Recruiter

Release 4.0.1

Jul 18, 2019

Contents

Recruiter is a Job Queue Manager built with PHP meant to be used in PHP projects.
It allow php developers to perform some operations in background.

Features and characteristics:

- *Jobs* are made persistent on MongoDB
- *Jobs* with complex and customizable strategies
- *Jobs* are stored by default in an history collection for after the fact inspection and analytics
- *Multiple queues are supported through grouping*
- Built to be robust, scalable and fault tolerant

At high level, it provides a few major components:

- A *recruiter*: a single instance long-running process who assign enqueued jobs to a worker
- A *worker*: a multiple instace long-running processes that each execute a single job at a time
- A *cleaner*: a single instance long-running process who takes care of cleaning up the dirty conditions that can happen (i.e. worker dead for a fatal error)

CHAPTER 1

What is Recruiter

Recruiter is a Job Queue Manager built with PHP meant to be used in PHP projects.
It allow php developers to perform some operations in background.

Features and characteristics:

- *Jobs* are made persistent on MongoDB
- *Jobs* with complex and customizable strategies
- *Jobs* are stored by default in an history collection for after the fact inspection and analytics
- *Multiple queues are supported through grouping*
- Built to be robust, scalable and fault tolerant

At high level, it provides a few major components:

- A *recruiter*: a single instance long-running process who assign enqueued jobs to a worker
- A *worker*: a multiple instace long-running processes that each execute a single job at a time
- A *cleaner*: a single instance long-running process who takes care of cleaning up the dirty conditions that can happen (i.e. worker dead for a fatal error)

1.1 Why

1.1.1 DA COMPLETARE

Onebip is a payment system (think PayPal with mobile devices in place of credit cards), things like: payment notifications, subscription renewals, remainder messages, ... are **really** important. You cannot skip or lose a job (notification are idempotent but payments are not). You cannot forgot to have completed a job (customer/merchant support must have data to do their job). You need to know if and when you can retry a failed job (external services have rate limits and are based on agreements/contracts). We have developed internally our job/queue solution called **Recruiter**. After

a year in production and many *billions* of jobs we have decided to put what we have learned into a stand alone project and to make it available to everyone.

CHAPTER 2

Processes

2.1 Recruiter process

Il processo *recruiter* é colui che si occupa di assegnare, al momento giusto, i job presenti in coda ai vari *worker* che sono in esecuzione.

E' importante che una sola instanza del processo *recruiter* sia live in un determinato momento, ma lo sviluppatore non deve preoccuparsene in quanto il *recruiter* include al suo interno un meccanismo di muta esclusione, é quindi possibile eseguire più processi in contemporanea (ad esempio se si hanno più server identici, ognuno dei quali lancia la propria istanza di *recruiter*) senza che ci siano problemi di concorrenza.
Per maggiori informazioni riguardo a questa funzionalità guardare il capitolo relativo a Geezer

Il processo *recruiter* per funzionare ha bisogno di collegarsi ad un istanza di mongodb, é possibile specificare l'URI tramite l'opzione **–target ${MONGOURI}** (se non specificata il processo *recruiter* proverà a collegarsi a *localhost:27017*)

Se si vuole approfittare degli *hook* messi a disposizione dal processo *recruiter* é indispensabile passare al comando uno script php da includere, in modo tale che le funzioni definite dall'utente siano visibili. Questo può essere fatto tramite l'opzione **–bootstrap**

Per lanciare il processo *recruiter* utilizzare il seguente comando:

```
$ php vendor/bin/recruiter start:recruiter --target 127.0.0.1:27017
```

Per una lista completa delle opzioni lanciare il comando:

```
$ php vendor/bin/recruiter help start:recruiter
```

2.2 Worker process

Il processo *worker* é colui che si occupa effettivamente di eseguire un determinato job.
Al proprio avvio il processo *worker* comunica al processo *recruiter* il fatto di essere disponibile ad accettare lavori.
E' possibile eseguire più processi *worker* in contemporanea, ognuno di questi eseguirà un singolo job alla volta.
E' possibile limitare un *worker* all'esecuzione di un solo specifico gruppo di lavori, questo é un metodo per poter gesitre in maniera blanda le *priorità*.

Il processo *worker* per funzionare ha bisogno di collegarsi ad un istanza di mongodb, é possibile specificare l'URI tramite l'opzione **–target ${MONGOURI}** (se non specificata il processo *worker* proverà a collegarsi a *localhost:27017*)

Ad eccezione di alcuni rari casi, il processo *worker* dovrà eseguire codice facente parte del progetto in cui viene incluso, é quindi indispensabile passare al comando uno script php da includere, in modo tale che le classi definite dall'utente siano visibili. Questo può essere fatto tramite l'opzione **–bootrap**

Per lanciare il processo *worker* utilizzare il seguente comando:

```
$ php vendor/bin/recruiter start:worker --target 127.0.0.1:27017 --bootstrap $APP_
↪BASE_PATH/worker-boostrap.php
```

Per una lista completa delle opzioni lanciare il comando:

```
$ php vendor/bin/recruiter help start:worker
```

2.3 Cleaner process

Il processo *cleaner* si occupa di mantenere coerente lo stato della libreria.
Ad esempio un determinato *worker* potrebbe morire in maniera fatale durante l'esecuzione di un job, lasciando il job lockato e quindi non più eseguibile da altri.
Grazie al processo *cleaner* i job possono essere rimessi nella coda di esecuzione dopo un determinato periodo di stallo.

Il processo *cleaner* per funzionare ha bisogno di collegarsi ad un istanza di mongodb, é possibile specificare l'URI tramite l'opzione **–target ${MONGOURI}** (se non specificata il processo *cleaner* proverà a collegarsi a *localhost:27017*)

Per lanciare il processo *cleaner* utilizzare il seguente comando:

```
$ php vendor/bin/recruiter start:cleaner --target 127.0.0.1:27017
```

Per una lista completa delle opzioni lanciare il comando:

```
$ php vendor/bin/recruiter help start:cleaner
```

2.4 Logging

Come abbiamo visto nei paragrafi precedenti, é possibile lanciare i vari processi (*recruiter*, *worker* e *cleaner*) grazie allo script php `vendor/bin/recruiter`.
Lo script php `vendor/bin/recruiter` non fa altro che creare una istanza di `Symfony\Component\Console\Application`, registrare i vari `Symfony\Component\Console\Command\Command` (Recruiter, Worker e Clenaer Commands) ed eseguire l'applicazione symfony.
Lo script crea i comandi Recruiter, Worker e Cleaner iniettandogli un istanza di `Psr\Log\LoggerInterface` che logga su standard output. Nel caso in cui si desiderasse una diversa tipologia di `Psr\Log\LoggerInterface` bisogna includere questi comandi nella propria `Symfony\Component\Console\Application` in modo tale da poterli inizializzare iniettandogli il logger che si vuole.

```php
<?php
// bin/my-command

use Recruiter\Geezer\Command\RobustCommandRunner;
use Recruiter\Factory;
use Recruiter\Infrastructure\Command\CleanerCommand;
use Recruiter\Infrastructure\Command\RecruiterCommand;
use Recruiter\Infrastructure\Command\WorkerCommand;
use Symfony\Component\Console\Application;
use Domain\MyLogger;

$logger = new MyLogger();

$application = new Application();

$application->add(RecruiterCommand::toRobustCommand(new Factory(), $logger));
$application->add(WorkerCommand::toRobustCommand(new Factory(), $logger));
$application->add(CleanerCommand::toRobustCommand(new Factory(), $logger));

$application->run();
```

CHAPTER 3

Setup

3.1 Dependencies

You need Php version > 7.2
You need a running Mongodb instance

3.2 Install

You can install Recruiter through Composer by running the following command in your terminal:

```
$ composer require recruiterphp/recruiter
```

3.3 Setup

Dovrai creare un file di bootstrap per i processi worker, in modo tale da includere le tue classi cossiché possano essere utilizzate dal worker.

Se ad esempio utilizzi l'autoloading di *composer* per il tuo progetto, puoi scrivere un file di bootstrap semplice come questo:

```
<?php
# src/recruiter-autoload.php

require_once __DIR__ . '/../vendor/autoload.php';
```

(continues on next page)

(continued from previous page)

```
// in the bootstrap file you have access to a Recruiter\Recruiter instance through␣
↪global variable `$recruiter`.
// $recruiter;
```

Dopodiché dovrai lanciare i processi *recruiter*, *worker* e *cleaner*

```
$ php vendor/bin/recruiter start:recruiter --target 127.0.0.1:27017 --bootstrap src/
↪recruiter-bootstrap.php
$ php vendor/bin/recruiter start:worker --target 127.0.0.1:27017 --bootstrap src/
↪recruiter-bootstrap.php
$ php vendor/bin/recruiter start:cleaner --target 127.0.0.1:27017
```

3.4 Sample

Here is an empty sample project using Recruiter.

CHAPTER 4

Cos'é un oggetto Workable?

Il workable é la classe che contiene la procedura che dovrà essere svolta in maniera asincrona.
Questa verrà poi encapsulata in un *Job* del recruiter per poter essere messo in una coda ed essere successivamente eseguito da un *Worker*.

4.1 Implementare un proprio *Workable*

Vediamo con un esempio come poter implementare un proprio *Workable*.
Supponiamo di avere una procedura dove ad un certo punto avremo bisogno di inviare una richiesta http.
Vogliamo che questa richiesta venga effettuata in maniera asincrona tramite *Recruiter*.

Avremo quindi bisogno di creare una classe che si occupi dell'invio della richiesta http e che possa poi essere encapsulata in un *Job* del recruiter.
Supponiamo di avere già nel nostro dominio una classe *Request* che si occupa di rappresentare la request da inviare, ed una classe *HttpClient* che si occupa dell'effetivo invio delle richieste.
Iniziamo:

```
<?php

use Http\Request;

class HttpRequestCommand
{
    /**
     * @var Request
     */
    private $request;
```

(continues on next page)

(continued from previous page)

```
    public function __construct(Request $request)
    {
        $this->request = $request;
    }

    public static function fromRequest(Request $request)
    {
        return new self ($request);
    }

    public function execute()
    {
        $httpClient = Container::instance()->get('Http\Client');
        $httpClient->send($this->request);
    }
}
```

Note:

Possiamo nominare a nostro piacimento il metodo che verrà poi richiamato dai *Worker*, in questo caso é stato scelto *execute()* ma nulla vieta di utilizzare un nome diverso.
Vedremo più avanti come istruire i *Worker* a richiamare il metodo che vogliamo.

Bene, ora che abbiamo la nostra classe che é in grado di inviare una *Request* vediamo come fare per poterla utilizzare tramite *Recruiter* e poter quindi eseguirla in maniera *asincrona*.
Per prima cosa dovremo fra si che la nostra *HttpRequestCommand* implementi l'interfaccia `Recruiter\Workable`.
Questa interfaccia si compone di 3 metodi, utili a trasformare il nostro *Workable* in un *Job* del recruiter e a poterlo importare ed esportare per il salvataggio a database e successivo ripristino.

```
<?php

use Recruiter\Recruiter;
use Recruiter\Workable;
use Http\Request;

class HttpRequestCommand implements Workable
{
    /**
     * @var Request
     */
    private $request;

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

    public static function fromRequest(Request $request)
    {
        return new self ($request);
```

(continues on next page)

(continued from previous page)

```
    }

    public function execute()
    {
        $httpClient = Container::instance()->get('Http\Client');
        $httpClient->send($this->request);
    }

    public function asJobOf(Recruiter $recruiter)
    {
        return $recruiter->jobOf($this);
    }

    public function export()
    {
        return ['request' => $this->request];
    }

    public static function import($parameters)
    {
        return new self(Request::box($parameters['request']));
    }
}
```

Ora il recruiter potrà creare un *Job* dedicato all'esecuzione di questa procedura, esportare i dati necessari all'esecuzione della procedura per poterli salvare su database e successivamente ricreare l'istanza del nostro *Workable* quando dovrà essere eseguito.

Warning:

Ricorda che l'istanza della tua classe *Workable* verrà storicizzata su Mongo, assicurati quindi che il metodo **export()** della tua classe ritorni un contenuto serializzabile.
In questo esempio diamo per scontato che la classe `Http\Client` non sia serializzabile, per questo motivo non é inclusa nell'export e viene ricavata tramite l'utilizzo di un "ServiceLocator".

Vediamo ora come utilizzarlo.

```
<?php

use Recruiter\Recruiter;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$request = Request::post($url, $body);
HttpRequestCommand::fromRequest($request)
    ->asJobOf($recruiter)
    ->inBackground()
```

(continues on next page)

(continued from previous page)

```
    ->execute() // this is the method defined in the Workable class
;
```

Ora la nostra *Request* é in coda, pronta per essere inviata non appena un *Worker* sarà disponibile.
Analizzando il codice possiamo notare che:
- abbiamo instanziato il nostro *Workable* **HttpRequestCommand** passandoglia una *Request*.
- abbiamo incapsulato il nostro *Workable* in un *Job*.
- abbiamo settato il *Job* per l'esecuzione in background.
- abbiamo istruito il *Worker* a chiamare il metodo **'execute()'** sull'istanza *Workable* contenuta nel *Job*.

Nel *prossimo capitolo* scopriremo tutte le opzioni disponibili per i vari *Job*.

CHAPTER 5

Jobs

5.1 Cos'é un Job

Possiamo definire i `Job` come le unità di lavoro eseguite dal recruiter.
Essi si occupano di incapsulare l'oggetto `Workable` che abbiamo visto in precedenza (e quindi la procedura da esegure) oltre a tutte le altre informazioni necessarie alla corretta esecuzione di questa procedura, come ad esempio:

- la data di schedulazione,
- la policy di retry in caso di fallimento,
- lo stato attuale (da eseguire, in esecuzione, eseguito),
- il gruppo a cui appartiene
- altri (i.e. data di creazione, numero di tentativi effettuati, tags, ecc.)

Un *job* può

- essere eseguito in process instantaneamente,
- essere schedulato per l'esecuzione in background il prima possibile,
- essere schedulato per l'esecuzione in background ad una determinata data/ora
- essere eseguito in process instantaneamente, ed in caso di fallimento essere schedulato per l'esecuzione in background in accordo con le proprie policy di retry.
- essere ritentato in caso di fallimento una o più volte, in accordo con delle specifiche politiche di retry.

Per poter accodare dei job da eseguire dovremo avere in mano un istanza della classe `Recruiter\Recruiter`.

Warning:

Il recruiter é studiato per garantire l'esecuzione di ogni job **almeno** una volta.

Questo significa che, indipendentemente dalle policy di retry, in alcuni casi (anomali) un job potrebbe essere eseguito più di una volta.

Ciò potrebbe accadere se, ad esempio, un *worker* dovesse morire fatalmente dopo aver eseguito il job ma prima di riuscire a modificare lo stato del job in `eseguito`; in quel caso il job verrà, dopo un certo periodo di tempo, assegnato ad un altro worker e quindi eseguito una seconda volta.

5.2 Hello World

Abbiamo già visto questo esempio nel *capitolo relativo alla creazione di un oggetto Workable*.

Tenendo conto di aver sviluppato una classe **HttpRequestCommand** il codice più semplice che potremmo scrivere per accodare una richiesta Http é il seguente:

```
<?php

use MyDomain\HttpRequestCommand;
use MyDomain\Request;
use Recruiter\Recruiter;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$request = Request::post($url, $body);
HttpRequestCommand::fromRequest($request)
   ->asJobOf($recruiter)
   ->inBackground()
   ->execute() // this is the method defined in the Workable class
;
```

in questo modo verrà schedulato un `job` che chiamerà il metodo `execute()` della classe `HttpRequestCommand` e verrà eseguito non appena un `worker` sarà disponibile.

5.3 Schedule a Job in the future

Nel caso in cui volessimo che l'esecuzione del job sia programmata per il futuro (invece che quasi "istantanea"), possiamo farlo tramite il metodo `scheduleAt()` a cui dovremo pasasre un instanza di `Timeless\Moment`
Se ad esempio volessimo programmare l'esecuzione di una chiamata http per il giorno `19 gennaio 2038` potremmo fare in questo modo:

```
<?php

use MyDomain\HttpRequestCommand;
use MyDomain\Request;
use Recruiter\Recruiter;
```

(continues on next page)

(continued from previous page)

```
use Timeless\Moment;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$request = Request::post($url, $body);
HttpRequestCommand::fromRequest($request)
    ->asJobOf($recruiter)
    ->scheduleAt(Moment::fromDateTime(new DateTime('2038-01-19T00:00:00.000000Z');))
    ->inBackground()
    ->execute()
;
```

In questo modo il job verrà messo in coda e verrà eseguto non appena ci sarà un worker libero disponibile successivamente alla data '2038-01-19T00:00:00.000000Z'

5.4 Retry

Negli esempi visti in precdenza i job verrano eseguiti una sola volta, indipendentemente dal fatto che abbiano successo o meno.

In caso di fallimento di un job il recruiter ci dà la possibilità di specificare il fatto la sua esecuzione possa essere ritentata.
Per fare ciò dovremo assegnare una `Recruiter\RetryPolicy` al job tramite il metodo `retryWithPolicy(RetryPolicy $retryPolicy)`.

Vedremo più avanti *come poter creare una propria RetryPolicy*, nel frattempo possiamo utilizzare le retry policies già incluse nella libreria recruiter.

Supponiamo ad esempio di voler ritentare la nostra chiamata http nel caso in cui fallisca, di volere eseguire fino ad un massimo di tre retry e di voler attendere 60 secondi tra un tentativo e l'altro:

```
<?php

use MyDomain\HttpRequestCommand;
use MyDomain\Request;
use Recruiter\Recruiter;
use Recruiter\RetryPolicy\RetryManyTimes;
use Timeless\Moment;


$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$retryPolicy = new RetryManyTimes(3, 60);

$request = Request::post($url, $body);
HttpRequestCommand::fromRequest($request)
```

(continues on next page)

(continued from previous page)

```
    ->asJobOf($recruiter)
    ->scheduleAt(Moment::fromDateTime(new DateTime('2038-01-19T00:00:00.000Z')))
    ->retryWithPolicy($retryPolicy)
    ->inBackground()
    ->execute()
;
```

In base a questo esempio il nostro job verrà eseguito fino ad un massimo di 4 volte,
la prima volta verrà eseguito in data: *2038-01-19T00:00:00.000Z* come da schedulazione, in seguito, in caso di fallimento, verranno fatti 3 nuovi tentativi distanziati 60 secondi l'uno dell'altro, che avranno quindi luogo nelle date:
2038-01-19T00:01:00.000Z
2038-01-19T00:02:00.000Z
2038-01-19T00:03:00.000Z

Questo é un semplice esempio di come poter ripetere un job in caso di fallimento, le *Retry Policies* possono avere anche logiche molto più complesse, date uno sguardo alla *pagina dedicata* per capirne le potenzialità.

5.5 Retriable Exceptions

Indipendentemente dalla *RetryPolicy* utilizzata, possiamo sempre specificare in quali casi eseguire un nuovo tentativo e in quali no.
Il metodo `retryWithPolicy` permette infatti di specificare, come secondo argomento, un array di eccezioni per le quali é consentito eseguire un nuovo tentativo.
Nel caso in cui questo array sia vuoto (come nel caso di default), il job verrà tentato di nuovo qualsiasi eccezione venga sollevata.
Nel caso invece in cui questo array contiene una o più eccezioni, allora verrà effettuato un nuovo tentativo solo nel caso in cui venga intercettata un eccezione che sia un istanza di una delle classi contenute in questo array.
Es.:

```
<?php

$retryPolicy = new RetryManyTimes(3, 60);
$retriableExceptionTypes = [
    \Psr\Http\Client\NetworkExceptionInterface::class
];

HttpCommand::fromRequest($request);
    ->asJobOf($recruiter)
    ->retryWithPolicy($retryPolicy, $retriableExceptionTypes)
    ->inBackground()
    ->execute()
;
```

In questo caso il job verrà ripetuto solo in caso avvenga un eccezione di tipo `Psr\Http\Client\NetworkExceptionInterface`, in tutti gli altri casi il job verrà archiviato.

5.6 Optimistic Jobs

Ci potrebbero essere dei casi in cui abbiamo bisogno che una procedura sia eseguita nella maniera più reattiva possibile

Facciamo finta di essere un sistema di pagamento, e vogliamo avvisare un ipotetico merchant di un ipotetico acquisto andato a buon fine da parte di un ipotetico cliente.

Per assicurare la migliore user experience possibile ci interessa ovviamente notificare l'avvenuto pagamento al Merchant il prima possibile, in modo tale che il cliente riceva subito il suo prodotto.

Nel caso in cui l'endpoint atto a ricevere le notifiche di pagamento del Merchant non sia raggiungibile vorremmo che l'invio della notifica sia tentato nuovamente, magari dopo qualche minuto, sperando che nel frattempo l'endpoint sia tornato raggiungibile, non vogliamo però che il nostro processo si blocchi per qualche minuto quando potrebbe andare avanti a fare altre cose nel frattempo.

Il recruiter ci viene incontro anche in questo caso, é possibile infatti fare in modo che un job sia eseguito *in process* nel momento in cui viene schedulato, e, solo in caso di fallimento, venga accodato per l'esecuzione in background in modo da poter eseguire i successivi retry.

Es.:

```
<?php

$retryPolicy = new RetryManyTimes(3, 60);
$retriableExceptionTypes = [
   \Psr\Http\Client\NetworkExceptionInterface::class
];

HttpCommand::fromRequest($request);
   ->asJobOf($recruiter)
   ->retryWithPolicy($retryPolicy, $retriableExceptionTypes)
   ->execute()
;
```

Come potete notare l'unica cosa che abbiamo fatto é stata togliere la chiamata al metodo `inBackground()`, in questo modo il comando verrà eseguito subito, e, solo in caso di fallimento, verrà inserito nella coda dei job da eseguire in background.

Nel caso in cui non venga settata una RetryPolicy, il processo verrà eseguito subito e, sia in caso di successo sia in caso di fallimento, verrà archiviato senza nessun successivo tentativo.

Note:

Il metodo *inBackground()* viene implicitamente invocato nel caso in cui il job venga schedulato per l'esecuzione futura tramite il metodo *scheduleAt()*

Perciò queste 2 chiamate sono identiche ed in entrambi i casi l'esecuzione del job sarà esclusivamente in background.

```
<?php

HttpCommand::fromRequest($request);
   ->asJobOf($recruiter)
   ->retryWithPolicy($retryPolicy, $retriableExceptionTypes)
```

(continues on next page)

(continued from previous page)

```
    ->inBackground()
    ->execute()
;

HttpCommand::fromRequest($request);
    ->asJobOf($recruiter)
    ->retryWithPolicy($retryPolicy, $retriableExceptionTypes)
    ->scheduleAt(Moment::fromDateTime(new DateTime('2151-02-21T15:03:01.012345Z');))
    ->execute()
;
```

5.7 Raggrupare i Job

I *worker* (i processi che eseguono il lavoro descritto dai jobs) possono essere lanciati con l'intento di eseguire qualsiasi job disponibile oppure possono essere limitati all'esecuzione di un solo gruppo di jobs.
Questo modalità può tornarci utile, ad esempio, per *gestire priorità di esecuzione diverse a seconda dei jobs*.

Ogni *job* può essere assegnato, al massimo, ad un singolo gruppo e per farlo si utilizza il metodo `inGroup($group)`

```
<?php

HttpCommand::fromRequest($request);
    ->asJobOf($recruiter)
    ->inGroup('http')
    ->inBackground()
    ->execute()
;
```

5.8 Tags

É anche possibile taggare i jobs in modo tale da agevolare la ricerca di jobs o altre attività di query (es. statistiche).

```
<?php

HttpCommand::fromRequest($request);
    ->asJobOf($recruiter)
    ->taggedAs(['userId:42', 'color:red'])
    ->inBackground()
    ->execute()
;
```

CHAPTER 6

Retry Policies

6.1 Implements a custom RetryPolicy

All'interno della libreria sono presenti delle RetryPolicy che coprono i casi più comuni.
In caso di necessità potremo comunque creare una nuova policy in modo da coprire la nostra necessità.
Per creare una nuova policy dovremo creare una classe che implementi l'interfaccia `Recruiter\RetryPolicy`

6.2 DoNotDoItAgain

This is the default (implicit) *RetryPolicy*, use it only if you want to make explicit the fact that the job should not be repeated.

```php
<?php

use Recruiter\Recruiter;
use Recruiter\Workable\ShellCommand;
use Recruiter\RetryPolicy\DoNotDoItAgain;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

ShellCommand::fromCommandLine("false");
   ->asJobOf($recruiter)
   ->retryWithPolicy(new DoNotDoItAgain())
   ->inBackground()
   ->execute()
;
```

6.3 ExponentialBackoff

Questa *RetryPolicy* permette di ritentare l'esecuzione di un job ad intervalli esponenziali. Ad esempio possiamo impostare di avere un massimo di 10 retry con un intervallo iniziale di 30 secondi. Questo significa che dopo il primo fallimento verrà effettuato un retry dopo 30 secondi, in caso anche questo fallisca verrà effettuato un altro retry dopo 60 secondi, nel caso in cui anche questo fallisca verrà effettuato un nuovo retry dopo 120 secondi e cosi via, fino ad un massimo di 10 nuovi tentativi.

L' *ExponentialBackoff* policy accetta come parametri il numero massimo di tentativi da effettuare ed i secondi iniziali di intervallo prima di effettuare il primo tentativo.

Examples:

```
<?php

use Recruiter\Recruiter;
use Recruiter\Workable\ShellCommand;
use Recruiter\RetryPolicy\ExponentialBackoff;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$retryPolicy = ExponentialBackoff::forTimes(10, 30);
//This is the same to the following:
// $retryPolicy = new ExponentialBackoff(10, 30);

ShellCommand::fromCommandLine("false");
   ->asJobOf($recruiter)
   ->retryWithPolicy($retryPolicy)
   ->inBackground()
   ->execute()
;
```

..TODO: verificare al parte seguente prima di pubblicarla .. Questa policy comprende anche un factory method che accetta il numero di secondi massimo in cui riprovare ed il numero di secondi iniziali di intervallo prima di effettuare il primo tentativo: .. .. Examples: .. .. .. code-block:: php .. .. <?php .. .. use Recruiter-Recruiter; .. use RecruiterWorkableShellCommand; .. use RecruiterRetryPolicyExponentialBackoff; .. .. $mongodbInstance = new MongoDBClient(...); .. $recruiter = new Recruiter($mongodbInstance); .. .. // In this case there will be a maximum of 4 attempts: after the first failure a retry will be made after 30 seconds, another one after 60 seconds, and the last after 120 seconds .. $retryPolicy = ExponentialBackoff::forAnInterval(120, 30); .. .. ShellCommand::fromCommandLine("false"); .. ->asJobOf($recruiter) .. ->retryWithPolicy($retryPolicy) .. ->scheduleAt(Moment::fromDateTime(new DateTime('2027-02-21T15:00:00.0000Z'));) .. ->inBackground() .. ->execute() .. ;

6.4 RetryForevers

Questa *RetryPolicy* permette di ritentare l'esecuzione di un job all'infinito specificando l'intervallo di tempo tra un tentativo e l'altro. Se ad esempio volessimo eseguire un job all'infinito aspettando 30 secondi tra un tentativo e l'altro possiamo scrivere:

```
<?php

use Recruiter\Recruiter;
use Recruiter\Workable\ShellCommand;
```

(continues on next page)

(continued from previous page)

```
use Recruiter\RetryPolicy\RetryForevers;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$retryPolicy = RetryForevers::afterSeconds(30);
//This is the same to the following:
// $retryPolicy = new RetryForevers(30);

ShellCommand::fromCommandLine("false");
   ->asJobOf($recruiter)
   ->retryWithPolicy($retryPolicy)
   ->inBackground()
   ->execute()
;
```

6.5 RetryManyTimes

Questa *RetryPolicy* permette di ritentare l'esecuzione di un job un numero finito di volte specificando l'intervallo di tempo tra un tentativo e l'altro. Se ad esempio vogliamo ritentare un job per 3 volte, aspettando 30 secondi tra un tentativoo e l'altro, possiamo scrivere:

```
<?php

use Recruiter\Recruiter;
use Recruiter\Workable\ShellCommand;
use Recruiter\RetryPolicy\RetryManyTimes;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$retryPolicy = RetryManyTimes::forTimes(3, 30);
//This is the same to the following:
// $retryPolicy = new RetryManyTimes(3, 30);

ShellCommand::fromCommandLine("false");
   ->asJobOf($recruiter)
   ->retryWithPolicy($retryPolicy)
   ->inBackground()
   ->execute()
;
```

6.6 TimeTable

Questa *RetryPolicy* permette di ritentare l'esecuzione di un job ad intervalli regolari dipendenti da quanto tempo é passato rispetto alla creazione del job.

Ad esempio se volessimo ritentare il job * ogni minuto per i primi 5 minuti di vita del job, * ogni 5 minuti per la prima ora (cioé i successivi 55 minuti) .. * ogni 5 minuti per i successivi 55 minuti * ed ogni ora per le prime 24 ore (cioé le successive 23 ore) .. * ogni ora per le successive 23 ore

Possiamo scrivere il seguente codice:

```php
<?php

use Recruiter\Recruiter;
use Recruiter\Workable\ShellCommand;
use Recruiter\RetryPolicy\TimeTable;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$retryPolicy = new TimeTable([
    '5 minutes ago' => '1 minute',
    '1 hour ago' => '5 minutes',
    '24 hours ago' => '1 hour',
]);

ShellCommand::fromCommandLine("false");
    ->asJobOf($recruiter)
    ->retryWithPolicy($retryPolicy)
    ->inBackground()
    ->execute()
;
```

Warning: Questa policy accetta un array chiave-valore dove sia le chiavi che i valori devono essere stringhe parsabili dalla funzione php strtotime

CHAPTER 7

Repeatable Jobs

Ci sono dei casi in cui la procedura da eseguire deve essere ripetuta nel tempo, indipendentemente dal fatto che venga completata con sucesso o meno.
Questo può essere svolto grazie a dei particolari job che prendono il nome di *RepeatableJob*.
Vediamo come utilizzarli tramite un esempio.

Supponiamo di voler inviare, ogni giorno alle 06:00 UTC, un report ad un determinato indirizzo email.
Avremo bisogno quindi di *creare il nostro Workable* che contenga la procedura per generare/inviare il report, procediamo, come visto in precedenza, ad implementare la classe `Recruiter\Workable` utilizzando il trait `Recruiter\WorkableBehaviour` per evitare di scrivere codice ridondante.

```php
<?php

use Recruiter\Recruiter;
use Recruiter\Workable;
use Recruiter\WorkableBehaviour;

class DailyReportCommand implements Workable
{
   use WorkableBehaviour;

   public function execute()
   {
      // ...
      // here we generate the report and send it to the desired recipient
      // ...
   }
}
```

Ora dobbiamo far si che la nostra classe implementi anche l'interfaccia `Recruiter\Repeatable` in modo che possa essere schedulato automaticamente secondo un determinato schema.

```php
<?php

use Recruiter\Recruiter;
use Recruiter\Repeatable;
use Recruiter\RepeatableBehaviour;
use Recruiter\Workable;
use Recruiter\WorkableBehaviour;

class DailyReportCommand implements Workable, Repeatable
{
   use WorkableBehaviour, RepeatableBehaviour;

   public function execute()
   {
      // ...
      // here we generate the report and send it to the desired recipient
      // ...
   }

   public function urn(): string
   {
      return 'report:daily';
   }

   public function unique(): bool
   {
      return false;
   }
}
```

Abbiamo quindi assegnato un nome univoco al nostro *Repeatable* (tramite il metodo `urn()`), ed indicato al *Recruiter* se é possibile o meno che 2 o più istanze di questo *job* si sovrappongano (tramite il metodo `unique()`)

Ora che abbiamo un *Repeatable* vediamo come poterlo schedulare ad intervalli regolari.
Per indicare la politica di esecuzione del job dovremo utilizzare una `Recruiter\SchedulePolicy`
All'interno della libreria recruiter troviamo una *SchedulePolicy* già esistente che prende il nome di `Recruiter\SchedulePolicy\Cron` e permette di specificare gli intervalli di esecuzione con la stessa sintassi utilizzata dal demone unix cron.
Quindi, per inviare il nostro report ogni giorno alle *06:00 UTC* dovremo fare in questo modo:

```php
<?php

use Recruiter\Recruiter;
use Recruiter\SchedulePolicy\Cron;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$schedulePolicy = new Cron('0 6 * * *');

$scheduler = (new DailyReportCommand())
   ->asRepeatableJobOf($this->recruiter)
```

(continues on next page)

(continued from previous page)

```
    ->repeatWithPolicy($schedulePolicy)
    ->retryWithPolicy(new DoNotDoItAgain()) // this is the default behaviour
    ->create()
;
```

CHAPTER 8

Recruiter Hooks

E' possibile definire, nel proprio progetto, delle `hook functions` che verranno richiamate dal *Recruiter* in determinati momenti/stati.

Queste funzioni riceveranno come argomento un instanza del *Recruiter* e sono:

- recruiter_became_master
- recruiter_stept_back

8.1 recruiter_became_master

Questa funzione verrà invocata dal processo recruiter nel momento in cui ottiene la leadership.

Per maggiori informazioni su cosa significhi ottenere la leadership vedere il capitolo relativo a Geezer

```php
<?php

use Recruiter\Recruiter;

function recruiter_became_master(Recruiter $recruiter): void
{
   // Schedule a Recruiter\Repeatable job
}
```

8.2 recruiter_stept_back

Questa funzione verrà invocata dal processo recruiter che perde la leadership.

Per maggiori informazioni su cosa significhi perdere la leadership vedere il capitolo relativo a Geezer

```php
<?php

use Recruiter\Recruiter;

function recruiter_stept_back(Recruiter $recruiter): void
{

}
```

CHAPTER 9

How to handle priority

Supponiamo di avere un applicazione di tipo e-commerce che abbia al suo interno queste funzionalità.

- invio di un messaggio email di double-option (per la verifica dell'indirizzo email prima dell'acquisto).
- invio di una email di conferma acquisto.
- invio di una email di follow-up agli utenti che non hanno completato l'acquisto dopo 7 giorni.

In tutti e tre i casi si tratta dell'invio di un email (possiamo avere quindi un solo tipo di `Workable`), ma tutte e tre hanno una diversa priorità:

Nel primo caso vogliamo che l'email arrivi il prima possibile, l'utente é davanti allo schermo che attende quell'email e vogliamo dargli il miglior servizio possibile, inoltre più tardi arriva e più é facile che l'acquisto venga abbandonato.

Nel secondo caso vogliamo che l'email arrivi non troppo dopo che l'acquisto sia stato completato ma senza neanche una particolare fretta, cercando un compromesso tra il dare il miglior servizio possibile all'utente e a non sovraccaricare troppo il nostro sistema.

Nell'ultimo caso invece non abbiamo nessuna fretta, infatti che l'email venga inviata dopo 7 giorni esatti o dopo 7 giorni e 2 minuti cambia molto poco, l'utente non se ne accorgerà ed anche il business (in questa nostra ipotesi) non ne risentirà.

Per attuare questa strategia possiamo innanzitutto *suddividere i jobs in diversi gruppi*, in questo specifico caso andremo quindi a creare 3 gruppi:

- *double-optin-email* (or *high-priority*)
- *confirmation-email* (or *generics*)

- *follow-up-email* (or *low-priority*)

Una volta fatto ciò, andremo ad istruire i `workers` per prendersi carico solo di una determinata tipologia di gruppo. In questo modo, grazie al numero di `workers` presenti su ogni coda (gruppo) avremo velocità di evasione dei `jobs` differenti.
Ad esempio supponiamo di voler abilitare sette workers, possiamo suddividerli in questa modalità:

- 1 worker che lavora sulla coda `follow-up-email` (or `low-priority`)
- 2 worker che lavorano sulla coda `confirmation-email` (or `generics`)
- 4 worker che lavorano sulla coda `double-optin-email` (or `high-priority`)

Cosi facendo verrà eseguito:

- un solo job facente parte del gruppo `follow-up-email` alla volta (Quindi nel caso in cui ci siano 2 jobs nel gruppo `low-priority` schedulati entrambi per lo stesso orario, il secondo verrà eseguito solo al termine del primo).
- due job facenti parti del gruppo `confirmation-email` in parallelo
- quattro job facenti parti del gruppo `double-optin-email` in parallelo

In linea di massima possiamo affermare che più worker ci sono per una determinata coda (gruppo) e più quella coda verrà smaltita velocemente.

Per limitare il lavoro di un worker ad uno specifico gruppo di jobs dovremo utilizzare l'opzione **work-on** al lancio del processo *worker*.
Ad esempio:

```
$ php vendor/bin/recruiter start:worker --work-on='double-optin-email' --target 127.0.
↪0.1:27017 --bootstrap $APP_BASE_PATH/worker-boostrap.php
$ php vendor/bin/recruiter start:worker --work-on='double-optin-email' --target 127.0.
↪0.1:27017 --bootstrap $APP_BASE_PATH/worker-boostrap.php
$ php vendor/bin/recruiter start:worker --work-on='double-optin-email' --target 127.0.
↪0.1:27017 --bootstrap $APP_BASE_PATH/worker-boostrap.php
$ php vendor/bin/recruiter start:worker --work-on='double-optin-email' --target 127.0.
↪0.1:27017 --bootstrap $APP_BASE_PATH/worker-boostrap.php

$ php vendor/bin/recruiter start:worker --work-on='confirmation-email' --target 127.0.
↪0.1:27017 --bootstrap $APP_BASE_PATH/worker-boostrap.php
$ php vendor/bin/recruiter start:worker --work-on='confirmation-email' --target 127.0.
↪0.1:27017 --bootstrap $APP_BASE_PATH/worker-boostrap.php

$ php vendor/bin/recruiter start:worker --work-on='follow-up-email' --target 127.0.0.
↪1:27017 --bootstrap $APP_BASE_PATH/worker-boostrap.php
```

CHAPTER 10

Analytics

Il recruiter mette a disposizione delle statistiche sullo stato attuale delle code.
Questo può essere utilizzato, ad esempio, per monitorare che tutte le code vengano smaltite correttamente.

Per poter ricavare i dati statistici sullo stato delle code bisogna chiamare il metodo `analytics` dell'oggetto `Recruiter\Recruiter`.
Il valore restituito sarà un array contenente:

- **jobs:**
  - **queued**: il numero di jobs in coda con una data di schedulazione passata (e quindi da eseguire), questo numero dovrebbe rimanere stabile.
  - **postponed**: il numero di jobs in coda con una data di schedulazione futura (da eseguire solo quando la data di schedulazione sarà passata).
- **throughput:**
  - **value**: numero di job eseguiti al minuto
  - **value_per_second**: numero di job eseguiti al secondo
- **latency:**
  - **average**: Il numero medio di secondi che passa dalla data di schedulazione alla data di esecuzione del job. Un valore alto significa che ci sono troppi pochi worker in esecuzione per quella specifica coda.
- **execution_time:**
  - **average**: il tempo di esecuzione medio di un job.

```php
<?php

use Recruiter\Recruiter;

$mongodbInstance = new MongoDB\Client(...);
```

(continues on next page)

(continued from previous page)

```php
$recruiter = new Recruiter($mongodbInstance);
$analytics = $recruiter->analytics();

var_export($analytics);
// array (
//     'jobs' => array (
//        'queued' => 10,
//        'postponed' => 30,
//        'zombies' => 0,
//     ),
//     'throughput' => array (
//        'value' => 3.0,
//        'value_per_second' => 0.05,
//     ),
//     'latency' => array (
//        'average' => 0.0,
//     ),
//     'execution_time' => array (
//        'average' => 0,
//     ),
// )
```

Per visualizzare le statistiche relative ad uno specifico gruppo di job é possibile passare il gruppo come primo argomento alla funzione analytics.
Per ulteriori modalità di utilizzo fare riferimento direttamente al codice sorgente del metodo “analytics”.

```php
<?php

$analytics = $recruiter->analytics('custom-group');
```

CHAPTER 11

Mongo Collections

Come abbiamo già visto, la libreria *recruiter* si appoggia a *Mongodb* per la storicizzazione dei dati.
Vediamo a grandi linee la struttura utilizzata in modo da possedere una conoscenza di massima che vi renderà più facile le indagini in caso di comportamenti anomali.

11.1 "roster" collection

La collezione **roster** contiene i dati relativi ai vari *worker* in esecuzione.

É grazie a questa collezione che il processo *recruiter* conosce quali worker sono presenti e quali di questi sono disponibili a prendere in carico un nuovo *job*, ed é sempre in questa collezione che il processo *recruiter* utilizza per memorizzare quale *job* é stato assegnato a quale *worker*. In questo modo ogni processo *worker* legge ripetutamente (polling) il proprio documento così da individuare quale sarà il prossimo *job* da eseguire.

Ogni processo *worker* registra all'avvio i propri dati in un documento di questa collezione, questo documento viene rimosso durante la fase di shutdown del worker.

Ogni processo *worker* aggiorna questo documento periodicamente con la data corrente, in maniera tale da rendere esplicito il fatto di essere ancora "vivo".

Grazie a questa data, il recruiter può capire che il *worker* non é più online, potendo rimuovere il documento relativo al *worker* morto ed evitando cosi di assegnargli dei lavori da eseguire.

11.2 “scheduled” collection

La collezione **scheduled** contiene i vari *jobs* da eseguire.

Il processo recruiter legge periodicamente (polling) questa collezione in modo da individuare quali *jobs* vanno eseguiti, in base alla loro data di schedulazione.

Nel caso in cui un *job* venga eseguito senza successo, la data di schedulazione verrà aggiornata in relazione alla proprio politica di retry, in caso del raggiungimento del numero massimo di retry il documento verrà spostato nella collezione **archived**

11.3 “archived” collection

La collezione **archived** contiene lo storico dei vari *jobs* eseguiti.

Un *job* viene spostato dalla collezione **scheduled** alla collezione **archived** nel caso in cui venga eseguito e completato con successo, oppure nel caso in cui l’esecuzione fallisca ed é stato raggiunto il massimo numero di tentativi di esecuzione.
Il processo *cleaner* si occupa di mantenere ridotte le dimensioni di questa collezione, cancellando i *jobs* più vecchi di 5 giorni (default).
É possibile modificare questa finestra temporale tramite l’opzione **clean-after** del processo *cleaner*.

Questa collezione risulta molto utile per 2 motivi:

- indagare i motivi del fallimento di un job (nel documento viene incluso lo stato del job (completato o meno) e il motivo dell’ultimo fallimento, più altri dati utili)
- *rischedulare un job*

11.4 “schedulers” collection

La collezione **schedulers** contiente un template dei *job* che devono essere eseguiti periodicamente.

Il processo *recruiter* legge periodicamente (polling) questa collezione in modo da creare e schedulare dei nuovi *job* da aggiungere alla collezione *scheduled*.

CHAPTER 12

Administration console

La libreria fornisce alcuni comandi da console per la gestione del recruiter:

12.1 Recovering a job

Nel caso in cui un volessimo rieseguire un *job* che si trova in *archivio* possiamo farlo tramite il comando console **job:recover**

Per poter eseguire questo comando é necessario conoscere l'id (MongoId del documento) del *job* che si vuole ripristinare, da passare come argomento del comando.

É possibile specificare la nuova data di schedulazione tramite l'opzione **scheduleAt**, altrimenti il job verrà rischedulato per l'esecuzione nella data corrente.

```
$ php vendor/bin/recruiter job:recover --target mongodb://localhost:27017/recruiter --
↪scheduleAt "2019-12-01T22:18:00Z" 5d27436e2bacd566a67e85e4
```

12.2 Analytics

É possibile visulizzare le *statistiche* anche in console tramite il comando: **bko:analytics**

É possibile specificare l'uri del server mongo al quale connettersi tramite l'opzione `target`.

É possibile limitare le statistiche ad un solo gruppo di job tramite l'opzione `group`.

```
$ php vendor/bin/recruiter bko:analytics --target mongodb://localhost:27017/recruiter␣
↪--group html
```

CHAPTER 13

Esecuzione dei jobs all'interno dei tests:

Durante l'esecuzione di test che esercitano del codice che utilizza la libreria *recruiter* per accodare dei *jobs*, ci troviamo davanti al problema di dover far si che quei *jobs* vengano eseguiti pena il possibile fallimento dei test.

Per fare in modo che i *jobs* in coda vengano eseguiti potremmo banalmente replicare quello che avviene nell'ambiente di produzione e quindi attivare i processi di *recruiter*, *jobs* e *cleaner*.

A seconda dell'ambiente di test in cui ci troviamo, questa soluzione può presentare degli svantaggi, come ad esempio:

- **Maggiore difficoltà di esecuzione dei test**: in quanto il nostro ambiente deve prevedere l'esecuzione di long running process ed assicurarsi che siano attivi durante l'intera esecuzione del test.
- **Diminuzione della velocità di esecuzione dei test**: Nel caso in cui i test dipendano dal risultato dell'esecuzione dei job dovremmo attendere la loro esecuzione da parte dei worker, che per quanto reattivi possano essere non possono essere istantanei; dobbiamo inoltre considerare il fatto che i job potrebbero essere schedulati nel futuro e che quindi i worker non potranno eseguirli finché la data di schedulazione non sia passata.
- **Impossibilità di dipendere da job schedulati molto avanti nel futuro**: Se nel caso in cui i job siano schedulati a qualche secondo di distanza dal momento corrente porta al solo rallentamento di esecuzione dei test, il caso in cui i job siano schedulati molto avanti nel futuro (es. il giorno seguente o il mese seguente) porta all'impossibilità di esecuzione dei test (non possiamo certo attendere cosi tanto tempo per terminarne l'esecuzione di un test).

Tutti questi punti possono essere risolti utilizzando un metodo, che ci mette a disposizione la classe `Recruiter\Recruiter`, che permette l'esecuzione, nel processo corrente, di tutti i job precedentemente accodati.

Il metodo é `flushJobsSynchronously()` e può essere chiamato su qualsiasi istanza della classe `Recruiter\Recruiter` (quindi non per forza la stessa istanza utilizzata per accodare i *job*).

Grazie a questo metodo possiamo assicurarci che tutti i job in coda vengano eseguiti senza dover avere un ambiente con i processi *recruiter*, *worker* e *cleaner* attivi e senza dover attendere la data di schedulazione di ognuno di essi.

```php
<?php

namespace Tests;

use Core\DomainService;

$mongodbInstance = new MongoDB\Client(...);
$recruiter = new Recruiter($mongodbInstance);

$domainService = new DomainService($recruiter);
$domainService->methodThatQueuesJobs();

$recruiter->flushJobsSynchronously(); // Here all previously queued jobs are executed
```

CHAPTER 14

Documentazione da scrivere:

- dipendenze
- perché esiste
- che problemi risolve é sufficiente dire che permette l'esecuzione di routine in background?
- **struttura**
  - cos'é recruiter
  - cos'é un worker
  - cos'é il cleaner
- **setup**
  - installazione
  - lanciare recruiter
  - lanciare workers
  - lanciare cleaners
- **esempi**
  - hello world: task singolo che deve essere completato
  - senza retry policy (hello world)
  - schedule subito
  - schedule nel futuro
  - con retry policy semplice
  - retryable exceptions
  - retry policy complesse (exponential backoff)
  - task ottimistico
- **approfondimenti**

- come si implementa una workable
- come si implementa una retry policy (non é dettagliato, ma uno si può guardare l'implementazione di una delle tante esistenti per capire come fare. . .)
- tags
- recruiter statistiche
- hooks
- repeatable jobs
- working-on (priority queues)
- collezioni (a che serve archived, etc.)
- come rimettere un job in coda
- esecuzione nei test
- iniettare il logger
- geezer