ActiveDataFlow: Source and Sink Design Patterns

1. Introduction

This document provides a detailed architectural overview of the `Source` and `Sink` design patterns within the `active_data_flow` Ruby on Rails engine. Inspired by the robust, scalable connector architecture of stream-processing frameworks like Apache Flink, these patterns establish a decoupled, pluggable framework for managing data ingress and egress [1].

The primary goal is to separate the logic of a data flow from the specifics of its external connections. A `DataFlow` object orchestrates the movement and transformation of data, while `Source` and `Sink` objects handle the low-level details of reading from and writing to various systems like message queues, databases, caches, and file systems.

This approach yields a highly modular and configurable system where data pipelines can be reconfigured and extended with minimal code changes.

2. The Source Abstraction

The `ActiveDataFlow::Source` object is a formal abstraction for reading data from an external system. It defines a contract for how a `DataFlow` consumes data, ensuring that any data source can be seamlessly integrated into a flow.

2.1. Base Source Class

All source implementations inherit from the `ActiveDataFlow::Source` base class. This class establishes the core interface, which consists of an initializer that accepts a configuration hash and an `each` method that yields records to a block.

```ruby
# app/models/active_data_flow/source.rb

module ActiveDataFlow
  # Base class for all data sources.
  # It defines the contract for reading data from an external system.
  class Source
    # Defines the configuration parameters required by the source.
    # Subclasses should override this to specify their own needs.
    def self.configuration_attributes
      {}
    end

    # Initializes the source with a given configuration.
    # @param configuration [Hash] A hash containing the source-specific
settings.
    def initialize(configuration)
      @configuration = configuration
    end

    # The core method for a source. It must be implemented by all
subclasses.
    # It is responsible for fetching data and yielding each record to the
provided block.
    def each(&block)
      raise NotImplementedError, "Subclasses must implement the #each
method."
    end
  end
end
```

2.2. Rafka Source Implementation

This implementation provides a source that reads from a `Rafka` stream. Rafka is a library that exposes a Kafka-like API but uses Redis as its backend, making it a lightweight option for message-based communication [2].

```ruby
# app/models/active_data_flow/source/rafka.rb

module ActiveDataFlow
  class Source
    # Implements a Source for reading from a Rafka (Redis) stream.
    class Rafka < ActiveDataFlow::Source
      # Defines the configuration needed to connect to a Rafka stream.
      def self.configuration_attributes
        {
          stream_name: :string,    # The name of the Redis stream.
          consumer_group: :string, # The consumer group to join.
          consumer_name: :string   # A unique name for this consumer.
        }
      end

      # Reads pending messages from the configured Rafka stream and yields
them.
      def each(&block)
        # NOTE: This is a simplified representation. A production
implementation
        # would involve robust error handling, connection management, and
message
        # acknowledgement logic.
        consumer = ::Rafka::Consumer.new(
          @configuration[:stream_name],
          @configuration[:consumer_group],
          @configuration[:consumer_name]
        )

        # The `read` method would fetch a batch of pending messages.
        messages = consumer.read

        messages.each do |message|
          # The block passed by the DataFlow object receives the message.
          yield message
        end
      end
    end
  end
end
```

2.3. File Source Implementation

To demonstrate the pattern’ s flexibility, here is a source that reads records from a local CSV file. This could be used for batch processing tasks where input data is provided in a file drop.

```ruby
# app/models/active_data_flow/source/file.rb

require 'csv'

module ActiveDataFlow
  class Source
    # Implements a Source for reading data from a local file.
    class File < ActiveDataFlow::Source
      # Defines the configuration for the file source.
      def self.configuration_attributes
        {
          file_path: :string, # The absolute path to the input file.
          format: :string      # The file format (e.g., 'csv', 'json').
        }
      end

      # Reads records from the configured file and yields them.
      def each(&block)
        case @configuration[:format]
        when 'csv'
          read_csv(&block)
        else
          raise "Unsupported file format: #{@configuration[:format]}"
        end
      end

      private

      def read_csv(&block)
        # The CSV library is used to parse the file row by row.
        # `headers: true` converts each row into a Hash.
        CSV.foreach(@configuration[:file_path], headers: true) do |row|
          yield row.to_h
        end
      end
    end
  end
end
```

3. The Sink Abstraction

The `ActiveDataFlow::Sink` object provides a formal abstraction for writing data to an external system. It defines a generic interface for data egress, allowing a `DataFlow` to remain agnostic about the final destination of the data it processes.

3.1. Base Sink Class

All sink implementations inherit from the `ActiveDataFlow::Sink` base class. This class establishes a simple contract: an initializer for configuration and a `write` method that accepts a single record.

```
# app/models/active_data_flow/sink.rb

module ActiveDataFlow
  # Base class for all data sinks.
  # It defines the contract for writing data to an external system.
  class Sink
    # Defines the configuration parameters required by the sink.
    def self.configuration_attributes
      {}
    end

    # Initializes the sink with a given configuration.
    # @param configuration [Hash] A hash containing the sink-specific
settings.
    def initialize(configuration)
      @configuration = configuration
    end

    # The core method for a sink. It must be implemented by all subclasses.
    # It is responsible for writing a single record to the destination.
    # @param record [Hash, Object] The data to be written.
    def write(record)
      raise NotImplementedError, "Subclasses must implement the #write
method."
    end
  end
end
```

3.2. ActiveRecord Sink Implementation

This implementation provides a sink that writes records directly to a relational database using `ActiveRecord`. It is highly versatile, as it can be configured to write to any `ActiveRecord` model within the Rails application.

```
# app/models/active_data_flow/sink/active_record.rb

module ActiveDataFlow
  class Sink
    # Implements a Sink for writing records to a database via ActiveRecord.
    class ActiveRecord < ActiveDataFlow::Sink
      # Defines the configuration needed to target an ActiveRecord model.
      def self.configuration_attributes
        {
          model_name: :string # The name of the ActiveRecord model (e.g.,
'Product').
        }
      end

      # Creates a new record in the target table.
      # @param record [Hash] A hash of attributes for the new record.
      def write(record)
        # The model name is resolved to its class constant.
        model_class = @configuration[:model_name].classify.constantize
        # A new record is created with the provided attributes.
        model_class.create!(record)
      end
    end
  end
end
```

3.3. Cache Sink Implementation

This implementation demonstrates a sink that writes data to a Rails cache, such as `ActiveSupport::Cache::RedisCacheStore` [3]. This is useful for `DataFlow` use cases where data has a limited lifecycle or needs to be accessed with very low latency, such as for session-based data.

```ruby
# app/models/active_data_flow/sink/cache.rb

module ActiveDataFlow
  class Sink
    # Implements a Sink for writing data to the Rails cache.
    class Cache < ActiveDataFlow::Sink
      # Defines the configuration for the cache sink.
      def self.configuration_attributes
        {
          key_attribute: :string, # The attribute in the record to use as
the cache key.
          expires_in: :integer   # Optional TTL for the cache entry, in
seconds.
        }
      end

      # Writes a record to the Rails cache.
      # @param record [Hash] The data to be cached.
      def write(record)
        key = record[@configuration[:key_attribute]].to_sym]
        raise "Cache key attribute not found in record" unless key

        options = {}
        if @configuration[:expires_in]
          options[:expires_in] = @configuration[:expires_in].seconds
        end

        # Rails.cache provides a unified interface to the configured cache
store.
        Rails.cache.write(key, record, options)
      end
    end
  end
end
```

4. Integration with DataFlows

The true power of the `Source` and `Sink` abstractions is realized when they are integrated into a `DataFlow` object. By using these patterns, the `DataFlow` is elevated to a pure orchestrator, responsible for coordinating the flow of data and applying transformations, without being coupled to the underlying systems.

4.1. Refactored DataFlow Object

Here is the `ProductUpdateFlow` from our initial example, now refactored to use the `Source` and `Sink` abstractions. The flow is no longer concerned with the details of `Rafka` or `ActiveRecord`. Instead, it operates on the generic `Source` and `Sink` interfaces.

```ruby
# app/data_flows/product_update_flow.rb

class ProductUpdateFlow
  include ActiveDataFlow::DataFlow

  # The configuration now specifies a source and a sink, rather than
  # implementation-specific details like a topic name.
  def self.configuration_attributes
    {
      source: :source, # A configuration object for the source.
      sink: :sink      # A configuration object for the sink.
    }
  end

  # The long-running process is now a simple `run` method within the flow
itself.
  # For more complex scenarios, this could still be delegated to an
ActiveJob.
  def run
    # Instantiate the configured source and sink.
    # The engine would handle resolving the correct class based on the
config.
    source = ActiveDataFlow::Source.new(configuration[:source])
    sink = ActiveDataFlow::Sink.new(configuration[:sink])

    # The core logic is a simple loop: read, transform, write.
    source.each do |message|
      transformed_message = transform(message)
      sink.write(transformed_message)
    end
  end

  private

  # Business logic for transforming the data is kept within the DataFlow.
  def transform(message)
    # This could involve data validation, enrichment, or reshaping.
    # Here, we simply add a timestamp.
    message.merge(processed_at: Time.current)
  end
end
```

4.2. Dynamic Configuration

With this new design, the behavior of the `ProductUpdateFlow` can be changed entirely through configuration, without touching its code. For example:

- **Scenario 1: Kafka to Database**
  - **Source Config**: `{ type: 'Rafka', stream_name: 'inventory_updates', ... }`
  - **Sink Config**: `{ type: 'ActiveRecord', model_name: 'Product' }`
- **Scenario 2: File to Cache**
  - **Source Config**: `{ type: 'File', file_path: '/var/data/daily_prices.csv', format: 'csv' }`
  - **Sink Config**: `{ type: 'Cache', key_attribute: 'product_sku', expires_in: 3600 }`

This dynamic nature makes `DataFlow` objects highly reusable and adaptable to evolving application requirements.

5. References

[1] Apache Flink. (n.d.). *DataStream Connectors*. Retrieved from https://nightlies.apache.org/flink/flink-docs-master/docs/connectors/datastream/overview/

[2] Skroutz. (n.d.). *rafka*. Retrieved from https://github.com/skroutz/rafka

[3] Ruby on Rails Guides. (n.d.). *Caching with Rails: An Overview*. Retrieved from https://guides.rubyonrails.org/caching_with_rails.html