Docs

Mail Project

Derafu Mail

Anonymous · 21/08/2026 · #php

Table of Contents

Mail Project

Derafu Mail

Anonymous · 21/08/2026 · #php

Derafu Mail

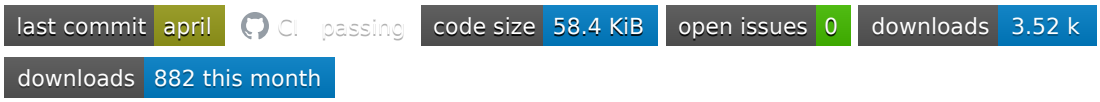
Last updated on 21/08/2026

Introduction

Elegant orchestration of email communications for PHP

Anonymous · 21/08/2026

Elegant orchestration of email communications for PHP



A flexible PHP email library, built on Derafu Backbone architecture, that leverages other libraries and orchestrates the entire sending and receiving process.

Overview

Derafu Mail provides a robust, extensible framework for sending and receiving emails in PHP applications. Built on the Derafu Backbone architecture, it offers a clean, maintainable structure with clear separation of concerns.

Features

- **Clean Architecture**: Follows the Derafu Backbone hierarchical structure.
- **Sending Emails**: SMTP support with easy extensibility for other transport methods.
- **Receiving Emails**: IMAP support with customizable search criteria and filtering.
- **Flexible Configuration**: Comprehensive options for both sending and receiving.
- **Robust Error Handling**: Proper exception handling throughout the library.
- **Attachment Management**: Support for both sending and receiving attachments.
- **Strategy Pattern**: Easily swap out sending/receiving implementations.

Installation

Install the package via Composer:

```
composer require derafu/mail
```

Quick Start

Sending Emails

```
use Derafu\Backbone\Contract\PackageRegistryInterface;
use Derafu\Mail\Model\Message;
use Derafu\Mail\Model\Envelope;
use Derafu\Mail\Model\Postman;
use Symfony\Component\Mime\Address;

// Find the package registry using dependency injection and get a $senderWorker
// Also you can inject directly the SenderWorkerInterface wherever you want.
$packageRegistry = $container->get(PackageRegistryInterface::class);
$mailPackage = $packageRegistry->getPackage('mail');
$exchangeComponent = $mailPackage->getExchangeComponent();
$senderWorker = $exchangeComponent->getSenderWorker();

// Create a message.
$message = new Message();
$message->subject('Hello World')
    ->text('This is a plain text message')
    ->html('<h1>Hello World</h1><p>This is an HTML message</p>')
    ->from(new Address('sender@example.com', 'Sender Name'))
    ->to(new Address('recipient@example.com', 'Recipient Name'));

// Create an envelope and add the message.
$envelope = new Envelope(
    new Address('sender@example.com', 'Sender Name'),
    [new Address('recipient@example.com', 'Recipient Name')]
);
$envelope->addMessage($message);

// Create a postman with SMTP configuration.
$postman = new Postman([
    'strategy' => 'smtp',
    'transport' => [
        'host' => 'smtp.example.com',
        'port' => 587,
        'encryption' => 'tls',
        'username' => 'your_username',
        'password' => 'your_password',
    ],
]);
$postman->addEnvelope($envelope);

// Send the email.
$envelopes = $senderWorker->send($postman);
```

Receiving Emails

```
use Derafu\Backbone\Contract\PackageRegistryInterface;
use Derafu\Mail\Model\Postman;

// Find the package registry using dependency injection and get a $receiverWorker
// Also you can inject directly the ReceiverWorkerInterface wherever you want.
$packageRegistry = $container->get(PackageRegistryInterface::class);
$mailPackage = $packageRegistry->getPackage('mail');
$exchangeComponent = $mailPackage->getExchangeComponent();
$receiverWorker = $exchangeComponent->getReceiverWorker();

// Create a postman with IMAP configuration.
$postman = new Postman([
    'strategy' => 'imap',
    'transport' => [
        'host' => 'imap.example.com',
        'port' => 993,
        'encryption' => 'ssl',
        'username' => 'your_username',
        'password' => 'your_password',
        'mailbox' => 'INBOX',
        'search' => [
            'criteria' => 'UNSEEN',
            'markAsSeen' => true,
            'attachmentFilters' => [
                'extension' => ['pdf', 'doc', 'docx'],
            ],
        ],
    ],
]);

// Receive emails.
$envelopes = $receiverWorker->receive($postman);

// Process received emails.
foreach ($envelopes as $envelope) {
    foreach ($envelope->getMessages() as $message) {
        echo "Subject: " . $message->getSubject() . PHP_EOL;
        echo "From: " . $message->getFrom()[0]->getAddress() . PHP_EOL;
        echo "Body: " . $message->getTextBody() . PHP_EOL;

        // Process attachments.
        foreach ($message->getAttachments() as $attachment) {
            file_put_contents('/path/to/save/' . $attachment->getFilename(),
$attachment->getBody());
        }
    }
}
```

Extending with New Strategies

The library is designed to be easily extended with new sending or receiving strategies:

1. Create a new strategy class implementing `SenderStrategyInterface` or `ReceiverStrategyInterface`.
2. Tag it with the appropriate attribute.

Example:

```
use Derafu\Backbone\Attribute\Strategy;
use
Derafu\Mail\Component\Exchange\Worker\Sender\Strategy\Abstract\AbstractMailerStrategy;
use
Derafu\Mail\Component\Exchange\Worker\Sender\Strategy\Contract\SenderStrategyInterface
;

#[Strategy(name: 'mailgun', worker: 'sender', component: 'exchange', package: 'mail')]
class MailgunStrategy extends AbstractMailerStrategy implements
SenderStrategyInterface
{
    // Implementation of SenderStrategyInterface that leverages
AbstractMailerStrategy.
}
```

Architecture

Derafu Mail follows the Derafu Backbone architecture:

- **Package**: MailPackage - The main entry point.
- **Component**: ExchangeComponent - Handles email exchange.
- **Workers**: SenderWorker and ReceiverWorker - Handle sending and receiving.
- **Handlers**: SendHandler and ReceiveHandler - Orchestrate the process.
- **Strategies**: Implement different methods of sending or receiving.

Last updated on 21/08/2026

Architecture

Derafu Mail Architecture

Anonymous · 21/08/2026

Derafu Mail Architecture

This guide explains the architecture of the Derafu Mail library, which is built on the Derafu Backbone architectural framework. Understanding this architecture will help you effectively use and extend the library.

Architectural Overview

Derafu Mail follows a hierarchical architecture that separates concerns into distinct layers, each with specific responsibilities:

1. **Package Layer**: The entry point and container for all components.
2. **Component Layer**: Functional modules within the package.
3. **Worker Layer**: Task executors within components.
4. **Handler Layer**: Process orchestrators that coordinate strategies.
5. **Strategy Layer**: Specific implementations of mail operations.
6. **Model Layer**: Domain models representing email-related entities.

Package Layer

The `MailPackage` serves as the entry point for the entire library, following the Package pattern from Derafu Backbone.

```
#[Package(name: 'mail')]
class MailPackage extends AbstractPackage implements MailPackageInterface
```

Responsibilities:

- Provides access to components (currently just `ExchangeComponent`).
- Acts as the root of the dependency tree.
- Maintains registry information for service discovery.

Component Layer

The `ExchangeComponent` represents a specific functional area within the mail domain, handling both sending and receiving of emails.

```
#[Component(name: 'exchange', package: 'mail')]
class ExchangeComponent extends AbstractComponent implements
ExchangeComponentInterface
```

Responsibilities:

- Manages workers for sending and receiving emails.
- Provides access to these workers through getter methods.
- Groups related functionality under a single namespace.

Worker Layer

Workers expose the public API for specific tasks:

SenderWorker

```
#[Worker(name: 'sender', component: 'exchange', package: 'mail')]
class SenderWorker extends AbstractWorker implements SenderWorkerInterface
```

Responsibilities:

- Provides a public `send()` method for sending emails.
- Delegates the actual sending process to handlers.
- Manages any worker-specific resources.

ReceiverWorker

```
#[Worker(name: 'receiver', component: 'exchange', package: 'mail')]
class ReceiverWorker extends AbstractWorker implements ReceiverWorkerInterface
```

Responsibilities:

- Provides a public `receive()` method for receiving emails.
- Delegates the receiving process to handlers.
- Manages any worker-specific resources.

Handler Layer

Handlers orchestrate complex processes, selecting appropriate strategies and managing the workflow:

SendHandler

```
class SendHandler extends AbstractHandler
```

Responsibilities:

- Selects the appropriate sending strategy based on configuration.
- Orchestrates the email sending process.
- Handles errors in a centralized manner.
- Manages options and configuration.

ReceiveHandler

```
class ReceiveHandler extends AbstractHandler
```

Responsibilities:

- Selects the appropriate receiving strategy based on configuration.
- Orchestrates the email receiving process.
- Handles errors in a centralized manner.
- Manages options and configuration.

Strategy Layer

Strategies implement specific methods for sending or receiving emails:

SmtpStrategy

```
#[Strategy(name: 'smtp', worker: 'sender', component: 'exchange', package: 'mail')]
class SmtpStrategy extends AbstractMailerStrategy implements SenderStrategyInterface
```

Responsibilities:

- Configures and uses Symfony Mailer for SMTP transport.
- Builds appropriate DSN string based on configuration.
- Handles the actual sending of emails.
- Manages SMTP-specific settings.

ImapStrategy

```
#[Strategy(name: 'imap', worker: 'receiver', component: 'exchange', package: 'mail')]
class ImapStrategy extends AbstractMailboxStrategy implements
ReceiverStrategyInterface
```

Responsibilities:

- Configures and uses PHP-IMAP for IMAP access.
- Builds appropriate DSN string for IMAP connection.
- Searches for and retrieves emails based on criteria.
- Manages IMAP-specific settings.

Abstract Base Classes

The library provides abstract base classes that implement common functionality:

AbstractMailerStrategy

```
abstract class AbstractMailerStrategy extends AbstractStrategy implements
SenderStrategyInterface
```

Responsibilities:

- Provides common code for email sending strategies.
- Handles the creation of Symfony Mailer instances.
- Processes envelopes and messages.

AbstractMailboxStrategy

```
abstract class AbstractMailboxStrategy extends AbstractStrategy implements
ReceiverStrategyInterface
```

Responsibilities:

- Provides common code for email receiving strategies.
- Handles the creation of Mailbox instances.
- Processes received emails into envelopes.

Model Layer

Domain models represent the core entities of the email domain:

Envelope

```
class Envelope extends SymfonyEnvelope implements EnvelopeInterface
```

Responsibilities:

- Contains sender and recipient information.
- Holds one or more messages.
- Provides a container for email communication.

Message

```
class Message extends SymfonyEmail implements MessageInterface
```

Responsibilities:

- Represents an individual email message.
- Contains subject, body, attachments, etc.
- Tracks sending/receiving errors.

Postman

```
class Postman implements PostmanInterface
```

Responsibilities:

- Acts as a transport container for envelopes.
- Holds configuration options for sending/receiving.
- Provides a unified interface for email operations.

Mailbox

```
class Mailbox implements MailboxInterface
```

Responsibilities:

- Represents an email mailbox (IMAP folder).
- Provides methods for searching and retrieving emails.
- Handles connection to mail servers.

Data Flow Examples

Sending an Email

1. Client code creates a Message and adds it to an Envelope.
2. Envelope is added to a Postman with SMTP configuration.
3. Client calls SenderWorker's send() method with the Postman.
4. SenderWorker delegates to SendHandler.
5. SendHandler selects SmtpStrategy based on configuration.
6. SmtpStrategy uses Symfony Mailer to send the emails.
7. Results are returned through the same chain.

Receiving Emails

1. Client creates a Postman with IMAP configuration.
2. Client calls ReceiverWorker's receive() method with the Postman.
3. ReceiverWorker delegates to ReceiveHandler.
4. ReceiveHandler selects ImapStrategy based on configuration.
5. ImapStrategy connects to the mailbox and retrieves emails.
6. Retrieved emails are converted to Envelopes with Messages.
7. Envelopes are added to the Postman and returned.

Extending the Library

The architecture makes it easy to extend the library with new strategies:

1. **New Sending Strategy**: Create a class that extends AbstractMailerStrategy or implements SenderStrategyInterface.
2. **New Receiving Strategy**: Create a class that extends AbstractMailboxStrategy or implements ReceiverStrategyInterface.
3. **Tag with Attribute**: Use the #[Strategy] attribute for automatic discovery.
4. **Use via Configuration**: Specify your strategy in the Postman options.

Key Benefits of This Architecture

1. **Separation of Concerns**: Each class has a single, well-defined responsibility.
2. **Extensibility**: Easy to add new strategies without modifying existing code.
3. **Testability**: Clean interfaces make unit testing straightforward.
4. **Configurability**: Comprehensive options at every level.
5. **Maintainability**: Clear structure makes code easy to understand and modify.

Conclusion

Derafu Mail's architecture provides a solid foundation for email operations in PHP applications. By leveraging the Derafu Backbone patterns, it achieves a clean separation of concerns while remaining

flexible and extensible.

Understanding this architecture will help you effectively use the library and extend it with new capabilities when needed.

Last updated on 21/08/2026

Custom Sender Strategy

Creating a Custom Sender Strategy for Derafu Mail

Anonymous · 21/08/2026

Creating a Custom Sender Strategy for Derafu Mail

This guide explains how to create custom sender strategies for Derafu Mail. Sender strategies allow you to implement different methods of sending emails beyond the default SMTP implementation.

Understanding Sender Strategies

A sender strategy in Derafu Mail is responsible for the actual transmission of email messages. The library comes with an SMTP implementation, but you might want to add support for:

- API-based email services (SendGrid, Mailgun, Postmark, etc.).
- Custom internal email systems.
- Database-based email queues.
- Testing/mock implementations.

Option 1: Extending AbstractMailerStrategy

The simplest approach is to extend the `AbstractMailerStrategy` class, which provides reusable functionality for most email sending scenarios that utilize Symfony Mailer internally.

When to Use This Approach

- When your email service has a Symfony Transport implementation.
- When you need to reuse the core sending logic.
- When your strategy follows a similar workflow to the SMTP strategy.

Implementation Steps

1. Create a new class that extends `AbstractMailerStrategy`:

```
<?php

declare(strict_types=1);

```

```
namespace YourNamespace\Strategy;

use Derafu\Backbone\Attribute\Strategy;
use Derafu\Config\Contract\OptionsInterface;
use
Derafu\Mail\Component\Exchange\Worker\Sender\Strategy\Abstract\AbstractMailerStrategy;
use
Derafu\Mail\Component\Exchange\Worker\Sender\Strategy\Contract\SenderStrategyInterface
;

#[Strategy(name: 'mailgun', worker: 'sender', component: 'exchange', package: 'mail')]
class MailgunStrategy extends AbstractMailerStrategy implements
SenderStrategyInterface
{
    /**
     * Schema of the options.
     *
     * @var array<string,array>
     */
    protected array $optionsSchema = [
        'strategy' => [
            'types' => 'string',
            'default' => 'mailgun',
        ],
        'transport' => [
            'types' => 'array',
            'schema' => [
                'api_key' => [
                    'types' => 'string',
                    'required' => true,
                ],
                'domain' => [
                    'types' => 'string',
                    'required' => true,
                ],
                'region' => [
                    'types' => 'string',
                    'default' => 'us',
                ],
                'dsn' => [
                    'types' => 'string',
                ],
                'endpoint' => [
                    'types' => 'string',
                ],
            ],
        ],
    ];

    /**
     * {@inheritDoc}
```

```
     */
    protected function resolveDsn(OptionsInterface $options): string
    {
        $transportOptions = $options->get('transport');

        if (!empty($transportOptions['dsn'])) {
            return $transportOptions['dsn'];
        }

        // Construct the DSN for Mailgun using Symfony's format.
        $dsn = sprintf(
            'mailgun://%s@%s?region=%s',
            $transportOptions['api_key'],
            $transportOptions['domain'],
            $transportOptions['region'] ?? 'us'
        );

        $options->set('transport.dsn', $dsn);

        return $dsn;
    }

    /**
     * {@inheritDoc}
     */
    protected function resolveEndpoint(OptionsInterface $options): string
    {
        $transportOptions = $options->get('transport');

        if (!empty($transportOptions['endpoint'])) {
            return $transportOptions['endpoint'];
        }

        $endpoint = sprintf(
            'mailgun://%s',
            $transportOptions['domain']
        );

        $options->set('transport.endpoint', $endpoint);

        return $endpoint;
    }
}
```

2. Define your options schema to specify the required configuration.

3. Implement the `resolveDsn()` method to build the appropriate DSN string for your service.

4. Implement the `resolveEndpoint()` method to provide a human-readable representation of the endpoint.

Key Benefits

- Leverages existing functionality from the abstract class.
- Reduces code duplication.
- Ensures consistent behavior across strategies.
- Automatically gets error handling and envelope processing.

Option 2: Implementing SenderStrategyInterface

For more specialized use cases, you can implement the `SenderStrategyInterface` directly.

When to Use This Approach

- When your sending mechanism is fundamentally different from Symfony Mailer.
- When you need complete control over the sending process.
- When you want to avoid dependencies on Symfony components.
- For custom integrations with proprietary systems

Implementation Steps

1. Create a new class that implements `SenderStrategyInterface`:

```
<?php

declare(strict_types=1);

namespace YourNamespace\Strategy;

use Derafu\Backbone\Abstract\AbstractStrategy;
use Derafu\Backbone\Attribute\Strategy;
use Derafu\Config\Contract\OptionsInterface;
use
Derafu\Mail\Component\Exchange\Worker\Sender\Strategy\Contract\SenderStrategyInterface
;
use Derafu\Mail\Exception\MailException;
use Derafu\Mail\Model\Contract\PostmanInterface;
use GuzzleHttp\Client;
use Throwable;

#[Strategy(name: 'custom-api', worker: 'sender', component: 'exchange', package:
'mail')]
class CustomApiStrategy extends AbstractStrategy implements SenderStrategyInterface
{
```

```
    /**
     * Schema of the options.
     *
     * @var array<string,array>
     */
    protected array $optionsSchema = [
        'strategy' => [
            'types' => 'string',
            'default' => 'custom-api',
        ],
        'transport' => [
            'types' => 'array',
            'schema' => [
                'api_url' => [
                    'types' => 'string',
                    'required' => true,
                ],
                'api_key' => [
                    'types' => 'string',
                    'required' => true,
                ],
                // Add any other configuration options needed.
            ],
        ],
    ];

    /**
     * HTTP client for API requests.
     */
    private Client $httpClient;

    /**
     * Constructor.
     */
    public function __construct()
    {
        $this->httpClient = new Client();
    }

    /**
     * {@inheritDoc}
     */
    public function send(PostmanInterface $postman): array
    {
        $options = $this->resolveOptions($postman->getOptions());
        $transportOptions = $options->get('transport');

        $apiUrl = $transportOptions['api_url'];
        $apiKey = $transportOptions['api_key'];

        foreach ($postman->getEnvelopes() as $envelope) {
```

```
            foreach ($envelope->getMessages() as $message) {
                try {
                    // Transform the message to your API format.
                    $payload = $this->transformMessageToApiPayload($message,
$envelope);

                    // Send via your custom API.
                    $response = $this->httpClient->post($apiUrl, [
                        'headers' => [
                            'Authorization' => 'Bearer ' . $apiKey,
                            'Content-Type' => 'application/json',
                        ],
                        'json' => $payload,
                    ]);

                    // Process response if needed.
                    if ($response->getStatusCode() >= 400) {
                        throw new MailException('API returned error: ' .
$response->getBody());
                    }

                } catch (Throwable $e) {
                    $message->error($e);
                }
            }
        }

        return $postman->getEnvelopes();
    }

    /**
     * Transforms a message to the format expected by the API.
     *
     * @param MessageInterface $message
     * @param EnvelopeInterface $envelope
     * @return array
     */
    private function transformMessageToApiPayload($message, $envelope): array
    {
        // Implement the transformation logic for your specific API.
        // This is where you map the Message and Envelope properties
        // to whatever format your API expects.

        return [
            'from' => $this->formatAddress($message->getFrom()[0]),
            'to' => array_map([$this, 'formatAddress'], $message->getTo()),
            'subject' => $message->getSubject(),
            'text' => $message->getTextBody(),
            'html' => $message->getHtmlBody(),
            // Handle attachments, CC, BCC, etc.
        ];
```

```
    }

    /**
     * Formats an email address for the API.
     */
    private function formatAddress($address): array
    {
        return [
            'email' => $address->getAddress(),
            'name' => $address->getName(),
        ];
    }
}
```

2. Define your options schema to specify the required configuration.

3. Implement the `send()` method to handle the entire sending process.

4. Add any helper methods needed for your specific implementation.

Key Considerations

When implementing from scratch:

- **Error Handling**: You must handle all exceptions and errors.
- **Message Processing**: You need to transform Derafu Mail messages to your API format.
- **State Management**: Consider how to track message status and handle failures.
- **Testing**: Create test cases for various scenarios and error conditions.

Usage

Once you've created your custom strategy, you can use it by specifying its name in the Postman configuration:

```
$postman = new Postman([
    'strategy' => 'mailgun', // Or 'custom-api'
    'transport' => [
        // Strategy-specific configuration options.
        'api_key' => 'your-api-key',
        'domain' => 'your-domain.com',
        // Other options...
    ],
]);
```

Best Practices

1. **Error Handling**: Always catch and properly handle exceptions.
2. **Logging**: Add appropriate logging to help troubleshoot issues.
3. **Configuration Validation**: Use the options schema to validate configuration.
4. **Comprehensive Documentation**: Document your strategy's requirements.
5. **Unit Testing**: Create tests for various scenarios including error cases.

Conclusion

Creating custom sender strategies allows you to extend Derafu Mail to work with any email service or system. Whether you extend the abstract class or implement the interface directly depends on your specific needs and how much you want to leverage the existing infrastructure.

For most API-based email services that have Symfony Transport implementations, extending `AbstractMailerStrategy` is recommended. For completely custom implementations, implementing `SenderStrategyInterface` directly gives you maximum flexibility.

Last updated on 21/08/2026

Custom Receiver Strategy

Creating a Custom Receiver Strategy for Derafu Mail

Anonymous · 21/08/2026

Creating a Custom Receiver Strategy for Derafu Mail

This guide explains how to create custom receiver strategies for Derafu Mail. Receiver strategies allow you to implement different methods of retrieving emails beyond the default IMAP implementation.

Understanding Receiver Strategies

A receiver strategy in Derafu Mail is responsible for connecting to mail sources and retrieving messages. The library comes with an IMAP implementation, but you might want to add support for:

- API-based email services (Gmail API, Microsoft Graph, etc.).
- Custom email storage systems.
- Database-stored emails.
- Webhook receivers for incoming emails.
- Testing/mock implementations.

Option 1: Extending AbstractMailboxStrategy

The simplest approach is to extend the `AbstractMailboxStrategy` class, which provides reusable functionality for email retrieval scenarios that use a mailbox-like interface.

When to Use This Approach

- When your email source follows a mailbox paradigm.
- When you can use the PHP-IMAP library or similar interfaces.
- When you need to reuse common mailbox operations.
- When your strategy follows a similar workflow to the IMAP strategy.

Implementation Steps

1. Create a new class that extends `AbstractMailboxStrategy`:

```
<?php
```

```
declare(strict_types=1);

namespace YourNamespace\Strategy;

use Derafu\Backbone\Attribute\Strategy;
use Derafu\Config\Contract\OptionsInterface;
use
Derafu\Mail\Component\Exchange\Worker\Receiver\Strategy\Abstract\AbstractMailboxStrate
gy;
use
Derafu\Mail\Component\Exchange\Worker\Receiver\Strategy\Contract\ReceiverStrategyInter
face;
use Derafu\Mail\Model\Mailbox;
use Derafu\Mail\Model\Contract\MailboxInterface;

#[Strategy(name: 'gmail-api', worker: 'receiver', component: 'exchange', package:
'mail')]
class GmailApiStrategy extends AbstractMailboxStrategy implements
ReceiverStrategyInterface
{
    /**
     * Schema of the options.
     *
     * @var array<string,array>
     */
    protected array $optionsSchema = [
        'strategy' => [
            'types' => 'string',
            'default' => 'gmail-api',
        ],
        'transport' => [
            'types' => 'array',
            'schema' => [
                'client_id' => [
                    'types' => 'string',
                    'required' => true,
                ],
                'client_secret' => [
                    'types' => 'string',
                    'required' => true,
                ],
                'refresh_token' => [
                    'types' => 'string',
                    'required' => true,
                ],
                'user_email' => [
                    'types' => 'string',
                    'required' => true,
                ],
                'label' => [
                    'types' => 'string',
```

```
                    'default' => 'INBOX',
                ],
                'dsn' => [
                    'types' => 'string',
                ],
                'endpoint' => [
                    'types' => 'string',
                ],
                'search' => [
                    'types' => 'array',
                    'schema' => [
                        'query' => [
                            'types' => 'string',
                            'default' => 'is:unread',
                        ],
                        'markAsSeen' => [
                            'types' => 'bool',
                            'default' => false,
                        ],
                        'attachmentFilters' => [
                            'types' => 'array',
                            'default' => [],
                        ],
                    ],
                ],
            ],
        ],
    ];

    /**
     * {@inheritDoc}
     */
    protected function createMailbox(OptionsInterface $options): MailboxInterface
    {
        // Instead of using the standard Mailbox, create a specialized Gmail API
mailbox.
        // This could be a custom class that implements MailboxInterface.

        $transportOptions = $options->get('transport');

        // This would be a custom implementation for Gmail API.
        return new GmailApiMailbox(
            $transportOptions['client_id'],
            $transportOptions['client_secret'],
            $transportOptions['refresh_token'],
            $transportOptions['user_email'],
            $transportOptions['label'] ?? 'INBOX'
        );
    }

    /**
```

```
     * {@inheritDoc}
     */
    protected function resolveDsn(OptionsInterface $options): string
    {
        $transportOptions = $options->get('transport');

        if (!empty($transportOptions['dsn'])) {
            return $transportOptions['dsn'];
        }

        // Construct a representative DSN for Gmail API.
        $dsn = sprintf(
            'gmail-api://%s',
            $transportOptions['user_email']
        );

        $options->set('transport.dsn', $dsn);

        return $dsn;
    }

    /**
     * {@inheritDoc}
     */
    protected function resolveEndpoint(OptionsInterface $options): string
    {
        $transportOptions = $options->get('transport');

        if (!empty($transportOptions['endpoint'])) {
            return $transportOptions['endpoint'];
        }

        $endpoint = sprintf(
            'https://gmail.googleapis.com/gmail/v1/users/%s',
            $transportOptions['user_email']
        );

        $options->set('transport.endpoint', $endpoint);

        return $endpoint;
    }
}

/**
 * Custom Mailbox implementation for Gmail API.
 * This class would need to implement all methods from MailboxInterface
 * but would use the Gmail API instead of IMAP.
 */
class GmailApiMailbox implements MailboxInterface
{
    // Implement all required methods from MailboxInterface.
```

```
    // This would use Google API Client or similar to fetch emails.
}
```

2. Define your options schema to specify the required configuration.

3. Override the `createMailbox()` method to return your custom mailbox implementation.

4. Implement the `resolveDsn()` and `resolveEndpoint()` methods.

5. Create a custom mailbox class that implements `MailboxInterface` if needed.

Key Benefits

- Leverages existing workflow and error handling.
- Preserves compatibility with the rest of the library.
- Reuses attachment filtering and other common functionality.
- Maintains consistent behavior across strategies.

Option 2: Implementing ReceiverStrategyInterface

For more specialized use cases, you can implement the `ReceiverStrategyInterface` directly.

When to Use This Approach

- When your receiving mechanism is fundamentally different from a mailbox model.
- When you need complete control over the receiving process.
- When you're creating a strategy for webhooks or other non-polling mechanisms.
- For custom integrations with proprietary systems.

Implementation Steps

1. Create a new class that implements `ReceiverStrategyInterface`:

```
<?php

declare(strict_types=1);

namespace YourNamespace\Strategy;

use Derafu\Backbone\Abstract\AbstractStrategy;
use Derafu\Backbone\Attribute\Strategy;
use
Derafu\Mail\Component\Exchange\Worker\Receiver\Strategy\Contract\ReceiverStrategyInter
```

```
face;
use Derafu\Mail\Exception\MailException;
use Derafu\Mail\Model\Contract\EnvelopeInterface;
use Derafu\Mail\Model\Contract\MessageInterface;
use Derafu\Mail\Model\Contract\PostmanInterface;
use Derafu\Mail\Model\Envelope;
use Derafu\Mail\Model\Message;
use Symfony\Component\Mime\Address;
use Throwable;

#[Strategy(name: 'webhook-receiver', worker: 'receiver', component: 'exchange',
package: 'mail')]
class WebhookReceiverStrategy extends AbstractStrategy implements
ReceiverStrategyInterface
{
    /**
     * Schema of the options.
     *
     * @var array<string,array>
     */
    protected array $optionsSchema = [
        'strategy' => [
            'types' => 'string',
            'default' => 'webhook-receiver',
        ],
        'transport' => [
            'types' => 'array',
            'schema' => [
                'webhook_data' => [
                    'types' => 'array',
                    'required' => true,
                ],
                'secret_key' => [
                    'types' => 'string',
                    'default' => '',
                ],
                // Add any other configuration options needed.
            ],
        ],
    ];

    /**
     * {@inheritDoc}
     */
    public function receive(PostmanInterface $postman): array
    {
        $options = $this->resolveOptions($postman->getOptions());
        $transportOptions = $options->get('transport');

        $webhookData = $transportOptions['webhook_data'];
        $secretKey = $transportOptions['secret_key'] ?? '';
```

```
        try {
            // Validate webhook data if a secret is configured.
            if ($secretKey && !$this->validateWebhookSignature($webhookData,
$secretKey)) {
                throw new MailException('Invalid webhook signature');
            }

            // Process the webhook data to extract email information.
            $emails = $this->processWebhookData($webhookData);

            // Create envelopes and add them to the postman.
            foreach ($emails as $emailData) {
                $envelope = $this->createEnvelope($emailData);
                $postman->addEnvelope($envelope);
            }

            // Optionally acknowledge receipt to the webhook source.
            $this->acknowledgeReceipt($webhookData);

        } catch (Throwable $e) {
            throw new MailException(
                sprintf(
                    'An error occurred while processing webhook data: %s',
                    $e->getMessage()
                ),
                0,
                $e
            );
        }

        return $postman->getEnvelopes();
    }

    /**
     * Validates the webhook signature to ensure authenticity.
     *
     * @param array $webhookData
     * @param string $secretKey
     * @return bool
     */
    private function validateWebhookSignature(array $webhookData, string $secretKey):
bool
    {
        // Implement signature validation logic.
        // The exact implementation depends on how your webhook source signs requests.
        return true; // Placeholder.
    }

    /**
     * Processes the webhook data to extract email information.
     *
```

```
     * @param array $webhookData
     * @return array
     */
    private function processWebhookData(array $webhookData): array
    {
        // Convert webhook data to a standardized email format.
        // This will depend entirely on the webhook format you're receiving.

        // Placeholder implementation - extract email data from webhook.
        $emails = [];

        if (isset($webhookData['emails']) && is_array($webhookData['emails'])) {
            foreach ($webhookData['emails'] as $email) {
                $emails[] = [
                    'from' => $email['sender'] ?? '',
                    'from_name' => $email['sender_name'] ?? '',
                    'to' => $email['recipient'] ?? '',
                    'to_name' => $email['recipient_name'] ?? '',
                    'subject' => $email['subject'] ?? '',
                    'text_body' => $email['plain_text'] ?? '',
                    'html_body' => $email['html'] ?? '',
                    'attachments' => $email['attachments'] ?? [],
                ];
            }
        }

        return $emails;
    }

    /**
     * Creates an envelope from email data.
     *
     * @param array $emailData
     * @return EnvelopeInterface
     */
    private function createEnvelope(array $emailData): EnvelopeInterface
    {
        // Create a sender address.
        $sender = new Address(
            $emailData['from'],
            $emailData['from_name'] ?? ''
        );

        // Create recipient addresses.
        $recipients = [
            new Address(
                $emailData['to'],
                $emailData['to_name'] ?? ''
            )
        ];
```

```
        // Create the envelope.
        $envelope = new Envelope($sender, $recipients);

        // Create and add the message.
        $message = $this->createMessage($emailData);
        $envelope->addMessage($message);

        return $envelope;
    }

    /**
     * Creates a message from email data.
     *
     * @param array $emailData
     * @return MessageInterface
     */
    private function createMessage(array $emailData): MessageInterface
    {
        // Create the message.
        $message = new Message();

        // Set basic properties
        $message->subject($emailData['subject'] ?? '');

        if (!empty($emailData['text_body'])) {
            $message->text($emailData['text_body']);
        }

        if (!empty($emailData['html_body'])) {
            $message->html($emailData['html_body']);
        }

        $message->from(new Address(
            $emailData['from'],
            $emailData['from_name'] ?? ''
        ));

        $message->to(new Address(
            $emailData['to'],
            $emailData['to_name'] ?? ''
        ));

        // Process attachments if any.
        if (!empty($emailData['attachments']) && is_array($emailData['attachments']))
{
            foreach ($emailData['attachments'] as $attachment) {
                if (isset($attachment['content'], $attachment['name'],
$attachment['type'])) {
                    $content = base64_decode($attachment['content']);
                    $message->attach(
                        $content,
```

```
                        $attachment['name'],
                        $attachment['type']
                    );
                }
            }
        }

        return $message;
    }

    /**
     * Acknowledges receipt to the webhook source if needed.
     *
     * @param array $webhookData
     * @return void
     */
    private function acknowledgeReceipt(array $webhookData): void
    {
        // Some webhook providers require an acknowledgement.
        // Implement if needed for your specific case.
    }
}
```

2. Define your options schema to specify the required configuration.

3. Implement the `receive()` method to handle the entire receiving process.

4. Add helper methods for your specific implementation needs.

Key Considerations

When implementing from scratch:

- **Error Handling**: Properly catch and handle all exceptions.
- **Data Transformation**: Carefully map external data to Derafu Mail models.
- **Security**: Validate webhook signatures or implement other security measures.
- **Consistency**: Ensure your implementation behaves consistently with other strategies.
- **Testing**: Create comprehensive tests for your implementation.

Usage

Once you've created your custom strategy, you can use it by specifying its name in the Postman configuration:

```
$postman = new Postman([
    'strategy' => 'gmail-api', // Or 'webhook-receiver'
    'transport' => [
        // Strategy-specific configuration options.
        'client_id' => 'your-client-id',
        'client_secret' => 'your-client-secret',
        'refresh_token' => 'your-refresh-token',
        'user_email' => 'user@example.com',
        // Other options...
    ],
]);

$receiverWorker = $exchangeComponent->getReceiverWorker();
$envelopes = $receiverWorker->receive($postman);
```

Best Practices

1. **Error Handling**: Always catch and handle exceptions appropriately.
2. **Logging**: Add detailed logging to help troubleshoot issues.
3. **Configuration Validation**: Use the options schema to validate configuration.
4. **Rate Limiting**: Consider rate limits for API-based strategies.
5. **Pagination**: Implement proper pagination for retrieving large volumes of emails.
6. **Performance**: Be mindful of memory usage when dealing with attachments.
7. **Documentation**: Document your strategy's specific requirements and limitations.

Conclusion

Creating custom receiver strategies allows you to extend Derafu Mail to work with any email source or system. Whether you extend the abstract class or implement the interface directly depends on your specific needs and how much you want to leverage the existing infrastructure.

For sources that follow a mailbox-like model, extending `AbstractMailboxStrategy` is recommended. For completely different mechanisms like webhooks, implementing `ReceiverStrategyInterface` directly gives you maximum flexibility.

Last updated on 21/08/2026