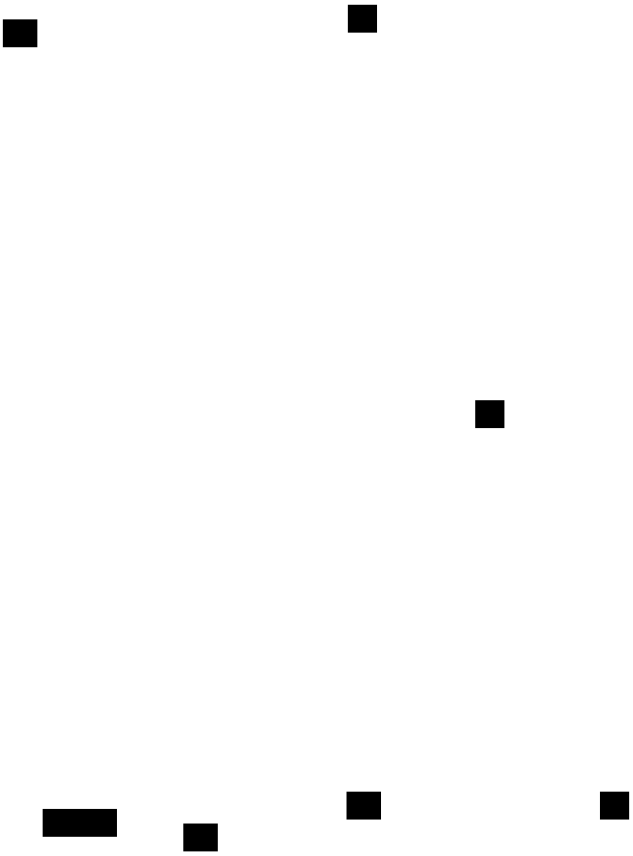
Decision Flow

Decision Flow: Choosing the Right Component

Anonymous · 09/09/2026

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 09/09/2026