Understanding the core of Drupal

Architecture, libraries, and modules of Drupal

- Low-level components
- Kernel libraries
- Frontend infrastructure of the core
- Physical structure of the core repository
- Core modules

Author: CésarMSFelipe (@cesamsfelipe)

Reviewer: Borja Vicente Céspedes (@nireneko)

Contents

Credits and license

Understanding Drupal core

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

Technical review: **nireneko** — Drupal.org · GitHub · LinkedIn

First edition, August 2026. Content verified against the Drupal 11.4.4 source code.

License

The code examples

The code examples are there to be used. You can put them into your own projects and your team's documentation without asking permission and without attribution. What can't be reproduced without permission is the text that explains them, which is where the work is.

Some snippets quote or adapt code from Drupal core, licensed under GPLv2 or later; those parts keep their original license.

Trademarks and affiliation

Drupal is a registered trademark of Dries Buytaert. This book is independent work: it is not affiliated with the Drupal Association, and it is endorsed neither by the Association nor by the project's maintainers.

The other product and company names that appear in the text are trademarks of their respective owners, and are used for identification and editorial purposes only.

On the accuracy of the content

The content was verified against the Drupal 11.4.4 source code at the time of publication. Drupal evolves, and a minor version can change what is described here: always check against the version you have in production before making an architecture decision on what you read here.

The book is provided "as is". Neither the author nor the reviewer accepts liability for damages arising from its use.

Preface

There is already Drupal documentation. What there isn't is a technical reference that treats core as what it is: a software system with its own architecture, explicit contracts, and design decisions that have consequences for the code you write on top of it.

The official documentation is extensive but scattered. Technical books cover the usual use cases, but they rarely get down into how the kernel works, what contracts each service exposes, or which design decisions explain the system's behavior. This book fills that gap: a reference for developers who already know Drupal and want to understand how it works underneath.

Who this is for

Developers who work with Drupal daily and need technical judgment, not just procedures. **Architects** who have to decide which layer of core to use for each problem. **Teams** that want to build a shared technical language for the system.

This is not introductory material. It assumes you already know Drupal.

Introduction

Drupal is, first and foremost, a content management system built on a service-oriented kernel. Understanding Drupal in depth means understanding that kernel: how it boots, what contracts it establishes, how it distributes responsibilities across layers, and what tools it offers to the people who extend it.

This study organizes core by its real architectural layers: it starts from the physical structure of the repository, climbs through the low-level subsystems, crosses the kernel services, and reaches the functional modules. The route is laid out so that every piece has context before you go into the detail.

It covers how the system works underneath.

> **Diagram:** the architecture layer stack in the architecture atlas.

What you'll find

Each part studies a layer of core as a system with its own design: its responsibilities, its boundaries, its contracts toward the layers above and its dependencies on the ones below. The analysis doesn't stop at "this does X"; it sets out to answer "why is it designed this way, and what does that mean for the code you build on top of it".

The entry point matters less than the judgment you develop by working through the complete system.

How it's organized

The book is divided into five parts. They are meant to be read in order the first time, though each one works independently once you have the complete map.

Core anatomy

The starting point. Before going deep into any subsystem, it's worth being clear about how core is organized on disk and what responsibility each layer holds. This part gives that overall view: physical structure, library layers, frontend infrastructure, and functional modules in a single pass.

Subsections: Kernel libraries · Core modules · Assets · Config · DataTypes · Debug · Includes · Profiles · Recipes · Root Files · Scripts · Tests · Themes · Claro · Olivero

---

Drupal Component

The lowest level of the stack. `Drupal\Component` gathers utilities completely decoupled from the CMS runtime: they can be used in PHP projects without a full Drupal installation. Discovery, plugins, events, serialization, file cache, utilities, and infrastructure support.

What makes Component interesting is not that it's reusable, but that it is where the design decisions everything else rests on actually live: how classes are discovered, how plugins are organized, how caching is handled at the file level. Studying Component is studying the foundations.

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

---

Core libraries

The kernel proper. `Drupal\Core` holds the services and contracts that make the system run at runtime: entities, render, cache, forms, routing, security, configuration, access, plugins, translation, session, queue, database, and system operation.

More than 100 subsystems organized by domain. If you want to understand why Drupal behaves a certain way in response to an HTTP request, the answer is almost always here.

Frontend / JS

Core's native frontend infrastructure, in `core/misc`. It is not a theme: it's the layer that materializes the system's interactivity. Behaviors, AJAX, htmx, form states, drag-and-drop, dialogs, accessibility, icons, and interface utilities. Everything that turns a render array into a user experience runs through this layer.

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

---

Core modules

The functional layer. 76 module chapters organized by domain: content, media, layout, navigation, internationalization, performance, APIs, configuration, editorial, migration, and database drivers.

This analysis differs from a surface reading of the modules in the angle: each module is studied as a piece of architecture — its internal dependencies, its extension contracts, and the boundaries where its responsibility ends and the kernel's begins.

How to use it

A first read should follow the order: Anatomy → Component → Core → Frontend → Modules. That builds the context the detail needs to make sense.

In day-to-day use, each part works as an independent reference. The cross-reference system exists precisely so that you can come in at any point and find the context you need without having to reread the whole book.

The book at a glance

This book documents Drupal core — the kernel, its contracts, its layers, and its modules — at the level of detail a developer who already works with Drupal needs in order to understand how it works from the inside.

It isn't a tutorial. It doesn't cover installation or administration. It's a technical reference organized by the real architectural layers, not by drupal.org's taxonomy. Every chapter answers "what does this piece of core do, and why is it designed this way", with usage examples, hooks written as PHP 8 attributes (`#[Hook]`, procedural only where core still doesn't accept attributes, such as the install and update hooks), and antipatterns documented in code.

The five layers

Layer	What it documents	Entry
Core anatomy	Physical structure of the repository, profiles, tests, assets, config, scripts	Go →
Drupal\Component	24 subsystems decoupled from the runtime: discovery, plugins, events, serialization	Go →
Drupal\Core	The runtime kernel: 100+ services and contracts of the CMS in execution	Go →
Frontend / core/misc	Native JS/CSS infrastructure: behaviors, AJAX, htmx, states, dialogs	Go →
Core modules	75 functional modules organized by domain	Go →

Recommended reading

If this is your first time through, follow the layer order the Introduction lays out. If you already know the structure, go straight to the chapter you need and follow the cross-references.

Ways in

- Preface — why this book exists and who it is written for.
- Introduction — how core is organized into layers and how to work through it.
- Afterword — the closing piece: the judgment you should take away after going through the system.

Navigation tools

- Glossary of architectural concepts — 130+ terms with precise definitions and links to the chapters where they apply.
- Architecture atlas (Mermaid) — 40 diagrams of the central flows, grouped by layer: the ones that cut across the whole system, `Drupal\Component`, the kernel, the Frontend, and core's modules.
- Index by reader profile — chapters grouped by role: architect, backend developer, tech lead.

Cache

The cache model of most frameworks is simple: store the result of an operation for N seconds, then compute it again. Drupal rejects that model almost entirely. The `Drupal\Core\Cache` namespace implements something qualitatively different: a dependency graph where every cached fragment declares which data it depends on, and the system invalidates exactly the affected entries the moment that data changes.

The practical result is that a block showing node 5 needs no TTL — it needs the `node:5` tag. When someone edits that node, Drupal invalidates every element carrying that tag immediately, with no clock to wait out.

The design has an important consequence for development: caching in Drupal is not something you configure externally and then forget. It is a contract each component signs by declaring its metadata. Understanding that contract — what tags are, what contexts are, what max-age is, and how the metadata travels through the render pipeline — is what separates code that works in development from code that scales in production.

The cacheability trinity (tags, contexts, max-age) is studied in detail in the section on the render pipeline. This chapter covers the underlying infrastructure: the backends where the data lives, the bins that organize it, and the API that manipulates it.

> **Diagram:** the bins, backends, and chained backend and the cacheability model: propagation of tags and contexts in the architecture atlas.

Anatomy

The cacheability trinity

Every operation that generates data or markup must declare three dimensions of metadata (`CacheableMetadata`):

1. **Cache tags (invalidation)**: they answer "which data did this element use?". If a block shows node 5, the tag is `node:5`. Edit the node and the system automatically invalidates every element tied to that tag.
2. **Cache contexts (variation)**: they answer "for whom, or under what conditions, does this content vary?". Common contexts include `languages`, `user.permissions`, and `url.query_args`. Since 11.4 there is also a cache context for the HTTP exception status code (CR 3580452), meant for caching error responses granularly.
3. **Cache max-age (expiration)**: it answers "how long is this piece of data valid for?". It is the fallback for time-based dependencies that can't be expressed as tags. Items are permanent by default: `set()` uses `Cache::PERMANENT` (value `-1`) as `$expire`, so an item with no explicit max-age never expires by time — it only goes away through tag invalidation or deletion.

Cache bins and backends

A bin is a logical storage space with specific semantics — in practice, bins work as cache categories. Core defines `bootstrap`, `config`, `default`, `entity`, `menu`, `render`,

`data`, and `discovery` among others, and several modules contribute their own: `page` (Page Cache), `dynamic_page_cache`, `toolbar`, or the JSON:API bins. The `render` bin stores HTML fragments with cacheability metadata; the `data` bin is general-purpose for modules.

The backend is the physical implementation. Core ships `database` (a `cache_*` table in the database, the default), `APCu` (the PHP process's shared memory, very fast for small data), `php` (`PhpBackend`, PHP files on disk through PhpStorage), and `memory` (in-request only, not persistent). There's also `chainedfast` (`ChainedFastBackend`), which combines a fast volatile backend with a persistent one: it serves from the fast one and, on a miss, falls back to the persistent one, propagating the value upward. In high-traffic production, putting `render` on a fast backend (Redis, Memcache, or APCu) is the most direct performance win available.

The DatabaseBackend row limit

With the default `DatabaseBackend`, each bin is capped at 5,000 rows (`DatabaseBackend::DEFAULT_MAX_ROWS`). Garbage collection keeps only the N most recent rows by the `created` timestamp — a pure FIFO by write time, blind to how often or how recently an entry was read. The consequence: a bin that goes over the limit constantly falls into churn — Drupal evicts entries that are still hot and regenerates them over and over, multiplying database reads and writes. The limit exists to keep the cache tables from growing into the gigabytes, and it's configurable in `settings.php`:

```
$settings['database_cache_max_rows']['default'] = 5000;
$settings['database_cache_max_rows']['bins']['render'] = 50000;
// -1 = no limit (DatabaseBackend::MAXIMUM_NONE).
```

11.4 adds a new bin: `cache.file_parsing` (CR 3593451). Unlike other bins that `drush cr` purges, this one is designed for file parsing results that must survive across deployments — parsing plugins, schemas, routes. It uses deferred writes: it groups the writes so the storage doesn't get swamped when many files are parsed at once.

Variation Cache

A specialized API for caching responses with several context variations. It is the foundation of Dynamic Page Cache: it stores responses indexed by active contexts and resolves the right entry for the current user without computing the full response. Available as the `variation_cache_factory` service (plus the `variation_cache.*` instances).

In practice

Any interaction with the cache needs the right specific bin and the right invalidation service. Not every bin has the same backend or the same semantics.

```
// Injecting the specific bin
$cache_bin = \Drupal::cache('data');

// Retrieval, checking that the entry exists
```

```php
if ($cache = $cache_bin->get($cid)) {
  return $cache->data;
}

// Persisting, with the dependencies declared
$cache_bin->set($cid, $complex_data, Cache::PERMANENT, ['my_custom_tag']);
```

Invalidation by tags

The `cache_tags.invalidator` service is the right entry point for invalidating by tags globally. Invalidate through it and every backend configured for those tags syncs its state.

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

// Invalidate every element that depends on node 5
\Drupal::service('cache_tags.invalidator')
  ->invalidateTags(['node:5', 'node_list']);

// Invalidate a custom tag
Cache::invalidateTags(['mymodule_announcements']);
```

Propagating metadata in render arrays

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

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

// The right way: propagate the object's cacheability
$cacheable_metadata = CacheableMetadata::createFromObject($entity);
$cacheable_metadata->applyTo($build);
```

As of 11.4, the `X-Drupal-Dynamic-Cache` header on 4xx and 5xx responses corrects its value (CR 3584640): instead of the misleading `UNCACHEABLE (poor cacheability)`, it now reports the real status code, `UNCACHEABLE (404)` for example. The header was already emitted on those error responses (when `http.response.debug_cacheability_headers` is enabled); what changed is the value, not whether it shows up. That makes the header-inspection workflow you use on normal pages directly useful for diagnosing the caching of 404s and 403s.

For responses that must never be stored whole there is the `page_cache_kill_switch` service: calling `\Drupal::service('page_cache_kill_switch')->trigger()` marks the current response as uncacheable at the page level. The `KillSwitch` response policy is registered against both Page Cache and Dynamic Page Cache, so a single trigger disables both for that request. It's a last resort: when the variation can be declared, the correct cache metadata (max-age 0, contexts) beats the kill switch.

Hooks

hook_cache_flush

Invoked during the global cache flush. The right extension point for purging external systems that are not natively integrated with Drupal's backend.

```php
namespace Drupal\mymodule\Hook;

use Drupal\Core\Hook\Attribute\Hook;
use Drupal\redis\ClientFactory;
// The redis.factory service is of class Drupal\redis\ClientFactory
// (which does NOT implement ClientInterface). Inject it and get the
// connected client through the getClient() instance method.

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

Adds extra cache metadata to an element before the Renderer processes it. Useful for propagating the tags of entities loaded in the preprocess.

```php
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()
    );
  }
}
```

Engineering standards

Using `Cache::PERMANENT` with the right tags beats arbitrary TTLs. Time-based expiration is acceptable as a fallback for data with a natural expiry — prices, availability, sessions — but not as a general strategy. An element with a 5-minute TTL can serve stale data 4:59 after it changed; an element with the right tags is invalidated the instant the change happens.

Objects implementing `CacheableDependencyInterface` already know which tags, contexts, and max-age belong to them. Using `addCacheableDependency($object)` instead of pulling the tags out by hand avoids mistakes and adapts automatically when the object changes its cache metadata.

Automatic metadata propagation works in the render tree: the children's tags rise to the parents. The mechanism works only if the components declare their own metadata correctly. A component that omits its tags breaks the propagation chain for the whole tree containing it, which leaves entire pages cached when they should have been invalidated.

Common antipatterns

Invalidating with `deleteAll()` or `invalidateAll()` in response to content changes *Symptom:* when a node is published, the code runs `\Drupal::cache('render')->invalidateAll()` to make sure the cache reflects the change. *Consequence:* every cached fragment in the render bin — including pages with no relationship at all to the published node — is invalidated. The next wave of traffic regenerates the entire cache from scratch, with the CPU and database cost that implies. *Root cause:* `invalidateAll()` "guarantees" the change is seen, and that apparent certainty makes it tempting — at the price of throwing the whole cache away for one localized change. Tags exist precisely to invalidate with precision: `invalidateTags(['node:5'])` touches only what depends on node 5. `deleteAll()`/`invalidateAll()` are the emergency operation for a system in a bad state, not an invalidation strategy.

Components with no cache metadata in custom code *Symptom:* a custom service or block builds a render array from entity data but declares no tag, context, or max-age. *Consequence:* the system caches the output as if it were invariant. When the underlying entity changes, the fragment stays stale until someone clears the cache by hand. *Root cause:* in Drupal, the absence of cache metadata means "cacheable unconditionally and forever". That is not the safest default — it is the most aggressive one. Getting the metadata right is the responsibility of the code building the render array, not of the cache system.

Modules that consume this library

- announcements_feed — caches the feed responses to avoid repeated requests
- big_pipe — placeholders are elements with `max-age: 0` that the cache system identifies and BigPipe replaces
- block — each block declares its own cache tags and contexts in `build()`
- dynamic_page_cache — caching of render arrays with full contexts; it depends on cache metadata being declared correctly
- node — the `node:{nid}` and `node_list` cache tags that Node declares automatically

- page_cache — full page caching for anonymous users; it consumes the kernel's cache backends directly
- taxonomy — the `taxonomy_term:{tid}` and `taxonomy_term_list` cache tags for automatic invalidation
- views — views have their own configurable cache, with tags and contexts per view **Layer below:** Drupal\Component | **Part index:** Core libraries

DrupalKernel

`Drupal\Core\DrupalKernel` is the framework's entry point. It is the class that turns an empty PHP script into a fully working Drupal system: it discovers and loads extensions, compiles Symfony's service container, processes the HTTP request, and returns a response. Everything that happens in a Drupal request passes through here. It is the concrete implementation of the DrupalKernelInterface contract, which Drush and the tests manipulate when they control the bootstrap without coupling to this class.

Drupal's bootstrap is split into phases, each one adding capability to the system. The early phases initialize the minimum needed to read the environment configuration; the late ones load the full container. This architecture lets the system fail gracefully and lets tools such as Drush initialize only the phases they need.

In the modern deployment model, the service container is compiled once and cached as a serialized definition in the `cache.container` backend (by default the database's `cache_container` table through `Drupal\Core\Cache\DatabaseBackend`). On each request, `DrupalKernel` retrieves that cached definition and instantiates the container without recompiling, reducing the bootstrap cost to a single read. Rebuilding it explicitly (`drush cr`) is only necessary when the service configuration changes or new modules are installed.

Diagram: the kernel bootstrap phases in the architecture atlas.

Anatomy

Bootstrap phases

`DrupalKernel` initializes the system in sequential phases. The configuration phase reads `settings.php`; the kernel phase creates the container; later phases load the session, the language, and the route. External tools such as Drush can bootstrap up to the phase they need without paying the cost of the later ones.

Compiling and dumping the container

The service container is compiled with `ContainerBuilder` and then serialized into an array of definitions by `OptimizedPhpArrayDumper` (`Drupal\Component\DependencyInjection\Dumper\OptimizedPhpArrayDumper`), not by Symfony's `PhpDumper`. That array (`$dumper->getArray()`) includes every service definition, their dependencies, and the parameters, and it is stored in the `cache.container` backend through `cacheDrupalContainer()`. On the next request, `getCachedContainerDefinition()` retrieves it and `Drupal\Core\DependencyInjection\Container` consumes it directly, with no recompilation.

Handle and terminate

`DrupalKernel::handle(Request $request)` is the main method: it builds the container if it needs to, resolves the request through the router and the event subscribers, and returns the `Response`. `terminate()` is invoked after the response has been sent. It delegates to Symfony's `http_kernel` if that implements `Terminable`

Interface, and it walks the kernel.destructable_services parameter to call destruct() on every DestructableInterface service initialized during the request. Since 11.4, what invokes this pair is no longer index.php directly, but the symfony/runtime runner (see “In practice”).

In practice

```php
// index.php in 11.4+ — core's real front controller.
// (Not custom code — this is the file that serves every web request.)
use Drupal\Core\DrupalKernel;

require_once 'autoload_runtime.php';

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

Since 11.4, index.php no longer runs the request cycle: it returns a closure that builds the kernel, and the symfony/runtime component does the rest. The chain is short. autoload_runtime.php (in the site root) sets Drupal\Core\Runtime\DrupalRuntime as the default runtime through the APP_RUNTIME variable and delegates to the vendor/autoload_runtime.php that the component’s Composer plugin generates. DrupalRuntime extends SymfonyRuntime, tuning the defaults to the Drupal world: it turns off Symfony’s error handler (Drupal registers its own), turns off .env loading, and sets prod as the environment when APP_ENV is undefined. With the kernel the closure returns, the component’s runner (HttpKernelRunner) executes the cycle that up to 11.3 lived literally in index.php:

```php
// The classic cycle — up to 11.3 this was the body of index.php; since 11.4
// the symfony/runtime runner executes it with the returned kernel.
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

The practical result is a formal separation between bootstrap (building the kernel, loading the container) and handling the request: symfony/runtime manages the PHP process lifecycle, and DrupalKernel limits itself to building the container and delegating to the http_kernel. Front controllers written in the classic format keep working unchanged “for the time being”. The CR announces no removal deadline, but it presents adopting the new pattern as the recommended path for custom scripts that bootstrap Drupal directly (CR 3553275).

```php
// In a service provider, alter the container during compilation.
// DrupalKernel invokes this before serializing the container to static PHP.
use Drupal\Core\DependencyInjection\ContainerBuilder;
use Drupal\Core\DependencyInjection\ServiceProviderInterface;

final class MyModuleServiceProvider implements ServiceProviderInterface {
  public function register(ContainerBuilder $container): void {
    // Register services or parameters during compilation.
    $container->setParameter('my_module.env', getenv('APP_ENV') ?: 'prod');
```

```
  }
}
```

Integration with kernel events

`DrupalKernel` defines no module hooks at this point. To intervene in the request cycle, the correct mechanism is subscribing to Symfony's kernel events (`REQUEST`, `RESPONSE`, `TERMINATE`, and so on).

```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;
    }

    // Lightweight initialization logic for the main request.
  }
}
```

Engineering standards

Don't extend `DrupalKernel`. `DrupalKernel` is an internal core class. Extending it to customize behavior breaks updates whenever Drupal changes the class. Use service providers to alter the container.

Environment configuration belongs in `settings.php`. The differences between environments (development/staging/production) must be expressed in `settings.php` or in environment variables, not in conditional PHP code.

`DRUPAL_TEST_IN_CHILD_SITE` and similar constants are test infrastructure. Don't use these constants for business logic in modules.

Common antipatterns

Extending `DrupalKernel` to customize the bootstrap

Symptom: the project has `class MyKernel extends DrupalKernel` with overridden methods. *Consequence:* every Drupal update that changes `DrupalKernel` (method

signatures, internal logic) requires updating the custom subclass. Major upgrades become far more expensive. *Root cause:* `DrupalKernel` is not part of the extensible API; its internal signature changes between versions with no warning, because core does not treat it as an extension point. Each legitimate hook has its own channel: `ServiceProviderInterface` for the container, `settings.php` for the environment, event subscribers for the request.

```
// ✗ Extending DrupalKernel
class MyCustomKernel extends DrupalKernel {
  protected function initializeSettings(Request $request): void {
    parent::initializeSettings($request);
    // Custom logic — it will break on updates
  }
}

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

Modules that consume this library

- system — the system module interacts with the kernel at bootstrap, rebuild, and maintenance **Layer below:** Drupal\Component | **Part index:** Core libraries

The complete book

This sample collects 6 of the 254 chapters of *Understanding Drupal core*. The rest of the book follows the same criterion: read the core's source and explain why it is written the way it is.

The chapters marked with ✓ are the ones you have just read.

Front matter

- Credits and license ✓
- Preface ✓
- Acknowledgments
- Introduction ✓
- The book at a glance ✓

Core anatomy

- Core anatomy
- Kernel libraries
- Core modules
- Assets
- Config
- Data Types
- Includes
- Debug
- Profiles
- Recipes
- Root Files
- Scripts
- Tests
- Themes
- Claro theme
- Olivero theme
- Default Admin theme

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

Core libraries

- Core libraries
- Access
- Accessibility
- Action
- Ajax
- Annotation (Core libraries)
- Archiver
- Asset
- Authentication
- Batch
- Block
- Breadcrumb
- Cache ✓
- ClassLoader
- Command
- Composer
- CoreServiceProvider
- Cron
- CronInterface
- Condition
- Config (Core libraries)
- Controller
- Database
- Datetime (Core libraries)
- DefaultContent
- DependencyInjection (Core libraries)
- DestructableInterface
- Dialog
- Diff (Core libraries)
- Discovery (Core libraries)
- 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 (Core libraries)
- Plugin (Core libraries)
- PreWarm
- PrivateKey
- ProxyBuilder (Core libraries)
- ProxyClass
- Queue
- Recipe
- Render (Core libraries)
- RouteProcessor
- Routing
- RoutingForm
- Security
- Serialization (Core libraries)
- Session
- Site
- StackMiddleware
- State
- States
- StreamWrapper

- StringTranslation
- Tabledrag
- Template
- TempStore
- Test
- Theme
- Token
- Transliteration (Core libraries)
- TypedData
- UIComponents
- Update
- Updater
- Url
- Utility (Core libraries)
- 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 and static assets
- States (JS (core/misc))
- Tabledrag (JS (core/misc))
- UIComponents (JS (core/misc))
- Messages
- Utilities

Core modules

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

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

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

Front matter

- Afterword

Reference

- Glossary
- Architecture atlas
- Index by role