
PMG Queue Documentation

Release 3.0.0

Christopher Davis

Feb 05, 2018

Contents

1	Contents	3
1.1	Messages	3
1.2	Producers	4
1.3	Consumers	5
1.4	Message Handlers	10
1.5	Drivers & Internals	13
2	Installation & Examples	17
3	READ THIS: Glossary & Core Concepts	19

`pmg/queue` is a production ready queue framework that powers many internal projects at [PMG](#).

It's simple and extensible a number of features we've found to be the most useful including automatic retries and multi-queue support.

1.1 Messages

Messages are objects that implement the `PMG\Queue\Message` interface. These objects are meant to be *serializable* and contain everything you need for a *handler* to do its job.

A message to send an alert to a user might look something like this:

1.1.1 Example Message

```
<?php
use PMG\Queue\Message;

final class SendAlert implements Message
{
    private $userId;

    public function __construct($userId)
    {
        $this->userId = $userId;
    }

    public function getUserId()
    {
        return $this->userId;
    }
}
```

Because messages are serialized to be put in a persistent backend they shouldn't include objects that require state. In the example above the message just contains a user's identifier rather than the full object. The *handler* would then look up the user.

See *Consumers* and *Producers* for more information about handlers and messages fit into the system as a whole.

1.2 Producers

Producers add messages to a driver backed for the *consumer* to pick up and handle.

interface Producer

Namespace PMG\Queue

send (PMG\Queue\Message \$message)

Send a message to a driver backend.

Parameters

- **\$message** – The message to send into the queue

Throws PMG\Queue\Exception\QueueNotFound if the message can't be routed to an appropriate queue.

The default producer implementation takes a driver and a router as its constructor arguments and uses the router (explained below) to send its messages into a drivers specific queue.

```
<?php

use PMG\Queue\DefaultProducer;
use PMG\Queue\Router\SimpleRouter;

$router = new SimpleRouter('queueName');

/** @var PMG\Queue\Driver $driver */
$producer = new DefaultProducer($driver, $router);
```

1.2.1 Routers

pmg/queue is built with multi-queue support in mind. To accomplish that on the producer side of things an implementation of PMG\Queue\Router is used.

interface Router

Namespace PMG\Queue

queueFor (PMG\Queue\Message \$message)

Looks a queue name for a given message.

Parameters

- **\$message** – the message to route

Returns A string queue name if found, null otherwise.

Return type string or null

Routing all Message to a Single Queue

Use PMG\Queue\SimpleRouter, which takes a queue name in the constructor and always returns it.

```
<?php

use PMG\Queue\Router\SimpleRouter;

// all message will go in the "queueName" queue
$router = new SimpleRouter('queueName');
```


Routing Messages Based on Their Name

Use `PMG\Queue\Router\MappingRouter`, which takes a map of message name => queue name pairs to its constructor.

```
<?php

use PMG\Queue\Router\MappingRouter;

$router = new MappingRouter([
    // the `SendAlert` message will go into the `Alerts` queue
    'SendAlert' => 'Alerts',
]);
```

Falling Back to a Default Queue

To avoid `QueueNotFound` exceptions, it's often a good idea to use `PMG\Queue\Router\FallbackRouter`.

```
<?php

use PMG\Queue\DefaultProducer;
use PMG\Queue\SimpleMessage;
use PMG\Queue\Router\FallbackRouter;
use PMG\Queue\Router\MappingRouter;

$router = new FallbackRouter(new MappingRouter([
    'SendAlert' => 'Alerts',
]), 'defaultQueue');

$producer = new DefaultProducer($driver, $router);

// goes into the `Alerts` queue
$producer->send(new SimpleMessage('SendAlert'));

// goes into `defaultQueue`
$producer->send(new SimpleMessage('OtherThing'));
```

1.3 Consumers

Implementations of `PMG\Queue\Consumer` pull message out of a driver backend and handle (process) them in some way. The default consumer accomplishes this a *message handler*.

In all cases `$queueName` in the consume should correspond to queues into which your *producer* put messages.

interface Consumer

Namespace `PMG\Queue`

run (*\$queueName*, *MessageLifecycle \$lifecycle=null*)
Consume and handle messages from *\$queueName* indefinitely.

Parameters

- **\$queueName** (*string*) – The queue from which the messages will be processed.

- **\$lifecycle** (*MessageLifecycle|null*) – An optional message lifecycle.

Throws `PMG\Queue\Exception\DriverError` If some things goes wrong with the underlying driver. Generally this happens if the persistent backend goes down or is unreachable. Without the driver the consumer can't do its work.

Returns An exit code

Return type `int`

once (*\$queueName, MessageLifecycle \$lifecycle=null*)
Consume and handle a single message from *\$queueName*

Parameters

- **\$queueName** (*string*) – The queue from which the messages will be processed.
- **\$lifecycle** (*MessageLifecycle|null*) – An optional message lifecycle.

Throws `PMG\Queue\Exception\DriverError` If some things goes wrong with the underlying driver. Generally this happens if the persistent backend goes down or is unreachable. Without the driver the consumer can't do its work.

Returns True or false to indicate if the message was handled successfully. null if no message was handled.

Return type `boolean` or `null`

stop (*int \$code*)
Used on a running consumer this will tell it to gracefully stop on its next iteration.

Parameters

- **\$code** (*int*) – The exit code to return from *run*

The script to run your consumer might look something like this. Check out the [handlers](#) documentation for more information about what *\$handler* is below.

```
<?php

use PMG\Queue\DefaultConsumer;
use PMG\Queue\Driver\MemoryDriver;

$driver = new MemoryDriver();

/** @var PMG\Queue\MessageHandler $handler */
$consumer = new DefaultConsumer($driver, $handler);

exit($consumer->run(isset($argv[1]) ? $argv[1] : 'defaultQueue'));
```

1.3.1 Retrying Messages

When a message fails – by throwing an exception or returns false from a `MessageHandler` – the consumer puts it back in the queue to retry up to 5 times by default. This behavior can be adjusted by providing a `RetrySpec` as the third argument to `DefaultConsumer`'s constructor. *pmg/queue* provides a few by default.

Retry specs look at `PMG\Queue\Envelope` instances, not raw messages. See the [internals documentation](#) for more info about them.

interface `RetrySpec`

Namespace `PMG\Queue`

canRetry (*PMG\Queue\Envelope \$env*)

Inspects an envelop to see if it can retry again.

Parameters

- **\$env** – The message envelope to check

Returns true if the message can be retried, false otherwise.

Return type boolean

Limited Retries

Use `PMG\Queue\Retry\LimitedSpec`.

```
<?php
use PMG\Queue\DefaultConsumer;
use PMG\Queue\Retry\LimitedSpec;

// five retries by default. This is what the consumer does automatically
$retry = new LimitedSpec();

// Or limit to a specific number of retries
$retry = new LimitedSpec(2);

// $driver and $handler as above
$consumer = new DefaultConsumer($driver, $handler, $retry);
```

Never Retry a Message

Sometimes you don't want to retry a message, for those cases use `PMG\Queue\Retry\NeverSpec`.

```
<?php
use PMG\Queue\DefaultConsumer;
use PMG\Queue\Retry\NeverSpec;

$retry = new NeverSpec();

// $driver and $handler as above
$consumer = new DefaultConsumer($driver, $handler, $retry);
```

1.3.2 Logging

When something goes wrong `DefaultConsumer` logs it with a [PSR-3 Logger](#) implementation. The default is to use a `NullLogger`, but you can provide your own logger as the fourth argument to `DefaultConsumer`'s constructor.

```
<?php
use PMG\Queue\DefaultConsumer;

$monolog = new Monolog\Logger('yourApp');

// $driver, $handler, $retry as above
$consumer = new DefaultConsumer($driver, $handler, $retry, $monolog);
```

1.3.3 Using Message Lifecycles

A `MessageLifecycle` implementation provides a look into a message as it moves through the consumer. The goal is to allow an application to hook into a consumer processing to take actions they want. Say an application requires sending a notification when a message fails and will not be retried.

```
<?php

use PMG\Queue\Lifecycle\NullLifecycle;
use App\Notifications\Notifier;
use App\Notifications\Notification;

// NullLifecycle provides all the lifecycle methods, so only what's
// required can be implemented here.
class NotifyingLifecycle extends NullLifecycle
{
    /** @var Notifier */
    private $notifier;

    // constructor, etc

    public function failed(Message $message, Consumer $consumer)
    {
        $this->notifier->send(new Notification(sprintf(
            '%s message failed',
            $message->getName()
        )));
    }
}
```

This custom lifecycle can be passed into `Consumer::run` or `Consumer::once`.

```
<?php

/** @var PMG\Queue\Consumer $consumer */
$consumer->run('someQueue', new NotifyingLifecycle(/* ... */));
```

Lifecycles Don't Know About Queue Names

This is on purpose. Because lifecycle objects are passed into consumers at the same time as the queue name, it's up to the implementation to decide if they care about that detail. If the implementation does care, it can take the queue name as a constructor argument.

We've found at PMG that most times queue name is a detail that simply does not matter to the application itself. It's just a way to distribute work.

Provided Message Lifecycles

A `NullLifecycle`, mentioned above, that does nothing. This makes a convenient base class to extend and implement what methods your application requires.

Additionally there are a few other provided `MessageLifecycle` implementations.

`DelegatingLifecycle` proxies to multiple child lifecycles. Use this to compose other lifecycles together. In the example below, both `NotifyingLifecycle` and `SomeOtherLifecycle` would be called for each stage through which the message moves.

```
<?php

use PMG\Queue\Lifecycle\DelegatingLifecycle;

$Lifecycle = new DelegatingLifecycle(
    new NotifyingLifecycle(/* ... */), // see above
    new SomeOtherLifecycle()
);

// Or create from an array
$Lifecycle = DelegatingLifecycle::fromArray([
    new NotifyingLifecycle(/* ... */),
    new SomeOtherLifecycle(),
]);
```

MappingLifecycle proxies to other lifecycles based on the incoming message name. Use this if specific MessageLifecycle implementations need to fire for specific messages. In the example below NotifyingLifecycle would track messageA through its lifecycle and SomeOtherLifecycle would track messageB. Any other message would fallback to FallbackLifecycle.

```
<?php

use PMG\Queue\Lifecycle\MappingLifecycle;

// can use an array or `ArrayAccess` implementation here
$Lifecycle = new MappingLifecycle([
    'messageA' => new NotifyingLifecycle(/* ... */),
    'messageB' => new SomeOtherLifecycle(),
], new FallbackLifecycle());

// or omit the fallback and it will default to `NullLifecycle`
// and do nothing.
$Lifecycle = new MappingLifecycle([
    'messageA' => new NotifyingLifecycle(/* ... */),
    'messageB' => new SomeOtherLifecycle(),
]);
```

These two implementations could be combined as well.

```
<?php

use PMG\Queue\Lifecycle\DelegatingLifecycle;
use PMG\Queue\Lifecycle\MappingLifecycle;

$Lifecycle = new DelegatingLifecycle(
    new FooLifecycle(),
    new MappingLifecycle([
        'messageA' => new DelegatingLifecycle(
            new BarLifecycle(),
            new BazLifecycle()
        ),
    ])
);
```

1.3.4 Build Custom Consumers

Extend `PMG\Queue\AbstractConsumer` to make things easy and implement the `once` method. Here's an example that decorates another Consumer with events.

```
<?php

use PMG\Queue\AbstractConsumer;
use PMG\Queue\Consumer;
use PMG\Queue\Message;
use Symfony\Component\EventDispatcher\Event;
use Symfony\Component\EventDispatcher\EventDispatcherInterface;

final class EventingConsumer extends AbstractConsumer
{
    /** @var Consumer */
    private $wrapped;

    /** @var EventDispatcherInterface $events */

    // constructor that takes a consumer and dispatcher to set the props ^

    public function once($queueName)
    {
        $this->events->dispatch('queue:before_once', new Event());
        $this->wrapped->once($queueName);
        $this->events->disaptch('queue:after_once', new Event());
    }
}
```

1.4 Message Handlers

A message handler is used by `DefaultConsumer` to do the actual work of processing a message. Handlers implement `PMG\Queue\MessageHandler` which accepts a message and a set of options from the the consumer as its arguments.

Every single message goes through a single handler. It's up to that handler to figure out how to deal with each message appropriately.

interface MessageHandler

Namespace `PMG\Queue`

An object that can handle (process or act upon) a single message.

handle (`PMG\Queue\Message $handle`, `array $options=[]`)

Parameters

- **\$handle** – The message to handle.
- **\$options** – A set of options from the consumer.

Returns A boolean indicated whether the message was handled successfully.

Return type boolean

1.4.1 Callable Handler

The simplest handler could just be a callable that invokes the provided callback with the message.

```
<?php

use PMG\Queue\DefaultConsumer;
use PMG\Queue\Message;
use PMG\Queue\Driver\MemoryDriver;
use PMG\Queue\Handler\CallableHandler;

$handler = new CallableHandler(function (Message $msg) {
    switch ($msg->getName()) {
        case 'SendAlert':
            sendAnAlertSomehow($msg);
            break;
        case 'OtherMessage':
            handleOtherMessageSomehow($msg);
            break;
    }
});

$consumer = new DefaultConsumer(new MemoryDriver(), $handler);
```

1.4.2 Multiple Handlers with Mapping Handler

The above *switch* statement is a lot of boilerplaint, so PMG provides a [mapping handler](#) that looks up callables for a message based on its name. For example, here's a callable for the *send alert message*.

```
<?php

final class SendAlertHandler
{
    private $users;
    private $mailer;

    public function __construct(UserRepository $users, \Swift_Mailer $mailer)
    {
        $this->users = $users;
        $this->mailer = $mailer;
    }

    public function __invoke(SendAlert $message)
    {
        $user = $this->users->getByIdentifierOrError($message->getUserId());

        $this->mailer->send(
            \Swift_Message::newInstance()
                ->setTo([$user->getEmail()])
                ->setFrom(['help@example.com'])
                ->setSubject('Hello')
                ->setBody('World')
        );
    }
}
```

Now pull in the mapping handler with `composer require pmg/queue-mapping-handler` and we can integrate the callable above with it.

```
<?php

use PMG\Queue\DefaultConsumer;
use PMG\Queue\Handler\MappingHandler;

$handler = MappingHandler::fromArray([
    'SendAlert' => new SendAlertHandler(/*...*/),
    //'OtherMessage' => new OtherMessageHandler()
    // etc
]);

/** @var PMG\Queue\Driver $driver */
$consumer = new DefaultConsumer($driver, $handler);
```

1.4.3 Using Tactician to Handle Messages

Tactician is a command bus from The PHP League. You can use it to do message handling with the queue.

```
composer install pmg/queue-tactician
```

Use the same command bus with each message.

```
<?php

use League\Tactician\CommandBus;
use PMG\Queue\DefaultConsumer;
use PMG\Queue\Handler\TacticianHandler;

$handler = new TacticianHandler(new CommandBus(/* ... */));

/** @var PMG\Queue\Driver $driver */
$consumer = new DefaultConsumer($driver, $handler);
```

Alternative, you can create a new command bus to handle each message with *CreatingTacticianHandler*. This is useful if you're using *forking child processes* to handle messages.

```
<?php

use League\Tactician\CommandBus;
use PMG\Queue\DefaultConsumer;
use PMG\Queue\Handler\CreatingTacticianHandler;

$handler = new TacticianHandler(function () {
    return new CommandBus(/* ... */);
});

/** @var PMG\Queue\Driver $driver */
$consumer = new DefaultConsumer($driver, $handler);
```

1.4.4 Handling Messages in Separate Processes

To handle messages in a forked process use the *PcntlForkingHandler* decorator.


```
<?php

use PMG\Queue\Handler\MappingHandler;
use PMG\Queue\Handler\PcntlForkingHandler;

// create an actual handler
$realHandler = MappingHandler::fromArray([
    // ...
]);

// decorate it with the forking handler
$handler = new PcntlForkingHandler($realHandler);
```

Forking is useful for memory management, but requires some consideration. For instance, database connections might need to be re-opened in the forked process. In such cases, the best bet is to simply create the resources on demand. that's why the `TaticianHandler` above takes a factory callable by default.

In cases where a process fails to fork, a `PMG\Queue\Exception\CouldNotFork` exception will be thrown and the consumer will exit with an unsuccessful status code. Your process manager (supervisord, upstart, systemd, etc) should be configured to restart the consumer when that happens.

1.5 Drivers & Internals

Behind the scenes *consumers* and *producers* use *driver* and *envelopes* to do their work.

1.5.1 Drivers

Drivers are the queue backend hidden behind the `PMG\Queue\Driver` interface. `pmg/queue` comes with two drivers built in: *memory* and *pheanstalk* (beanstalkd).

Drivers have method for enqueueing and dequeueing messages as well as methods for acknowledging a message is complete, retrying a message, or marking a message as failed.

1.5.2 Envelopes

Envelopes wrap up *messages* to allow drivers to add additional metadata. One example of such metadata is a *retry count* that the *consumers* may use to determine if a message should be retried. The *pheanstalk driver* implements its own envelop class so it can track the beanstalkd job identifier for the message.

Drivers are free to do whatever they need to do as long as their envelope implements `PMG\Queue\Envelope`.

1.5.3 Driver Implementations

The core `pmg/queue` library provides a in memory driver and PMG maintains a *driver for beanstalkd* that uses the *pheanstalk* library.

The Memory Driver & Testing

The memory driver is provided to make prototyping and testing easy. It uses `SplQueue` instances and only keeps messages in memory.

```
<?php
use PMG\Queue\DefaultConsumer;
use PMG\Queue\Driver\MemoryDriver;

// ...

$driver = new MemoryDriver();

// $handler instanceof PMG\Queue\MessageHandler
$consumer = new DefaultConsumer($driver, $handler);
```

The memory driver is not very useful outside of testing. For instance, while doing end to end tests, you may want to switch out your producers library to use the memory driver then verify the expected messages when into it.

```
<?php
use PMG\Queue\Driver\MemoryDriver;

class SomeTest extends \PHPUnit_Framework_TestCase
{
    const TESTQ = 'TestQueue';

    /** @var MemoryDriver $driver */
    private $driver;

    public function testSomething()
    {
        // imagine some stuff happened before this, now we need to verify that

        $envelope = $this->driver->dequeue(self::TESTQ);

        $this->assertNotNull($envelope);
        $msg = $envelope->unwrap();
        $this->assertInstanceOf(SendAlert::class, $msg);
        $this->assertEquals(123, $msg->getUserId());
    }
}
```

Pheanstalk Driver

The pheanstalk driver is backed by [beanstalkd](#) and is a *persistent* driver: messages persist across multiple requests or queue runs.

To use it, use composer to install `pmg/queue-pheanstalk` and pass an instance of `Pheanstalk\Pheanstalk` and a *serializer* to its constructor.

```
<?php
use Pheanstalk\Pheanstalk;
use PMG\Queue\Driver\PheanstalkDriver;
use PMG\Queue\Driver\Serializer\NativeSerializer;

$driver = new PheanstalkDriver(
    new Pheanstalk('localhost', 11300),
    NativeSerializer::fromSigningKey('this is a key used to sign messages')
);
```

See the [pheanstalk driver repository](#) for more information and examples.

1.5.4 Serializers

Persistent drivers require some translation from *envelopes* and *messages* to something the persistent backend can store. Similarly, whatever is stored in the queue backend needs to be turned back into a message. **Serializers** make that happen.

All serializers implements `PMG\Queue\Serializer\Serializer` and one implementation is provided by default: `NativeSerializer`.

`NativeSerializer` uses PHP's built in `serialize` and `unserialize` functions. Serialized envelopes are base64 encoded and signed (via a `Signer`). The signature is a way to authenticate the message: make sure it came from a known source and hasn't been tampered with

```
<?php
use PMG\Queue\Signer\HmacSha256;
use PMG\Queue\Serializer\NativeSerializer;

$serializer = new NativeSerializer(new HmacSha256('super secret key'));
// identical to...
$serializer = NativeSerializer::fromSigningKey('super secret key');

// ...
```

Should want to use `ext-libsodium` or the built in `libsodium` support in PHP 7.2+ there is also a `SodiumCryptoAuth` signer.

```
<?php
use PMG\Queue\Signer\SodiumCryptoAuth;
use PMG\Queue\Serializer\NativeSerializer;

$serializer = new NativeSerializer(new SodiumCryptoAuth(
    'aKeyThatIsExactly32characterLong' // or sodium complains
));

// ...
```

Allowed Classes in PHP 7

`NativeSerializer` supports PHP 7's `allowed_classes` option in `unserialize` to whitelist classes. Just pass an array of message class names as the second argument to `NativeSerializer`'s constructor.

Because drivers have their own envelope classes, the *pheanstalk driver* (or any other drivers that extend `PMG\Queue\Driver\AbstractPersistenceDriver`) provides a static `allowedClasses` method that returns an array of envelope classes to whitelist.

```
<?php
use PMG\Queue\Serializer\NativeSerializer;
use PMG\Queue\Driver\PheanstalkDriver;

$serializer = new NativeSerializer('YourSecretKeyHere', array_merge([
    // your message classes
    SendAlert::class,
    // ...
], PheanstalkDriver::allowedClasses()));
```

1.5.5 Implementing Your Own Drivers

Persistent drivers are not required to use serializers (or anything else), but if they do `PMG\Queue\Driver\AbstractPersistenceDriver` provides helpers for the usage of serializers.

CHAPTER 2

Installation & Examples

You should require the driver library of your choice with [composer](#) rather than `pmg/queue` directly. If you're planning to use `beanstalkd` as your backend:

```
composer require pmg/queue-pheanstalk:~1.0
```

See the core [examples directory](#) on the [pheanstalk examples](#) for some code samples on gluing everything together.

READ THIS: Glossary & Core Concepts

- A **message** is a serializable object that goes into the queue for later processing.
- A **producer** adds messages to the queue backend via a *driver* and a *router*.
- A **consumer** pulls messages out of the queue via *driver* and processes them via *handlers*.
- A **driver** is PHP representation of the queue backend. There is an in memory driver included in this library as an example (and for testing), and an implementation of a [beanstalkd](#) driver [available](#).
- A **router** is used by a producer to look up the correct queue for a message.
- A **message handler** is used by the default *consumer* to actually do the work of processing a message
- An **envelope** is used internally to wrap up messages with retry information as well as metadata specific to drivers. Users need not worry about this unless they are implementing their own *driver*.

C

`canRetry()` (RetrySpec method), [6](#)
`Consumer` (interface), [5](#)

H

`handle()` (MessageHandler method), [10](#)

M

`MessageHandler` (interface), [10](#)

O

`once()` (Consumer method), [6](#)

P

`Producer` (interface), [4](#)

Q

`queueFor()` (Router method), [4](#)

R

`RetrySpec` (interface), [6](#)
`Router` (interface), [4](#)
`run()` (Consumer method), [5](#)

S

`send()` (Producer method), [4](#)
`stop()` (Consumer method), [6](#)