Entendiendo el core de Drupal

Arquitectura, librerías y módulos de Drupal

- Componentes de bajo nivel
- Librerías del kernel
- Infraestructura frontend del core
- Estructura física del repositorio core/
- Módulos del core

Autor: CésarMSFelipe (@cesarmsfelipe)
Revisión: Borja Vicente Céspedes (@nireneko)

Índice

Créditos y licencia

Entendiendo el core de Drupal

Autor: **CesarMSFelipe** — Drupal.org · GitHub · LinkedIn

Revisión técnica: **nireneko** — Drupal.org · GitHub · LinkedIn

Primera edición, Agosto de 2026. Contenido verificado contra el código fuente de Drupal 11.4.4.

Licencia

Los ejemplos de código

Los ejemplos de código están para usarse. Puedes incorporarlos a tus proyectos y a la documentación de tu equipo sin pedir permiso y sin atribución. Lo que no se puede reproducir sin autorización es el texto que los explica, que es donde está el trabajo.

Algunos fragmentos citan o adaptan código del core de Drupal, licenciado bajo GPLv2 o posterior; esas partes conservan su licencia original.

Marcas y afiliación

Drupal es una marca registrada de Dries Buytaert. Este libro es un trabajo independiente: no está afiliado a Drupal Association ni cuenta con su respaldo ni con el de los mantenedores del proyecto.

El resto de nombres de productos y empresas que aparecen en el texto son marcas de sus respectivos titulares, y se usan únicamente con fines identificativos y editoriales.

Sobre la exactitud del contenido

El contenido se verificó contra el código fuente de Drupal 11.4.4 en el momento de publicarse. Drupal evoluciona, y una versión menor puede cambiar lo que aquí se describe: comprueba siempre contra la versión que tengas en producción antes de tomar una decisión de arquitectura sobre lo que leas aquí.

El libro se ofrece «tal cual». Ni el autor ni el revisor asumen responsabilidad por daños derivados de su uso.

Prefacio

Ya hay documentación sobre Drupal en castellano. Lo que no hay es una referencia técnica que trate el core como lo que es: un sistema de software con arquitectura propia, contratos explícitos y decisiones de diseño con consecuencias en el código que escribes encima.

La documentación oficial es extensa pero dispersa. Los libros técnicos cubren casos de uso habituales pero rara vez bajan al detalle de cómo funciona el kernel, qué contratos expone cada servicio o qué decisiones de diseño explican el comportamiento del sistema. Este libro cubre ese hueco: una referencia para desarrolladores que ya conocen Drupal y quieren entender su funcionamiento interno.

A quién va dirigido

A **desarrolladores** que trabajan con Drupal a diario y necesitan criterio técnico, no solo procedimientos. A **arquitectos** que tienen que decidir qué capa del core usar para cada problema. A **equipos** que quieren construir un lenguaje técnico común sobre el sistema.

No es un material de introducción. Asume conocimiento previo de Drupal.

Introducción

Drupal es, ante todo, un sistema de gestión de contenido construido sobre un kernel orientado a servicios. Entender Drupal en profundidad significa entender ese kernel: cómo arranca, qué contratos establece, cómo distribuye responsabilidades entre capas y qué herramientas ofrece a quienes lo extienden.

Este estudio organiza el core por capas arquitectónicas reales: parte de la estructura física del repositorio, sube por los subsistemas de bajo nivel, atraviesa los servicios del kernel y llega hasta los módulos funcionales. El recorrido está pensado para que cada pieza tenga contexto antes de que entres en el detalle.

Cubre cómo funciona el sistema por dentro.

> **Diagrama:** el stack de capas de arquitectura en el atlas de arquitectura.

Qué encontrarás

Cada bloque estudia una capa del core como un sistema con diseño propio: sus responsabilidades, sus límites, sus contratos hacia las capas superiores y sus dependencias hacia las inferiores. El análisis no se detiene en «esto hace X»; busca responder «por qué está diseñado así y qué significa para el código que construyes encima».

El punto de entrada importa menos que el criterio que desarrollas recorriendo el sistema completo.

Cómo está organizado

El libro se divide en cinco bloques. Están pensados para leerse en orden la primera vez, aunque cada uno funciona de forma independiente una vez que tienes el mapa completo.

Anatomía del Core

El punto de partida. Antes de profundizar en ningún subsistema, conviene tener claro cómo se organiza el core en disco y qué responsabilidad ocupa cada capa. Esta sección da esa visión de conjunto: estructura física, capas de librerías, infraestructura frontend y módulos funcionales en un único recorrido.

Subsecciones: Librerías del Kernel · Módulos del Core · Assets · Config · DataTypes · Debug · Includes · Profiles · Recipes · Root Files · Scripts · Tests · Themes · Claro · Olivero

---

Drupal Component

El nivel más bajo del stack. `Drupal\Component` reúne utilidades completamente desacopladas del runtime del CMS: pueden usarse en proyectos PHP sin una instalación Drupal completa. Discovery, plugins, eventos, serialización, caché de fichero, utilidades y soporte de infraestructura.

Lo que hace interesante a Component no es que sea reutilizable, sino que ahí viven las decisiones de diseño sobre las que se apoya todo lo demás: cómo se descubren

clases, cómo se organizan los plugins, cómo se gestiona la caché a nivel de fichero. Estudiar Component es estudiar los cimientos.

24 subsistemas: Annotation · Assertion · ClassFinder · Datetime · DependencyInjection · Diff · Discovery · EventDispatcher · FileCache · FileSecurity · FileSystem · FrontMatter · Gettext · Graph · HttpFoundation · PhpStorage · Plugin · ProxyBuilder · Render · Serialization · Transliteration · Utility · Uuid · Version

---

Librerías del Core

El kernel propiamente dicho. `Drupal\Core` contiene los servicios y contratos que hacen funcionar el sistema en tiempo de ejecución: entidades, render, caché, formularios, routing, seguridad, configuración, acceso, plugins, traducción, sesión, cola, base de datos y operación del sistema.

Más de 100 subsistemas organizados por dominio. Si quieres entender por qué Drupal se comporta de una determinada manera ante una solicitud HTTP, la respuesta casi siempre está aquí.

Frontend / JS

La infraestructura frontend nativa del core, en `core/misc`. No es un tema: es la capa que materializa la interactividad del sistema. Behaviors, AJAX, HTMX, estados de formularios, drag-and-drop, diálogos, accesibilidad, iconos y utilidades de interfaz. Todo lo que convierte un render array en una experiencia de usuario funciona a través de esta capa.

16 áreas: Drupal JS API · Ajax · HtmxAdvanced · FormsAdvanced · Autocomplete · States · Tabledrag · Dialog · Dropbutton · Messages · Accessibility · UIComponents · Icons · Utilities · Print · Batch

---

Módulos del Core

La capa funcional. 76 capítulos de módulos organizados por dominio: contenido, medios, maquetación, navegación, internacionalización, rendimiento, APIs, configuración, editorial, migración y drivers de base de datos.

Lo que distingue este análisis de una lectura superficial de los módulos es el enfoque: cada módulo se estudia como una pieza de arquitectura — sus dependencias internas, sus contratos de extensión y los límites donde termina su responsabilidad y empieza la del kernel.

Cómo usarlo

La primera lectura debería seguir el orden: Anatomía → Component → Core → Frontend → Módulos. Eso construye el contexto necesario para que el detalle tenga sentido.

En el uso cotidiano, cada sección funciona como referencia independiente. El sistema de referencias cruzadas está pensado precisamente para que puedas

entrar por cualquier punto y encontrar el contexto que necesitas sin tener que releer el libro completo.

Panorama general del libro

Este libro documenta el núcleo de Drupal — el kernel, sus contratos, sus capas y sus módulos — al nivel de detalle que necesita un desarrollador que ya trabaja con Drupal y quiere entender cómo funciona por dentro.

No es un tutorial. No cubre instalación ni administración. Es una referencia técnica organizada por capas arquitectónicas reales, no por la taxonomía de drupal.org. Cada capítulo responde a «qué hace esta pieza del core y por qué está diseñada así», con ejemplos de uso, hooks con atributos PHP 8 (`#[Hook]`, procedurales solo donde el core aún no admite atributos, como los hooks de instalación y actualización), y antipatrones documentados con código.

Las cinco capas

Capa	Qué documenta	Entrada
Anatomía del Core	Estructura física del repositorio, perfiles, tests, assets, config, scripts	Ir →
Drupal\Component	24 subsistemas desacoplados del runtime: discovery, plugins, eventos, serialización	Ir →
Drupal\Core	El kernel de runtime: +100 servicios y contratos del CMS en ejecución	Ir →
Frontend / core/misc	Infraestructura JS/CSS nativa: behaviors, AJAX, HTMX, estados, diálogos	Ir →
Módulos del Core	75 módulos funcionales organizados por dominio	Ir →

Lectura recomendada

Si es tu primera lectura, sigue el orden por capas que detalla la Introducción. Si ya conoces la estructura, entra directamente por el capítulo que necesites y sigue las referencias cruzadas.

Puertas de entrada

- Prefacio — para qué existe este libro y a quién va dirigido.
- Introducción — cómo está organizado el core por capas y cómo recorrerlo.
- Epílogo — el cierre: qué criterio deberías llevarte después de recorrer el sistema.

Herramientas de navegación

- Glosario de conceptos arquitectónicos — 130+ términos con definiciones precisas y links a los capítulos donde se aplican.
- Mapas de arquitectura (Mermaid) — 40 diagramas de los flujos centrales, agrupados por capa: los que atraviesan todo el sistema, `Drupal\Component`, el kernel, el Frontend y los módulos del core.
- Índice por perfil de lector — capítulos agrupados por rol: arquitecto, desarrollador backend, tech lead.

Cache

El modelo de caché de la mayoría de frameworks es simple: almacena el resultado de una operación durante N segundos y luego vuelve a calcularlo. Drupal rechaza ese modelo casi por completo. El namespace `Drupal\Core\Cache` implementa algo cualitativamente diferente: un grafo de dependencias donde cada fragmento cacheado declara de qué datos depende, y el sistema invalida exactamente las entradas afectadas en el momento en que esos datos cambian.

El resultado práctico es que un bloque que muestra el Nodo 5 no necesita un TTL — necesita el tag `node:5`. Cuando alguien edita ese nodo, Drupal invalida todos los elementos con ese tag de forma inmediata, sin esperar a que expire ningún reloj.

Este diseño tiene una consecuencia importante para el desarrollo: la caché en Drupal no es algo que se configura externamente y se olvida. Es un contrato que cada componente firma al declarar su metadata. Entender ese contrato — qué son los tags, qué son los contexts, qué es el max-age y cómo se propaga la metadata a través del render pipeline — es lo que separa el código que funciona en desarrollo del código que escala en producción.

La trinidad de la cacheabilidad (tags, contexts, max-age) se estudia en detalle en la sección dedicada al Render Pipeline. Este capítulo cubre la infraestructura subyacente: los backends donde viven los datos, los bins que los organizan, y la API que los manipula.

> **Diagrama:** los bins, backends y chained backend y el modelo de cacheabilidad: propagación de tags y contexts en el atlas de arquitectura.

Anatomía

La trinidad de la cacheabilidad

Toda operación que genere datos o marcado debe declarar tres dimensiones de metadatos (`CacheableMetadata`):

1. **Cache Tags (Invalidación)**: Responden a "¿Qué datos ha usado este elemento?". Si un bloque muestra el Nodo 5, el tag es `node:5`. Al editar el nodo, el sistema invalida automáticamente todos los elementos vinculados a ese tag.
2. **Cache Contexts (Variaciones)**: Responden a "¿Para quién o bajo qué condiciones varía este contenido?". Contextos comunes incluyen `languages`, `user.permissions` o `url.query_args`. Desde 11.4 existe además un cache context para el status code de la excepción HTTP (CR 3580452), pensado para cachear respuestas de error de forma granular.
3. **Cache Max-age (Expiración)**: Responde a "¿Durante cuánto tiempo es válido este dato?". Es el fallback para dependencias temporales que no se pueden expresar como tags. Por defecto los ítems son permanentes: `set()` usa `Cache::PERMANENT` (valor `-1`) como `$expire`, así que un ítem sin max-age explícito nunca expira por tiempo — solo desaparece por invalidación de tags o borrado.

Cache Bins y Backends

Un bin es un espacio lógico de almacenamiento con semántica específica — en la práctica, los bins funcionan como categorías de caché. El core define entre otros `bootstrap`, `config`, `default`, `entity`, `menu`, `render`, `data` y `discovery`, y varios módulos aportan el suyo: `page` (Page Cache), `dynamic_page_cache`, `toolbar` o los bins de JSON:API. El bin `render` almacena fragmentos HTML con metadata de cacheabilidad; el bin `data` es de propósito general para módulos.

El backend es la implementación física: `database` (tabla `cache_*` en la BD, por defecto), `APCu` (memoria compartida del proceso PHP, muy rápido para datos pequeños), `php` (`PhpBackend`, archivos PHP en disco vía PhpStorage), `memory` (solo en-request, no persistente), y `chainedfast` (`ChainedFastBackend`, combina un backend rápido y volátil con uno persistente: sirve del rápido y, en fallo, cae al persistente propagando el valor hacia arriba). En producción con alto tráfico, configurar `render` sobre un backend rápido (Redis, Memcache o APCu) es la mejora de rendimiento más directa disponible.

El límite de filas del DatabaseBackend

Con el `DatabaseBackend` por defecto, cada bin está limitado a 5.000 filas (`DatabaseBackend::DEFAULT_MAX_ROWS`). La recolección de basura conserva solo las N filas más recientes según el timestamp `created` — un FIFO puro por momento de escritura, que ignora con qué frecuencia o cuán recientemente se leyó una entrada. La consecuencia: un bin que supera el límite de forma constante entra en churn — Drupal expulsa entradas que siguen calientes y las regenera una y otra vez, multiplicando lecturas y escrituras en la BD. El límite existe para evitar que las tablas de caché crezcan hasta los gigabytes, y es configurable en `settings.php`:

```
$settings['database_cache_max_rows']['default'] = 5000;
$settings['database_cache_max_rows']['bins']['render'] = 50000;
// -1 = sin límite (DatabaseBackend::MAXIMUM_NONE).
```

11.4 añade un bin nuevo: `cache.file_parsing` (CR 3593451). A diferencia de otros bins que se purgan con `drush cr`, este está diseñado para resultados de parseo de archivos que deben sobrevivir entre deploys — el parseo de plugins, schemas, rutas. Usa escritura diferida: agrupa las escrituras para no saturar el storage cuando se parsean muchos archivos a la vez.

Variation Cache

API especializada para cachear respuestas con múltiples variaciones de contextos. Es la base del Dynamic Page Cache: almacena respuestas indexadas por contextos activos, y resuelve la entrada correcta para el usuario actual sin calcular la respuesta completa. Disponible como servicio `variation_cache_factory` (más las instancias `variation_cache.*`).

En práctica

La interacción con la caché requiere el bin específico correcto y el servicio de invalidación adecuado. No todos los bins tienen el mismo backend ni la misma semántica.

```php
// Inyección del bin específico
$cache_bin = \Drupal::cache('data');

// Recuperación con validación de existencia
if ($cache = $cache_bin->get($cid)) {
  return $cache->data;
}

// Persistencia con declaración de dependencias
$cache_bin->set($cid, $complex_data, Cache::PERMANENT, ['my_custom_tag']);
```

Invalidación por tags

El servicio `cache_tags.invalidator` es el punto de entrada correcto para invalidar por tags de forma global. Cuando invalidas por aquí, todos los backends configurados para esos tags sincronizan su estado.

```php
use Drupal\Core\Cache\Cache;

// Invalidar todos los elementos que dependen del Nodo 5
\Drupal::service('cache_tags.invalidator')
  ->invalidateTags(['node:5', 'node_list']);

// Invalidar un tag custom
Cache::invalidateTags(['mymodule_announcements']);
```

Propagar metadata en render arrays

```php
use Drupal\Core\Cache\CacheableMetadata;

$build = ['#markup' => $this->t('...')];

// Forma correcta: propagar la cacheabilidad del objeto
$cacheable_metadata = CacheableMetadata::createFromObject($entity);
$cacheable_metadata->applyTo($build);
```

A partir de 11.4, el header `X-Drupal-Dynamic-Cache` en respuestas 4xx y 5xx corrige su valor (CR 3584640): en lugar del engañoso `UNCACHEABLE (poor cacheability)`, ahora informa el código de estado real, por ejemplo `UNCACHEABLE (404)`. El header ya se emitía antes en esas respuestas de error (cuando `http.response.debug_cacheability_headers` está activo); lo que cambió es el valor, no su aparición. Esto hace que el workflow de inspección de headers que usas en páginas normales sea directamente útil para diagnosticar la caché de 404s y 403s.

Para respuestas que nunca deben almacenarse completas existe el servicio `page_cache_kill_switch`: llamar a `\Drupal::service('page_cache_kill_switch')->trigger()` marca la respuesta actual como no cacheable a nivel de página. La response policy `KillSwitch` está registrada tanto contra Page Cache como contra Dynamic Page Cache, así que un solo disparo desactiva ambas para esa petición. Es un último recurso: cuando la variación se puede declarar, la metadata de caché correcta (max-age 0, contexts) es preferible al kill switch.

Hooks

hook_cache_flush

Invocado durante la limpieza global de cachés. El punto de extensión correcto para purgar sistemas externos no integrados nativamente en el backend de Drupal.

```
namespace Drupal\mymodule\Hook;

use Drupal\Core\Hook\Attribute\Hook;
use Drupal\redis\ClientFactory;
// El servicio redis.factory es de clase Drupal\redis\ClientFactory
// (que NO implementa ClientInterface). Se inyecta y se obtiene el
// cliente conectado con el método de instancia getClient().

class CacheHooks {

  public function __construct(
    private readonly ClientFactory $redisFactory,
  ) {}

  #[Hook('cache_flush')]
  public function cacheFlush(): void {
    $redis = $this->redisFactory->getClient();
    $redis->flushDb();
  }
}
```

hook_preprocess_HOOK

Añade metadatos de caché adicionales a un elemento antes de que sea procesado por el Renderer. Útil para propagar tags de entidades que se cargan en el preprocess.

```
namespace Drupal\mymodule\Hook;

use Drupal\Core\Hook\Attribute\Hook;

class CacheHooks {

  #[Hook('preprocess_node')]
  public function preprocessNode(array &$variables): void {
    /** @var \Drupal\node\NodeInterface $node */
    $node = $variables['node'];
    $variables['#cache']['tags'] = array_merge(
      $variables['#cache']['tags'] ?? [],
      $node->getCacheTags()
    );
  }
}
```

Estándares de Ingeniería

Usar `Cache::PERMANENT` con tags apropiados es preferible a TTLs arbitrarios. La expiración por tiempo es aceptable como fallback para datos con caducidad natural — precios, disponibilidad, sesiones — pero no como estrategia general. Un elemento con TTL de 5 minutos puede servir datos obsoletos 4:59 minutos después de que cambiaron; un elemento con tags correctos se invalida en el instante del cambio.

Los objetos que implementan `CacheableDependencyInterface` ya saben qué tags, contexts y max-age les corresponden. Usar `addCacheableDependency($object)` en lugar de extraer tags manualmente evita errores y se adapta automáticamente cuando el objeto cambia sus metadatos de caché.

La propagación automática de metadata funciona en el render tree: los tags de los hijos ascienden a los padres. Este mecanismo solo funciona si los componentes declaran correctamente sus propios metadatos. Un componente que omite sus tags rompe la cadena de propagación para todo el árbol que lo contiene, lo que provoca que páginas enteras permanezcan cacheadas cuando deberían haberse invalidado.

Antipatrones frecuentes

Invalidar con `deleteAll()` o `invalidateAll()` como respuesta a cambios de contenido *Síntoma:* cuando se publica un nodo, el código ejecuta `\Drupal::cache('render')->invalidateAll()` para asegurarse de que la caché refleja el cambio. *Consecuencia:* todos los fragmentos cacheados en el bin de render — incluyendo páginas que no tienen ninguna relación con el nodo publicado — se invalidan. El siguiente tráfico regenera toda la caché desde cero, con el coste de CPU y BD correspondiente. *Causa raíz:* `invalidateAll()` "garantiza" que el cambio se vea, y esa certeza aparente es lo que lo hace tentador — a costa de tirar toda la caché por un cambio localizado. Los tags existen precisamente para invalidar con precisión: `invalidateTags(['node:5'])` toca solo lo que depende del Nodo 5. `deleteAll()`/`invalidateAll()` son la operación de emergencia para un sistema en mal estado, no una estrategia de invalidación.

Componentes sin metadata de caché en código custom *Síntoma:* un servicio o bloque custom construye un render array con datos de entidades pero no declara ningún tag, context ni max-age. *Consecuencia:* el sistema cachea el output como si fuera invariante. Cuando la entidad subyacente cambia, el fragmento permanece obsoleto hasta que alguien borra la caché manualmente. *Causa raíz:* en Drupal, la ausencia de metadata de caché significa "cacheable sin condiciones y para siempre". No es el comportamiento por defecto más seguro — es el más agresivo. La metadata correcta es una responsabilidad del código que construye el render array, no del sistema de caché.

Módulos que consumen esta librería

- announcements_feed — cachea las respuestas del feed para evitar peticiones repetidos
- big_pipe — los placeholders son elementos con `max-age: 0` que el sistema de caché identifica y BigPipe reemplaza
- block — cada bloque declara sus propios cache tags y contexts en `build()`
- dynamic_page_cache — caché de render arrays con contexts completos; depende de la correcta declaración de cache metadata

- node — cache tags `node:{nid}` y `node_list` que Node declara automáticamente
- page_cache — caché de página completa para usuarios anónimos; consume directamente los backends de caché del kernel
- taxonomy — cache tags `taxonomy_term:{tid}` y `taxonomy_term_list` para invalidación automática
- views — las vistas tienen caché propia configurable con tags y contexts por vista **Capa inferior:** Drupal\Component | **Índice del bloque:** Librerías del Core

DrupalKernel

`Drupal\Core\DrupalKernel` es el punto de entrada del framework. Es la clase que transforma un script PHP vacío en un sistema Drupal completamente funcional: descubre y carga extensiones, compila el contenedor de servicios de Symfony, procesa la petición HTTP y devuelve una respuesta. Todo lo que ocurre en un request de Drupal pasa por aquí. Es la implementación concreta del contrato DrupalKernelInterface, que es lo que Drush y los tests manipulan cuando controlan el bootstrap sin acoplarse a esta clase.

El bootstrap de Drupal está dividido en fases, cada una de las cuales añade capacidad al sistema. Las fases tempranas inicializan lo mínimo necesario para leer la configuración de entorno; las tardías cargan el contenedor completo. Esta arquitectura permite que el sistema falle de forma elegante y que herramientas como Drush puedan inicializar solo las fases que necesitan.

En el modelo de despliegue moderno, el contenedor de servicios se compila una vez y se cachea como una definición serializada en el backend `cache.container` (por defecto la tabla `cache_container` de la base de datos vía `Drupal\Core\Cache\DatabaseBackend`). En cada request, `DrupalKernel` recupera esa definición cacheada e instancia el contenedor sin recompilar, reduciendo el coste del bootstrap a una sola lectura. Reconstruirlo explícitamente (`drush cr`) solo es necesario cuando cambia la configuración de servicios o se instalan módulos nuevos.

Diagrama: las fases de bootstrap del kernel en el atlas de arquitectura.

Anatomía

Fases de bootstrap

`DrupalKernel` inicializa el sistema en fases secuenciales. La fase de configuración lee `settings.php`; la fase de kernel crea el contenedor; fases posteriores cargan la sesión, el idioma y la ruta. Herramientas externas como Drush pueden bootstrapear hasta la fase necesaria sin pagar el coste de las fases posteriores.

Compilación y dump del contenedor

El contenedor de servicios se compila con `ContainerBuilder` y luego se serializa a un array de definiciones mediante `OptimizedPhpArrayDumper` (`Drupal\Component\DependencyInjection\Dumper\OptimizedPhpArrayDumper`), no con el `PhpDumper` de Symfony. Ese array (`$dumper->getArray()`) incluye todas las definiciones de servicios, sus dependencias y los parámetros, y se guarda en el backend `cache.container` mediante `cacheDrupalContainer()`. En el siguiente request, `getCachedContainerDefinition()` lo recupera y `Drupal\Core\DependencyInjection\Container` lo consume directamente, sin recompilación.

Handle y terminate

`DrupalKernel::handle(Request $request)` es el método principal: construye el contenedor si es necesario, resuelve la petición a través del router y los event subscribers, y devuelve la `Response`. `terminate()` se invoca después de que

la respuesta es enviada: delega en el `http_kernel` de Symfony si implementa `TerminableInterface`, y recorre el parámetro `kernel.destructable_services` para llamar `destruct()` sobre cada servicio `DestructableInterface` que se haya inicializado durante el request. Desde 11.4, quien invoca este par ya no es `index.php` directamente, sino el runner de `symfony/runtime` (ver «En práctica»).

En práctica

```php
// index.php en 11.4+ — el front controller real del core.
// (No es código custom — es el archivo que sirve todas las peticiones web.)
use Drupal\Core\DrupalKernel;

require_once 'autoload_runtime.php';

return static function () {
  return new DrupalKernel('prod', require 'autoload.php');
};
```

Desde 11.4, `index.php` ya no ejecuta el ciclo de request: devuelve una closure que construye el kernel, y el componente `symfony/runtime` hace el resto. La cadena es corta. `autoload_runtime.php` (en la raíz del sitio) fija `Drupal\Core\Runtime\DrupalRuntime` como runtime por defecto vía la variable `APP_RUNTIME` y delega en el `vendor/autoload_runtime.php` que genera el plugin de Composer del componente. `DrupalRuntime` extiende `SymfonyRuntime` ajustando los defaults al mundo Drupal: desactiva el error handler de Symfony (Drupal registra los suyos), desactiva la carga de `.env` y fija `prod` como entorno cuando `APP_ENV` no está definida. Con el kernel que devuelve la closure, el runner del componente (`HttpKernelRunner`) ejecuta el ciclo que hasta 11.3 vivía literalmente en `index.php`:

```php
// El ciclo clásico — hasta 11.3 era el cuerpo de index.php; desde 11.4
// lo ejecuta el runner de symfony/runtime con el kernel devuelto.
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

El resultado práctico es la separación formal entre bootstrap (construir el kernel, cargar el container) y handling de la request: `symfony/runtime` gestiona el ciclo de vida del proceso PHP, y `DrupalKernel` se limita a construir el container y delegar en el `http_kernel`. Los front controllers con el formato clásico siguen funcionando sin cambios «por el momento»: el CR no anuncia plazo de retirada, pero presenta la adopción del nuevo patrón como el camino recomendado para scripts custom que bootstrapean Drupal directamente (CR 3553275).

```php
// En un service provider, alterar el contenedor durante compilación.
// DrupalKernel invoca esto antes de serializar el contenedor a PHP estático.
use Drupal\Core\DependencyInjection\ContainerBuilder;
use Drupal\Core\DependencyInjection\ServiceProviderInterface;

final class MyModuleServiceProvider implements ServiceProviderInterface {
  public function register(ContainerBuilder $container): void {
    // Registrar servicios o parámetros durante la compilación.
    $container->setParameter('my_module.env', getenv('APP_ENV') ?: 'prod');
```

```
  }
}
```

Integración con eventos del kernel

`DrupalKernel` no define hooks de módulo en este punto. Para intervenir en el ciclo de request, el mecanismo correcto es suscribirse a eventos Symfony del kernel (`REQUEST`, `RESPONSE`, `TERMINATE`, etc.).

```php
namespace Drupal\mymodule\EventSubscriber;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;
use Symfony\Component\HttpKernel\Event\RequestEvent;
use Symfony\Component\HttpKernel\KernelEvents;

final class KernelRequestSubscriber implements EventSubscriberInterface {

  public static function getSubscribedEvents(): array {
    return [
      KernelEvents::REQUEST => ['onRequest', 100],
    ];
  }

  public function onRequest(RequestEvent $event): void {
    if (!$event->isMainRequest()) {
      return;
    }

    // Lógica de inicialización ligera para el request principal.
  }
}
```

Estándares de Ingeniería

No extender `DrupalKernel`. `DrupalKernel` es una clase interna del core. Extenderla para personalizar comportamiento rompe las actualizaciones cuando Drupal cambia la clase. Usar service providers para alterar el contenedor.

La configuración del entorno pertenece a `settings.php`. Las diferencias entre entornos (desarrollo/staging/producción) deben expresarse en `settings.php` o variables de entorno, no en código PHP condicional.

`DRUPAL_TEST_IN_CHILD_SITE` y constantes similares son para infraestructura de tests. No usar estas constantes para lógica de negocio en módulos.

Antipatrones frecuentes

Extender `DrupalKernel` para personalizar el bootstrap

Síntoma: el proyecto tiene `class MyKernel extends DrupalKernel` con métodos sobrescritos. *Consecuencia:* cada actualización de Drupal que cambie

`DrupalKernel` (firma de métodos, lógica interna) requiere actualizar la subclase custom. Las actualizaciones mayores son mucho más costosas. *Causa raíz:* `DrupalKernel` no forma parte del API extensible; su firma interna cambia entre versiones sin previo aviso porque el core no lo trata como punto de extensión. Cada gancho legítimo tiene su vía propia: `ServiceProviderInterface` para el contenedor, `settings.php` para el entorno, event subscribers para el request.

```
// ✗ Extender DrupalKernel
class MyCustomKernel extends DrupalKernel {
  protected function initializeSettings(Request $request): void {
    parent::initializeSettings($request);
    // Lógica custom — se romperá en actualizaciones
  }
}

// ✓ Service provider para alterar el container
final class MyModuleServiceProvider extends ServiceProviderBase {
  public function alter(ContainerBuilder $container): void {
    if ($container->hasDefinition('renderer')) {
      $container->getDefinition('renderer')
        ->setClass(MyCustomRenderer::class);
    }
  }
}
```

Módulos que consumen esta librería

- system — el módulo system interactúa con el kernel en bootstrap, rebuild y mantenimiento **Capa inferior:** Drupal\Component | **Índice del bloque:** Librerías del Core

El libro completo

Esta muestra recoge 6 de los 254 capítulos de *Entendiendo el core de Drupal*. El resto del libro sigue el mismo criterio: leer el código del core y explicar por qué está escrito así.

Los capítulos marcados con ✓ son los que acabás de leer.

Preliminares

- Créditos y licencia ✓
- Prefacio ✓
- Agradecimientos
- Introducción ✓
- Panorama general del libro ✓

Anatomía del Core

- Anatomía del Core
- Librerías del Kernel
- Módulos del Core
- Assets
- Config
- Data Types
- Includes
- Debug
- Profiles
- Recipes
- Root Files
- Scripts
- Tests
- Themes
- Tema Claro
- Tema Olivero
- Tema Default Admin

Drupal Component

- Drupal Component
- Annotation
- Assertion
- ClassFinder
- Datetime
- DependencyInjection
- Diff
- Discovery
- EventDispatcher
- FileCache
- FileSecurity
- FileSystem
- FrontMatter

- Gettext
- Graph
- HttpFoundation
- PhpStorage
- Plugin
- ProxyBuilder
- Render
- Serialization
- Transliteration
- Utility
- Uuid
- Version

Librerías del Core

- Librerías del Core
- Access
- Accessibility
- Action
- Ajax
- Annotation (Librerías del Core)
- Archiver
- Asset
- Authentication
- Batch
- Block
- Breadcrumb
- Cache ✓
- ClassLoader
- Command
- Composer
- CoreServiceProvider
- Cron
- CronInterface
- Condition
- Config (Librerías del Core)
- Controller
- Database
- Datetime (Librerías del Core)
- DefaultContent
- DependencyInjection (Librerías del Core)
- DestructableInterface
- Dialog
- Diff (Librerías del Core)
- Discovery (Librerías del Core)
- Display
- DrupalKernel ✓
- DrupalKernelInterface
- Entity
- EventSubscriber
- Executable
- Extension
- Field
- File

- FileTransfer
- FinalCore
- Flood
- Form
- FormsAdvanced
- General
- GeneratedButton
- GeneratedLink
- GeneratedNoLink
- GeneratedUrl
- Hook
- Htmx
- HtmxAdvanced
- Http
- Icons
- Image
- ImageToolkit
- Installer
- KeyValueStore
- Language
- Layout
- Link
- Locale
- Lock
- Logger
- Mail
- Mailer
- Menu
- Messenger
- PageCache
- Pager
- ParamConverter
- Password
- Path
- PathProcessor
- PhpStorage (Librerías del Core)
- Plugin (Librerías del Core)
- PreWarm
- PrivateKey
- ProxyBuilder (Librerías del Core)
- ProxyClass
- Queue
- Recipe
- Render (Librerías del Core)
- RouteProcessor
- Routing
- RoutingForm
- Security
- Serialization (Librerías del Core)
- Session
- Site
- StackMiddleware
- State
- States
- StreamWrapper

- StringTranslation
- Tabledrag
- Template
- TempStore
- Test
- Theme
- Token
- Transliteration (Librerías del Core)
- TypedData
- UIComponents
- Update
- Updater
- Url
- Utility (Librerías del Core)
- Validation

Frontend / JS (core/misc)

- JS (core/misc)
- Accessibility (JS (core/misc))
- Ajax (JS (core/misc))
- Autocomplete
- Batch (JS (core/misc))
- Dialog (JS (core/misc))
- Dropbutton
- Drupal JS API
- FormsAdvanced (JS (core/misc))
- HtmxAdvanced (JS (core/misc))
- Icons (JS (core/misc))
- Print y Activos Estáticos
- States (JS (core/misc))
- Tabledrag (JS (core/misc))
- UIComponents (JS (core/misc))
- Messages
- Utilities

Módulos del Core

- Módulos del Core / Inicio
- Announcements Feed
- Automated Cron
- Ban
- Basic Auth
- Big Pipe
- Block (Módulos del Core)
- Block Content
- Breakpoint
- CKEditor 5
- Comment
- Config (Módulos del Core)
- Config Translation
- Contact
- Content Moderation

- Content Translation
- Contextual
- Datetime (Módulos del Core)
- Datetime Range
- DBLog
- Dynamic Page Cache
- Editor
- Field (Módulos del Core)
- Field Layout
- Field UI
- File (Módulos del Core)
- Filter
- Help
- History
- Image (Módulos del Core)
- Inline Form Errors
- JSON API
- Language (Módulos del Core)
- Layout Builder
- Layout Discovery
- Link (Módulos del Core)
- Locale (Módulos del Core)
- Media
- Media Library
- Menu Link Content
- Menu UI
- Migrate
- Migrate Drupal
- Migrate Drupal UI
- MySQL
- MySQLi
- Navigation
- Node
- Options
- Package Manager
- Page Cache
- Path (Módulos del Core)
- Path Alias
- PostgreSQL
- Phpass
- Responsive Image
- REST
- SDC
- Search
- Serialization (Módulos del Core)
- Settings Tray
- Shortcut
- SQLite
- Syslog
- System
- Taxonomy
- Telephone
- Text
- Toolbar
- Update (Módulos del Core)

- User
- Views
- Views UI
- Workflows
- Workspaces
- Workspaces UI
- Node + Field

Preliminares

- Epilogo

Referencia

- Glosario
- Mapas de arquitectura
- Índice por rol