Custom Sender Strategy

Creating a Custom Sender Strategy for Derafu Mail

Anonymous · 09/09/2026

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 09/09/2026