

Blog » Extender sin tocar el núcleo: por qué LibreDTE Lib Core se construye sobre derafu/backbone

Extender sin tocar el núcleo: por qué LibreDTE Lib Core se construye sobre derafu/backbone

Qué le permite derafu/backbone a LibreDTE Lib Core, por qué agregar variantes y formas de acceso es tan simple, y qué habría costado hacerlo con otros diseños.



LibreDTE · 27/10/2026 · #arquitectura

Extender sin tocar el núcleo: por qué LibreDTE Lib Core se construye sobre derafu/backbone

En el [artículo anterior](#) vimos los paquetes de Derafu sobre los que se apoya **LibreDTE Lib Core**, y dijimos que `derafu/backbone` merecía un artículo propio. Este es ese artículo.

La idea central es simple: con Backbone, **agregar un tipo de documento, un formato de entrada, una forma de enviar o una API no obliga a modificar el núcleo**. Y lo mejor es que casi todo eso es poco código. Veamos por qué.

El problema: crecer en dos direcciones

Una biblioteca de facturación electrónica crece en dos ejes a la vez.

Variantes. Hay muchos tipos de documentos (facturas, boletas, guías, notas de crédito y débito...), cada uno con sus reglas. Hay distintos formatos de entrada (JSON, XML, YAML, formularios, archivos de otros sistemas). Hay distintos canales de intercambio (correo, SII). Y todo eso cambia con el tiempo.

Formas de acceso. Quien integra LibreDTE puede estar en PHP, pero también puede necesitar una API HTTP, un comando de consola o una llamada desde Python.

Si cada variante y cada forma de acceso exigen tocar el núcleo, la biblioteca se vuelve frágil: más condicionales, más código repetido y más riesgo de romper algo que ya funcionaba.

Backbone en cinco minutos

`derafu/backbone` es un framework pequeño, de unas 2.700 líneas, que organiza el código en capas con responsabilidades claras:

- **Package:** un dominio completo. Lib Core tiene `billing` (facturación) y `system`.
- **Component:** un área funcional del dominio, como `document`, `exchange` o `integration`.
- **Worker:** quien expone las operaciones al exterior y coordina el trabajo.
- **Job:** una tarea atómica, como pedir un token al SII.
- **Handler:** quien orquesta varios pasos, por ejemplo el envío de un sobre.

- **Strategy:** una variante intercambiable de una operación.

Cada pieza se declara con un **atributo de PHP 8** que dice qué es y dónde va. No hay archivos de configuración largos que mantener sincronizados con el código: Backbone lee los atributos y arma el resto.

Cómo se ve en Lib Core

El resultado, en números redondos: 2 packages, 10 componens, unos 30 workers y jobs cada uno, 3 handlers y **más de 60 strategies**. Casi todas las variantes del sistema son strategies.

El caso más claro son los tipos de documento. Cada uno es una strategy en cada uno de los workers que lo procesan: builder, normalizer, sanitizer y validator. Para el worker parser, que transforma datos de entrada en un documento, cada formato es otra strategy: default.json, default.xml, default.yaml, form.estandar. Para el intercambio, email.smtp envía y email.imap recibe.

¿Cuánto cuesta agregar una variante? Por ejemplo, un parser para el formato de tu ERP es una clase con este aspecto:

```
#[Strategy(name: 'mi_erp.json', worker: 'parser', component: 'document', package:
'billing')]
class MiErpParserStrategy extends AbstractStrategy implements ParserStrategyInterface
{
    public function parse(string $data): array
    {
        // Convertir el formato de tu ERP al arreglo que entiende Lib Core.
    }
}
```

Se declara como servicio con una línea y, desde ese momento, se usa pidiendo `mi_erp.json` como estrategia del parser. No hay que modificar el `ParserWorker`, ni agregar un `if`, ni registrar nada en otro lugar. El worker elige la estrategia por nombre y la ejecuta.

Las alternativas, y lo que habrían costado

Backbone no es la única forma de resolver esto. Estas son otras formas posibles, y lo que cada una obliga a escribir y mantener:

Alternativa	Qué hay que escribir y mantener	Dónde se complica
Condicionales o herencia por tipo	Un <code>switch</code> o una jerarquía de clases que se modifican por cada tipo nuevo.	Cada variante nueva toca el núcleo. Los cambios se mezclan y las pruebas se vuelven globales.

Alternativa	Qué hay que escribir y mantener	Dónde se complica
Registro manual de servicios	Una lista de fábricas o de servicios, escrita a mano, que debe estar al día con el código.	Es fácil agregar la clase y olvidar registrarla. La lista crece y nadie la revisa completa.
Un sistema de plugins o eventos propio	El mecanismo de descubrimiento, las reglas de orden, la documentación y las pruebas del mecanismo mismo.	Hay que mantener un framework, además de la biblioteca.
Código escrito a mano por cada forma de acceso	Un controlador, una ruta, un comando y una conversión de tipos por cada operación y por cada superficie.	Cada operación nueva se multiplica por cada forma de acceso. Las superficies se desincronizan.

Con Backbone, el criterio de qué es una variante, cómo se nombra y cómo se descubre está resuelto una sola vez y vale para todo el código. Eso es lo que reduce el trabajo de crear **y** el de mantener.

Lo que se obtiene sin escribirlo: una fuente, muchas superficies

Como todas las operaciones de Lib Core están declaradas del mismo modo, se pueden descubrir y llamar de forma genérica. De eso se encarga una familia de paquetes construidos sobre Backbone:



- **Dispatcher:** permite invocar y explorar cualquier operación por su identificador, con

parámetros simples, en tres niveles. Desde el directo, que devuelve valores de PHP, hasta el seguro, que nunca lanza excepciones y devuelve fallas estructuradas. Una política decide qué operaciones se pueden llamar desde afuera, y lo recomendado es exponer solo las marcadas con `#[Operation]`.

- **API HTTP:** convierte cada operación en un endpoint, con documentación OpenAPI 3.1, **sin escribir un controlador por operación**.
- **Consola:** convierte cada operación en un comando, invocable desde cualquier lenguaje con `docker exec`.
- **Bridge Python:** permite llamar a las operaciones desde Python.

La evidencia está en el tamaño: la API de Lib Core es un proyecto de dos archivos y menos de cien líneas de PHP, y la consola, de dos archivos y poco más de ciento cincuenta. Todo lo demás lo aportan los paquetes genéricos, que se reutilizan en cualquier proyecto construido con Backbone.

Y cuando se agrega una operación nueva a Lib Core, aparece sola en las cuatro formas de acceso. No hay nada que sincronizar.

Puntos de extensión

En tu propia aplicación

Puedes sumar tus propias strategies a los workers de Lib Core, como en el ejemplo anterior, o agregar tus propios packages con sus componentes. Algunos casos de ejemplo:

- Un formato de entrada específico de tu ERP o de tu sistema de ventas.
- Una nueva forma de entregar los documentos emitidos, como un servicio de almacenamiento o una cola de mensajes.
- Una validación adicional para un tipo de documento, propia de tu negocio.
- Un dominio nuevo con sus operaciones, que se expone a la API y a la consola sin código adicional.

Como producto: LibreDTE Lib Pro

El mejor ejemplo de extensión es **LibreDTE Lib Pro**, la versión privada de la biblioteca, exclusiva de la Edición Enterprise de LibreDTE y de otros proyectos relacionados. Es otro proyecto que:

- usa Lib Core como dependencia y **reutiliza sus contratos y clases base**,
- importa su configuración de servicios,
- arma su **propio registro de paquetes**, con su paquete de facturación y paquetes adicionales para contabilidad y recursos humanos, y toma el paquete `system` de Lib Core tal como está.

Con eso, Pro agrega por ejemplo más formatos de entrada, como formularios específicos de rubros (retail, hospedaje, servicios digitales) y archivos de otros sistemas de facturación, procesamiento por lotes desde plataformas de comercio electrónico, y cargadores de libros desde archivos CSV o XLSX.

Todo son strategies y workers del mismo modelo.

Además, como comparte el modelo, las mismas superficies funcionan sin trabajo extra: Pro tiene su propio Dispatcher y su propio bridge Python, contruidos con las mismas piezas genéricas.

Esto es lo que permite que el núcleo abierto y la versión privada evolucionen por separado sin que una rompa a la otra.

Lo que cuesta

Backbone no es gratis. Conviene decirlo:

- **Más clases y archivos.** Separar en Package, Component, Worker, Job, Handler y Strategy significa más archivos que un diseño plano. A cambio, cada uno hace una sola cosa.
- **Algo de indirección.** Para ver qué ocurre al llamar a una operación hay que saber qué worker la expone y qué strategy se elige.
- **Una curva de aprendizaje.** Hay que conocer el vocabulario. La documentación incluye una guía de decisión para saber cuándo usar un Job, un Handler o una Strategy.
- **Dependencia del contenedor del kernel.** Backbone descubre los servicios a través del contenedor de inyección de dependencias de `derafu/kernel`.

Para una biblioteca que crece en variantes y formas de acceso, esos costos se pagan una vez. Los de las alternativas se pagan en cada cambio.

Lo que viene

Hay varios temas que se pueden profundizar en próximos artículos: cómo escribir tu propia strategy paso a paso, el Dispatcher y sus políticas, la API automática en detalle y el uso de LibreDTE desde Python. **Síguenos para no perdértelos.**

La documentación de Backbone y de los paquetes que lo extienden está en derafu.dev, y el código de LibreDTE Lib Core en [GitHub](https://github.com).

Última actualización el 23/10/2026