Bernard

Release latest

Jul 06, 2018

Contents

CHAPTER 1

Installation

The recommended way to install Bernard is using Composer. If your projects do not already use this, it is highly recommended to start using it.

To install Bernard, run:

```
$ composer require bernard/bernard
```

Then look at what kind of drivers and serializers are available and install the ones you need before you are going to use Bernard.

CHAPTER 2

Examples

There are numerous examples of running Bernard in the `example` directory. The files are named after the driver they are using. Each file takes the argument `consume` or `produce`. For instance, to use the `Predis` driver, use:

```
$ php ./example/predis.php consume
$ php ./example/predis.php produce
```

And you would see properly a lot of output showing an error. This is because the `ErrorLogMiddleware` is registered and shows all exceptions. In this case, the exception is caused by `rand()` always returning 7.

This directory is a good source for setting stuff up and can be used as a go to guide.

2.1 Producing messages

Any message sent to Bernard must be an instance of `Bernard\Message`, which has a `getName`. `getName` is used when working on messages and identifies the worker service that should work on it.

A message is given to a producer that sends the message to the right queue. It is also possible to get the queue directly from the queue factory and push the message there. But remember to wrap the message in an `Envelope` object. The easiest way is to give it to the producer, as the queue name is taken from the message object.

To make it easier to send messages and not require every type to be implemented in a separate class, a `Bernard\Message\PlainMessage` is provided. It can hold any number of properties and only needs a name for the message. The queue name is then generated from that. When generating the queue name it will insert a "_" before any uppercase letter and then lowercase the name.

```
<?php

use Bernard\Message\PlainMessage;
use Bernard\Producer;
use Bernard\QueueFactory\PersistentFactory;
use Bernard\Serializer;

```

(continues on next page)

(continued from previous page)

```php
//.. create $driver
$factory = new PersistentFactory($driver, new Serializer());
$producer = new Producer($factory);

$message = new PlainMessage('SendNewsletter', array(
    'newsletterId' => 12,
));

$producer->produce($message);
```

2.2 Queues

Bernard comes with a few built-in queues

2.2.1 Persistent queue

The default queue to use, it produces message to and consumes messages from a driver's queue

2.2.2 Roundrobin queue

With the roundrobin queue you can produce messages to multiple queues

2.2.3 In Memory Queue

Bernard comes with an implementation for `SplQueue` which is completely in memory. It is useful for development and/or testing, when you don't necessarily want actions to be performed.

2.3 Drivers

Several different types of drivers are supported. Currently these are available:

- *Google AppEngine*
- *Doctrine DBAL*
- *Flatfile*
- *IronMQ*
- *MongoDB*
- *Pheanstalk*
- *PhpAmqp / RabbitMQ*
- *Redis Extension*
- *Predis*
- *Amazon SQS*
- Queue Interop

2.3.1 Google AppEngine

The Google AppEngine has support for PHP and PushQueue just as IronMQ. The AppEngine driver for Bernard is a minimal driver that uses its TaskQueue to push messages. Visit the official docs to get more information on the usage of the AppEngine api.

Important: This driver only works on AppEngine or with its development server as it needs access to its SDK. It must also be autoloadable. If it is in the include path you can use `"config" : { "use-include-path" : true } }` in Composer.

The driver takes a list of queue names and mappings to an endpoint. This is because queues are created at runtime and their endpoints are not preconfigured.

```php
<?php

use Bernard\Driver\AppEngine\Driver;

$driver = new Driver(array(
    'queue-name' => '/url_endpoint',
));
```

To consume messages, you need to create an url endpoint matching the one given to the drivers constructor. For the actual dispatching of messages, you can do something like this:

```php
<?php

namespace Acme\Controller;

use Bernard\Consumer
use Bernard\Serializer;
use Bernard\QueueFactory;
use Symfony\Component\HttpFoundation\Request;

class QueueController
{
    protected $consumer;
    protected $queues;
    protected $serializer;

    public function __construct(Consumer $consumer, QueueFactory $queues, Serializer
↪$serializer)
    {
        $this->consumer = $consumer;
        $this->queues = $queues;
        $this->serializer = $serializer;
    }

    public function queueAction(Request $request)
    {
        $envelope = $this->serializer->deserialize($request->getContent());

        // This will invoke the right service and middleware, and lastly it will␣
↪acknowledge
        // the message.
        $this->consumer->invoke($envelope, $this->queues->create($envelope->
↪getMessage()->getQueue()));
```

(continues on next page)

(continued from previous page)

```
    }
}
```

2.3.2 Doctrine DBAL

For small usecases or testing, there is a Doctrine DBAL driver which supports all of the major database platforms.

The driver uses transactions to make sure that a single consumer always get the message popped from the queue.

Important: To use Doctrine DBAL remember to setup the correct schema.

Creating the needed bernard tables can be automated by creating a console application with custom commands. Just configure a connection or entity manager as appropriate for your use case.

```php
<?php
// doctrine.php

use Bernard\Driver\Doctrine\Command as BernardCommands;
use Doctrine\DBAL\Tools\Console\ConsoleRunner;
use Doctrine\DBAL\Tools\Console\Helper\ConnectionHelper;
use Symfony\Component\Console\Application;
use Symfony\Component\Console\Helper\HelperSet;

$connection = ...;
$commands = [
    new BernardCommands\CreateCommand(),
    new BernardCommands\DropCommand(),
    new BernardCommands\UpdateCommand(),
];

// To create a new application from scratch ...
$helperSet = new HelperSet(['connection' => new ConnectionHelper($connection)]);
$cli = new Application('Bernard Doctrine Command Line Interface');
$cli->setCatchExceptions(true);
$cli->setHelperSet($helperSet);
$cli->addCommands($commands);

// ... or, if you're using Doctrine ORM 2.5+,
// just re-use the existing Doctrine application ...
$entityManager = ...;
$helperSet = ConsoleRunner::createHelperSet($entityManager);
$cli = ConsoleRunner::createApplication($helperSet, $commands);

// Finally, run the application
$cli->run();
```

And run the console application like so:

```
php doctrine.php bernard:doctrine:create
```

Alternatively, use the following method for creating the tables manually.

```php
<?php
```

(continues on next page)

(continued from previous page)

```
use Bernard\Driver\Doctrine\MessagesSchema;
use Doctrine\DBAL\Schema\Schema;

MessagesSchema::create($schema = new Schema);

// setup Doctrine DBAL
$connection = ...;

$sql = $schema->toSql($connection->getDatabasePlatform());

foreach ($sql as $query) {
    $connection->exec($query);
}
```

And here is the setup of the driver for doctrine dbal:

```
{
    "require" : {
        "doctrine/dbal" : "~2.3"
    }
}
```

```
<?php

use Bernard\Driver\Doctrine\Driver;
use Doctrine\DBAL\DriverManager;

$connection = DriverManager::getConnection(array(
    'dbname'   => 'bernard',
    'user'     => 'root',
    'password' => null,
    'driver'   => 'pdo_mysql',
));


$driver = new Driver($connection);
```

2.3.3 Flatfile

The flat file driver provides a simple job queue without any database

```
<?php

use Bernard\Driver\FlatFile\Driver;

$driver = new Driver('/dir/to/store/messages');
```

2.3.4 IronMQ

IronMQ from Iron.io is a “message queue in the cloud”. The IronMQ driver supports prefetching messages, which reduces the number of HTTP request. This is configured as the second parameter in the drivers constructor.

Important: You need to create an account with iron.io to get a `project-id` and `token`.

Important: When using prefetching the timeout value for each message much be greater than the time it takes to consume all of the fetched message. If one message takes 10 seconds to consume and the driver is prefetching 5 message the timeout value must be greater than 10 seconds.

```
{
    "require" : {
        "iron-io/iron_mq" : "~1.4"
    }
}
```

```
<?php

use Bernard\Driver\IronMQ\Driver;

$connection = new IronMQ(array(
    'token'      => 'your-ironmq-token',
    'project_id' => 'your-ironmq-project-id',
));


$driver = new Driver($connection);

// or with a prefetching number
$driver = new Driver($connection, 5);
```

It is also possible to use push queues with some additional logic. Basically, it is needed to deserialize the message in the request and route it to the correct service. An example of this:

```
<?php

namespace Acme\Controller;

use Bernard\Consumer
use Bernard\Serializer;
use Bernard\QueueFactory;
use Symfony\Component\HttpFoundation\Request;

class QueueController
{
    protected $consumer;
    protected $queues;
    protected $serializer;

    public function __construct(Consumer $consumer, QueueFactory $queues, Serializer
↪$serializer)
    {
        $this->consumer = $consumer;
        $this->queues = $queues;
        $this->serializer = $serializer;
    }
```

(continues on next page)

(continued from previous page)

```
    public function queueAction(Request $request)
    {
        $envelope = $this->serializer->deserialize($request->getContent());

        // This will invoke the right service and middleware, and lastly it will␣
↪acknowledge
        // the message.
        $this->consumer->invoke($envelope, $this->queues->create($envelope->
↪getMessage()->getQueue()));
    }
}
```

2.3.5 MongoDB

The MongoDB driver requires the mongo PECL extension. On platforms where the PECL extension is unavailable, such as HHVM, mongofill may be used instead.

The driver should be constructed with two MongoCollection objects, which corresponding to the queue and message collections, respectively.

```
<?php

$mongoClient = new \MongoClient();
$driver = new \Bernard\Driver\MongoDB\Driver(
    $mongoClient->selectCollection('bernardDatabase', 'queues'),
    $mongoClient->selectCollection('bernardDatabase', 'messages'),
);
```

Note: If you are using Doctrine MongoDB or the ODM, you can access the MongoCollection objects through the `getMongoCollection()` method on the `Doctrine\MongoDB\Collection` wrapper class, which in turn may be retrieved from a `Doctrine\MongoDB\Database` wrapper or DocumentManager directly.

To support message queries, the following index should also be created:

```
<?php

$mongoClient = new \MongoClient();
$collection = $mongoClient->selectCollection('bernardDatabase', 'messages');
$collection->createIndex([
    'queue' => 1,
    'visible' => 1,
    'sentAt' => 1,
]);
```

2.3.6 Pheanstalk

Requires the installation of pda/pheanstalk. Add the following to your `composer.json` file for this:

```
{
    "require" : {
        "pda/pheanstalk" : "~3.0"
```

(continues on next page)

(continued from previous page)

```
    }
}
```

```
<?php

use Bernard\Driver\Pheanstalk\Driver;
use Pheanstalk\Pheanstalk;

$pheanstalk = new Pheanstalk('localhost');

$driver = new Driver($pheanstalk);
```

2.3.7 PhpAmqp / RabbitMQ

The RabbitMQ driver uses the php-amqp library by php-amqplib.

The driver should be constructed with a class that extends *AbstractConnection* (for example *AMQPStreamConnection* or *AMQPSocketConnection*), an exchange name and optionally the default message parameters.

```
<?php

$connection = new \PhpAmqpLib\Connection\AMQPStreamConnection('localhost', 5672, 'foo
↪', 'bar');

$driver = new \Bernard\Driver\PhpAmqpDriver($connection, 'my-exchange');

// Or with default message params
$driver = new \Bernard\Driver\PhpAmqpDriver(
    $connection,
    'my-exchange',
    ['content_type' => 'application/json', 'delivery_mode' => 2]
);
```

2.3.8 Redis Extension

Requires the installation of the pecl extension. You can add the following to your `composer.json` file, to make sure it is installed:

```
{
    "require" : {
        "ext-redis" : "~2.2"
    }
}
```

```
<?php

use Bernard\Driver\PhpRedis\Driver;

$redis = new Redis();
$redis->connect('127.0.0.1', 6379);
$redis->setOption(Redis::OPT_PREFIX, 'bernard:');

$driver = new Driver($redis);
```

2.3.9 Predis

Requires the installation of predis. Add the following to your `composer.json` file for this:

```
{
    "require" : {
        "predis/predis" : "~0.8"
    }
}
```

```
<?php

use Bernard\Driver\Predis\Driver;
use Predis\Client;

$predis = new Client('tcp://localhost', array(
    'prefix' => 'bernard:',
));

$driver = new Driver($predis);
```

2.3.10 Amazon SQS

This driver implements the SQS (Simple Queuing System) part of Amazons Web Services (AWS). The SQS driver supports prefetching messages which reduces the number of HTTP request. It also supports aliasing specific queue urls to a queue name. If queue aliasing is used the queue names provided will not require a HTTP request to amazon to be resolved.

Important: You need to create an account with AWS to get SQS access credentials, consisting of an API key and an API secret. In addition, each SQS queue is setup in a specific region, eg `eu-west-1` or `us-east-1`.

Important: When using prefetching, the timeout value for each message should be greater than the time it takes to consume all of the fetched message. If one message takes 10 seconds to consume and the driver is prefetching 5 message the timeout value must be greater than 10 seconds.

```
{
    "require" : {
        "aws/aws-sdk-php" : "~2.4"
    }
}
```

```
<?php

use Aws\Sqs\SqsClient;
use Bernard\Driver\Sqs\Driver;

$connection = SqsClient::factory(array(
    'key'    => 'your-aws-access-key',
    'secret' => 'your-aws-secret-key',
    'region' => 'the-aws-region-you-choose'
));
```

(continues on next page)

(continued from previous page)

```
$driver = new Driver($connection);

// or with prefetching
$driver = new Driver($connection, array(), 5);

// or with aliased queue urls
$driver = new Driver($connection, array(
    'queue-name' => 'queue-url',
));
```

2.3.11 Queue Interop

This driver adds ability to use any queue interop compatible transport. For example we choose enqueue/fs one to demonstrate how it is working.

```
{
    "require" : {
        "enqueue/fs" : "^0.7"
    }
}
```

```
<?php

use Bernard\Driver\Interop\Driver;
use Enqueue\Fs\FsConnectionFactory;

$context = (new FsConnectionFactory('file://'.__DIR__.'/queues'))->createContext();

$driver = new InteropDriver($context);
```

2.4 Serializers

Bernard uses the Symfony Serializer Component to serialize messages as JSON for persistent storage.

2.4.1 Default serializer

By default Bernard can handle serializing the `Bernard\Envelope` and `Bernard\Message\PlainMessage` classes, which should be enough when you are just starting out:

```
<?php

use Bernard\Serializer;

$serializer = new Serializer();
$json = $serializer->serialize($envelope);
```

2.4.2 Adding normalizers

If you are using your own custom message classes, you **must** provide a normalizer for them. This example assumes your message contains getters and setters for the properties it needs serializing:

```
<?php

use Bernard\Normalizer\PlainMessageNormalizer;
use Bernard\Normalizer\EnvelopeNormalizer;
use Bernard\Serializer;
use Normalt\Normalizer\AggregateNormalizer;
use Symfony\Component\Serializer\Normalizer\GetSetMethodNormalizer;

$aggregateNormalizer = new AggregateNormalizer([
    new EnvelopeNormalizer(),
    new GetSetMethodNormalizer(),
    new PlainMessageNormalizer(),
]);

$serializer = new Serializer($aggregateNormalizer);
$envelope = $serializer->deserialize($json);
```

The `AggregateNormalizer` will check each normalizer passed to its constructor and use the first one that can handle the object given to it. You should always pass the `EnvelopeNormalizer` first. And it's a good idea to add the `PlainMessageNormalizer` last as a fallback when none other match.

More normalizers are available from Symfony, along with the `DoctrineNormalizer` and `RecursiveReflectionNormalizer` from Normalt.

2.5 Consuming Messages

Consuming messages has two requirements:

- the system needs to know how messages should be handled
- the system needs to provide extension points for certain events

The first requirement is fulfilled by message routing, the second is by the event dispatcher system.

```
<?php

use Symfony\Component\EventDispatcher\EventDispatcher;

// $router = see bellow
$eventDispatcher = new EventDispatcher();

// Create a Consumer and start the loop.
$consumer = new Consumer($router, $eventDispatcher);

// The second argument is optional and is an array
// of options. Currently only ``max-runtime`` is supported which specifies the max␣
↪runtime
// in seconds.
$consumer->consume($queueFactory->create('send-newsletter'), array(
    'max-runtime' => 900,
));
```

2.5.1 Routing

A single message represents a job that needs to be performed, and as described earlier, by default a message's name is used to determine which receiver should receive that message.

A receiver can be any of the following:

- callable
- class with a static method with the name of the message with the first letter lower cased
- object with a method with the name of the message with the first letter lower cased
- object implementing the `Bernard\Receiver` interface

For the system to know which receiver should handle which messages, you are required to register them first.

```
<?php

use Bernard\Router\ReceiverMapRouter;
use Bernard\Consumer;

// create driver and a queuefactory
// NewsletterMessageHandler is a pseudo receiver that has a sendNewsletter method.

$router = new ReceiverMapRouter([
    'SendNewsletter' => new NewsletterMessageHandler(),
]);
```

Message routing can also happen based on the message class instead of the message name.

```
<?php

use Bernard\Router\ClassNameRouter;
use Bernard\Consumer;

// create driver and a queuefactory
// NewsletterMessageHandler is a pseudo receiver that has a sendNewsletter method.
// NewsletterMessage is a pseudo message

$router = new ClassNameRouter([
    NewsletterMessage::class => new NewsletterMessageHandler(),
]);
```

In some cases the above described receiver rules might not be enough. The provided router implementations also accept a receiver resolver which can be used for example to resolve receivers from a Dependency Injection container. A good example for that is the PSR-11 container resolver implementation that comes with this package.

```
<?php

use Bernard\Router\ReceiverMapRouter;
use Bernard\Router\ContainerReceiverResolver;
use Bernard\Consumer;

// create driver and a queuefactory
// NewsletterMessageHandler is a pseudo receiver that has a sendNewsletter method.
// $container = your PSR-11 compatible container

$router = new ReceiverMapRouter(
    [
```

(continues on next page)

(continued from previous page)

```
        'SendNewsletter' => NewsletterMessageHandler::class,
    ],
    new ContainerReceiverResolver($container),
);
```

2.5.2 Commandline Interface

Bernard comes with a `ConsumeCommand` which can be used with Symfony Console component.

```
<?php

use Bernard\Command\ConsumeCommand;

// create $console application
$console->add(new ConsumeCommand($consumer, $queueFactory));
```

It can then be used as any other console command. The argument given should be the queue that your messages are on. If we use the earlier example with sending a newsletter, it would look like this.

```
$ /path/to/console bernard:consume send-newsletter
```

2.5.3 Internals

When a message is dequeued it is also marked as invisible (if the driver supports this) and when the message have been consumed then it will also be acknowledged. Some drivers have a timeout on the invisible state and will automatically requeue a message after that time. Therefore it is important to have a timeout greater than it takes for you to consume a single message.

2.6 Framework Integration

To make it easier to get started and have it "just work" with sending messages, a number of integrations have been created.

2.6.1 Symfony

The bernard/bernard-bundle integrates Bernard with a Symfony application.

2.6.2 Silex

There is a bernard/silex package which enables usage of Bernard in your Silex applications.

2.6.3 Laravel

The officially supported Laravel package can be found at bernard/laravel.

2.7 Cookbook

2.7.1 Monitoring

Having a message queue where it is not possible to know what is in the queue and the contents of the messages is not very handy, so for that there is Juno.

It is implemented in Silex and is very lightweight. Also if needed, it can be embedded in other Silex or Flint applications.