Docs » Base for Applications Category » Backbone Project » File Structure

File Structure

Recommended File Structure for Projects

Anonymous · 09/09/2026

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 09/09/2026