Docs

Backbone Project

Derafu Backbone

Anonymous · 21/08/2026 · #php

Table of Contents

Backbone Project

Derafu Backbone

Anonymous · 21/08/2026 · #php

Derafu Backbone

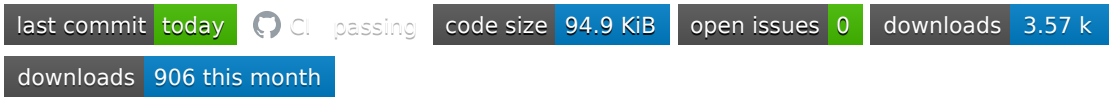
Last updated on 21/08/2026

Introduction

The Architectural Spine for PHP Libraries

Anonymous · 21/08/2026

The Architectural Spine for PHP Libraries



Derafu Backbone is a lightweight architectural framework that provides a consistent structure for building modular, maintainable PHP libraries.

Features

- **Hierarchical Organization**: Clear structure with Packages, Components, and Workers.
- **Separation of Concerns**: Jobs for atomic operations, Handlers for orchestration, Strategies for implementation variants.
- **Attribute-Based Discovery**: Use PHP 8 attributes instead of rigid namespace conventions.
- **Extensible Architecture**: Designed to grow with your application.

Key Benefits

- **Consistent Structure**: Standardized approach to organizing domain logic.
- **Reduced Complexity**: Clear responsibilities for each architectural element.
- **Improved Testability**: Isolated components are easier to test.
- **Enhanced Collaboration**: Common vocabulary and patterns for development teams.
- **Flexible Implementation**: Adapt to different domains without changing the core architecture.

Installation

```
composer require derafu/backbone
```

Quick Example

```
#[Package(name: 'billing')]
class BillingPackage extends AbstractPackage implements PackageInterface
{
    // Package implementation.
```

```
}

#[Component(name: 'document', package: 'billing')]
class DocumentComponent extends AbstractComponent implements ComponentInterface
{
    // Component implementation.
}

#[Worker(name: 'renderer', component: 'document', package: 'billing')]
class RendererWorker extends AbstractWorker implements WorkerInterface
{
    // Worker implementation.
}
```

Last updated on 21/08/2026

Architecture

Architecture

Anonymous · 21/08/2026

Architecture

Derafu Backbone provides a structured framework for building modular and maintainable PHP libraries. It follows a hierarchical organization that separates concerns into distinct components, making your code more organized, testable, and extensible.

Package

Definition: A high-level container representing a complete domain or subdomain within your application.

Responsibility: Groups functionally related components that work together to provide domain-specific capabilities.

Characteristics:

- Represents a complete business domain (e.g., Billing, Accounting, Human Resources).
- Contains multiple components.
- Provides domain-wide configuration and services.
- Acts as the primary entry point for domain functionality.

Examples: `BillingPackage`, `AccountingPackage`, `HumanResourcesPackage`

Component

Definition: A functional area or module within a domain package.

Responsibility: Groups workers that handle specific aspects of the domain.

Characteristics:

- Represents a distinct functional area within a domain.
- Contains multiple workers focused on related tasks.
- Provides component-specific configuration.
- Manages cross-cutting concerns within its functional area.

Examples: `DocumentComponent`, `ExchangeComponent`, `TradingPartiesComponent`

Worker

Definition: Coordinator of business operations related to a specific aspect of functionality.

Responsibility: Exposes public methods that can be called by clients and coordinates Jobs, Handlers, and Strategies to perform work.

Characteristics:

- Acts as a facade for the underlying implementation.
- Exposes domain operations through public methods.
- Coordinates the execution of jobs and handlers.
- May implement simple operations directly.
- Delegates complex processing to handlers.

Examples: `BuilderWorker`, `RendererWorker`

Job

Definition: An atomic, self-contained unit of work that performs a single operation.

Responsibility: Executes a specific task with clear inputs and outputs.

Characteristics:

- Focused on doing one thing well.
- Encapsulates a single operation.
- Has clear, well-defined inputs and outputs.
- Does not orchestrate other jobs.
- Reusable across different contexts.
- Stateless and idempotent when possible.

Examples: `NormalizeBoletaAfectaJob`, `NormalizeFacturaAfectaJob`

Handler

Definition: Orchestrator of complex processes involving multiple operations.

Responsibility: Coordinates multiple jobs or strategies to complete complex workflows.

Characteristics:

- Orchestrates multiple operations in sequence.
- Manages workflow and process state.
- Selects appropriate strategies based on context.
- Handles errors and transactional boundaries.
- Implements business process logic.
- Can use multiple jobs and strategies.

Examples: `EmailSenderHandler`, `SiiSenderHandler`

Strategy

Definition: Specific implementation of an algorithm or approach to solving a problem.

Responsibility: Provides variant implementations for specific operations that can be interchanged.

Characteristics:

- Implements a specific algorithm or approach.
- Allows for swappable implementations.
- Used by handlers to provide different behaviors.
- Focuses on “how” something is done.
- Follows the Strategy design pattern.
- Encapsulates specific implementation details.

Examples: `JsonParserStrategy`, `XmlParserStrategy`, `YamlParserStrategy`

Benefits of This Architecture

1. **Separation of Concerns**: Each component has a clear, single responsibility.
2. **Modularity**: Components can be developed, tested, and deployed independently.
3. **Extensibility**: New implementations can be added without modifying existing code.
4. **Testability**: Clear boundaries make testing easier and more focused.
5. **Maintainability**: Organized structure makes the codebase easier to understand and maintain.
6. **Flexibility**: Strategies allow for varying implementations without changing coordination logic.

Design Pattern Comparisons

- **Jobs**: Similar to the Command Pattern.
- **Handlers**: Combination of Chain of Responsibility and Mediator patterns.
- **Strategies**: Direct implementation of the Strategy Pattern.
- **Workers**: Follow the Facade Pattern.
- **Overall Structure**: Influenced by Hexagonal/Ports and Adapters Architecture.

This architecture provides a clear separation of responsibilities, allowing for component reuse and making your system more adaptable to changing requirements.

Last updated on 21/08/2026

Relationships

Relationships and Interactions

Anonymous · 21/08/2026

Relationships and Interactions

Package → Component Relationship

- A Package contains multiple Components.
- Components within a Package are thematically related.
- Components access Package-level configuration and services.

Component → Worker Relationship

- A Component contains multiple Workers.
- Workers are grouped logically within a Component.
- Workers share Component-level resources and configuration.

Worker → Job/Handler/Strategy Relationship

- Workers expose public methods that can be called by clients.
- Workers delegate to Jobs for simple operations.
- Workers delegate to Handlers for complex workflows.
- Workers don’t typically interact directly with Strategies (Handlers do).

Handler → Job/Strategy Relationship

- Handlers coordinate the execution of multiple Jobs.
- Handlers select and use appropriate Strategies based on context.
- Handlers implement orchestration logic while Jobs and Strategies provide implementation.

Strategy Relationships

- Strategies implement interchangeable algorithms.
- Handlers select appropriate Strategies based on conditions.
- Strategies focus on specific implementation details.

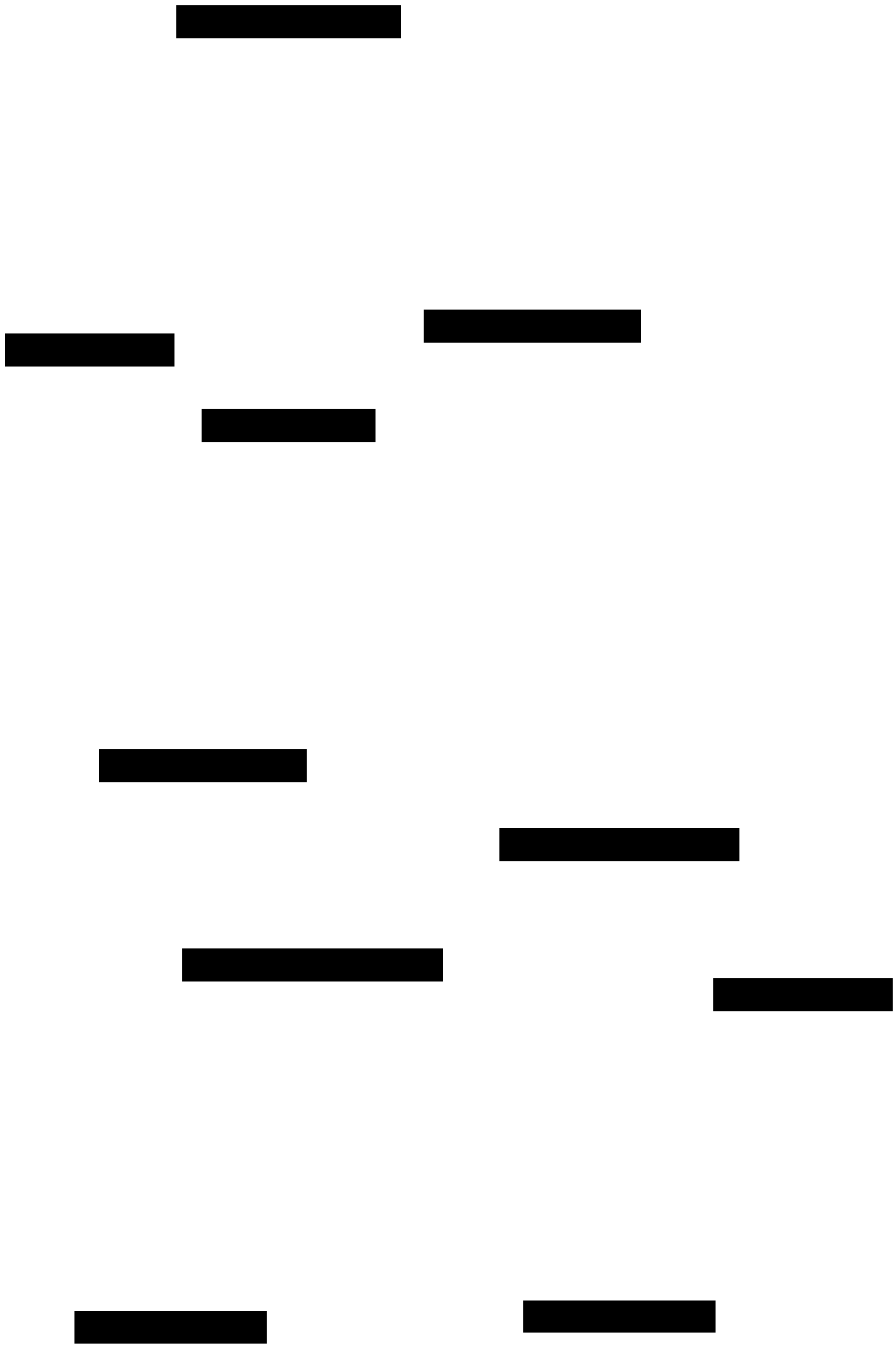
Last updated on 21/08/2026

Workflow

Typical Workflow

Anonymous · 21/08/2026

Typical Workflow



1. A client interacts with a public method on a Worker.

2. The Worker can:

    - Execute simple logic internally.
    - Delegate to a Job for a specific task.
    - Delegate to a Handler for a complex process.

3. If a Handler is used:

    - The Handler coordinates the workflow.
    - May execute multiple Jobs in sequence.
    - May select different Strategies based on context.
    - Manages errors and transactions.

4. Strategies allow for varying the implementation of specific steps without changing the Handler's logic.

Last updated on 21/08/2026

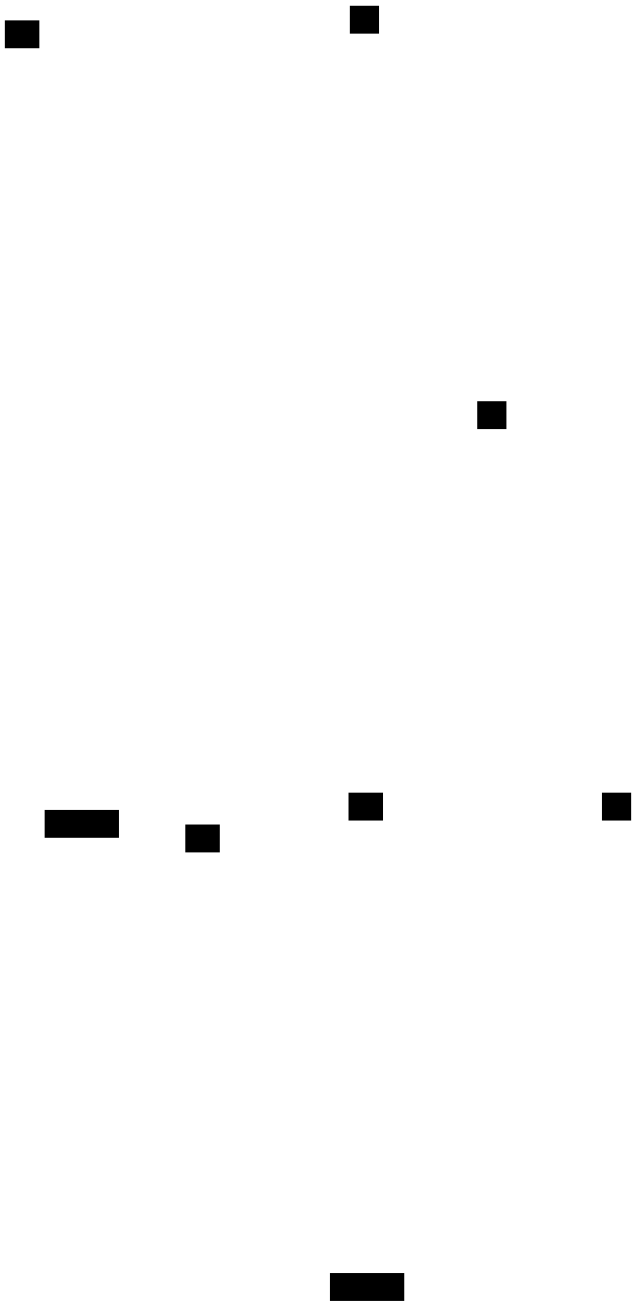
Decision Flow

Decision Flow: Choosing the Right Component

Anonymous · 21/08/2026

Decision Flow: Choosing the Right Component

This document expands on the Decision Flow Diagram, helping you understand when to use each architectural component in Derafu Backbone.



Understanding the Decision Process

When implementing a new piece of functionality in Derafu Backbone, one of the most important decisions is determining which architectural component should handle that functionality. The diagram provides a decision tree to guide this process, but let's explore the reasoning and implications of each choice.

When to Use Jobs

Jobs are the workhorses of Backbone architecture. You should choose a Job when:

- The operation performs a **single, well-defined task**
- The operation has **clear inputs and outputs**
- The operation doesn't need to manage complex flows or state
- The operation could potentially be reused in different contexts

Jobs are particularly valuable when you need to implement operations that might be used by multiple handlers or directly by workers. They represent atomic units of work that should follow the Single Responsibility Principle.

Example scenarios for Jobs:

- Validating input data
- Sending a notification
- Storing an entity in a repository
- Transforming data from one format to another
- Performing a calculation

When to Use Handlers

Handlers should be your choice when:

- The operation involves **multiple steps** or jobs
- You need to **orchestrate a complex workflow**
- There's **conditional logic** determining the flow
- The operation manages **transactional boundaries**
- You need to **coordinate between different components**

Handlers encapsulate complex business processes and provide a higher level of abstraction. They know how to execute a complete business use case by coordinating the execution of multiple jobs and selecting appropriate strategies.

Example scenarios for Handlers:

- Processing a payment (validating, charging, recording transaction, sending receipt)
- Generating a report (gathering data, applying business rules, formatting, delivering)

- User registration flow (validating, creating account, sending welcome email, initializing user settings)

When to Use Strategies

Strategies are the right choice when:

- You need **multiple implementations** of the same operation
- The implementation should be **selectable at runtime**
- The implementation choice depends on **context or configuration**
- You want to **eliminate conditional logic** from your handlers and jobs

Strategies allow your system to adapt to different circumstances without changing the orchestration logic. They encapsulate the "how" of an operation, while jobs and handlers focus on the "what".

Example scenarios for Strategies:

- Different export formats (PDF, CSV, Excel)
- Multiple payment processors (Stripe, PayPal, Bank Transfer)
- Various notification channels (Email, SMS, Push Notification)
- Different storage backends (File System, S3, Database)

When to Use Direct Worker Implementation

In some cases, you might implement functionality directly in the Worker:

- For very simple operations that don't warrant a separate Job
- For operations that are specific to a particular Worker and won't be reused
- When acting as a simple facade over third-party libraries
- For convenience methods that combine calls to Jobs or Handlers

Practical Applications

Component Relationships

Notice how Jobs and Strategies are often used by Handlers. This relationship is key to understanding the architecture:

- **Jobs** provide the atomic operations
- **Handlers** orchestrate these operations
- **Strategies** provide implementation variants

Benefits of Following This Decision Flow

By correctly choosing the appropriate architectural component:

1. **Improved Testability**: Jobs and Strategies are easier to test in isolation.
2. **Enhanced Maintainability**: Clear separation of concerns makes the code more maintainable.
3. **Greater Flexibility**: Strategies enable system adaptability without widespread changes.
4. **Better Reusability**: Jobs can be reused across multiple contexts.
5. **Clearer Intent**: The architecture communicates the intent of each piece of code.

Edge Cases and Considerations

- **Size and Complexity**: Very simple applications might not need the full hierarchy. Start with Jobs and add Handlers and Strategies as complexity grows.
- **Performance**: While this architecture promotes good design, be mindful of potential overhead from excessive layering in performance-critical paths.
- **Boundaries**: Consider your domain boundaries carefully when organizing packages and components.

By following this decision flow, you can create a clean, maintainable, and adaptable codebase that effectively leverages the full potential of Derafu Backbone's architecture.

Last updated on 21/08/2026

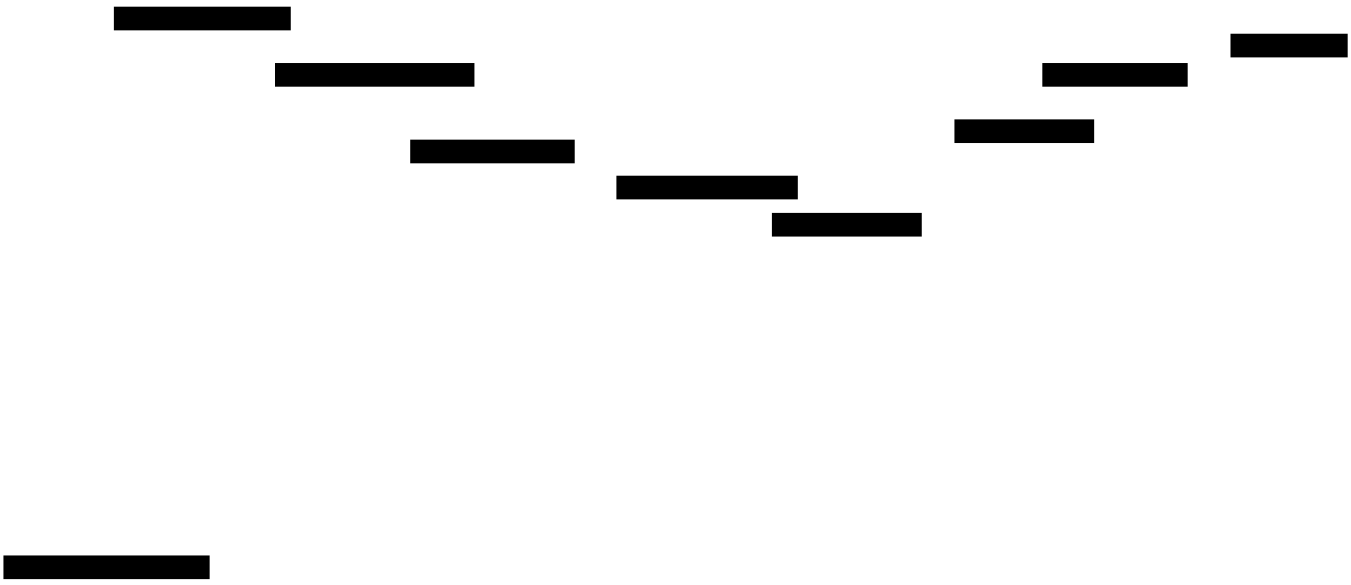
Component Interaction

Component Interaction Sequence

Anonymous · 21/08/2026

Component Interaction Sequence

This document explores how the various components of Derafu Backbone interact during typical operations, providing insight into the architectural flow and communication patterns.



The Flow of Control

The sequence diagram illustrates the typical flow of control when using Derafu Backbone for a business operation. Understanding this flow is crucial for effectively working with the architecture and designing new components.

1. Entry Point: The Registry

All interaction with a Backbone-based system typically begins with the Package Registry. This is the service locator that provides access to the domain packages in your application. The Package Registry

serves as the entry point into your domain architecture.

```
// Obtaining the package registry (typically from a kernel or container).
$packageRegistry = $container->get(PackageRegistryInterface::class);
```

The Package Registry is responsible for:

- Maintaining references to all registered packages.
- Providing type-safe access to specific packages.
- Acting as the gateway to your domain architecture.

2. Accessing a Package

After obtaining the registry, the client accesses a specific domain package:

```
// Get a specific package by name.
$billingPackage = $packageRegistry->getPackage('billing');
// Or using a type-safe helper method.
$billingPackage = $packageRegistry->getBillingPackage();
```

Packages represent complete domains or subdomains in your application. They contain all components related to a specific business area. The clean organization of packages helps maintain separation between different domains.

3. Accessing a Component

From a package, the client accesses a specific component:

```
// Access a component within the package.
$invoiceComponent = $billingPackage->getComponent('invoice');
// Or using a type-safe helper method.
$invoiceComponent = $billingPackage->getInvoiceComponent();
```

Components group related functionality within a domain. They represent functional areas within your domain and contain workers that handle specific aspects of that functionality.

4. Accessing a Worker

From a component, the client accesses a specific worker:

```
// Access a worker within the component.
$generatorWorker = $invoiceComponent->getWorker('generator');
// Or using a type-safe helper method.
$generatorWorker = $invoiceComponent->getGeneratorWorker();
```

Workers are the coordinators that expose domain operations through public methods. They manage jobs and handlers to perform specific tasks within a component.

5. Using Jobs or Handlers

Finally, the client accesses and executes a job or handler:

```
// Using a job.
$invoice = $generatorWorker->getCreateJob()->execute($data);

// Or using a handler.
$result = $generatorWorker->getProcessHandler()->handle($invoice, $options);
```

Jobs and handlers do the actual work in the system. Jobs perform atomic operations, while handlers orchestrate complex processes that might involve multiple jobs and strategies.

Direct Access vs. Hierarchical Access

While the diagram shows a step-by-step hierarchical flow, it's important to note that once a client has obtained the registry, it can access any level directly:

```
// Hierarchical access (step by step).
$invoice = $packageRegistry
    ->getBillingPackage()
    ->getInvoiceComponent()
    ->getGeneratorWorker()
    ->getCreateJob()
    ->execute($data);

// Direct access (if you already have a reference to the worker).
$invoice = $generatorWorker->getCreateJob()->execute($data);
```

Both approaches are valid, but the hierarchical access pattern is generally recommended for application code as it:

- Makes the dependency hierarchy explicit.
- Follows a natural discovery pattern.
- Makes the code more readable and self-documenting.

The direct access pattern can be useful in:

- Unit tests where you want to focus on testing a specific component.
- Performance-critical paths where you can cache references to frequently used services.
- Situations where you already have a reference to a higher-level component.

Alternative Path: Using Handlers

For more complex operations, handlers provide an alternative path:

```
// Using a handler for complex operations.
$result = $generatorWorker->getProcessHandler()->handle($invoice, $options);
```

Handlers orchestrate complex workflows and can:

- Execute multiple jobs in sequence.
- Apply conditional logic.
- Select appropriate strategies based on input or configuration.
- Manage transactional boundaries.
- Handle cross-cutting concerns like logging or error handling.

Communication Patterns

The diagram illustrates several important communication patterns in Backbone:

1. **Hierarchical Organization**: Components are organized in a clear hierarchy that reflects your domain structure.
2. **Dependency Injection**: Components receive their dependencies through constructor injection.
3. **Lazy Loading**: Services are typically loaded lazily, meaning they're only instantiated when needed.
4. **Interface-Based Programming**: Communication happens through well-defined interfaces.

Performance Considerations

The hierarchical structure of Backbone might raise concerns about performance overhead. However, several aspects mitigate this:

1. **Lazy Loading**: Services are loaded only when needed.
2. **Cached References**: In performance-critical paths, you can cache references to frequently used services.
3. **DI Container Optimization**: Modern DI containers can optimize service instantiation.
4. **Proxy Generation**: Backbone can use proxies to further optimize lazy loading.

By understanding these interaction patterns, you can effectively design, implement, and use components within Derafu Backbone to create maintainable and extensible applications.

Last updated on 21/08/2026

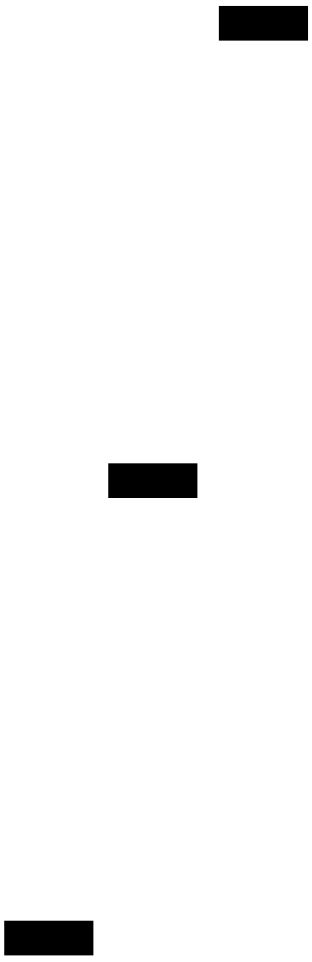
Configuration Layers

Configuration Layers

Anonymous · 21/08/2026

Configuration Layers

This document explores how configuration cascades through the architectural layers in Derafu Backbone, providing a powerful and flexible way to configure your application.



Configuration Hierarchy

Derafu Backbone implements a hierarchical configuration system that follows the same structure as the architectural components. This cascade system allows for powerful customization while

maintaining sensible defaults.

Package Configuration

At the top of the hierarchy is the Package configuration, which provides the foundation for all configuration within a domain:

```
# Example package configuration.
example_package:
  options:
    debug: false
    cache_enabled: true
  components:
    example_component:
      # Component-specific configuration.
    another_component:
      # Another component's configuration.
```

Package configuration is ideal for:

- Domain-wide settings that affect all components.
- Default values that components can override.
- Shared service configuration.
- Environment-specific settings for an entire domain.
- Feature flags that affect the entire package.

The Package configuration establishes the baseline for all contained components.

Component Configuration

The Component configuration provides more specific settings for a functional area within a package:

```
# Component configuration within a package.
example_package:
  components:
    example_component:
      options:
        timeout: 30
        max_retries: 3
      workers:
        example_worker:
          # Worker-specific configuration.
        another_worker:
          # Another worker's configuration.
```

Component configuration is appropriate for:

- Feature-specific settings.
- Overriding package defaults for a specific component.
- API keys or endpoints for external services used by the component.
- Component-specific validation rules or constraints.
- Logging or monitoring settings for a component.

Component configuration inherits from the package level but can override any setting as needed.

Worker Configuration

Worker configuration provides even more specific settings for individual workers within a component:

```
# Worker configuration within a component
example_package:
  components:
    example_component:
      workers:
        example_worker:
          options:
            batch_size: 100
            processing_mode: "async"
          jobs:
            # Job-specific configuration
          handlers:
            # Handler-specific configuration
```

Worker configuration is useful for:

- Task-specific settings.
- Overriding component defaults for a specific worker.
- Performance tuning parameters.
- Feature flags for specific functionality.
- Worker-specific thresholds or limits.

Worker configuration inherits from both the package and component levels.

Job/Handler/Strategy Configuration

At the lowest level, individual jobs, handlers, and strategies can have their own configuration:

```
# Job configuration within a worker
example_package:
  components:
    example_component:
      workers:
        example_worker:
          jobs:
```

```
            example_job:
              options:
                validation_level: "strict"
                timeout: 10
```

This level of configuration is ideal for:

- Operation-specific settings.
- Fine-grained control over individual operations.
- Feature flags for specific functionality.
- Operation-specific thresholds or limits.

Accessing Configuration

Derafu Backbone provides a consistent way to access configuration at any level through the `ConfigurableInterface` and the `ConfigurableTrait`:

```
// Inside a package, component, or worker.
$config = $this->getConfiguration();

// Accessing specific options.
$debug = $config->get('options.debug', false); // With default value.
$timeout = $config->get('options.timeout'); // Without default.

// Accessing nested configuration.
$workerConfig = $config->get('workers.example_worker');
```

The configuration system is designed to be:

- **Type-safe**: Configurations can be defined with schemas for validation.
- **Environment-aware**: Different configurations can be loaded based on the environment.
- **Hierarchical**: Configurations cascade from more general to more specific.
- **Defaulted**: Sensible defaults can be provided at any level.

Configuration Inheritance Mechanism

The key benefit of Backbone's layered configuration is inheritance:

1. **Default Values**: Each layer can provide default values that are used if not overridden.
2. **Selective Overrides**: Lower levels can override specific settings without affecting others.
3. **Aggregation**: Configuration is aggregated from all levels before being used.

For example, if you have these configurations:

```
# Package level.
```

```
example_package:
  options:
    debug: false
    cache_enabled: true
    timeout: 30

# Component level (in the same file).
example_package:
  components:
    example_component:
      options:
        timeout: 60
```

Then, when accessing configuration from within `ExampleComponent`:

```
$config = $this->getConfiguration();
$debug = $config->get('options.debug'); // false (inherited from package)
$cacheEnabled = $config->get('options.cache_enabled'); // true (inherited from
package)
$timeout = $config->get('options.timeout'); // 60 (overridden at component level)
```

Best Practices for Configuration Management

When working with Backbone's configuration layers:

1. **Define defaults at the highest appropriate level**: Place configuration that applies broadly at the package level.
2. **Override only what's necessary**: Only override configuration when you need to change it.
3. **Be explicit about types and validation**: Use configuration schemas to validate configuration.
4. **Use environment-specific configuration**: Leverage environment variables and profiles for environment-specific settings.
5. **Document your configuration**: Clearly document what configuration options are available and what they do.
6. **Keep sensitive information separate**: Use environment variables or dedicated configuration stores for secrets.

By understanding and effectively using Backbone's configuration layers, you can create flexible, configurable applications that are easy to adapt to different environments and use cases.

Last updated on 21/08/2026

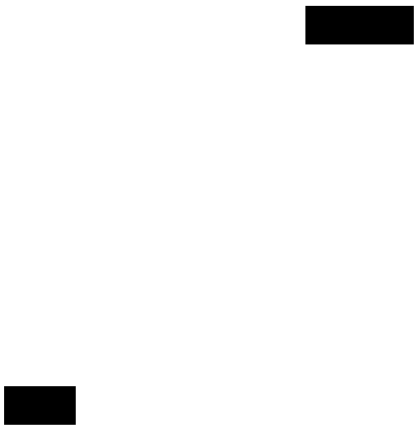
Service Lifecycle

Service Lifecycle

Anonymous · 21/08/2026

Service Lifecycle

This document explains the lifecycle of services within Derafu Backbone, from definition to usage, providing insights into how services are discovered, configured, and instantiated.



Three-Phase Lifecycle

Derafu Backbone services go through three distinct phases during their lifecycle:

1. **Definition Phase**: How services are defined in code.
2. **Discovery Phase**: How services are discovered and registered.
3. **Usage Phase**: How services are instantiated and used.

Understanding this lifecycle is crucial for developing with Backbone and troubleshooting any issues that may arise.

1. Definition Phase

The definition phase is where you define your Backbone services in code.

Class Definition with PHP 8 Attributes

The cornerstone of Backbone's service definition is PHP 8 attributes. These attributes provide metadata about your services and eliminate the need for verbose configuration:

```
#[Package(name: 'billing', description: 'Handles billing operations')]
class BillingPackage extends AbstractPackage implements PackageInterface
{
    // Package implementation.
}

#[Component(name: 'document', package: 'billing')]
class DocumentComponent extends AbstractComponent implements ComponentInterface
{
    // Component implementation.
}

#[Worker(name: 'renderer', component: 'document', package: 'billing')]
class RendererWorker extends AbstractWorker implements WorkerInterface
{
    // Worker implementation
}
```

The attributes provide several key pieces of information:

- **Service type**: Package, Component, Worker, Job, Handler, or Strategy
- **Service name**: The identifier used to reference the service
- **Service hierarchy**: The parent services (package, component, worker)
- **Description**: Optional description for better documentation

Abstract Base Classes and Interfaces

All Backbone services extend abstract base classes and implement interfaces:

- **Abstract classes** provide common functionality like configuration handling, ID generation, and standardized constructors.
- **Interfaces** define the contract that each service must fulfill.

This combination ensures consistent behavior and makes services interchangeable and testable.

Dependency Injection

Services declare their dependencies through constructor injection:

```
#[Component(name: 'document', package: 'billing')]
class DocumentComponent extends AbstractComponent implements ComponentInterface
{
    public function __construct(
        private readonly BuilderWorker $builderWorker,
        private readonly RendererWorker $rendererWorker
    ) {
    }

    public function getWorkers(): array
    {
        return [
            'builder' => $this->builderWorker,
            'renderer' => $this->rendererWorker,
        ];
    }

    // Other component implementation.
    // It's recommended to create getters for the dependencies. This will
    // provide safe type hinting and autocompletion.
}
```

This explicit declaration of dependencies:

- Makes dependencies clear and traceable.
- Facilitates testing through mocking.
- Allows the DI container to resolve dependencies automatically.
- Ensures services are properly initialized.

2. Discovery Phase

The discovery phase is where defined services are found, processed, and registered in the dependency injection container.

Compiler Passes

Backbone uses Symfony's compiler passes to discover and process services:

ServiceProcessingCompilerPass

This compiler pass:

1. Scans all service definitions in the container.

2. Identifies classes with Backbone attributes.
3. Extracts metadata from attributes.
4. Creates and registers service definitions.
5. Adds appropriate tags for later reference.
6. Mark all services as lazy services.
7. Mark only packages as public services.

The result is that all Backbone services are properly registered in the container, with appropriate tags and metadata.

ServiceConfigurationCompilerPass

This compiler pass:

1. Finds services that implement `ConfigurableInterface`.
2. Matches them with configuration from parameters.
3. Adds method calls to set configuration when the service is created.

This ensures that services receive their configuration during instantiation.

Registration and Aliasing

During the discovery phase, services are registered in the container and aliased for easier access:

```
// Original service ID (class name).
App\Billing\DocumentComponent

// Aliased as (from attribute metadata).
billing.document
```

These aliases make it easier to reference services by their logical names rather than class names.

3. Usage Phase

The usage phase is where services are actually instantiated and used at runtime.

Lazy Loading

By default, Backbone services are registered as lazy services. This means:

- They’re only instantiated when actually needed.
- A proxy is created that loads the real service on first method call.
- This improves performance by avoiding unnecessary service instantiation.

Accessing Services

Services can be accessed through the Package Registry:

```
// Get a package from the registry.
$billingPackage = $packageRegistry->getPackage('billing');

// Get a component from the package.
$documentComponent = $billingPackage->getComponent('document');

// Get a worker from the component.
$rendererWorker = $documentComponent->getWorker('renderer');

// Use the worker.
$pdf = $rendererWorker->render($data);
```

Each level in the hierarchy provides access to the level below it, creating a discoverable API.

Configuration Application

When a service is instantiated:

1. The container resolves its dependencies.
2. Dependencies are injected via the constructor.
3. Configuration is applied via a method call (if the service is configurable).
4. The service is ready to use.

This ensures that services are fully initialized before use.

Understanding the Flow

The complete flow from definition to usage:

1. **Define** services using PHP 8 attributes.
2. **Compile** the container, triggering discovery and registration.
3. **Build** the container for runtime use.
4. **Access** services through the registry.
5. **Use** services to perform domain operations.

Benefits of This Lifecycle

This lifecycle approach provides several benefits:

1. **Discoverability**: Services are easily discoverable through attributes.
2. **Configuration**: Services receive appropriate configuration automatically.

3. **Lazy Loading**: Only services that are actually used are instantiated.
4. **Type Safety**: The full hierarchy is type-safe through interfaces.
5. **Testability**: Dependencies are explicit and can be mocked for testing.

Troubleshooting the Lifecycle

Common issues and how to address them:

1. **Service not found**: Ensure attributes are correctly defined and the compiler pass is registered.
2. **Configuration not applied**: Check that your configuration keys match the expected structure.
3. **Dependencies not resolved**: Verify that all dependencies are correctly registered in the container.
4. **Performance issues**: Consider caching the compiled container in production.

By understanding the complete service lifecycle in Derafu Backbone, you can effectively develop, configure, and troubleshoot your application.

Last updated on 21/08/2026

Design Patterns

Backbone and Design Patterns

Anonymous · 21/08/2026

Backbone and Design Patterns

This document explores how Derafu Backbone implements and leverages established design patterns to create a robust, maintainable architecture for PHP applications.

Design Patterns in Backbone

Derafu Backbone draws inspiration from several well-established design patterns. Understanding these patterns and their implementation in Backbone helps developers use the framework effectively and extend it appropriately.

Jobs and the Command Pattern

Jobs in Backbone are a direct implementation of the Command Pattern.

Command Pattern Overview

The Command Pattern encapsulates a request as an object, allowing:

- Parameterization of clients with different requests.
- Queueing of requests.
- Logging of requests.
- Support for undoable operations.

How Jobs Implement Command

Jobs in Backbone embody these principles:

- Each job is a class that encapsulates a single operation.
- Jobs have a standardized execution method (`execute()`).
- Jobs can be parameterized through their execute method.
- Jobs are self-contained and can be invoked by various clients.

```
#[Job(name: 'create', worker: 'generator', component: 'invoice', package: 'billing')]
class CreateInvoiceJob extends AbstractJob implements JobInterface
{
    public function execute(array $data): Invoice
    {
        // Validate input.
        $this->validateInput($data);

        // Create invoice
        $invoice = new Invoice();
        $invoice->setCustomer($data['customer']);
        $invoice->setItems($data['items']);

        // Return result.
        return $invoice;
    }

    private function validateInput(array $data): void
```

```
    {
        // Validation logic.
    }
}
```

Benefits of the Command Pattern in Jobs

This implementation provides several advantages:

- **Single Responsibility**: Each job has a clear, specific purpose.
- **Reusability**: Jobs can be reused in different contexts.
- **Testability**: Jobs are easy to test in isolation.
- **Extensibility**: New jobs can be added without modifying existing code.
- **Queueability**: Jobs can be serialized and queued for asynchronous processing.

Handlers and the Mediator/Chain of Responsibility Patterns

Handlers in Backbone combine aspects of both the Mediator and Chain of Responsibility patterns.

Mediator Pattern Overview

The Mediator Pattern defines an object that encapsulates how a set of objects interact, promoting loose coupling by preventing objects from referring to each other explicitly.

Chain of Responsibility Overview

The Chain of Responsibility Pattern passes a request along a chain of handlers, with each handler deciding either to process the request or pass it to the next handler.

How Handlers Implement These Patterns

Handlers in Backbone combine these concepts:

- They coordinate interactions between multiple Jobs (Mediator).
- They orchestrate a sequence of operations (Chain of Responsibility).
- They encapsulate complex workflows.

```
#[Handler(name: 'process', worker: 'processor', component: 'invoice', package:
'billing')]
class ProcessInvoiceHandler extends AbstractHandler implements HandlerInterface
{
    public function handle(Invoice $invoice, array $options = []): Result
    {
        // Validate the invoice.
        $validationResult = $this->getJob('validate')->execute($invoice);
```

```
        if (!$validationResult->isValid()) {
            return Result::failure($validationResult->getErrors());
        }

        // Determine processing strategy.
        $strategyName = $options['strategy'] ?? 'default';
        $strategy = $this->getStrategy($strategyName);

        // Process the invoice.
        $processingResult = $strategy->process($invoice);
        if (!$processingResult->isSuccessful()) {
            return Result::failure($processingResult->getErrors());
        }

        // Send notifications.
        $this->getJob('notify')->execute($invoice, $options['notifications'] ?? []);

        // Return success.
        return Result::success(['invoice' => $invoice, 'processed' => true]);
    }
}
```

Benefits of These Patterns in Handlers

This implementation provides several advantages:

- **Decoupling**: Components don't need to know about each other.
- **Centralized Control**: Complex workflows are managed in a single place.
- **Flexibility**: Processing steps can be changed without affecting clients.
- **Transactional Boundaries**: Handlers can manage transactions across multiple operations.
- **Error Handling**: Centralized error handling for multi-step processes.

Strategies and the Strategy Pattern

Strategies in Backbone are a direct implementation of the Strategy Pattern.

Strategy Pattern Overview

The Strategy Pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.

How Strategies Implement the Pattern

Strategies in Backbone embody these principles:

- They provide alternative implementations of an algorithm.

- They share a common interface.
- They can be selected and switched at runtime.

```
#[Strategy(name: 'pdf', worker: 'renderer', component: 'document', package:
'billing')]
class PdfRenderStrategy extends AbstractStrategy implements RenderStrategyInterface
{
    public function render(Document $document): string
    {
        // PDF rendering implementation.
        return $this->pdfRenderer->renderDocument($document);
    }
}

#[Strategy(name: 'html', worker: 'renderer', component: 'document', package:
'billing')]
class HtmlRenderStrategy extends AbstractStrategy implements RenderStrategyInterface
{
    public function render(Document $document): string
    {
        // HTML rendering implementation.
        return $this->htmlRenderer->renderDocument($document);
    }
}
```

Benefits of the Strategy Pattern

This implementation provides several advantages:

- **Encapsulation**: Different algorithms are encapsulated in separate classes.
- **Interchangeability**: Strategies can be swapped without changing client code.
- **Elimination of Conditionals**: Complex conditional logic is replaced with polymorphism.
- **Runtime Selection**: Algorithms can be selected based on runtime conditions.
- **Testability**: Each strategy can be tested independently.

Workers and the Facade Pattern

Workers in Backbone implement the Facade Pattern.

Facade Pattern Overview

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. It defines a higher-level interface that makes the subsystem easier to use.

How Workers Implement the Facade

Workers in Backbone act as facades:

- They provide a simplified interface to complex subsystems.
- They handle the complexity of coordinating jobs, handlers, and strategies.
- They expose domain operations through a clean API.

```
#[Worker(name: 'processor', component: 'invoice', package: 'billing')]
class InvoiceProcessorWorker extends AbstractWorker implements WorkerInterface
{
    // This method is part of the public API.
    public function processInvoice(Invoice $invoice, array $options = []): Result
    {
        // Delegate to the appropriate handler.
        return $this->getHandler('process')->handle($invoice, $options);
    }

    // Another public API method.
    public function validateInvoice(Invoice $invoice): ValidationResult
    {
        // Delegate to a job.
        return $this->getJob('validate')->execute($invoice);
    }
}
```

Benefits of the Facade Pattern in Workers

This implementation provides several advantages:

- **Simplified Interface**: Clients interact with a clean, focused API.
- **Reduced Coupling**: Clients don’t need to know about the subsystem’s components.
- **Unified Entry Point**: Workers provide a single entry point to related functionality.
- **Abstraction**: Implementation details are hidden behind the facade.

Hexagonal Architecture Influence

The overall architecture of Backbone is inspired by Hexagonal Architecture (also known as Ports and Adapters).

Hexagonal Architecture Overview

Hexagonal Architecture aims to create loosely coupled application components that can be easily connected to their software environment by means of ports and adapters.

How Backbone Implements Hexagonal Concepts

Backbone incorporates these principles:

- **Domain-Centric**: The architecture focuses on domain logic.
- **Ports**: Interfaces define how components interact.
- **Adapters**: Implementations connect the domain to external systems.
- **Inversion of Control**: Dependencies point inward toward the domain.

The Package-Component-Worker structure creates clear boundaries within the domain, while Strategies often serve as adapters to external systems.

Practical Application

When applying these design patterns in your Backbone applications:

1. **Identify the Pattern**: Recognize which pattern applies to your situation.
2. **Follow the Template**: Use the appropriate Backbone component.
3. **Respect the Boundaries**: Maintain separation between different components.
4. **Leverage Polymorphism**: Use strategies for variant implementations.
5. **Focus on Composition**: Prefer composition over inheritance.

By understanding and applying these design patterns within Backbone, you can create well-structured, maintainable applications that are flexible enough to adapt to changing requirements.

Last updated on 21/08/2026

File Structure

Recommended File Structure for Projects

Anonymous · 21/08/2026

Recommended File Structure for Projects

This document provides guidelines for organizing your code in Derafu Backbone projects, explaining the rationale behind the recommended structure and best practices for maintaining a clean, maintainable codebase.

Domain-First Organization

Derafu Backbone encourages a domain-first approach to organizing your codebase. This means that the primary organizing principle is the business domain, not technical concerns.

Root Structure

A typical Backbone project is organized as follows:

```
src/
├── Domain1/
├── Domain2/
├── Domain3/
└── Registry.php
config/
└── services.yaml
```

This structure puts domains at the forefront, making it immediately clear what business capabilities your application provides.

Domain Structure

Each domain (represented by a Package) follows a consistent internal structure:

```
Domain/
├── DomainPackage.php
├── Component/
│   ├── Component1.php
│   └── Component2.php
├── Model/
│   ├── Model1.php
│   └── Model2.php
└── Exception/
    └── DomainException.php
```

Key Elements

- **DomainPackage.php**: The package class that serves as the entry point to the domain
- **Component/**: Directory containing all components of this domain
- **Model/**: Directory containing domain models (entities, value objects, etc.)
- **Exception/**: Domain-specific exceptions

Component Structure

Each component has its own structure that houses its workers and related classes:

```
Component/
├── Component.php
└── Worker/
    ├── Worker1.php
    ├── Worker2.php
    ├── Job/
    │   ├── Job1.php
    │   └── Job2.php
    ├── Handler/
    │   ├── Handler1.php
    │   └── Handler2.php
    └── Strategy/
        ├── Strategy1.php
        └── Strategy2.php
```

Key Elements

- **Component.php**: The component class that serves as the entry point to this functional area
- **Worker/**: Directory containing workers and their related classes
- **Job/**: Directory containing jobs used by workers
- **Handler/**: Directory containing handlers used by workers
- **Strategy/**: Directory containing strategies used by handlers and jobs

Namespacing

The file structure directly corresponds to the namespace structure, following PSR-4 autoloading standards:

```
// DomainPackage.php
namespace App\Domain;

// Component.php
namespace App\Domain\Component;

// Worker.php
namespace App\Domain\Component\Worker;

// Job.php
namespace App\Domain\Component\Worker\Job;
```

This clear correspondence between namespaces and directories makes it easy to locate files and

understand their role in the architecture.

The Benefits of This Structure

1. Domain Discovery

The domain-first structure makes it easy to discover what domains your application handles. New team members can quickly understand the application's purpose by examining the top-level directories.

2. Component Cohesion

By grouping related components within a domain, the structure promotes cohesion. Classes that work together are located near each other, making it easier to understand and modify related functionality.

3. Clear Dependencies

The hierarchical structure reflects the dependency hierarchy in Backbone, making it clear how components relate to each other.

4. Consistent Navigation

Once familiar with the structure, developers can quickly navigate to any part of the codebase, even in unfamiliar domains, because the pattern is consistent.

5. Scalable Organization

The structure scales well from small applications to large enterprise systems. As your application grows, you can add new domains without restructuring existing code.

Best Practices

Naming Conventions

Adopting consistent naming conventions enhances the clarity of your codebase:

- **Packages**: Use singular nouns (e.g., `Billing`, not `Bills`).
- **Components**: Use singular nouns that describe their functionality (e.g., `Document`, `Exchange`).
- **Workers**: Use a noun followed by "Worker" (e.g., `BuilderWorker`, `RendererWorker`).
- **Jobs**: Use a verb in the imperative followed by a noun (e.g., `CreateInvoice`, `SendNotification`).
- **Handlers**: Use a verb in the imperative followed by a noun and "Handler" (e.g., `ProcessPaymentHandler`).
- **Strategies**: Use a descriptive adjective or noun followed by the purpose and "Strategy" (e.g.,

`PdfRenderStrategy`, `StripePaymentStrategy`).

File Organization Tips

1. **Group Related Files**: Keep files that are likely to change together in the same directory.
2. **Domain Boundaries**: Be careful about cross-domain dependencies. If components in different domains need to communicate, consider defining interfaces.
3. **Package Size**: If a package grows too large (more than 7-10 components), consider splitting it into multiple packages.
4. **Shared Code**: Place shared code that's used across multiple domains in a separate `Shared` or `Common` package.
5. **Infrastructure Code**: Place infrastructure concerns (like database access, HTTP clients, etc.) in appropriate domains rather than in a separate "infrastructure" layer.

Exception Hierarchy

Match your exception hierarchy to your package/component hierarchy:

```
Exception/
├── DomainException.php (base exception for the domain).
├── ComponentException.php (base exception for a component).
├── SpecificException1.php (specific exception type).
└── SpecificException2.php (specific exception type).
```

This makes error handling more consistent and helps identify the source of exceptions.

Real-World Adaptations

While the recommended structure provides a solid foundation, you may need to adapt it to your specific needs.

Microservice Adaptations

In a microservice architecture, each service might represent a single domain or even a single component:

```
services/
├── billing-service/
│   └── src/
│       └── Billing/
└── customer-service/
    └── src/
        └── Customer/
```

Legacy Integration Adaptations

When integrating with legacy systems, you might need a different structure for adapter code:

```
src/
├── Domain/
│   └── ...
└── Legacy/
    └── Adapter/
        └── ...
```

Infrastructure Concerns

For complex applications, you might introduce additional directories for infrastructure concerns:

```
src/
├── Domain/
├── Infrastructure/
│   ├── Database/
│   ├── Queue/
│   └── Cache/
└── Registry.php
```

However, try to keep these separate from your domain logic and limit dependencies on them from your domain code.

Conclusion

The recommended file structure for Derafu Backbone projects emphasizes domain-driven organization, consistent patterns, and clear separation of concerns. By following these guidelines, you can create codebases that are easy to navigate, maintain, and extend, regardless of the size or complexity of your application.

Remember that the structure should serve your team and your application's needs. While consistency is important, don't be afraid to adapt the recommended structure when necessary to better suit your specific context.

Last updated on 21/08/2026