Docs » Base for Applications Category » Backbone Project » Design Patterns

Design Patterns

Backbone and Design Patterns

Anonymous · 09/09/2026

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.

```php
#[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 09/09/2026