prácticos. Estos te ayudarán a sentirte cómodo con las DevTools antes de que empecemos a construir componentes OWL.

Ejercicio 1: Exploración de la Consola

1. **Abre cualquier instancia de Odoo** (o cualquier sitio web) en tu navegador
2. **Abre las DevTools** presionando `F12` o haciendo clic derecho y seleccionando “Inspeccionar”
3. **Ve a la pestaña Console**
4. **Prueba estos comandos** (escribe cada uno y presiona Enter):

```
console.log("¡Hola, mundo OWL!");
console.warn("Este es un mensaje de advertencia");
console.error("Este es un mensaje de error");
console.table([{name: "Alice", age: 25}, {name: "Bob", age: 30}]);
```

5. **Observa cómo diferentes métodos de consola** muestran información de diferentes maneras

Ejercicio 2: Inspección de Elementos

1. **Navega a una vista de formulario de Odoo** (como editar un contacto o producto)
2. **Haz clic derecho en cualquier botón** y selecciona “Inspeccionar”
3. **Observa la estructura HTML** en la pestaña Elements
4. **Intenta cambiar el texto del botón** haciendo doble clic en el texto en el HTML y escribiendo algo nuevo
5. **Añade un estilo CSS** haciendo clic en el panel “Styles” y añadiendo una nueva propiedad como `background-color: red;`
6. **Observa cómo cambia el botón** en tiempo real

Ejercicio 3: Monitoreo de Red

1. **Abre la pestaña Network** antes de realizar cualquier acción
2. **Haz clic en “Guardar” en cualquier formulario de Odoo** (como editar un contacto)
3. **Observa las solicitudes aparecer** en la pestaña Network
4. **Haz clic en una de las solicitudes** para ver los detalles
5. **Mira la pestaña “Payload” o “Request”** para ver qué datos se enviaron
6. **Mira la pestaña “Response”** para ver qué envió el servidor de vuelta

Ejercicio 4: Estableciendo tu Primer Breakpoint

1. **Encuentra cualquier archivo JavaScript** en la pestaña Sources (busca archivos `.js`)
2. **Haz clic en un número de línea** para establecer un breakpoint (aparecerá un punto rojo)
3. **Realiza una acción** que activaría ese código
4. **Cuando el código se pause**, pasa el cursor sobre variables para ver sus valores
5. **Usa los controles** (play, step over, step into) para controlar la ejecución

Ejercicio 5: Verificación de Configuración de VS Code

1. **Instala VS Code** si no lo has hecho ya
2. **Instala al menos 3 extensiones** de la lista anterior
3. **Abre una carpeta de addon de Odoo** en VS Code
4. **Intenta abrir un archivo Python** y verifica que funcione el resaltado de sintaxis
5. **Intenta abrir un archivo XML** y verifica que funcione la extensión XML

Solución de Problemas Comunes de Configuración

Las DevTools no se abren: Prueba diferentes métodos—F12, Ctrl+Shift+I (Windows/Linux), o Cmd+Option+I (Mac).

No hay resaltado de sintaxis en VS Code: Asegúrate de haber instalado las extensiones de lenguaje apropiadas y que VS Code reconozca el tipo de archivo (revisa la esquina inferior derecha).

Odoo no recarga cambios: Verifica que estés ejecutando con `--dev=all` y revisa la consola para cualquier mensaje de error.

No puedo encontrar archivos JavaScript en DevTools: Busca tus archivos bajo la pestaña "Sources", a menudo en una estructura de carpetas que refleja tu addon de Odoo.

Con las DevTools de tu navegador y un editor y servidor configurados apropiadamente, tienes un taller de grado profesional. Los ejercicios prácticos anteriores te ayudarán a sentirte cómodo con estas herramientas antes de que nos sumerjamos en los fundamentos de JavaScript. Recuerda: mientras más cómodo te sientas con la depuración e inspección, más efectivo y eficiente serás como desarrollador.

TL;DR: Las DevTools de tu navegador (Console, Sources, Network) más un editor y un servidor con `--dev=all` bien configurados son el kit de herramientas mínimo no negociable para cualquier depuración de OWL que hagas en este libro — y en proyectos reales.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch2_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

En el siguiente capítulo, empezaremos a construir el conocimiento de JavaScript que necesitas para crear componentes OWL poderosos.

Capítulo 3: Fundamentos de JavaScript Moderno, Parte I

> **¿Por qué este capítulo?** Todavía veo a desarrolladores junior entregar código con `var` y loops `for` manuales en proyectos OWL — funciona, pero pelea contra el lenguaje sobre el que OWL fue diseñado, y se nota en cada code review.
>
> **¿Qué trata de resolver Odoo con esto?** OWL 2.0 (y el propio código JS de Odoo) asume que te sientes cómodo con JavaScript moderno — `let`/`const`, funciones de flecha, template literals y métodos de array no son extras opcionales, son la base en la que está escrito el propio código fuente del framework.
>
> **Aplicación en la vida real:** Cada `.map()`/`.filter()` que vayas a escribir para transformar resultados del ORM en algo que un template pueda renderizar, y cada template literal que uses para armar una clase dinámica o un mensaje, sale directamente de este capítulo.

Ahora que conoces tus herramientas, es hora de aprender el lenguaje. JavaScript moderno (a menudo llamado ES6 o más nuevo) introdujo características poderosas que ahora son el estándar para el desarrollo web. OWL está construido completamente sobre estos conceptos modernos.

Este capítulo y el siguiente son tu campo de entrenamiento de JavaScript. Nos enfocaremos en los elementos absolutamente esenciales que usarás todos los días al escribir componentes OWL. Para mantener las cosas claras, todos los ejemplos aquí son JavaScript puro, sin código de Odoo u OWL involucrado. Construyamos los cimientos.

Variables Re-imaginadas: `let` y `const`

Durante años, JavaScript tuvo solo una forma de declarar una variable: `var`. Era confuso y tenía comportamientos extraños. JavaScript moderno arregló esto dándonos dos nuevas palabras clave mucho más predecibles: `let` y `const`.

`let` es para variables cuyo valor podría necesitar cambiar más tarde. **`const`** es para constantes—variables cuyo valor, una vez asignado, nunca cambiará.

Regla general: Siempre usa `const` por defecto. Solo usa `let` si sabes que necesitarás reasignar la variable. Casi nunca necesitarás usar el viejo `var`.

```
// Usa let para una variable que cambiará.
let score = 0;
console.log(score); // Salida: 0

score = 10; // Esto está permitido.
console.log(score); // Salida: 10
```

```
// Usa const para un valor que no debería cambiar.
const playerName = "Alice";
console.log(playerName); // Salida: "Alice"

// Si tratas de cambiar un const, obtienes un error.
// playerName = "Bob"; // ¡Esto causaría un error!
```

La ventaja principal de `let` y `const` es que tienen "alcance de bloque", lo que significa que solo existen dentro de las llaves `{}` en las que están definidas. Esto hace que tu código sea mucho más fácil de razonar y previene muchos errores comunes.

```
if (true) {
    const message = "Dentro del bloque";
    console.log(message); // Funciona bien
}

// console.log(message); // ¡Error! message no existe fuera del bloque
```

Funciones de Flecha: Una Sintaxis Más Limpia

Las funciones son los bloques de construcción de cualquier aplicación. Las funciones de flecha (`=>`) proporcionan una forma más corta y limpia de escribirlas.

Aquí hay una función tradicional:

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

Y aquí está la misma función exacta escrita como una función de flecha:

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

Aún mejor, si tu función es solo una línea, puedes hacerla aún más corta. El `return` es implícito:

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

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

Para funciones con solo un parámetro, puedes omitir los paréntesis:

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

console.log(double(5)); // Salida: 10
console.log(greet("Alice")); // Salida: ¡Hola, Alice!
```

Verás funciones de flecha usadas en todas partes en JavaScript moderno y código OWL porque son concisas y ayudan a evitar algunos problemas complicados con la palabra clave `this`.

Template Literals: Construyendo Strings con Poder

Una de las características más útiles de JavaScript moderno son los **template literals**, escritos con comillas inversas (`) en lugar de comillas regulares. Hacen que crear strings dinámicos sea mucho más fácil y legible.

La Forma Antigua: Concatenación de Strings

Antes de los template literals, construir strings dinámicos era torpe:

```
const name = "Juan Pérez";
const age = 35;
const department = "Ventas";

// La forma antigua y dolorosa
const message = "Hola, " + name + "! Tienes " + age + " años y trabajas en " + department
    ↪  + ".";
console.log(message);
// Salida: Hola, Juan Pérez! Tienes 35 años y trabajas en Ventas.
```

La Forma Moderna: Template Literals

Con template literals, usas comillas inversas y `${}` para embeber expresiones directamente en el string:

```
const name = "Juan Pérez";
const age = 35;
const department = "Ventas";

// La forma moderna y limpia
const message = `Hola, ${name}! Tienes ${age} años y trabajas en ${department}.`;
console.log(message);
// Salida: Hola, Juan Pérez! Tienes 35 años y trabajas en Ventas.
```

Por Qué los Template Literals Son Perfectos para OWL

Los template literals son especialmente útiles en el desarrollo de OWL por varias razones:

1. Clases CSS Dinámicas:

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

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

2. Construyendo URLs para llamadas RPC:

```
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. Strings multi-línea (genial para depuración):

```
const debugInfo = `
    Estado del Componente:
```

```
    - Nombre: ${this.state.name}
    - Activo: ${this.state.isActive}
    - Registros: ${this.state.records.length}
`;
console.log(debugInfo);
```

4. Generación dinámica de HTML (aunque usarás templates para esto en OWL):

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

Template Literals vs. Strings Regulares

Aquí hay una comparación para mostrar cuándo usar cada uno:

```
// Usa strings regulares para texto estático
const staticMessage = "Bienvenido a Odoo";
const errorCode = "ERR_404";

// Usa template literals cuando necesites contenido dinámico
const welcomeMessage = `¡Bienvenido de vuelta, ${userName}!`;
const apiEndpoint = `${baseUrl}/api/v1/${resourceType}/${id}`;
const debugOutput = `Procesando registro ${recordId} en ${new Date()}`;

// Usa template literals para contenido multi-línea
const emailTemplate = `
    Estimado ${customerName},

    Su pedido #${orderNumber} ha sido ${status}.

    Saludos cordiales,
    El Equipo de Odoo
`;
```

Entendiendo Objetos: La Piedra Angular de los Datos

Un objeto es una colección de datos relacionados y funcionalidad, almacenados como pares clave-valor. Esta es la forma más común en que estructurarás tus datos en JavaScript. Piénsalo como una tarjeta de contacto.

```
const contact = {
    // clave: valor
    name: "Juan Pérez",
    email: "juan@ejemplo.com",
    age: 35,
    isClient: true,

    // Los objetos también pueden tener funciones (llamadas métodos)
    sayHello() {
        console.log("¡Hola!");
    }
};
```

```
// Puedes acceder a los datos usando notación de punto.
console.log(contact.name); // Salida: "Juan Pérez"

// Y llamar métodos de la misma manera.
contact.sayHello(); // Salida: "¡Hola!"
```

También puedes acceder a propiedades usando notación de corchetes, lo cual es útil cuando el nombre de la propiedad es dinámico:

```
const propertyName = "email";
console.log(contact[propertyName]); // Salida: "juan@ejemplo.com"

// Esto es especialmente útil con template literals
const fieldName = "name";
const value = contact[fieldName];
const message = `El campo ${fieldName} contiene: ${value}`;
```

Los objetos son fundamentales para OWL porque el `state` de un componente siempre es un objeto.

Dominando Arrays: Trabajando con Listas

Un array es una lista ordenada de elementos. Podría ser una lista de números, strings, o incluso otros objetos.

```
const tasks = [
    { id: 1, text: "Escribir capítulo 3", completed: true },
    { id: 2, text: "Crear ejemplos de código", completed: false },
    { id: 3, text: "Beber café", completed: false }
];
```

Constantemente necesitarás transformar o filtrar arrays para mostrarlos en tu UI. JavaScript moderno nos da métodos poderosos para hacer esto sin escribir bucles complicados. Aquí están los tres que usarás más a menudo:

`.map()` - Para Transformar un Array

El método `.map()` crea un nuevo array transformando cada elemento en el array original. Es perfecto para cuando quieres crear una lista de elementos de UI desde una lista de datos.

```
// Obtengamos un array de solo las descripciones de las tareas.
const taskTexts = tasks.map(task => task.text);

console.log(taskTexts);
// Salida: ["Escribir capítulo 3", "Crear ejemplos de código", "Beber café"]

// Usando template literals con map para transformaciones más complejas
const taskSummaries = tasks.map(task => {
    const status = task.completed ? "Hecho" : "Pendiente";
    return `${task.text} - ${status}`;
});

console.log(taskSummaries);
// Salida: ["Escribir capítulo 3 - Hecho", "Crear ejemplos de código - Pendiente", ...]
```

`.filter()` - Para Seleccionar Elementos de un Array

El método `.filter()` crea un nuevo array conteniendo solo los elementos que pasan una cierta prueba.

```
// Obtengamos solo las tareas que aún no están completadas.
const incompleteTasks = tasks.filter(task => task.completed === false);

console.log(incompleteTasks);
// Salida: Un array con los objetos "Crear ejemplos de código" y "Beber café".

// Filtrado más complejo con template literals para logging
const highPriorityTasks = tasks.filter(task => {
    const isIncomplete = !task.completed;
    const isImportant = task.text.includes("café"); // ¡Muy importante!

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

`.find()` - Para Obtener un Solo Elemento de un Array

El método `.find()` devuelve el primer elemento en un array que pasa una prueba. Es útil para encontrar un objeto específico por su ID.

```
// Encontremos la tarea con id 2.
const specificTask = tasks.find(task => task.id === 2);

console.log(specificTask);
// Salida: { id: 2, text: "Crear ejemplos de código", completed: false }

// Usando find con lógica más compleja
const findTaskByPartialText = (searchText) => {
    const foundTask = tasks.find(task =>
        task.text.toLowerCase().includes(searchText.toLowerCase())
    );

    if (foundTask) {
        console.log(`Encontrada tarea: ${foundTask.text}`);
        return foundTask;
    } else {
        console.log(`No se encontró tarea que contenga "${searchText}"`);
        return null;
    }
};

findTaskByPartialText("café"); // Encontrará la tarea "Beber café"
```

Juntándolo Todo: Un Ejemplo del Mundo Real

Combinemos todos estos conceptos en un ejemplo práctico que podrías encontrar en el desarrollo de OWL:

```
// Datos de muestra que podrían venir de un modelo de Odoo
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 }
];

// Encontrar clientes activos con más de 10 pedidos
const vipCustomers = customers
    .filter(customer => customer.isActive && customer.totalOrders > 10)
    .map(customer => {
        // Crear un resumen usando template literals
        const status = customer.isActive ? "Activo" : "Inactivo";
        const summary = `${customer.name} (${customer.email}) - ${status} -
            ↪  ${customer.totalOrders} pedidos`;

        return {
            ...customer, // El operador spread copia todas las propiedades del objeto
                ↪  original a este nuevo objeto, así no hace falta listarlas una por
                ↪  una
            summary: summary,
            displayName: `VIP: ${customer.name}`
        };
    });

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

// Esto podría mostrar:
// Clientes VIP:
// Acme Corp (info@acme.com) - Activo - 25 pedidos
// Global Industries (contact@global.com) - Activo - 42 pedidos
```

Errores Comunes

Confundir `.map()`, `.filter()` y `.forEach()`. `.map()` y `.filter()` devuelven un array *nuevo* y están pensados para usarse con ese resultado (asignado, renderizado, encadenado). `.forEach()` devuelve `undefined`—solo sirve para ejecutar efectos secundarios (como `console.log`) en cada elemento, nunca para construir una lista nueva.

Mutar el array u objeto original. `tasks.push(...)` o `contact.name = "..."` cambia los datos originales directamente. En OWL, mutar un objeto o array plano de esta forma no dispara un re-render y puede causar bugs difíciles de rastrear. Prefiere crear arrays/objetos nuevos (como ya hacen `.map()` y `.filter()`) o usar `useState`, que veremos en el Capítulo 7.

Usar == en vez de ===. == convierte los tipos antes de comparar (`"5" == 5` da `true`), lo que provoca bugs confusos. Usa siempre === (y !==) salvo que tengas una razón específica para no hacerlo.

Olvidar que const impide la *reasignación*, no la mutación. `const contact = {...}` significa que no puedes hacer `contact = otroObjeto`, pero sí puedes seguir haciendo `contact.name = "Nuevo Nombre"`. `const` solo bloquea la variable en sí, no el contenido del objeto.

Ejercicios

1. Escribe una función de flecha `formatPrice(amount, currency)` que use un template literal para devolver un string como `"$42.50"` (o `"42.50 USD"` si prefieres otro formato).
2. Dado `const numbers = [4, 15, 8, 23, 16, 42];`, usa `.filter()` para obtener solo los números mayores a 10, y luego `.map()` para duplicar cada uno. Intenta escribirlo como una sola expresión encadenada.
3. Usando el array `tasks` de este capítulo, usa `.find()` para localizar la tarea con `id: 3`, y registra un mensaje que indique si está completada o no.

TL;DR: `let`/`const`, funciones de flecha, template literals y los métodos de array `.map()`/`.filter()`/`.find()` son el vocabulario de JavaScript moderno en el que está escrito el propio código fuente de OWL — y cada ejemplo de este libro.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch3_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

Dominar `let`, `const`, funciones de flecha, template literals, objetos, y estos métodos de array te dará una base sólida para construir componentes OWL poderosos y limpios. En el siguiente capítulo, exploraremos conceptos más avanzados — clases, destructuring, módulos y `async`/`await` — que son cruciales para entender cómo funcionan internamente los componentes OWL.

Capítulo 4: Fundamentos de JavaScript Moderno, Parte II

> **¿Por qué este capítulo?** Todo componente OWL que vayas a escribir es una `class`, cada prop y resultado del ORM que manejes se beneficia del destructuring, y cada llamada al servidor es asíncrona — este es el capítulo donde esas cuatro piezas dejan de ser sintaxis abstracta y pasan a ser cómo escribirás código a diario.
>
> **¿Qué trata de resolver Odoo con esto?** El modelo de componentes de OWL (`class ... extends Component`), su sistema de módulos (`/** @odoo-module **/` e `import`/`export`) y sus servicios (`orm`, `notification`, etc.) están construidos sobre estas cuatro características de JavaScript — no hay forma de escribir un componente OWL sin ellas.
>
> **Aplicación en la vida real:** Los patrones de `async`/`await` y `try`/`catch` de aquí son exactamente lo que usarás cada vez que un componente hable con el servidor de Odoo — si te equivocas aquí, obtienes fallos silenciosos o errores de "unhandled promise rejection" en producción.

En el capítulo anterior, cubrimos los bloques de construcción básicos de JavaScript moderno. Ahora, exploraremos cuatro conceptos más avanzados que no solo son importantes—son la estructura misma sobre la cual está construido OWL. Entender clases, destructuring, módulos y código asíncrono es el paso final antes de que puedas realmente entender cómo funciona un componente OWL.

La Palabra Clave `class`: Un Plano para Componentes

Un componente OWL es, en su núcleo, una **clase** de JavaScript. Una clase es un plano para crear objetos. Agrupa datos (propiedades) y las funciones que operan sobre esos datos (métodos) en un paquete único y reutilizable.

Piensa en una `class` como el plano arquitectónico de una casa. Puedes usar el mismo plano para construir muchas casas idénticas. Cada casa que construyes es una "instancia" de la clase.

Definamos una clase simple `Dog` (Perro):

```
class Dog {
    // El constructor es un método especial que se ejecuta cuando se crea una nueva
        ↪ instancia.
    // Se usa para configurar propiedades iniciales.
    constructor(name, breed) {
        this.name = name;
        this.breed = breed;
        this.energy = 100;
    }
```

```
    // Un método es una función que pertenece a la clase.
    bark() {
        console.log(`${this.name} dice: ¡Guau!`);
    }

    play() {
        this.energy -= 10;
        console.log(`${this.name} está jugando. La energía ahora es ${this.energy}.`);
    }

    // Los métodos pueden devolver información sobre la instancia
    getInfo() {
        return `${this.name} es un ${this.breed} con ${this.energy} de energía.`;
    }
}

// Ahora, creemos dos instancias (dos perros diferentes) desde nuestro plano.
const dog1 = new Dog("Rex", "Pastor Alemán");
const dog2 = new Dog("Buddy", "Golden Retriever");

dog1.bark();      // Salida: Rex dice: ¡Guau!
dog2.play();      // Salida: Buddy está jugando. La energía ahora es 90.
console.log(dog1.getInfo()); // Salida: Rex es un Pastor Alemán con 100 de energía.
```

Extendiendo una Clase

El verdadero poder viene de la palabra clave `extends`. Te permite crear una nueva clase que hereda todas las propiedades y métodos de una existente. Este es el prerequisito directo para entender los componentes OWL, que siempre comienzan con `class MyComponent extends Component`.

Creemos una clase especializada `Puppy` (Cachorro) que herede de `Dog`:

```
class Puppy extends Dog {
    constructor(name, breed) {
        // Llamar al constructor padre con super()
        super(name, breed);

        // Los cachorros empiezan con menos energía
        this.energy = 50;
        this.isTeething = true;
    }

    // Podemos añadir un nuevo método que solo tienen los cachorros.
    chewShoe() {
        if (this.isTeething) {
            console.log(`${this.name} está masticando un zapato! ¡Mal cachorro!`);
        } else {
            console.log(`${this.name} es muy viejo para masticar zapatos.`);
        }
    }

    // También podemos sobrescribir un método existente.
    bark() {
        console.log(`${this.name} dice: ¡Yip! ¡Yip!`); // El ladrido de un cachorro es
            ↪ diferente.
    }
```

```
    // Podemos extender métodos existentes mientras usamos la funcionalidad del padre
    play() {
        super.play(); // Llamar al método play del padre
        if (this.energy < 20) {
            console.log(`${this.name} se está cansando y podría dormir pronto.`);
        }
    }
}

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

myPuppy.play();      // Heredado de Dog, pero con comportamiento específico de cachorro
myPuppy.bark();      // Sobrescrito. Salida: Leo dice: ¡Yip! ¡Yip!
myPuppy.chewShoe(); // Nuevo método. Salida: Leo está masticando un zapato! ¡Mal
    ↪  cachorro!
```

Cuando escribes `class MyComponent extends Component`, estás creando una nueva clase que hereda todas las características poderosas de la clase base `Component` de OWL, tal como nuestro cachorro heredó del Perro.

Destructuring Assignment: Extrayendo Datos con Estilo

Destructuring es una característica poderosa que te permite extraer valores de arrays y objetos en variables separadas. Es una de las características más elegantes de JavaScript moderno y la verás en todas partes en el código OWL.

Object Destructuring: Desempacando Propiedades

En lugar de acceder a propiedades de objetos de la forma antigua, destructuring te permite extraer múltiples valores en una declaración limpia:

```
// La forma antigua y repetitiva
const contact = { name: "Juan Pérez", email: "juan@example.com", age: 35, department:
    ↪  "Ventas" };

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

// La forma moderna y limpia con destructuring
const contact = { name: "Juan Pérez", email: "juan@example.com", age: 35, department:
    ↪  "Ventas" };

// Extraer múltiples propiedades en una línea
const { name, email, age } = contact;

console.log(name);  // "Juan Pérez"
console.log(email); // "juan@example.com"
console.log(age);   // 35
```

Patrones Avanzados de Object Destructuring

Renombrando variables durante destructuring:

```
const contact = { name: "Juan Pérez", email: "juan@example.com" };

// Renombrar variables durante destructuring
const { name: fullName, email: emailAddress } = contact;

console.log(fullName);     // "Juan Pérez"
console.log(emailAddress); // "juan@example.com"
```

Estableciendo valores por defecto:

```
const contact = { name: "Juan Pérez" }; // Sin propiedad email

// Proporcionar valores por defecto para propiedades faltantes
const { name, email = "sin-email@example.com", age = 0 } = contact;

console.log(email); // "sin-email@example.com"
console.log(age);   // 0
```

Destructuring anidado:

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

// Extraer propiedades anidadas directamente
const {
    name,
    address: { city, country },
    contact: { email }
} = customer;

console.log(`${name} está ubicada en ${city}, ${country}`);
console.log(`Contáctalos en ${email}`);
```

Array Destructuring: Extrayendo por Posición

Array destructuring extrae valores basado en su posición en el array:

```
const colors = ["rojo", "verde", "azul", "amarillo"];

// Extraer los primeros elementos
const [primary, secondary, tertiary] = colors;

console.log(primary);   // "rojo"
console.log(secondary); // "verde"
console.log(tertiary);  // "azul"

// Saltar elementos que no necesitas
const [first, , third] = colors; // Nota el espacio vacío para saltar "verde"
```

```
console.log(first); // "rojo"
console.log(third); // "azul"
```

El Patrón Rest: Recolectando lo que Sobra

La sintaxis `...` que verás dentro de un destructuring se llama **patrón rest**. Recolecta las propiedades (o elementos de un array) que no se extrajeron en su propia variable, agrupándolas en un nuevo objeto (o array):

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

// Extraer "id" por separado, y agrupar todo lo demás en "rest"
const { id, ...rest } = customer;

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

Esto se ve idéntico al **operador spread** del Capítulo 3 (`{ ...customer, summary }`), pero la dirección es inversa: spread *expande* las propiedades de un objeto hacia uno nuevo, mientras que rest *recolecta* las propiedades restantes en uno nuevo. Cuál de los dos comportamientos obtienes depende únicamente de dónde aparece el `...`—del lado izquierdo del `=` (un patrón que se está destructurando) es rest; del lado derecho (construyendo un valor nuevo) es spread.

Destructuring en Parámetros de Función

Aquí es donde destructuring se vuelve increíblemente poderoso para el desarrollo OWL:

```
// En lugar de acceder a propiedades dentro de la función
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>`;
};

// Destructurar directamente en los parámetros de la función
const createUserCard = ({ name, email, isActive = true }) => {
    return `<div class="user-card ${isActive ? 'active' : 'inactive'}">
        <h3>${name}</h3>
        <p>${email}</p>
    </div>`;
};

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

Por Qué Destructuring es Perfecto para OWL

Destructuring es especialmente común en OWL para varios escenarios:

1. Extrayendo props en componentes:

```
// En un componente OWL
setup() {
    const { recordId, model, readonly = false } = this.props;
    // Ahora puedes usar recordId, model, y readonly directamente
}
```

2. Trabajando con respuestas de servicios:

```
// orm.read devuelve un array - toma el primer registro con destructuring de array
const [partner] = await this.orm.read("res.partner", [partnerId], ["name", "email"]);
const { name, email } = partner;
```

3. Manejo de eventos:

```
// Extrayendo datos de eventos
onButtonClick({ target, currentTarget }) {
    // Trabajar directamente con target y currentTarget
}
```

4. Gestión de estado:

```
// Extrayendo valores de estado
const { isLoading, errorMessage, data = [] } = this.state;
```

Módulos: `import` y `export`

A medida que tu aplicación crece, no puedes mantener todo tu código en un archivo gigante. Los **módulos** te permiten dividir tu código en archivos separados y reutilizables. Puedes `export` (exportar) variables, funciones o clases de un archivo e `import` (importarlas) en otro donde se necesiten.

Así es como funciona el sistema de assets de Odoo. Cada archivo JavaScript es un módulo.

Export e Import Básicos

Imaginemos que tenemos un archivo llamado `utils.js` con algunas funciones auxiliares:

```
// Archivo: utils.js

export const PI = 3.14159;

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

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

// Export por defecto (uno por archivo)
const calculateTax = (amount, rate = 0.21) => {
    return amount * rate;
};

export default calculateTax;
```

Ahora, en otro archivo, `main.js`, podemos importar y usar ese código:

```
// Archivo: main.js

// Imports nombrados - especifica lo que quieres dentro de llaves
import { PI, add, formatCurrency } from "./utils.js";

// Import por defecto - no se necesitan llaves
import calculateTax from "./utils.js";

// También puedes importar todo
import * as Utils from "./utils.js";

console.log("El valor de PI es:", PI); // Salida: El valor de PI es: 3.14159
console.log("2 + 3 es:", add(2, 3));   // Salida: 2 + 3 es: 5
console.log(formatCurrency(99.99));     // Salida: EUR 99.99
console.log("IVA de €100:", calculateTax(100)); // Salida: IVA de €100: 21

// Usando el import de namespace
console.log("PI del namespace:", Utils.PI);
console.log("Suma:", Utils.add(5, 7));
```

Los Imports Nombrados Se Parecen al Destructuring

Los imports nombrados usan llaves, por lo que se ven igual que el destructuring (aunque técnicamente son una sintaxis propia):

```
// Elige exactamente las piezas de la librería que necesitas
import { useState, useEffect } from "@odoo/owl";
```

Ejemplo Real de Import en OWL

Así se ve un archivo típico de componente OWL:

```
// Archivo: my_component.js

// Importar la clase Component base y hooks
import { Component, useState, useService } from "@odoo/owl";

// Importar funciones utilitarias de otros archivos
import { formatDate, validateEmail } from "./utils.js";

// Definir y exportar nuestro componente
export class MyComponent extends Component {
    static template = "my.Component";

    setup() {
        // useState devuelve un objeto reactivo (no un par [valor, setter] como en
            ↪ React)
        this.state = useState({
            email: "",
            isValid: false
        });

        this.orm = useService("orm");

        // Usar funciones utilitarias importadas
```

```
        this.validateAndSetEmail = (email) => {
            this.state.email = email;
            this.state.isValid = validateEmail(email);
        };
    }
}

// Exportar para usar en otros módulos
export default MyComponent;
```

Cuando ves `import { Component } from "@odoo/owl";` al inicio de un archivo OWL, simplemente estás importando la clase base `Component` del módulo de la librería OWL para que puedas extenderla.

JavaScript Asíncrono: Manejando Retrasos

El tema más difícil pero más importante es el código **asíncrono**. No todas las tareas son instantáneas. Cuando le pides datos al servidor de Odoo (una llamada RPC), hay un retraso mientras la solicitud viaja por la red y el servidor la procesa.

Si JavaScript esperara por la respuesta, toda tu aplicación—toda la interfaz de usuario—se congelaría. Esto se llama código "bloqueante", y es una experiencia de usuario terrible.

El código asíncrono permite a JavaScript comenzar una tarea de larga duración (como una solicitud de red), y luego continuar con otro trabajo. Cuando la tarea finalmente termina, notifica a tu código para que puedas manejar el resultado.

La Forma Antigua: Promises

La solución original para esto fue un objeto llamado **Promise** (Promesa). Una Promise es un objeto que representa un valor futuro. Adjuntas `.then()` para manejar un resultado exitoso y `.catch()` para manejar un error.

```
// Este es un ejemplo simplificado de obtener datos.
fetch('https://api.example.com/data')
    .then(response => {
        // Este código se ejecuta cuando el servidor responde exitosamente.
        return response.json(); // Obtener los datos de la respuesta.
    })
    .then(data => {
        // Este código se ejecuta después de que los datos han sido procesados.
        console.log("Datos recibidos:", data);
    })
    .catch(error => {
        // Este código se ejecuta si algo salió mal.
        console.error("Falló al obtener datos:", error);
    });

console.log("¡Solicitud enviada! Esperando respuesta...");
```

La Forma Moderna: `async/await`

Aunque las Promises funcionan, encadenar `.then()` puede volverse complicado. JavaScript moderno introdujo una sintaxis mucho más limpia llamada `async/await`. Te permite escribir código asíncrono que *se ve* como código normal y síncrono.

- `async`: Pones esta palabra clave antes de una declaración de función para decirle a JavaScript que contiene operaciones asíncronas.
- `await`: Usas esta palabra clave dentro de una función `async` para decirle a JavaScript que pause la ejecución en esa línea hasta que la Promise se resuelva, sin congelar la UI.

Aquí está el mismo ejemplo usando `async/await`. Esta es la sintaxis que usarás para todas las llamadas RPC en OWL:

```
// Definimos una función async para contener nuestra lógica.
const fetchData = async () => {
    try {
        // La palabra clave 'await' pausa la función aquí hasta que el fetch se
            ↪ complete.
        const response = await fetch('https://api.example.com/data');

        // Una vez que la respuesta regresa, la función continúa.
        const data = await response.json();

        console.log("Datos recibidos:", data);

        // Puedes devolver los datos para uso en otro lugar
        return data;
    } catch (error) {
        // Si cualquier llamada 'await' falla, el bloque catch se ejecutará.
        console.error("Falló al obtener datos:", error);
        throw error; // Re-lanzar si es necesario
    }
};

fetchData();
console.log("¡Solicitud enviada! Esperando respuesta...");
```

Patrones Async del Mundo Real para OWL

Aquí hay patrones asíncronos comunes que usarás en el desarrollo OWL:

1. Cargando datos al inicio del componente:

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

        this.loadData();
    }

    async loadData() {
        try {
            // searchRead devuelve un array de objetos de registro
            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. Manejando acciones de usuario de forma asíncrona:

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

    try {
        // Mostrar estado de carga
        this.state.isSaving = true;

        // Guardar en el servidor (create devuelve un array de ids nuevos)
        const [recordId] = await this.orm.create("res.partner", [{ name, email }]);

        // Actualizar UI con éxito
        this.state.isSaving = false;
        this.state.lastSavedId = recordId;

        // Mostrar notificación
        this.notification.add("¡Registro guardado exitosamente!", { type: "success" });

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

3. Usando destructuring con funciones async:

```
async fetchCustomerData(customerId) {
    try {
        // Destructurar la respuesta directamente
        const [customer] = await this.orm.read("res.partner", [customerId], ["name",
            ↪ "email", "phone"]);

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

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

    } catch (error) {
        console.error(`Falló al obtener cliente ${customerId}:`, error);
        return null;
    }
}
```

Esta sintaxis `async/await` con un bloque `try...catch` es la forma estándar y moderna de manejar cualquier operación que involucre un retraso, y es fundamental para comunicarse con el servidor de Odoo desde tus componentes OWL.

Juntándolo Todo: Un Ejemplo Completo

Combinemos todos los conceptos de este capítulo en un ejemplo realista estilo OWL. Lo iremos construyendo método por método en vez de todo de una vez.

Primero, los imports y el método `setup()`, que prepara los servicios y el estado inicial del componente, y luego dispara la carga de datos desde el servidor:

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

// Imports de módulos con destructuring
import { Component, useState, useService } from "@odoo/owl";
import { formatDate, validateEmail } from "./utils.js";

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

    setup() {
        // Destructurar servicios
        const orm = useService("orm");
        const notification = useService("notification");

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

        // Almacenar servicios para uso en métodos
        this.orm = orm;
        this.notification = notification;

        // Cargar datos iniciales
        this.loadCustomers();
    }
```

Luego, `loadCustomers()` muestra el patrón completo `async/await` + `try/catch` de este capítulo, combinado con destructuring para leer los filtros activos y el patrón rest/spread para enriquecer cada registro con campos adicionales para mostrar:

```
    async loadCustomers() {
        try {
            // Destructurar filtros del estado
            const { activeOnly, hasEmail } = this.state.filters;

            this.state.isLoading = true;

            // Construir dominio basado en filtros
            const domain = [];
            if (activeOnly) domain.push(["active", "=", true]);
            if (hasEmail) domain.push(["email", "!=", false]);

            // searchRead devuelve un array de objetos de registro
            const records = await this.orm.searchRead(
                "res.partner",
```

```
            domain,
            ["name", "email", "phone", "active", "create_date"]
        );
        const totalCount = records.length;

        // Transformar los datos usando métodos de array y destructuring
        const processedCustomers = records.map(customer => {
            const { name, email, phone, create_date } = customer;

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

        // Actualizar estado
        this.state.customers = processedCustomers;
        this.state.isLoading = false;

        this.notification.add(
            `Cargados ${totalCount} clientes exitosamente`,
            { type: "success" }
        );

    } catch (error) {
        this.state.isLoading = false;
        console.error("Falló al cargar clientes:", error);
        this.notification.add(
            `Error cargando clientes: ${error.message}`,
            { type: "danger" }
        );
    }
}
```

`onCustomerSelect()` es más pequeño: solo demuestra el uso de `.find()` con un parámetro destructurado (`({ id }) => ...`) para localizar un registro por su id:

```
async onCustomerSelect(customerId) {
    // Encontrar cliente usando método de array con destructuring
    const selectedCustomer = this.state.customers.find(({ id }) => id ===
        ↪ customerId);

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

        // Destructurar para logging
        const { name, email } = selectedCustomer;
        console.log(`Cliente seleccionado: ${name} (${email || 'sin email'})`);
    }
}
```

Por último, `updateCustomerEmail()` pone en práctica el patrón rest de verdad: extrae `id` por separado y guarda el resto de las propiedades en `otherProps`, así actualizar el email no requiere volver a listar cada campo:

```
    // Método de clase usando async/await y destructuring
    async updateCustomerEmail(customerId, newEmail) {
        try {
            // Validar usando utilidad importada
            if (!validateEmail(newEmail)) {
                throw new Error("Formato de email inválido");
            }

            // Actualizar en el servidor
            await this.orm.write("res.partner", [customerId], { email: newEmail });

            // Actualizar estado local usando array map con 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 actualizado exitosamente!", { type: "success"
                ↪  });

        } catch (error) {
            console.error("Falló al actualizar email:", error);
            this.notification.add(`Error: ${error.message}`, { type: "danger" });
        }
    }
}

// Export por defecto
export default CustomerManager;
```

Este ejemplo demuestra: - **Clases y herencia** (extendiendo Component) - **Destructuring** en múltiples contextos (imports, estado, parámetros de función, métodos de array) - **Módulos** (import/export) - **Async/await** para comunicación con el servidor - **Template literals** para formateo de strings - **Métodos modernos de array** (map, find) combinados con destructuring

Estos son los bloques de construcción fundamentales que hacen que los componentes OWL sean poderosos y elegantes. En el siguiente capítulo, ¡empezaremos a construir tu primer componente OWL real!

Errores Comunes

Confundir spread y rest. Ambos usan `...`, pero spread (a la derecha del `=`, construyendo un valor nuevo: `{ ...customer, email }`) expande propiedades hacia afuera, mientras que rest (a la izquierda del `=`, en un patrón de destructuring: `const { id, ...rest } = customer`) recolecta las propiedades sobrantes hacia adentro. Si tu código no hace lo que esperas, revisa de qué lado del `=` está tu `...`.

Usar `this` antes de `super()` en una subclase. En una clase que hace `extends` de otra, `super(...)` debe ser la primera línea del constructor si usas `this` en cualquier parte de él. JavaScript lanzará un `ReferenceError` si intentas acceder a `this` antes.

Olvidar `await` en una llamada async. `const records = this.orm.searchRead(...)` (sin `await`) te da un objeto `Promise` pendiente, no los datos reales. Si intentas usar `.map()` o `.length` sobre algo y no funciona, revisa si olvidaste un `await`.

No envolver el `await` en `try/catch`. Si una promesa es rechazada (por ejemplo, el servidor devuelve un error) y no hay un `catch`, obtienes un "unhandled promise rejection" y tu componente falla silenciosamente en vez de mostrar retroalimentación útil al usuario.

Ejercicios

1. Extiende la clase `Puppy` de este capítulo con una nueva subclase `ServiceDog` que agregue una propiedad `task` (ej. `"guía"`, `"alerta"`) y un método `performTask()` que registre `"${name} está realizando: ${task}"`. Asegúrate de llamar a `super()` correctamente.
2. Dado `const order = { id: 5, product: "Widget", qty: 3, customer: { name: "Alice", email: "alice@example.com" } };`, destructura `id`, `qty`, y el `name` anidado (renombrado a `customerName`) en una sola instrucción.
3. Escribe una función async `checkStock(productId)` que simule una llamada de red con `await new Promise(resolve => setTimeout(resolve, 500))`, y luego devuelva `{ productId, inStock: true }`. Llámala desde otra función async usando `try/catch` y registra el resultado.

TL;DR: Los componentes OWL están construidos por completo sobre la sintaxis de `class`, el destructuring, los módulos ES (`import`/`export`) y `async`/`await` — domina estos cuatro y no quedará sintaxis de OWL que te sorprenda.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch4_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

Con JavaScript moderno en tus manos, el Capítulo 5 es donde todo esto rinde frutos: escribirás tu primer componente OWL real, registrado como una client action, y verás correr en el navegador la primera línea de código del framework en este libro.

Capítulo 5: ¡Hola, OWL! Tu Primer Componente

> **¿Por qué este capítulo?** Cada característica de OWL que aprenderás más adelante se apoya en el mismo esqueleto de tres archivos que armas aquí — si dominas esta parte, el resto del libro son adiciones, no sorpresas.
>
> **¿Qué trata de resolver Odoo con esto?** Una estructura predecible y consistente para cada pieza de interfaz personalizada del ecosistema, para que cualquier desarrollador de Odoo pueda abrir la carpeta `static/src` de cualquier addon y saber de inmediato dónde mirar.
>
> **Aplicación en la vida real:** Este es exactamente el andamiaje que reutilizarás el primer día de cualquier proyecto de cliente — un widget de dashboard personalizado, un formulario de portal, una tarjeta de kanban modificada — antes de escribir una sola línea de lógica de negocio.

¡Felicidades por completar los fundamentos! Ahora tienes todo el conocimiento de JavaScript que necesitas para comenzar a construir con OWL. En este capítulo, la teoría termina y la aplicación práctica comienza. Vamos a construir nuestro primer componente clásico "¡Hola, Mundo!".

Este es el momento donde todo lo que has aprendido—clases, módulos, plantillas literales, destructuración y la mentalidad declarativa—se une. Al final de este capítulo, tendrás un componente OWL funcionando dentro de Odoo. Comencemos.

Anatomía de un Componente OWL

Un componente OWL no es un solo archivo. Es un pequeño ecosistema autónomo típicamente compuesto por tres archivos, cada uno con una responsabilidad específica. Esta separación de responsabilidades es un principio central que mantiene tu código limpio, organizado y mantenible.

Piensa en ello como construir una casa:

1. **El Archivo JavaScript (`.js`)**: Este es el **cerebro** de tu componente. Contiene la lógica, el estado, los métodos y define cómo se comporta el componente. Aquí es donde todas tus habilidades modernas de JavaScript entran en juego.
2. **El Archivo XML (`.xml`)**: Este es el **esqueleto**. Define la estructura y diseño de la UI de tu componente usando QWeb, el poderoso lenguaje de plantillas de Odoo. Esto determina lo que ven los usuarios.
3. **El Archivo CSS/SCSS (`.scss`)**: Esta es la **piel**. Contiene todas las reglas de estilo para hacer que tu componente se vea profesional y coincida con el sistema de diseño de Odoo. (Cubriremos el estilo en detalle en capítulos posteriores.)

Para nuestro primer componente, nos enfocaremos en los primeros dos: el archivo JavaScript y el XML. No te preocupes por el estilo por ahora—primero hagamos que la funcionalidad funcione.

Configurando la Estructura de tu Módulo

Antes de escribir cualquier código, establezcamos una estructura de archivos adecuada. La organización importa, especialmente cuando tus componentes crecen en complejidad.

Crea la siguiente estructura de directorios en tu módulo de Odoo:

```
my_odoo_module/
|-- __manifest__.py
|-- static/
|   |-- src/
|       |-- components/
|           |-- hello_world/
|               |-- hello_world.js
|               |-- hello_world.xml
|               |-- hello_world.scss (lo añadiremos después)
|-- views/
    |-- hello_world_views.xml
```

¿Por qué esta estructura? - **`static/src/components/`**: Esta es la ubicación estándar para componentes OWL en módulos de Odoo - **Carpetas específicas de componentes**: Cada componente tiene su propia carpeta (`hello_world/`) para mantener los archivos relacionados juntos - **Nomenclatura consistente**: Los archivos se nombran según el componente para fácil identificación

---

El Archivo JavaScript: La Lógica del Componente

Creemos nuestro primer componente. Crea este archivo:

`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("¡El componente HelloWorld se está configurando!");
    }
}

// Registrar el componente como acción de cliente para que Odoo pueda mostrarlo
registry.category("actions").add("hello_world_app", HelloWorld);
```

Analicemos esto línea por línea:

`/** @odoo-module **/`: Este es un comentario especial que le dice al sistema de assets de Odoo que trate este archivo como un módulo. Es obligatorio para todos los archivos JavaScript en módulos de

Odoo. Sin él, tu componente no será reconocido.

`import { Component } from "@odoo/owl";`: Aquí estamos usando la sintaxis de importación con destructuración que aprendimos. Estamos extrayendo la clase `Component` de la librería OWL. Esto nos da acceso a todas las características poderosas de los componentes OWL.

`export class HelloWorld extends Component { ... }`: Esta línea hace varias cosas importantes: - `export`: Hace nuestra clase disponible para otras partes del sistema (¿recuerdas los módulos?) - `class HelloWorld`: Define nuestra clase de componente con un nombre descriptivo - `extends Component`: Hereda toda la funcionalidad de OWL (¿recuerdas la herencia de clases?)

`static template = "my_odoo_module.HelloWorld";`: Este es el enlace crucial entre nuestra lógica JavaScript y nuestra plantilla XML. El nombre debe ser: - Único en toda tu instancia de Odoo - Seguir la convención: `nombre_modulo.NombreComponente` - Coincidir exactamente con lo que definimos en el archivo XML

`setup() { ... }`: Este es el enfoque moderno de OWL 2.0 para la inicialización de componentes. Reemplaza el patrón del constructor anterior y se ejecuta cuando se crea el componente. El `console.log` nos ayudará a ver cuándo nuestro componente está funcionando.

`registry.category("actions").add("hello_world_app", HelloWorld);`: Esto registra nuestro componente en el **registro de acciones** de Odoo bajo el tag `hello_world_app`. En un momento crearemos una acción de cliente en el servidor cuyo campo `tag` coincida con este nombre—así es como Odoo sabe qué componente renderizar cuando se dispara la acción.

Entendiendo el Ciclo de Vida del Componente

Incluso en este ejemplo simple, OWL está haciendo mucho detrás de escena:

1. **Creación del Componente**: OWL crea una instancia de nuestra clase `HelloWorld`
2. **Ejecución del Setup**: Nuestro método `setup()` se ejecuta, permitiéndonos inicializar estado y servicios
3. **Renderizado de Plantilla**: OWL encuentra la plantilla que especificamos y la renderiza
4. **Inserción en el DOM**: El HTML renderizado se inserta en la página

---

El Archivo XML: La Plantilla del Componente

Ahora creemos la plantilla que nuestro archivo JavaScript referencia. Crea este archivo:

`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>
          ¡Hola, Mundo OWL!
        </h1>
        <p class="mb-0">
          ¡Este es mi primer componente OWL, y está funcionando perfectamente!
        </p>
        <hr class="my-3"/>
```

```
        <p class="mb-0">
          <strong>Nombre del Componente:</strong> HelloWorld<br/>
          <strong>Plantilla:</strong> my_odoo_module.HelloWorld<br/>
          <strong>Framework:</strong> OWL 2.0
        </p>
      </div>
    </div>
  </t>

</templates>
```

Examinemos las partes importantes:

`<templates xml:space="preserve">`: Este es el envoltorio estándar para todos los archivos de plantilla de Odoo. El `xml:space="preserve"` asegura que el espacio en blanco en tus plantillas se maneje correctamente.

`<t t-name="my_odoo_module.HelloWorld" owl="1">`: Aquí es donde ocurre la magia: - `<t>`: Una etiqueta de plantilla QWeb que no se renderiza a sí misma, solo su contenido - `t-name="my_odoo_module.HelloWorld"`: El identificador único que **debe coincidir exactamente** con la propiedad `static template` en nuestra clase JavaScript - `owl="1"`: Este atributo crucial le dice al motor QWeb de Odoo que procese esta plantilla usando el renderizador OWL en lugar del renderizador heredado

El Contenido HTML: Dentro de la etiqueta `<t>` está el HTML estándar que se convertirá en la estructura DOM real de tu componente. He usado clases de Bootstrap (que Odoo incluye) para que se vea profesional de inmediato.

¿Por qué Esta Estructura de Plantilla?

- **HTML Semántico**: Usamos elementos HTML5 apropiados y clases de Bootstrap
- **Retroalimentación visual clara**: El estilo de alerta hace obvio cuándo el componente se renderiza
- **Información de depuración**: Incluimos los nombres del componente y la plantilla para fácil identificación
- **Apariencia profesional**: Incluso un componente "Hello World" debería verse bien en Odoo

---

Registrando el Componente con Odoo

Hemos creado los archivos del componente, pero Odoo aún no sabe sobre ellos. Necesitamos registrarlos con el sistema de assets.

Paso 1: Actualizar el Manifiesto del Módulo

Abre el archivo `__manifest__.py` de tu módulo y añade los nuevos archivos al diccionario `assets`:

```python
# En __manifest__.py
{
    'name': 'Mi Módulo Hello World OWL',
    '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,
}
```

Puntos importantes sobre assets: - **`web.assets_backend`**: Este bundle se carga en el backend de Odoo (donde viven la mayoría de las aplicaciones de negocio) - **El orden importa**: Los archivos JavaScript generalmente deberían venir antes que los archivos XML - **Precisión en las rutas**: Revisa dos veces las rutas de tus archivos—los errores tipográficos aquí causarán fallas silenciosas

Paso 2: Crear una Vista para Mostrar el Componente

Crea un archivo de vista en `my_odoo_module/views/hello_world_views.xml`:

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

    <!-- Definir una acción de cliente para mostrar nuestro componente.
         El "tag" debe coincidir con el nombre usado en
    ↪  registry.category("actions").add(...) -->
    <record id="action_hello_world" model="ir.actions.client">
      <field name="name">Hola Mundo OWL</field>
      <field name="tag">hello_world_app</field>
      <field name="target">current</field>
    </record>

    <!-- Añadir un elemento de menú para que los usuarios puedan encontrar tu componente
        ↪  -->
    <menuitem id="menu_hello_world"
              name="Hola Mundo OWL"
              action="action_hello_world"
              parent="base.menu_administration"
              sequence="100"/>

  </data>
</odoo>
```

Entendiendo esta estructura:

Acción de Cliente: A diferencia de las vistas regulares de Odoo (formulario, lista, etc.), los componentes OWL a menudo usan "acciones de cliente"—acciones especiales que renderizan aplicaciones JavaScript personalizadas en lugar de vistas estándar.

El campo `tag` es el enlace: Cuando el usuario dispara esta acción, Odoo busca `hello_world_app` en el registro de acciones de JavaScript y encuentra el componente `HelloWorld` que registramos ahí. No se necesita ninguna plantilla del lado del servidor—la propia plantilla QWeb del componente se encarga del renderizado.

Integración de Menú: El `<menuitem>` les da a los usuarios una forma de acceder a tu componente a través del sistema de menú estándar de Odoo.

Probando tu Componente

¡Ahora el momento de la verdad! Vamos a hacer funcionar tu componente:

Paso 1: Instalar/Actualizar tu Módulo

```
# Si este es un módulo nuevo
./odoo-bin -d tu_base_de_datos -i my_odoo_module --dev=all

# Si estás actualizando un módulo existente
./odoo-bin -d tu_base_de_datos -u my_odoo_module --dev=all
```

La bandera `--dev=all` es crucial—asegura que tus cambios de JavaScript y XML se carguen inmediatamente sin necesidad de reiniciar el servidor.

Paso 2: Navegar a tu Componente

1. Inicia sesión en tu instancia de Odoo
2. Ve a **Configuración** (o donde hayas puesto tu elemento de menú)
3. Haz clic en **"Hola Mundo OWL"**

Si todo funcionó correctamente, ¡deberías ver tu hermoso componente renderizado con el estilo de alerta de Bootstrap!

Paso 3: Verificar en las Herramientas de Desarrollo del Navegador

Abre las DevTools de tu navegador (F12) y:

1. **Revisa la Consola**: Deberías ver el mensaje "¡El componente HelloWorld se está configurando!"
2. **Inspecciona los Elementos**: Deberías ver la estructura HTML de tu plantilla
3. **Revisa la pestaña Network**: Verifica que tus archivos `.js` y `.xml` se cargaron

---

Problemas Comunes y Soluciones

Error "Component not found": - Verifica que tu propiedad `static template` coincida exactamente con el `t-name` en tu XML - Verifica que ambos archivos estén listados en `__manifest__.py` - Asegúrate de haber actualizado tu módulo

Error "Template not found": - Confirma que el atributo `owl="1"` esté presente en tu plantilla - Revisa errores tipográficos en el nombre de la plantilla - Verifica que el archivo XML sea válido (sin errores de sintaxis)

El componente no aparece: - Revisa la consola del navegador para errores de JavaScript - Verifica que tu acción de cliente y elemento de menú estén definidos correctamente - Asegúrate de estar ejecutando con `--dev=all`

El estilo se ve mal: - Odoo incluye Bootstrap por defecto, así que nuestras clases deberían funcionar - Intenta inspeccionar los elementos para ver qué CSS se está aplicando - Recuerda que no hemos añadido CSS personalizado aún—eso viene en capítulos posteriores

---

Lo que has Logrado

¡Felicidades! Acabas de construir tu primer componente OWL e integrarlo en Odoo. Este ejemplo aparentemente simple demuestra varios conceptos cruciales:

Arquitectura de Componentes: Has visto cómo la lógica JavaScript y las plantillas XML trabajan juntas

JavaScript Moderno en la Práctica: Has usado `import`, `export`, `class` y `extends` en una aplicación real

Integración con Odoo: Entiendes cómo los componentes OWL encajan en el sistema de módulos de Odoo

Flujo de Trabajo de Desarrollo: Sabes cómo crear, registrar y probar componentes

Fundamentos de Depuración: Tienes los conceptos básicos para solucionar problemas cuando las cosas salen mal

TL;DR: Un componente OWL es una clase JS (`static template` + `setup()`) emparejada con una plantilla XML QWeb (`t-name`, `owl="1"`) cuyos nombres deben coincidir exactamente; se registra con `registry.category("actions").add(tag, Component)` y un registro `ir.actions.client` que comparte ese `tag`.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch5_ex1`. Las instrucciones de instalación están en el README del repositorio.

Ejercicios

1. **Personaliza el saludo.** Cambia el componente para que salude a un nombre específico (ej. `"¡Hola, Ana!"`) agregando una propiedad JavaScript simple en `setup()` y mostrándola con `t-esc` en la plantilla.
2. **Rómpelo a propósito.** Renombra el `t-name` en tu archivo XML para que ya no coincida con `static template` en el archivo JavaScript, recarga la página, y lee el error exacto que Odoo muestra en la consola. Luego arréglalo. Reconocer este mensaje de error te ahorrará tiempo más adelante.
3. **Añade una segunda acción.** Añade un botón que llame a un nuevo método que registre un mensaje en la consola con `console.log`, y verifícalo en la pestaña Console de las DevTools del navegador del Capítulo 2.

¿Qué Sigue?

Ahora mismo este componente es estático — muestra el mismo texto sin importar nada. El Capítulo 6 lo hace dinámico: aprenderás las directivas QWeb (`t-esc`, `t-if`, `t-foreach` y más) que permiten que tu plantilla reaccione a los datos que vienen del lado JavaScript.

Capítulo 6: Renderizado con Plantillas QWeb

> **¿Por qué este capítulo?** El HTML estático te da una demo; las directivas de QWeb te dan una aplicación real. Este es el vocabulario que vas a leer y escribir en cada archivo de plantilla de aquí en adelante.
>
> **¿Qué trata de resolver Odoo con esto?** Un único lenguaje de plantillas consistente compartido en toda la plataforma — el mismo `t-if`/`t-foreach`/`t-esc` que aprendes aquí funciona igual en cualquier módulo de Odoo, así que el conocimiento se transfiere entre proyectos en vez de reiniciarse cada vez.
>
> **Aplicación en la vida real:** Cada vista de lista, tarjeta kanban y widget de dashboard que construyas para un cliente se apoya en estas cinco o seis directivas — domínalas aquí y el 80 % de las preguntas de "cómo muestro este dato" se responden solas.

En el capítulo anterior, creamos nuestro primer componente y lo vinculamos a un archivo de plantilla. Esa plantilla contenía HTML estático, lo que significa que nunca cambiaba. Sin embargo, el verdadero poder de un framework de UI es su capacidad para renderizar datos dinámicos que responden a las interacciones del usuario y cambios de estado.

Aquí es donde QWeb, el lenguaje de plantillas de Odoo, brilla verdaderamente. Las plantillas QWeb no son solo HTML simple; están mejoradas con atributos especiales (directivas) que pueden mostrar variables, ejecutar bucles, tomar decisiones y manejar interacciones del usuario. Estas directivas son el puente entre la lógica JavaScript de tu componente y el HTML final que ve el usuario.

Para ilustrar estos conceptos de manera integral, mejoremos nuestro componente `HelloWorld` con una estructura de estado más realista que demuestre todas las características clave de QWeb.

El Estado de Nuestro Componente Mejorado (en `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: "Alicia Johnson",
        isAdmin: true,
        avatar: "/web/static/img/user_menu_avatar.png",
        lastLogin: "2024-03-15T10:30:00Z"
      },
      counter: 5,
      tasks: [
        { id: 1, text: "Aprender plantillas QWeb", completed: true, priority: "high" },
        { id: 2, text: "Construir un componente", completed: false, priority: "medium" },
```

```
        { id: 3, text: "Añadir interacciones de usuario", completed: false, priority:
    ↪  "high" },
        { id: 4, text: "Desplegar a producción", completed: false, priority: "low" }
      ],
      notifications: [
        { type: "success", message: "¡Bienvenido de vuelta!" },
        { type: "warning", message: "Mantenimiento del sistema en 1 hora" }
      ],
      settings: {
        theme: "dark",
        showCompleted: true,
        maxItems: 10
      },
      rawHtml: markup("<span>Esto contiene contenido HTML
          ↪  <strong>formateado</strong>.</span>")
    });
  }

  // Métodos que usaremos en nuestros ejemplos de plantilla
  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++;
  }
}
```

Ahora, exploremos cómo usar estos ricos datos de `state` en nuestra plantilla XML con todas las características esenciales de QWeb.

Mostrando Datos: `t-esc` y `t-out`

La tarea más fundamental es mostrar variables de tu estado. QWeb proporciona dos formas principales para renderizar datos:

`t-esc`: La Opción Segura por Defecto

La directiva `t-esc` evalúa una expresión y renderiza de forma segura el resultado como texto. Esta es tu opción por defecto para mostrar datos, ya que protege contra ataques de Cross-Site Scripting (XSS) escapando automáticamente cualquier carácter HTML.

```
<!-- Mostrar datos básicos -->
<div class="user-profile">
  <h2>¡Bienvenido, <t t-esc="state.user.name"/>!</h2>
  <p>Contador actual: <t t-esc="state.counter"/></p>
```

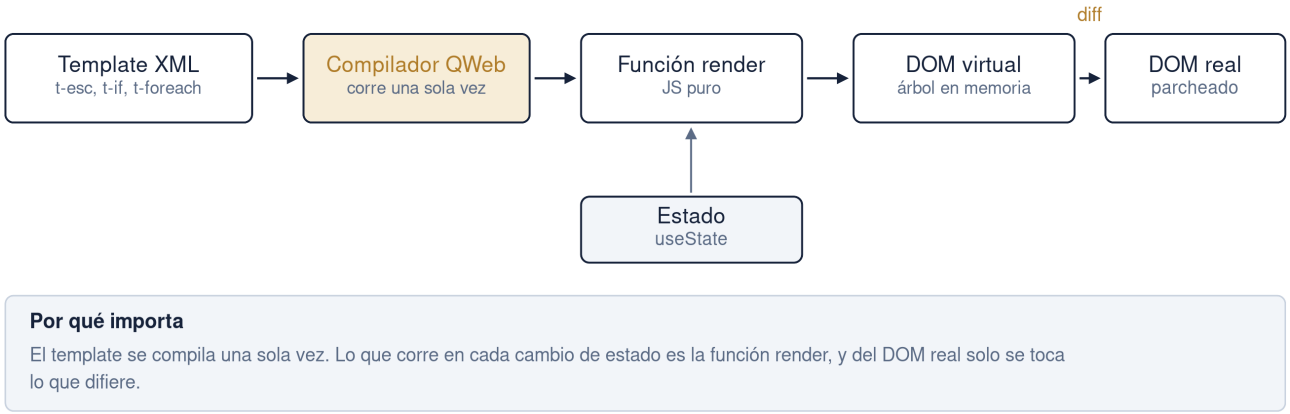


Figura 1: El pipeline de renderizado: el template se compila una sola vez, la función render corre en cada cambio.

```
  <p>Tareas completadas: <t t-esc="state.tasks.filter(t => t.completed).length"/> / <t
    ↪  t-esc="state.tasks.length"/></p>
  <small class="text-muted">Último login: <t t-esc="state.user.lastLogin"/></small>
</div>
```

Resultado:

```
<div class="user-profile">
    <h2>¡Bienvenido, Alicia Johnson!</h2>
    <p>Contador actual: 5</p>
    <p>Tareas completadas: 1 / 4</p>
    <small class="text-muted">Último login: 2024-03-15T10:30:00Z</small>
</div>
```

Puntos Clave sobre `t-esc`: - Puede ejecutar expresiones JavaScript, no solo variables simples - Automáticamente escapa HTML para prevenir ataques XSS - Perfecto para contenido generado por usuarios o cualquier dato no confiable - Soporta expresiones complejas como filtrar arrays o llamar métodos

`t-out`: Renderizando Contenido HTML Confiable

A veces tienes una variable que contiene HTML real que necesita ser renderizado como marcado. Usar `t-esc` mostraría las etiquetas HTML como texto plano. Para estos casos, OWL 2.0 proporciona `t-out`.

Por defecto, `t-out` escapa su contenido exactamente igual que `t-esc`. Para renderizar HTML real, el valor debe marcarse explícitamente como seguro con el helper `markup()` de OWL—por eso envolvimos `rawHtml` en `markup()` en el componente de arriba. (La antigua directiva `t-raw` de OWL 1.x fue eliminada precisamente porque hacía demasiado fácil renderizar HTML no confiable.)

Advertencia de Seguridad: Solo envuelve contenido en `markup()` cuando confíes completamente en él (ej., HTML generado por tu propio sistema o contenido sanitizado). Nunca lo uses con datos proporcionados por el usuario.

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

```
</div>
```

Resultado:

```
<div class="content-area">
    <div class="formatted-content">
        <span>Esto contiene contenido HTML <strong>formateado</strong>.</span>
    </div>
</div>
```

Atributos Dinámicos: `t-att` y `t-attf`

Los atributos estáticos son directos, pero los atributos dinámicos basados en el estado de tu componente son donde QWeb se vuelve poderoso.

Atributos Dinámicos Simples con `t-att`

El prefijo `t-att-` seguido del nombre del atributo te permite establecer atributos dinámicamente:

```
<!-- Clases y atributos dinámicos -->
<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">
    Contar: <t t-esc="state.counter"/>
  </button>
</div>
```

Atributos Dinámicos Complejos con `t-attf`

Para valores de atributos más complejos que necesitan interpolación de cadenas, usa `t-attf-` (formato de atributo) con marcadores `{{}}`:

```
<!-- Interpolación de cadenas en atributos -->
<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 ? 'Administrador' : 'Usuario'"/>
  </div>
</div>
```

Vinculación de Clases Basada en Objetos

Un patrón poderoso para clases condicionales:

```
<!-- Sintaxis de objeto para múltiples clases condicionales -->
<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>Panel de Control</h3>
</div>
```

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

Controla la visibilidad y renderizado de secciones de plantilla basándote en tu estado:

Condicionales Básicos

```
<div class="user-status">
  <t t-if="state.user.isAdmin">
    <div class="alert alert-info">
      <i class="fa fa-crown"></i>
      Tienes privilegios de administrador.
    </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>
      ¡Tienes muchas tareas pendientes!
    </div>
  </t>
  <t t-else="">
    <div class="alert alert-success">
      <i class="fa fa-check-circle"></i>
      ¡Todo se ve bien!
    </div>
  </t>
</div>
```

Condicionales Anidados y Lógica Compleja

```
<!-- Lógica condicional más compleja -->
<div class="task-summary">
  <t t-if="state.tasks.length > 0">
    <h4>Vista General de Tareas</h4>
    <t t-if="state.settings.showCompleted">
      <p>Mostrando todas las tareas (incluyendo completadas)</p>
    </t>
    <t t-else="">
      <p>Mostrando solo tareas pendientes</p>
    </t>

    <!-- Mostrar advertencia para tareas incompletas de alta prioridad -->
    <t t-if="state.tasks.filter(t => !t.completed and t.priority === 'high').length > 0">
      <div class="alert alert-danger">
        <strong>Atención:</strong> ¡Tienes tareas de alta prioridad pendientes!
      </div>
```

```
    </t>
  </t>
  <t t-else="">
    <div class="empty-state">
      <p>No hay tareas aún. ¡Crea tu primera tarea para empezar!</p>
    </div>
  </t>
</div>
```

Bucles: `t-foreach` y `t-as`

Renderiza listas de elementos de arrays en tu estado:

Estructura Básica de Bucle

- `t-foreach`: Especifica el array sobre el cual iterar
- `t-as`: Nombra la variable para cada elemento
- `t-key`: Proporciona una clave única para un rendimiento óptimo (crucial para la eficiencia de renderizado de OWL)

```
<div class="task-list">
  <h3>Mis Tareas (<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)}}">
            Prioridad <t t-esc="task.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 ? 'Deshacer' : 'Completar'"/>
          </button>
        </div>
      </div>
    </t>
  </div>
</div>
```

Patrones Avanzados de Bucles

Bucle con Índice

```
<!-- Acceder al índice actual con _index -->
<ol class="numbered-list">
  <t t-foreach="state.tasks" t-as="task" t-key="task.id">
    <li>
```

```
      <strong>Elemento #<t t-esc="task_index + 1"/>:</strong>
      <t t-esc="task.text"/>
    </li>
  </t>
</ol>
```

Bucles Filtrados

```
<!-- Mostrar solo tareas incompletas -->
<div class="pending-tasks">
  <h4>Tareas Pendientes</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>
```

Bucles Anidados

```
<!-- Agrupar tareas por prioridad -->
<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)}}">
          Prioridad <t t-esc="priority.toUpperCase()"/>
          (<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>
```

Características Avanzadas de QWeb

Estableciendo Variables con `t-set`

Crea variables locales dentro de tu plantilla:

```
<!-- Calcular y almacenar valores para reutilización -->
<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"/>% Completo
      </div>
    </div>

    <p>¡Has completado <t t-esc="completedTasks.length"/> de <t
      ↪  t-esc="state.tasks.length"/> tareas!</p>
</div>
```

Llamando Métodos del Componente

```
<!-- Los métodos pueden ser llamados directamente en las plantillas -->
<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>
```

Ejemplo Completo: Juntando Todo

Aquí tienes una plantilla integral que demuestra todos los conceptos que hemos cubierto. La veremos en tres partes para que sea más fácil de digerir.

Parte 1 — el encabezado y las notificaciones. Esta sección renderiza un saludo personalizado, alterna una clase de tema oscuro con la sintaxis de objeto de `t-att-class`, y cambia el color de la insignia según `state.user.isAdmin`. Las notificaciones se renderizan con `t-foreach`, cada una descartable:

```
<?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' }">

      <!-- Encabezado de Usuario con Contenido Dinámico -->
      <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">¡Bienvenido, <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 ? 'Administrador' : 'Usuario'"/>
          </span>
        </div>
      </div>
```

```
    <!-- Notificaciones -->
    <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>
```

Parte 2 — el contador y la lista de tareas (misma plantilla, continuación). La sección del contador deshabilita su botón al alcanzar `state.settings.maxItems`. La sección de tareas usa `t-set` para precalcular una sola vez el conteo de completadas, anida un `t-if` dentro de `t-foreach` para respetar la configuración `showCompleted`, y recurre a un `t-else` para el estado vacío cuando no hay tareas:

```
<!-- Sección del Contador -->
<div class="counter-section mb-4">
  <div class="card">
    <div class="card-body text-center">
      <h3>Contador: <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>
        Incrementar
      </button>
      <t t-if="state.counter >= state.settings.maxItems">
        <p class="text-warning mt-2">¡Límite máximo alcanzado!</p>
      </t>
    </div>
  </div>
</div>

<!-- Sección de Tareas -->
<div class="tasks-section">
  <div class="d-flex justify-content-between align-items-center mb-3">
    <h3>Tareas</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"/> completadas
    </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">
        <!-- Solo mostrar tarea si la configuración lo permite o está incompleta -->
        <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)}}">
                  Prioridad <t t-esc="task.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 ? 'Deshacer' : 'Completar'"/>
              </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 hay tareas aún</h4>
      <p class="text-muted">¡Crea tu primera tarea para empezar!</p>
    </div>
  </t>
</div>
```

Parte 3 — HTML confiable, y cerrando la plantilla. Finalmente, el contenido HTML presanitizado se renderiza con `t-out`, y cerramos todas las etiquetas que abrimos arriba:

```
  <!-- Ejemplo de HTML Crudo -->
  <div class="html-content mt-4">
    <h4>Contenido Formateado:</h4>
    <div class="border p-3 rounded">
      <t t-out="state.rawHtml"/>
    </div>
  </div>
  </div>
  </t>

</templates>
```

Mejores Prácticas y Consejos

Consideraciones de Rendimiento: - Siempre usa `t-key` en bucles para un rendimiento de renderizado óptimo - Prefiere `t-esc` (o `t-out` sin `markup()`) a menos que específicamente necesites renderizado HTML - Los cálculos complejos deberían hacerse en métodos JavaScript, no en plantillas

Seguridad: - Nunca envuelvas contenido proporcionado por usuarios en `markup()` - Siempre valida los datos antes de renderizar - Usa `t-esc` para cualquier contenido dinámico que pueda contener HTML

Mantenibilidad: - Mantén la lógica de plantillas simple; mueve la lógica compleja a métodos del componente - Usa nombres descriptivos de variables en directivas `t-as` - Agrupa secciones de plantilla

relacionadas con comentarios

Patrones Comunes: - Usa `t-set` para cálculos complejos que se usan múltiples veces - Combina `t-if` con `t-foreach` para renderizado condicional de listas - Usa sintaxis de objeto para `t-att-class` cuando manejes múltiples clases condicionales

Estos cuatro conceptos fundamentales—mostrar datos, atributos dinámicos, condicionales y bucles—son los bloques de construcción de QWeb. Al dominarlos y entender sus patrones avanzados, puedes construir interfaces de usuario sofisticadas y basadas en datos que reaccionan instantánea y eficientemente a los cambios en el estado de tu componente.

TL;DR: Las directivas de QWeb (`t-esc`/`t-out`, `t-att*`, `t-if`/`t-elif`/`t-else`, `t-foreach`+`t-key`) son cómo tu plantilla lee y reacciona a los datos de tu componente — siempre combina bucles con un `t-key` estable, y nunca pongas contenido de usuario en `markup()`.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch6_ex1`. Las instrucciones de instalación están en el README del repositorio.

Errores Comunes

Error 1: Olvidar `t-key` en `t-foreach`

```
<!-- Incorrecto - sin t-key: OWL no puede rastrear qué elemento es cuál entre
    ↪ renderizados -->
<t t-foreach="state.tasks" t-as="task">
  <div><t t-esc="task.text"/></div>
</t>

<!-- Correcto - una clave única y estable por elemento -->
<t t-foreach="state.tasks" t-as="task" t-key="task.id">
  <div><t t-esc="task.text"/></div>
</t>
```

Sin `t-key`, OWL puede reutilizar el nodo DOM equivocado para el elemento equivocado, causando texto desactualizado, pérdida de foco en inputs, o parpadeo cuando la lista cambia.

Error 2: Recurrir a `t-raw`

```
<!-- Incorrecto - t-raw fue eliminado en OWL 2 -->
<div t-raw="state.rawHtml"/>

<!-- Correcto - marca explícitamente el string como seguro, luego usa t-out -->
<!-- en el componente: this.state.rawHtml = markup("<b>Negrita</b>"); -->
<div t-out="state.rawHtml"/>
```

`t-out` renderiza texto plano de forma segura por defecto; solo renderiza HTML cuando el valor fue envuelto explícitamente en `markup()` en JavaScript. Este requisito de dos pasos existe específicamente para prevenir XSS accidental.

Error 3: Confundir `t-att-class` con `t-attf-class`

```
<!-- t-att-class espera una expresión JS: un string O un objeto {nombreClase: booleano}
    ↪ -->
<div t-att-class="{ 'active': state.isActive }"/>
```

```
<!-- t-attf-class espera un string plano con marcadores {{ }} para interpolación -->
<div t-attf-class="badge bg-{{state.color}}"/>
```

Confundirlos—pasar un objeto a `t-attf-class` o una plantilla `{{ }}` a `t-att-class`—lanzará un error o renderizará silenciosamente la clase equivocada.

Error 4: Mutar Estado que Aún No es Reactivo

```
<!-- La plantilla solo se vuelve a renderizar cuando cambia una propiedad *reactiva*.
    ↪ -->
<!-- Si state.tasks nunca fue envuelto en useState() en el setup() del componente, -->
<!-- hacer push aquí no tiene efecto visible en la página. -->
```

Si una lista u objeto deja de actualizar la interfaz, lo primero que hay que revisar es si fue creado con `useState(...)` en el componente (lo veremos en el próximo capítulo)—no si la sintaxis de la plantilla está mal.

Ejercicios

1. **Alterna una clase a mano.** Añade un nuevo `<button>` a la sección del contador que llame a un método `toggleHighlight()`, y usa `t-att-class` (sintaxis de objeto) para añadir una clase `bg-warning` a la tarjeta solo mientras `state.highlighted` sea `true`.
2. **Arregla la clave faltante.** Toma la lista de tareas del "Ejemplo Completo" de arriba, elimina `t-key="task.id"`, y recarga la página después de reordenar `state.tasks` desde la consola. Observa qué se rompe, luego restaura `t-key` y confirma que se arregla.
3. **Construye un bucle filtrado.** Añade una nueva sección que liste solo las tareas con `priority === "high"`, usando un `t-foreach` combinado con un `t-if` dentro (como se muestra en "Patrones Avanzados de Bucles").

¿Qué Sigue?

Las plantillas solo pueden mostrar lo que tu componente les da. El Capítulo 7 profundiza en dónde vive realmente ese dato — `useState` y el sistema de reactividad de OWL — para que entiendas *por qué* cambiar un valor dispara automáticamente todo lo que acabas de aprender sobre re-renderizado.

Capítulo 7: Estado y Reactividad: Haciendo Componentes Dinámicos

> **¿Por qué este capítulo?** Este es el modelo mental sobre el que se construye todo lo demás en OWL — props, hooks, servicios, incluso las signals de OWL 3 (Capítulo 18) son variaciones de "cambia el dato, deja que el framework maneje el DOM".
>
> **¿Qué trata de resolver Odoo con esto?** Reemplazar la sincronización manual del DOM (encontrar el elemento, actualizarlo, esperar no haberte olvidado de algo) por un sistema donde la UI es una función garantizada y automática de tus datos.
>
> **Aplicación en la vida real:** Cualquier dashboard, wizard o formulario interactivo que construyas profesionalmente depende de entender bien esto — la mayoría de los tickets de soporte de "la UI no se actualizó" se rastrean hasta un malentendido de exactamente lo que cubre este capítulo.

Este capítulo es el más importante de todo el libro. Si entiendes esto, entiendes la filosofía central de todos los frameworks de UI modernos. Estamos a punto de descubrir la "magia" que hace a OWL tan poderoso: **la reactividad**.

La reactividad es el mecanismo por el cual la interfaz de usuario se actualiza automáticamente cuando los datos subyacentes cambian. No le dices a la UI *cómo* cambiar; simplemente cambias los datos, y la UI reacciona. Este es el corazón del paradigma declarativo que discutimos en el Capítulo 1.

Piensa en ello como una hoja de cálculo: cuando cambias el valor de una celda, todas las fórmulas que dependen de esa celda automáticamente se recalculan y actualizan. OWL trae este mismo comportamiento de actualización automática a las interfaces web.

El Cerebro del Componente: El Método `setup()`

Hasta ahora, nuestros componentes han sido simples planos. Para darles memoria, comportamiento y la capacidad de cambiar con el tiempo, necesitamos introducir un método especial llamado `setup()`.

El método `setup()` es parte del ciclo de vida de un componente. Se ejecuta solo una vez, justo después de que el componente es creado pero antes de que sea renderizado en la pantalla. Es el lugar perfecto para:

- Inicializar los datos internos del componente (estado)
- Configurar servicios y conexiones externas
- Registrar event listeners
- Definir valores computados
- Configurar el comportamiento del componente

Comencemos con un ejemplo simple y construyamos complejidad:

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

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

  setup() {
    console.log("¡El componente se está configurando!");
    console.log("Esto se ejecuta una vez, antes del primer render");

    // Inicializaremos nuestro estado aquí
    // Configuraremos servicios aquí
    // Registraremos event listeners aquí
  }
}
```

El Hook `useState`: Dándole Memoria a tu Componente

El **estado** de un componente es un objeto que contiene todos los datos que el componente necesita recordar durante su vida—cosas como valores de entrada de formularios, estados de carga, preferencias del usuario, o una lista de elementos a mostrar.

Pero aquí está el punto crucial: un objeto JavaScript simple no es suficiente. Si cambias una propiedad en un objeto simple, OWL no tiene forma de saber que necesita re-renderizar la UI. La interfaz permanecería congelada, mostrando datos obsoletos.

Para crear un estado "reactivo" que OWL pueda monitorear para cambios, usamos una herramienta especial llamada el **hook `useState`**.

¿Qué es un Hook?

Un "hook" en OWL es una función que te permite "engancharte" a las características poderosas de OWL como la gestión de estado, eventos del ciclo de vida, o servicios. Los hooks solo pueden ser llamados dentro del método `setup()` y le dan superpoderes a tu componente.

Usando `useState`

Así es como creas estado reactivo:

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

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

  setup() {
    // 2. Crear un objeto de estado reactivo
    this.state = useState({
      counter: 0,
      userName: "Invitado",
      isVisible: true,
      items: []
    });
```

```
    console.log("Estado inicial:", this.state);
  }
}
```

Ahora, `this.state` no es solo un objeto regular—es un **proxy reactivo**. OWL lo está monitoreando activamente. Cualquier cambio hecho a `this.state.counter`, `this.state.userName`, o cualquier otra propiedad automáticamente desencadenará un re-render del componente.

La Magia Detrás de `useState`

Cuando llamas a `useState`, OWL:

1. **Envuelve tu objeto** en un Proxy que intercepta todo acceso y modificación de propiedades
2. **Rastrea dependencias** entre tu plantilla y las propiedades del estado
3. **Programa re-renders** cuando el estado cambia, pero los agrupa por rendimiento
4. **Actualiza solo lo que cambió** en lugar de re-renderizar todo el componente

¡Por eso las aplicaciones OWL son tan rápidas y responsivas!

El Momento Mágico: Modificando Estado desde Eventos de Usuario

Veamos la reactividad en acción. Crearemos un componente con múltiples elementos interactivos para demostrar cómo los cambios de estado impulsan las actualizaciones de la UI.

1. JavaScript Mejorado (`hello_world.js`)

`setup()` inicializa un único objeto reactivo con todo lo que la plantilla necesitará: un contador, un nombre de usuario editable, un indicador de tema y una lista de tareas.

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

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

  setup() {
    this.state = useState({
      counter: 0,
      userName: "Usuario Anónimo",
      theme: "light",
      todos: [
        { id: 1, text: "Aprender reactividad OWL", completed: false },
        { id: 2, text: "Construir componentes increíbles", completed: false }
      ],
      isEditing: false,
      newTodoText: ""
    });

    console.log("Componente inicializado con estado:", this.state);
  }

  // ... más métodos abajo, todavía dentro de la misma clase
```

Estos métodos solo mutan `this.state`—nunca el DOM directamente. Los métodos del contador incrementan, decrementan o reinician un solo número; el método de tema alterna un string entre dos

valores; los métodos de nombre de usuario alternan un flag `isEditing` y leen de un evento de input:

```
  // Métodos del contador
  increment() {
    this.state.counter++;
    console.log("Contador incrementado a:", this.state.counter);
  }

  decrement() {
    this.state.counter--;
    console.log("Contador decrementado a:", this.state.counter);
  }

  reset() {
    this.state.counter = 0;
    console.log("Contador reiniciado a:", this.state.counter);
  }

  // Métodos de tema
  toggleTheme() {
    this.state.theme = this.state.theme === "light" ? "dark" : "light";
    console.log("Tema cambiado a:", this.state.theme);
  }

  // Métodos de nombre de usuario
  startEditing() {
    this.state.isEditing = true;
  }

  saveUserName() {
    if (this.state.userName.trim()) {
      this.state.isEditing = false;
      console.log("Nombre de usuario guardado:", this.state.userName);
    }
  }

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

Los métodos de todos usan métodos de array (`push`, `find`, `splice`) directamente sobre el array reactivo `todos`—la reactividad de OWL también envuelve arrays, así que mutarlos en el lugar sigue disparando un re-render. Los tres `get` accessors al final son propiedades computadas: no se guardan en el estado, se recalculan desde él cada vez que la plantilla los lee:

```
  // Métodos de todos
  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("Nuevo todo añadido:", newTodo);
    }
```

```
  }

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

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

  // Propiedades computadas (calculadas desde el estado)
  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. Plantilla Integral (`hello_world.xml`)

Veremos esta plantilla en tres partes. Primero, el encabezado: alterna entre un saludo de solo lectura y un input editable según `state.isEditing`, más un botón para alternar el tema:

```
<?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' }">

      <!-- Encabezado con Nombre de Usuario -->
      <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">
                ¡Hola, <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="Ingresa tu nombre"/>
            <button class="btn btn-success" t-on-click="saveUserName">
              <i class="fa fa-check"></i> Guardar
            </button>
          </div>
        </t>
      </div>

      <!-- Alternar Tema -->
      <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>
        Tema <t t-esc="state.theme === 'dark' ? 'Claro' : 'Oscuro'"/>
      </button>
    </div>
  </div>
```

Ahora, el contador (misma plantilla, continuación). Nota el `t-att-class` de tres vías sobre el número, y la cadena `t-if`/`t-elif`/`t-else` que elige un mensaje distinto según si el contador es cero, positivo o negativo:

```
<!-- Sección del Contador -->
<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> Decrementar
        </button>
        <button class="btn btn-secondary"
                t-att-disabled="state.counter === 0"
                t-on-click="reset">
          Reiniciar
        </button>
```

```
        <button class="btn btn-success" t-on-click="increment">
          <i class="fa fa-plus"></i> Incrementar
        </button>
      </div>

      <!-- Mensajes dinámicos basados en el valor del contador -->
      <div class="mt-3">
        <t t-if="state.counter === 0">
          <p class="text-muted mb-0">¡Comienza a contar!</p>
        </t>
        <t t-elif="state.counter > 0">
          <p class="text-success mb-0">
            <t t-if="state.counter === 1">¡Has hecho clic una vez!</t>
            <t t-else="">¡Has hecho clic <t t-esc="state.counter"/> veces!</t>
          </p>
        </t>
        <t t-else="">
          <p class="text-danger mb-0">Yendo negativo: <t
↪  t-esc="Math.abs(state.counter)"/> bajo cero</p>
        </t>
      </div>
    </div>
  </div>
</div>
```

Finalmente, la lista de tareas y un pequeño panel de depuración, y cerramos cada etiqueta abierta arriba. La lista combina `t-foreach` con un `t-if` anidado (para el banner de "todo completado") y un `t-else` para el estado vacío:

```
<!-- Sección de Lista de Todos -->
<div class="todo-section">
  <div class="d-flex justify-content-between align-items-center mb-3">
    <h3>Lista de Tareas</h3>
    <div class="todo-stats">
      <span class="badge bg-primary me-2">
        <t t-esc="remainingCount"/> pendientes
      </span>
      <span class="badge bg-success">
        <t t-esc="completedCount"/> completadas
      </span>
    </div>
  </div>

  <!-- Añadir Nuevo Todo -->
  <div class="add-todo mb-3">
    <div class="input-group">
      <input type="text"
             class="form-control"
             placeholder="¿Qué necesitas hacer?"
             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> Añadir Tarea
```

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

    <!-- Lista de Todos -->
    <t t-if="state.todos.length > 0">
      <!-- Mensaje de éxito cuando todo esté completado -->
      <t t-if="allCompleted">
        <div class="alert alert-success">
          <i class="fa fa-trophy"></i>
          ¡Felicidades! ¡Has completado todas tus tareas!
        </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 hay tareas aún</h4>
        <p class="text-muted">¡Añade tu primera tarea arriba para empezar!</p>
      </div>
    </t>
  </div>

  <!-- Información de Depuración -->
  <div class="debug-info mt-4 p-3 border rounded" t-att-class="{ 'border-light':
    ↪  state.theme === 'dark' }">
    <h5>Información de Depuración</h5>
```

```
    <small class="text-muted">
      <strong>Resumen del Estado:</strong><br/>
      Contador: <t t-esc="state.counter"/>,
      Tema: <t t-esc="state.theme"/>,
      Todos: <t t-esc="state.todos.length"/>,
      Editando: <t t-esc="state.isEditing"/>
    </small>
  </div>
  </div>
  </t>

</templates>
```

3. ¡Prueba la Magia!

Ahora, actualiza tu módulo y ve el componente. Aquí está lo que experimentarás:

1. **Haz clic en los botones del contador**: Observa el número cambiar instantáneamente, junto con el color y el mensaje
2. **Alterna el tema**: Ve toda la interfaz cambiar entre modos claro y oscuro inmediatamente
3. **Edita tu nombre**: Nota cómo aparece el campo de entrada y el encabezado se actualiza en tiempo real
4. **Añade todos**: Observa nuevos elementos aparecer en la lista instantáneamente
5. **Marca/desmarca todos**: Ve las estadísticas de finalización actualizarse automáticamente
6. **Elimina todos**: Observa elementos desaparecer y las estadísticas recalcularse

Lo notable: Nunca escribimos una sola línea de código para actualizar el DOM directamente. Solo cambiamos los datos, y OWL manejó todas las actualizaciones visuales automáticamente.

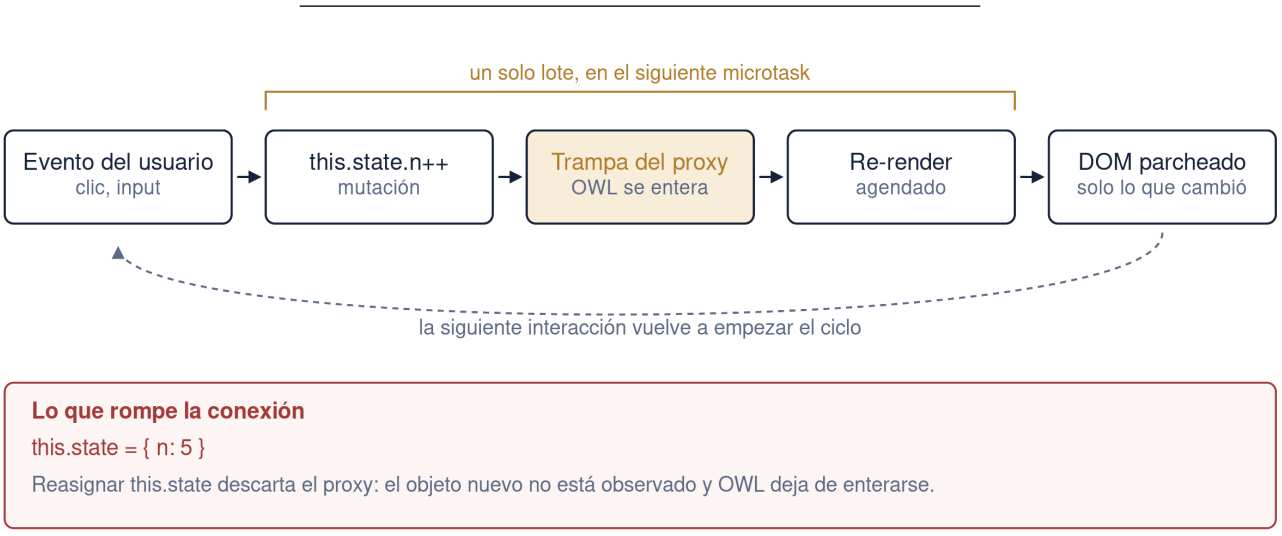


Figura 2: El ciclo de reactividad, y la asignación que lo rompe.

Entendiendo la Reactividad en Profundidad

¿Qué Desencadena un Re-render?

OWL re-renderiza tu componente cuando:

1. **Las propiedades del estado cambian**: `this.state.counter++`
2. **Mutaciones de arrays**: `this.state.items.push()`, `this.state.items.splice()`

3. **Actualizaciones de propiedades de objetos**: `this.state.user.name = "Nuevo Nombre"`
4. **Cambios de propiedades anidadas**: `this.state.settings.theme = "dark"`

¿Qué NO Desencadena un Re-render?

```
// Esto no funcionará - reemplazar el objeto de estado
this.state = { counter: 5 }; // OWL pierde la conexión reactiva

// Esto no funcionará - modificar objetos no reactivos
const plainObject = { count: 0 };
plainObject.count++; // No es reactivo

// Estos funcionan - modificar el estado reactivo
this.state.counter = 5;           // Asignación directa
this.state.items.push(newItem);   // Mutación de array
delete this.state.temporaryData;  // Eliminación de propiedad
```

Rendimiento y Agrupación

OWL es inteligente sobre el rendimiento. No re-renderiza después de cada cambio de estado. En su lugar:

1. **Agrupación**: Múltiples cambios de estado en el mismo ciclo de ejecución JavaScript se agrupan en un solo re-render
2. **Diffing Eficiente**: Solo las partes del DOM que realmente cambiaron se actualizan
3. **Rastreo de Dependencias**: OWL sabe exactamente qué partes de la plantilla dependen de qué propiedades del estado

```
// Esto causará solo UN re-render, no tres
updateMultipleValues() {
  this.state.counter++;        // Cambio 1
  this.state.userName = "Bob"; // Cambio 2
  this.state.theme = "dark";   // Cambio 3
  // OWL los agrupa y re-renderiza una vez
}
```

Patrones de Estado Avanzados

Propiedades Computadas con Getters

Crea valores derivados que se actualizan automáticamente cuando cambian sus dependencias:

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

// Estos getters automáticamente "recomputan" cuando el estado cambia
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);
}
```

Usar en plantilla:

```
<h3>¡Bienvenido, <t t-esc="fullName"/>!</h3>
<p>Tienes <t t-esc="expensiveItems.length"/> elementos caros por valor de $<t
    ↪  t-esc="totalValue"/></p>
```

Estructuras de Estado Complejas

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

// Todos estos desencadenan re-renders:
updateUserName(newName) {
  this.state.user.profile.name = newName; // Cambio de propiedad anidada
}

changeTheme(theme) {
  this.state.user.profile.preferences.theme = theme; // Cambio anidado profundo
}

addError(error) {
  this.state.application.errors.push(error); // Mutación de array
```

```
}
```

Validación de Estado y Restricciones

```
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) {
  // Restringir contador entre 0 y 100
  this.state.counter = Math.max(0, Math.min(100, value));
}
```

Escapando de la Reactividad: `markRaw` y `toRaw`

`useState` envuelve el objeto que le des en un proxy reactivo, para que OWL pueda notar cuándo se lee una propiedad (para rastrearla) y cuándo se escribe (para disparar un re-render). Normalmente eso es exactamente lo que quieres—pero no siempre.

A veces un objeto no debería volverse reactivo en absoluto: un conjunto de datos grande que nunca mutarás en el lugar, o una instancia de una librería de terceros (una librería de gráficos, un widget de mapas) que gestiona su propio estado interno y no espera estar envuelta en un proxy. Envolverla de todos modos desperdicia rendimiento rastreando claves que nadie lee, e incluso puede romper librerías que hacen comparaciones de identidad (`===`) internamente, ya que un proxy nunca es `===` al objeto que envuelve.

`markRaw(objeto)` le dice al sistema de reactividad de OWL que deje un objeto en paz. Si en algún momento queda anidado dentro de un objeto reactivo, se guarda y se devuelve tal cual—intacto, sin observar:

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

setup() {
  // chartInstance gestiona su propio estado interno; nunca queremos que
  // OWL observe sus propiedades ni la envuelva en un proxy.
  const chartInstance = markRaw(new ThirdPartyChart());

  this.state = useState({
    counter: 0,
    chart: chartInstance,
```

```
  });

  // Mutar las propias propiedades de chart no disparará un re-render-tal como se busca.
  this.state.chart.zoomLevel = 2;
}
```

`toRaw(objetoReactivo)` hace el trabajo opuesto: dado un objeto reactivo (o una parte de uno), devuelve el objeto original, sin proxy, que está por debajo. Esto es útil para comparaciones de identidad (`obj === toRaw(state.obj)` puede ser `true` incluso cuando `obj === state.obj` es `false`, ya que `state.obj` es un proxy) y para depurar—loguear un objeto sin proxy es más fácil de leer que loguear un proxy.

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

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

No recurrirás a ninguno de los dos muy seguido, pero vale la pena conocerlos para el día que integres un gráfico, un mapa, o cualquier otra librería que gestione su propio estado.

Errores Comunes y Soluciones

Error 1: Perder Reactividad

```
// Incorrecto - Esto rompe la reactividad
someMethod() {
  this.state = { items: [1, 2, 3] }; // Reemplaza completamente el objeto reactivo
}

// Correcto - Modifica propiedades, no reemplaces el objeto
someMethod() {
  this.state.items = [1, 2, 3]; // Modifica el estado reactivo
}
```

Error 2: Olvidar `useState`

```
// Incorrecto - objeto plano: los cambios son invisibles para OWL, la UI nunca se
    ↪ actualiza
setup() {
  this.state = { counter: 0 };
}

// Correcto - objeto reactivo: los cambios disparan re-renders
setup() {
  this.state = useState({ counter: 0 });
}
```

Error 3: Actualizaciones de Estado Asíncronas

```
// Posibles condiciones de carrera
async fetchData() {
  this.state.loading = true;
  const data = await this.service.getData();
```

```
  this.state.data = data;
  this.state.loading = false; // ¿Y si el componente fue destruido mientras tanto?
}

// Mejor enfoque: el helper status() de OWL te dice si el componente sigue vivo
import { status } from "@odoo/owl";

async fetchData() {
  this.state.loading = true;
  try {
    const data = await this.service.getData();
    if (status(this) === "destroyed") return; // El componente ya no existe, detente
        ↪ aquí
    this.state.data = data;
  } finally {
    if (status(this) !== "destroyed") {
      this.state.loading = false;
    }
  }
}
```

Estado vs Props: Cuándo Usar Cada Uno

Usa useState cuando: - Los datos pertenecen a este componente - Los datos cambian basándose en interacciones del usuario en este componente - Los datos son temporales o específicos de la sesión - Necesitas modificar los datos

Usa Props cuando: - Los datos vienen de un componente padre - Los datos son configuración o ajustes - Múltiples componentes necesitan mostrar los mismos datos - Los datos son de solo lectura para este componente

```
// Estado: Datos propiedad del componente
this.state = useState({
  inputValue: "",        // Usuario escribiendo en este componente
  isLoading: false,      // Estado de carga de este componente
  showModal: false       // Estado de UI de este componente
});

// Props: Datos proporcionados por el padre
static props = {
  userId: { type: Number },      // Configuración del padre
  readonly: { type: Boolean },   // Configuración de comportamiento
  initialData: { type: Array }   // Datos a mostrar
};
```

Depurando Estado Reactivo

Consejos de Herramientas de Desarrollo del Navegador

1. **Owl DevTools**: Instala la extensión de navegador de Owl para inspección de componentes y estado

2. **Console Logging**: Añade declaraciones `console.log` estratégicas en métodos
3. **Breakpoints**: Establece breakpoints en métodos que cambian el estado
4. **Snapshots de Estado**: Registra todo el estado en momentos clave

```
// Método auxiliar de depuración
debugState(action) {
  console.group(`Cambio de Estado: ${action}`);
  console.log("Estado actual:", JSON.parse(JSON.stringify(this.state)));
  console.log("Componente:", this);
  console.groupEnd();
}

// Usar en métodos
increment() {
  this.state.counter++;
  this.debugState("increment");
}
```

Escenarios Comunes de Depuración

```
<!-- Añadir depuración temporal a plantillas -->
<div class="debug-panel" style="position: fixed; top: 10px; right: 10px; background:
    ↪  yellow; padding: 10px;">
  <pre t-esc="JSON.stringify(state, null, 2)"/>
</div>

// Método para reiniciar estado para pruebas
resetToInitialState() {
  Object.assign(this.state, {
    counter: 0,
    userName: "Invitado",
    todos: [],
    // ... otros valores iniciales
  });
}
```

La Mentalidad Declarativa

Este capítulo te introdujo al cambio fundamental de pensamiento que requieren los frameworks de UI modernos:

En lugar de pensar: "Cuando el usuario haga clic en este botón, necesito encontrar el elemento contador en el DOM y actualizar su texto"

Piensa: "Cuando el usuario haga clic en este botón, necesito actualizar el valor del contador en mi estado, y la UI automáticamente reflejará este cambio"

Este enfoque declarativo hace que tu código sea:

- **Más predecible**: La UI siempre es una función de tu estado
- **Más fácil de depurar**: Puedes inspeccionar el estado para entender la UI
- **Más mantenible**: No hay manipulación manual del DOM de la que hacer seguimiento
- **Más testeable**: Puedes probar cambios de estado sin involucrar el DOM

TL;DR: Envuelve los datos en `useState()` para hacerlos reactivos; muta sus propiedades directamente (`state.counter++`) en vez de reasignar todo el objeto, y la plantilla se vuelve a renderizar automáticamente — cambias el dato, OWL actualiza el DOM.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch7_ex1`. Las instrucciones de instalación están en el README del repositorio.

Ejercicios

1. **Añade un botón de reinicio total.** Añade un botón que reinicie `counter` a `0`, `todos` a un array vacío, y `userName` de vuelta a `"Usuario Anónimo"`—todo en un solo método—y confirma que toda la UI se actualiza a la vez.
2. **Provoca el Error 1 a propósito.** En `reset()`, reemplaza temporalmente `this.state.counter = 0` por `this.state = { counter: 0 }` (como se muestra en el Error 1 arriba). Haz clic en el botón y observa que la UI silenciosamente deja de actualizarse. Deshaz el cambio y confirma que vuelve a funcionar.
3. **Añade una propiedad computada.** Añade un nuevo getter, `oldestPendingTodo`, que devuelva la primera tarea en `state.todos` donde `completed` sea `false` (o `null` si no hay ninguna), y muéstrala en la plantilla.

¿Qué Sigue?

El estado reactivo es genial para datos que un componente posee, pero las aplicaciones reales están hechas de componentes que se comunican entre sí. El Capítulo 8 cubre las **props**, el mecanismo para pasar datos de un componente padre hacia sus hijos.

Capítulo 8: Props: Comunicación de Padre a Hijo

> **¿Por qué este capítulo?** Casi todo reporte de bug que recibo que empieza con "el widget muestra datos desactualizados" termina en un componente que lee o modifica datos que nunca debió poseer. Las props son la regla que previene esa clase de bug por completo.
>
> **¿Qué trata de resolver Odoo con esto?** El propio backend de Odoo — vistas de lista alimentando widgets de celda, vistas de formulario alimentando widgets de campo — está construido exactamente sobre este contrato de padre a hijo. Entender props es lo que te permite extender esas vistas en vez de pelear contra ellas.
>
> **Aplicación en la vida real:** Cualquier dashboard personalizado donde un panel de filtros controla lo que muestra un gráfico o tabla es este patrón en producción: el padre posee los filtros, los hijos solo renderizan lo que reciben.

Hasta ahora, nuestros componentes han vivido en aislamiento, gestionando su propio estado y renderizando su propio contenido. Pero en una aplicación real, los componentes están anidados unos dentro de otros, formando una estructura similar a un árbol. Un componente `Dashboard` podría contener múltiples componentes `Widget`. Una `TodoList` podría contener múltiples componentes `TodoItem`. Un `UserProfile` podría contener componentes `Avatar`, `ContactInfo`, y `PreferenceSettings`.

Esta composición de componentes plantea una pregunta crítica: ¿cómo se comunica un componente padre con sus hijos? ¿Cómo le dice la `TodoList` a cada `TodoItem` qué texto mostrar y si está completado? ¿Cómo pasa un `Dashboard` datos de configuración a sus widgets?

La respuesta son las **props** (abreviatura de propiedades).

Las props son el mecanismo principal para pasar datos de un componente padre hacia abajo a sus hijos. Piensa en ellas como argumentos que pasas a una función—el padre proporciona los datos, y el hijo los recibe y los usa. Esto crea un flujo de datos predecible y unidireccional que hace que tu aplicación sea fácil de entender, depurar y mantener.

Entendiendo el Árbol de Componentes

Antes de profundizar en las props, visualicemos cómo los componentes forman una jerarquía:

```
Dashboard (Padre)
|-- Header (Hijo de Dashboard)
|   |-- Logo (Hijo de Header)
|   |-- UserMenu (Hijo de Header)
|-- Sidebar (Hijo de Dashboard)
|   |-- Navigation (Hijo de Sidebar)
```

```
|   |-- QuickActions (Hijo de Sidebar)
|-- MainContent (Hijo de Dashboard)
    |-- TodoList (Hijo de MainContent)
    |   |-- TodoItem (Hijo de TodoList)
    |   |-- TodoItem (Hijo de TodoList)
    |   |-- AddTodoForm (Hijo de TodoList)
    |-- Statistics (Hijo de MainContent)
```

En este árbol: - Los **padres** necesitan pasar datos hacia abajo a sus **hijos** - Los **hijos** reciben estos datos como **props** - Los datos fluyen en una dirección: hacia abajo del árbol

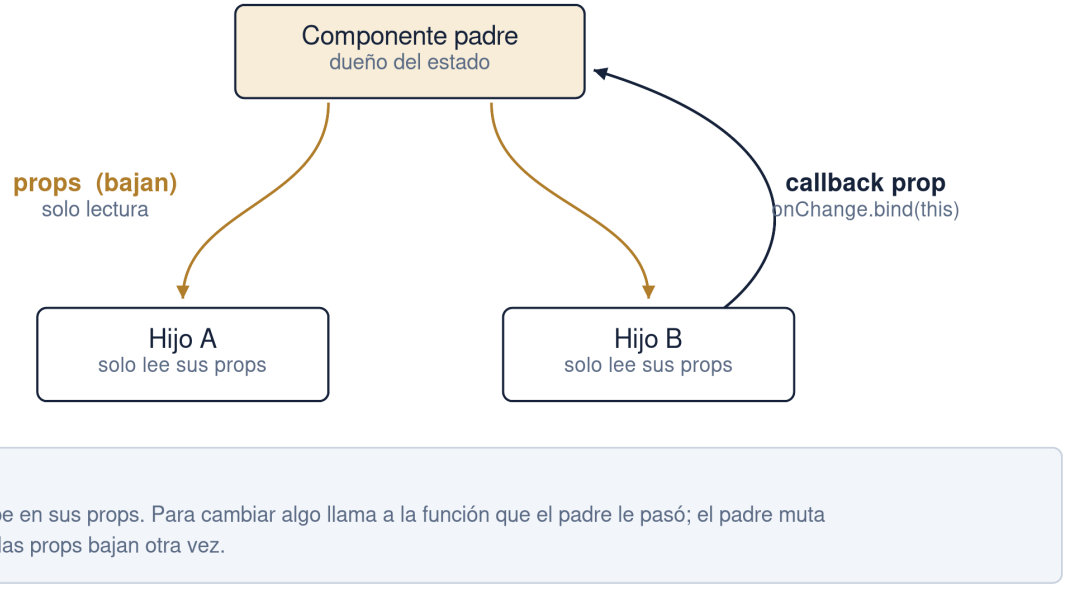


Figura 3: Las props bajan, los callbacks vuelven a subir.

Definiendo Props en el Componente Hijo

Antes de que un componente hijo pueda recibir props, debe declarar lo que espera recibir. Esto se hace usando una definición `static props` en la clase del componente hijo. Definir props se considera una mejor práctica porque:

- **Documentación**: Muestra claramente qué datos necesita el componente
- **Validación**: OWL puede verificar que se estén pasando los tipos de datos correctos
- **Seguridad de Desarrollo**: Ayuda a detectar bugs temprano en el desarrollo
- **Soporte de IDE**: Mejor autocompletado y detección de errores

Creemos un componente `UserCard` integral que demuestre varios tipos de props:

En `user_card.js`:

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

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

  // Definición integral de props
  static props = {
```

```
    // Props requeridas
    user: {
      type: Object,
      shape: {
        id: Number,
        name: String,
        email: String,
        avatar: { type: String, optional: true },
        role: String,
        isActive: Boolean,
        lastLogin: { type: String, optional: true }
      }
    },

    // Props opcionales con valores por defecto
    showDetails: { type: Boolean, optional: true },
    size: { type: String, optional: true }, // 'small', 'medium', 'large'
    theme: { type: String, optional: true },

    // Props de función (callbacks)
    onUserClick: { type: Function, optional: true },
    onEditUser: { type: Function, optional: true },
    onDeleteUser: { type: Function, optional: true },

    // Props avanzadas
    customActions: { type: Array, optional: true },
    permissions: { type: Array, optional: true },

    // Validación con validador personalizado
    priority: {
      type: String,
      optional: true,
      validate: (value) => ['low', 'medium', 'high'].includes(value)
    }
  };

  // Valores por defecto para props opcionales
  static defaultProps = {
    showDetails: false,
    size: 'medium',
    theme: 'light',
    customActions: [],
    permissions: [],
    priority: 'medium'
  };

  setup() {
    // Las props están disponibles como this.props
    console.log("Props de UserCard:", this.props);
  }

  // Método para manejar clic interno de la tarjeta
  handleCardClick() {
    if (this.props.onUserClick) {
      this.props.onUserClick(this.props.user);
```

```
    }
  }

  // Método para manejar acción de editar
  handleEdit() {
    if (this.props.onEditUser) {
      this.props.onEditUser(this.props.user.id);
    }
  }

  // Método para manejar acción de eliminar
  handleDelete() {
    if (this.props.onDeleteUser) {
      this.props.onDeleteUser(this.props.user.id);
    }
  }

  // Propiedades computadas basadas en 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;
  }
}
```

La plantilla correspondiente (`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">
```

```
    <!-- Encabezado de la Tarjeta -->
    <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>

        <!-- Nombre y Estado -->
        <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 ? 'Activo' : 'Inactivo'"/>
          </small>
        </div>
      </div>

      <!-- Botones de Acción -->
      <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>
```

```
      <!-- Cuerpo de la Tarjeta (Detalles Opcionales) -->
      <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>Rol:</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>Último Login:</strong>
                <small class="text-muted">
                  <t t-esc="props.user.lastLogin"/>
                </small>
              </p>
            </t>
          </div>

          <!-- Acciones Personalizadas -->
          <t t-if="props.customActions.length > 0">
            <div class="custom-actions mt-3">
              <h6>Acciones Rápidas:</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>
```

Pasando Props desde el Componente Padre

Ahora creemos un componente padre que use nuestro `UserCard` y demuestre varias formas de pasar props.

JavaScript del Componente Padre (`user_dashboard.js`):

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

```
import { UserCard } from "../user_card/user_card"; // Importar el componente hijo

export class UserDashboard extends Component {
  static template = "my_module.UserDashboard";
  static components = { UserCard }; // Registrar el componente hijo

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

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

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

    // Definir acciones personalizadas que se pasarán como props
    this.customActions = [
      {
        id: 'message',
        label: 'Enviar Mensaje',
        icon: 'fa fa-envelope',
        handler: this.sendMessageToUser.bind(this)
```

```
      },
      {
        id: 'schedule',
        label: 'Programar Reunión',
        icon: 'fa fa-calendar',
        handler: this.scheduleMeetingWith.bind(this)
      }
    ];
  }
```

`setup()` solo hace dos cosas: coloca la lista de usuarios y la configuración de vista en estado reactivo, y prepara un array `customActions` cuyos handlers están pre-vinculados con `.bind(this)` para poder llamarse de forma segura desde el hijo más adelante.

A continuación vienen los métodos que se entregarán a `UserCard` como callback props — esta es la mitad de "la comunicación fluye hacia arriba a través de llamadas a funciones" del patrón:

```
  // Manejadores de eventos que se pasarán como callbacks de props
  onUserClick(user) {
    console.log("Usuario clickeado:", user);
    this.state.selectedUserId = user.id;

    // Mostrar detalles del usuario en un modal o sidebar
    this.showUserDetails(user);
  }

  onEditUser(userId) {
    console.log("Editar usuario:", userId);
    const user = this.state.users.find(u => u.id === userId);
    if (user) {
      // Abrir modal de edición o navegar al formulario de edición
      this.openEditModal(user);
    }
  }

  onDeleteUser(userId) {
    console.log("Eliminar usuario:", userId);
    if (confirm("¿Estás seguro de que quieres eliminar este usuario?")) {
      this.state.users = this.state.users.filter(u => u.id !== userId);
    }
  }

  // Manejadores de acciones personalizadas
  sendMessageToUser(user) {
    console.log("Enviando mensaje a:", user.name);
    // Abrir interfaz de mensajería
  }

  scheduleMeetingWith(user) {
    console.log("Programando reunión con:", user.name);
    // Abrir interfaz de programación de calendario
  }
```

El resto es manejo de estado de UI local — alternar configuraciones de vista y calcular listas derivadas — nada de esto involucra props, y ese es el punto: solo el padre necesita saber *cómo* se almacenan y filtran los usuarios.

```
  // Manejadores de cambios de configuración
  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';
  }

  // Métodos auxiliares
  showUserDetails(user) {
    // Implementación para mostrar detalles del usuario
    console.log("Mostrando detalles para:", user);
  }

  openEditModal(user) {
    // Implementación para abrir modal de edición
    console.log("Abriendo modal de edición para:", user);
  }

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

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

Plantilla del Componente Padre (`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">

      <!-- Encabezado del Dashboard -->
      <div class="dashboard-header mb-4">
        <div class="d-flex justify-content-between align-items-center">
          <div>
            <h2>Dashboard de Usuarios</h2>
            <p class="text-muted">
              <t t-esc="activeUsers.length"/> activos,
              <t t-esc="inactiveUsers.length"/> usuarios inactivos
            </p>
          </div>

          <!-- Controles de Vista -->
          <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
                      ? 'Ocultar Detalles' : 'Mostrar Detalles'"/>
          </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')">
            Pequeño
          </button>
          <button class="btn btn-sm"
                  t-att-class="state.viewSettings.cardSize === 'medium'
                               ? 'btn-secondary' : 'btn-outline-secondary'"
                  t-on-click="() => this.changeCardSize('medium')">
            Mediano
          </button>
          <button class="btn btn-sm"
                  t-att-class="state.viewSettings.cardSize === 'large'
                               ? 'btn-secondary' : 'btn-outline-secondary'"
                  t-on-click="() => this.changeCardSize('large')">
            Grande
          </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>
          Tema <t t-esc="state.viewSettings.theme === 'dark' ? 'Claro' : 'Oscuro'"/>
        </button>
      </div>
    </div>
  </div>

  <!-- Cuadrícula de Tarjetas de Usuario -->
  <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">
            <!-- ¡Aquí es donde se pasan las props! -->
            <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 === 'Administrador' ? 'high' : 'medium'"
                  />
                </div>
              </div>
            </t>
          </div>
        </div>

        <!-- Estado Vacío -->
        <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 se encontraron usuarios</h4>
            <p class="text-muted">¡Añade algunos usuarios para verlos aquí!</p>
          </div>
        </t>

        <!-- Información del Usuario Seleccionado (si existe) -->
        <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>Usuario Seleccionado: <t t-esc="selectedUser.name"/></h5>
            <p class="mb-0">Haz clic en otra tarjeta de usuario para seleccionarla.</p>
          </div>
        </t>
      </div>
    </t>

</templates>
```

Entendiendo la Sintaxis de Paso de Props

Analicemos las diferentes formas de pasar props:

1. Valores Estáticos

```
<!-- Pasando una cadena literal -->
<UserCard title="'Perfil de Usuario'"/>

<!-- Pasando un número -->
<UserCard maxItems="10"/>

<!-- Pasando un booleano -->
<UserCard readonly="true"/>
```

Importante: Nota las comillas simples dentro de las comillas dobles para cadenas. El valor del atributo es una expresión JavaScript, por lo que `"'Hola'"` se evalúa como la cadena `"Hola"`.

2. Valores Dinámicos del Estado

```
<!-- Pasando propiedades del estado -->
<UserCard user="state.currentUser"/>
<UserCard showDetails="state.viewSettings.showDetails"/>
<UserCard theme="state.theme"/>
```

3. Valores Computados

```
<!-- Pasando el resultado de una expresión -->
<UserCard priority="user.role === 'Administrador' ? 'high' : 'medium'"/>
<UserCard isSelected="state.selectedUserId === user.id"/>
<UserCard canEdit="state.permissions.includes('edit') and user.isActive"/>
```

4. Referencias de Función

```
<!-- Pasando referencias de métodos (callbacks) -->
<!-- El sufijo .bind vincula la función al componente padre,
     para que `this` funcione correctamente cuando el hijo la llame -->
<UserCard onUserClick.bind="onUserClick"/>
<UserCard onEditUser.bind="onEditUser"/>
<UserCard onDeleteUser.bind="onDeleteUser"/>
```

5. Objetos y Arrays

```
<!-- Pasando objetos complejos -->
<UserCard user="user" customActions="customActions"/>
<UserCard settings="state.userSettings"/>
<UserCard permissions="['read', 'write', 'delete']"/>
```

Patrones Avanzados de Props

Props Condicionales

```
<!-- Solo pasar ciertas props bajo condiciones -->
<UserCard
    user="user"
    t-props="{
        showDetails: state.viewSettings.showDetails,
        ...(user.role === 'admin' ? { adminActions: adminActionList } : {}),
        ...(state.currentUser.id === user.id ? { highlight: true } : {})
    }"
/>
```

Patrón de Spread Props

```
<!-- Expandir un objeto completo como props -->
<UserCard t-props="user" additionalProp="someValue"/>

<!-- Combinar múltiples fuentes de props -->
<UserCard t-props="{
    ...user,
    ...state.cardSettings,
```

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

Props con Valores por Defecto

```
// En el componente hijo
static defaultProps = {
    size: 'medium',
    theme: 'light',
    showActions: true
};

// Estas props tendrán valores por defecto si no las proporciona el padre
```

Validación y Manejo de Errores

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

La Regla de Oro: Flujo de Datos Unidireccional

Esto nos lleva al principio más importante sobre las props: **Un componente hijo nunca, jamás, debe modificar sus propias props.**

Piensa en las props como un **contrato de solo lectura** del padre. El hijo puede usar los datos, mostrarlos y tomar decisiones basándose en ellos, pero no puede cambiarlos.

Por Qué Existe Esta Regla

1. **Predictibilidad**: Siempre sabes de dónde vienen los cambios de datos (el padre)
2. **Depuración**: Más fácil rastrear el flujo de datos y encontrar bugs
3. **Reutilización**: Los componentes hijos permanecen puros y reutilizables
4. **Rendimiento**: OWL puede optimizar el renderizado cuando el flujo de datos es predecible

Lo Que Esto Significa en la Práctica

```
// NUNCA hagas esto en un componente hijo
setup() {
```

```
  // NO modifiques las props directamente
  this.props.user.name = "Nombre Modificado"; // ¡Esto rompe el contrato!
  this.props.showDetails = true; // ¡Esto causará problemas!
}

// En su lugar, usa las props como datos de solo lectura
setup() {
  // Usa props para inicializar estado local si es necesario
  this.state = useState({
    localShowDetails: this.props.showDetails,
    editedName: this.props.user.name
  });

  // O crea propiedades computadas
  this.displayName = this.props.user.name.toUpperCase();
}
```

Cuando un Hijo Necesita "Cambiar" Props

Si un componente hijo necesita señalar que algo debería cambiar, debe **emitir un evento** hacia arriba al padre. El padre escucha el evento y decide si actualizar su propio estado, que luego fluye de vuelta hacia abajo como nuevas props.

```
// Método del componente hijo
requestNameChange(newName) {
  // No cambies props.user.name directamente
  // En su lugar, pídele al padre que lo cambie
  this.props.onNameChangeRequest?.(this.props.user.id, newName);
}

// El componente padre maneja la solicitud
onNameChangeRequest(userId, newName) {
  const user = this.state.users.find(u => u.id === userId);
  if (user) {
    user.name = newName; // El padre actualiza su propio estado
    // Esto fluirá de vuelta hacia abajo al hijo como nuevas props
  }
}
```

Este patrón mantiene el flujo de datos unidireccional mientras permite que los hijos influyan en el estado del padre a través de canales de comunicación bien definidos.

Props vs Estado: Marco de Decisión

Entender cuándo usar props vs estado es crucial para construir componentes mantenibles:

Usa Props Cuando:

- **Los datos vienen de un componente padre**
- **Los datos son configuración** o ajustes para el componente
- **Múltiples componentes necesitan los mismos datos**
- **Los datos representan las "entradas"** a tu componente
- **Los datos no deben ser modificados** por este componente

```
// Ejemplos de props
static props = {
  userId: { type: Number },         // ID para mostrar datos
  readonly: { type: Boolean },      // Opción de configuración
  theme: { type: String },          // Configuración de toda la app
  onSave: { type: Function }        // Callback al padre
};
```

Usa Estado Cuando:

- **Los datos pertenecen a este componente**
- **Los datos cambian basándose en interacciones del usuario** en este componente
- **Los datos son temporales o específicos de la UI** (como "¿está abierto el modal?")
- **Los datos se derivan de la entrada del usuario** en este componente

```
// Ejemplos de estado
this.state = useState({
  inputValue: "",           // Usuario escribiendo en este componente
  isLoading: false,         // Estado de carga de este componente
  showModal: false,         // Estado de UI de este componente
  validationErrors: []      // Estado de validación de este componente
});
```

Patrones Comunes de Props y Mejores Prácticas

1. Patrón de Props Callback

```
<!-- El padre proporciona callbacks para que el hijo se comunique de vuelta -->
<TodoItem
  todo="todo"
  onToggle="(id) => this.toggleTodo(id)"
  onEdit="(id, newText) => this.editTodo(id, newText)"
  onDelete="(id) => this.deleteTodo(id)"
/>
```

2. Patrón de Props de Configuración

```
<!-- El padre configura el comportamiento del hijo -->
<DataTable
  data="state.users"
  columns="tableColumns"
  sortable="true"
  filterable="true"
  pageSize="10"
  showPagination="true"
/>
```

3. Patrón de Render Props

```
<!-- El padre proporciona lógica de renderizado -->
<Modal
  isOpen="state.showModal"
  onClose="closeModal"
```

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

4. Patrón de Componente Compuesto

```
<!-- Múltiples componentes hijos relacionados -->
<Card>
  <CardHeader title="'Perfil de Usuario'" actions="headerActions"/>
  <CardBody content="userDetails"/>
  <CardFooter buttons="footerButtons"/>
</Card>
```

Depurando Props

Problemas Comunes de Props y Soluciones

Problema 1: Props que no se actualizan

```
<!-- Problema: Pasar un valor estático en lugar de estado reactivo -->
<UserCard user="staticUserObject"/>

<!-- Solución: Pasar estado reactivo -->
<UserCard user="state.currentUser"/>
```

Problema 2: Props undefined o tipo incorrecto

```
// Verificar props en el setup del componente
setup() {
  console.log("Props recibidas:", this.props);

  // Validar props críticas
  if (!this.props.user) {
    console.error("¡UserCard requiere una prop user!");
  }
}
```

Problema 3: Props de función que no funcionan

```
<!-- Problema: Llamar función en lugar de pasar referencia -->
<UserCard onUserClick="onUserClick()"/>

<!-- Solución: Pasar referencia de función (vinculada con .bind) -->
<UserCard onUserClick.bind="onUserClick"/>
```

Plantilla de Depuración de Props

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

Consideraciones de Rendimiento

Optimizando Props para Rendimiento

1. **Evita Crear Objetos en Plantillas**:

```
<!-- Malo: Crea nuevo objeto en cada render -->
<UserCard settings="{ theme: 'dark', size: 'large' }"/>

<!-- Bueno: Almacenar objeto en estado del componente o método -->
<UserCard settings="cardSettings"/>
```

2. **Memoriza Cálculos Complejos** (*memoization* significa cachear el resultado de un cálculo costoso para no recalcularlo cuando las entradas no han cambiado):

```
// Calcula props costosas una vez
get expensiveComputedProp() {
  // Memoriza o cachea este cálculo
  return this.state.data.reduce(...);
}
```

3. **Usa Referencias de Función Estables**:

```
// Crea métodos enlazados una vez en setup
setup() {
  this.boundHandleClick = this.handleClick.bind(this);
}

<!-- Usa la referencia estable en la plantilla -->
<UserCard onUserClick="boundHandleClick"/>
```

Errores Comunes

Error 1: Mutar Props Directamente

```
// NO HAGAS ESTO: rompe el contrato de flujo de datos unidireccional
this.props.user.name = "Nuevo Nombre";
```

Las props pertenecen al padre. Si el hijo necesita datos distintos, cópialos a un estado local (`useState`) o pide al padre que los cambie mediante una callback prop.

Error 2: Omitir la Validación con `static props`

Sin una declaración `static props`, OWL no puede detectar por ti una prop faltante o con el tipo equivocado — el error aparece después como un fallo confuso en tiempo de ejecución en vez de un mensaje claro al renderizar. Declara siempre `static props` en los componentes que reciben datos de un padre.

Error 3: Olvidar `.bind` en las Callback Props

```
<!-- `this` dentro de onEditUser será undefined -->
<UserCard onEditUser="onEditUser"/>

<!-- Correcto -->
<UserCard onEditUser.bind="onEditUser"/>
```

Error 4: No Definir `defaultProps` para Props Opcionales

Si una prop es `optional: true` pero el componente igual asume que tiene un valor (ej. `this.props.size.toUpperCase()`), un padre que la omita hará que el hijo falle. Acompaña cada prop opcional con una entrada razonable en `static defaultProps`.

Ejercicios

Ejercicio 1: Validar y Definir Valores por Defecto

Toma el componente `UserCard` de este capítulo y añade una nueva prop opcional `badgeText` (`String`). Dale un valor por defecto `"Member"` mediante `static defaultProps`, y muéstrala junto al nombre del usuario solo cuando `props.showDetails` sea verdadero.

Ejercicio 2: Disciplina de Solo Lectura

Escribe un pequeño componente hijo que reciba una prop `counter` (`Number`). Intenta mutar `this.props.counter` dentro de un método y confirma (en la consola del navegador) que OWL te advierte o que el cambio no persiste. Luego corrígelo copiando la prop a un estado local con `useState` en `setup()`.

Ejercicio 3: Callback Prop Desde Cero

Construye un componente `RatingWidget` que reciba una prop `value` y una callback prop `onRate`. Cuando el usuario haga clic en una de 5 estrellas, el componente debe llamar a `this.props.onRate(starIndex)` — nunca debe modificar `this.props.value` directamente. Conéctalo a un padre que guarde la calificación actual en `useState`.

Las props son los bloques de construcción fundamentales de la comunicación entre componentes en OWL. Al dominar los patrones de props y entender el principio de flujo de datos unidireccional, construirás aplicaciones que son predecibles, mantenibles y escalables.

TL;DR: Las props fluyen en una sola dirección, de padre a hijo; un hijo nunca muta sus propias props, y declara `static props`/`static defaultProps` para que los errores salgan a la luz temprano en vez de como bugs confusos en tiempo de ejecución.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch8_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

Ya cubrimos cómo fluyen los datos hacia abajo, de padre a hijo. En el próximo capítulo, aprenderemos sobre la otra mitad de la comunicación padre-hijo: cómo los hijos pueden comunicarse de vuelta a sus padres usando **callback props**.

Capítulo 9: Manejo de Eventos: Comunicación de Hijo a Padre

> **¿Por qué este capítulo?** Heredé más de un addon donde un widget hijo se metía directamente al estado de su padre para "arreglar" algo, y cada uno de esos se volvió frágil en cuanto alguien cambiaba el padre. Las callback props son la alternativa disciplinada.
>
> **¿Qué trata de resolver Odoo con esto?** OWL eliminó a propósito el viejo sistema de `trigger`/bus de eventos de OWL 1 — Odoo necesitaba comunicación hijo-a-padre rastreable desde el propio template del padre, no dispersa en listeners de eventos que tienes que ir a buscar.
>
> **Aplicación en la vida real:** Un `TodoItem` diciéndole a su `TodoList` "bórrame" o una fila de tabla diciéndole a su padre "me hicieron clic" tienen la misma forma que un widget de campo de formulario diciéndole a la vista de formulario "mi valor cambió" — el patrón que usarás constantemente en cuanto empieces a construir UI real de Odoo.

En el Capítulo 8, establecimos la regla de oro de las props: **los datos fluyen en una dirección, del padre hacia el hijo**. Un componente hijo nunca debe modificar sus propias props directamente. Esto crea aplicaciones predecibles y mantenibles donde siempre sabes dónde se originan los cambios de datos.

Pero esto plantea una pregunta importante: ¿qué sucede cuando un componente hijo necesita comunicarse de vuelta con su padre? ¿Qué pasa si un usuario hace clic en un botón "Eliminar" dentro de un componente `TodoItem`? El hijo no puede eliminar los datos por sí mismo porque esos datos pertenecen al padre y fueron pasados hacia abajo como prop.

Aquí es donde entra la segunda mitad del ciclo de comunicación: las **callback props** (props de función).

Mientras los datos fluyen hacia abajo a través de props, **la comunicación fluye hacia arriba a través de llamadas a funciones**. El padre le pasa una función al hijo como prop; cuando algo sucede dentro del hijo (un clic, un envío, un error), el hijo simplemente llama a esa función. El padre decide qué hacer con la información. Esto completa el ciclo de flujo de datos mientras mantiene al padre firmemente en control del estado de la aplicación.

> **¿Vienes de OWL 1.x?** El framework antiguo tenía un mecanismo `trigger()` que despachaba eventos personalizados hacia arriba en el árbol de componentes, y los padres los escuchaban con `t-on-nombre-evento` en la etiqueta del componente. **Ese mecanismo fue eliminado en OWL 2.0.** En OWL 2.0, `t-on-` funciona solo en elementos DOM reales (para eventos nativos como `click` e `input`), y la comunicación hijo-a-padre se hace exclusivamente con callback props. Si ves `this.trigger("algun-evento")` en código antiguo, eso es OWL 1 legacy (cubriremos su migración en el Capítulo 17).

Entendiendo el Flujo de Comunicación

Visualicemos cómo funciona esta comunicación bidireccional:

```
Componente Padre
|-- Estado: { todos: [...], users: [...] }
|-- Los datos fluyen HACIA ABAJO vía props ↓
|-- Los callbacks fluyen HACIA ABAJO vía props ↓
|
Componente Hijo
|-- Recibe: props de datos (solo lectura) + callback props
|-- Ocurre una interacción del usuario
|-- El hijo LLAMA al callback ↑
|
Componente Padre
|-- El callback se ejecuta en el padre
|-- Actualiza su propio estado
|-- El nuevo estado fluye HACIA ABAJO como props ↓
```

Este patrón asegura que: - **Los padres controlan los datos** y la lógica de negocio - **Los hijos permanecen puros y reutilizables** - **El flujo de datos es predecible** y fácil de depurar - **Los componentes están débilmente acoplados** y son mantenibles

---

Patrón Básico de Callbacks

Empecemos con un ejemplo simple. Crearemos un componente `TodoItem` que puede notificar a su padre cuando necesita ser eliminado.

El Componente Hijo: TodoItem

JavaScript de TodoItem (`todo_item.js`):

```javascript
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: funciones que el padre nos da para "llamar a casa"
    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
  };

  // Manejador de evento para el botón eliminar
  onDeleteClick() {
    console.log("TodoItem: Clic en eliminar, llamando al callback onDelete");

    // Llamar a la función del padre con un objeto de payload
    this.props.onDelete({
      todoId: this.props.todo.id,
      todoText: this.props.todo.text
    });
  }

  // Manejador de evento para alternar completado
  onToggleClick() {
    console.log("TodoItem: Clic en toggle, llamando al callback onToggle");

    this.props.onToggle({
      todoId: this.props.todo.id,
      currentStatus: this.props.todo.completed
    });
  }

  // Manejador de evento para solicitar edición
  onEditClick() {
    console.log("TodoItem: Clic en editar, llamando al callback onEdit");

    // Los callbacks opcionales pueden ser undefined - usa optional chaining
    this.props.onEdit?.({
      todoId: this.props.todo.id,
      currentText: this.props.todo.text
    });
  }

  // Propiedades computadas para estilos
  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';
  }
}
```

Plantilla de TodoItem (`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">

      <!-- Checkbox para Alternar Completado -->
      <div class="todo-toggle me-3">
        <input type="checkbox"
               class="form-check-input"
               t-att-checked="props.todo.completed"
               t-on-click="onToggleClick"/>
      </div>

      <!-- Contenido del Todo -->
      <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>

            <!-- Insignia de Prioridad -->
            <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>

          <!-- Fecha Límite -->
          <t t-if="props.todo.dueDate">
            <small class="text-muted">
              Vence: <t t-esc="props.todo.dueDate"/>
            </small>
          </t>
        </div>
      </div>

      <!-- Botones de Acción -->
      <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>
```

Fíjate en que `t-on-click` se usa **en elementos DOM reales** (el checkbox y los botones)—eso es exactamente para lo que sirve `t-on-` en OWL 2.0: escuchar eventos nativos del navegador. El puente hacia el padre es la callback prop.

El Componente Padre: TodoList

Ahora creemos el componente padre que proporciona esos callbacks:

JavaScript de TodoList (`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: "Aprender lo básico de OWL",
          completed: true,
          priority: "medium",
          dueDate: "2024-03-20"
        },
        {
          id: 2,
          text: "Construir una app de todos",
          completed: false,
          priority: "high",
          dueDate: "2024-03-25"
        },
        {
          id: 3,
          text: "Dominar la comunicación entre componentes",
          completed: false,
          priority: "low",
          dueDate: "2024-03-30"
```

```
      }
    ],

    editingTodoId: null,
    newTodoText: "",
    showCompleted: true,
    showPriority: true
  });
}

// Callback: el hijo nos pidió eliminar un todo
onTodoDelete({ todoId, todoText }) {
  console.log(`TodoList: onTodoDelete llamado para ID ${todoId}`);

  // Mostrar confirmación (opcional)
  if (confirm(`¿Estás seguro de que quieres eliminar "${todoText}"?`)) {
    // Actualizar el estado del padre - esto causará un re-render
    this.state.todos = this.state.todos.filter(todo => todo.id !== todoId);

    console.log(`TodoList: Todo ${todoId} eliminado exitosamente`);
  }
}

// Callback: el hijo nos pidió alternar el completado
onTodoToggle({ todoId }) {
  console.log(`TodoList: onTodoToggle llamado para ID ${todoId}`);

  // Encontrar y actualizar el todo
  const todo = this.state.todos.find(t => t.id === todoId);
  if (todo) {
    todo.completed = !todo.completed;
    console.log(`TodoList: Estado de completado del todo ${todoId} cambiado a
        ↪ ${todo.completed}`);
  }
}

// Callback: el hijo nos pidió empezar a editar un todo
onTodoEdit({ todoId, currentText }) {
  console.log(`TodoList: onTodoEdit llamado para ID ${todoId}`);

  // Entrar en modo de edición
  this.state.editingTodoId = todoId;
  this.state.newTodoText = currentText;
}

// Métodos locales para gestionar 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;
  }

  // Propiedades computadas
  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;
  }
}
```

Plantilla de TodoList (`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">

      <!-- Encabezado -->
```

```
    <div class="todo-header mb-4">
      <div class="d-flex justify-content-between align-items-center">
        <div>
          <h2>Mi Lista de Todos</h2>
          <p class="text-muted mb-0">
            <t t-esc="completedCount"/> completados de <t t-esc="totalCount"/> todos
          </p>
        </div>

        <!-- Controles de Vista -->
        <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 ? 'Ocultar Completados' : 'Mostrar Todos'"/>
          </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>
            Prioridad
          </button>
        </div>
      </div>
    </div>

    <!-- Formulario para Añadir Nuevo Todo -->
    <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="Añadir un nuevo 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>
            Añadir Todo
          </button>
        </div>
      </div>
    </div>

    <!-- Formulario de Edición -->
    <t t-if="state.editingTodoId">
      <div class="edit-todo-form mb-4 p-3 bg-warning bg-opacity-10 rounded">
        <h5>Editar 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>
                Guardar
              </button>
              <button class="btn btn-secondary" t-on-click="cancelEdit">
                <i class="fa fa-times"></i>
                Cancelar
              </button>
            </div>
          </div>
        </div>
      </t>

      <!-- Elementos Todo -->
      <div class="todo-items">
        <t t-if="visibleTodos.length > 0">
          <t t-foreach="visibleTodos" t-as="todo" t-key="todo.id">
            <!-- Aquí es donde le entregamos nuestros callbacks al hijo -->
            <TodoItem
                todo="todo"
                canEdit="true"
                canDelete="true"
                showPriority="state.showPriority"
                onDelete.bind="onTodoDelete"
                onToggle.bind="onTodoToggle"
                onEdit.bind="onTodoEdit"
            />
          </t>
        </t>

        <!-- Estado Vacío -->
        <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 hay todos que mostrar</h4>
            <p class="text-muted">
              <t t-if="!state.showCompleted">
                ¡Todos los todos están completados! Activa "Mostrar Todos" para verlos.
              </t>
              <t t-else="">
                Añade tu primer todo arriba para empezar.
              </t>
            </p>
          </div>
        </t>
      </div>
    </div>
  </t>

</templates>
```

El Sufijo `.bind`: Detalle Pequeño, Gran Diferencia

Mira de cerca cómo se pasan los callbacks:

```
<TodoItem onDelete.bind="onTodoDelete"/>
```

El sufijo `.bind` le dice a OWL que vincule (bind) la función al componente padre antes de pasarla hacia abajo. Sin él, cuando el hijo llame a `this.props.onDelete(...)`, el `this` dentro de `onTodoDelete` sería `undefined` y `this.state` fallaría.

Tienes tres opciones equivalentes—elige una y sé consistente:

```
<!-- Opción 1 (recomendada): el sufijo .bind -->
<TodoItem onDelete.bind="onTodoDelete"/>

<!-- Opción 2: una función flecha inline (vincula `this` automáticamente) -->
<TodoItem onDelete="(payload) => this.onTodoDelete(payload)"/>
```

```
// Opción 3: vincular manualmente en setup()
setup() {
  this.onTodoDelete = this.onTodoDelete.bind(this);
}
```

El sufijo `.bind` es la forma idiomática de OWL 2.0 y lo que verás en todo el código de Odoo.

Patrones Avanzados de Callbacks

1. Eventos Nativos y Detener la Propagación

`t-on-` en elementos DOM soporta modificadores útiles:

```
<!-- .stop llama a event.stopPropagation() por ti -->
<button t-on-click.stop="onDeleteClick">Eliminar</button>

<!-- .prevent llama a event.preventDefault() -->
<form t-on-submit.prevent="onSubmit">...</form>

<!-- .enter se dispara solo con la tecla Enter -->
<input t-on-keyup.enter="addNewTodo"/>
```

Usa estos para controlar el evento DOM nativo; luego llama a tu callback prop como de costumbre.

2. Invocación Condicional de Callbacks

```
// Solo llamar al callback si se cumplen ciertas condiciones
onDeleteClick() {
  if (this.props.todo.completed) {
    this.props.onDelete({ todoId: this.props.todo.id });
  } else {
    this.props.onConfirmDelete?.({
      todoId: this.props.todo.id,
      message: "Este todo no está completado. ¿Estás seguro?"
    });
  }
}
```

3. Callbacks Asíncronos

Los callbacks pueden ser `async`, lo que permite al hijo esperar el trabajo del padre (por ejemplo, para mostrar un spinner mientras el padre guarda):

```
// Hijo
async onSaveClick() {
  this.state.saving = true;
  try {
    await this.props.onSave({ data: this.state.formData });
  } finally {
    this.state.saving = false;
  }
}
```

4. Validación en el Padre

```
// El padre valida el payload antes de procesarlo
onTodoUpdate({ todoId, newText }) {
  // Validar los datos
  if (!todoId || typeof newText !== 'string' || newText.trim().length === 0) {
    console.error("Actualización de todo inválida:", { todoId, newText });
    return;
  }

  // Proceder con la actualización
  const todo = this.state.todos.find(t => t.id === todoId);
  if (todo) {
    todo.text = newText.trim();
  }
}
```

Convenciones de Nombres para Callbacks

Seguir convenciones de nombres consistentes hace tu código más mantenible:

```
// En la definición de props del hijo: "on" + lo que pasó, en camelCase
static props = {
  onTodoCreated: { type: Function },
  onUserSelected: { type: Function },
  onFormSubmitted: { type: Function },
  onError: { type: Function, optional: true },
};
```

Mejores Prácticas:

1. **Prefijo `on`**: `onDelete`, `onSave`, `onUserSelected`—se lee instantáneamente como "callback"
2. **Usa camelCase**: las callback props son props normales, así que siguen la convención de nombres de props (`onTodoDeleted`, no `on-todo-deleted`)
3. **Sé descriptivo**: `onUserProfileUpdated` en vez de `onUpdate` cuando el contexto no sea obvio
4. **Incluye contexto en los payloads**: pasa un objeto (`{ todoId, todoText }`) en vez de argumentos sueltos—es auto-documentado y extensible
5. **Marca los callbacks realmente opcionales con `optional: true`** y llámalos con `?.()`

Patrón de Comunicación Complejo

Creemos un ejemplo más sofisticado con un padre `UserManager` y múltiples componentes hijos:

Padre UserManager (`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: "Ventas" },
        { id: 3, name: "Carol Brown", role: "user", active: false, department: "RRHH" }
      ],
      filters: {
        role: "all",
        department: "all",
        active: true
      },
      editingUser: null,
      showForm: false,
      searchText: ""
    });
  }

  // Callbacks para UserCard
  onUserEdit({ user }) {
    console.log("UserManager: Edición de usuario solicitada", user);
    this.state.editingUser = { ...user }; // Clonar para editar
    this.state.showForm = true;
  }

  onUserDelete({ userId, userName }) {
    console.log("UserManager: Eliminación de usuario solicitada", userId);

    if (confirm(`¿Eliminar usuario ${userName}?`)) {
      this.state.users = this.state.users.filter(u => u.id !== userId);
    }
  }

  onUserToggleStatus({ userId }) {
    console.log("UserManager: Alternar estado del usuario", userId);

    const user = this.state.users.find(u => u.id === userId);
    if (user) {
      user.active = !user.active;
```

```
  }
}

// Callbacks para UserForm
onUserSave({ user }) {
  console.log("UserManager: Guardar usuario", user);

  if (user.id) {
    // Actualizar usuario existente
    const index = this.state.users.findIndex(u => u.id === user.id);
    if (index !== -1) {
      this.state.users[index] = user;
    }
  } else {
    // Crear nuevo usuario
    user.id = Math.max(...this.state.users.map(u => u.id), 0) + 1;
    this.state.users.push(user);
  }

  this.onFormCancel();
}

onFormCancel() {
  console.log("UserManager: Formulario cancelado");
  this.state.showForm = false;
  this.state.editingUser = null;
}
```

Tres hijos distintos (`UserCard`, `UserForm`, `UserFilters`) tienen arriba sus propios métodos de callback dedicados — `UserManager` nunca necesita adivinar qué hijo lo llamó, ya que cada callback se pasa explícitamente a un único componente en la plantilla.

```
// Callbacks para UserFilters
onFiltersChanged({ filters }) {
  console.log("UserManager: Filtros cambiados", filters);
  this.state.filters = { ...this.state.filters, ...filters };
}

onSearchChanged({ searchText }) {
  console.log("UserManager: Búsqueda cambiada", searchText);
  this.state.searchText = searchText;
}

// Acciones locales
openNewUserForm() {
  this.state.editingUser = {
    id: null,
    name: "",
    role: "user",
    active: true,
    department: ""
  };
  this.state.showForm = true;
}
```

Finalmente, `filteredUsers` es un simple getter computado — sin props ni callbacks involucrados — que combina los filtros activos y el texto de búsqueda en la lista que realmente se muestra en la

plantilla:

```
  // Propiedades computadas
  get filteredUsers() {
    let filtered = this.state.users;

    // Aplicar filtro de rol
    if (this.state.filters.role !== "all") {
      filtered = filtered.filter(u => u.role === this.state.filters.role);
    }

    // Aplicar filtro de departamento
    if (this.state.filters.department !== "all") {
      filtered = filtered.filter(u => u.department === this.state.filters.department);
    }

    // Aplicar filtro de activos
    if (this.state.filters.active !== null) {
      filtered = filtered.filter(u => u.active === this.state.filters.active);
    }

    // Aplicar búsqueda
    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;
  }
}
```

Y en la plantilla, cada hijo recibe exactamente los callbacks que necesita:

```
<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>
```

Este ejemplo demuestra cómo un solo padre puede coordinar múltiples componentes hijos a través de callbacks, manteniendo la gestión de estado centralizada mientras permite que los hijos permanezcan enfocados y reutilizables.

Props de Datos vs Callback Props: Marco de Decisión

Ambas viajan hacia abajo como props, pero sirven a direcciones opuestas de comunicación:

Usa Callback Props Cuando:

- **El hijo necesita notificar al padre** de acciones del usuario
- **El hijo necesita solicitar** al padre que cambie datos
- **El hijo encuentra un error** que el padre debería manejar
- **El hijo completa una operación asíncrona**

```
static props = {
  onDeleted: { type: Function },        // "Algo pasó"
  onSaveRequested: { type: Function }, // "Por favor haz algo"
  onError: { type: Function },          // "Algo salió mal"
};
```

Usa Props de Datos Cuando:

- **El padre necesita configurar** el comportamiento del hijo
- **El padre necesita pasar datos** al hijo
- **El padre controla** el estado de visualización del hijo

```
static props = {
  user: { type: Object },           // Prop de datos
  readonly: { type: Boolean },      // Prop de configuración
  showDetails: { type: Boolean }    // Prop de estado
};
```

> **¿Y los componentes que no son padre e hijo?** Los callbacks funcionan a través del árbol de componentes. Para comunicación entre componentes distantes (hermanos, o a través de toda la app), usarás estado compartido y el bus del entorno—ambos cubiertos en el Capítulo 15.

Depurando Callbacks

Patrón de Console Logging

```
// En el componente hijo
onAction() {
  console.log("Hijo: Llamando al callback onAction", { payload: "data" });
  this.props.onAction({ payload: "data" });
}

// En el componente padre
```

```
onChildAction(payload) {
  console.log("Padre: onChildAction recibió", payload);
  // Manejarlo...
}
```

Problemas Comunes y Soluciones

Problema 1: "Cannot read properties of undefined (reading 'state')"

```
<!-- Problema: la función sin vincular pierde `this` -->
<TodoItem onDelete="onTodoDelete"/>

<!-- Solución: usa el sufijo .bind -->
<TodoItem onDelete.bind="onTodoDelete"/>
```

Problema 2: "props.onEdit is not a function"

```
// El padre no pasó un callback opcional.
// Decláralo opcional y llámalo con seguridad:
static props = {
  onEdit: { type: Function, optional: true },
};

onEditClick() {
  this.props.onEdit?.({ todoId: this.props.todo.id });
}
```

Problema 3: El callback se ejecuta inmediatamente al renderizar

```
<!-- Problema: esto LLAMA a la función durante el renderizado -->
<TodoItem onDelete="onTodoDelete(todo.id)"/>

<!-- Solución: pasa una función, no la llames -->
<TodoItem onDelete="() => this.onTodoDelete(todo.id)"/>
```

Consideraciones de Rendimiento

1. Aplica Debounce a Callbacks de Alta Frecuencia

```
// Malo: llama al padre en cada pulsación de tecla
onInputChange(event) {
  this.props.onTextChanged({ value: event.target.value });
}

// Bueno: callback con debounce. "Debounce" significa retrasar una llamada
// hasta que una ráfaga de eventos disparadores (ej. pulsaciones de tecla)
// se detenga durante un período dado, para no hacer una llamada por cada
// tecla. `debounce` está disponible en @web/core/utils/timing.
setup() {
  this.debouncedNotify = debounce((value) => {
    this.props.onTextChanged({ value });
  }, 300);
}

onInputChange(event) {
```

```
  this.debouncedNotify(event.target.value);
}
```

2. Minimiza el Tamaño del Payload

```
// Malo: payload grande con datos innecesarios
this.props.onUserSelected({
  fullUser: this.props.user,   // El objeto de usuario completo
  allUsers: this.props.users   // Datos innecesarios
});

// Bueno: payload mínimo
this.props.onUserSelected({
  userId: this.props.user.id   // Solo el ID
});
```

3. Usa Referencias de Función Estables

El sufijo `.bind` crea la función vinculada una sola vez, así que los hijos no ven una prop "nueva" en cada render. Prefiérelo sobre definir funciones flecha nuevas en las plantillas cuando el callback no necesita variables del bucle.

Errores Comunes

Error 1: Olvidar `.bind`

```
<!-- `this` dentro de onTodoDelete será undefined -->
<TodoItem onDelete="onTodoDelete"/>
```

Usa siempre `.bind`, una función flecha en línea, o un `.bind(this)` manual en `setup()` — ver "El Sufijo `.bind`" más arriba.

Error 2: Llamar al Callback en Vez de Pasarlo

```
<!-- Problema: esto LLAMA a onTodoDelete durante el renderizado, no al hacer clic -->
<TodoItem onDelete="onTodoDelete(todo.id)"/>

<!-- Solución -->
<TodoItem onDelete="() => this.onTodoDelete(todo.id)"/>
```

Error 3: Tratar los Callbacks Opcionales Como Siempre Presentes

Si una callback prop es `optional: true`, llamarla directamente (`this.props.onEdit(...)`) lanza un error cuando el padre no la pasó. Usa encadenamiento opcional: `this.props.onEdit?.(...)`.

Error 4: Enviar Payloads Voluminosos

Pasar todo el estado del componente, o datos no relacionados, en el payload de un callback acopla al hijo con detalles internos que no debería conocer. Envía lo mínimo que el padre necesita (un id, un objeto pequeño) en vez de objetos o arrays completos.

Ejercicios

Ejercicio 1: Añade un Callback

Extiende el ejemplo `TodoItem`/`TodoList` con una nueva callback prop `onPriorityChange` que el hijo llame cuando el usuario haga clic en un botón "subir prioridad". El padre debe actualizar la prioridad del todo en su estado.

Ejercicio 2: Corrige el Bug

Dado `<TodoItem onDelete="onTodoDelete"/>` (sin `.bind`, sin función flecha), predice qué pasa al hacer clic en el botón de eliminar, y luego verifícalo en la consola del navegador. Corrígelo usando el sufijo `.bind`.

Ejercicio 3: Búsqueda con Debounce

Construye un componente `SearchBox` con un input de texto y una callback prop `onSearch`. Aplica debounce al callback para que solo se dispare 300ms después de que el usuario deja de escribir, usando `debounce` de `@web/core/utils/timing`.

Las callback props son la clave para construir aplicaciones OWL mantenibles donde los componentes pueden comunicarse efectivamente mientras permanecen débilmente acoplados. Al dominar este patrón—datos hacia abajo, callbacks hacia arriba—creas aplicaciones que son tanto poderosas como predecibles.

TL;DR: Los hijos le hablan a los padres llamando a una función que el padre pasó hacia abajo como prop (siempre con `.bind`), nunca mutando el estado del padre ni recurriendo a un bus de eventos legado.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch9_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

Con datos hacia abajo y callbacks hacia arriba, el ciclo de comunicación padre-hijo está completo. En el siguiente capítulo, exploraremos los hooks del ciclo de vida del componente que te permiten aprovechar momentos clave en la vida de un componente para realizar inicialización, limpieza y optimización.

Capítulo 10: Los Hooks del Ciclo de Vida del Componente

> **¿Por qué este capítulo?** Más incidentes de producción de los que he atendido se remontan a errores de ciclo de vida que a casi cualquier otra cosa — un fetch que se dispara antes de que exista el DOM, un listener que nunca se limpia y va filtrando memoria lentamente a lo largo de una sesión larga.
>
> **¿Qué trata de resolver Odoo con esto?** Una vista de backend puede montar y desmontar docenas de widgets mientras un usuario navega — los hooks de ciclo de vida son cómo Odoo garantiza que cada uno se inicialice y se desmonte limpiamente, sin que queden timers o listeners acumulándose.
>
> **Aplicación en la vida real:** Cargar los registros relacionados de un partner antes del primer render (`onWillStart`), enfocar un input cuando se abre un formulario (`onMounted`), y limpiar un intervalo cuando se quita una tarjeta kanban (`onWillUnmount`) son todos hooks de ciclo de vida haciendo exactamente el trabajo que describe este capítulo.

Un componente, al igual que un organismo vivo, tiene un **ciclo de vida**. Nace (se crea y se monta en el DOM), vive (se actualiza en respuesta a cambios de estado y props), y eventualmente, muere (se desmonta y se remueve del DOM).

OWL proporciona una forma poderosa de aprovechar estos momentos clave en la vida de un componente usando **hooks del ciclo de vida**. Estos hooks son funciones especiales que puedes llamar dentro de tu método `setup` para registrar callbacks que se ejecutarán en puntos específicos del ciclo de vida.

Entender estos hooks es esencial para manejar efectos secundarios, obtener datos correctamente y limpiar recursos para prevenir fugas de memoria. Exploremos los hooks más importantes que usarás en tu desarrollo día a día.

El Ciclo de Vida de un Vistazo

Antes de profundizar en los hooks individuales, visualicemos el viaje completo de un componente:

1. **Fase de Creación:**
   - `setup()`: Se ejecuta la lógica de configuración principal del componente y registra los hooks.
   - `onWillStart`: Se ejecuta antes del primer renderizado, perfecto para obtener datos iniciales.
   - La instancia del componente se crea y se prepara.
2. **Fase de Montaje (Añadido a la página):**
   - El componente renderiza su plantilla HTML.
   - El HTML se inserta en el DOM.
   - `onMounted`: El componente ahora está vivo en la página.
3. **Fase de Actualización (Durante su vida - se repite según sea necesario):**

- El padre pasa nuevas props → `onWillUpdateProps`.
- Los cambios de estado activan re-renderizado.
- El componente re-renderiza su HTML.
- El DOM se actualiza con los cambios.

4. **Fase de Desmontaje (Remoción de la página):**
   - `onWillUnmount`: El componente está a punto de ser destruido.
   - El componente es removido del DOM.
   - La memoria y recursos se limpian.

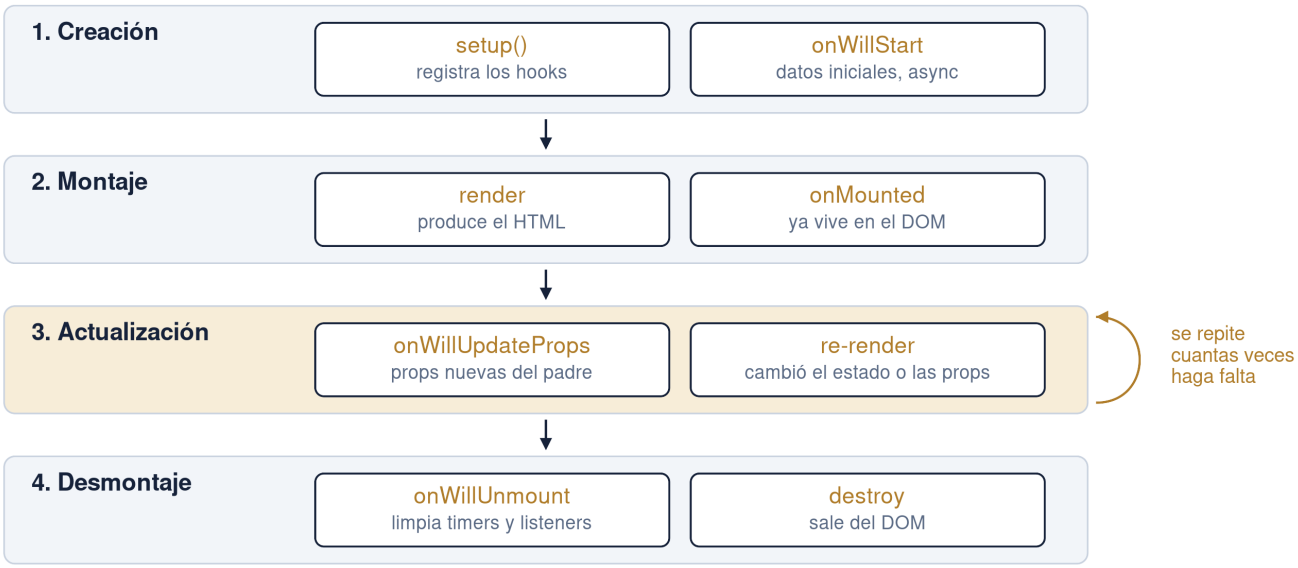


Figura 4: Las cuatro fases de la vida de un componente, y qué hook se dispara en cada una.

`onWillStart`: Obteniendo Datos Iniciales

El hook `onWillStart` es único porque es el **único hook que puede ser asíncrono**. Se ejecuta antes del renderizado inicial del componente, lo que lo hace el lugar perfecto para obtener datos que el componente necesita para su primera aparición.

- **Cuándo se ejecuta:** Después de que `setup()` termina, pero antes del renderizado inicial.
- **Característica clave:** Puede ser async - el componente espera a que la promesa se resuelva.
- **Caso de uso:** Obtener datos esenciales que deben estar listos antes del primer renderizado.

Ejemplo Básico: Lista de Productos

Creemos un componente `ProductList` que obtiene productos antes de renderizar:

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: Obteniendo productos antes del primer renderizado...");
        const data = await this.orm.searchRead(
          "product.product",
          [],
          ["name", "list_price", "categ_id"]
        );
        this.state.products = data;
        console.log(`ProductList: Se cargaron ${data.length} productos`);
      } catch (error) {
        console.error("Falló la carga de productos:", error);
        this.state.error = "Falló la carga de productos";
      } finally {
        this.state.loading = false;
      }
    });
  }

  get formattedProducts() {
    return this.state.products.map(product => ({
      ...product,
      formattedPrice: `$${product.list_price.toFixed(2)}`
    }));
  }
}
```

Plantilla (`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">
      <!-- Estado de Carga -->
      <t t-if="state.loading">
        <div class="text-center py-4">
          <div class="spinner-border text-primary" role="status">
            <span class="visually-hidden">Cargando...</span>
          </div>
          <p class="mt-2 text-muted">Cargando productos...</p>
        </div>
      </t>

      <!-- Estado de Error -->
      <t t-elif="state.error">
        <div class="alert alert-danger" t-esc="state.error"/>
      </t>
```

```
      <!-- Cuadrícula de Productos -->
      <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>
```

Al obtener datos en `onWillStart`, previenes que el componente renderice un estado vacío y luego "parpadee" cuando llegan los datos. El componente esperará a que la promesa de `onWillStart` se resuelva antes de realizar su renderizado inicial.

Patrones Avanzados de onWillStart

Múltiples Operaciones Asíncronas:

```
onWillStart(async () => {
  // Ejecutar múltiples operaciones en paralelo
  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;
});
```

Patrón de Recuperación de Errores:

```
onWillStart(async () => {
  try {
    // Intentar fuente de datos principal
    this.state.data = await this.fetchFromPrimarySource();
  } catch (primaryError) {
    console.warn("Falló la fuente principal, intentando respaldo:", primaryError);

    try {
      // Intentar fuente de datos de respaldo
      this.state.data = await this.fetchFromFallbackSource();
      this.state.usingFallback = true;
    } catch (fallbackError) {
```

```
      console.error("Fallaron todas las fuentes de datos:", fallbackError);
      this.state.error = "No se pueden cargar datos de ninguna fuente";
    }
  }
});
```

`onMounted`: Interactuando con el DOM

El hook `onMounted` se ejecuta **después** de que el componente ha sido renderizado por primera vez y su HTML ha sido añadido al DOM. Esta es tu puerta de entrada para manipulación del DOM e integración de librerías de terceros.

- **Cuándo se ejecuta:** Después del renderizado inicial e inserción en el DOM.
- **Caso de uso:** Cualquier tarea que requiera que el HTML del componente exista en el DOM.

Casos de Uso Clave:

- **Enfocar elementos input**
- **Inicializar librerías de terceros** (gráficos, mapas, calendarios)
- **Medir dimensiones de elementos**
- **Configurar event listeners del DOM**

Para obtener una referencia a un elemento específico en tu plantilla, usas la directiva `t-ref` y el hook `useRef`.

Ejemplo: Componente de Búsqueda con Enfoque Automático

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: []
    });

    // Crear una referencia al input de búsqueda
    this.searchInputRef = useRef("search-input");

    onMounted(() => {
      console.log("SearchComponent: Componente montado, enfocando input");

      // El elemento input ahora está en el DOM y puede ser enfocado
      if (this.searchInputRef.el) {
        this.searchInputRef.el.focus();
        console.log("SearchComponent: Input enfocado exitosamente");
      }
    });
  }
```

```
  onSearchInput(event) {
    this.state.searchTerm = event.target.value;
    console.log("Término de búsqueda cambiado:", this.state.searchTerm);
    // Aquí podrías activar una llamada a API de búsqueda
  }
}
```

Plantilla (`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="Buscar productos..."
            t-model="state.searchTerm"
            t-on-input="onSearchInput"
        />
      </div>

      <div class="search-results">
        <t t-if="state.searchTerm">
          <p class="text-muted">
            Buscando: "<t t-esc="state.searchTerm"/>"
          </p>
        </t>

        <!-- Los resultados irían aquí -->
        <div class="results-list">
          <!-- Implementación de resultados -->
        </div>
      </div>
    </div>
  </t>
</templates>
```

Ejemplo Avanzado de onMounted: Integración de Gráficos

Aquí hay un ejemplo más complejo que se integra con 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: Componente montado, inicializando gráfico");
      await this.loadSalesData();
      this.initializeChart();
    });

    onWillUnmount(() => {
      // Limpiar el gráfico cuando el componente se destruya
      if (this.chartInstance) {
        console.log("SalesChart: Destruyendo instancia de gráfico");
        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("Falló la carga de datos de ventas:", error);
      this.state.loading = false;
    }
  }

  initializeChart() {
    if (!this.chartCanvasRef.el || this.state.loading) {
      return;
    }

    // Asumiendo que Chart.js está cargado globalmente
    const ctx = this.chartCanvasRef.el.getContext('2d');

    // Procesar datos para el gráfico
    const processedData = this.processChartData();

    this.chartInstance = new Chart(ctx, {
      type: 'line',
      data: {
        labels: processedData.labels,
        datasets: [{
          label: 'Ingresos por Ventas',
          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: 'Rendimiento de Ventas'
          }
        }
      }
    });

    console.log("SalesChart: Gráfico inicializado exitosamente");
  }

  processChartData() {
    // Agrupar ventas por mes y sumar cantidades
    const monthlyData = {};

    this.state.chartData.forEach(order => {
      const month = order.date_order.substring(0, 7); // Formato YYYY-MM
      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])
    };
  }
}
```

Plantilla (`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">Cargando datos del gráfico...</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`: Reaccionando a Cambios de Props

Este hook se llama cuando un componente padre está a punto de pasar nuevas props al hijo. Permite que el hijo reaccione a los cambios entrantes *antes* de que se re-renderice.

- **Cuándo se ejecuta:** Cuando se reciben nuevas props, antes de que el componente se actualice.
- **Caso de uso:** Cuando un componente hijo necesita obtener nuevos datos basados en props cambiadas.

Ejemplo: Componente de Perfil de Usuario

Imagina un componente `UserProfile` que muestra detalles del usuario. Cuando el padre cambia la prop `userId`, el componente necesita obtener los datos del nuevo usuario:

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");

    // Cargar datos iniciales del usuario
    onWillStart(async () => {
      await this.loadUser(this.props.userId);
    });

    // Reaccionar a cambios de props
    onWillUpdateProps(async (nextProps) => {
      console.log("UserProfile: Las props se van a actualizar", {
        current: this.props.userId,
        next: nextProps.userId
      });

      // Solo recargar si el ID del usuario realmente cambió
      if (this.props.userId !== nextProps.userId) {
```

```
        console.log(`UserProfile: ID de usuario cambió de ${this.props.userId} a
            ↪ ${nextProps.userId}`);
        this.state.loading = true;
        await this.loadUser(nextProps.userId);
      }
    });
  }

  async loadUser(userId) {
    try {
      console.log(`UserProfile: Cargando datos de usuario para 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: Usuario ${users[0].name} cargado exitosamente`);
      } else {
        throw new Error(`Usuario con ID ${userId} no encontrado`);
      }

    } catch (error) {
      console.error(`Falló la carga del usuario ${userId}:`, error);
      this.state.error = `Falló la carga del usuario: ${error.message}`;
      this.state.user = null;
    } finally {
      this.state.loading = false;
    }
  }
}
```

Plantilla (`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>Perfil de Usuario</h5>
      </div>
      <div class="card-body">
        <!-- Estado de Carga -->
        <t t-if="state.loading">
          <div class="text-center py-3">
            <div class="spinner-border text-primary" role="status">
              <span class="visually-hidden">Cargando usuario...</span>
            </div>
            <p class="mt-2 text-muted">Cargando perfil de usuario...</p>
          </div>
        </t>
```

```
        <!-- Estado de Error -->
        <t t-elif="state.error">
          <div class="alert alert-danger" t-esc="state.error"/>
        </t>

        <!-- Datos del Usuario -->
        <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>Teléfono:</strong>
                  <span t-esc="state.user.phone"/>
                </div>
              </t>
              <div class="mb-2">
                <strong>ID de Partner:</strong>
                <span t-esc="state.user.partner_id[1]"/>
              </div>
            </div>
          </div>
        </t>
      </div>
    </div>
  </t>
</templates>
```

Ejemplo de Componente Padre

Aquí hay un ejemplo de cómo un componente padre podría usar el `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: Seleccionando usuario", userId);
    this.state.selectedUserId = userId;
  }
}
```

Plantilla (`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">
        <!-- Selección de Usuario -->
        <div class="col-md-4">
          <div class="card">
            <div class="card-header">
              <h6>Seleccionar Usuario</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>

        <!-- Mostrar Perfil de Usuario -->
        <div class="col-md-8">
          <UserProfile userId="state.selectedUserId"/>
        </div>
      </div>
    </div>
  </t>
</templates>
```

Este patrón asegura que el componente `UserProfile` siempre muestre datos consistentes con la prop `userId` que recibe de su padre. Cuando el usuario hace clic en un botón de usuario diferente en el padre, el `UserProfile` automáticamente detectará el cambio de prop y obtendrá los datos del nuevo usuario.

`onWillUnmount`: Limpieza

El hook `onWillUnmount` es el último aliento del componente. Se ejecuta justo antes de que el componente sea removido del DOM. Esto es **crucial** para la limpieza y prevenir fugas de memoria.

- **Cuándo se ejecuta:** Justo antes de que el componente sea destruido.

- **Caso de uso crítico:** Limpiar recursos que de otra manera persistirían después de la destrucción del componente.

Qué Limpiar:

- **Timers e intervalos** (`setInterval`, `setTimeout`)
- **Event listeners globales** (en `window`, `document`)
- **Instancias de librerías de terceros** (gráficos, mapas, etc.)
- **Conexiones WebSocket**
- **Peticiones API en curso** (si son cancelables)

Ejemplo: Componente Basado en Timer

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
    });

    // Almacenar referencia del timer para limpieza
    this.timer = null;

    onMounted(() => {
      console.log("LiveClock: Iniciando timer del reloj");
      this.startClock();
    });

    onWillUnmount(() => {
      console.log("LiveClock: Limpiando timer del reloj");
      this.stopClock();
    });
  }

  startClock() {
    // Actualizar tiempo cada segundo
    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 limpiado exitosamente");
    }
```

```
  }

  toggleClock() {
    this.state.isRunning = !this.state.isRunning;
    console.log("LiveClock: Reloj", this.state.isRunning ? "reanudado" : "pausado");
  }
}
```

Plantilla (`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 ? 'Pausar' : 'Reanudar'"/>
        </button>
      </div>
    </div>
  </t>
</templates>
```

Ejemplo Avanzado de Limpieza: Múltiples Recursos

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
    });

    // Almacenar referencias de limpieza
    this.cleanupTasks = [];

    onMounted(() => {
      this.initializeSystem();
    });

    onWillUnmount(() => {
      console.log("NotificationSystem: Realizando limpieza completa");
      this.performCleanup();
    });
  }

  initializeSystem() {
    // 1. Configurar chequeo periódico de salud
    const healthCheckTimer = setInterval(() => {
```

```
      this.checkSystemHealth();
    }, 5000);

    this.cleanupTasks.push(() => {
      clearInterval(healthCheckTimer);
      console.log("NotificationSystem: Timer de chequeo de salud limpiado");
    });

    // 2. Configurar listeners de focus/blur de ventana
    const handleWindowFocus = () => {
      console.log("NotificationSystem: Ventana enfocada, refrescando notificaciones");
      this.refreshNotifications();
    };

    const handleWindowBlur = () => {
      console.log("NotificationSystem: Ventana desenfocada");
    };

    window.addEventListener('focus', handleWindowFocus);
    window.addEventListener('blur', handleWindowBlur);

    this.cleanupTasks.push(() => {
      window.removeEventListener('focus', handleWindowFocus);
      window.removeEventListener('blur', handleWindowBlur);
      console.log("NotificationSystem: Event listeners de ventana removidos");
    });

    // 3. Configurar listeners de online/offline
    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: Event listeners de red removidos");
    });

    // 4. Inicializar estado de conexión
    this.state.isConnected = navigator.onLine;
  }
```

Fíjate en el patrón: cada vez que `initializeSystem` configura algo (un timer, un par de listeners), inmediatamente agrega una función "deshacer" correspondiente a `this.cleanupTasks`. Eso es lo que permite que la única llamada a `performCleanup()` de abajo pueda revertir todo, sin importar cuántos recursos se hayan iniciado.

```
  checkSystemHealth() {
```

```
    // Simular chequeo de salud del sistema
    console.log("NotificationSystem: Realizando chequeo de salud");
  }

  refreshNotifications() {
    // Simular refresco de notificaciones
    console.log("NotificationSystem: Refrescando notificaciones");
  }

  performCleanup() {
    // Ejecutar todas las tareas de limpieza
    this.cleanupTasks.forEach((cleanup, index) => {
      try {
        cleanup();
      } catch (error) {
        console.error(`NotificationSystem: Tarea de limpieza ${index} falló:`, error);
      }
    });

    // Limpiar el array de tareas de limpieza
    this.cleanupTasks = [];

    console.log("NotificationSystem: Todas las tareas de limpieza completadas");
  }
}
```

Si olvidas limpiar en `onWillUnmount`, recursos como timers seguirían ejecutándose en el fondo para siempre, incluso después de que el componente haya desaparecido, creando fugas de memoria. Usar `onWillUnmount` para limpieza es una parte crítica de escribir aplicaciones robustas.

Mejores Prácticas del Ciclo de Vida

QUÉ HACER

1. **Siempre limpiar en onWillUnmount** - Limpiar timers, remover event listeners, destruir instancias de terceros
2. **Usar onWillStart para datos críticos** - Datos que deben estar listos antes del primer renderizado
3. **Hacer onWillStart asíncrono** - Aprovechar su capacidad asíncrona única
4. **Manejar errores graciosamente** - Envolver operaciones asíncronas en bloques try-catch
5. **Loggear eventos del ciclo de vida** - Ayuda con debugging durante desarrollo

QUÉ NO HACER

1. **Nunca mutar props** - Las props son de solo lectura en todos los hooks del ciclo de vida
2. **No olvidar la limpieza** - Las fugas de memoria dañarán el rendimiento de tu aplicación
3. **No asumir que el DOM existe** - Solo `onMounted` y hooks posteriores tienen acceso al DOM
4. **No realizar cómputos pesados** - Mantén los hooks del ciclo de vida rápidos y enfocados

Patrones Comunes

Patrón 1: Carga Condicional de Datos

```
onWillUpdateProps(async (nextProps) => {
  // Solo obtener si la prop importante cambió
  if (this.props.customerId !== nextProps.customerId) {
    await this.loadCustomerData(nextProps.customerId);
  }
});
```

Patrón 2: Helper de Limpieza

```
setup() {
  this.cleanupFunctions = [];

  onMounted(() => {
    // Configurar algo que necesita limpieza
    const timer = setInterval(this.updateData, 1000);
    this.cleanupFunctions.push(() => clearInterval(timer));
  });

  onWillUnmount(() => {
    this.cleanupFunctions.forEach(cleanup => cleanup());
  });
}
```

Patrón 3: Límite de Error

```
onWillStart(async () => {
  try {
    await this.loadCriticalData();
  } catch (error) {
    console.error("Falló la carga de datos críticos:", error);
    this.state.error = "Sistema temporalmente no disponible";
    // También podría despachar al padre o mostrar UI de respaldo
  }
});
```

Debuggeando Problemas del Ciclo de Vida

Al trabajar con hooks del ciclo de vida, los problemas comunes incluyen:

Problema: El componente no se actualiza cuando cambian las props **Solución**: Añadir manejador de `onWillUpdateProps`:

```
onWillUpdateProps(async (nextProps) => {
  if (this.props.dataId !== nextProps.dataId) {
    await this.loadNewData(nextProps.dataId);
  }
});
```

Problema: Fugas de memoria por timers olvidados **Solución**: Siempre limpiar en `onWillUnmount`:

```
onMounted(() => {
  this.timer = setInterval(this.doSomething, 1000);
});

onWillUnmount(() => {
  if (this.timer) {
```

```
    clearInterval(this.timer);
  }
});
```

Problema: El componente renderiza vacío y luego parpadea cuando los datos cargan **Solución**: Usar `onWillStart` en lugar de `onMounted`:

```
// Esto causa parpadeo
onMounted(async () => {
  this.state.data = await this.fetchData();
});

// Esto previene el parpadeo
onWillStart(async () => {
  this.state.data = await this.fetchData();
});
```

Manejo de Errores con `onError` y Error Boundaries

Todo lo anterior asume que el renderizado sale bien. No siempre es así. Si un componente hijo lanza un error mientras renderiza —una referencia `null`, una respuesta de API con forma inesperada, lo que sea— el comportamiento por defecto de OWL es destruir **toda la aplicación**, no solo el componente que falló. Un bug en un widget pequeño puede tumbar toda la página.

Un **error boundary** (límite de errores) es un componente que captura los errores lanzados en cualquier parte de su subárbol y muestra una interfaz de reemplazo en vez de dejar que el fallo se propague. OWL provee el hook `onError` exactamente para esto.

`onError(callback)` registra una función que OWL llama cada vez que ocurre un error de renderizado o de ciclo de vida en algún descendiente de este componente. **No** captura errores lanzados dentro de manejadores de eventos (un manejador `t-on-click` que lanza un error es territorio de tu propio `try/catch`) — solo errores del mecanismo de renderizado/ciclo de vida.

```
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">Algo salió mal.</div>
        </t>
        <t t-else="" t-slot="default"/>`;

    setup() {
        this.state = useState({ error: false });
        onError(() => {
            this.state.error = true;
        });
    }
}
```

Envuelve cualquier parte del árbol en la que no confíes del todo (un widget que renderiza contenido generado por el usuario, una integración de terceros) con este componente, pasando el contenido riesgoso como su slot por defecto y opcionalmente un slot `fallback` con un mensaje personalizado. Si tu propio manejador `onError` no puede recuperarse, puede relanzar el error para que un boundary

más arriba en el árbol tenga la oportunidad de manejarlo. Eso sí, asegúrate de que la plantilla de fallback nunca lance un error—eso crearía un bucle infinito de fallos.

Otros Hooks de Ciclo de Vida

`onWillStart`, `onMounted`, `onWillUpdateProps` y `onWillUnmount` cubren los hooks que usarás a diario. OWL expone algunos más para necesidades menos comunes y más avanzadas—vale la pena saber que existen para reconocerlos en código que no escribiste tú:

- **`onWillRender()`** — se ejecuta justo antes de que la función de plantilla de un componente se ejecute (padres antes que hijos). Rara vez se necesita directamente.
- **`onRendered()`** — se ejecuta justo después de que la función de plantilla se ejecuta, pero antes de que el resultado se aplique al DOM real. También se necesita rara vez directamente.
- **`onWillPatch()`** — se ejecuta justo antes de que se actualice el DOM de un componente *ya montado* (padres antes que hijos). Útil para leer el estado del DOM —como una posición de scroll— antes de que cambie.
- **`onPatched()`** — se ejecuta justo después de que se actualizó el DOM de un componente ya montado (hijos antes que padres). Este sí lo usarás alguna vez: por ejemplo, para volver a medir el tamaño de un elemento después de que su contenido cambió.

  ```
  onPatched(() => {
      // el DOM acaba de actualizarse con el estado/props más recientes
      this.scrollToBottom();
  });
  ```

- **`onWillDestroy()`** — siempre se llama cuando un componente está siendo destruido, incluso si nunca terminó de montarse. Úsalo para limpieza que debe ocurrir pase lo que pase (a diferencia de `onWillUnmount`, que solo se dispara para componentes que realmente llegaron a montarse).

Para un desarrollador junior, la regla práctica es: recurre primero a los cuatro hooks vistos antes en este capítulo, y solo mira este segundo grupo cuando tengas un problema específico relacionado con actualizaciones del DOM.

Errores Comunes

Error 1: Cargar Datos en `onMounted` en Vez de `onWillStart`

Cargar datos en `onMounted` hace que el componente se renderice vacío primero y luego parpadee cuando llegan los datos. Usa `onWillStart` para los datos que el componente necesita para su primer render.

Error 2: Olvidar la Limpieza en `onWillUnmount`

Los timers, listeners en `window`/`document` e instancias de librerías de terceros iniciados en `onMounted` siguen ejecutándose después de que el componente se destruye, a menos que los limpies en `onWillUnmount` — una fuente clásica de fugas de memoria.

Error 3: Asumir que el DOM Existe Demasiado Pronto

`this.miRef.el` solo está garantizado a partir de `onMounted`. Acceder a un `useRef` dentro de `setup()` u `onWillStart` te dará `undefined`.

Error 4: Recargar Datos en Cada Llamada a `onWillUpdateProps`

`onWillUpdateProps` se ejecuta cuando *cualquier* prop cambia, no solo la que te interesa. Compara siempre el valor anterior y el nuevo (`this.props.x !== nextProps.x`) antes de volver a cargar datos.

Error 5: Olvidar un Error Boundary en Algún Punto del Árbol

Sin un manejador `onError` en algún lugar por encima de él, un solo componente hijo que lanza un error tumba *toda* la aplicación, no solo a sí mismo. Envuelve los subárboles riesgosos (widgets de terceros, cualquier cosa que renderice datos impredecibles) en un componente `ErrorBoundary`.

Ejercicios

Ejercicio 1: Corrige el Parpadeo

Toma un componente que carga sus datos dentro de `onMounted` y mueve la carga a `onWillStart`. Confirma en el navegador que el parpadeo de "cargando" desaparece en el primer render.

Ejercicio 2: Tapa la Fuga

Escribe un componente que inicie un `setInterval` en `onMounted` sin limpiarlo. Abre la consola, monta y desmonta el componente varias veces, y observa que el timer sigue disparándose. Luego corrígelo con `onWillUnmount`.

Ejercicio 3: Recarga Selectiva

Construye un componente con dos props, `userId` y `theme`. En `onWillUpdateProps`, vuelve a cargar los datos del usuario solo cuando `userId` cambia — no cuando cambia `theme`. Añade llamadas a `console.log` para probar que la carga se omite en actualizaciones que solo cambian el tema.

Ejercicio 4: Construye un Error Boundary

Escribe un componente `Buggy` cuyo `setup()` lance un error si una prop `crash` es `true`. Envuélvelo en el componente `ErrorBoundary` de este capítulo y confirma que, cuando `crash` es `true`, el resto de la página sigue funcionando y muestra el mensaje de fallback en vez de una pantalla en blanco.

Resumen

Los hooks del ciclo de vida son la base de componentes OWL robustos. Te permiten:

- **`onWillStart`**: Obtener datos críticos antes del renderizado
- **`onMounted`**: Interactuar con el DOM e inicializar librerías de terceros

- **`onWillUpdateProps`**: Reaccionar a cambios de props y obtener nuevos datos
- **`onWillUnmount`**: Limpiar recursos para prevenir fugas de memoria

Domina estos hooks, y podrás crear componentes que no solo son funcionales, sino también performantes y confiables.

TL;DR: `onWillStart` carga datos antes del primer render, `onMounted` toca el DOM, `onWillUpdateProps` reacciona a nuevas props, y `onWillUnmount` limpia — te saltas el último y filtras memoria.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch10_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

En el próximo capítulo, exploraremos el poder de los hooks personalizados y cómo pueden ayudarte a crear patrones de ciclo de vida reutilizables a través de tu aplicación.

Capítulo 11: Hooks Modernos - useEffect y useRef

> **¿Por qué este capítulo?** Los hooks de ciclo de vida por sí solos te empujan a escribir código organizado por "cuándo", no por "por qué" — la configuración de un timer termina en `onMounted`, su limpieza en `onWillUnmount`, y ambos se separan a medida que el componente crece. Los hooks modernos te permiten mantener todo lo relacionado con una responsabilidad junto.
>
> **¿Qué trata de resolver Odoo con esto?** Los propios componentes del backend de Odoo manejan muchos recursos externos — mediciones del DOM, suscripciones, timers, manejo de foco — y `useEffect`/`useRef` dan un lugar único y predecible para configurarlos y limpiarlos sin esparcir esa lógica entre varios métodos de ciclo de vida.
>
> **Aplicación en la vida real:** Cajas de búsqueda con debounce, foco automático al abrir un diálogo, y sincronizar una librería de gráficos con el estado del componente son cosas que he entregado en addons de clientes usando exactamente los patrones de este capítulo.

En el Capítulo 10, exploramos los hooks de ciclo de vida como `onMounted`, `onWillStart` y `onWillUnmount`. Estos hooks son poderosos e intuitivos, pero el desarrollo moderno en OWL fomenta un enfoque más funcional y flexible usando **hooks modernos**, principalmente `useEffect` y `useRef`.

Los hooks modernos representan un cambio de paradigma en cómo pensamos sobre la lógica de componentes. En lugar de organizar el código por eventos del ciclo de vida ("cuando el componente se monta, haz esto"), lo organizamos por **características** o **responsabilidades** ("todo lo relacionado con este temporizador vive junto"). Este enfoque hace que el código sea más mantenible, comprobable y más fácil de razonar.

Profundicemos en estos hooks modernos y aprendamos cómo pueden hacer tus componentes más poderosos y elegantes.

`useRef`: Más allá de las Referencias DOM

Encontramos brevemente `useRef` en el Capítulo 10 para acceder a elementos DOM. Aunque el acceso al DOM es su caso de uso más común, `useRef` es en realidad un hook mucho más versátil para crear **referencias estables** a cualquier valor.

Entendiendo las Refs

En OWL, una ref es un objeto con una sola propiedad `.el` que apunta al elemento DOM marcado con la directiva `t-ref` correspondiente en tu plantilla. La característica clave de una ref es que es **estable** a través de re-renderizados—siempre apunta a la misma instancia de objeto, y `.el` se mantiene actualizada conforme el DOM cambia.

> **¿Vienes de React?** OWL no tiene "value refs" con una propiedad `.current`. No las necesita: los componentes OWL son instancias de clase, así que para valores mutables que no deben disparar re-renders (IDs de temporizadores, valores anteriores, instancias de librerías) simplemente usas propiedades de instancia normales como `this.timer`. En OWL, `useRef` es exclusivamente para acceder al DOM.

Referencias DOM: El Caso de Uso Clásico

Comencemos con el patrón familiar de referencia DOM:

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
    });

    // Crear refs para múltiples elementos
    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: Enfocado input de nombre");
    }
  }

  focusEmail() {
    if (this.emailInputRef.el) {
      this.emailInputRef.el.focus();
      this.state.focusCount++;
      console.log("FocusManager: Enfocado input de email");
    }
  }

  focusSubmit() {
    if (this.submitButtonRef.el) {
      this.submitButtonRef.el.focus();
      this.state.focusCount++;
      console.log("FocusManager: Enfocado botón de envío");
    }
  }

  handleKeyDown(event) {
    // Navegar entre campos con Tab
    if (event.key === "Tab" && event.shiftKey) {
```

```
      // Manejar navegación personalizada con tab si es necesario
      console.log("FocusManager: Shift+Tab presionado");
    }
  }

  onInputChange(event) {
    this.state.inputValue = event.target.value;

    // Auto-enfocar siguiente campo cuando el actual esté lleno
    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>Demo de Gestor de Enfoque</h5>
        <small class="text-muted">Eventos de enfoque: <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">Nombre</label>
            <input
                type="text"
                class="form-control"
                t-ref="name-input"
                placeholder="Ingresa tu nombre"
                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="Ingresa tu email"
            />
          </div>

          <button
              type="submit"
              class="btn btn-primary me-2"
              t-ref="submit-button">
            Enviar
```

```
          </button>
        </form>

        <div class="mt-3">
          <small class="text-muted">Enfoque Rápido:</small><br/>
          <button class="btn btn-sm btn-outline-secondary me-1" t-on-click="focusName">
            Enfocar Nombre
          </button>
          <button class="btn btn-sm btn-outline-secondary me-1" t-on-click="focusEmail">
            Enfocar Email
          </button>
          <button class="btn btn-sm btn-outline-secondary" t-on-click="focusSubmit">
            Enfocar Envío
          </button>
        </div>
      </div>
    </div>
  </t>
</templates>
```

Almacenando Datos Mutables: Propiedades de Instancia Normales

¿Y qué pasa con los valores mutables que **no deberían desencadenar re-renderizados** cuando cambian—IDs de temporizadores, valores anteriores para comparación, instancias de librerías de terceros? En OWL no necesitas ningún hook para esto. Como tu componente es una instancia de clase, una propiedad normal hace el trabajo:

Ejemplo: Temporizador con Propiedades de Instancia

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

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

  setup() {
    this.state = useState({
      seconds: 0,
      isRunning: false
    });

    // Propiedades de instancia normales - cambiarlas nunca causa un re-render
    this.timer = null;            // ID del temporizador
    this.previousSeconds = 0;     // valor anterior para comparación
    this.startCount = 0;          // cuántas veces se inició el temporizador
  }

  startTimer() {
    if (this.state.isRunning) return;

    console.log("TimerComponent: Iniciando temporizador");
    this.state.isRunning = true;
    this.startCount += 1;

    // Almacenar el ID del temporizador en la instancia - esto no desencadenará
        ↪ re-renderizado
    this.timer = setInterval(() => {
```

```
      this.state.seconds++;

      // Registrar cada 10 segundos usando comparación de valor anterior
      if (this.state.seconds % 10 === 0 && this.state.seconds !== this.previousSeconds)
          ↪ {
        console.log(`TimerComponent: ${this.state.seconds} segundos transcurridos`);
        this.previousSeconds = this.state.seconds;
      }
    }, 1000);
  }

  stopTimer() {
    if (!this.state.isRunning) return;

    console.log("TimerComponent: Deteniendo temporizador");
    this.state.isRunning = false;

    // Limpiar temporizador usando ID almacenado
    if (this.timer) {
      clearInterval(this.timer);
      this.timer = null;
    }
  }

  resetTimer() {
    this.stopTimer();
    this.state.seconds = 0;
    this.previousSeconds = 0;
    console.log("TimerComponent: Temporizador reiniciado");
  }

  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')}`;
  }
}
```

La regla general: **los datos reactivos que la plantilla muestra van en `useState`; todo lo demás es una propiedad de instancia normal; `useRef` es solo para elementos DOM.**

`useEffect`: La Navaja Suiza de los Efectos Secundarios

El hook `useEffect` es el hook más poderoso y flexible en OWL moderno. Reemplaza múltiples hooks de ciclo de vida con una sola API unificada que agrupa la lógica relacionada.

> **¿Qué es un "efecto secundario"?** Es cualquier cosa que hace un código que va más allá de calcular su valor de retorno: iniciar un temporizador, agregar un listener de eventos, llamar al servidor, cambiar `document.title`. El renderizado debería ser un cálculo puro de "estado entra, DOM sale"; `useEffect` es donde va todo lo que no lo es.

Entendiendo las Dependencias

La magia de `useEffect` radica en su segundo parámetro: una **función de dependencias** que devuelve la lista de valores de los que depende el efecto. OWL llama a esta función en cada renderizado y re-ejecuta el efecto solo cuando alguno de los valores devueltos cambió.

> **Cuidado si conoces React:** en React las dependencias son un array plano (`[state.count]`). En OWL son una *función que devuelve un array* (`() => [state.count]`). Pasar un array plano es un bug de migración muy común.

```
// Se ejecuta una vez después del montaje, limpieza al desmontar (como onMounted +
    ↪ onWillUnmount)
useEffect(
  () => {
    console.log("Componente montado");
  },
  () => []
);

// Se ejecuta después de cada renderizado (generalmente no es lo que quieres) - omite la
    ↪ función de dependencias
useEffect(() => {
  console.log("Componente renderizado");
});

// Se ejecuta cuando valores específicos cambian
useEffect(
  () => {
    console.log("Count o name cambiaron");
  },
  () => [this.state.count, this.state.name]
);
```

Patrón 1: Efectos Solo de Montaje (Reemplazando onMounted)

Creemos un componente que configura múltiples cosas cuando se monta:

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");

    // Efecto solo de montaje - se ejecuta una vez después del montaje del componente
    useEffect(() => {
      console.log("Dashboard: Componente montado, inicializando...");

      // 1. Configurar reloj
      const clockInterval = setInterval(() => {
        this.state.currentTime = new Date().toLocaleTimeString();
      }, 1000);

      // 2. Enfocar el título
      if (this.titleRef.el) {
        this.titleRef.el.focus();
      }

      // 3. Cargar datos iniciales
      this.loadDashboardData();

      // 4. Configurar listener de enfoque de ventana
      const handleWindowFocus = () => {
        console.log("Dashboard: Ventana enfocada, actualizando datos");
        this.loadDashboardData();
      };

      window.addEventListener('focus', handleWindowFocus);

      // Función de limpieza - se ejecuta cuando el componente se desmonta
      return () => {
        console.log("Dashboard: Limpiando efectos de montaje");
        clearInterval(clockInterval);
        window.removeEventListener('focus', handleWindowFocus);
      };
    }, () => []); // Lista de dependencias vacía = ejecutar una vez en el montaje
  }

  async loadDashboardData() {
    try {
      console.log("Dashboard: Cargando estadísticas del dashboard");

      // Simular carga de múltiples estadísticas en paralelo
      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 // Cálculo de ingresos simulado
      };

      this.state.loading = false;
      console.log("Dashboard: Estadísticas cargadas exitosamente");

    } catch (error) {
      console.error("Dashboard: Falló la carga de estadísticas:", error);
      this.state.loading = false;
    }
  }
}
```

Patrón 2: Efectos Reactivos (Reemplazando onWillUpdateProps)

Los efectos pueden reaccionar a cambios específicos de estado o props:

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");

    // Efecto que reacciona a cambios en la prop userId
    useEffect(() => {
      if (!this.props.userId) {
        console.log("UserProfile: No se proporcionó ID de usuario");
        this.state.user = null;
        return;
      }

      console.log(`UserProfile: Cargando datos de usuario para ID ${this.props.userId}`);
      this.loadUserData(this.props.userId);

    }, () => [this.props.userId]); // Se ejecuta cuando userId cambia

    // Efecto separado para registrar intentos de carga
    useEffect(() => {
      if (this.state.loadCount > 0) {
        console.log(`UserProfile: Intento de carga #${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: Cargado exitosamente ${users[0].name}`);
      } else {
        throw new Error(`Usuario ${userId} no encontrado`);
      }

    } catch (error) {
      console.error(`UserProfile: Falló la carga del usuario ${userId}:`, error);
      this.state.error = error.message;
      this.state.user = null;
    } finally {
      this.state.loading = false;
    }
  }
}
```

Patrón 3: Múltiples Efectos Independientes

Una de las mayores ventajas de `useEffect` es que puedes tener múltiples efectos, cada uno manejando una responsabilidad específica:

JavaScript (`notification_center.js`):

Primero, el componente configura su estado y su primer efecto, que solo se preocupa de si el navegador está conectado:

```
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");

// Efecto 1: Gestión de conexión
useEffect(() => {
  console.log("NotificationCenter: Configurando monitoreo de conexión");

  const checkConnection = () => {
    this.state.connectionStatus = navigator.onLine ? 'connected' : 'disconnected';
  };

  // Verificación inicial
  checkConnection();

  // Configurar listeners
  window.addEventListener('online', checkConnection);
  window.addEventListener('offline', checkConnection);

  return () => {
    window.removeEventListener('online', checkConnection);
    window.removeEventListener('offline', checkConnection);
  };
}, () => []);
```

Después, un segundo efecto observa `connectionStatus` y solo empieza a sondear notificaciones nuevas cuando el navegador realmente está conectado — nota que su función de dependencias devuelve `[this.state.connectionStatus]`, así que se re-ejecuta cada vez que ese valor cambia:

```
// Efecto 2: Actualización periódica de notificaciones
useEffect(() => {
  if (this.state.connectionStatus !== 'connected') {
    console.log("NotificationCenter: Saltando actualización - no conectado");
    return;
  }

  console.log("NotificationCenter: Configurando actualización periódica");

  const refreshInterval = setInterval(() => {
    this.refreshNotifications();
  }, 30000); // Actualizar cada 30 segundos

  // Carga inicial
  this.refreshNotifications();

  return () => {
    clearInterval(refreshInterval);
  };
}, () => [this.state.connectionStatus]); // Re-ejecutar cuando cambie el estado de
↪  conexión
```

Finalmente, un tercer efecto, completamente independiente, recalcula el contador de no leídas y mantiene sincronizado el título de la pestaña cada vez que cambia la lista de notificaciones — no sabe ni le importa la lógica de conexión de arriba:

```
// Efecto 3: Cálculo de contador de no leídas
useEffect(() => {
```

```
      const unread = this.state.notifications.filter(n => !n.is_read).length;
      this.state.unreadCount = unread;

      console.log(`NotificationCenter: ${unread} notificaciones no leídas`);

      // Actualizar título del navegador si hay notificaciones no leídas
      if (unread > 0) {
        document.title = `(${unread}) Odoo - Notificaciones`;
      } else {
        document.title = "Odoo";
      }

      return () => {
        // Limpiar título en el desmontaje
        document.title = "Odoo";
      };
    }, () => [this.state.notifications.length]);
  }

  async refreshNotifications() {
    try {
      console.log("NotificationCenter: Actualizando notificaciones");

      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: Falló la actualización de notificaciones:",
          ↪  error);
    }
  }

  get currentUserId() {
    // Esto vendría de un servicio de usuario en una app real
    return 1; // Simplificado para el ejemplo
  }
}
```

Tres efectos, tres responsabilidades, cero lógica enredada — ese es el resultado de organizar por característica en vez de por momento del ciclo de vida.

Patrones Avanzados de useEffect

Patrón: Efecto con Lógica Asíncrona

```
useEffect(() => {
  let cancelled = false;

  const fetchData = async () => {
    try {
      const result = await this.orm.searchRead("some.model", [], []);

      // Solo actualizar estado si el efecto no ha sido cancelado
      if (!cancelled) {
        this.state.data = result;
      }
    } catch (error) {
      if (!cancelled) {
        console.error("Falló la obtención:", error);
      }
    }
  };

  fetchData();

  // Limpieza: marcar como cancelado para prevenir actualizaciones de estado
  return () => {
    cancelled = true;
  };
}, () => [this.state.someDependency]);
```

Patrón: Efecto con Debounce

Debounce (o "postergar") significa retrasar un trabajo hasta que una ráfaga de cambios que lo disparan se detenga durante un tiempo determinado — en vez de buscar en cada tecla presionada, esperas a que el usuario deje de escribir por 300ms.

```
useEffect(() => {
  const timeoutId = setTimeout(() => {
    // Realizar operación costosa solo después de que el valor haya estado estable por
        ↪ 300ms
    this.performSearch(this.state.searchTerm);
  }, 300);

  return () => {
    clearTimeout(timeoutId);
  };
}, () => [this.state.searchTerm]);
```

Combinando useRef y useEffect: Patrones Poderosos

Cuando combinas `useRef` y `useEffect`, desbloqueas algunos patrones muy poderosos:

Patrón: Comparación de Valor Anterior

```
setup() {
  this.state = useState({ count: 0 });
  this.previousCount = undefined; // propiedad de instancia normal
```

```
  useEffect(
    () => {
      console.log(`Count cambió de ${this.previousCount} a ${this.state.count}`);

      // Almacenar valor actual para próxima comparación
      this.previousCount = this.state.count;
    },
    () => [this.state.count]
  );
}
```

Patrón: Integración con Biblioteca de Terceros

```
setup() {
  this.chartCanvasRef = useRef("chart-canvas"); // ref DOM (t-ref="chart-canvas")
  this.chart = null;                            // propiedad normal para la instancia de
      ↪  la librería
  this.state = useState({ data: [] });

  // Configurar gráfico en el montaje
  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();
        }
      };
    },
    () => []
  );

  // Actualizar gráfico cuando los datos cambien
  useEffect(
    () => {
      if (this.chart) {
        this.chart.data = this.processChartData();
        this.chart.update();
      }
    },
    () => [this.state.data.length]
  );
}
```

Guía de Migración: De Hooks de Ciclo de Vida a Hooks Modernos

Si tienes componentes existentes usando hooks de ciclo de vida, aquí tienes cómo migrarlos:

Antes: Hooks de Ciclo de Vida

```
setup() {
  this.state = useState({ data: null });
  this.orm = useService("orm");

  onWillStart(async () => {
    this.state.data = await this.orm.searchRead("model", [], []);
  });

  onMounted(() => {
    document.title = "Mi Componente";
    this.timer = setInterval(this.refresh, 5000);
  });

  onWillUnmount(() => {
    document.title = "Odoo";
    if (this.timer) {
      clearInterval(this.timer);
    }
  });
}
```

Después: Hooks Modernos

```
setup() {
  this.state = useState({ data: null });
  this.orm = useService("orm");

  // Mantén onWillStart para datos pre-renderizado: useEffect se ejecuta DESPUÉS
  // del montaje, así que reemplazarlo con un efecto traería de vuelta el parpadeo
  onWillStart(async () => {
    this.state.data = await this.orm.searchRead("model", [], []);
  });

  // Efectos de montaje (reemplaza onMounted + onWillUnmount)
  useEffect(
    () => {
      document.title = "Mi Componente";
      const timer = setInterval(this.refresh, 5000);

      return () => {
        document.title = "Odoo";
        clearInterval(timer);
      };
    },
    () => []
  );
}
```

Nota que `onWillStart` se queda: es el único hook que puede retrasar el **primer** renderizado, así que no tiene equivalente en `useEffect`.

Hooks de Entorno: `useEnv`, `useSubEnv`, `useChildSubEnv`

Las props son geniales para pasar datos un nivel hacia abajo, de un padre a su hijo directo. Pero ¿qué pasa con datos que docenas de componentes esparcidos por todo el árbol necesitan—el usuario actual, una función de traducción, feature flags? Pasarlos como prop a través de cada componente intermedio (**prop drilling**, el mismo problema que motiva el store de estado global del Capítulo 15) se vuelve doloroso rápidamente.

La respuesta de OWL es el **entorno** (`env`): un objeto plano compartido por todos los componentes del árbol, definido una vez al montar la aplicación y disponible en todas partes sin tener que pasarlo por props.

`useEnv()` lee el entorno del componente actual:

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

class UserBadge extends Component {
    static template = xml`<span t-esc="env.currentUser.name"/>`;

    setup() {
        this.env = useEnv();
    }
}
```

A veces un componente quiere agregar o sobrescribir algo en el entorno para su propio subárbol—un tema, una configuración específica—sin cambiarlo para toda la app. Dos hooks hacen esto, y la diferencia entre ellos importa:

- **`useSubEnv(nuevosValores)`** — mezcla `nuevosValores` en el entorno para **este componente y todos sus hijos**.
- **`useChildSubEnv(nuevosValores)`** — mezcla `nuevosValores` en el entorno solo para **los hijos**, dejando el `env` propio de este componente sin tocar.

```
setup() {
    // Todo componente por debajo de este (hijos, nietos, ...) ahora ve
    // env.theme === "dark". Este componente también, porque useSubEnv
    // también afecta a quien lo llama.
    useSubEnv({ theme: "dark" });
}
```

Usa `useChildSubEnv` cuando un componente quiere configurar a sus descendientes sin implicar que la misma configuración le aplica a sí mismo—por ejemplo, un componente `<Section title="...">` que define un nivel de encabezado para sus hijos sin convertirse él mismo en un encabezado.

`useExternalListener`: Escuchar Fuera del Componente

El Capítulo 1 agregó un listener de `click` a mano con `addEventListener`, y hubo que recordar llamar `removeEventListener` en `onWillUnmount` para evitar una fuga. `useExternalListener` hace ambos pasos por ti en una sola llamada, que es exactamente lo que quieres para escuchar en `window` o `document`—objetivos fuera del DOM propio del componente que OWL no gestiona.

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

class DropdownMenu extends Component {
    setup() {
        // Se agrega automáticamente al montar y se quita al desmontar-
        // sin par onMounted/onWillUnmount que escribir ni olvidar.
        useExternalListener(window, "click", this.closeIfOutside.bind(this));
    }

    closeIfOutside(ev) {
        if (!this.el.contains(ev.target)) {
            this.state.open = false;
        }
    }
}
```

Recibe el objetivo (`window`, `document`, o cualquier nodo DOM), el nombre del evento y un manejador, y reenvía cualquier argumento extra a `addEventListener`—así que `useExternalListener(window, "keydown", this.onKeyDown, { capture: true })` también funciona.

Otro hook que vale la pena conocer, aunque rara vez lo llamarás directamente: `useComponent()` devuelve la instancia del componente actual. Existe principalmente como bloque de construcción para escribir tus *propios* hooks personalizados que necesiten acceder al componente que los llama—el código de aplicación casi nunca lo necesita.

Mejores Prácticas y Errores Comunes

Qué Hacer

1. **Agrupar lógica relacionada** - Poner configuración y limpieza para la misma característica en un efecto
2. **Usar múltiples efectos** - Separar responsabilidades en diferentes efectos
3. **Incluir todas las dependencias** - La función de dependencias debe devolver todos los valores reactivos que el efecto usa
4. **Retornar funciones de limpieza** - Prevenir memory leaks limpiando recursos
5. **Usar el almacenamiento correcto** - Datos reactivos de la plantilla en `useState`, valores mutables no reactivos (IDs de temporizadores, valores anteriores, instancias de librerías) como propiedades de instancia normales, `useRef` solo para elementos DOM

Qué No Hacer

1. **No pasar un array plano como dependencias** - OWL espera una función que devuelve un array (`() => [..]`), no el array en sí
2. **No usar efectos para todo** - Alguna lógica pertenece a event handlers, no a efectos
3. **No reemplazar `onWillStart` con un efecto** - Los efectos se ejecutan después del montaje; solo `onWillStart` puede retrasar el primer renderizado
4. **No sobre-complicar** - A veces los hooks de ciclo de vida son más claros para casos simples

Error Común: Array en Lugar de Función

```
// MALO: array plano estilo React - OWL no rastreará estas dependencias
useEffect(() => {
```

```
  this.syncWithServer(this.state.filter);
}, [this.state.filter]);

// BUENO: una función que devuelve el array de dependencias
useEffect(
  () => {
    this.syncWithServer(this.state.filter);
  },
  () => [this.state.filter]
);
```

¿Por qué una función? OWL la llama en **cada renderizado** para obtener valores frescos y compararlos con los anteriores. Un array plano se evaluaría una sola vez, cuando `setup()` se ejecutó.

Error Común: Esperar que `useSubEnv` No Afecte a Quien lo Llama

```
// El env.theme de ESTE MISMO componente también se vuelve "dark" aquí,
// no solo el de sus hijos-useSubEnv siempre incluye a quien lo llama.
useSubEnv({ theme: "dark" });
```

Si quieres configurar a los descendientes sin cambiar nada para el componente actual, usa `useChildSubEnv` en su lugar. Confundir uno con el otro es un bug fácil de pasar por alto: todo lo que está por debajo del componente se comporta correctamente, pero el renderizado del propio componente cambia de forma inesperada.

Resumen

Los hooks modernos representan una evolución significativa en cómo escribimos componentes:

- **`useRef`** crea referencias estables a elementos DOM (y las propiedades de instancia cubren cualquier otro valor mutable)
- **`useEffect`** maneja todos los efectos secundarios con control preciso sobre cuándo se ejecutan
- **Las funciones de dependencias** hacen que los efectos sean predecibles y eficientes
- **Funciones de limpieza** previenen memory leaks y conflictos de recursos

La perspectiva clave es organizacional: en lugar de pensar "qué pasa cuando el componente se monta", piensa "qué necesita esta característica para funcionar correctamente". Agrupa toda la lógica para cada característica—configuración, actualizaciones y limpieza—en efectos enfocados.

Este enfoque hace que tus componentes sean más mantenibles, comprobables y más fáciles de entender. En el próximo capítulo, exploraremos cómo comunicarnos con el servidor usando servicios, construyendo sobre la sólida base de hooks modernos que hemos establecido aquí.

TL;DR: `useRef` te da un identificador estable a un elemento DOM o un valor mutable; `useEffect` ejecuta (y limpia) efectos secundarios según una función de dependencias explícita, así que agrupas la configuración y el desmontaje de una responsabilidad en un solo lugar en vez de esparcirlos entre hooks de ciclo de vida.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch11_ex1`. Las instrucciones de instalación están en el README del repositorio.

Ejercicios

1. **Enfocar al montar:** Construye un componente pequeño con un input de texto y un `useRef`. Usa `useEffect` con una función de dependencias vacía (`() => []`) para enfocar el input tan pronto como el componente se monte.
2. **Corrige el bug de dependencias:** Toma este fragmento roto y corrígelo para que el efecto realmente se re-ejecute cuando `state.query` cambie:

```
useEffect(() => {
  console.log("Buscando", this.state.query);
}, [this.state.query]);
```

3. **Contador con debounce:** Agrega un `useEffect` que registre `"¡Estable!"` en la consola solo después de que `state.count` no haya cambiado durante 500ms (pista: `setTimeout` en el efecto, `clearTimeout` en la función de limpieza — la misma forma que el ejemplo de búsqueda con debounce de arriba).

¿Qué Sigue?

Ahora tienes componentes que manejan sus propias referencias al DOM y efectos secundarios — pero los addons reales de Odoo rara vez viven aislados del servidor. El Capítulo 12 cubre `orm` y `rpc`, los servicios que permiten que tus componentes lean y escriban datos de negocio reales.

Capítulo 12: Comunicación con el Servidor usando useService

> **¿Por qué este capítulo?** Todo addon de Odoo no trivial eventualmente necesita leer o escribir registros reales — un componente que solo gestiona estado local es un juguete. Este capítulo es donde tus componentes empiezan a hablar con la base de datos real.
>
> **¿Qué trata de resolver Odoo con esto?** Odoo necesita una forma consistente y segura de que el código frontend llegue al ORM y a controladores personalizados sin que cada desarrollador arme a mano sus propias llamadas AJAX, manejo de errores y validaciones de seguridad.
>
> **Aplicación en la vida real:** Cada widget de dashboard, acción de kanban personalizada y formulario de portal que he construido para un cliente eventualmente se reduce a llamadas `orm.searchRead/create/write` exactamente como las de este capítulo.

En los capítulos anteriores, hemos construido componentes que pueden gestionar estado, responder a eventos y manejar escenarios complejos de ciclo de vida. Pero una aplicación web moderna no está completa sin la capacidad de comunicarse con el servidor. Tus hermosos componentes OWL necesitan obtener datos reales, guardar cambios de usuario e integrarse perfectamente con el backend de Odoo.

Aquí es donde entra el **sistema de servicios**. La arquitectura de servicios de Odoo proporciona una forma limpia y consistente para que tus componentes frontend se comuniquen con el backend de Python, muestren notificaciones, activen acciones y mucho más.

La puerta de entrada a este poderoso sistema es el hook `useService`—el puente de tus componentes hacia todo el ecosistema de Odoo.

Entendiendo la Arquitectura de Servicios de Odoo

Los servicios en Odoo son **objetos singleton** que proporcionan funcionalidad específica a través de toda tu aplicación. Están diseñados para ser:

- **Centralizados**: Una instancia por tipo de servicio
- **Consistentes**: La misma API en todas partes de la app
- **Integrados**: Conectados perfectamente al backend de Odoo
- **Comprobables**: Fáciles de simular y probar

Piensa en los servicios como las "utilidades" de tu aplicación—herramientas especializadas que manejan responsabilidades específicas para que tus componentes puedan enfocarse en su trabajo principal: renderizar UI y manejar interacciones de usuario.

El Hook `useService`

El hook `useService` es notablemente simple:

```
import { useService } from "@web/core/utils/hooks";

// Dentro del setup() de tu componente
const serviceName = useService("service_name");
```

¡Eso es todo! Obtienes una instancia de servicio completamente configurada lista para usar. Exploremos los servicios más importantes que usarás diariamente.

El Servicio `orm`: Tu Puerta de Entrada a la Base de Datos

El servicio `orm` es tu herramienta principal para operaciones de base de datos. Proporciona una API limpia basada en Promises que refleja los métodos ORM de Python de Odoo, haciendo que la comunicación con el servidor se sienta natural e intuitiva.

Operaciones CRUD Básicas

Construyamos un componente completo de gestión de productos que demuestre todas las operaciones básicas del ORM:

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: ""
      }
    });

    // Obtener el servicio ORM
    this.orm = useService("orm");

    // Cargar datos iniciales cuando el componente se monte
    useEffect(
      () => {
        this.loadInitialData();
      },
      () => []
```

```
    );
  }

  async loadInitialData() {
    try {
      console.log("ProductManager: Cargando datos iniciales");
      this.state.loading = true;
      this.state.error = null;

      // Cargar productos y categorías en paralelo para mejor rendimiento
      const [products, categories] = await Promise.all([
        this.loadProducts(),
        this.loadCategories()
      ]);

      this.state.products = products;
      this.state.categories = categories;

      console.log(`ProductManager: Cargados ${products.length} productos y
          ↪ ${categories.length} categorías`);

    } catch (error) {
      console.error("ProductManager: Falló la carga de datos iniciales:", error);
      this.state.error = "Falló la carga de datos. Por favor, actualiza la página.";
    } finally {
      this.state.loading = false;
    }
  }

  async loadProducts() {
    // searchRead: Buscar registros y leer sus campos en una sola llamada
    return await this.orm.searchRead(
      "product.product",                    // Nombre del modelo
      [["sale_ok", "=", true]],             // Dominio (filtros de búsqueda)
      ["id", "name", "list_price", "categ_id", "qty_available"], // Campos a leer
      {
        order: "name asc",                  // Orden de clasificación
        limit: 100                          // Máximo de registros
      }
    );
  }

  async loadCategories() {
    return await this.orm.searchRead(
      "product.category",
      [],                                   // Dominio vacío = todos los registros
      ["id", "name"],
      { order: "name asc" }
    );
  }

  async createProduct() {
    if (!this.state.formData.name.trim()) {
      this.state.error = "El nombre del producto es requerido";
      return;
```

```
    }

    try {
      console.log("ProductManager: Creando nuevo producto", this.state.formData);

      // create: Crear nuevos registros
      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: Producto creado con ID ${newProductIds[0]}`);

      // Recargar productos para mostrar el nuevo
      await this.refreshProducts();

      // Reiniciar formulario
      this.resetForm();

      this.state.error = null;

    } catch (error) {
      console.error("ProductManager: Falló la creación del producto:", error);
      this.state.error = `Falló la creación del producto: ${error.message}`;
    }
  }

  async updateProduct(productId, updates) {
    try {
      console.log(`ProductManager: Actualizando producto ${productId}`, updates);

      // write: Actualizar registros existentes
      await this.orm.write(
        "product.product",
        [productId],                    // Lista de IDs de registros a actualizar
        updates                         // Diccionario de actualizaciones de campos
      );

      console.log(`ProductManager: Producto ${productId} actualizado exitosamente`);

      // Actualizar la lista de productos
      await this.refreshProducts();

    } catch (error) {
      console.error(`ProductManager: Falló la actualización del producto ${productId}:`,
          ↪ error);
      this.state.error = `Falló la actualización del producto: ${error.message}`;
    }
  }
```

```
async deleteProduct(productId) {
  try {
    console.log(`ProductManager: Eliminando producto ${productId}`);

    // unlink: Eliminar registros
    await this.orm.unlink("product.product", [productId]);

    console.log(`ProductManager: Producto ${productId} eliminado exitosamente`);

    // Remover del estado local inmediatamente para mejor UX
    this.state.products = this.state.products.filter(p => p.id !== productId);

    // Limpiar selección si el producto eliminado estaba seleccionado
    if (this.state.selectedProduct?.id === productId) {
      this.state.selectedProduct = null;
    }

  } catch (error) {
    console.error(`ProductManager: Falló la eliminación del producto ${productId}:`,
        ↪  error);
    this.state.error = `Falló la eliminación del producto: ${error.message}`;
  }
}

async refreshProducts() {
  try {
    const products = await this.loadProducts();
    this.state.products = products;
    console.log("ProductManager: Productos actualizados exitosamente");
  } catch (error) {
    console.error("ProductManager: Falló la actualización de productos:", error);
  }
}

// Métodos auxiliares para manejo de formularios
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;
  }

  // Acciones rápidas para gestión de productos
  async quickUpdatePrice(productId, newPrice) {
    await this.updateProduct(productId, { list_price: newPrice });
  }

  async toggleProductAvailability(productId, currentStatus) {
    await this.updateProduct(productId, { sale_ok: !currentStatus });
  }

  // Propiedades computadas para el template
  get formattedProducts() {
    return this.state.products.map(product => ({
      ...product,
      formattedPrice: `$${product.list_price.toFixed(2)}`,
      categoryName: product.categ_id ? product.categ_id[1] : "Sin Categoría",
      stockStatus: product.qty_available > 0 ? "En Stock" : "Agotado"
    }));
  }

  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">
      <!-- Encabezado -->
      <div class="d-flex justify-content-between align-items-center mb-4">
        <h2>Gestor de Productos</h2>
        <button
            class="btn btn-outline-secondary btn-sm"
            t-on-click="refreshProducts"
            t-att-disabled="state.loading">
          <i class="fa fa-refresh"></i> Actualizar
        </button>
      </div>

      <!-- Mostrar Errores -->
      <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>

<!-- Estado de Carga -->
<t t-if="state.loading">
  <div class="text-center py-5">
    <div class="spinner-border text-primary" role="status">
      <span class="visually-hidden">Cargando productos...</span>
    </div>
    <p class="mt-2 text-muted">Cargando productos y categorías...</p>
  </div>
</t>

<!-- Contenido Principal -->
<t t-else="">
  <div class="row">
    <!-- Formulario de Producto -->
    <div class="col-md-4">
      <div class="card">
        <div class="card-header">
          <h5 class="mb-0">
            <t t-if="state.selectedProduct">Editar Producto</t>
            <t t-else="">Nuevo Producto</t>
          </h5>
        </div>
        <div class="card-body">
          <form t-on-submit.prevent="createProduct">
            <!-- Nombre del Producto -->
            <div class="mb-3">
              <label class="form-label">Nombre del Producto *</label>
              <input
                  type="text"
                  class="form-control"
                  t-model="state.formData.name"
                  placeholder="Ingresa el nombre del producto"
                  required
              />
            </div>

            <!-- Precio -->
            <div class="mb-3">
              <label class="form-label">Precio</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>

            <!-- Categoría -->
            <div class="mb-3">
```

```
                <label class="form-label">Categoría</label>
                <select
                    class="form-select"
                    t-model="state.formData.categ_id">
                  <option value="">Seleccionar Categoría</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>

              <!-- Botones de Acción -->
              <div class="d-flex gap-2">
                <button
                    type="submit"
                    class="btn btn-primary flex-fill"
                    t-att-disabled="!isFormValid">
                  <t t-if="state.selectedProduct">Actualizar Producto</t>
                  <t t-else="">Crear Producto</t>
                </button>

                <button
                    type="button"
                    class="btn btn-outline-secondary"
                    t-on-click="resetForm">
                  Limpiar
                </button>
              </div>
            </form>
          </div>
        </div>
      </div>

      <!-- Lista de Productos -->
      <div class="col-md-8">
        <div class="card">
          <div class="card-header">
            <h5 class="mb-0">
              Productos (<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 se encontraron productos. ¡Crea tu primer producto!</p>
              </div>
            </t>
            <t t-else="">
              <div class="table-responsive">
                <table class="table table-hover mb-0">
                  <thead class="table-light">
                    <tr>
                      <th>Nombre</th>
```

```
                    <th>Categoría</th>
                    <th>Precio</th>
                    <th>Stock</th>
                    <th class="text-end">Acciones</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="Editar">
                            <i class="fa fa-edit"></i>
                          </button>
                          <button
                              class="btn btn-outline-danger"
                              t-on-click="() => this.deleteProduct(product.id)"
                              title="Eliminar">
                            <i class="fa fa-trash"></i>
                          </button>
                        </div>
                      </td>
                    </tr>
                  </t>
                </tbody>
              </table>
            </div>
          </t>
        </div>
      </div>
    </div>
  </t>
```

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

Este ejemplo demuestra todas las operaciones clave del ORM:

- **searchRead**: Buscar y leer registros en una operación
- **create**: Crear nuevos registros
- **write**: Actualizar registros existentes
- **unlink**: Eliminar registros

Operaciones Avanzadas del ORM

El servicio `orm` proporciona métodos más avanzados para escenarios complejos:

Usando `read` para Registros Específicos:

```
// Leer registros específicos por ID
const products = await this.orm.read(
  "product.product",
  [1, 2, 3],                          // IDs de registros específicos
  ["name", "list_price", "description"] // Campos a leer
);
```

Usando `search` y `searchCount`:

```
// Obtener solo IDs de registros que coincidan con criterios
const productIds = await this.orm.search(
  "product.product",
  [["sale_ok", "=", true]],
  { limit: 5, offset: 10 }
);

// Contar registros sin obtenerlos
const totalProducts = await this.orm.searchCount(
  "product.product",
  [["sale_ok", "=", true]]
);
```

Llamando Métodos Personalizados del Modelo:

```
// Llamar cualquier método en tu modelo Python
const result = await this.orm.call(
  "product.product",      // Nombre del modelo
  "my_custom_method",     // Nombre del método
  [recordId],            // Args (generalmente IDs de registros)
  {                      // Kwargs
    param1: "value1",
    param2: "value2"
  }
);
```

El Servicio rpc: Comunicación Directa con Controladores

Mientras que el servicio `orm` es perfecto para operaciones de modelo, a veces necesitas llamar rutas de controlador directamente. El servicio `rpc` (abreviatura de **Remote Procedure Call**, "llamada a procedimiento remoto": invocar una función que en realidad se ejecuta en el servidor, como si fuera local) proporciona esta capacidad:

Ejemplo JavaScript:

```
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: Generando reporte personalizado");

      // Llamar tu ruta de controlador personalizada
      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: Reporte generado exitosamente");

    } catch (error) {
      console.error("CustomReportGenerator: Falló la generación del reporte:", error);
      // Manejar error apropiadamente
    } finally {
      this.state.loading = false;
    }
  }

  async downloadReportPDF() {
    try {
      console.log("CustomReportGenerator: Descargando reporte PDF");
```

```
      // Esto podría retornar un blob o URL para descarga
      const pdfData = await this.rpc("/my_module/generate_report", {
        filters: this.state.filters,
        format: "pdf"
      });

      // Manejar lógica de descarga PDF aquí

    } catch (error) {
      console.error("CustomReportGenerator: Falló la descarga PDF:", error);
    }
  }
}
```

Manejo de Errores y Retroalimentación de Usuario

Las aplicaciones profesionales necesitan un manejo robusto de errores. Aquí tienes cómo manejar escenarios comunes:

Patrón de Manejo Completo de Errores:

```
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("Falló la operación de base de datos:", error);

    // Diferentes tipos de error necesitan diferente manejo
    if (error.code === 403) {
      this.state.error = "No tienes permisos para acceder a estos datos.";
    } else if (error.code === 404) {
      this.state.error = "Los datos solicitados no fueron encontrados.";
    } else if (error.message.includes("network")) {
      this.state.error = "Error de red. Por favor verifica tu conexión.";
    } else {
      this.state.error = `Falló la operación: ${error.message}`;
    }

  } finally {
    this.state.loading = false;
  }
}
```

Patrón de Actualizaciones Optimistas:

```
async quickUpdate(recordId, field, value) {
  // Almacenar valor original para rollback
  const originalData = [...this.state.records];

  try {
    // Actualizar UI inmediatamente (optimista)
    const record = this.state.records.find(r => r.id === recordId);
    if (record) {
      record[field] = value;
    }

    // Luego actualizar servidor
    await this.orm.write("model.name", [recordId], { [field]: value });

    console.log("Actualización rápida exitosa");

  } catch (error) {
    // Rollback UI en error
    this.state.records = originalData;
    console.error("Falló la actualización rápida, rollback realizado:", error);

    // Mostrar mensaje de error amigable
    this.showError(`Falló la actualización de ${field}. Por favor intenta de nuevo.`);
  }
}
```

Patrones de Integración de Servicios

Patrón 1: Composición de Servicios

```
setup() {
  // Múltiples servicios trabajando juntos
  this.orm = useService("orm");
  this.rpc = useService("rpc");
  this.notification = useService("notification");
  this.action = useService("action");

  // Operación compuesta usando múltiples servicios
  this.performComplexOperation = async () => {
    try {
      // 1. Guardar datos vía ORM
      const [recordId] = await this.orm.create("model.name", [this.state.formData]);

      // 2. Activar procesamiento del lado del servidor vía RPC
      await this.rpc("/custom/process_record", { record_id: recordId });

      // 3. Mostrar notificación de éxito
      this.notification.add("¡Registro creado y procesado exitosamente!", {
        type: "success"
      });

      // 4. Navegar al nuevo registro
```

```
      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("Falló la operación. Por favor intenta de nuevo.", {
        type: "danger"
      });
    }
  };
}
```

Patrón 2: Abstracción de Servicios

```
setup() {
  this.orm = useService("orm");

  // Crear capa de abstracción para tu dominio específico
  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);
    }
  };
}
```

Consideraciones de Rendimiento

Operaciones en Lote

```
// MALO: Múltiples llamadas individuales
for (const productId of productIds) {
  await this.orm.write("product.product", [productId], { active: false });
}

// BUENO: Una sola llamada en lote
await this.orm.write("product.product", productIds, { active: false });
```

Selección Eficiente de Campos

```
// MALO: Cargar datos innecesarios
const products = await this.orm.searchRead("product.product", [], []);

// BUENO: Solo cargar lo que necesitas
const products = await this.orm.searchRead(
  "product.product",
  [],
  ["id", "name", "list_price"] // Solo los campos que realmente usas
);
```

Patrón de Cache Inteligente

```
setup() {
  this.orm = useService("orm");
  this.cache = new Map();

  this.getCachedData = async (cacheKey, fetcher) => {
    if (this.cache.has(cacheKey)) {
      console.log(`Usando datos en cache para ${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"])
    );
  };
}
```

Errores Comunes y Soluciones

Error 1: Asumir que `searchRead` Devuelve un Objeto Envoltorio

```
// Malo - algunos ORMs envuelven los resultados en un objeto; el de Odoo no
const { records } = await this.orm.searchRead("res.partner", [], ["name"]);

// Correcto - searchRead resuelve directamente a un array de registros
const records = await this.orm.searchRead("res.partner", [], ["name"]);
```

Error 2: Olvidar `await`

```
// Malo - todo método de orm devuelve una Promise; sin await guardas la Promise misma
loadProducts() {
  this.state.products = this.orm.searchRead("product.product", [], ["name"]);
}
```

```
// Correcto
async loadProducts() {
  this.state.products = await this.orm.searchRead("product.product", [], ["name"]);
}
```

Error 3: No Manejar Errores del Servidor

```
// Malo - un rechazo no manejado puede romper toda la acción
async saveRecord() {
  await this.orm.create("res.partner", [this.state.formData]);
}

// Correcto - envuelve en try/catch y da retroalimentación al usuario
async saveRecord() {
  try {
    await this.orm.create("res.partner", [this.state.formData]);
  } catch (error) {
    this.state.error = "No se pudo guardar el registro. Intenta de nuevo.";
  }
}
```

Error 4: Olvidar que `create` Devuelve un Array de IDs

```
// Malo - asumir que create resuelve a un solo id
const id = await this.orm.create("res.partner", [{ name: "Test" }]);

// Correcto - create siempre resuelve a un array, incluso para un solo registro
const [id] = await this.orm.create("res.partner", [{ name: "Test" }]);
```

Probando la Integración de Servicios

Como los componentes reciben los servicios a través de `useService`, las pruebas pueden sustituirlos por mocks. Los helpers exactos dependen de tu versión de Odoo (cubriremos el framework de pruebas real de Odoo en el Capítulo 16 Parte 1), pero la idea siempre se ve así:

```
// Pseudo-código: la forma de una prueba con servicios simulados

describe("ProductManager", () => {
  test("debería cargar productos al montar", async () => {
    const mockOrm = {
      searchRead: jest.fn().mockResolvedValue([
        { id: 1, name: "Producto de Prueba", list_price: 10.0 }
      ])
    };

    const component = await makeTestComponent(ProductManager, {
      services: {
        orm: mockOrm
      }
    });

    await nextTick(); // Esperar a que los efectos se completen
```

```
    expect(mockOrm.searchRead).toHaveBeenCalledWith(
      "product.product",
      [["sale_ok", "=", true]],
      expect.any(Array),
      expect.any(Object)
    );

    expect(component.state.products).toHaveLength(1);
  });
});
```

Resumen

El hook `useService` y el sistema de servicios de Odoo proporcionan:

- **Servicio `orm`**: Operaciones CRUD completas para modelos de Odoo
- **Servicio `rpc`**: Comunicación directa con controladores personalizados
- **API consistente**: Los mismos patrones a través de toda tu aplicación
- **Manejo de errores**: Gestión y recuperación de errores incorporada
- **Rendimiento**: Comunicación optimizada con el backend

Principios clave para el uso efectivo de servicios:

1. **Usar el servicio correcto** para cada tarea (ORM para modelos, RPC para controladores)
2. **Manejar errores graciosamente** con mensajes amigables para el usuario
3. **Operaciones en lote** cuando sea posible para mejor rendimiento
4. **Cache estratégico** para reducir solicitudes al servidor
5. **Probar con simulaciones** para asegurar confiabilidad

En el próximo capítulo, exploraremos más servicios incorporados de Odoo como notificaciones y acciones, que te ayudarán a crear experiencias de usuario verdaderamente integradas.

TL;DR: Usa `useService("orm")` para CRUD contra modelos de Odoo (`searchRead`, `create`, `write`, `unlink`) y `useService("rpc")` para controladores personalizados — siempre con `await`, siempre envolviendo las llamadas en `try/catch`, y recordando que `create` devuelve un array de ids, no un id suelto.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch12_ex1`. Las instrucciones de instalación están en el README del repositorio.

Ejercicios

1. **CRUD básico:** Construye un componente pequeño que liste registros de `res.partner` (`searchRead` con solo `name` y `email`), y agrega un botón que cree un nuevo contacto llamado “Contacto de Prueba” usando `orm.create`. Recuerda que `create` devuelve un array de ids.
2. **Maneja el error:** Envuelve una llamada a `orm.write` en un `try/catch` y muestra una alerta de Bootstrap con un mensaje amigable si falla. Pruébalo pasando un nombre de modelo inválido y observa qué pasa sin el try/catch primero.

3. **Cuenta antes de traer:** Usa `orm.searchCount` para mostrar "X contactos encontrados" antes de cargar la lista completa con `searchRead`, para que el usuario vea un número de inmediato mientras los datos se cargan.

¿Qué Sigue?

Con datos fluyendo entre tus componentes y el servidor, el Capítulo 13 pasa al resto de los servicios integrados de Odoo — notificaciones, diálogos, acciones y permisos — las piezas que hacen que un componente se sienta parte nativa de Odoo en vez de un widget pegado con cinta adhesiva.

Capítulo 13 Parte 1: Servicios Integrados de Odoo

> **¿Por qué este capítulo?** Porque "funciona, pero no se siente como Odoo" es la queja más común sobre un addon a medida, y casi nunca viene de la lógica: viene de haberse armado un toast propio, un modal propio y una redirección propia en lugar de usar los que Odoo ya trae.
>
> **¿Qué trata de resolver Odoo con esto?** Odoo tiene miles de pantallas que necesitan notificar, navegar y confirmar de la misma manera. Exponer `notification`, `action` y `dialog` como servicios inyectables es como la plataforma sostiene esa coherencia sin que cada autor de addon la reimplemente — y la reimplemente distinto.
>
> **Aplicación en la vida real:** Todo addon de cliente que he entregado termina llamando a `notification.add()` después de un guardado, a `action.doAction()` para dejar al usuario en el registro correcto, y a `dialog.add()` para confirmar algo destructivo. Entre esos tres se va la mayor parte de la fontanería de UI del día a día.

En el Capítulo 12, dominamos la comunicación con el servidor usando los servicios `orm` y `rpc`. Pero el ecosistema de servicios de Odoo se extiende mucho más allá de las operaciones de datos. La plataforma proporciona una rica colección de servicios integrados que manejan todo, desde notificaciones de usuario hasta navegación, gestión de diálogos e integración del sistema.

Estos servicios son el secreto para crear componentes que se sienten como partes nativas de la experiencia de Odoo. En lugar de construir soluciones personalizadas para patrones comunes de UI, puedes aprovechar los servicios probados en batalla de Odoo para proporcionar interacciones de usuario consistentes y pulidas.

Este capítulo está dividido en dos partes. La Parte 1 (esta) cubre los servicios que usarás en casi todos los proyectos: `notification`, `action`, `dialog`, y algunos esenciales adicionales. La Parte 2 cubre patrones avanzados de composición, optimización de rendimiento y recuperación de errores para escenarios más complejos — puedes avanzar a ella una vez que domines lo básico aquí.

El Servicio `notification`: Retroalimentación del Usuario Hecha Correctamente

La retroalimentación del usuario es crucial para una buena UX. Los usuarios necesitan saber cuándo las operaciones tienen éxito, cuándo ocurren errores, y cuándo suceden eventos importantes. El servicio `notification` proporciona las notificaciones toast características de Odoo que aparecen en la esquina superior derecha de la pantalla.

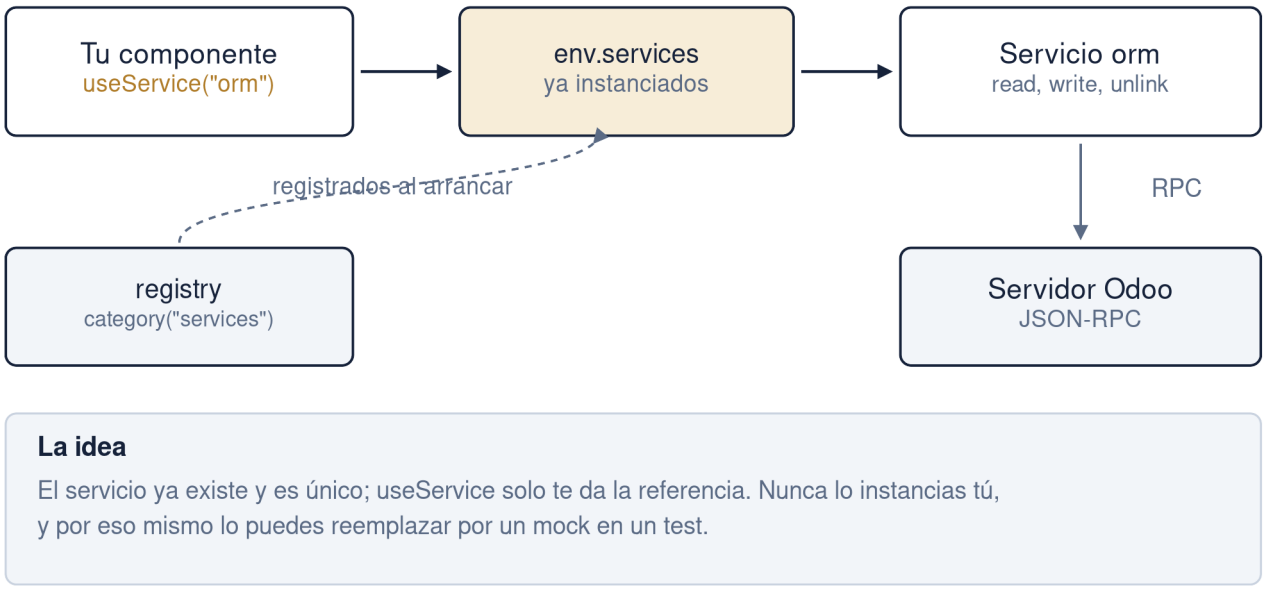


Figura 5: De useService al servidor: el servicio ya existe, tú solo recibes una referencia.

Uso Básico de Notificaciones

Construyamos un ejemplo completo que demuestre todos los tipos de notificación y características avanzadas. Construiremos la clase `NotificationDemo` por grupos de métodos — cada bloque de código a continuación continúa directamente dentro del mismo cuerpo de la clase.

JavaScript (`notification_demo.js`):

Primero, los imports, la declaración de la clase, y `setup()`, donde obtenemos los dos servicios que necesitaremos:

```
import { Component, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class NotificationDemo extends Component {
  static template = "mi_modulo.NotificationDemo";

  setup() {
    this.state = useState({
      textoMensaje: "¡Operación completada exitosamente!",
      conteoNotificaciones: 0
    });

    this.notification = useService("notification");
    this.orm = useService("orm");
  }
```

Ahora, los cuatro tipos básicos de notificación. Nota el patrón: llamar a `this.notification.add(mensaje, opciones)` pasando un `type` que corresponda a la situación (`success`, `info`, `warning`, o `danger`):

```
  // Tipos básicos de notificación
  mostrarNotificacionExito() {
    this.notification.add(this.state.textoMensaje, {
      title: "Éxito",
```

```
      type: "success",
      sticky: false // Auto-descartar después de unos segundos
    });

    this.incrementarConteoNotificaciones();
    console.log("NotificationDemo: Notificación de éxito mostrada");
  }

  mostrarNotificacionInfo() {
    this.notification.add("Aquí tienes información útil para ti.", {
      title: "Información",
      type: "info"
    });

    this.incrementarConteoNotificaciones();
  }

  mostrarNotificacionAdvertencia() {
    this.notification.add("Por favor revisa esto cuidadosamente antes de proceder.", {
      title: "Advertencia",
      type: "warning",
      sticky: true // Permanece hasta ser descartada manualmente
    });

    this.incrementarConteoNotificaciones();
  }

  mostrarNotificacionError() {
    this.notification.add("Algo salió mal. Por favor inténtalo de nuevo.", {
      title: "Error",
      type: "danger",
      sticky: true
    });

    this.incrementarConteoNotificaciones();
  }
```

Ahora dos métodos que muestran notificaciones haciendo trabajo real: uno agrega una clase CSS personalizada, el otro envuelve una operación asíncrona que puede fallar, mostrando retroalimentación antes, en éxito, y en error:

```
  // Notificación avanzada con contenido personalizado
  mostrarNotificacionRica() {
    this.notification.add("Tu documento ha sido procesado.", {
      title: "Procesamiento de Documento Completado",
      type: "success",
      sticky: false,
      className: "estilo-notificacion-personalizada" // Clase CSS personalizada
    });

    this.incrementarConteoNotificaciones();
  }

  // Ejemplo del mundo real: Operación de guardar con retroalimentación
  async guardarDatosConRetroalimentacion() {
    try {
```

```
      // Mostrar retroalimentación inmediata
      this.notification.add("Guardando tus cambios...", {
        title: "Guardando",
        type: "info"
      });

      // Simular operación de guardado
      await this.simularOperacionAsincrona();

      // Mostrar retroalimentación de éxito
      this.notification.add("Tus cambios han sido guardados exitosamente.", {
        title: "Guardado",
        type: "success"
      });

      console.log("NotificationDemo: Datos guardados exitosamente");

    } catch (error) {
      // Mostrar retroalimentación de error
      this.notification.add(`Error al guardar: ${error.message}`, {
        title: "Guardado Fallido",
        type: "danger",
        sticky: true // Mantener error visible hasta que el usuario lo descarte
      });

      console.error("NotificationDemo: Guardado falló:", error);
    }

    this.incrementarConteoNotificaciones();
  }
```

Los últimos dos métodos manejan una operación masiva de varios pasos con actualizaciones de progreso, y un flujo de validación de formulario que da retroalimentación en cualquier caso:

```
  // Operación masiva con retroalimentación de progreso
  async realizarOperacionMasiva() {
    const elementos = ['Elemento A', 'Elemento B', 'Elemento C', 'Elemento D'];
    let procesados = 0;

    try {
      // Notificación inicial
      this.notification.add(`Iniciando operación masiva en ${elementos.length}
          ↪ elementos...`, {
        title: "Operación Masiva",
        type: "info"
      });

      for (const elemento of elementos) {
        await this.simularOperacionAsincrona(500); // Simular tiempo de procesamiento
        procesados++;

        // Notificación de progreso
        this.notification.add(`Procesados ${procesados}/${elementos.length} elementos`, {
          title: "Actualización de Progreso",
          type: "info"
        });
```

```
    }

    // Notificación de finalización
    this.notification.add(`¡Todos los ${elementos.length} elementos procesados
        ↪ exitosamente!`, {
      title: "Operación Masiva Completada",
      type: "success",
      sticky: false
    });

  } catch (error) {
    this.notification.add(`Operación masiva falló después de procesar ${procesados}
        ↪ elementos.`, {
      title: "Operación Fallida",
      type: "danger",
      sticky: true
    });
  }
}

// Ejemplo de retroalimentación de validación
validarYNotificar() {
  if (!this.state.textoMensaje.trim()) {
    this.notification.add("El texto del mensaje no puede estar vacío.", {
      title: "Error de Validación",
      type: "warning"
    });
    return false;
  }

  if (this.state.textoMensaje.length > 100) {
    this.notification.add("El texto del mensaje es demasiado largo (máximo 100
        ↪ caracteres).", {
      title: "Error de Validación",
      type: "warning"
    });
    return false;
  }

  this.notification.add("¡Validación del mensaje exitosa!", {
    title: "Entrada Válida",
    type: "success"
  });

  return true;
}
```

Finalmente, los pequeños métodos auxiliares que mantienen el demo ordenado, y la llave de cierre de la clase:

```
// Métodos auxiliares
incrementarConteoNotificaciones() {
  this.state.conteoNotificaciones++;
}

async simularOperacionAsincrona(retraso = 1000) {
```

```
    return new Promise((resolve, reject) => {
      setTimeout(() => {
        // Simular fallas ocasionales
        if (Math.random() < 0.1) {
          reject(new Error("Error de red simulado"));
        } else {
          resolve();
        }
      }, retraso);
    });
  }

  alCambiarMensaje(event) {
    this.state.textoMensaje = event.target.value;
  }

  reiniciarDemo() {
    this.state.textoMensaje = "¡Operación completada exitosamente!";
    this.state.conteoNotificaciones = 0;
  }
}
```

Plantilla (`notification_demo.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="mi_modulo.NotificationDemo" owl="1">
    <div class="notification-demo card">
      <div class="card-header">
        <h4>Demo del Servicio de Notificación</h4>
        <small class="text-muted">
          Notificaciones mostradas: <span class="badge bg-secondary"
    ↪  t-esc="state.conteoNotificaciones"/>
        </small>
      </div>

      <div class="card-body">
        <!-- Entrada de Mensaje Personalizado -->
        <div class="mb-4">
          <label class="form-label">Mensaje Personalizado</label>
          <div class="input-group">
            <input
                type="text"
                class="form-control"
                t-model="state.textoMensaje"
                placeholder="Ingresa mensaje de notificación"
                maxlength="100"
            />
            <button
                class="btn btn-outline-secondary"
                t-on-click="validarYNotificar">
              Validar
            </button>
          </div>
          <div class="form-text">
            Longitud del mensaje: <t t-esc="state.textoMensaje.length"/>/100 caracteres
```

```
    </div>
  </div>

  <!-- Tipos Básicos de Notificación -->
  <div class="mb-4">
    <h6>Tipos Básicos de Notificación</h6>
    <div class="btn-group me-2 mb-2" role="group">
      <button
          class="btn btn-success btn-sm"
          t-on-click="mostrarNotificacionExito">
        <i class="fa fa-check"></i> Éxito
      </button>
      <button
          class="btn btn-info btn-sm"
          t-on-click="mostrarNotificacionInfo">
        <i class="fa fa-info-circle"></i> Info
      </button>
      <button
          class="btn btn-warning btn-sm"
          t-on-click="mostrarNotificacionAdvertencia">
        <i class="fa fa-exclamation-triangle"></i> Advertencia
      </button>
      <button
          class="btn btn-danger btn-sm"
          t-on-click="mostrarNotificacionError">
        <i class="fa fa-times"></i> Error
      </button>
    </div>
  </div>

  <!-- Ejemplos Avanzados -->
  <div class="mb-4">
    <h6>Ejemplos del Mundo Real</h6>
    <div class="d-flex gap-2 flex-wrap">
      <button
          class="btn btn-primary btn-sm"
          t-on-click="mostrarNotificacionRica">
        <i class="fa fa-star"></i> Notificación Rica
      </button>

      <button
          class="btn btn-outline-primary btn-sm"
          t-on-click="guardarDatosConRetroalimentacion">
        <i class="fa fa-save"></i> Guardar con Retroalimentación
      </button>

      <button
          class="btn btn-outline-secondary btn-sm"
          t-on-click="realizarOperacionMasiva">
        <i class="fa fa-tasks"></i> Operación Masiva
      </button>
    </div>
  </div>

  <!-- Controles del Demo -->
```

```
        <div class="border-top pt-3">
          <button
              class="btn btn-outline-secondary btn-sm"
              t-on-click="reiniciarDemo">
            <i class="fa fa-refresh"></i> Reiniciar Demo
          </button>
        </div>
      </div>
    </div>
  </t>
</templates>
```

Mejores Prácticas de Notificación

Qué hacer: - Usar tipos apropiados de notificación (`success` para acciones completadas, `warning` para info importante, `danger` para errores) - Hacer que las notificaciones de error sean adhesivas para que los usuarios puedan leerlas completamente - Proporcionar mensajes claros y accionables - Usar notificaciones para retroalimentación de acciones del usuario

Qué no hacer: - No spam a los usuarios con demasiadas notificaciones - No usar notificaciones para errores críticos del sistema (usar diálogos en su lugar) - Evitar mensajes genéricos como "Ocurrió un error" - No usar notificaciones para información que necesita respuesta del usuario

---

El Servicio `action`: Navegación e Integración

El servicio `action` es tu puerta de entrada al sistema de acciones de Odoo. Puede disparar cualquier acción en el sistema—abrir formularios, mostrar listas, ejecutar reportes, ejecutar acciones de servidor, y mucho más.

Entendiendo los Tipos de Acción

Odoo soporta varios tipos de acción: - **Acciones de Ventana**: Abrir vistas (formulario, lista, kanban, etc.) - **Acciones de Servidor**: Ejecutar código Python en el servidor - **Acciones de Reporte**: Generar y mostrar reportes - **Acciones de URL**: Navegar a URLs externas - **Acciones de Cliente**: Abrir componentes personalizados del lado del cliente

Construyamos un ejemplo completo, de nuevo grupo de métodos por grupo.

JavaScript (`action_navigator.js`):

Primero, el setup y el cargador de datos que llena los dos selectores que usaremos para elegir un registro:

```
import { Component, useState } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ActionNavigator extends Component {
  static template = "mi_modulo.ActionNavigator";

  setup() {
    this.state = useState({
      idPartnerSeleccionado: null,
      idProductoSeleccionado: null,
      partners: [],
```

```
      productos: []
    });

    this.action = useService("action");
    this.orm = useService("orm");
    this.notification = useService("notification");

    // Cargar datos de muestra
    this.cargarDatosMuestra();
  }

  async cargarDatosMuestra() {
    try {
      const [partners, productos] = 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.productos = productos;
    } catch (error) {
      console.error("ActionNavigator: Error al cargar datos de muestra:", error);
    }
  }
```

Con los datos de muestra cargados, agreguemos los métodos básicos de acciones de ventana — abrir un formulario, una lista, un kanban, más las variantes para abrir en una nueva ventana, en un modal, o esperar el resultado con un callback:

```
  // Acciones básicas de ventana
  abrirFormularioCliente(idPartner = null) {
    console.log(`ActionNavigator: Abriendo formulario de cliente para ID: ${idPartner ||
        ↪ 'nuevo'}`);

    this.action.doAction({
      type: "ir.actions.act_window",
      name: idPartner ? "Editar Cliente" : "Nuevo Cliente",
      res_model: "res.partner",
      res_id: idPartner || false,
      views: [[false, "form"]],
      target: "current", // Abre en vista actual
      context: {
        default_is_company: true,
        default_customer_rank: 1
      }
    });
  }

  abrirListaClientes() {
    console.log("ActionNavigator: Abriendo vista de lista de clientes");

    this.action.doAction({
      type: "ir.actions.act_window",
      name: "Clientes",
      res_model: "res.partner",
      views: [
```

```
        [false, "list"],
        [false, "form"]
      ],
      domain: [["is_company", "=", true]],
      target: "current"
    });
  }

  abrirKanbanClientes() {
    this.action.doAction({
      type: "ir.actions.act_window",
      name: "Pipeline de Clientes",
      res_model: "res.partner",
      views: [
        [false, "kanban"],
        [false, "form"]
      ],
      domain: [["is_company", "=", true]],
      target: "current"
    });
  }

  // Abrir en nueva pestaña/ventana
  abrirEnNuevaVentana(idPartner) {
    console.log(`ActionNavigator: Abriendo cliente ${idPartner} en nueva ventana`);

    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "res.partner",
      res_id: idPartner,
      views: [[false, "form"]],
      target: "new" // Abre en nueva ventana/pestaña
    });
  }

  // Diálogos modales
  abrirEnModal(idPartner) {
    console.log(`ActionNavigator: Abriendo cliente ${idPartner} en modal`);

    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "res.partner",
      res_id: idPartner,
      views: [[false, "form"]],
      target: "new",
      flags: {
        mode: "readonly" // Opcional: abrir en modo solo lectura
      }
    });
  }

  // Acción con callback
  async abrirConCallback(idPartner) {
    console.log(`ActionNavigator: Abriendo cliente ${idPartner} con callback`);
```

```
    try {
      const resultado = await this.action.doAction({
        type: "ir.actions.act_window",
        res_model: "res.partner",
        res_id: idPartner,
        views: [[false, "form"]],
        target: "new"
      });

      console.log("ActionNavigator: Acción completada con resultado:", resultado);

      this.notification.add("¡Formulario de cliente abierto exitosamente!", {
        type: "success"
      });

    } catch (error) {
      console.error("ActionNavigator: Acción falló:", error);

      this.notification.add("Error al abrir formulario de cliente.", {
        type: "danger"
      });
    }
  }
```

Más allá de abrir formularios, el servicio `action` también puede abrir registros relacionados, ejecutar acciones de servidor y reportes, navegar a URLs externas, y abrir tus propias acciones personalizadas de cliente:

```
  // Abrir registros relacionados
  abrirVentasCliente(idPartner) {
    console.log(`ActionNavigator: Abriendo ventas para cliente ${idPartner}`);

    this.action.doAction({
      type: "ir.actions.act_window",
      name: "Ventas del Cliente",
      res_model: "sale.order",
      views: [
        [false, "list"],
        [false, "form"]
      ],
      domain: [["partner_id", "=", idPartner]],
      context: {
        default_partner_id: idPartner,
        search_default_partner_id: idPartner
      },
      target: "current"
    });
  }

  // Acciones de servidor
  async ejecutarAccionServidor(xmlIdAccion) {
    console.log(`ActionNavigator: Ejecutando acción de servidor: ${xmlIdAccion}`);

    try {
      const resultado = await this.action.doAction(xmlIdAccion, {
        // Contexto adicional para la acción de servidor
```

```
        active_ids: [this.state.idPartnerSeleccionado],
        active_id: this.state.idPartnerSeleccionado,
        active_model: "res.partner"
      });

      this.notification.add("¡Acción de servidor ejecutada exitosamente!", {
        type: "success"
      });

      console.log("ActionNavigator: Resultado de acción de servidor:", resultado);

    } catch (error) {
      console.error("ActionNavigator: Acción de servidor falló:", error);

      this.notification.add(`Acción de servidor falló: ${error.message}`, {
        type: "danger"
      });
    }
  }

  // Acciones de reporte
  abrirReporteCliente(idPartner) {
    console.log(`ActionNavigator: Abriendo reporte para cliente ${idPartner}`);

    this.action.doAction({
      type: "ir.actions.report",
      report_name: "base.report_partner_ledger", // Reporte de ejemplo
      context: {
        active_ids: [idPartner],
        active_model: "res.partner"
      }
    });
  }

  // Acciones de URL
  abrirUrlExterna(url) {
    console.log(`ActionNavigator: Abriendo URL externa: ${url}`);

    this.action.doAction({
      type: "ir.actions.act_url",
      url: url,
      target: "new"
    });
  }

  // Acciones personalizadas de cliente
  abrirAccionClientePersonalizada() {
    console.log("ActionNavigator: Abriendo acción personalizada de cliente");

    this.action.doAction({
      type: "ir.actions.client",
      tag: "mi_accion_personalizada", // Tu tag de acción personalizada
      name: "Dashboard Personalizado",
      params: {
        // Parámetros personalizados para tu acción
```

```
        customer_id: this.state.idPartnerSeleccionado
      }
    });
  }

  // Métodos de navegación
  irAMenuAplicaciones() {
    this.action.doAction("base.open_module_tree");
  }

  irAConfiguracion() {
    this.action.doAction("base.action_res_config_settings");
  }
```

Finalmente, algunos pequeños auxiliares que la plantilla necesita para leer y reaccionar a la selección actual:

```
  // Métodos auxiliares para la plantilla
  alSeleccionarPartner(event) {
    this.state.idPartnerSeleccionado = parseInt(event.target.value) || null;
  }

  alSeleccionarProducto(event) {
    this.state.idProductoSeleccionado = parseInt(event.target.value) || null;
  }

  get partnerSeleccionado() {
    return this.state.partners.find(p => p.id === this.state.idPartnerSeleccionado);
  }

  get tienePartnerSeleccionado() {
    return this.state.idPartnerSeleccionado !== null;
  }
}
```

Plantilla (`action_navigator.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<templates xml:space="preserve">
  <t t-name="mi_modulo.ActionNavigator" owl="1">
    <div class="action-navigator">
      <div class="row">
        <!-- Panel de Selección -->
        <div class="col-md-4">
          <div class="card">
            <div class="card-header">
              <h5>Seleccionar Registros</h5>
            </div>
            <div class="card-body">
              <!-- Selección de Partner -->
              <div class="mb-3">
                <label class="form-label">Cliente</label>
                <select
                    class="form-select"
                    t-on-change="alSeleccionarPartner">
                  <option value="">Selecciona un cliente...</option>
                  <t t-foreach="state.partners" t-as="partner" t-key="partner.id">
```

```
                <option
                    t-att-value="partner.id"
                    t-att-selected="state.idPartnerSeleccionado === partner.id"
                    t-esc="partner.name"/>
                </t>
              </select>
            </div>

            <!-- Selección de Producto -->
            <div class="mb-3">
              <label class="form-label">Producto</label>
              <select
                  class="form-select"
                  t-on-change="alSeleccionarProducto">
                <option value="">Selecciona un producto...</option>
                <t t-foreach="state.productos" t-as="producto" t-key="producto.id">
                  <option
                    t-att-value="producto.id"
                    t-att-selected="state.idProductoSeleccionado === producto.id"
                    t-esc="producto.name"/>
                </t>
              </select>
            </div>

            <!-- Información de Selección -->
            <t t-if="tienePartnerSeleccionado">
              <div class="alert alert-info">
                <small>
                  Seleccionado: <strong t-esc="partnerSeleccionado.name"/>
                </small>
              </div>
            </t>
          </div>
        </div>
      </div>

      <!-- Panel de Acciones -->
      <div class="col-md-8">
        <div class="card">
          <div class="card-header">
            <h5>Ejemplos de Acciones</h5>
          </div>
          <div class="card-body">
            <!-- Acciones de Ventana -->
            <div class="mb-4">
              <h6>Acciones de Ventana</h6>
              <div class="btn-group mb-2 me-2" role="group">
                <button
                    class="btn btn-primary btn-sm"
                    t-on-click="() => this.abrirFormularioCliente()">
                  <i class="fa fa-plus"></i> Nuevo Cliente
                </button>
                <button
                    class="btn btn-outline-primary btn-sm"
                    t-on-click="() =>
    ↪ this.abrirFormularioCliente(state.idPartnerSeleccionado)"
```

```
                t-att-disabled="!tienePartnerSeleccionado">
              <i class="fa fa-edit"></i> Editar Seleccionado
            </button>
          </div>

          <div class="btn-group mb-2" role="group">
            <button
                class="btn btn-info btn-sm"
                t-on-click="abrirListaClientes">
              <i class="fa fa-list"></i> Lista de Clientes
            </button>
            <button
                class="btn btn-info btn-sm"
                t-on-click="abrirKanbanClientes">
              <i class="fa fa-columns"></i> Kanban de Clientes
            </button>
          </div>
        </div>

        <!-- Opciones de Destino -->
        <div class="mb-4">
          <h6>Destinos de Apertura</h6>
          <div class="btn-group mb-2" role="group">
            <button
                class="btn btn-outline-secondary btn-sm"
                t-on-click="() =>
↪ this.abrirEnNuevaVentana(state.idPartnerSeleccionado)"
                t-att-disabled="!tienePartnerSeleccionado">
              <i class="fa fa-external-link"></i> Nueva Ventana
            </button>
            <button
                class="btn btn-outline-secondary btn-sm"
                t-on-click="() => this.abrirEnModal(state.idPartnerSeleccionado)"
                t-att-disabled="!tienePartnerSeleccionado">
              <i class="fa fa-window-restore"></i> Modal
            </button>
            <button
                class="btn btn-outline-secondary btn-sm"
                t-on-click="() =>
↪ this.abrirConCallback(state.idPartnerSeleccionado)"
                t-att-disabled="!tienePartnerSeleccionado">
              <i class="fa fa-reply"></i> Con Callback
            </button>
          </div>
        </div>

        <!-- Registros Relacionados -->
        <div class="mb-4">
          <h6>Registros Relacionados</h6>
          <button
              class="btn btn-success btn-sm"
              t-on-click="() =>
↪ this.abrirVentasCliente(state.idPartnerSeleccionado)"
              t-att-disabled="!tienePartnerSeleccionado">
            <i class="fa fa-shopping-cart"></i> Ventas del Cliente
```

```
                </button>
              </div>

              <!-- Reportes y Externos -->
              <div class="mb-4">
                <h6>Reportes y Enlaces Externos</h6>
                <div class="btn-group mb-2 me-2" role="group">
                  <button
                      class="btn btn-warning btn-sm"
                      t-on-click="() =>
    ↪ this.abrirReporteCliente(state.idPartnerSeleccionado)"
                      t-att-disabled="!tienePartnerSeleccionado">
                    <i class="fa fa-file-pdf-o"></i> Reporte de Cliente
                  </button>
                </div>

                <div class="btn-group mb-2" role="group">
                  <button
                      class="btn btn-outline-info btn-sm"
                      t-on-click="() => this.abrirUrlExterna('https://www.odoo.com')">
                    <i class="fa fa-globe"></i> Abrir Odoo.com
                  </button>
                </div>
              </div>

              <!-- Navegación del Sistema -->
              <div class="mb-4">
                <h6>Navegación del Sistema</h6>
                <div class="btn-group" role="group">
                  <button
                      class="btn btn-secondary btn-sm"
                      t-on-click="irAMenuAplicaciones">
                    <i class="fa fa-th"></i> Menú de Apps
                  </button>
                  <button
                      class="btn btn-secondary btn-sm"
                      t-on-click="irAConfiguracion">
                    <i class="fa fa-cog"></i> Configuración
                  </button>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </t>
</templates>
```

El Servicio `dialog`: Interacciones del Usuario

El servicio `dialog` proporciona diálogos modales sofisticados para interacciones del usuario:

Ejemplo JavaScript:

```
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 = "mi_modulo.DialogDemo";

  setup() {
    this.dialog = useService("dialog");
    this.notification = useService("notification");
  }

  // Diálogo de confirmación
  mostrarDialogoConfirmacion() {
    this.dialog.add(ConfirmationDialog, {
      title: "Confirmar Eliminación",
      body: "¿Estás seguro de que quieres eliminar este registro? Esta acción no se puede
          ↪ deshacer.",
      confirm: () => {
        this.notification.add("¡Registro eliminado exitosamente!", { type: "success" });
        console.log("Usuario confirmó eliminación");
      },
      cancel: () => {
        this.notification.add("Eliminación cancelada.", { type: "info" });
        console.log("Usuario canceló eliminación");
      }
    });
  }

  // Diálogo personalizado con formulario
  mostrarDialogoPersonalizado() {
    this.dialog.add(DialogFormularioPersonalizado, {
      title: "Crear Nuevo Elemento",
      onSave: (datos) => {
        console.log("Diálogo guardado con datos:", datos);
        this.notification.add("¡Elemento creado exitosamente!", { type: "success" });
      }
    });
  }
}
```

Servicios Esenciales Adicionales

El Servicio `user`

Acceder a información del usuario actual:

```
setup() {
  this.user = useService("user");

  console.log("Usuario actual:", this.user.name);
  console.log("Es admin:", this.user.isAdmin);
```

```
  console.log("Contexto del usuario:", this.user.context);

  // Las verificaciones de pertenencia a grupos son asíncronas
  onWillStart(async () => {
    this.esGerenteVentas = await this.user.hasGroup("sales_team.group_sale_manager");
  });
}
```

El Servicio `router`

Manejar enrutamiento de URL:

```
setup() {
  this.router = useService("router");

  // Navegar a una ruta específica
  this.navegarAClientes = () => {
    this.router.pushState({ action: "clientes" });
  };
}
```

El Servicio `company`

Acceder a información de la compañía actual:

```
setup() {
  this.company = useService("company");

  console.log("Compañía actual:", this.company.currentCompany.name);
  console.log("Compañías permitidas:", this.company.allowedCcompanies);
}
```

Patrones de Integración de Servicios

Patrón 1: Uso Coordinado de Servicios

```
async crearClienteConRetroalimentacion(datosCliente) {
  try {
    // 1. Mostrar notificación de progreso
    this.notification.add("Creando cliente...", { type: "info" });

    // 2. Crear el cliente (create devuelve un array de ids nuevos)
    const [idCliente] = await this.orm.create("res.partner", [datosCliente]);

    // 3. Mostrar notificación de éxito
    this.notification.add("¡Cliente creado exitosamente!", { type: "success" });

    // 4. Navegar al nuevo cliente
    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "res.partner",
      res_id: idCliente,
      views: [[false, "form"]],
      target: "current"
```

```
    });

  } catch (error) {
    this.notification.add(`Error al crear cliente: ${error.message}`, {
      type: "danger",
      sticky: true
    });
  }
}
```

Patrón 2: Acciones Condicionales Basadas en Permisos del Usuario

```
async eliminarRegistro(idRegistro) {
  // Verificar si el usuario tiene permisos de eliminación (hasGroup es asíncrono)
  const puedeEliminar = this.user.isAdmin || (await
      ↪  this.user.hasGroup("base.group_system"));
  if (!puedeEliminar) {
    this.notification.add("No tienes permiso para eliminar registros.", {
      type: "warning"
    });
    return;
  }

  // Mostrar diálogo de confirmación
  this.dialog.add(ConfirmationDialog, {
    title: "Confirmar Eliminación",
    body: "Esto eliminará permanentemente el registro.",
    confirm: async () => {
      try {
        await this.orm.unlink("nombre.modelo", [idRegistro]);
        this.notification.add("¡Registro eliminado exitosamente!", { type: "success" });
      } catch (error) {
        this.notification.add("Error al eliminar registro.", { type: "danger" });
      }
    }
  });
}
```

Probando Integración de Servicios

Los servicios pueden ser simulados para pruebas:

```
describe("IntegracionServicios", () => {
  test("debería mostrar notificación en éxito", async () => {
    const mockNotification = {
      add: jest.fn()
    };

    const mockAction = {
      doAction: jest.fn().mockResolvedValue(true)
    };

    const componente = await makeTestComponent(MiComponente, {
```

```
      services: {
        notification: mockNotification,
        action: mockAction,
        orm: mockOrm
      }
    });

    await componente.realizarAccion();

    expect(mockNotification.add).toHaveBeenCalledWith(
      "¡Operación completada exitosamente!",
      { type: "success" }
    );

    expect(mockAction.doAction).toHaveBeenCalledWith({
      type: "ir.actions.act_window",
      res_model: "modelo.prueba",
      views: [[false, "form"]],
      target: "current"
    });
  });

  test("debería manejar errores de servicio graciosamente", async () => {
    const mockNotification = {
      add: jest.fn()
    };

    const mockOrm = {
      create: jest.fn().mockRejectedValue(new Error("Error de base de datos"))
    };

    const componente = await makeTestComponent(MiComponente, {
      services: {
        notification: mockNotification,
        orm: mockOrm
      }
    });

    await componente.crearRegistro();

    expect(mockNotification.add).toHaveBeenCalledWith(
      "Error al crear registro: Error de base de datos",
      { type: "danger", sticky: true }
    );
  });
});
```

Mejores Prácticas de Servicios

Qué hacer

1. **Usar servicios apropiados** para cada tarea
2. **Manejar errores graciosamente** con mensajes amigables para el usuario

3. **Proporcionar retroalimentación** para todas las acciones del usuario
4. **Cachear operaciones costosas** cuando sea posible
5. **Probar interacciones de servicios** con mocks apropiados
6. **Usar diálogos de confirmación** para acciones destructivas
7. **Coordinar servicios** para flujos de trabajo complejos

Qué no hacer

1. **No ignorar errores de servicios** - siempre manejar excepciones
2. **No spam con notificaciones** - ser selectivo sobre lo que requiere atención del usuario
3. **No bloquear la UI** - usar estados de carga para operaciones largas
4. **No hardcodear IDs de acción** - usar IDs XML cuando sea posible
5. **No olvidar limpiar** - limpiar timers y suscripciones en callbacks de servicios

---

Errores Comunes

Error Común 1: Tratar `hasGroup()` como Síncrono

```
// Incorrecto - hasGroup() devuelve una Promise; el `if` siempre entra en esta
// rama, porque un objeto Promise es "truthy" sin importar lo que resuelva
setup() {
  this.user = useService("user");
  if (this.user.hasGroup("base.group_system")) {
    // ¡Se ejecuta incondicionalmente, antes de que la verificación se resuelva!
  }
}

// Correcto - esperarlo con await, típicamente dentro de onWillStart u otro método async
onWillStart(async () => {
  this.esAdminSistema = await this.user.hasGroup("base.group_system");
});
```

Error Común 2: Solicitar un Servicio Fuera de `setup()`

```
// Incorrecto - useService solo funciona mientras se está configurando el
// componente; llamarlo después (ej. desde un event handler) lanza un error
alHacerClicBoton() {
  const notification = useService("notification"); // ¡Error!
  notification.add("¡Listo!");
}

// Correcto - solicitar el servicio una vez en setup(), luego reutilizar la referencia
setup() {
  this.notification = useService("notification");
}

alHacerClicBoton() {
  this.notification.add("¡Listo!");
}
```

Error Común 3: Ignorar Silenciosamente Errores del ORM

```
// Incorrecto - si orm.create() rechaza, la promesa queda sin manejar y el
// usuario no ve que pasó absolutamente nada
async guardarRegistro(datos) {
  await this.orm.create("res.partner", [datos]);
}

// Correcto - capturar el error y decirle al usuario qué pasó
async guardarRegistro(datos) {
  try {
    await this.orm.create("res.partner", [datos]);
    this.notification.add("¡Guardado!", { type: "success" });
  } catch (error) {
    this.notification.add(`Error al guardar: ${error.message}`, {
      type: "danger",
      sticky: true
    });
  }
}
```

Error Común 4: Usar una Notificación Cuando Necesitas un Diálogo

```
// Incorrecto - un toast es fácil de pasar por alto, desaparece solo, y no
// puede bloquear la ejecución de la siguiente línea de código
this.notification.add("Esto eliminará permanentemente el registro. ¿Continuar?", {
  type: "warning"
});
this.orm.unlink("res.partner", [id]); // ¡Se ejecuta de inmediato, pase lo que pase!

// Correcto - una acción destructiva necesita un diálogo de confirmación, que
// pausa el flujo hasta que el usuario realmente decide
this.dialog.add(ConfirmationDialog, {
  title: "Confirmar Eliminación",
  body: "Esto eliminará permanentemente el registro. ¿Continuar?",
  confirm: () => this.orm.unlink("res.partner", [id]),
});
```

Ejercicios

1. **Variedad de notificaciones**: Agrega un botón que muestre una notificación `warning` con `sticky: true`, y otro que muestre una notificación `info` que se auto-descarte después de unos segundos. Confirma que puedes ver la diferencia de comportamiento.
2. **Eliminación protegida**: Escribe un método `eliminarRegistro(id)` que primero verifique `await this.user.hasGroup("base.group_system")`. Si el usuario no pertenece a ese grupo, muestra una notificación `warning` en lugar de eliminar.
3. **Confirmar, luego actuar**: Usa el servicio `dialog` para mostrar un `ConfirmationDialog` antes de llamar a `this.orm.unlink(...)`. Al cancelar, muestra una notificación `info` diciendo que la eliminación fue cancelada.

TL;DR: Recurre a `notification` para dar retroalimentación, `dialog` para confirmaciones y diálogos personalizados, `action` para navegar, y recuerda que `user.hasGroup()` es asíncrono — siempre inyecta los servicios con `useService`, nunca construyas los tuyos propios.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch13_1_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

En la Parte 2, construiremos sobre estos fundamentos con patrones de composición de servicios para flujos de trabajo de varios pasos, técnicas de rendimiento como debounce y caché, y estrategias para recuperarse graciosamente de fallas — los patrones a los que recurrirás una vez que tus componentes hagan más que una sola llamada a la API.

Capítulo 13 Parte 2: Patrones Avanzados de Servicios y Optimización

> **¿Por qué este capítulo?** Una sola llamada a `notification.add()` es fácil; coordinar tres servicios en un mismo flujo, evitar que una caja de búsqueda golpee al servidor en cada tecla, y recuperarse de una llamada de red inestable son las cosas que realmente consumen un día de trabajo una vez que un addon crece más allá de su primera versión.
>
> **¿Qué trata de resolver Odoo con esto?** La interfaz de Odoo tiene que seguir respondiendo y siendo resiliente bajo uso real — conexiones inestables, endpoints lentos, usuarios que hacen doble clic — así que el framework se apoya en patrones como debounce, caché y reintentos estructurados en vez de dejar que cada desarrollador los reinvente.
>
> **Aplicación en la vida real:** Búsqueda con debounce, reintento con backoff en una llamada inestable a una API externa, y flujos de trabajo de varios pasos que abarcan varios servicios son patrones a los que recurro constantemente una vez que un addon de cliente pasa de su primera prueba de concepto.

En la Parte 1 cubrimos los servicios que usarás todos los días — `notification`, `action`, `dialog`, y algunos esenciales — junto con los errores comunes que cometen los desarrolladores junior con ellos. Ahora abordaremos escenarios más exigentes: componer varios servicios en un solo flujo de trabajo, mantener rápidas las llamadas costosas a servicios, y recuperarnos con gracia cuando una llamada falla.

Patrones Avanzados de Servicios

Patrón 3: Composición de Servicios para Flujos de Trabajo Complejos

```
async procesarOrdenCliente(datosOrden) {
  const pasos = [
    { nombre: "Validando datos de orden", accion: () =>
    ↪  this.validarDatosOrden(datosOrden) },
    { nombre: "Creando cliente", accion: () => this.crearCliente(datosOrden.cliente) },
    { nombre: "Creando orden", accion: () => this.crearOrden(datosOrden) },
    { nombre: "Enviando confirmación", accion: () =>
    ↪  this.enviarEmailConfirmacion(datosOrden) }
  ];

  let pasosCompletados = 0;

  try {
    // Notificación inicial de progreso
```

```
    this.notification.add(`Iniciando procesamiento de orden (${pasos.length} pasos)...`,
        ↪ {
      type: "info"
    });

    for (const paso of pasos) {
      console.log(`Procesando paso: ${paso.nombre}`);

      // Ejecutar paso
      await paso.accion();
      pasosCompletados++;

      // Notificación de progreso
      this.notification.add(
        `${paso.nombre} completado (${pasosCompletados}/${pasos.length})`,
        { type: "info" }
      );
    }

    // Notificación de éxito y navegación
    this.notification.add("¡Orden procesada exitosamente!", {
      type: "success",
      sticky: false
    });

    // Navegar a la orden
    this.action.doAction({
      type: "ir.actions.act_window",
      res_model: "sale.order",
      res_id: datosOrden.idOrden,
      views: [[false, "form"]],
      target: "current"
    });

  } catch (error) {
    console.error(`Procesamiento de orden falló en paso ${pasosCompletados + 1}:`,
        ↪ error);

    this.notification.add(
      `Procesamiento de orden falló en paso: ${pasos[pasosCompletados]?.nombre ||
          ↪ 'Desconocido'}. ${error.message}`,
      { type: "danger", sticky: true }
    );

    // Opcionalmente mostrar diálogo de rollback
    this.dialog.add(ConfirmationDialog, {
      title: "Procesamiento Falló",
      body: "¿Te gustaría deshacer los cambios hechos hasta ahora?",
      confirm: () => this.deshacerCambios(pasosCompletados),
      cancel: () => console.log("Usuario eligió no deshacer")
    });
  }
}
```

Patrón 4: Gestión de Estado Basada en Servicios

Este patrón combina varios servicios dentro de un solo componente: `useState` para datos reactivos locales, y `orm/action/notification/dialog/user` coordinados en conjunto. Construiremos `DashboardClientes` en tres partes.

Primero, el `setup()` inicial: estado, referencias a servicios, y un pequeño objeto `cargadorDatos` que agrupa las consultas de solo lectura que necesita el dashboard:

```
export class DashboardClientes extends Component {
  static template = "mi_modulo.DashboardClientes";

  setup() {
    this.state = useState({
      clientes: [],
      clienteSeleccionado: null,
      cargando: false,
      filtros: {
        activo: true,
        pais: null,
        categoria: null
      }
    });

    // Inicializar servicios
    this.orm = useService("orm");
    this.action = useService("action");
    this.notification = useService("notification");
    this.dialog = useService("dialog");
    this.user = useService("user");

    // Cargador de datos basado en servicios
    this.cargadorDatos = {
      clientes: () => this.orm.searchRead(
        "res.partner",
        this.construirDominioClientes(),
        ["name", "email", "phone", "country_id", "category_id"],
        { order: "name asc" }
      ),

      paises: () => this.orm.searchRead(
        "res.country",
        [],
        ["name", "code"],
        { order: "name asc" }
      ),

      categorias: () => this.orm.searchRead(
        "res.partner.category",
        [],
        ["name", "color"],
        { order: "name asc" }
      )
    };

    // Inicializar dashboard
    useEffect(
```

```
      () => {
        this.inicializarDashboard();
      },
      () => []
    );

    // Actualizaciones reactivas de filtros
    useEffect(
      () => {
        this.refrescarClientes();
      },
      () => [this.state.filtros.activo, this.state.filtros.pais,
    ↪  this.state.filtros.categoria]
    );
  }
```

Después, el método que se ejecuta una vez al montar, cargando los tres conjuntos de datos en paralelo e informando éxito o fallo:

```
  async inicializarDashboard() {
    try {
      this.state.cargando = true;

      // Cargar todos los datos requeridos en paralelo
      const [clientes, paises, categorias] = await Promise.all([
        this.cargadorDatos.clientes(),
        this.cargadorDatos.paises(),
        this.cargadorDatos.categorias()
      ]);

      this.state.clientes = clientes;
      this.state.paises = paises;
      this.state.categorias = categorias;

      this.notification.add("¡Dashboard cargado exitosamente!", {
        type: "success"
      });

    } catch (error) {
      console.error("Inicialización de dashboard falló:", error);

      this.notification.add("Error al cargar datos del dashboard.", {
        type: "danger",
        sticky: true
      });
    } finally {
      this.state.cargando = false;
    }
  }
```

Finalmente, los métodos de acción masiva: una tabla de posibles acciones (archivar, eliminar, exportar), un paso de confirmación para las peligrosas, y el método que realmente ejecuta la acción elegida:

```
  // Acciones orquestadas por servicios
  async realizarAccionMasiva(accion, idsClientes) {
    const mapaAcciones = {
      archivar: {
```

```
        titulo: "Archivar Clientes",
        metodo: () => this.orm.write("res.partner", idsClientes, { active: false }),
        mensajeExito: `${idsClientes.length} clientes archivados exitosamente`
      },

      eliminar: {
        titulo: "Eliminar Clientes",
        metodo: () => this.orm.unlink("res.partner", idsClientes),
        mensajeExito: `${idsClientes.length} clientes eliminados exitosamente`,
        peligrosa: true
      },

      exportar: {
        titulo: "Exportar Clientes",
        metodo: () => this.action.doAction({
          type: "ir.actions.report",
          report_name: "base.report_partner_list",
          context: { active_ids: idsClientes }
        }),
        mensajeExito: "Exportación de clientes iniciada"
      }
    };

    const configAccion = mapaAcciones[accion];
    if (!configAccion) return;

    // Mostrar confirmación para acciones peligrosas
    if (configAccion.peligrosa) {
      this.dialog.add(ConfirmationDialog, {
        title: `Confirmar ${configAccion.titulo}`,
        body: `Esto ${accion} permanentemente ${idsClientes.length} clientes. Esta acción
            ↪ no se puede deshacer.`,
        confirm: () => this.ejecutarAccionMasiva(configAccion)
      });
    } else {
      await this.ejecutarAccionMasiva(configAccion);
    }
  }

  async ejecutarAccionMasiva(configAccion) {
    try {
      this.notification.add(`${configAccion.titulo} en progreso...`, {
        type: "info"
      });

      await configAccion.metodo();

      this.notification.add(configAccion.mensajeExito, {
        type: "success"
      });

      // Refrescar datos después de acción masiva
      await this.refrescarClientes();

    } catch (error) {
```

```
      console.error(`${configAccion.titulo} falló:`, error);

      this.notification.add(`${configAccion.titulo} falló: ${error.message}`, {
        type: "danger",
        sticky: true
      });
    }
  }
}
```

Optimización de Rendimiento de Servicios

Llamadas de Servicio con Debounce

Debounce significa retrasar una llamada a una función hasta que una ráfaga de eventos que la disparan se detenga durante un período determinado — por ejemplo, esperar a que el usuario deje de escribir antes de disparar una búsqueda, en lugar de enviar una solicitud por cada tecla presionada. Es una técnica común para evitar que llamadas costosas a servicios (como una búsqueda ORM) saturen el servidor.

```
setup() {
  this.orm = useService("orm");

  // Debounce para búsquedas costosas
  this.busquedaConDebounce = this.debounce(async (terminoBusqueda) => {
    if (!terminoBusqueda.trim()) return;

    try {
      const resultados = await this.orm.searchRead(
        "res.partner",
        [["name", "ilike", terminoBusqueda]],
        ["name", "email"],
        { limit: 10 }
      );

      this.state.resultadosBusqueda = resultados;
    } catch (error) {
      console.error("Búsqueda falló:", error);
    }
  }, 300); // Esperar 300ms después de que el usuario pare de escribir
}

debounce(func, espera) {
  let timeout;
  return (...args) => {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, args), espera);
  };
}

alCambiarBusqueda(event) {
  const terminoBusqueda = event.target.value;
  this.state.terminoBusqueda = terminoBusqueda;
```

```
  this.busquedaConDebounce(terminoBusqueda);
}
```

Odoo ya incluye un helper de debounce listo para usar, así que en addons reales rara vez necesitas escribir el tuyo: `import { debounce } from "@web/core/utils/timing";`.

Caché de Respuestas de Servicio

```
setup() {
  this.orm = useService("orm");
  this.cache = new Map();
  this.expiracionCache = new Map();

  this.obtenerDatosEnCache = async (claveCache, obtenedor, ttl = 5 * 60 * 1000) => {
    // Verificar si tenemos datos válidos en caché
    if (this.cache.has(claveCache)) {
      const expiracion = this.expiracionCache.get(claveCache);
      if (Date.now() < expiracion) {
        console.log(`Usando datos en caché para: ${claveCache}`);
        return this.cache.get(claveCache);
      }
    }

    // Obtener datos frescos
    console.log(`Obteniendo datos frescos para: ${claveCache}`);
    const datos = await obtenedor();

    // Cachear el resultado
    this.cache.set(claveCache, datos);
    this.expiracionCache.set(claveCache, Date.now() + ttl);

    return datos;
  };
}

async cargarPaises() {
  return this.obtenerDatosEnCache(
    'paises',
    () => this.orm.searchRead("res.country", [], ["name", "code"]),
    10 * 60 * 1000 // Cachear por 10 minutos
  );
}
```

Estrategias de Recuperación de Errores de Servicio

Reintento Automático con Backoff Exponencial

Backoff exponencial significa esperar cada vez más tiempo entre cada intento de reintento (1s, luego 2s, luego 4s, y así sucesivamente) en lugar de reintentar de inmediato — esto le da espacio a un servidor con problemas para recuperarse, en vez de bombardearlo con reintentos instantáneos.

```
async realizarOperacionResilientte(operacion, maxReintentos = 3) {
  let ultimoError;
```

```
  for (let intento = 1; intento <= maxReintentos; intento++) {
    try {
      const resultado = await operacion();

      if (intento > 1) {
        this.notification.add("Operación exitosa después de reintento.", {
          type: "success"
        });
      }

      return resultado;

    } catch (error) {
      ultimoError = error;
      console.warn(`Intento ${intento} falló:`, error);

      if (intento < maxReintentos) {
        const retraso = Math.pow(2, intento - 1) * 1000; // Backoff exponencial

        this.notification.add(`Operación falló, reintentando en ${retraso/1000}s...
            ↪ (${intento}/${maxReintentos})`, {
          type: "warning"
        });

        await new Promise(resolve => setTimeout(resolve, retraso));
      }
    }
  }

  // Todos los reintentos fallaron
  this.notification.add(`Operación falló después de ${maxReintentos} intentos:
      ↪ ${ultimoError.message}`, {
    type: "danger",
    sticky: true
  });

  throw ultimoError;
}

// Uso
async guardarDatos() {
  await this.realizarOperacionResiliente(async () => {
    return await this.orm.create("nombre.modelo", [this.state.datosFormulario]);
  });
}
```

Ejemplo de Integración del Mundo Real

Aquí hay un ejemplo completo mostrando cómo múltiples servicios trabajan juntos en un escenario de producción. Lo veremos en dos partes: el setup y la carga de datos, y luego la acción que procesa las órdenes seleccionadas.

```
export class CentroProcesamiento extends Component {
  static template = "mi_modulo.CentroProcesamiento";

  setup() {
    this.state = useState({
      ordenes: [],
      procesando: false,
      ordenesSeleccionadas: new Set(),
      estadisticas: {
        total: 0,
        pendientes: 0,
        procesando: 0,
        completadas: 0
      }
    });

    // Inicialización de servicios
    this.orm = useService("orm");
    this.action = useService("action");
    this.notification = useService("notification");
    this.dialog = useService("dialog");
    this.user = useService("user");

    // Cargar datos al montar
    useEffect(
      () => {
        this.cargarOrdenes();
        this.cargarEstadisticas();
      },
      () => []
    );
  }

  async cargarOrdenes() {
    try {
      const ordenes = 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.ordenes = ordenes;
    } catch (error) {
      this.notification.add("Error al cargar órdenes.", { type: "danger" });
    }
  }

  async cargarEstadisticas() {
    try {
      const stats = await this.orm.call("sale.order", "get_order_statistics");
      this.state.estadisticas = stats;
    } catch (error) {
      console.error("Error al cargar estadísticas:", error);
    }
```

```
}
```

La parte interesante es `procesarOrdenesSeleccionadas`: valida la selección, confirma con el usuario, y luego coordina una llamada ORM por lotes, una notificación de éxito, un refresco de datos, y finalmente abre un reporte — los cinco servicios trabajando juntos alrededor de una sola acción del usuario:

```
async procesarOrdenesSeleccionadas() {
  if (this.state.ordenesSeleccionadas.size === 0) {
    this.notification.add("Por favor selecciona órdenes para procesar.", {
      type: "warning"
    });
    return;
  }

  const idsOrdenes = Array.from(this.state.ordenesSeleccionadas);

  // Diálogo de confirmación
  this.dialog.add(ConfirmationDialog, {
    title: "Procesar Órdenes",
    body: `¿Procesar ${idsOrdenes.length} órdenes seleccionadas?`,
    confirm: async () => {
      try {
        this.state.procesando = true;

        // Notificación de progreso
        this.notification.add(`Procesando ${idsOrdenes.length} órdenes...`, {
          type: "info"
        });

        // Procesar órdenes por lotes
        await this.orm.call("sale.order", "action_confirm", idsOrdenes);

        // Retroalimentación de éxito
        this.notification.add(`¡${idsOrdenes.length} órdenes procesadas
            ↪ exitosamente!`, {
          type: "success"
        });

        // Limpiar selección y refrescar
        this.state.ordenesSeleccionadas.clear();
        await this.cargarOrdenes();
        await this.cargarEstadisticas();

        // Abrir reporte de procesamiento
        this.action.doAction({
          type: "ir.actions.report",
          report_name: "sale.action_report_saleorder",
          context: { active_ids: idsOrdenes }
        });

      } catch (error) {
        console.error("Procesamiento de órdenes falló:", error);

        this.notification.add(`Procesamiento falló: ${error.message}`, {
          type: "danger",
```

```
            sticky: true
          });
        } finally {
          this.state.procesando = false;
        }
      }
    });
  }
}
```

Resumen

Los servicios integrados de Odoo proporcionan un kit de herramientas integral para crear aplicaciones profesionales e integradas:

- **`notification`**: Retroalimentación del usuario y actualizaciones de estado
- **`action`**: Navegación e integración con Odoo
- **`dialog`**: Interacciones del usuario y confirmaciones
- **`user`**, **`company`**, **`router`**: Información del sistema y navegación

Al dominar estos servicios, tus componentes: - Se sentirán nativos a la experiencia de Odoo - Proporcionarán interacciones consistentes del usuario - Manejarán errores graciosamente - Se integrarán perfectamente con los flujos de trabajo de Odoo

La clave del éxito es entender cuándo y cómo usar cada servicio, combinarlos efectivamente para flujos de trabajo complejos, y siempre priorizar la experiencia del usuario con retroalimentación clara e interacciones intuitivas.

En el próximo capítulo, exploraremos patrones avanzados de componentes y consideraciones arquitectónicas para construir aplicaciones OWL a gran escala.

TL;DR: Compón servicios de forma deliberada (una notificación por cada paso de un flujo de varios pasos), aplica debounce a todo lo que se dispara en cada tecla con `@web/core/utils/timing`, y envuelve las llamadas poco confiables en una estrategia de reintento con backoff en vez de fallar en silencio.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch13_2_ex1`. Las instrucciones de instalación están en el README del repositorio.

Ejercicios

1. **Flujo compuesto**: Escribe un método que cree un partner, y luego cree inmediatamente un contacto relacionado para él, mostrando una notificación de progreso por cada paso (similar al Patrón 3). Asegúrate de que una falla en cualquiera de los pasos muestre cuál falló.
2. **Debounce por tu cuenta**: Implementa un debounce de 300ms sobre un input de búsqueda en vivo respaldado por `orm.searchRead`, sin usar el helper `debounce` incorporado de Odoo — luego reemplázalo con `import { debounce } from "@web/core/utils/timing"` y confirma que el comportamiento es el mismo.
3. **Reintento con backoff**: Toma `realizarOperacionResilientte` y adáptalo para reintentar una llamada `orm.call` hasta 3 veces, luego escribe una prueba rápida (real o en papel) describiendo qué notificación debería ver el usuario después de cada intento fallido.

¿Qué Sigue?

Ya cubriste toda la capa de servicios, desde la primera notificación hasta flujos de trabajo resilientes de varios pasos. El Capítulo 14 pasa a la composición — slots, scoped slots y `t-portal` — las herramientas para construir componentes reutilizables y flexibles en vez de componentes de un solo uso.

Capítulo 14: Composición Avanzada con Slots

> **¿Por qué este capítulo?** En trabajo real con clientes, rara vez construyes un componente una sola vez y nunca más lo tocas — el siguiente proyecto pide "el mismo modal, pero con un footer distinto" o "la misma lista, pero cada fila necesita una acción personalizada." Los slots son lo que te permite decir que sí sin duplicar el componente.
>
> **¿Qué trata de resolver Odoo con esto?** La propia UI de Odoo (tarjetas kanban, diálogos, filas de lista) tiene que mantenerse flexible entre modelos y casos de uso muy distintos sin una explosión de widgets casi idénticos. Los slots permiten que un solo componente sea dueño de la estructura mientras quien lo llama es dueño del contenido.
>
> **Aplicación en la vida real:** Un wrapper de diálogo de confirmación reutilizable en una docena de addons para el mismo cliente, o un componente de tabla genérico cuyo renderizado de celdas cambia por vista sin bifurcar la tabla misma.

Hasta ahora, nuestros componentes han sido unidades autocontenidas. Un componente `Card` siempre tiene un título y contenido. Una `PartnerList` siempre renderiza una lista de partners en un formato predefinido. Pero, ¿qué pasa si queremos crear componentes flexibles y reutilizables que puedan adaptarse a diferentes casos de uso?

Imagina que estás construyendo un componente genérico `Modal`. El modal en sí proporciona el overlay de fondo, el contenedor blanco, el posicionamiento y la funcionalidad del botón de cerrar. Pero el contenido *dentro* del modal—ya sea un mensaje de confirmación, un formulario complejo, una lista de imágenes o un reproductor de video—debería depender completamente del componente padre que lo usa.

Aquí es donde la **composición** se vuelve esencial, y el mecanismo de OWL para lograr esto son los **slots**. Los slots son marcadores de posición en la plantilla de un componente hijo que los componentes padre pueden llenar con su propio contenido. Piensa en ellos como "agujeros de contenido" que los padres pueden tapar con markup personalizado, creando infinitas posibilidades desde un solo componente bien diseñado.

Este patrón es fundamental para construir librerías de componentes escalables y se usa extensivamente a lo largo de la interfaz de Odoo.

Entendiendo el Problema: Por Qué Necesitamos Composición

Antes de sumergirnos en slots, entendamos el problema que resuelven con un ejemplo concreto.

El Enfoque Rígido (Sin Slots)

Imagina que creamos un componente `ProductCard` que muestra información del producto:

```
// ProductCard.js
import { Component, xml } from "@odoo/owl";

export class ProductCard extends Component {
    static template = xml`
        <div class="product-card">
            <h3 t-esc="props.producto.nombre"/>
            <p t-esc="props.producto.descripcion"/>
            <div class="precio">$<t t-esc="props.producto.precio"/></div>
            <button class="btn btn-primary">Agregar al Carrito</button>
        </div>
    `;
}
```

Esto funciona bien para un listado básico de productos. Pero ¿qué pasa cuando necesitas: - Una versión con miniatura de imagen? - Una versión con reseñas de clientes? - Una versión con selectores de cantidad? - Una versión con etiquetas de descuento?

Sin composición, terminarías creando múltiples componentes similares (`ProductCardWithImage`, `ProductCardWithReviews`, etc.) o agregando docenas de props condicionales (`showImage`, `showReviews`, `showQuantity`). Ambos enfoques llevan a código rígido y difícil de mantener.

El Enfoque Flexible (Con Slots)

Con slots, creas un solo componente flexible `ProductCard` que proporciona la estructura y el estilo, mientras permite a los padres inyectar contenido personalizado:

```
// FlexibleProductCard.js
export class FlexibleProductCard extends Component {
    static template = xml`
        <div class="product-card">
            <t t-slot="header"/>
            <h3 t-esc="props.producto.nombre"/>
            <t t-slot="content"/>
            <div class="precio">$<t t-esc="props.producto.precio"/></div>
            <t t-slot="actions"/>
        </div>
    `;
}
```

Ahora los padres pueden personalizar cada sección:

```
<!-- En plantilla del padre -->
<FlexibleProductCard producto="producto">
  <t t-set-slot="header">
    <img t-att-src="producto.urlImagen" class="producto-miniatura"/>
    <span class="badge badge-oferta" t-if="producto.enOferta">¡OFERTA!</span>
  </t>

  <t t-set-slot="content">
    <p t-esc="producto.descripcion"/>
    <div class="reseñas">
      <t t-foreach="producto.reseñas.slice(0, 3)" t-as="reseña" t-key="reseña.id">
        <div class="reseña"><t t-esc="reseña.texto"/></div>
```

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

  <t t-set-slot="actions">
    <input type="number" t-model="state.cantidad" min="1" max="10"/>
    <button class="btn btn-primary" t-on-click="agregarAlCarrito">
      Agregar <t t-esc="state.cantidad"/> al Carrito
    </button>
  </t>
</FlexibleProductCard>
```

Este único componente ahora puede manejar infinitas variaciones sin volverse hinchado o complejo.

El Slot Por Defecto: Tu Primer Paso en la Composición

El caso de uso más común para slots es cuando necesitas un simple marcador de posición de contenido. Esto se maneja con el **slot por defecto**—un único slot que no necesita ser nombrado.

Creando un Componente Modal

Construyamos un componente reutilizable `Modal` paso a paso:

1. Definir la Estructura del Modal (`modal.xml`)

```
<t t-name="mi_modulo.Modal" owl="1">
  <div class="modal-backdrop" t-on-click="cerrarSiClickAfuera">
    <div class="modal-dialog" t-on-click.stop="">
      <div class="modal-content">
        <!-- Encabezado del Modal -->
        <div class="modal-header">
          <h5 class="modal-title" t-esc="props.titulo or 'Modal'"/>
          <button type="button"
                  class="btn-close"
                  t-on-click="cerrar"
                  aria-label="Cerrar">
            <span aria-hidden="true">&times;</span>
          </button>
        </div>

        <!-- Cuerpo del Modal - Aquí va el contenido del padre -->
        <div class="modal-body">
          <t t-slot="default"/>
        </div>

        <!-- Pie del Modal (si se proporciona) -->
        <div class="modal-footer" t-if="props.slots.footer">
          <t t-slot="footer"/>
        </div>
      </div>
    </div>
  </div>
</t>
```

2. La Clase del Componente Modal (`modal.js`)

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

export class Modal extends Component {
    static template = "mi_modulo.Modal";

    setup() {
        // Cerrar modal cuando se presiona Escape
        this.onKeydown = this.onKeydown.bind(this);
        document.addEventListener('keydown', this.onKeydown);

        onWillUnmount(() => {
            document.removeEventListener('keydown', this.onKeydown);
        });
    }

    onKeydown(event) {
        if (event.key === 'Escape') {
            this.cerrar();
        }
    }

    cerrar() {
        if (this.props.onCerrar) {
            this.props.onCerrar();
        }
    }

    cerrarSiClickAfuera(event) {
        // Solo cerrar si se hace click en el backdrop, no en el contenido del modal
        if (event.target === event.currentTarget) {
            this.cerrar();
        }
    }
}
```

3. Usando el Modal desde un Componente Padre

```
<t t-name="mi_modulo.Dashboard" owl="1">
  <div class="dashboard">
    <h1>Mi Dashboard</h1>
    <button class="btn btn-primary" t-on-click="abrirModalConfirmacion">
      Eliminar Todos los Datos
    </button>

    <!-- Modal de Confirmación -->
    <t t-if="state.mostrarModalConfirmacion">
      <Modal titulo="Confirmar Eliminación" onCerrar="cerrarModalConfirmacion">
        <!-- Todo entre estas etiquetas va al slot por defecto -->
        <div class="alert alert-danger">
          <h4>Advertencia</h4>
          <p>Esta acción eliminará permanentemente todos tus datos. Esto no se puede
    ↪  deshacer.</p>
          <p>Escribe <strong>ELIMINAR</strong> para confirmar:</p>
          <input type="text"
                 class="form-control"
                 t-model="state.textoConfirmacion"
```

```
                placeholder="Escribe ELIMINAR aquí"/>
        </div>

        <!-- Slot de footer para botones personalizados -->
        <t t-set-slot="footer">
          <button class="btn btn-secondary" t-on-click="cerrarModalConfirmacion">
            Cancelar
          </button>
          <button class="btn btn-danger"
                  t-att-disabled="state.textoConfirmacion !== 'ELIMINAR'"
                  t-on-click="realizarEliminacion">
            Eliminar Todo
          </button>
        </t>
      </Modal>
    </t>
  </div>
</t>
```

4. La Lógica del Componente Dashboard

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

export class Dashboard extends Component {
    static template = "mi_modulo.Dashboard";
    static components = { Modal };

    setup() {
        this.state = useState({
            mostrarModalConfirmacion: false,
            textoConfirmacion: ""
        });
    }

    abrirModalConfirmacion() {
        this.state.mostrarModalConfirmacion = true;
        this.state.textoConfirmacion = "";
    }

    cerrarModalConfirmacion() {
        this.state.mostrarModalConfirmacion = false;
    }

    realizarEliminacion() {
        if (this.state.textoConfirmacion === 'ELIMINAR') {
            // Realizar la eliminación actual
            console.log("Eliminando todos los datos...");
            this.cerrarModalConfirmacion();
        }
    }
}
```

Beneficios Clave de Este Enfoque

1. Reutilización: El mismo componente `Modal` puede mostrar diálogos de confirmación, formularios, galerías de imágenes, o cualquier otro contenido.

2. Separación de Responsabilidades: El modal maneja posicionamiento, backdrop, eventos de teclado y estilo. El padre maneja el contenido específico y la lógica de negocio.

3. Mantenibilidad: El comportamiento del modal está centralizado. Arreglar un bug o agregar una característica (como animación) beneficia a todos los modales en tu aplicación.

Slots Nombrados: Múltiples Áreas de Contenido

Cuando necesitas más control sobre dónde van diferentes piezas de contenido, los **slots nombrados** te permiten definir múltiples áreas distintas para inyección de contenido.

Construyendo un Layout Flexible de Página

Creemos un componente `PageLayout` que proporciona una estructura estándar de página:

1. La Plantilla PageLayout (`page_layout.xml`)

```
<t t-name="mi_modulo.PageLayout" owl="1">
  <div class="page-container">
    <!-- Área de Navegación -->
    <nav class="page-navigation" t-if="props.slots.navigation">
      <t t-slot="navigation"/>
    </nav>

    <!-- Sección de Encabezado -->
    <header class="page-header">
      <t t-slot="header">
        <!-- Contenido de encabezado por defecto si no se proporciona -->
        <h1 t-esc="props.titulo or 'Título de Página'"/>
      </t>
    </header>

    <!-- Barra Lateral (si se proporciona) -->
    <aside class="page-sidebar" t-if="props.slots.sidebar">
      <t t-slot="sidebar"/>
    </aside>

    <!-- Área de Contenido Principal -->
    <main class="page-content" t-att-class="{ 'with-sidebar': props.slots.sidebar }">
      <t t-slot="default">
        <!-- Contenido por defecto si no se proporciona -->
        <p>No se proporcionó contenido.</p>
      </t>
    </main>

    <!-- Barra de Acciones -->
    <div class="page-actions" t-if="props.slots.actions">
      <t t-slot="actions"/>
    </div>

    <!-- Pie de Página -->
    <footer class="page-footer" t-if="props.slots.footer">
      <t t-slot="footer"/>
    </footer>
```

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

2. Usando Slots Nombrados desde Múltiples Padres

Ejemplo 1: Una Página de Configuración

```
<PageLayout titulo="Configuración de Usuario">
  <t t-set-slot="navigation">
    <ul class="nav-menu">
      <li><a href="#perfil">Perfil</a></li>
      <li><a href="#seguridad">Seguridad</a></li>
      <li><a href="#notificaciones">Notificaciones</a></li>
    </ul>
  </t>

  <t t-set-slot="sidebar">
    <div class="settings-sidebar">
      <h3>Configuración Rápida</h3>
      <label>
        <input type="checkbox" t-model="state.modoOscuro"/>
        Modo Oscuro
      </label>
      <label>
        <input type="checkbox" t-model="state.notificaciones"/>
        Notificaciones por Email
      </label>
    </div>
  </t>

  <!-- El contenido principal va al slot por defecto -->
  <div class="settings-content">
    <h2>Configuración de Perfil</h2>
    <form t-on-submit.prevent="guardarConfiguracion">
      <div class="form-group">
        <label>Nombre Completo</label>
        <input type="text" t-model="state.usuario.nombre" class="form-control"/>
      </div>
      <div class="form-group">
        <label>Email</label>
        <input type="email" t-model="state.usuario.email" class="form-control"/>
      </div>
    </form>
  </div>

  <t t-set-slot="actions">
    <button class="btn btn-secondary"
    ↪  t-on-click="reiniciarConfiguracion">Reiniciar</button>
    <button class="btn btn-primary" t-on-click="guardarConfiguracion">Guardar
    ↪  Cambios</button>
  </t>
</PageLayout>
```

Ejemplo 2: Una Página de Catálogo de Productos

```
<PageLayout titulo="Catálogo de Productos">
  <t t-set-slot="header">
    <div class="catalog-header">
```

```
      <h1>Nuestros Productos</h1>
      <div class="search-bar">
        <input type="text"
               placeholder="Buscar productos..."
               t-model="state.consultaBusqueda"
               class="form-control"/>
      </div>
    </div>
  </t>

  <t t-set-slot="sidebar">
    <div class="filtros">
      <h3>Filtros</h3>
      <div class="filter-group">
        <label>Categoría</label>
        <select t-model="state.categoriaSeleccionada" class="form-control">
          <option value="">Todas las Categorías</option>
          <t t-foreach="state.categorias" t-as="categoria" t-key="categoria.id">
            <option t-att-value="categoria.id" t-esc="categoria.nombre"/>
          </t>
        </select>
      </div>
      <div class="filter-group">
        <label>Rango de Precio</label>
        <input type="range" t-model="state.precioMaximo" min="0" max="1000"/>
        <span>Hasta $<t t-esc="state.precioMaximo"/></span>
      </div>
    </div>
  </t>

  <!-- Cuadrícula de productos en contenido principal -->
  <div class="product-grid">
    <t t-foreach="productosFiltrados" t-as="producto" t-key="producto.id">
      <ProductCard producto="producto"/>
    </t>
  </div>

  <t t-set-slot="footer">
    <div class="paginacion">
      <button t-att-disabled="state.paginaActual === 1"
              t-on-click="paginaAnterior">
        Anterior
      </button>
      <span>Página <t t-esc="state.paginaActual"/> de <t
    ↪  t-esc="state.totalPaginas"/></span>
      <button t-att-disabled="state.paginaActual === state.totalPaginas"
              t-on-click="paginaSiguiente">
        Siguiente
      </button>
    </div>
  </t>
</PageLayout>
```

Entendiendo la Detección de Slots

Nota el patrón `t-if="props.slots.sidebar"` en nuestra plantilla. OWL automáticamente proporciona un objeto `props.slots` que te dice qué slots ha llenado el padre:

```
<!-- Solo renderizar el contenedor sidebar si el padre proporcionó contenido sidebar -->
<aside class="page-sidebar" t-if="props.slots.sidebar">
  <t t-slot="sidebar"/>
</aside>

<!-- Aplicar clase CSS diferente basada en si existe el sidebar -->
<main class="page-content" t-att-class="{ 'with-sidebar': props.slots.sidebar }">
  <t t-slot="default"/>
</main>
```

Esto permite a tu componente adaptar su layout dinámicamente basado en qué contenido proporciona el padre.

Slots con Ámbito: Compartiendo Datos Entre Hijo y Padre

El patrón de slot más avanzado son los **slots con ámbito**, donde el componente hijo pasa datos al contenido del slot del padre. Esto permite componentes increíblemente flexibles que manejan lógica de datos mientras dan a los padres control completo sobre la presentación.

Construyendo una Tabla de Datos Genérica

Creemos un componente `DataTable` que maneja ordenamiento, filtrado y paginación, pero permite a los padres personalizar completamente cómo se renderiza cada fila:

1. El Componente DataTable (`data_table.js`)

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

export class DataTable extends Component {
    static template = "mi_modulo.DataTable";

    setup() {
        this.state = useState({
            campoOrden: null,
            direccionOrden: 'asc',
            paginaActual: 1,
            tamañoPagina: this.props.tamañoPagina || 10,
            consultaBusqueda: ""
        });
    }

    get datosFiltrados() {
        let datos = this.props.datos || [];

        // Aplicar filtro de búsqueda
        if (this.state.consultaBusqueda) {
            const consulta = this.state.consultaBusqueda.toLowerCase();
            datos = datos.filter(item =>
                Object.values(item).some(value =>
                    String(value).toLowerCase().includes(consulta)
```

```
                )
            );
        }

        // Aplicar ordenamiento
        if (this.state.campoOrden) {
            datos = [...datos].sort((a, b) => {
                const aVal = a[this.state.campoOrden];
                const bVal = b[this.state.campoOrden];

                if (aVal < bVal) return this.state.direccionOrden === 'asc' ? -1 : 1;
                if (aVal > bVal) return this.state.direccionOrden === 'asc' ? 1 : -1;
                return 0;
            });
        }

        return datos;
    }

    get datosPaginados() {
        const inicio = (this.state.paginaActual - 1) * this.state.tamañoPagina;
        const fin = inicio + this.state.tamañoPagina;
        return this.datosFiltrados.slice(inicio, fin);
    }

    get totalPaginas() {
        return Math.ceil(this.datosFiltrados.length / this.state.tamañoPagina);
    }

    ordenarPor(campo) {
        if (this.state.campoOrden === campo) {
            this.state.direccionOrden = this.state.direccionOrden === 'asc' ? 'desc' :
                ↪  'asc';
        } else {
            this.state.campoOrden = campo;
            this.state.direccionOrden = 'asc';
        }
        this.state.paginaActual = 1; // Reiniciar a la primera página al ordenar
    }

    paginaSiguiente() {
        if (this.state.paginaActual < this.totalPaginas) {
            this.state.paginaActual++;
        }
    }

    paginaAnterior() {
        if (this.state.paginaActual > 1) {
            this.state.paginaActual--;
        }
    }
}
```

2. La Plantilla DataTable (`data_table.xml`)

```
<t t-name="mi_modulo.DataTable" owl="1">
```

```
<div class="data-table-container">
  <!-- Barra de Búsqueda -->
  <div class="table-controls">
    <input type="text"
           class="form-control search-input"
           placeholder="Buscar..."
           t-model="state.consultaBusqueda"/>
  </div>

  <!-- Encabezado de Tabla (si se proporciona) -->
  <div class="table-header" t-if="props.slots.header">
    <t t-slot="header"
       ordenarPor="ordenarPor"
       campoOrden="state.campoOrden"
       direccionOrden="state.direccionOrden"/>
  </div>

  <!-- Cuerpo de Tabla -->
  <div class="table-body">
    <t t-if="datosPaginados.length === 0">
      <div class="empty-state">
        <t t-slot="empty">
          <p>No hay datos disponibles</p>
        </t>
      </div>
    </t>
    <t t-else="">
      <t t-foreach="datosPaginados" t-as="item" t-key="item.id">
        <!-- Pasar datos del item y utilidades al slot del padre -->
        <t t-slot="row"
           item="item"
           indice="item_index"
           esPar="item_index % 2 === 0"/>
      </t>
    </t>
  </div>

  <!-- Paginación -->
  <div class="table-pagination" t-if="totalPaginas > 1">
    <button class="btn btn-sm btn-secondary"
            t-att-disabled="state.paginaActual === 1"
            t-on-click="paginaAnterior">
      Anterior
    </button>

    <span class="pagination-info">
      Página <t t-esc="state.paginaActual"/> de <t t-esc="totalPaginas"/>
      (<t t-esc="datosFiltrados.length"/> elementos totales)
    </span>

    <button class="btn btn-sm btn-secondary"
            t-att-disabled="state.paginaActual === totalPaginas"
            t-on-click="paginaSiguiente">
      Siguiente
    </button>
```

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

3. Usando la Tabla de Datos con Ámbito

Ahora los padres pueden usar esta tabla poderosa mientras tienen control completo sobre cómo se muestran los datos:

```
<!-- En un componente padre -->
<DataTable datos="state.clientes" tamañoPagina="5">
  <!-- Encabezado personalizado ordenable -->
  <t t-set-slot="header" t-slot-scope="headerProps">
    <div class="table-header-row">
      <div class="header-cell sortable"
           t-on-click="() => headerProps.ordenarPor('nombre')"
           t-att-class="{
               'sort-asc': headerProps.campoOrden === 'nombre' &&
    ↪  headerProps.direccionOrden === 'asc',
               'sort-desc': headerProps.campoOrden === 'nombre' &&
    ↪  headerProps.direccionOrden === 'desc'
           }">
        Nombre del Cliente
      </div>
      <div class="header-cell sortable"
           t-on-click="() => headerProps.ordenarPor('email')">
        Email
      </div>
      <div class="header-cell sortable"
           t-on-click="() => headerProps.ordenarPor('totalOrdenes')">
        Total de Órdenes
      </div>
      <div class="header-cell">Acciones</div>
    </div>
  </t>

  <!-- Renderizado personalizado de fila -->
  <t t-set-slot="row" t-slot-scope="rowProps">
    <div class="table-row" t-att-class="{ 'even': rowProps.esPar }">
      <div class="cell">
        <img t-att-src="rowProps.item.avatar" class="avatar-sm"/>
        <strong t-esc="rowProps.item.nombre"/>
        <span t-if="rowProps.item.esVip" 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.totalOrdenes"/>
        <small t-if="rowProps.item.fechaUltimaOrden">
          (Última: <t t-esc="formatearFecha(rowProps.item.fechaUltimaOrden)"/>)
        </small>
      </div>

      <div class="cell actions">
```

```
        <button class="btn btn-sm btn-primary"
                t-on-click="() => this.editarCliente(rowProps.item)">
          Editar
        </button>
        <button class="btn btn-sm btn-danger"
                t-on-click="() => this.eliminarCliente(rowProps.item)">
          Eliminar
        </button>
      </div>
    </div>
  </t>

  <!-- Estado vacío personalizado -->
  <t t-set-slot="empty">
    <div class="text-center p-4">
      <h3>No se encontraron clientes</h3>
      <p>Intenta ajustar tus criterios de búsqueda o agrega tu primer cliente.</p>
      <button class="btn btn-primary" t-on-click="abrirModalAgregarCliente">
        Agregar Primer Cliente
      </button>
    </div>
  </t>
</DataTable>
```

Entendiendo las Props de Slots con Ámbito

La clave de los slots con ámbito es la directiva `t-slot-scope` en `<t t-set-slot>`:

```
<!-- El hijo pasa datos como atributos en la llamada a t-slot -->
<t t-slot="row"
   item="item"
   indice="item_index"
   esPar="item_index % 2 === 0"/>

<!-- El padre nombra el objeto de ámbito con t-slot-scope y recibe todo en él -->
<t t-set-slot="row" t-slot-scope="rowProps">
  <!-- rowProps contiene: { item: {...}, indice: 0, esPar: true } -->
  <div>Nombre del item: <t t-esc="rowProps.item.nombre"/></div>
</t>
```

Este patrón permite al componente hijo proporcionar tanto datos como funciones de utilidad a la lógica de renderizado del padre.

Renderizando Fuera del Árbol: `t-portal`

Los slots dejan que un padre controle *qué* renderiza un hijo. `t-portal` resuelve un problema distinto: deja que un componente renderice *dónde* en el DOM real aparece su contenido, independientemente de dónde esté ese componente en el árbol de componentes de OWL.

¿Por qué querrías eso? Piensa en un diálogo modal o un tooltip disparado desde muy adentro de una página—digamos, desde una fila de una tabla dentro de una tarjeta dentro de una barra lateral. Si el modal se renderiza normalmente, hereda cualquier `overflow: hidden` o contexto de apilamiento `z-index` que sus ancestros hayan definido, lo que puede recortarlo o enterrarlo detrás de otro contenido. La solución que los desarrolladores web usan desde hace años es renderizar el DOM de ese modal

como hijo directo de `<body>`, evitando el estilo de todos sus ancestros—mientras sigue siendo un componente OWL normal, totalmente reactivo, lógicamente propiedad de su padre.

`t-portal` hace exactamente eso: la posición lógica del componente en el árbol (props, estado, ciclo de vida) no cambia, solo *dónde termina su DOM*.

```
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 });
    }
}
```

El atributo `t-portal` recibe un selector CSS, como string, indicando dónde debe adjuntarse el contenido—aquí, `'body'`. OWL inserta un nodo marcador vacío en la ubicación original del tooltip para llevar la cuenta internamente, pero el elemento `.tooltip-content` real se agrega a `<body>`, libre de cualquier recorte o problema de apilamiento de sus ancestros lógicos.

Recurre a `t-portal` específicamente para modales, tooltips y menús desplegables—contenido que necesita escapar visualmente del layout de su padre mientras sigue siendo, desde el punto de vista de OWL, una parte normal del componente que lo creó.

Mejores Prácticas para Arquitectura Basada en Slots

1. Diseña Slots con Propósito

Cada slot debe tener una responsabilidad clara y única: - **`header`**: Títulos de página, navegación, barras de búsqueda - **`actions`**: Botones, controles, menús de acción - **`footer`**: Paginación, totales, texto de ayuda - **`default`**: Área de contenido principal

Evita slots genéricos como `slot1`, `slot2` que no transmiten significado.

2. Proporciona Valores Por Defecto Sensatos

Siempre proporciona contenido por defecto para slots cuando tenga sentido:

```
<t t-slot="header">
  <!-- Encabezado por defecto si el padre no proporciona uno -->
  <h1 t-esc="props.titulo or 'Página Sin Título'"/>
</t>
```

Esto hace que tu componente funcione inmediatamente mientras sigue siendo personalizable.

3. Documenta tu API de Slots

Al crear componentes reutilizables, documenta qué slots están disponibles y qué props proporcionan los slots con ámbito:

```
/**
 * Componente DataTable
 *
 * Slots:
 * - header: Encabezados de columna de tabla (props: { ordenarPor, campoOrden,
    ↪  direccionOrden })
 * - row: Renderizado de fila individual (props: { item, indice, esPar })
 * - empty: Se muestra cuando no hay datos disponibles
 *
 * Props:
 * - datos: Array de objetos para mostrar
 * - tamañoPagina: Número de elementos por página (por defecto: 10)
 */
export class DataTable extends Component {
    // ...
}
```

4. Usa Renderizado Condicional de Slots

Adapta el layout de tu componente basado en qué slots se proporcionan:

```
<!-- Ajustar el ancho del contenido principal basado en la presencia del sidebar -->
<main t-att-class="{
    'col-12': !props.slots.sidebar,
    'col-9': props.slots.sidebar
}">
  <t t-slot="default"/>
</main>
```

5. Combina Slots con Props para Máxima Flexibilidad

Algunas áreas de contenido podrían ser lo suficientemente simples para props, mientras otras necesitan el poder completo de los slots:

```
// Texto simple puede ser una prop
static props = {
    titulo: { type: String, optional: true },
    subtitulo: { type: String, optional: true },
    // Contenido complejo usa slots
};

<div class="card">
  <!-- Contenido simple via props -->
  <h3 t-if="props.titulo" t-esc="props.titulo"/>
  <p t-if="props.subtitulo" t-esc="props.subtitulo"/>

  <!-- Contenido complejo via slots -->
  <t t-slot="default"/>
</div>
```

Errores Comunes y Cómo Evitarlos

1. Uso Excesivo de Slots

No todo necesita ser un slot. Si el contenido raramente cambia, una prop podría ser más simple:

```
// En lugar de esto...
<Modal>
    <t t-set-slot="titulo">¿Guardar Cambios?</t>
</Modal>

// Considera esto...
<Modal titulo="¿Guardar Cambios?">
```

2. Olvidar Manejar Slots Vacíos

Siempre verifica si un slot tiene contenido antes de renderizar su contenedor:

```
<!-- Bueno: Solo renderizar footer si se proporciona contenido -->
<footer t-if="props.slots.footer" class="modal-footer">
  <t t-slot="footer"/>
</footer>

<!-- Malo: div de footer vacío si no hay contenido -->
<footer class="modal-footer">
  <t t-slot="footer"/>
</footer>
```

3. Lógica Compleja en Plantillas

Mantén la lógica de slots simple. Mueve computaciones complejas a getters o métodos:

```
// Bueno: Lógica en componente
get deberiaMostrarSidebar() {
    return this.props.slots.sidebar && !this.props.ocultarSidebar;
}

<!-- Plantilla simple -->
<aside t-if="deberiaMostrarSidebar">
  <t t-slot="sidebar"/>
</aside>
```

Dominar slots transforma cómo piensas sobre el diseño de componentes. En lugar de crear componentes rígidos de un solo propósito, construyes fundamentos flexibles que se adaptan a infinitos casos de uso. Este enfoque es esencial para construir aplicaciones Odoo mantenibles y escalables y se usa a lo largo del código base de Odoo para máxima reutilización y personalización.

En el próximo capítulo, exploraremos cómo gestionar estado que necesita ser compartido entre componentes que no están directamente relacionados en el árbol de componentes.

TL;DR: Los slots dejan que un padre inyecte contenido (por defecto, nombrado o scoped) dentro de la estructura de un hijo, así construyes un componente flexible en vez de muchos rígidos y casi idénticos.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch14_ex1`. Las instrucciones de instalación están en el README del repositorio.

Ejercicios

1. **Modal con dos slots.** Construye un componente `Modal` con un slot por defecto para el cuerpo y un slot nombrado `footer`. Renderízalo una vez solo con cuerpo, y otra vez con cuerpo y footer, confirmando que el área del footer desaparece limpiamente cuando no se usa (ver "Olvidar Manejar Slots Vacíos" arriba).
2. **Práctica de scoped slot.** Extiende el `DataTable` de este capítulo para que el slot del padre reciba no solo el `record` de la fila, sino también su `index`. Úsalo en el template del padre para renderizar colores alternados por fila.
3. **Slot vs. prop.** Toma un componente que hayas construido con un slot y pregúntate: ¿este contenido podría ser en cambio una prop simple? Reescríbelo como prop si es así, y escribe una oración sobre cuál versión es más fácil de usar desde el lado del padre.

¿Qué Sigue?

Ya cubrimos cómo fluye el contenido entre componentes directamente relacionados como padre e hijo. El Capítulo 15 aborda el caso más difícil: estado compartido entre componentes que no están relacionados en absoluto dentro del árbol, usando un store global.

Capítulo 15: Gestión de Estado Global con useStore

> **¿Por qué este capítulo?** Tarde o temprano, dos componentes que no tienen nada que ver entre sí en el árbol necesitan reaccionar al mismo cambio — un badge de notificaciones y una lista sin relación que ambos deben reflejar un registro nuevo. Las props solas no pueden cruzar el árbol, y recurrir a un objeto global mutable o a eventos del DOM es la forma de terminar con bugs que nadie puede rastrear.
>
> **¿Qué trata de resolver Odoo con esto?** Coordinar estado de UI compartido sin abandonar el modelo de reactividad de OWL — un store debería sentirse como `useState`, solo que visible desde más de un lugar, registrado de la misma forma que cualquier otro servicio.
>
> **Aplicación en la vida real:** Un flag de "cambios sin guardar" que muestran varios widgets en la misma vista de formulario, o un estado compartido de borrador/carrito en una pantalla personalizada de varios pasos donde los pasos no tienen una relación directa padre-hijo.

Hemos dominado los fundamentos del flujo de datos en OWL: las props fluyen hacia abajo de padre a hijo, y los eventos burbujean hacia arriba de hijo a padre. Este patrón funciona hermosamente para componentes que están directamente relacionados—un formulario padre y sus campos de entrada, una lista y sus elementos, una tarjeta y sus botones.

Pero, ¿qué sucede cuando los componentes están muy separados en el árbol de componentes, o cuando múltiples componentes no relacionados necesitan compartir los mismos datos?

Imagina que tienes un componente `UserMenu` en el encabezado principal que muestra el nombre y avatar del usuario actual. En algún otro lugar de tu aplicación, tienes una página `ProfileSettings` donde los usuarios pueden actualizar su información. Cuando un usuario cambia su nombre en la página `ProfileSettings`, ¿cómo se entera el `UserMenu` en el encabezado—posiblemente renderizado por una parte completamente diferente de la aplicación?

Pasar props hacia abajo a través de docenas de componentes intermedios (un antipatrón doloroso conocido como "prop drilling") es ineficiente y crea una pesadilla de mantenimiento. Emitir un evento que tiene que burbujear por todo el árbol de la aplicación es igualmente desordenado y frágil.

La solución es un **store de estado global**—una fuente única y centralizada de verdad para datos que necesitan ser compartidos a través de toda tu aplicación. Cualquier componente puede acceder a estos datos o disparar cambios en ellos, y cualquier componente que use esos datos se actualizará automáticamente cuando cambien.

Entendiendo el Problema: Cuando el Estado Local No Es Suficiente

Antes de sumergirnos en la gestión de estado global, entendamos exactamente cuándo y por qué la necesitas.

El Problema del Prop Drilling

Considera esta jerarquía de componentes en una aplicación Odoo:

```
App
|-- Header
|   |-- Navigation
|   |-- UserMenu (necesita datos del usuario)
|-- Main
|   |-- Sidebar
|   |   |-- QuickActions (necesita permisos del usuario)
|   |-- Content
|       |-- Dashboard (necesita preferencias del usuario)
|       |-- ProfilePage
|           |-- ProfileForm (actualiza datos del usuario)
|-- Footer
    |-- UserInfo (necesita datos del usuario)
```

Sin gestión de estado global, compartir datos del usuario requeriría:

1. **Almacenar datos del usuario en el nivel superior** (componente App)
2. **Pasarlos hacia abajo** a través de Header → UserMenu
3. **Pasarlos hacia abajo** a través de Main → Sidebar → QuickActions

4. **Pasarlos hacia abajo** a través de Main → Content → Dashboard
5. **Pasar funciones de actualización hacia arriba** desde ProfileForm a través de Content → Main → App
6. **Pasar datos hacia abajo** a Footer → UserInfo

Esto crea una red de props que no tienen nada que ver con los componentes intermedios—son solo mensajeros de datos. Es frágil, verboso y hace que refactorizar sea difícil.

El Problema de la Cadena de Callbacks

Podrías pensar, "¡Solo usaré callback props!" Pero a través de largas distancias este enfoque tiene sus propios problemas:

```
// En ProfileForm, profundo en el árbol de componentes
this.props.onUserUpdated(newUserData);

// Cada componente en la cadena necesita recibir el callback y pasarlo hacia abajo
// App -> Main -> Content -> ProfilePage -> ProfileForm...
// y los datos actualizados aún tienen que viajar de vuelta hacia UserMenu, QuickActions,
    ↪ etc.
```

Esto crea acoplamiento fuerte entre componentes que no deberían conocerse entre sí, y es fácil romper la cadena.

Cuándo Necesitas Estado Global

Deberías considerar gestión de estado global cuando:

- **Preocupaciones transversales**: Autenticación de usuario, configuraciones de tema, preferencias de idioma
- **Datos compartidos**: Información del usuario actual, contenidos del carrito de compras, conteo de notificaciones
- **Caché de datos remotos**: Respuestas de API que múltiples componentes necesitan
- **Configuraciones de aplicación**: Feature flags, configuración, permisos
- **Actualizaciones en tiempo real**: Datos de WebSocket que afectan múltiples componentes

---

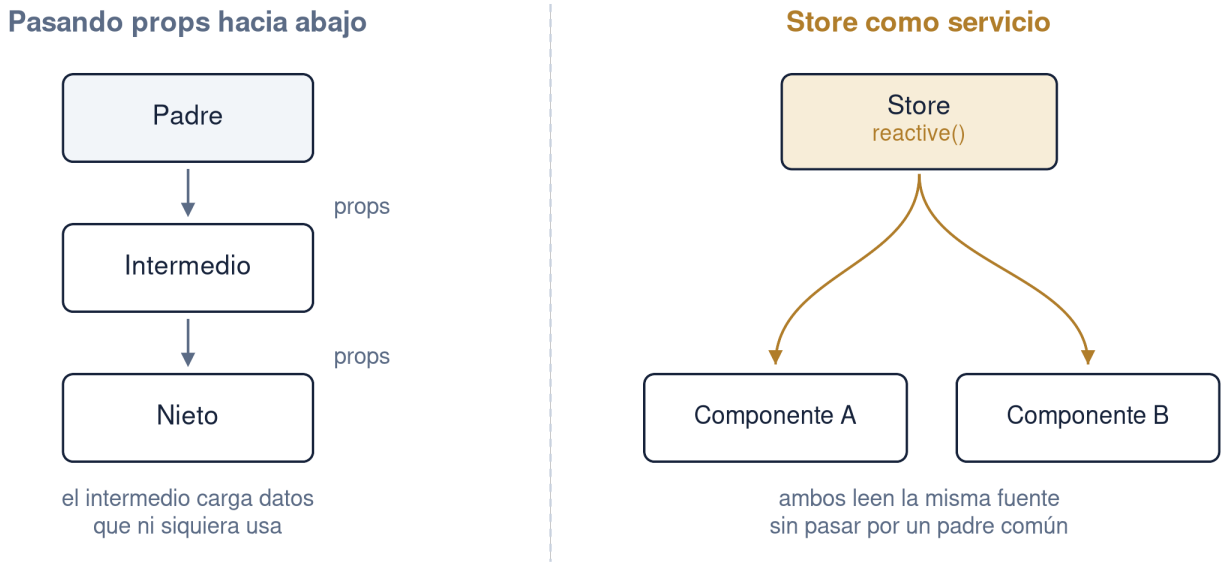


Figura 6: Pasar props hacia abajo frente a un store compartido como servicio.

Entendiendo los Stores en Odoo

En la arquitectura de Odoo, un **store** es un servicio especializado que gestiona estado reactivo. A diferencia de servicios regulares que proporcionan funcionalidad, los stores están específicamente diseñados para mantener datos a los que los componentes pueden suscribirse.

La Anatomía de un Store

Un store en Odoo típicamente tiene estas características:

1. **Estado Reactivo**: Usa la función `reactive` de OWL para hacer que los cambios de datos disparen actualizaciones de componentes
2. **Lógica Centralizada**: Todas las operaciones que modifican el estado están contenidas dentro del store
3. **Integración de Servicios**: Registrado como un servicio de Odoo, haciéndolo disponible en toda la aplicación
4. **Sistema de Eventos**: Puede emitir eventos cuando ocurren cambios significativos

Stores Integrados en Odoo

Antes de crear stores personalizados, es importante saber qué proporciona Odoo ya:

Servicio User: Información sobre el usuario actual

```
const user = useService("user");
console.log(user.name, user.isAdmin, user.partnerId);
```

Servicio Company: Información de la compañía actual

```
const company = useService("company");
console.log(company.currentCompany.name, company.allowedCompanies);
```

Servicio Notification: Para mostrar mensajes

```
const notification = useService("notification");
notification.add("¡Guardado exitoso!", { type: "success" });
```

Ahora vamos a crear nuestro propio store para entender el patrón completamente.

Creando Tu Primer Store: Gestión de Temas

Construyamos un store integral de temas que gestiona modo oscuro/claro, preferencias de tamaño de fuente, y personalización de colores a través de toda una aplicación Odoo.

1. Construyendo el Servicio Theme Store

Archivo: `mi_modulo/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() {
        // Cargar preferencias guardadas desde localStorage
        const preferenciasGuardadas = this.cargarPreferencias();

        // Crear estado reactivo
        this.state = reactive({
            modo: preferenciasGuardadas.modo || "claro",
            tamañoFuente: preferenciasGuardadas.tamañoFuente || "mediano",
            colorPrimario: preferenciasGuardadas.colorPrimario || "#007bff",
            radioBorde: preferenciasGuardadas.radioBorde || "mediano",
            animaciones: preferenciasGuardadas.animaciones !== false, // por defecto
                ↪ true
        });

        // Aplicar tema inmediatamente cuando se crea el store
        this.aplicarTema();
    }

    /**
     * Cargar preferencias desde localStorage
     */
    cargarPreferencias() {
        try {
```

```
            const guardado = browser.localStorage.getItem('odoo_preferencias_tema');
            return guardado ? JSON.parse(guardado) : {};
        } catch (error) {
            console.warn("Falló la carga de preferencias de tema:", error);
            return {};
        }
    }

    /**
     * Guardar preferencias actuales en localStorage
     */
    guardarPreferencias() {
        try {
            const preferencias = {
                modo: this.state.modo,
                tamañoFuente: this.state.tamañoFuente,
                colorPrimario: this.state.colorPrimario,
                radioBorde: this.state.radioBorde,
                animaciones: this.state.animaciones,
            };
            browser.localStorage.setItem('odoo_preferencias_tema',
    ↪  JSON.stringify(preferencias));
        } catch (error) {
            console.warn("Falló el guardado de preferencias de tema:", error);
        }
    }

    /**
     * Alternar entre modo claro y oscuro
     */
    alternarModo() {
        this.state.modo = this.state.modo === "claro" ? "oscuro" : "claro";
        this.aplicarTema();
        this.guardarPreferencias();
    }

    /**
     * Establecer un modo de tema específico
     */
    establecerModo(modo) {
        if (["claro", "oscuro", "auto"].includes(modo)) {
            this.state.modo = modo;
            this.aplicarTema();
            this.guardarPreferencias();
        }
    }

    /**
     * Establecer tamaño de fuente
     */
    establecerTamañoFuente(tamaño) {
        if (["pequeño", "mediano", "grande", "xl"].includes(tamaño)) {
            this.state.tamañoFuente = tamaño;
            this.aplicarTema();
            this.guardarPreferencias();
```

```
        }
    }

    /**
     * Establecer color primario
     */
    establecerColorPrimario(color) {
        // Validar formato de color (hex)
        if (/^#[0-9A-F]{6}$/i.test(color)) {
            this.state.colorPrimario = color;
            this.aplicarTema();
            this.guardarPreferencias();
        }
    }

    /**
     * Establecer preferencia de radio de borde
     */
    establecerRadioBorde(radio) {
        if (["ninguno", "pequeño", "mediano", "grande"].includes(radio)) {
            this.state.radioBorde = radio;
            this.aplicarTema();
            this.guardarPreferencias();
        }
    }

    /**
     * Alternar animaciones encendido/apagado
     */
    alternarAnimaciones() {
        this.state.animaciones = !this.state.animaciones;
        this.aplicarTema();
        this.guardarPreferencias();
    }

    /**
     * Restablecer al tema por defecto
     */
    restablecerADefecto() {
        this.state.modo = "claro";
        this.state.tamañoFuente = "mediano";
        this.state.colorPrimario = "#007bff";
        this.state.radioBorde = "mediano";
        this.state.animaciones = true;

        this.aplicarTema();
        this.guardarPreferencias();
    }

    /**
     * Aplicar tema actual al documento
     */
    aplicarTema() {
        const root = document.documentElement;
```

```
        // Aplicar propiedades personalizadas de CSS
        root.style.setProperty('--color-primario', this.state.colorPrimario);

        // Mapeo de tamaños de fuente
        const tamañosFuente = {
            pequeño: '14px',
            mediano: '16px',
            grande: '18px',
            xl: '20px'
        };
        root.style.setProperty('--tamaño-fuente-base',
            ↪ tamañosFuente[this.state.tamañoFuente]);

        // Mapeo de radios de borde
        const radiosBorde = {
            ninguno: '0',
            pequeño: '4px',
            mediano: '8px',
            grande: '16px'
        };
        root.style.setProperty('--radio-borde', radiosBorde[this.state.radioBorde]);

        // Aplicar clase de modo de tema
        root.className = root.className.replace(/tema-\w+/g, '');
        root.classList.add(`tema-${this.state.modo}`);

        // Manejar animaciones
        if (!this.state.animaciones) {
            root.classList.add('sin-animaciones');
        } else {
            root.classList.remove('sin-animaciones');
        }

        // Modo auto: detectar preferencia del sistema
        if (this.state.modo === 'auto') {
            const prefiereOscuro = window.matchMedia('(prefers-color-scheme:
                ↪ dark)').matches;
            root.classList.add(`tema-${prefiereOscuro ? 'oscuro' : 'claro'}`);
        }
    }

    /**
     * Obtener valores computados del tema
     */
    get temaComputado() {
        return {
            esOscuro: this.state.modo === 'oscuro' ||
                      (this.state.modo === 'auto' &&
                       window.matchMedia('(prefers-color-scheme: dark)').matches),
            esClaro: this.state.modo === 'claro' ||
                     (this.state.modo === 'auto' &&
                      !window.matchMedia('(prefers-color-scheme: dark)').matches),
            ...this.state
        };
    }
```

```
}

// Registrar como servicio de Odoo
export const themeStoreService = {
    dependencies: [],
    start(env) {
        const store = new ThemeStore();

        // Escuchar cambios de tema del sistema cuando esté en modo auto
        window.matchMedia('(prefers-color-scheme: dark)').addEventListener('change', ()
            ↪ => {
            if (store.state.modo === 'auto') {
                store.aplicarTema();
            }
        });

        return store;
    },
};

registry.category("services").add("themeStore", themeStoreService);
```

2. Accediendo al Store con useService

Ahora cualquier componente puede acceder a este theme store:

Componente: `theme_display.js`

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

import { Component, xml } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ThemeDisplay extends Component {
    static template = xml`
        <div class="visualizador-tema">
            <h3>Tema Actual</h3>
            <div class="info-tema">
                <p><strong>Modo:</strong> <span t-esc="themeStore.state.modo"/></p>
                <p><strong>Tamaño de Fuente:</strong>
                   <span t-esc="themeStore.state.tamañoFuente"/></p>
                <p><strong>Color Primario:</strong>
                   <span t-esc="themeStore.state.colorPrimario"/>
                   <div class="vista-previa-color"
                        t-att-style="\`background-color:
    ↪ \${themeStore.state.colorPrimario}\`"/>
                </p>
                <p><strong>Radio de Borde:</strong>
                   <span t-esc="themeStore.state.radioBorde"/></p>
                <p><strong>Animaciones:</strong>
                   <span t-esc="themeStore.state.animaciones
                               ? 'Habilitadas' : 'Deshabilitadas'"/></p>
            </div>
        </div>
    `;
```

```
    setup() {
        this.themeStore = useService("themeStore");
    }
}
```

Componente: `theme_controls.js`

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

import { Component, xml } from "@odoo/owl";
import { useService } from "@web/core/utils/hooks";

export class ThemeControls extends Component {
    static template = xml`
        <div class="controles-tema">
            <h3>Configuración de Tema</h3>

            <!-- Selección de Modo -->
            <div class="grupo-control">
                <label>Modo de Tema:</label>
                <select t-model="themeStore.state.modo" t-on-change="alCambiarModo">
                    <option value="claro">Claro</option>
                    <option value="oscuro">Oscuro</option>
                    <option value="auto">Auto (Sistema)</option>
                </select>
            </div>

            <!-- Tamaño de Fuente -->
            <div class="grupo-control">
                <label>Tamaño de Fuente:</label>
                <select t-model="themeStore.state.tamañoFuente"
                        t-on-change="alCambiarTamañoFuente">
                    <option value="pequeño">Pequeño</option>
                    <option value="mediano">Mediano</option>
                    <option value="grande">Grande</option>
                    <option value="xl">Extra Grande</option>
                </select>
            </div>

            <!-- Color Primario -->
            <div class="grupo-control">
                <label>Color Primario:</label>
                <input type="color"
                       t-att-value="themeStore.state.colorPrimario"
                       t-on-change="alCambiarColor"/>
            </div>

            <!-- Radio de Borde -->
            <div class="grupo-control">
                <label>Radio de Borde:</label>
                <select t-model="themeStore.state.radioBorde"
    ↪  t-on-change="alCambiarRadio">
                    <option value="ninguno">Ninguno</option>
                    <option value="pequeño">Pequeño</option>
                    <option value="mediano">Mediano</option>
                    <option value="grande">Grande</option>
```

```
                </select>
            </div>

            <!-- Alternar Animaciones -->
            <div class="grupo-control">
                <label>
                    <input type="checkbox"
                           t-att-checked="themeStore.state.animaciones"
                           t-on-change="alAlternarAnimacion"/>
                    Habilitar Animaciones
                </label>
            </div>

            <!-- Acciones Rápidas -->
            <div class="acciones-rapidas">
                <button class="btn btn-secondary" t-on-click="themeStore.alternarModo">
                    Alternar Modo Rápido
                </button>
                <button class="btn btn-outline-secondary"
                        t-on-click="themeStore.restablecerADefecto">
                    Restablecer a Defecto
                </button>
            </div>
        </div>
    `;

    setup() {
        this.themeStore = useService("themeStore");
    }

    alCambiarModo(event) {
        this.themeStore.establecerModo(event.target.value);
    }

    alCambiarTamañoFuente(event) {
        this.themeStore.establecerTamañoFuente(event.target.value);
    }

    alCambiarColor(event) {
        this.themeStore.establecerColorPrimario(event.target.value);
    }

    alCambiarRadio(event) {
        this.themeStore.establecerRadioBorde(event.target.value);
    }

    alAlternarAnimacion(event) {
        this.themeStore.alternarAnimaciones();
    }
}
```

3. Integración CSS

Agrega CSS que responda a las variables de tu tema:

Archivo: `mi_modulo/static/src/css/tema.css`

```css
/* Propiedades Personalizadas de CSS que el store controla */
:root {
    --color-primario: #007bff;
    --tamaño-fuente-base: 16px;
    --radio-borde: 8px;
}

/* Tipografía base */
body {
    font-size: var(--tamaño-fuente-base);
    transition: font-size 0.3s ease;
}

/* Estilos de modo de tema */
.tema-claro {
    --color-fondo: #ffffff;
    --color-texto: #333333;
    --color-borde: #dee2e6;
}

.tema-oscuro {
    --color-fondo: #1a1a1a;
    --color-texto: #ffffff;
    --color-borde: #444444;
}

body {
    background-color: var(--color-fondo);
    color: var(--color-texto);
    border-color: var(--color-borde);
}

/* Estilos de componentes que usan variables de tema */
.btn-primary {
    background-color: var(--color-primario);
    border-radius: var(--radio-borde);
}

.card {
    border-radius: var(--radio-borde);
    border-color: var(--color-borde);
}

/* Deshabilitar animaciones cuando se solicite */
.sin-animaciones * {
    animation-duration: 0s !important;
    transition-duration: 0s !important;
}

/* Estilo de controles de tema */
.controles-tema .grupo-control {
    margin-bottom: 1rem;
}

.controles-tema label {
```

```
    display: block;
    margin-bottom: 0.5rem;
    font-weight: 500;
}

.vista-previa-color {
    width: 20px;
    height: 20px;
    border-radius: var(--radio-borde);
    display: inline-block;
    margin-left: 8px;
    border: 1px solid var(--color-borde);
}
```

Patrones Avanzados de Store

1. Store con Operaciones Asíncronas

Muchos stores necesitan interactuar con APIs. Aquí hay un patrón para un store de carrito de compras:

```
export class ShoppingCartStore {
    constructor(orm, notification) {
        this.orm = orm;
        this.notification = notification;

        this.state = reactive({
            elementos: [],
            cargando: false,
            total: 0,
        });

        this.cargarCarrito();
    }

    async cargarCarrito() {
        this.state.cargando = true;
        try {
            const datosCarrito = await this.orm.call("sale.order", "get_current_cart",
                ↪ []);
            this.state.elementos = datosCarrito.elementos;
            this.state.total = datosCarrito.total;
        } catch (error) {
            this.notification.add("Falló la carga del carrito", { type: "danger" });
        } finally {
            this.state.cargando = false;
        }
    }

    async agregarElemento(idProducto, cantidad = 1) {
        this.state.cargando = true;
        try {
            await this.orm.call("sale.order", "add_to_cart", [idProducto, cantidad]);
            await this.cargarCarrito(); // Refrescar datos del carrito
```

```
            this.notification.add("Elemento agregado al carrito", { type: "success" });
        } catch (error) {
            this.notification.add("Falló agregar elemento", { type: "danger" });
        } finally {
            this.state.cargando = false;
        }
    }

    async removerElemento(idElemento) {
        this.state.cargando = true;
        try {
            await this.orm.call("sale.order", "remove_from_cart", [idElemento]);
            await this.cargarCarrito();
            this.notification.add("Elemento removido", { type: "info" });
        } catch (error) {
            this.notification.add("Falló remover elemento", { type: "danger" });
        } finally {
            this.state.cargando = false;
        }
    }

    get conteoElementos() {
        return this.state.elementos.reduce((total, elemento) => total +
            ↪  elemento.cantidad, 0);
    }

    get estaVacio() {
        return this.state.elementos.length === 0;
    }
}

export const shoppingCartService = {
    dependencies: ["orm", "notification"],
    start(env, { orm, notification }) {
        return new ShoppingCartStore(orm, notification);
    },
};
```

2. Store con Propiedades Computadas

Para estado derivado complejo, usa getters:

```
export class NotificationStore {
    constructor() {
        this.state = reactive({
            notificaciones: [],
            configuraciones: {
                email: true,
                push: true,
                sms: false,
            }
        });
    }

    get conteoNoLeidas() {
        return this.state.notificaciones.filter(n => !n.leida).length;
```

```
    }

    get notificacionesUrgentes() {
        return this.state.notificaciones.filter(n => n.prioridad === 'urgente' &&
            ↪  !n.leida);
    }

    get tieneNotificacionesUrgentes() {
        return this.notificacionesUrgentes.length > 0;
    }

    get notificacionesPorTipo() {
        return this.state.notificaciones.reduce((acc, notificacion) => {
            if (!acc[notificacion.tipo]) {
                acc[notificacion.tipo] = [];
            }
            acc[notificacion.tipo].push(notificacion);
            return acc;
        }, {});
    }
}
```

3. Store con Emisión de Eventos

Para aplicaciones complejas, los stores pueden emitir eventos:

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

export class UserActivityStore extends EventBus {
    constructor() {
        super();
        this.state = reactive({
            estaEnLinea: navigator.onLine,
            ultimaActividad: Date.now(),
            tiempoInactivo: 0,
        });

        this.configurarRastreoActividad();
    }

    configurarRastreoActividad() {
        // Rastrear estado online/offline
        window.addEventListener('online', () => {
            this.state.estaEnLinea = true;
            this.trigger('conectividad-cambio', { enLinea: true });
        });

        window.addEventListener('offline', () => {
            this.state.estaEnLinea = false;
            this.trigger('conectividad-cambio', { enLinea: false });
        });

        // Rastrear actividad del usuario
        const reiniciarActividad = () => {
            this.state.ultimaActividad = Date.now();
            this.state.tiempoInactivo = 0;
```

```
        };

        ['mousedown', 'mousemove', 'keypress', 'scroll', 'touchstart'].forEach(evento =>
    ↪ {
            document.addEventListener(evento, reiniciarActividad, true);
        });

        // Verificar usuarios inactivos cada minuto
        setInterval(() => {
            this.state.tiempoInactivo = Date.now() - this.state.ultimaActividad;

            if (this.state.tiempoInactivo > 5 * 60 * 1000) { // 5 minutos
                this.trigger('usuario-inactivo', { tiempoInactivo:
                    ↪ this.state.tiempoInactivo });
            }
        }, 60000);
    }
}
```

Mejores Prácticas para Gestión de Store

1. Mantén los Stores Enfocados

Cada store debe tener una sola responsabilidad clara:

```
// Bueno: Stores enfocados
- UserStore: Información del usuario actual y autenticación
- ThemeStore: Preferencias de UI y tematización
- CartStore: Estado y operaciones del carrito de compras
- NotificationStore: Notificaciones dentro de la app

// Malo: Store objeto dios
- AppStore: Todo mezclado junto
```

2. Usa Getters para Valores Computados

No almacenes valores derivados en estado—calcúlalos con getters:

```
// Bueno: Propiedad computada
get precioTotal() {
    return this.state.elementos.reduce(
        (suma, elemento) => suma + elemento.precio * elemento.cantidad, 0);
}

// Malo: Valor derivado almacenado (puede desincronizarse)
this.state.precioTotal = /* valor computado */;
```

3. Haz los Cambios de Estado Explícitos

Siempre proporciona métodos para modificar estado, no permitas mutaciones directas:

```
// Bueno: Mutaciones controladas
agregarNotificacion(mensaje, tipo = 'info') {
    this.state.notificaciones.push({
```

```
        id: Date.now(),
        mensaje,
        tipo,
        marcaTiempo: new Date(),
        leida: false
    });
}

// Malo: Mutación directa de estado desde componentes
// notificationStore.state.notificaciones.push(...);
```

4. Maneja Estados de Carga

Para operaciones asíncronas, siempre proporciona indicadores de carga:

```
async cargarDatosUsuario() {
    this.state.cargando = true;
    this.state.error = null;

    try {
        const datosUsuario = await this.orm.call("res.users", "get_current_user", []);
        this.state.usuario = datosUsuario;
    } catch (error) {
        this.state.error = error.message;
    } finally {
        this.state.cargando = false;
    }
}
```

5. Proporciona APIs Claras

Documenta la interfaz pública de tu store:

```
/**
 * Theme Store
 *
 * Estado:
 * - modo: 'claro' | 'oscuro' | 'auto'
 * - tamañoFuente: 'pequeño' | 'mediano' | 'grande' | 'xl'
 * - colorPrimario: string de color hex
 *
 * Métodos:
 * - alternarModo(): Alternar entre claro/oscuro
 * - establecerModo(modo): Establecer modo específico
 * - establecerTamañoFuente(tamaño): Establecer tamaño de fuente
 * - establecerColorPrimario(color): Establecer color primario
 * - restablecerADefecto(): Restablecer todas las configuraciones
 *
 * Computado:
 * - temaComputado: Objeto con valores de tema resueltos
 */
```

Cuándo NO Usar Estado Global

El estado global es poderoso, pero no siempre es la solución correcta:

Usa Estado Local Cuando:

- Los datos solo son usados por un componente y sus hijos
- Los datos no necesitan persistir entre montajes/desmontajes de componentes
- La relación entre componentes es clara y directa

Usa Props/Eventos Cuando:

- Los componentes tienen una relación padre-hijo clara
- El flujo de datos es simple y unidireccional
- Quieres mantener componentes débilmente acoplados

Usa Estado Global Cuando:

- Múltiples componentes no relacionados necesitan los mismos datos
- Los datos necesitan persistir a través de la navegación
- Estás lidiando con preferencias de usuario o configuraciones de aplicación
- Necesitas coordinar estado a través de diferentes partes de la app

---

Debuggeando Problemas de Store

1. Agrega Logging a Métodos de Store

```
establecerModo(modo) {
    console.log(`Modo de tema cambiando de ${this.state.modo} a ${modo}`);
    this.state.modo = modo;
    this.aplicarTema();
    this.guardarPreferencias();
}
```

2. Usa Browser DevTools

Los stores son objetos JavaScript regulares, así que puedes inspeccionarlos en la consola:

```
// En consola del navegador
const themeStore = odoo.__SERVICES__.themeStore;
console.log(themeStore.state);
```

3. Agrega Validación

```
establecerTamañoFuente(tamaño) {
    const tamañosValidos = ["pequeño", "mediano", "grande", "xl"];
    if (!tamañosValidos.includes(tamaño)) {
        console.error(
            `Tamaño de fuente inválido: ${tamaño}.` +
            ` Opciones válidas: ${tamañosValidos.join(', ')}`);
        return;
    }
```

```
    this.state.tamañoFuente = tamaño;
    this.aplicarTema();
    this.guardarPreferencias();
}
```

Ejemplo de Integración del Mundo Real

Aquí hay un ejemplo completo mostrando cómo múltiples servicios trabajan juntos en un escenario de producción:

```
export class DashboardStore {
    constructor(orm, notification, user) {
        this.orm = orm;
        this.notification = notification;
        this.user = user;

        this.state = reactive({
            // Datos del dashboard
            metricas: {
                ventasTotales: 0,
                clientesActivos: 0,
                pedidosPendientes: 0,
                ingresosMensuales: 0
            },

            // Estados de carga
            cargandoMetricas: false,
            cargandoGraficos: false,

            // Configuración del dashboard
            layoutConfigurado: this.user.dashboardLayout || 'grid',
            widgetsVisibles: this.user.visibleWidgets || ['ventas', 'clientes',
                ↪  'pedidos'],

            // Datos de gráficos
            datosVentas: [],
            datosClientes: [],

            // Filtros
            rangoFecha: 'ultimo-mes',
            equipoSeleccionado: null,

            // Errores
            errores: {}
        });

        // Cargar datos iniciales
        this.inicializar();
    }

    async inicializar() {
        try {
```

```
            await Promise.all([
                this.cargarMetricas(),
                this.cargarDatosGraficos()
            ]);

            this.notification.add("Dashboard cargado exitosamente", {
                type: "success"
            });
        } catch (error) {
            this.notification.add("Error al cargar dashboard", {
                type: "danger",
                sticky: true
            });

            console.error("Error de inicialización de dashboard:", error);
        }
    }
```

El constructor arma la forma reactiva de `state` y de inmediato dispara `inicializar()`, que carga todo lo que necesita el dashboard en paralelo y muestra una notificación en ambos casos. A continuación vienen los dos métodos de carga que `inicializar()` invoca — cada uno sigue el mismo patrón de bandera `cargando...` y `try/catch/finally` que ya viste antes en este capítulo:

```
    async cargarMetricas() {
        this.state.cargandoMetricas = true;
        this.state.errores.metricas = null;

        try {
            const metricas = await this.orm.call(
                "dashboard.analytics",
                "get_metrics",
                [],
                {
                    fecha_desde: this.obtenerFechaDesde(),
                    fecha_hasta: this.obtenerFechaHasta(),
                    equipo_id: this.state.equipoSeleccionado
                }
            );

            this.state.metricas = {
                ventasTotales: metricas.total_sales,
                clientesActivos: metricas.active_customers,
                pedidosPendientes: metricas.pending_orders,
                ingresosMensuales: metricas.monthly_revenue
            };
        } catch (error) {
            this.state.errores.metricas = error.message;
            console.error("Error al cargar métricas:", error);
        } finally {
            this.state.cargandoMetricas = false;
        }
    }

    async cargarDatosGraficos() {
        this.state.cargandoGraficos = true;
        this.state.errores.graficos = null;
```

```
        try {
            const [datosVentas, datosClientes] = await Promise.all([
                this.orm.call("dashboard.analytics", "get_sales_chart_data", [], {
                    periodo: this.state.rangoFecha
                }),
                this.orm.call("dashboard.analytics", "get_customer_chart_data", [], {
                    periodo: this.state.rangoFecha
                })
            ]);

            this.state.datosVentas = datosVentas;
            this.state.datosClientes = datosClientes;
        } catch (error) {
            this.state.errores.graficos = error.message;
            console.error("Error al cargar datos de gráficos:", error);
        } finally {
            this.state.cargandoGraficos = false;
        }
    }
```

Con la carga resuelta, el resto del store es la API pública que realmente usan otros componentes: métodos pequeños que cambian una sola parte del estado y, cuando hace falta, disparan una recarga o un guardado. Fíjate que ninguno de estos métodos accede a `this.state` desde afuera del store — cada mutación pasa por un método con nombre, que es justamente lo que mantiene depurable un store que va creciendo:

```
    // Métodos para cambiar configuración
    cambiarRangoFecha(nuevoRango) {
        if (this.state.rangoFecha !== nuevoRango) {
            this.state.rangoFecha = nuevoRango;
            this.actualizarDatos();
        }
    }

    seleccionarEquipo(equipoId) {
        if (this.state.equipoSeleccionado !== equipoId) {
            this.state.equipoSeleccionado = equipoId;
            this.actualizarDatos();
        }
    }

    alternarWidget(widgetId) {
        const widgets = [...this.state.widgetsVisibles];
        const indice = widgets.indexOf(widgetId);

        if (indice > -1) {
            widgets.splice(indice, 1);
        } else {
            widgets.push(widgetId);
        }

        this.state.widgetsVisibles = widgets;
        this.guardarConfiguracionUsuario();
    }
```

```
    cambiarLayout(nuevoLayout) {
        this.state.layoutConfigurado = nuevoLayout;
        this.guardarConfiguracionUsuario();
    }

    async actualizarDatos() {
        await Promise.all([
            this.cargarMetricas(),
            this.cargarDatosGraficos()
        ]);
    }

    async guardarConfiguracionUsuario() {
        try {
            await this.orm.call("res.users", "save_dashboard_config", [], {
                layout: this.state.layoutConfigurado,
                visible_widgets: this.state.widgetsVisibles
            });

            this.notification.add("Configuración guardada", { type: "success" });
        } catch (error) {
            this.notification.add("Error al guardar configuración", { type: "warning" });
        }
    }
```

Finalmente, un par de métodos auxiliares simples y tres getters computados exponen datos derivados (¿algo está cargando? ¿hay errores? ¿cuál es la configuración completa actual?) sin duplicarlos en `state`, seguidos del registro del servicio que convierte esta clase en un **singleton** — una única instancia compartida, creada una vez y reutilizada en todas partes — a la que cada componente accede vía `useService("dashboardStore")`:

```
    obtenerFechaDesde() {
        const ahora = new Date();
        switch (this.state.rangoFecha) {
            case 'ultima-semana':
                return new Date(ahora.getTime() - 7 * 24 * 60 * 60 * 1000);
            case 'ultimo-mes':
                return new Date(ahora.getFullYear(), ahora.getMonth() - 1,
                    ↪ ahora.getDate());
            case 'ultimo-trimestre':
                return new Date(ahora.getFullYear(), ahora.getMonth() - 3,
                    ↪ ahora.getDate());
            case 'ultimo-año':
                return new Date(ahora.getFullYear() - 1, ahora.getMonth(),
                    ↪ ahora.getDate());
            default:
                return new Date(ahora.getFullYear(), ahora.getMonth() - 1,
                    ↪ ahora.getDate());
        }
    }

    obtenerFechaHasta() {
        return new Date();
    }
```

```
    // Getters computados
    get tieneErrores() {
        return Object.keys(this.state.errores).some(key => this.state.errores[key]);
    }

    get estaCargando() {
        return this.state.cargandoMetricas || this.state.cargandoGraficos;
    }

    get configuracionCompleta() {
        return {
            layout: this.state.layoutConfigurado,
            widgets: this.state.widgetsVisibles,
            filtros: {
                rangoFecha: this.state.rangoFecha,
                equipo: this.state.equipoSeleccionado
            }
        };
    }
}

// Servicio del Dashboard Store
export const dashboardStoreService = {
    dependencies: ["orm", "notification", "user"],
    start(env, { orm, notification, user }) {
        return new DashboardStore(orm, notification, user);
    },
};

registry.category("services").add("dashboardStore", dashboardStoreService);
```

Errores Comunes

1. Olvidar suscribirse con `useState`

Crear el store con `reactive({...})` solo hace reactivos los datos en la *fuente*. Un componente que lee el store directamente (`useService("dashboardStore").state.metricas`) sin envolverlo en `useState` no se va a re-renderizar cuando esos datos cambien:

```
// MAL: lee el estado reactivo una vez, el componente nunca se re-renderiza
this.store = useService("dashboardStore");

// BIEN: suscribe este componente a la reactividad del store
this.store = useState(useService("dashboardStore").state);
```

2. Mutar el store desde cualquier lugar

Si cualquier componente puede hacer `this.store.state.equipoSeleccionado = 5` directamente, pierdes el rastro de quién cambia qué y por qué. Mantén cada mutación detrás de un método con nombre en el store (`seleccionarEquipo(id)`), incluso cuando el método sea de una sola línea — te da un único lugar donde agregar logging, validación o un efecto secundario más adelante.

3. Una instancia de store por componente

Registrar el store como servicio (como se mostró arriba) garantiza una única instancia compartida. Instanciar `new DashboardStore(...)` directamente dentro del `setup()` de un componente crea una copia privada que ningún otro componente puede ver — anulando todo el propósito del estado global.

4. Recurrir al estado global demasiado pronto

No todo necesita vivir en un store. Si solo un componente (y sus hijos directos, vía props) lee un dato, un `useState` local es más simple y fácil de razonar. Promueve el estado a un store solo cuando dos o más componentes sin relación directa realmente necesitan compartirlo.

La gestión de estado global con stores es esencial para construir aplicaciones Odoo complejas e interconectadas. Al centralizar el estado compartido y proporcionar APIs claras para acceder y modificarlo, creas aplicaciones que son más mantenibles, más predecibles y más fáciles de debuggear.

La clave es saber cuándo usar estado global versus estado local, y diseñar tus stores con responsabilidades claras e interfaces limpias. Domina este patrón, y podrás construir aplicaciones sofisticadas que escalen graciosamente a medida que crecen en complejidad.

En el próximo capítulo, exploraremos cómo asegurar que tus componentes funcionen correctamente a través de pruebas y técnicas efectivas de debugging.

TL;DR: Registra un store `reactive()` como servicio y consúmelo con `useState(useService(...).state)` donde haga falta, así componentes sin relación pueden compartir estado reactivo sin prop drilling.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch15_ex1`. Las instrucciones de instalación están en el README del repositorio.

Ejercicios

1. **Store de contador compartido.** Crea un servicio `counterStore` usando `reactive({ count: 0 })` con métodos `increment()` y `decrement()`. Consúmelo desde dos componentes hermanos con `useState(useService("counterStore").state)` y confirma que al hacer clic en un botón de uno se actualiza el contador mostrado en el otro.
2. **Agrega un getter computado.** Extiende el store del Ejercicio 1 con un getter `get isPositive()`, y úsalo en un template para mostrar condicionalmente una advertencia cuando el contador se vuelva negativo.
3. **Local vs. global.** Toma un componente que hayas construido en los ejercicios de un capítulo anterior (o el `DataTable` del Capítulo 14) y decide: ¿alguno de sus estados realmente necesita ser global? Escribe una oración justificando tu respuesta antes de escribir código.

¿Qué Sigue?

Que el estado sea correcto no es lo mismo que el estado en el que puedas confiar frente a un cliente — el Capítulo 16 Parte 1 cubre cómo probar y depurar componentes OWL para detectar regresiones antes de que ellos lo hagan.

Capítulo 16 Parte 1: Fundamentos de Pruebas y Depuración

> **¿Por qué este capítulo?** "Funciona en mi máquina" no alcanza cuando eres tú quien responde por la instancia de producción de un cliente — las pruebas son lo que te permite tocar código con confianza después de la primera entrega, en vez de temerle a cada cambio.
>
> **¿Qué trata de resolver Odoo con esto?** Un framework de pruebas JS que pueda levantar un entorno Odoo simulado con los servicios correctos rápidamente, así probar un addon pequeño no requiere un servidor y una base de datos completos corriendo.
>
> **Aplicación en la vida real:** Detectar una regresión en una revisión de código, antes de que una actualización de rutina rompa un dashboard que el equipo de ventas de un cliente usa todas las mañanas.

Escribir código que funciona hoy es bueno. Escribir código que puedas modificar con confianza mañana sin romper todo es excepcional. La diferencia radica en tener una estrategia integral de pruebas y dominar las técnicas de depuración.

Las pruebas no son solo para encontrar errores: se tratan de diseñar mejores componentes, documentar el comportamiento esperado y crear una red de seguridad que te permita refactorizar y mejorar tu código con confianza. Cuando se combinan con técnicas efectivas de depuración, las pruebas transforman el desarrollo de una serie de experimentos esperanzadores en un proceso metódico y predecible.

Odoo proporciona un poderoso framework de pruebas JavaScript construido sobre `@odoo/hoot` (el runner de pruebas unitarias moderno, usado desde Odoo 17.4, que reemplazó al framework anterior basado en QUnit), específicamente diseñado para probar componentes OWL en condiciones realistas. Esta parte del capítulo te lleva desde escribir tu primera prueba simple hasta probar componentes que dependen de servicios, además de las herramientas de depuración y las buenas prácticas que usarás día a día.

Por qué Importan las Pruebas en el Desarrollo OWL

Antes de sumergirnos en el "cómo", entendamos el "por qué" de probar componentes OWL.

La Realidad de la Interdependencia de Componentes

Los componentes OWL no existen de forma aislada. Ellos: - Reciben props de los padres y pasan props a los hijos - Emiten eventos que los padres escuchan - Usan servicios para interactuar con el backend de Odoo - Manipulan el DOM a través de plantillas - Administran el estado local que afecta el renderizado

Cada una de estas interacciones es un punto potencial de falla. Sin pruebas, básicamente estás volando a ciegas, haciendo cambios y esperando que nada se rompa.

El Costo de las Pruebas Manuales

Considera un flujo de trabajo típico de desarrollo sin pruebas automatizadas:

1. **Hacer un cambio** en un componente
2. **Iniciar el servidor de Odoo** (30-60 segundos)
3. **Navegar a la página** donde se usa el componente
4. **Interactuar manualmente** con el componente para verificar que funciona
5. **Verificar casos límite** creando manualmente diferentes escenarios
6. **Repetir** para cada componente que podría verse afectado

Para un solo cambio pequeño, este proceso podría tomar 5-10 minutos. Multiplica eso por docenas de cambios por día, y estás gastando horas en pruebas manuales repetitivas.

Los Beneficios de las Pruebas Automatizadas

Las pruebas automatizadas cambian esto completamente:

- **Velocidad**: Las pruebas se ejecutan en segundos, no en minutos
- **Consistencia**: Las pruebas verifican las mismas cosas cada vez, de la misma manera
- **Cobertura**: Las pruebas pueden verificar casos límite que podrías olvidar manualmente
- **Confianza**: Las pruebas verdes significan que tus cambios no han roto la funcionalidad existente
- **Documentación**: Las pruebas sirven como ejemplos ejecutables de cómo deben comportarse los componentes

---

Configurando tu Entorno de Pruebas

1. Organización de Archivos de Prueba

Odoo sigue una convención clara para organizar archivos de prueba:

```
mi_modulo/
|-- 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
```

Esta organización mantiene las pruebas cerca del código que están probando, haciéndolas fáciles de encontrar y mantener.

2. Estructura Básica de Archivo de Prueba

Cada archivo de prueba JavaScript de Odoo sigue este patrón:

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

import { describe, test, beforeEach } from "@odoo/hoot";
import { mountWithCleanup } from "@web/../tests/web_test_helpers";

// Importar el componente que estás probando
import { Counter } from "../counter";

describe("Counter", () => {
    beforeEach(() => {
        // Se ejecuta antes de cada prueba en este bloque describe()
    });

    test("renderizado básico", async () => {
        // La implementación de la prueba va aquí
    });
});
```

Helpers clave explicados: - **`describe()`**: Agrupa pruebas relacionadas, como una suite de pruebas - **`test()`**: Define un caso de prueba individual - **`beforeEach()`**: Ejecuta código de configuración antes de cada prueba dentro del `describe()` que la contiene - **`mountWithCleanup()`**: Monta un componente en la página de pruebas y lo desmonta automáticamente al terminar la prueba — sin necesidad de gestionar `target/env/.destroy()` a mano como con el framework antiguo basado en QUnit

3. Agregando Pruebas a tu Módulo

No olvides incluir tus archivos de prueba en `__manifest__.py`:

```
{
    'name': 'Mi Módulo',
    # ... otros datos del manifest ...
    'assets': {
        'web.assets_backend': [
            'mi_modulo/static/src/components/**/*.js',
            'mi_modulo/static/src/components/**/*.xml',
        ],
        'web.assets_unit_tests': [
            'mi_modulo/static/src/components/**/tests/*.test.js',
        ],
    },
}
```

Escribiendo tu Primera Prueba de Componente

Comencemos con un ejemplo simple pero realista: probar un componente `Counter`.

El Componente Counter

Primero, aquí está el componente que vamos a probar:

Archivo: `counter.js`

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

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

export class Counter extends Component {
    static template = xml`
        <div class="counter-widget">
            <h3>Contador: <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">
                    Reiniciar
                </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>Mín: <t t-esc="props.min"/>, Máx: <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);
        }
    }
}
```

Suite de Pruebas Integral

Ahora escribamos pruebas exhaustivas para este componente. Las construiremos de forma incremental, unas pocas a la vez, para que sea más fácil ver qué verifica cada grupo.

Archivo: `counter.test.js`

Primero, las pruebas de renderizado — verifican que el componente muestra lo correcto según distintas 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("renderiza con props por defecto", async () => {
        // Montar componente sin props (debería usar valores por defecto)
        await mountWithCleanup(Counter);

        // Verificar visualización inicial
        expect(".counter-value").toHaveText("0");

        // Verificar que los botones existen
        expect(".increment-btn").toHaveCount(1);
        expect(".decrement-btn").toHaveCount(1);
        expect(".reset-btn").toHaveCount(1);

        // Verificar que la info no se muestra (min/max por defecto)
        expect(".counter-info").toHaveCount(0);
```

```
    });

    test("renderiza con props personalizadas", async () => {
        await mountWithCleanup(Counter, {
            props: {
                initialValue: 5,
                min: 2,
                max: 8,
            },
        });

        // Verificar valor inicial personalizado
        expect(".counter-value").toHaveText("5");

        // Verificar que se muestra la info, con los valores min/max personalizados
        expect(".counter-info").toHaveCount(1);
        expect(".counter-info").toHaveText("Mín: 2, Máx: 8");
    });
});
```

Matchers clave explicados: - **`expect(selector).toHaveText(texto)`**: Afirma que el elemento que coincide con `selector` tiene exactamente ese texto - **`expect(selector).toHaveCount(n)`**: Afirma que exactamente `n` elementos coinciden con `selector` (usa `0` para afirmar que algo está ausente)

A continuación, las pruebas de interacción — hacer clic en los botones de incremento, decremento y reinicio, y verificar que se respetan los límites `min`/`max`. Se agregan dentro del mismo bloque `describe("Counter", ...)` mostrado arriba, y usan `click()` de `@odoo/hoot-dom`, que simula un clic real de usuario y espera a que OWL termine de re-renderizar antes de resolver:

```
    test("funcionalidad de incremento", async () => {
        await mountWithCleanup(Counter, {
            props: { initialValue: 0, max: 3 },
        });

        // Probar incremento normal
        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");

        // Probar que el botón se desactiva en el máximo
        expect(".increment-btn").toHaveProperty("disabled", true);

        // Hacer clic en un botón desactivado no hace nada
        await click(".increment-btn");
        expect(".counter-value").toHaveText("3");
    });

    test("funcionalidad de decremento", async () => {
        await mountWithCleanup(Counter, {
            props: { initialValue: 3, min: 1 },
```

```
    });

    // Probar decremento normal
    await click(".decrement-btn");
    expect(".counter-value").toHaveText("2");

    await click(".decrement-btn");
    expect(".counter-value").toHaveText("1");

    // Probar que el botón se desactiva en el mínimo
    expect(".decrement-btn").toHaveProperty("disabled", true);

    // Hacer clic en un botón desactivado no hace nada
    await click(".decrement-btn");
    expect(".counter-value").toHaveText("1");
});

test("funcionalidad de reinicio", async () => {
    await mountWithCleanup(Counter, {
        props: { initialValue: 5, min: 0, max: 10 },
    });

    // Cambiar el valor
    await click(".increment-btn");
    await click(".increment-btn");
    expect(".counter-value").toHaveText("7");

    // Probar reinicio
    await click(".reset-btn");
    expect(".counter-value").toHaveText("5");
});
```

Recuerda agregar `import { click } from "@odoo/hoot-dom";` junto al import de `@odoo/hoot` al principio del archivo.

Por último, las pruebas de callback y de casos límite — verifican que `onValueChange` se dispara con los valores correctos, y que combinaciones inusuales de props (como `min === max`) no rompen el componente. Mismo bloque `describe("Counter", ...)` de arriba, cerrado al final. Nota que cada escenario tiene su propio `test()` — siguiendo la práctica de "mantener las pruebas independientes" que se cubre más adelante en este capítulo, en vez de amontonar varios montajes en una sola prueba:

```
test("callback onValueChange", async () => {
    const valueChanges = [];
    const onValueChange = (newValue) => {
        valueChanges.push(newValue);
    };

    await mountWithCleanup(Counter, {
        props: {
            initialValue: 2,
            onValueChange,
        },
    });

    // Probar callback de incremento
    await click(".increment-btn");
```

```
        // Probar callback de decremento
        await click(".decrement-btn");
        // Probar callback de reinicio
        await click(".reset-btn");

        // incrementar a 3, decrementar a 2, reiniciar a 2
        expect(valueChanges).toEqual([3, 2, 2]);
    });

    test("incremento y decremento están desactivados cuando 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("el valor inicial fuera de límites se preserva", async () => {
        // Nota: En una implementación real, podrías querer limitar el valor inicial.
        // Esta prueba documenta el comportamiento actual.
        await mountWithCleanup(Counter, {
            props: { initialValue: 100, min: 1, max: 10 },
        });

        expect(".counter-value").toHaveText("100");
    });
});
```

Matchers clave explicados: - `expect(valor).toEqual(otro)`: Afirma igualdad profunda entre dos valores (arrays, objetos) — usa `.toBe()` en vez de esto para primitivos como strings, números y booleanos - `expect(selector).toHaveProperty(nombre, valor)`: Afirma que una propiedad del DOM (como `disabled`) del elemento coincidente es igual a `valor`

Probando Componentes con Servicios

Muchos componentes del mundo real dependen de servicios de Odoo. Así es como probarlos.

Componente que Usa Servicios

Este componente `TaskManager` carga tareas con el servicio `orm` al montarse, y muestra retroalimentación con el servicio `notification`. Veamos primero la mitad que carga los datos:

```
/** @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>Mis Tareas</h3>
            <div class="task-input">
                <input type="text"
```

```
                    t-model="state.newTaskText"
                    placeholder="Agregar una nueva tarea..."
                    t-on-keydown="onKeydown"/>
                <button class="btn btn-primary"
                        t-on-click="addTask"
                        t-att-disabled="!state.newTaskText">
                    Agregar Tarea
                </button>
            </div>

            <div class="task-list" t-if="state.loading">
                <p>Cargando tareas...</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)">
                        Eliminar
                    </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("Error al cargar tareas", { type: "danger" });
        } finally {
            this.state.loading = false;
        }
    }
```

Y el resto de la clase — agregar, marcar y eliminar tareas, más el atajo de la tecla Enter. Sigue siendo la misma clase `TaskManager`, continuada:

```
    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("Tarea agregada exitosamente", { type: "success" });
        } catch (error) {
            this.notification.add("Error al agregar tarea", { type: "danger" });
        }
    }

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

        try {
            // En una implementación real, actualizarías el stage_id
            task.completed = !task.completed;
            this.notification.add(
                task.completed ? "¡Tarea completada!" : "Tarea reabierta",
                { type: "info" }
            );
        } catch (error) {
            // Revertir en caso de error
            task.completed = !task.completed;
            this.notification.add("Error al actualizar tarea", { 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("Tarea eliminada", { type: "info" });
        } catch (error) {
            this.notification.add("Error al eliminar tarea", { type: "danger" });
        }
    }

    onKeydown(event) {
        if (event.key === "Enter") {
            this.addTask();
        }
    }
}
```

Probando con Servicios Mock

Para probar `TaskManager` sin un servidor Odoo real, hacemos mock de la capa RPC en lugar de golpear la base de datos, usando el helper `onRpc()` para interceptar llamadas ORM específicas. Primero, las pruebas que cubren cargar datos y agregar una nueva tarea:

```
/** @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("carga y muestra tareas", async () => {
        // Mock de search_read del ORM para el modelo project.task
        const mockTasks = [
            { id: 1, name: "Comprar comida", stage_id: [1, "Por Hacer"] },
            { id: 2, name: "Pasear al perro", stage_id: [2, "Completado"] },
            { id: 3, name: "Escribir pruebas", stage_id: [1, "Por Hacer"] },
        ];
        onRpc("project.task", "search_read", () => mockTasks);

        await mountWithCleanup(TaskManager);

        // Verificar que el estado de carga se fue
        expect(".task-list p").toHaveCount(0);

        // Verificar que las tareas se muestran
        expect(".task-item").toHaveCount(3);
        expect(queryAllTexts(".task-text")).toEqual([
            "Comprar comida",
            "Pasear al perro",
            "Escribir pruebas",
        ]);

        // Verificar estado completado (la segunda tarea tiene stage_id "Completado")
        expect(".task-item:nth-child(2)").toHaveClass("completed");
    });

    test("agrega una nueva tarea", async () => {
```

```
        let createdTask = null;
        onRpc("project.task", "search_read", () => []); // Comenzar con lista de tareas
            ↪ vacía
        onRpc("project.task", "create", ({ args }) => {
            createdTask = args[0]; // Capturar datos de la tarea creada
            return 123; // ID mock
        });

        await mountWithCleanup(TaskManager);

        // Escribir nueva tarea y hacer clic en agregar
        await edit(".task-input input", "Nueva tarea de prueba");
        await click(".btn-primary");

        // Verificar que se hizo la llamada ORM
        expect(createdTask).not.toBe(null);
        expect(createdTask.name).toBe("Nueva tarea de prueba");

        // Verificar que la tarea aparece en la UI
        expect(".task-item").toHaveCount(1);
        expect(queryAllTexts(".task-text")).toEqual(["Nueva tarea de prueba"]);

        // Verificar que el input se limpia
        expect(".task-input input").toHaveValue("");
    });
});
```

Helpers clave explicados: - **`onRpc(modelo, método, callback)`**: Intercepta una llamada ORM específica en vez de golpear el servidor real; el callback recibe `{ args, kwargs }` (los argumentos que el componente pasó a `orm.create`/`orm.write`/etc.) y su valor de retorno se convierte en el resultado del RPC - **`edit(selector, valor)`**: Escribe `valor` en el input coincidente y dispara los eventos que la escritura real de un usuario dispararía (de `@odoo/hoot-dom`) - **`queryAllTexts(selector)`**: Devuelve el texto recortado de cada elemento que coincide con `selector`, como un array — útil para comparar toda una lista de una sola vez

Las dos pruebas restantes verifican el manejo de errores y el atajo de teclado. Se agregan dentro del mismo bloque `describe("TaskManager", ...)` mostrado arriba, cerrado al final:

```
    test("maneja errores RPC elegantemente", async () => {
        const notificationMessages = [];
        onRpc("project.task", "search_read", () => {
            throw new Error("Error de red");
        });
        mockService("notification", {
            add: (message, options) => {
                notificationMessages.push({ message, options });
            },
        });

        await mountWithCleanup(TaskManager);

        // Verificar que se mostró la notificación de error
        expect(notificationMessages.length).toBe(1);
        expect(notificationMessages[0].message).toBe("Error al cargar tareas");
        expect(notificationMessages[0].options.type).toBe("danger");
```

```
        // Verificar que el estado de carga se limpia incluso en caso de error
        expect(".task-list p").toHaveCount(0);
    });

    test("los atajos de teclado funcionan", async () => {
        let taskCreated = false;
        onRpc("project.task", "search_read", () => []);
        onRpc("project.task", "create", () => {
            taskCreated = true;
            return 456;
        });

        await mountWithCleanup(TaskManager);

        // Escribir texto de la tarea y presionar Enter
        await edit(".task-input input", "Tarea por tecla Enter");
        await press("Enter");

        expect(taskCreated).toBe(true);
    });
});
```

Agrega `import { mockService } from "@web/../tests/web_test_helpers";` e `import { press } from "@odoo/hoot-dom";` junto a los demás imports al inicio del archivo. **`mockService()`** reemplaza un servicio real de Odoo por una implementación falsa durante la prueba — aquí sustituye `notification` para poder capturar los mensajes que recibe en vez de mostrar toasts reales. **`press(tecla)`** simula presionar una tecla del teclado sobre el elemento que tiene el foco actualmente.

Técnicas Efectivas de Depuración

Aunque las pruebas detectan muchos problemas, aún necesitarás depurar problemas. Aquí están las estrategias profesionales de depuración:

1. Usando las DevTools del Navegador Efectivamente

Estableciendo Breakpoints Estratégicos:

```
// En tu componente
increment() {
    debugger; // La ejecución se pausará aquí
    if (this.state.count < this.props.max) {
        this.state.count++;
        this._notifyChange();
    }
}
```

Breakpoints Condicionales: Haz clic derecho en un número de línea en la pestaña Sources de DevTools y selecciona “Agregar breakpoint condicional”:

```
this.state.count > 5  // Solo pausar cuando count exceda 5
```

Observando Variables: En el debugger de DevTools, agrega variables al panel “Watch” para ver cómo cambian mientras recorres el código.

2. Logging Estratégico en la Consola

Logging del Ciclo de Vida del Componente:

```
setup() {
    console.log("Configuración del Counter", this.props);
    this.state = useState({ count: this.props.initialValue });
}

increment() {
    console.log("Antes del incremento:", this.state.count);
    if (this.state.count < this.props.max) {
        this.state.count++;
        console.log("Después del incremento:", this.state.count);
        this._notifyChange();
    } else {
        console.log("Incremento bloqueado - en el máximo:", this.props.max);
    }
}
```

Seguimiento de Cambios de Estado:

```
setup() {
    this.state = useState({ count: this.props.initialValue });

    // Registrar todos los cambios de estado
    const originalState = this.state;
    Object.defineProperty(this, 'state', {
        get: () => originalState,
        set: (newState) => {
            console.log("Estado cambiando de", originalState, "a", newState);
            originalState = newState;
        }
    });
}
```

3. Herramientas de Inspección de Componentes

OWL DevTools: Instala la extensión de navegador OWL DevTools para: - Ver la estructura del árbol de componentes - Inspeccionar props y estado de componentes - Rastrear actualizaciones y re-renderizados de componentes

Accediendo Componentes desde la Consola: OWL no expone una forma pública y soportada de obtener la instancia de un componente montado a partir de un elemento del DOM — para eso está justamente la extensión OWL DevTools de arriba. Si necesitas acceso programático mientras desarrollas, expón la instancia tú mismo, de forma deliberada, como un atajo solo para depuración:

```
// En el componente, solo durante el desarrollo:
setup() {
    onMounted(() => {
        // Gancho de depuración intencional - nunca confíes en esto en producción.
        this.el.__debugComponent = this;
    });
}

// Luego, en la consola del navegador:
const counterComponent = document.querySelector('.counter-widget').__debugComponent;
```

```
console.log(counterComponent.state);
console.log(counterComponent.props);
```

4. Depurando Problemas Comunes

Props No Se Actualizan:

```
// Verificar si el padre realmente está pasando nuevas props
setup() {
    onWillUpdateProps((nextProps) => {
        console.log("Props actuales:", this.props);
        console.log("Próximas props:", nextProps);
        console.log("Props cambiaron:", JSON.stringify(this.props) !==
            ↪ JSON.stringify(nextProps));
    });
}
```

Estado No Desencadena Re-renderizado:

```
// Asegurarse de usar useState correctamente
setup() {
    // Bueno - estado reactivo
    this.state = useState({ count: 0 });

    // Malo - no reactivo
    this.state = { count: 0 };
}
```

Manejadores de Eventos No Funcionan:

```
// Verificar vinculación de eventos en la plantilla
static template = xml`
    <!-- Bueno - vinculación apropiada -->
    <button t-on-click="increment">+</button>

    <!-- Malo - llamando función inmediatamente -->
    <button t-on-click="increment()">+</button>

    <!-- Bueno - función flecha para parámetros -->
    <button t-on-click="() => this.incrementBy(5)">+5</button>
`;
```

Mejores Prácticas de Pruebas

1. Escribir Nombres Descriptivos de Pruebas

```
// Malo - nombres de prueba vagos
test("probar contador", async () => { ... });
test("prueba de botón", async () => { ... });

// Bueno - nombres de prueba descriptivos
test("el contador muestra el valor inicial de las props", async () => { ... });
test("el botón de incremento se desactiva en el valor máximo", async () => { ... });
```

2. Probar Comportamiento, No Implementación

```
// Malo - probando detalles de implementación
test("state.count aumenta en 1", async () => {
    const counter = await mountWithCleanup(Counter);
    counter.state.count = 5; // Manipulación directa del estado
    expect(counter.state.count).toBe(5);
});

// Bueno - probando comportamiento visible para el usuario
test("hacer clic en el botón de incremento aumenta el valor mostrado", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
    expect(".counter-value").toHaveText("1");
});
```

3. Usar el Patrón Organizar-Actuar-Afirmar

```
test("el botón de reinicio restaura el valor inicial", async () => {
    // Organizar
    await mountWithCleanup(Counter, { props: { initialValue: 5 } });
    await click(".increment-btn"); // Cambiar valor

    // Actuar
    await click(".reset-btn");

    // Afirmar
    expect(".counter-value").toHaveText("5");
});
```

4. Mantener las Pruebas Independientes

```
// Malo - las pruebas dependen unas de otras
let sharedCounter;

test("configurar contador", async () => {
    sharedCounter = await mountWithCleanup(Counter);
    expect(sharedCounter).toBeTruthy();
});

test("incrementar contador", async () => {
    await click(".increment-btn"); // Depende de que la prueba anterior siga montada
    // ...
});

// Bueno - cada prueba es independiente
test("el contador puede ser incrementado", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
    expect(".counter-value").toHaveText("1");
});
```

5. Probar Casos Límite

```
// Una prueba por escenario mantiene los fallos fáciles de identificar
test("maneja valores extremos", async () => {
```

```
    await mountWithCleanup(Counter, {
        props: { initialValue: Number.MAX_SAFE_INTEGER, max: Number.MAX_SAFE_INTEGER },
    });
    expect(".counter-value").toHaveCount(1);
});

test("maneja rangos negativos", async () => {
    await mountWithCleanup(Counter, {
        props: { initialValue: -5, min: -10, max: 0 },
    });
    expect(".counter-value").toHaveText("-5");
});

test("maneja min igual a max", async () => {
    await mountWithCleanup(Counter, {
        props: { initialValue: 5, min: 5, max: 5 },
    });
    expect(".increment-btn").toHaveProperty("disabled", true);
});
```

Errores Comunes y Soluciones

Error Común 1: Olvidar `await` en Interacciones Asíncronas

```
// Malo - la aserción se ejecuta antes de que el DOM se haya vuelto a renderizar
test("inestable", async () => {
    await mountWithCleanup(Counter);
    click(".increment-btn"); // ¡no se esperó!
    expect(".counter-value").toHaveText("1");
});

// Correcto - esperar el clic; click() de hoot solo resuelve después de que OWL
    ↪ re-renderiza
test("confiable", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
    expect(".counter-value").toHaveText("1");
});
```

Error Común 2: Componentes que Sobreviven entre Pruebas

El antiguo framework basado en QUnit exigía llamar tú mismo a `counter.destroy()` al final de cada prueba, y olvidarlo dejaba un componente montado con el que la siguiente prueba tropezaba. `mountWithCleanup()` elimina este error de raíz — siempre desmonta el componente automáticamente cuando termina la prueba. El error ahora es recurrir por costumbre al `mount()` plano de OWL en su lugar:

```
// Malo - el mount() plano de @odoo/owl nunca se limpia automáticamente
import { mount } from "@odoo/owl";

test("el incremento funciona", async () => {
    await mount(Counter, document.body);
    await click(".increment-btn");
```

```
    // este componente sigue montado cuando empieza la siguiente prueba
});

// Correcto - mountWithCleanup() siempre desmonta al terminar la prueba, pase o falle
test("el incremento funciona", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn");
});
```

Error Común 3: Recurrir a `console.log` en Vez del Inspector de Componentes

Poner `console.log` por todos lados funciona, pero es lento de iterar y fácil de olvidar quitar después. Para cualquier cosa más allá de una verificación puntual rápida, instala la extensión OWL DevTools (ver arriba) e inspecciona `state`/`props` directamente sobre el árbol de componentes en vivo — es más rápido y nunca termina accidentalmente en producción.

Error Común 4: Depurar Sin Leer el Stack Trace Completo

Es tentador saltar directo a un `debugger` o a un `console.log` cuando algo se rompe. Primero lee el stack trace completo en la consola — normalmente indica el componente, método y línea exactos, lo que te dice dónde poner un breakpoint en vez de adivinar.

Ejercicios

Ejercicio 1: Probar un Nuevo Comportamiento

Agrega una prop `step` a `Counter` (con valor por defecto `1`) que controle cuánto cambian `increment`/`decrement` el conteo. Escribe una prueba que monte el componente con `props: { step: 5 }` y verifique que un solo clic en el botón de incremento mueve el valor de `0` a `5`.

Ejercicio 2: Hacer Mock de un Fallo de Servicio

Usando `TaskManager`, escribe una prueba donde `onRpc("project.task", "create", ...)` lance un error (no solo `search_read`) y verifica que el componente muestra una notificación de "Error al agregar tarea" en vez de agregar la tarea a la lista.

Ejercicio 3: Depurar un Manejador Roto

Toma `onKeydown` y rómpelo intencionalmente comparando `event.key === "enter"` (minúscula) en vez de `"Enter"`. Usa un breakpoint o logging en consola para encontrar el error antes de mirar la solución, y luego corrígelo.

Ahora tienes los fundamentos: por qué importan las pruebas, cómo configurar tu entorno, cómo escribir y ejecutar pruebas para componentes con y sin servicios, cómo depurar problemas con herramientas del navegador, y las buenas prácticas que mantienen una suite de pruebas confiable. En la Parte 2 construiremos sobre esta base con patrones de pruebas avanzados, Desarrollo Dirigido por Pruebas, pruebas de rendimiento e integración, integración continua, y las técnicas que los equipos profesionales usan para depurar problemas que solo aparecen en producción.

TL;DR: Escribe pruebas con `@odoo/hoot` usando `mountWithCleanup`, haz mock de los servicios de los que depende un componente, y usa las OWL DevTools del navegador en vez de esparcir `console.log` — esa combinación es lo que hace que cambiar código después sea seguro en vez de aterrador.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch16_1_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

Ya puedes escribir y confiar en una suite de pruebas básica. El Capítulo 16 Parte 2 va más allá — patrones de pruebas avanzados, Desarrollo Dirigido por Pruebas, y las técnicas de depuración que necesitas cuando un bug solo aparece en producción.

Capítulo 16 Parte 2: Pruebas Avanzadas, TDD y Depuración en Producción

> **¿Por qué este capítulo?** Los proyectos reales tarde o temprano se topan con los problemas que no aparecen en una revisión manual rápida — una fuga de memoria que solo aparece después de semanas corriendo, una prueba inestable que erosiona la confianza en toda la suite, un bug que solo se reproduce bajo carga de producción. Este es el instrumental para esa etapa de la vida de un proyecto.
>
> **¿Qué trata de resolver Odoo con esto?** Darle al equipo una forma de probar la integración entre servicios, conectar CI para detectar regresiones antes del despliegue, y depurar problemas de producción de forma segura — en vez de sesiones improvisadas de `console.log` en el servidor en vivo de un cliente.
>
> **Aplicación en la vida real:** Diagnosticar una fuga de memoria que un cliente reporta después de un mes de uso diario, o configurar CI para que cada merge se pruebe automáticamente antes de llegar a su instancia.

En la Parte 1 cubrimos los fundamentos de pruebas y depuración de componentes OWL: escribir tus primeras pruebas, probar componentes que dependen de servicios, usar las DevTools del navegador efectivamente y seguir buenas prácticas de pruebas. Ahora iremos más lejos: patrones de pruebas avanzados, Desarrollo Dirigido por Pruebas, pruebas de rendimiento e integración, integración continua, errores en los que caen incluso desarrolladores experimentados, y cómo depurar problemas una vez que tu código está corriendo en producción.

Patrones Avanzados de Pruebas

1. Probando Comunicación de Componentes

Al probar la comunicación padre-hijo:

```
/** @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("Integración de Counter", () => {
    test("comunicación padre-hijo", async () => {
        const receivedEvents = [];

        // Crear un componente padre que usa nuestro componente probado
        class TestParent extends Component {
```

```
            static template = xml`
                <div>
                    <Counter initialValue="5" onValueChange.bind="onCounterChange"/>
                    <p class="parent-display">El padre ve: <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);

        // Interactuar con el componente hijo
        await click(".increment-btn");

        // Verificar que el padre recibió el evento
        expect(receivedEvents).toEqual([6]);
        expect(".parent-display").toHaveText("El padre ve: 6");
    });
});
```

2. Probando con Cambios de Props

Probar cómo los componentes reaccionan a cambios de props. Como en OWL las props siempre fluyen hacia abajo desde un padre, la forma más limpia de probar esto es envolver el componente en un pequeño padre de prueba y cambiar lo que le pasa hacia abajo — el mismo patrón usado arriba:

```
describe("Integración de Counter", () => {
    test("reacciona a cambios de props", async () => {
        class TestParent extends Component {
            static template = xml`<Counter max="state.max"/>`;
            static components = { Counter };
            setup() {
                this.state = useState({ max: 5 });
            }
        }

        await mountWithCleanup(TestParent);

        // Incrementar hasta el máximo
        for (let i = 0; i < 5; i++) {
            await click(".increment-btn");
        }
        expect(".increment-btn").toHaveProperty("disabled", true);

        // Subir el máximo en el estado del padre; OWL re-renderiza el hijo con las
            ↪  nuevas props
```

```
        // Nota: este fragmento solo ilustra la idea - el estado de TestParent no es
        // accesible desde afuera, así que en una prueba real expondrías una forma de
        // actualizarlo (p. ej. un botón en la plantilla de TestParent) y harías clic
            ↪ ahí.
    });
});
```

3. Probando Operaciones Asíncronas

Para componentes con operaciones asíncronas:

```
describe("Integración de TaskManager", () => {
    test("maneja operaciones asíncronas", async () => {
        let resolvePromise;
        const asyncPromise = new Promise((resolve) => {
            resolvePromise = resolve;
        });
        onRpc("project.task", "search_read", () => asyncPromise);

        await mountWithCleanup(TaskManager);

        // Verificar estado de carga
        expect(".task-list p").toHaveCount(1);

        // Resolver la operación asíncrona
        resolvePromise([]);
        await animationFrame(); // dejar que OWL procese el re-render tras resolver la
            ↪ promesa

        // Verificar que la carga desapareció
        expect(".task-list p").toHaveCount(0);
    });
});
```

Agrega `import { animationFrame } from "@odoo/hoot-dom";` al principio — es el helper genérico de "esperar un tick a que el DOM se actualice" en hoot, útil cuando resuelves una promesa a mano en vez de pasar por `click()` o `edit()` (que ya esperan internamente).

Desarrollo Dirigido por Pruebas (TDD) con OWL

El Desarrollo Dirigido por Pruebas es un enfoque poderoso donde escribes pruebas antes de implementar la funcionalidad:

1. Ciclo Rojo-Verde-Refactorizar

Rojo: Escribir una prueba que falle

```
test("el contador se muestra correctamente", async () => {
    await mountWithCleanup(Counter);
    expect(".counter-value").toHaveText("0");
});
```

Verde: Escribir código mínimo para que pase

```
export class Counter extends Component {
    static template = xml`
        <div>
            <span class="counter-value">0</span>
        </div>
    `;
}
```

Refactorizar: Mejorar el código manteniendo las pruebas verdes

```
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. Beneficios del TDD

- **Requisitos Claros:** Las pruebas sirven como especificaciones
- **Mejor Diseño:** Escribir pruebas primero lleva a código más testeable y modular
- **Protección contra Regresiones:** Suite integral de pruebas detecta rupturas futuras
- **Confianza:** Pruebas verdes significan software funcionando

---

Pruebas de Rendimiento y Depuración

1. Midiendo el Rendimiento de Componentes

```
test("rendimiento del contador con muchas actualizaciones", async () => {
    await mountWithCleanup(Counter, { props: { max: 1000 } });

    const startTime = performance.now();

    // Simular clics rápidos
    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. Detección de Fugas de Memoria

Esta prueba gestiona el montaje/destrucción a mano en vez de usar `mountWithCleanup()`, ya que su propósito es verificar que la limpieza manual no deja nada atrás — así que importa `mount` directamente de `@odoo/owl`:

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

test("el contador se limpia apropiadamente", async () => {
    const initialComponents = document.querySelectorAll(".counter-widget").length;
    const container = document.createElement("div");

    // Crear y destruir múltiples componentes
    for (let i = 0; i < 10; i++) {
        const counter = await mount(Counter, container);
        counter.destroy();
        container.innerHTML = ""; // Limpiar el contenedor
    }

    // Forzar recolección de basura si está disponible
    if (window.gc) {
        window.gc();
    }

    const finalComponents = document.querySelectorAll(".counter-widget").length;
    expect(finalComponents).toBe(initialComponents);
});
```

Pruebas de Integración

Probando Componentes en Escenarios Reales

A veces necesitas probar cómo los componentes trabajan juntos en condiciones realistas:

```
test("el contador funciona en contexto de formulario", async () => {
    // Crear un componente padre realista
    class TestForm extends Component {
        static template = xml`
            <form t-on-submit.prevent="onSubmit">
                <div class="form-group">
                    <label>Cantidad:</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">Ordenar</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() {
            // Lógica de envío del formulario
        }
    }

    await mountWithCleanup(TestForm);

    // Probar comportamiento integrado
    await click(".increment-btn");

    expect(".counter-value").toHaveText("2");
    expect(".form-group:nth-child(2)").toHaveText("Total: $20");
});
```

Integración Continua y Pruebas

1. Ejecutando Pruebas en CI/CD

Crear un script de prueba para tu módulo:

Archivo: `mi_modulo/tests/test_js.py`

```
import odoo.tests


@odoo.tests.tagged('post_install', '-at_install')
class TestJavaScript(odoo.tests.HttpCase):

    def test_js_modules(self):
        """Probar que los módulos JavaScript cargan y las pruebas unitarias de hoot
            ↪ pasan"""
        self.browser_js(
            "/web/tests?module=mi_modulo&failfast",
            code="",
            timeout=60,
        )
```

2. Reportes de Cobertura de Pruebas

La cobertura para las pruebas de hoot se habilita desde el propio runner de pruebas (p. ej. un parámetro de URL `coverage=1` en `/web/tests`), no desde dentro del archivo de prueba. Una vez que termina una ejecución con cobertura habilitada, el objeto de cobertura estándar de Istanbul queda disponible en la página para inspeccionarlo o exportarlo:

```
if (window.__coverage__) {
    console.log("Datos de cobertura:", window.__coverage__);
```

```
}
```

Errores Comunes de Pruebas y Soluciones

1. Pruebas Inestables

Problema: Pruebas que a veces pasan y a veces fallan

Solución: Asegurar el manejo apropiado de async

```
// Malo - no esperar operaciones asíncronas
test("prueba inestable", async () => {
    await mountWithCleanup(Counter);
    click(".increment-btn"); // ¡No se esperó!
    expect(".counter-value").toHaveText("1");
});

// Bueno - esperar apropiadamente operaciones asíncronas
test("prueba confiable", async () => {
    await mountWithCleanup(Counter);
    await click(".increment-btn"); // click() de hoot ya espera el re-renderizado
    expect(".counter-value").toHaveText("1");
});
```

2. Probando Métodos Privados

Problema: Querer probar métodos internos del componente

Solución: Probar a través de la interfaz pública

```
// Malo - probando métodos privados
test("_calculateTotal funciona", async () => {
    const component = await mountWithCleanup(MyComponent);
    const result = component._calculateTotal(5, 10);
    expect(result).toBe(50);
});

// Bueno - probando a través del comportamiento público
test("muestra el total correcto cuando cambia la cantidad", async () => {
    await mountWithCleanup(MyComponent);
    await edit(".quantity-input", "5");
    expect(".total").toHaveText("Total: 50");
});
```

3. Exceso de Mocking

Problema: Hacer mock de demasiadas dependencias

Solución: Hacer mock solo de lo necesario

```
// Malo - exceso de mocking de cada servicio que toca el componente
mockService("orm", mockOrm);
mockService("notification", mockNotification);
mockService("user", mockUser);
mockService("company", mockCompany);
```

```
// ... hacer mock de todo vuelve la prueba frágil y esconde bugs reales de integración

// Bueno - hacer mock solo de la llamada RPC específica que le importa a esta prueba
onRpc("mi.modelo", "mi_metodo_especifico", () => mockData);
// el resto de las solicitudes pasan por el manejo normal
```

Depurando Problemas de Producción

1. Límites de Error y Logging

```
export class ErrorBoundary extends Component {
    static template = xml`
        <div>
            <t t-if="state.hasError">
                <div class="alert alert-danger">
                    <h4>Algo salió mal</h4>
                    <p t-esc="state.error.message"/>
                    <button t-on-click="retry">Intentar de nuevo</button>
                </div>
            </t>
            <t t-else="">
                <t t-slot="default"/>
            </t>
        </div>
    `;

    setup() {
        this.state = useState({
            hasError: false,
            error: null
        });
    }

    catchError(error) {
        console.error("Error de componente capturado:", error);

        // Enviar al servicio de reporte de errores
        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 para Despliegues Seguros

```
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")) {
            // Nueva implementación
            this.state.count += this.props.step || 1;
        } else {
            // Respaldo seguro
            this.state.count++;
        }
    }
}
```

Ejercicios

Ejercicio 1: Escribe un Ciclo de TDD

Siguiendo el ciclo Rojo-Verde-Refactorizar, escribe una prueba que falle para un nuevo getter `doubled` en `Counter` que devuelva `state.count * 2`, implementa el código mínimo para que pase, y luego refactoriza si ves margen de mejora.

Ejercicio 2: Hacer Mock de un Servicio y Verificar sus Argumentos

Escribe una prueba para `TaskManager.deleteTask` que haga mock del servicio `orm` directamente con `mockService("orm", ...)` (en vez de pasar por `onRpc`) y verifique que `orm.unlink` fue llamado con `["project.task", [taskId]]`.

Ejercicio 3: Esboza una Verificación de CI

Esboza (en comentarios, sin necesidad de ejecutarlo) qué agregarías a `test_js.py` para que CI falle el build si alguna prueba de hoot de tu módulo falla. En 2-3 frases, explica por qué ejecutar pruebas JS en CI importa aunque ya pasen en tu máquina.

Resumen: Construyendo Aplicaciones OWL Robustas

Las pruebas y la depuración no son ideas tardías: son partes integrales de construir aplicaciones OWL profesionales. Aquí está tu hoja de ruta hacia la excelencia:

1. Estrategia de Pruebas

- **Pruebas Unitarias**: Probar componentes individuales de forma aislada
- **Pruebas de Integración**: Probar interacciones entre componentes

- **Pruebas de Extremo a Extremo**: Probar flujos de trabajo completos del usuario
- **Pruebas de Rendimiento**: Asegurar tiempos de respuesta aceptables
- **Pruebas de Manejo de Errores**: Verificar modos de falla elegantes

2. Kit de Herramientas de Depuración

- **DevTools del Navegador**: Tu interfaz principal de depuración
- **Logging Estratégico**: Capturar cambios importantes de estado
- **Límites de Error**: Manejo elegante de errores
- **Monitoreo de Rendimiento**: Rastrear y optimizar operaciones lentas

3. Prácticas Profesionales

- **Desarrollo Dirigido por Pruebas**: Escribir pruebas antes de la implementación
- **Integración Continua**: Ejecución automatizada de pruebas
- **Cobertura de Código**: Asegurar cobertura adecuada de pruebas
- **Documentación**: Especificaciones claras y ejecutables

4. Mentalidad de Mantenimiento

- **Refactorizar con Confianza**: Las buenas pruebas permiten cambios seguros
- **Prevención de Regresiones**: Las pruebas detectan rupturas inesperadas
- **Documentación Viviente**: Las pruebas muestran cómo deberían funcionar los componentes
- **Colaboración en Equipo**: Entendimiento compartido a través de pruebas

Al dominar estas técnicas de pruebas y depuración, te transformas de alguien que escribe código que funciona hoy en alguien que construye aplicaciones robustas y mantenibles que continúan funcionando mientras evolucionan. Esta es la marca distintiva del desarrollo profesional de OWL.

Recuerda: cada error que detectes en una prueba es un error que tus usuarios nunca verán. Cada técnica de depuración que domines es tiempo ahorrado en futuras investigaciones. Invierte en estas habilidades, y te pagarán dividendos a lo largo de tu carrera profesional.

TL;DR: TDD, mocking de servicios, CI y depuración segura en producción son lo que separa "funciona" de "sigue funcionando" — invierte en ellos en cuanto un componente hace algo de lo que un cliente realmente depende.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch16_2_ex1`. Las instrucciones de instalación están en el README del repositorio.

> ## ¿Qué Sigue?
>
> Pruebas y depuración asumen que sigues escribiendo OWL 2.0 puro. El Capítulo 17 Parte 1 cubre la realidad de la mayoría de los proyectos de Odoo: puentear componentes OWL nuevos con el sistema de widgets legado que todavía corre en producción.

Capítulo 17 Parte 1: El Puente Legacy-OWL - Comprensión e Implementación Básica

> **¿Por qué este capítulo?** Porque casi ningún proyecto real de Odoo parte de cero — vas a pasar mucho más tiempo trabajando dentro de una base de código que ya tiene años de widgets legado que construyendo algo nuevo en OWL puro.
>
> **¿Qué trata de resolver Odoo con esto?** Odoo necesita una forma de que el código frontend viejo y nuevo convivan durante la migración de varios años del sistema de widgets legado a OWL, sin forzar una reescritura riesgosa y de una sola vez de todas las personalizaciones de cada cliente.
>
> **Aplicación en la vida real:** Esta es exactamente la situación que me llevó a escribir este libro: la actualización de un cliente rompió todos los widgets personalizados que habíamos construido para ellos, y las técnicas de puente de este capítulo son las que te permiten modernizar un módulo pieza por pieza en vez de congelar un proyecto hasta terminar una reescritura completa.

¡Felicidades por llegar al último capítulo central del libro! Has dominado JavaScript moderno, construido componentes OWL sofisticados, implementado gestión de estado global y aprendido prácticas profesionales de pruebas. Estás completamente equipado para construir aplicaciones de vanguardia en Odoo desde cero.

Sin embargo, la realidad del desarrollo profesional de Odoo es más matizada. La mayor parte de tu trabajo no involucrará proyectos desde cero donde puedas usar OWL puro en todas partes. En su lugar, trabajarás con bases de datos de Odoo existentes llenas de años de personalizaciones, módulos de terceros e interfaces de usuario construidas con el framework JavaScript más antiguo de Odoo: el **sistema de widgets legado**.

Esto crea un desafío crítico: ¿Cómo modernizar aplicaciones de forma incremental? ¿Cómo introducir componentes OWL poderosos en interfaces legado existentes sin romper todo? ¿Cómo aprovechar widgets legado existentes en nuevas aplicaciones OWL cuando reconstruirlos tomaría meses?

La respuesta está en el **puente legacy-OWL** de Odoo: una capa de compatibilidad sofisticada que permite interoperabilidad perfecta entre código viejo y nuevo. Esto no es solo una curiosidad técnica; es una herramienta esencial para cualquier desarrollador profesional de Odoo trabajando en proyectos del mundo real.

Entendiendo el Panorama Legado

Antes de sumergirnos en las técnicas del puente, entendamos entre qué estamos haciendo el puente.

El Sistema de Widgets Legado

El framework JavaScript legado de Odoo, usado desde la versión 8 hasta partes de la versión 16, fue construido alrededor de:

Arquitectura Basada en Widgets: Los componentes eran clases que extendían `Widget`

```
const MyWidget = Widget.extend({
    template: 'my_module.MyTemplate',

    start: function() {
        // Lógica de inicialización
        return this._super.apply(this, arguments);
    },

    destroy: function() {
        // Lógica de limpieza
        this._super.apply(this, arguments);
    }
});
```

Uso Intensivo de jQuery: Manipulación directa del DOM y manejo de eventos

```
events: {
    'click .my-button': '_onButtonClick',
},

_onButtonClick: function(event) {
    this.$('.result').text('¡Botón clickeado!');
}
```

Plantillas QWeb: Compilación de plantillas del lado del servidor

```
<t t-name="my_module.MyTemplate">
  <div class="my-widget">
    <button class="my-button">Haz clic</button>
    <div class="result"></div>
  </div>
</t>
```

¿Por Qué Hacer un Puente en Lugar de Reescribir?

Realidad Práctica: Las instalaciones grandes de Odoo tienen cientos de widgets personalizados que representan años de lógica de negocio y refinamientos de interfaz de usuario.

Gestión de Riesgos: Una reescritura completa introduce riesgo significativo de romper funcionalidad existente de la que los usuarios dependen diariamente.

Limitaciones de Recursos: Reescribir todo de una vez requeriría recursos masivos de desarrollo y cronogramas de proyecto extendidos.

Continuidad del Negocio: Los usuarios necesitan nuevas características mientras la funcionalidad existente continúa funcionando de manera confiable.

El enfoque del puente permite **modernización gradual**: introducir componentes OWL incrementalmente mientras se mantiene la estabilidad del sistema.

Visión General de la Arquitectura del Puente

El puente legacy-OWL funciona en ambas direcciones:

Figura 7: La dirección 1 monta OWL dentro del widget; la dirección 2 envuelve el widget para poder usarlo desde OWL.

Dirección 1: OWL en Legado - Montar componentes OWL modernos dentro de widgets legado existentes - Agregar nuevas características sin reescribir pantallas completas - Modernizar gradualmente las interfaces de usuario

Dirección 2: Legado en OWL - Usar widgets legado existentes dentro de nuevas aplicaciones OWL - Aprovechar funcionalidad legado compleja sin reconstruir - Mantener compatibilidad con widgets de terceros

---

Escenario 1: Incrustando Componentes OWL en Widgets Legado

Este es el patrón de modernización más común. Tienes un formulario o dashboard legado existente, y quieres agregar una nueva característica sofisticada construida con OWL.

Ejemplo: Agregando un Gráfico Moderno a un Dashboard Legado

Digamos que tienes un widget de dashboard de ventas legado, y quieres agregar un gráfico de ingresos interactivo construido con OWL.

1. El Widget Dashboard Legado

```
odoo.define('sales_dashboard.LegacyDashboard', function (require) {
    "use strict";

    const Widget = require('web.Widget');
    const core = require('web.core');
    const { mount } = require("@odoo/owl");

    // Importar nuestro componente OWL moderno
```

```
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 = {}; // Rastrear componentes OWL montados
    },

    async start() {
        await this._super(...arguments);

        // Inicializar funcionalidad legado
        this._setupLegacyFeatures();

        // Montar componentes OWL
        await this._mountOWLComponents();

        // Cargar datos iniciales
        await this._loadDashboardData();
    },

    _setupLegacyFeatures: function() {
        // Funcionalidad basada en jQuery legado
        this.$('.legacy-counter').each(function(index, element) {
            $(element).data('value', 0);
        });

        // Inicializar selector de fecha legado
        this.$('.date-filter').datepicker({
            format: 'yyyy-mm-dd',
            onSelect: this._onDateFilterChange.bind(this)
        });
    },

    async _mountOWLComponents() {
        try {
            // Montar el gráfico de ingresos en su contenedor designado
            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),
                        // Pasar referencia del widget legado para interacciones
                            ↪ complejas
```

```
                    legacyWidget: this,
                }
            });
        }

        // Montar componentes OWL adicionales según sea necesario
        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('Falló al montar componentes OWL:', error);
        // Degradación elegante - mostrar mensaje de error o UI de respaldo
        this._showComponentError(error);
    }
},
```

Una vez montados los componentes OWL, el widget carga sus datos y los envía tanto a los contadores legados como a los props de los componentes OWL:

```
async _loadDashboardData() {
    try {
        const data = await this._rpc({
            model: 'sale.order',
            method: 'get_dashboard_data',
            args: [this._getDateRange()],
        });

        // Actualizar widgets legado
        this._updateLegacyCounters(data.counters);

        // Actualizar componentes OWL
        await this._updateOWLComponents(data);

    } catch (error) {
        console.error('Falló al cargar datos del dashboard:', error);
        this._showError('Falló al cargar datos del dashboard');
    }
},

async _updateOWLComponents(data) {
    // Actualizar props del componente OWL
    if (this.owlComponents.revenueChart) {
        this.owlComponents.revenueChart.props.data = data.revenue;
        // Disparar re-renderizado actualizando 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) {
        // Actualizaciones basadas en jQuery legado
        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);
            }
        });
    },

    // Manejadores de eventos para funcionalidad legado
    _onRefreshData: function(event) {
        event.preventDefault();
        this._loadDashboardData();
    },

    _onDateFilterChange: function(event) {
        const newDateRange = this._getDateRange();
        this._loadDashboardData();
    },

    // Manejadores de eventos para interacciones de componentes OWL
    _onChartDataPointClick: function(dataPoint) {
        // Manejar clics desde gráfico OWL - tal vez mostrar diálogo de desglose
            ↪ legado
        this._showDrillDownDialog(dataPoint);
    },

    _onChartDateRangeChange: function(dateRange) {
        // Actualizar filtros de fecha legado cuando el gráfico OWL cambia el rango
            ↪ de fecha
        this.$('.date-filter').datepicker('setDate', dateRange.start);
        this._loadDashboardData();
    },

    _onMetricClick: function(metric) {
        // Navegar a vista de lista legado
        this.do_action({
```

```
                type: 'ir.actions.act_window',
                res_model: 'sale.order',
                views: [[false, 'list']],
                domain: metric.domain,
                context: metric.context,
            });
        },

        // Métodos utilitarios
        _getDateRange: function() {
            return {
                start: this.$('.date-start').val(),
                end: this.$('.date-end').val(),
            };
        },

        _showDrillDownDialog: function(dataPoint) {
            // Implementación de diálogo legado
            const dialog = new Dialog(this, {
                title: `Detalles para ${dataPoint.label}`,
                size: 'medium',
                $content: $(`<div>Ingresos: ${dataPoint.value}</div>`),
            });
            dialog.open();
        },

        _showComponentError: function(error) {
            this.$('.owl-error-container').show().find('.error-message').text(
                'Falló al cargar componentes interactivos. Por favor, recarga la página.'
            );
        },

        _showError: function(message) {
            // Visualización de error legado
            this.$('.error-container').show().find('.error-text').text(message);
        },
```

La última pieza es la más importante de este capítulo: limpiar los componentes OWL montados. Como la clase legada `Widget` no tiene ningún hook de ciclo de vida automático para esto, hay que hacerlo a mano dentro de `destroy`, o cada componente OWL montado generará una fuga de memoria cada vez que el widget legado se cierre y se vuelva a abrir.

```
        // Crítico: Limpiar componentes OWL para prevenir fugas de memoria
        destroy: function() {
            // Desmontar todos los componentes OWL
            Object.values(this.owlComponents).forEach(component => {
                if (component && component.destroy) {
                    component.destroy();
                }
            });
            this.owlComponents = {};

            // Llamar al destroy padre
            this._super.apply(this, arguments);
        },
    });
```

```
    return LegacyDashboard;
});
```

2. La Plantilla Legado

```
<t t-name="sales_dashboard.LegacyDashboardTemplate">
  <div class="legacy-sales-dashboard">
    <!-- Encabezado legado con controles -->
    <div class="dashboard-header">
      <h2>Dashboard de Ventas (Legado)</h2>
      <div class="dashboard-controls">
        <input type="text" class="date-filter date-start" placeholder="Fecha Inicio"/>
        <input type="text" class="date-filter date-end" placeholder="Fecha Fin"/>
        <button class="btn btn-primary refresh-data">Actualizar</button>
      </div>
    </div>

    <!-- Contadores legado -->
    <div class="legacy-counters row">
      <div class="col-md-3">
        <div class="counter-widget">
          <h4>Ventas Totales</h4>
          <div class="counter" data-metric="total_sales">0</div>
        </div>
      </div>
      <div class="col-md-3">
        <div class="counter-widget">
          <h4>Clientes Nuevos</h4>
          <div class="counter" data-metric="new_customers">0</div>
        </div>
      </div>
      <div class="col-md-3">
        <div class="counter-widget">
          <h4>Negocios Activos</h4>
          <div class="counter" data-metric="active_deals">0</div>
        </div>
      </div>
      <div class="col-md-3">
        <div class="counter-widget">
          <h4>Tasa de Conversión</h4>
          <div class="counter" data-metric="conversion_rate">0%</div>
        </div>
      </div>
    </div>

    <!-- Componentes OWL modernos montados aquí -->
    <div class="modern-components">
      <div class="row">
        <div class="col-md-8">
          <div class="card">
            <div class="card-header">
              <h5>Tendencias de Ingresos (Componente OWL Moderno)</h5>
            </div>
            <div class="card-body">
              <!-- El componente OWL será montado aquí -->
```

```
            <div class="revenue-chart-container"></div>
            <!-- Respaldo cuando el componente OWL falla -->
            <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>Métricas Clave (Componente OWL Moderno)</h5>
          </div>
          <div class="card-body">
            <!-- Otro componente OWL montado aquí -->
            <div class="metrics-widget-container"></div>
          </div>
        </div>
      </div>
    </div>

    <!-- Tabla legado -->
    <div class="legacy-table-section">
      <h4>Órdenes Recientes (Tabla Legado)</h4>
      <table class="table table-striped">
        <thead>
          <tr>
            <th>Orden #</th>
            <th>Cliente</th>
            <th>Monto</th>
            <th>Fecha</th>
          </tr>
        </thead>
        <tbody class="recent-orders-tbody">
          <!-- Poblado por JavaScript legado -->
        </tbody>
      </table>
    </div>

    <!-- Contenedor de errores -->
    <div class="error-container" style="display: none;">
      <div class="alert alert-danger">
        <span class="error-text"></span>
      </div>
    </div>
  </div>
</t>
```

3. El Componente OWL Moderno RevenueChart

```
/** @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">Gráfico de Línea</option>
                    <option value="bar">Gráfico de Barras</option>
                    <option value="area">Gráfico de Área</option>
                </select>
                <button class="btn btn-sm btn-secondary" t-on-click="exportChart">
                    Exportar
                </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">
                    Mostrando <t t-esc="chartSummary.totalDataPoints"/> puntos de datos.
                    Mayor: <t t-esc="chartSummary.highest"/>
                    Promedio: <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 }, // Referencia al widget legado
    };

    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() {
        // Inicializar Chart.js o librería de gráficos similar
        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 `Ingresos: $${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: 'Ingresos',
                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("Gráfico exportado exitosamente", { 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;
    }
}
```

Este ejemplo demuestra varios principios clave:

1. **Separación Clara**: El código legado y OWL permanece separado pero se comunica a través de interfaces bien definidas
2. **Degradación Elegante**: Si los componentes OWL fallan al cargar, la funcionalidad legado continúa funcionando
3. **Comunicación Bidireccional**: Los componentes OWL pueden notificar a widgets legado de eventos, y viceversa
4. **Gestión de Memoria**: Limpieza apropiada previene fugas de memoria
5. **Manejo de Errores**: Manejo robusto de errores asegura estabilidad del sistema

---

Beneficios Clave de Este Enfoque

1. Modernización Incremental: Reemplazamos el gráfico simple con un componente OWL sofisticado mientras mantenemos características legado complejas

2. Gestión de Riesgos: La funcionalidad de exportación y diálogos legado continúa funcionando

3. Experiencia de Usuario Mejorada: Gráficos y tablas modernos e interactivos

4. Mantenibilidad: Las nuevas características pueden construirse en OWL mientras las características legado permanecen estables

5. Rendimiento: Solo cargar componentes modernos cuando sea necesario

---

Mejores Prácticas para el Escenario 1

1. Aislamiento de Componentes

Mantén los componentes OWL autocontenidos:

```
// Bueno: El componente OWL es independiente
const owlComponent = await mount(ChartComponent, target, {
    props: {
        data: this.salesData,
        onEvent: this.handleEvent.bind(this)
    }
});

// Malo: El componente OWL depende de las partes internas del widget legado
const owlComponent = await mount(ChartComponent, target, {
    props: {
        legacyWidget: this, // Demasiado acoplamiento
        domElement: this.$('.some-element') // Dependencia directa del DOM
    }
});
```

2. Límites de Error

Siempre implementa respaldos elegantes:

```
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(`Falló al montar ${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>Componente No Disponible</h6>
                <p>Esta característica está temporalmente no disponible. Por favor,
 ↪  recarga la página.</p>
                <button class="btn btn-sm btn-secondary" onclick="location.reload()">
                    Recargar Página
                </button>
            </div>
        `;
    }
}
```

3. Gestión de Memoria

Crítico para prevenir fugas:

```
destroy: function() {
    // Limpiar componentes OWL primero
    this._destroyOWLComponents();

    // Luego llamar al destroy padre
    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 destruyendo componente ${key}:`, error);
        }
    });
    this.owlComponents = {};
}
```

TL;DR: Envuelve widgets legado desde dentro de un componente OWL con `useRef` + `onMounted`/`onWillUnmount`, o monta componentes OWL dentro del `start()`/`destroy()` de un widget legado — cualquiera de las dos direcciones funciona mientras mantengas explícita la interfaz entre ambos.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch17_1_ex1`. Las instrucciones de instalación están en el README del repositorio.

Errores Comunes

Olvidar destruir los componentes OWL montados. Si montas un componente OWL dentro de `_mountOWLComponents` pero no lo desmontas en `destroy`, el componente (y todo lo que retiene: listeners de eventos, timers, suscripciones a servicios) genera una fuga cada vez que el widget legado se cierra y se vuelve a abrir.

Interactuar con el DOM del widget legado antes de que esté listo. El `this.el` del widget legado solo existe una vez que `start()` se ha ejecutado. Montar un componente OWL o hacer `this.el.querySelector(...)` desde `init()` fallará silenciosamente o lanzará un error — hazlo siempre desde dentro (o después) de `start()`.

Mezclar los dos sistemas de eventos sin un límite claro. Es tentador hacer que los componentes OWL llamen directamente a `this.trigger_up(...)` del widget legado, o que el widget legado acceda a los internos de OWL. Mantén la interfaz entre ambos explícita — callback props en una dirección, métodos públicos del widget legado en la otra — para no terminar con dos objetos que ambos creen ser dueños del mismo estado.

Asumir que los props del componente OWL son "reactivos" automáticamente. Los widgets legados mutan objetos planos; OWL solo vuelve a renderizar cuando detecta un cambio a través de su sistema de reactividad. Modificar directamente un valor en `this.owlComponents.revenueChart.props.data` (como hace el ejemplo anterior) NO dispara automáticamente un re-render — por eso el ejemplo llama explícitamente a `.render()` después.

Ejercicios

1. Toma el widget `LegacyDashboard` y añade un tercer componente OWL (por ejemplo, un `TopCustomersList` simple) montado en un nuevo contenedor. Asegúrate de que se destruya correctamente junto con los otros dos.
2. Escribe a mano el método `_destroyOWLComponents` (sin mirar el ejemplo de "Gestión de Memoria") para un widget que rastrea dos componentes OWL montados en `this.owlComponents`.
3. Modifica `_onChartDataPointClick` para que, en vez de abrir un diálogo legado, llame a un callback prop pasado desde un padre OWL — explica en una frase por qué esto no es posible en esta dirección (los widgets legados no son componentes OWL y no reciben props).

En la Parte 2, exploraremos el Escenario 2 (usar widgets legado en componentes OWL), patrones de puente avanzados, estrategias de pruebas y planificación de migración. Esta comprensión fundamental del Escenario 1 te preparará para los desafíos de integración más complejos que vienen.

¿Qué Sigue?

La Parte 2 invierte la dirección que acabas de aprender — en vez de montar OWL dentro de un widget legado, vas a envolver un widget legado dentro de un componente OWL, además de cubrir patrones de puente avanzados y un cronograma práctico de migración.

Capítulo 17 Parte 2: Patrones Avanzados de Puente y Estrategias de Migración

> **¿Por qué este capítulo?** Porque puentear en una sola dirección no alcanza en un proyecto real — con la misma frecuencia vas a necesitar reutilizar un widget legado probado en batalla (un pad de firma, una integración de escáner de código de barras) desde dentro de una pantalla OWL nueva, y vas a necesitar un plan para eventualmente retirar el puente por completo.
>
> **¿Qué trata de resolver Odoo con esto?** Más allá de la interoperabilidad básica, Odoo necesita patrones suficientemente robustos para comunicación bidireccional y pruebas entre código viejo y nuevo, además de una ruta realista e incremental para salir del sistema de widgets legado en vez de depender de él indefinidamente.
>
> **Aplicación en la vida real:** Cronogramas de migración como el de este capítulo son lo que realmente le entrego a mis clientes — no "reescribamos todo", sino un plan por fases que mantiene el sistema funcionando mientras los widgets legado se reemplazan módulo por módulo.

En la Parte 1, exploramos cómo incrustar componentes OWL modernos dentro de widgets legado. Ahora abordaremos el escenario inverso, patrones de comunicación avanzados y enfoques estratégicos para la modernización completa.

Escenario 2: Usando Widgets Legado en Componentes OWL

A veces necesitas incluir funcionalidad legado compleja en nuevas aplicaciones OWL. Esto podría ser un widget de campo especializado, un componente complejo de terceros o lógica de negocio legado que tomaría meses reescribir.

Ejemplo: Incluyendo un Widget de Campo Legado en un Formulario OWL

Digamos que estás construyendo una interfaz moderna OWL de gestión de clientes, pero necesitas incluir un widget legado complejo de firmas que maneja firmas electrónicas.

1. El Formulario de Cliente OWL Moderno

```
/** @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,
        });

        // Configuración para widget legado de firma
        this.legacySignatureConfig = {
            Widget: "web_digital_sign.DigitalSignWidget", // Nombre de clase de widget
                ↪ legado
            widgetOptions: {
                signatureMode: "draw",
                required: true,
                width: 400,
                height: 200,
                onSignatureChange: this.onSignatureChange.bind(this),
                onSignatureClear: this.onSignatureClear.bind(this),
            }
        };

        // Configuración para widget legado de dirección (otro ejemplo)
        this.legacyAddressConfig = {
            Widget: "base_address.AddressWidget",
            widgetOptions: {
                countryField: "country_id",
                stateField: "state_id",
                onAddressValidated: this.onAddressValidated.bind(this),
            }
        };
    }
```

Con la configuración lista, el componente carga el registro del cliente a través del servicio `orm`, exactamente igual que cualquier componente OWL normal — los widgets legado no cambian cómo obtienes los datos, solo cómo se renderiza parte del formulario:

```
    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];
                // Actualizar datos de widgets legado si es necesario
```

```
            this._updateLegacyWidgets();
        }
    } catch (error) {
        this.notification.add("Falló al cargar datos del cliente", { type: "danger"
            ↪ });
    } finally {
        this.state.loading = false;
    }
}

async saveCustomer() {
    if (!this._validateForm()) {
        return;
    }

    this.state.loading = true;
    try {
        // Obtener datos de widgets legado antes de guardar
        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("Cliente guardado exitosamente", { type: "success" });

    } catch (error) {
        this.notification.add("Falló al guardar cliente", { type: "danger" });
    } finally {
        this.state.loading = false;
    }
}

// Callbacks para interacciones de widgets legado
onSignatureChange(signatureData) {
    this.state.customer.signature = signatureData;
    this.state.signatureRequired = false;
    console.log("Firma actualizada:", signatureData);
}

onSignatureClear() {
    this.state.customer.signature = null;
    this.state.signatureRequired = true;
    console.log("Firma borrada");
}
```

```
    onAddressValidated(addressData) {
        // Actualizar datos del cliente con dirección validada
        Object.assign(this.state.customer, addressData);
        console.log("Dirección validada:", addressData);
    }

    // Validación de formulario incluyendo validación de widget legado
    _validateForm() {
        if (!this.state.customer.name.trim()) {
            this.notification.add("El nombre del cliente es requerido", { type:
                ↪  "warning" });
            return false;
        }

        if (!this.state.customer.email.trim()) {
            this.notification.add("El email es requerido", { type: "warning" });
            return false;
        }

        // Validar widgets legado
        if (!this._validateLegacyWidget('signature')) {
            this.notification.add("Se requiere una firma válida", { type: "warning" });
            return false;
        }

        if (!this._validateLegacyWidget('address')) {
            this.notification.add("Se requiere una dirección válida", { type: "warning"
                ↪  });
            return false;
        }

        return true;
    }

    // Métodos ayudantes para interacción con widgets legado
    _updateLegacyWidgets() {
        // Disparar actualizaciones en widgets legado cuando cambia el estado OWL
        // Esto dependería de las APIs específicas de los widgets legado
    }

    _getLegacyWidgetData(widgetType) {
        // Extraer datos de widgets legado
        // La implementación depende de cómo los widgets legado exponen sus datos
        return {};
    }

    _validateLegacyWidget(widgetType) {
        // Validar datos de widget legado
        // La implementación depende de APIs específicas de validación de widgets
        return true;
    }

    // Manejadores de eventos para elementos de formulario OWL
    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() {
        // Borrar programáticamente el widget de firma legado
        // Esto llamaría métodos en la instancia del widget legado
    }
}
```

Fíjate que `saveCustomer` extrae los datos *desde* los widgets legado (vía `_getLegacyWidgetData`) justo antes de guardar, en lugar de mantener esos datos en el estado de OWL en todo momento — los widgets legado de firma y dirección son la fuente de verdad de sus propios campos hasta el momento en que necesitas persistirlos.

2. La Plantilla OWL

```
<t t-name="customer_management.CustomerForm" owl="1">
  <div class="customer-form-container">
    <div class="form-header">
      <h2>Información del Cliente</h2>
      <div class="form-actions">
        <button class="btn btn-primary"
                t-on-click="saveCustomer"
                t-att-disabled="state.loading">
          <t t-if="state.loading">Guardando...</t>
          <t t-else="">Guardar Cliente</t>
        </button>
      </div>
    </div>

    <!-- Campos de formulario OWL modernos -->
    <div class="row">
      <div class="col-md-6">
        <div class="card modern-form-section">
          <div class="card-header">
            <h5>Información Básica (OWL Moderno)</h5>
          </div>
          <div class="card-body">
            <div class="form-group">
              <label for="customer-name">Nombre del Cliente *</label>
              <input type="text"
                     id="customer-name"
                     class="form-control"
                     t-att-value="state.customer.name"
                     t-on-input="onNameChange"
                     placeholder="Ingrese nombre del cliente"/>
            </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="cliente@ejemplo.com"/>
          </div>

          <div class="form-group">
            <label for="customer-phone">Teléfono</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">
      <!-- Widget legado de dirección -->
      <div class="card legacy-widget-section">
        <div class="card-header">
          <h5>Información de Dirección (Widget Legado)</h5>
        </div>
        <div class="card-body">
          <LegacyComponent
              widget="legacyAddressConfig.Widget"
              widgetOptions="legacyAddressConfig.widgetOptions"
              record="state.customer"
          />
        </div>
      </div>
    </div>
  </div>

  <!-- Widget legado de firma -->
  <div class="row">
    <div class="col-12">
      <div class="card legacy-widget-section">
        <div class="card-header">
          <h5>Firma Digital (Widget Legado)</h5>
          <div class="card-actions">
            <button class="btn btn-sm btn-secondary"
                    t-on-click="clearSignature">
              Borrar Firma
            </button>
          </div>
        </div>
        <div class="card-body">
          <div t-if="state.signatureRequired" class="alert alert-warning">
            Por favor, proporcione una firma digital
```

```
            </div>

            <!-- Widget legado de firma montado aquí -->
            <LegacyComponent
                widget="legacySignatureConfig.Widget"
                widgetOptions="legacySignatureConfig.widgetOptions"
                record="state.customer"
            />
          </div>
        </div>
      </div>
    </div>

    <!-- Overlay de carga -->
    <div t-if="state.loading" class="loading-overlay">
      <div class="spinner-border" role="status">
        <span class="sr-only">Cargando...</span>
      </div>
    </div>
  </div>
</t>
```

3. CSS de Apoyo para Arquitectura Mixta

```
/* Estilos específicos del formulario de cliente */
.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;
}

/* Diferenciar secciones modernas vs legado */
.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: " (Legado)";
    font-size: 0.8em;
    opacity: 0.7;
}

/* Overlay de carga */
.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;
}

/* Estados de validación de formulario */
.form-group.has-error .form-control {
    border-color: #dc3545;
}

.form-group.has-success .form-control {
    border-color: #28a745;
}
```

Entendiendo el Envoltorio LegacyComponent

El `LegacyComponent` es un componente OWL especial proporcionado por Odoo que actúa como un adaptador. Así es como funciona internamente:

La Implementación de LegacyComponent (Simplificada)

```
// Esta es una versión simplificada de cómo funciona el LegacyComponent de Odoo
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 {
            // Obtener la clase de widget legado por nombre
            const WidgetClass =
                ↪ this.env.services.legacy_widget_registry.get(this.props.widget);

            if (!WidgetClass) {
                throw new Error(`Widget legado '${this.props.widget}' no encontrado`);
            }

            // Crear instancia del widget legado
            this.legacyWidget = new WidgetClass(this, this.props.widgetOptions || {});

            // Configurar vinculación de datos si se proporciona registro
            if (this.props.record) {
                this.legacyWidget.set('record', this.props.record);
            }

            // Montar el widget legado
            await this.legacyWidget.appendTo(this.legacyContainerRef.el);

            // Configurar comunicación bidireccional
            this.setupLegacyEvents();

        } catch (error) {
            console.error('Falló al montar widget legado:', error);
            this.showError(error.message);
        }
    }
```

Una vez montado el widget, `setupLegacyEvents` conecta las dos direcciones de comunicación: los eventos legado se convierten en callbacks de OWL, y (si quien llama proporcionó uno) un prop `onValueChange` se reenvía al propio evento del widget legado.

```
    setupLegacyEvents() {
        if (!this.legacyWidget) return;

        // Escuchar eventos de widget legado
        this.legacyWidget.on('value_changed', this, this.onLegacyValueChange);
        this.legacyWidget.on('validation_error', this, this.onLegacyValidationError);

        // Reenviar eventos OWL a widget legado
        if (this.props.widgetOptions.onValueChange) {
            this.legacyWidget.on('value_changed', this, (newValue) => {
                this.props.widgetOptions.onValueChange(newValue);
            });
        }
    }
```

```
    onLegacyValueChange(newValue) {
        // Actualizar el registro si se proporciona
        if (this.props.record && this.legacyWidget.field_name) {
            this.props.record[this.legacyWidget.field_name] = newValue;
        }

        // Disparar re-renderizado de OWL si es necesario
        this.render();
    }

    onLegacyValidationError(error) {
        console.warn('Error de validación de widget legado:', error);
        // Podría disparar que el componente OWL muestre estado de error
    }

    unmountLegacyWidget() {
        if (this.legacyWidget) {
            // Remover event listeners
            this.legacyWidget.off('value_changed', this);
            this.legacyWidget.off('validation_error', this);

            // Destruir el widget legado
            this.legacyWidget.destroy();
            this.legacyWidget = null;
        }
    }

    showError(message) {
        // Renderizar estado de error en componente OWL
        this.legacyContainerRef.el.innerHTML = `
            <div class="alert alert-danger">
                <strong>Error de Widget:</strong> ${message}
            </div>
        `;
    }
}
```

Patrones Avanzados de Puente

1. Comunicación de Eventos entre Legado y OWL

A menudo, necesitas comunicación sofisticada entre widgets legado y componentes OWL:

```
// En el widget legado
const LegacyWidget = Widget.extend({
    start: function() {
        this._super(...arguments);

        // Escuchar eventos de componentes OWL
        this.eventBus = core.bus;
        this.eventBus.on('owl_component_event', this, this._onOWLEvent);
    },
```

```
    _onOWLEvent: function(data) {
        console.log('Widget legado recibió evento OWL:', data);
        // Reaccionar a eventos de componentes OWL
        this._updateLegacyUI(data);
    },

    _triggerEventForOWL: function(data) {
        // Enviar eventos a componentes OWL
        this.eventBus.trigger('legacy_widget_event', data);
    },

    destroy: function() {
        this.eventBus.off('owl_component_event', this);
        this._super(...arguments);
    }
});

// En el componente OWL
export class ModernComponent extends Component {
    setup() {
        this.eventBus = useService("bus");

        // Escuchar eventos de widgets legado
        this.eventBus.addEventListener('legacy_widget_event',
            ↪  this.onLegacyEvent.bind(this));
    }

    onLegacyEvent(event) {
        console.log('Componente OWL recibió evento legado:', event.detail);
        // Reaccionar a eventos de widgets legado
    }

    sendEventToLegacy(data) {
        // Enviar eventos a widgets legado
        this.eventBus.trigger('owl_component_event', data);
    }
}
```

2. Gestión de Estado Compartido

Para escenarios complejos donde componentes legado y OWL necesitan compartir estado:

```
// Servicio de estado compartido que tanto legado como OWL pueden usar
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;

            // Notificar oyentes
            if (this.listeners[key]) {
                this.listeners[key].forEach(callback => {
                    callback(value, oldValue);
                });
            }

            // Disparar evento global
            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);

            // Retornar función de desuscripción
            return () => {
                const index = this.listeners[key].indexOf(callback);
                if (index > -1) {
                    this.listeners[key].splice(index, 1);
                }
            };
        }
    }

    // Crear instancia singleton
    const sharedState = new SharedStateManager();

    return sharedState;
});

// Uso en widget legado
const LegacyWidget = Widget.extend({
    start: function() {
        this._super(...arguments);
        this.sharedState = require('shared_state.StateManager');

        // Suscribirse a cambios de estado
        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);
    }
});

// Uso en componente OWL
export class ModernComponent extends Component {
    setup() {
        this.sharedState = useService("shared_state");

        this.state = useState({
            currentUser: this.sharedState.getState('currentUser')
        });

        // Suscribirse a cambios de estado compartido
        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. Estrategia de Migración Progresiva

Aquí hay un enfoque sistemático para migrar de legado a OWL:

```
// Envoltorio de migración que puede cambiar entre implementaciones legado y OWL
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() {
            // Lógica para determinar si usar OWL o legado
            // Podría basarse en feature flags, preferencias de usuario, 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;
});
```

Probando Componentes Puente

Probar componentes que usan el puente legacy-OWL requiere consideraciones especiales:

1. Probando Componentes OWL con Dependencias Legado

El framework de pruebas moderno de Odoo es `@odoo/hoot` (Capítulo 16), no el runner legado de QUnit — aplican las mismas convenciones: `describe`/`test`/`expect` en vez de `QUnit.module`/`QUnit.test`/`assert`, y `mountWithCleanup` en vez de un `mount` manual + `destroy()`.

```
import { describe, test, expect, beforeEach } from "@odoo/hoot";
import { mountWithCleanup } from "@web/../tests/web_test_helpers";

describe("Componentes Puente", () => {
    let mockLegacyWidget;
    let widgetFueCreado;
    let datosFueronObtenidos;

    beforeEach(() => {
        widgetFueCreado = false;
        datosFueronObtenidos = false;

        // Objeto simple que simula la instancia del widget legado
        mockLegacyWidget = {
            start: () => Promise.resolve(),
            destroy: () => {},
            getData: () => {
                datosFueronObtenidos = true;
                return { signature: "mock_signature" };
            },
            validate: () => true,
        };

        // Mock del constructor de widget legado referenciado por nombre
        window.MockLegacyWidget = function () {
            widgetFueCreado = true;
            return mockLegacyWidget;
        };
    });

    test("Componente OWL se integra con widget legado", async () => {
        const customerForm = await mountWithCleanup(CustomerForm, {
            props: {
                legacyWidgetConfig: {
                    Widget: "MockLegacyWidget",
                    widgetOptions: { required: true },
                },
            },
        });

        // El widget legado debería haber sido instanciado
        expect(widgetFueCreado).toBe(true);

        // Debería haber ocurrido el intercambio de datos con el widget legado
        await customerForm.saveCustomer();
        expect(datosFueronObtenidos).toBe(true);

        // No hace falta destroy() manual - mountWithCleanup desmonta
        // automáticamente después de la prueba.
    });
});
```

Mejores Prácticas de Migración

1. Empezar Pequeño e Iterativo

```
// Fase 1: Agregar nuevos componentes OWL a pantallas legado
// - Agregar gráficos modernos a dashboards existentes
// - Introducir nuevos widgets interactivos
// - Mantener funcionalidad legado intacta

// Fase 2: Reemplazar componentes legado no críticos
// - Migrar widgets simples primero
// - Reemplazar controles de formulario y elementos básicos de UI
// - Usar puente para interacciones complejas

// Fase 3: Migrar funcionalidad principal
// - Reemplazar widgets principales de formulario
// - Migrar vistas de lista y kanban
// - Mantener puente solo para casos extremos

// Fase 4: Modernización completa
// - Migración completa a OWL
// - Remover código de puente
// - Optimizar rendimiento
```

2. Ejemplo de Cronograma de Migración

Trimestre 1: Fundación - Configurar infraestructura de puente - Migrar 2-3 componentes simples - Entrenar equipo en patrones de puente - Establecer procedimientos de prueba

Trimestre 2: Aceleración - Migrar 5-10 componentes de complejidad media - Construir utilidades de puente reutilizables - Optimizar cuellos de botella de rendimiento - Documentar mejores prácticas

Trimestre 3: Características Principales - Migrar componentes de lógica de negocio principal - Reemplazar interfaces principales de formulario y lista - Minimizar uso de puente - Optimización de rendimiento

Trimestre 4: Finalización - Migrar casos extremos restantes - Remover código de puente - Implementación completa de OWL - Validación de rendimiento

3. Métricas de Éxito

Rastrear el progreso de tu migración:

```
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());
```

Consideraciones de Rendimiento

1. Gestión del Tamaño del Bundle

Al hacer puente entre legado y OWL, ten en cuenta el tamaño del bundle JavaScript:

```
// Carga perezosa para evitar cargar código innecesario
const loadLegacyWidget = async (widgetName) => {
    // Solo cargar widget legado cuando sea necesario
    const { [widgetName]: Widget } = await import(`./legacy_widgets/${widgetName}`);
    return Widget;
};

const loadOWLComponent = async (componentName) => {
    // Solo cargar componente OWL cuando sea necesario
    const { [componentName]: Component } = await
        ↪ import(`./owl_components/${componentName}`);
    return Component;
};
```

2. Gestión de Memoria

La limpieza adecuada es crucial cuando se mezcla legado y 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);
        });
    }
};

// Asegurar limpieza al descargar página
window.addEventListener('beforeunload', () => {
    BridgeManager.cleanupAll();
});
```

Errores Comunes

Olvidar `onWillUnmount` en el wrapper `LegacyComponent`. Si `unmountLegacyWidget` no se llama cuando se destruye el componente OWL, la instancia del widget legado y sus listeners de eventos DOM quedan vivos para siempre — esta es la fuente más común de fugas de memoria en código puente.

Leer el estado del widget legado como si fuera reactivo. `this.legacyWidget.field_name` o valores similares son propiedades planas de un objeto legado; OWL no tiene forma de saber cuándo cambian. Cada vez que los datos del widget legado necesiten afectar el renderizado de OWL, deben pasar por un evento explícito (como `value_changed`) que actualice `this.state`.

Construir un nuevo mecanismo de "estado compartido" por cada puente en vez de reutilizar uno. El patrón `SharedStateManager` mostrado arriba está pensado como un único servicio registrado, no algo que reinventes para cada par legado/OWL — de lo contrario, los widgets legado y los componentes OWL terminan hablando con objetos de estado distintos y desconectados.

Saltarse el plan de migración. Es tentador dejar un componente puente en su lugar indefinidamente porque "funciona". Todo componente puente debería tener una fecha objetivo o condición disparadora documentada para su migración completa — de lo contrario, la capa de puente en sí misma se convierte en deuda técnica permanente.

Ejercicios

1. Extiende el ejemplo `CustomerForm` para que el callback `onSignatureChange` del widget legado de firma también llame a un nuevo método `_markFormDirty()`, y úsalo para deshabilitar el botón de Guardar hasta que el usuario proporcione una firma.
2. Usando la implementación de `LegacyComponent` como referencia, escribe (en papel o en código) el hook `onWillUnmount` que necesitarías si `LegacyComponent` no proporcionara ya uno — ¿qué exactamente hay que limpiar, y en qué orden?

3. Mira el ejemplo `SharedStateManager`: reescribe `subscribe` para que un componente que olvida llamar a la función de desuscripción devuelta no cause una fuga de memoria (pista: piensa qué pasa si `subscribe` se llama cada vez que un componente se monta, pero `unsubscribe` solo se llama manualmente).

Conclusión: Conectando el Pasado y el Futuro

El puente legacy-OWL representa más que solo una solución técnica: es un enfoque estratégico para la evolución del software. Reconoce la realidad de que en el desarrollo profesional de software, rara vez tienes el lujo de comenzar desde cero. En su lugar, debes construir el futuro mientras mantienes el presente.

A lo largo de este libro, has aprendido:

1. **Fundamentos de JavaScript moderno** que impulsan el desarrollo web contemporáneo
2. **Arquitectura de componentes OWL** para construir interfaces mantenibles y escalables
3. **Patrones avanzados** como gestión de estado, composición y pruebas
4. **Técnicas de integración del mundo real** para trabajar con sistemas existentes

Los patrones de puente en este capítulo completan tu conjunto de herramientas, dándote las habilidades para:

- **Modernizar incrementalmente** sin romper funcionalidad existente
- **Integrar suavemente** entre paradigmas arquitectónicos diferentes
- **Gestionar complejidad** en entornos de tecnología mixta
- **Planificar migraciones** estratégicamente con progreso medible

Tus Próximos Pasos

Mientras aplicas estos conceptos en tus propios proyectos:

1. **Empezar pequeño**: Elige componentes de bajo riesgo para tus primeras implementaciones de puente
2. **Documentar todo**: La documentación clara ayuda a tu equipo a entender la estrategia de migración
3. **Probar exhaustivamente**: Los componentes puente requieren probar ambos lados de la integración
4. **Planificar el futuro**: Cada componente puente debería tener una ruta de migración a OWL puro
5. **Compartir conocimiento**: Ayuda a tu equipo a entender tanto patrones legado como modernos

La Mentalidad del Desarrollador Profesional

Los desarrolladores profesionales de Odoo entienden que la maestría no se trata solo de conocer la tecnología más reciente, se trata de saber cuándo y cómo aplicar la herramienta correcta para cada situación. A veces eso son componentes OWL de vanguardia. A veces son widgets legado confiables. A menudo, es el puente entre ellos.

Las habilidades que has aprendido en este libro te servirán bien mientras Odoo continúa evolucionando. Ahora tienes la base para adaptarte a nuevos frameworks, integrar con tecnologías futuras y, lo más importante, entregar valor a los usuarios mientras mantienes la estabilidad del sistema.

Desafío Final

Al cerrar este libro, aquí tienes un desafío para consolidar tu aprendizaje:

Construye un módulo completo de Odoo que demuestre todo lo que has aprendido: 1. Componentes OWL modernos con hooks y gestión de estado 2. Integración con servicios de Odoo y modelos backend 3. Suite integral de pruebas 4. Integración de puente con al menos un componente legado 5. Documentación profesional y organización de código

Este proyecto servirá como tu pieza de portafolio y prueba de maestría. Más importante aún, te dará la confianza para abordar cualquier desafío de desarrollo OWL que se te presente.

Bienvenido a las filas de desarrolladores profesionales de Odoo. Estás listo para construir el futuro, un componente a la vez.

¡Feliz programación, y que tus puentes sean fuertes y tus migraciones suaves!

TL;DR: Los widgets legado también pueden vivir dentro de componentes OWL (vía `LegacyComponent` + limpieza en `onWillUnmount`), el estado compartido bidireccional necesita un único servicio reutilizable en vez de un truco específico del puente, y todo componente puente debería tener una fecha límite de migración explícita para no volverse permanente.

Pruébalo tú mismo: El ejemplo de este capítulo está disponible como addon instalable de Odoo 19: `simplifyit_owl_book_ch17_2_ex1`. Las instrucciones de instalación están en el README del repositorio.

¿Qué Sigue?

Ya tienes OWL 2.0 cubierto de punta a punta — el capítulo final es un vistazo hacia dónde va el framework después, para que no te tome por sorpresa cuando OWL 3 finalmente llegue.

Fin del Capítulo 17

Capítulo 18: OWL 3 — Lo que Viene (en Alpha)

> **ADVERTENCIA — Software en fase alpha, para conocimiento, no para producción.** Todo lo que describe este capítulo corresponde a **OWL 3**, una versión publicada actualmente como `3.0.0-alpha.45` en el repositorio odoo/owl. El documento oficial de diseño lo dice sin rodeos: *"the design is still a work in progress"* (el diseño todavía está en desarrollo). No hay fecha anunciada para una versión estable. Nada de lo que verás aquí debería usarse hoy en un proyecto real — las APIs que aprendiste en el resto de este libro (OWL 2.0) son las que corren en las versiones actuales de Odoo, incluyendo Odoo 19. Este capítulo existe para que, cuando OWL 3 llegue, nada de esto te tome por sorpresa.

¿Por qué este capítulo? Porque el OWL que acabas de dominar durante diecisiete capítulos no es el estado final — ya hay una reescritura mayor en marcha, y conocer su forma ahora significa que no tendrás que reaprender todo desde cero más adelante.

¿Qué trata de resolver Odoo con esto? La reactividad de OWL 2.0 está atada a las instancias de componente de una forma que limita cómo se puede compartir, derivar u observar el estado fuera de un ciclo de render — OWL 3 es una reconstrucción desde cero pensada para eliminar esa limitación.

Aplicación en la vida real: Todavía ninguna, y ese es justamente el punto — esto es preparación, no una técnica para llevar a un proyecto de cliente. La aplicación realista hoy es reconocer el vocabulario (signals, Plugins, `useProps()`) cuando empiece a aparecer en notas de lanzamiento, para que una futura migración no te tome por sorpresa.

Todo framework, tarde o temprano, reescribe las partes de sí mismo que resultaron limitantes. React lo hizo al pasar de componentes de clase a hooks. Vue lo hizo al pasar de la Options API a la Composition API. OWL está haciendo algo similar ahora con su sistema de reactividad — y el efecto dominó llega hasta las props, los servicios, los hooks de ciclo de vida e incluso el compilador de templates.

Este capítulo es un recorrido guiado por los cambios más grandes, basado directamente en el documento de diseño propio de OWL y en la guía de migración en borrador del repositorio odoo/owl. Piénsalo como un adelanto, no como un tutorial — aquí no construirás nada, solo aprenderás a reconocer lo que viene.

¿Por Qué una Nueva Versión Mayor?

En OWL 2.0, la reactividad se construye sobre `useState`, que envuelve un objeto en un `Proxy` (Capítulo 7). Ese proxy está atado al componente que lo creó: OWL rastrea "este componente leyó esta propiedad, así que hay que volver a renderizarlo cuando cambie". Funciona bien, pero tiene un límite estructural — los valores reactivos son difíciles de compartir, calcular u observar *fuera* del ciclo

de renderizado de un componente.

OWL 3 reconstruye la reactividad alrededor de **signals** (señales): pequeños contenedores reactivos que no están atados a ningún componente en particular. Cualquier cosa que *lea* una signal — el render de un componente, un valor calculado, un efecto — se suscribe automáticamente a ella, sin importar dónde viva ese código. Ese único cambio es la raíz de casi todo lo demás en este capítulo.

El Modelo de Reactividad: De Proxies a Signals

Donde OWL 2.0 te da `useState()` y `reactive()`, OWL 3 te da cuatro bloques de construcción:

- **`signal(valor)`** — un único valor reactivo, de lectura/escritura.
- **`proxy(objeto)`** — el equivalente más cercano al `useState()`/`reactive()` de hoy: un objeto cuyas propiedades están respaldadas cada una por una signal.
- **`computed(() => ...)`** — un valor *derivado* que se recalcula automáticamente cuando cambian las signals que lee, y queda cacheado hasta entonces.
- **`effect(() => ...)`** — ejecuta un callback cada vez que cambian las signals que lee (similar en espíritu a `useEffect`, pero no atado a un componente).

```
// OWL 2.0 (este libro)
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);
}
```

Qué reemplaza esto: el patrón del Capítulo 7 donde calculabas un valor derivado directamente en el template, o lo recalculabas en cada render dentro de un getter. `computed()` cachea el resultado y solo recalcula cuando cambian sus dependencias reales — cercano a cómo funciona la `memoization` (Capítulo 8), pero automático.

Props y Entorno: Un Nuevo Modelo de Inyección

Este es el cambio con mayor radio de impacto. En OWL 2.0, todo componente recibe automáticamente `this.props` y `this.env` (Capítulos 8 y 11). En OWL 3, **ambos dejan de ser miembros implícitos del componente.**

Las props se declaran explícitamente con `useProps()` y un validador construido con el helper `t`:

```
// OWL 2.0 (este libro)
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` y toda la capa de servicios (`useService`, Capítulo 13) se reemplazan por un **sistema de Plugins**: en vez de un objeto de entorno global del que cualquier componente puede sacar servicios,

defines una clase `Plugin` y declaras explícitamente qué componentes dependen de ella.

```
// OWL 3 (alpha) - boceto, la API todavía está en movimiento
class OrmPlugin extends Plugin { /* ... */ }

setup() {
    this.orm = usePlugin(OrmPlugin);
}
```

Por qué te importa esto: todo lo que el Capítulo 13 enseñó sobre `useService("orm")`, `useService("notification")`, etc. se traduce conceptualmente en "inyectar un Plugin" en OWL 3 — el *patrón* (declarar una dependencia, no acceder a un global) sobrevive, pero la API es completamente distinta.

Cambios en el Ciclo de Vida

Tres cambios aquí afectan directamente hooks que ya conoces del Capítulo 10:

- **`onWillUpdateProps` se elimina, sin reemplazo directo.** Según lo que estuvieras haciendo con él, la guía de migración te dirige hacia `computed()` (si derivabas estado a partir de props), `useEffect()` (si ejecutabas un efecto puntual), o un nuevo hook `asyncComputed()` (si cargabas datos según una prop — la prop tiene que ser una signal para que esto funcione).
- **`onPatched`/`onWillPatch` se vuelven más estrictos.** En OWL 2.0 se disparan cuando *cualquier* descendiente se vuelve a renderizar, incluso a través de un slot. En OWL 3 solo se disparan cuando el propio componente se re-renderiza — el re-render de un descendiente ya no burbujea hacia arriba.
- **`this.render()`, `onWillRender` y `onRendered` se eliminan directamente**, sin reemplazo directo listado todavía.

Cambios en el Compilador de Templates

Algunos cambios afectan cómo escribes templates QWeb:

- **Ya no hay variables libres implícitas.** En OWL 2.0 puedes escribir `t-on-click="onClick"` y OWL resuelve `onClick` como `this.onClick`. OWL 3 exige el `this.` explícito: `t-on-click="this.onClick"`.
- **`t-esc` se elimina.** Usa `t-out` para todo — en OWL 3, `t-out` se extendió para manejar de forma segura tanto valores simples como markup, así que cubre lo que antes hacía `t-esc` (Capítulo 6).
- **`t-ref` y `t-model` ahora toman una signal, no un string.** En OWL 2.0 escribes `t-ref="miRef"` y lees `this.miRef.el`; en OWL 3 la ref misma es una signal que creas y pasas.
- **`t-slot` se renombra a `t-call-slot`.** El renombre busca dejar más claro que esta directiva *inserta* el contenido de un slot, a diferencia de `t-set-slot`, que sigue *definiendo* qué recibe un slot (Capítulo 14).

Eventos

`useExternalListener` (el hook que usarías para escuchar en `window` o `document` con limpieza automática) se renombra a **`useListener`**. Funcionalmente es la misma idea: adjuntar un listener que se elimina automáticamente cuando el componente se desmonta.

Las directivas `t-on-*` también ganan un nuevo modificador `.passive` para listeners sensibles al rendimiento como `scroll` o `touchmove` — le indica al navegador que el handler nunca llamará a

`preventDefault()`, lo que le permite optimizar el scroll. Es mutuamente excluyente con `.prevent` por la misma razón.

Qué se Elimina sin Reemplazo Directo

Una lista corta de cosas que este libro cubre (o que existen en OWL 2.0) que la guía de migración actualmente marca como eliminadas, sin un reemplazo 1:1 documentado todavía:

- **`t-portal`** (renderizar fuera del árbol del componente)
- **`useComponent()`** (el hook de bajo nivel para acceder a la instancia del componente actual)
- **`loadFile`**

Si construyes algo con esto hoy, ten en cuenta que una futura migración a OWL 3 requerirá repensar esa parte del código, no solo reemplazar el nombre de una API.

Estado Actual y Cronograma

Al momento de escribir esto, `odoo/owl` está publicando pre-releases alpha (la más reciente al momento de la investigación era `3.0.0-alpha.45`) desde una rama `master` que ya fue reestructurada en paquetes separados (`owl-core`, `owl-compiler`, `owl-runtime`, `owl`). El propio documento `owl3_design.md` afirma que el diseño todavía está evolucionando, y no hay fecha anunciada para una versión estable `3.0.0`. El código de Odoo (la rama `saas-19.4`) ya tiene commits que vendorizan builds alpha de OWL 3 — una señal de que las pruebas internas están en marcha — pero eso no es lo mismo que OWL 3 estando listo para autores de addons.

¿Deberías Aprender OWL 3 Ahora?

Para trabajo de producción, no — todavía no. Todo lo que construiste en los ejemplos de este libro y en los addons de ejemplo es OWL 2.0, y eso es lo que corre en toda versión de Odoo actualmente soportada, incluyendo Odoo 19. Trata este capítulo como un mapa de *hacia dónde van los conceptos*, para que términos como "signals", "Plugins" y `useProps()` no te resulten desconocidos cuando aparezcan en notas de lanzamiento o artículos.

Si quieres profundizar una vez que domines todo lo de los Capítulos 1 a 17, las fuentes primarias son: - `doc/v3/owl/owl3_design.md` en el repositorio odoo/owl (la justificación del diseño) - `doc/v3/owl/migration_owl2_to_owl3.md` en el mismo repositorio (la guía de migración práctica, cambio disruptivo por cambio disruptivo)

Ambos son documentos vivos — espera que cambien antes de que OWL 3 se estabilice.

Resumen

OWL 3 es una reconstrucción desde cero del sistema de reactividad de OWL, pasando de proxies atados a un componente (`useState`) a **signals** independientes (`signal`, `proxy`, `computed`, `effect`). Ese único cambio se propaga hacia cómo se declaran las props (`useProps()` en vez de `static props`), cómo se inyectan los servicios (un sistema de **Plugins** en vez de `env`/`useService`), qué hooks de ciclo de vida existen (`onWillUpdateProps` desaparece, `onPatched`/`onWillPatch` se vuelven más estrictos), e incluso pequeñas reglas de sintaxis de templates (`this.` explícito, `t-esc` desaparece, `t-slot` se renombra a `t-call-slot`). Todo esto sigue en fase **alpha** y explícitamente en desarrollo — lee este capítulo para reconocer la forma de lo que viene, no para empezar a construir con ello.

TL;DR: OWL 3 sigue en alpha (`3.0.0-alpha.45`, sin fecha estable) y reconstruye la reactividad alrededor de signals, reemplazando `useState`/props/servicios/varios hooks en el camino — lee este capítulo para estar al tanto, no para empezar a construir con ello.

¿Qué Sigue?

No hay siguiente capítulo — este es el final del libro. El mejor próximo paso es volver atrás y poner a trabajar los Capítulos 1 al 17: construir algo real con los addons de ejemplo, romperlo, arreglarlo, y que así sea como OWL 2.0 realmente se te quede. Cuando vuelva la curiosidad por OWL 3, `doc/v3/owl/` en el repositorio odoo/owl es donde el diseño sigue evolucionando.

Fin del Capítulo 18 Fin de "OWL 2.0: Donde Termina Odoo y Empiezas Tú"

Sigue Construyendo con OWL

Llegaste al final del libro — JavaScript moderno, cada concepto central de OWL 2.0, un adelanto de lo que viene en OWL 3, y 20 addons de ejemplo instalables para acompañarlo. Acá va hacia dónde seguir.

Ten los Ejemplos a Mano

El ejemplo de cada capítulo vive como un addon instalable de Odoo 19 en el repositorio de ejemplos (`simplifyit_owl_book_ch1_ex1` hasta `simplifyit_owl_book_ch17_2_ex1`) — instala los que necesites y úsalos como referencia de trabajo la próxima vez que te trabes en un proyecto real. Revisa el README del repositorio para las instrucciones de instalación.

Sigue el Contenido

Nuevos capítulos y artículos sobre OWL/Odoo se publican en el blog: **https://www.simplifyit.com.bo/blog**

¿Necesitas Ayuda en un Proyecto Real?

Si estás migrando una instancia legada de Odoo, construyendo un addon a medida, o simplemente atascado en un problema de OWL que este libro no cubrió, Simplify IT S.R.L. hace exactamente este tipo de trabajo.

- Sitio web: **https://simplifyit.com.bo**
- Email: **gm@simplifyit.com.bo**

Gracias por leer — ahora ve a construir algo.