Effective Background Processing with CakePHP 3

The need for background processing

"Why is the web server down again?"

Me, 2010

Maybe you know this already, but it is important

Maybe you know this already, but it is important

- There are a limited number of "workers" in your web server.
- The busier those workers are, the less requests they can handle
- You don't want to let your users waiting!
- Web requests are actually quite difficult to debug and reproduce.
- A platform where you have insight of what is going on the inside is the best one

When is background processing a good idea?

- Dealing with IO (emails, info from external API)
- Batch processing (data processing, cleaning up, loading data)
- Caching (database de-normalization, computing columns)
- Scheduled tasks
- Document generation (PDF, videos, images...)
- Auditing (integrity checks, changesets storage)
- Recommendation engines and machine learning

Types of background processing

Scheduled

E.g. Cron jobs

Scheduled

- Can be any code that can be invoked through the command line
- Requires a scheduler to executed the program (for example, cron)

Important questions to ask:

- What happens is two or more processes are trying to execute the same work?
- How do I know if the job was not executed successfully?
- How can I debug my program without breaking anything else?
- What happens if the process dies in the middle of its execution?
- How do I know how much time left until the process finishes once it starts?

Scheduled

Advantages

- No need to worry about connection timeouts
- No need to worry about messing up with global environment
- Very easy to debug

Disadvantages

- You have to manually prevent multiple workers doing the same job
- Can be difficult to know what is running at a specific time
- It is difficult to "resume" a job if it dies before it is done

A simple worker

```
// Shell/Recommender.php

class RecommenderShell extends Shell
{
  public function getOptionParser() {
    return parent::getOptionParser()
      ->description('Calculates recommendations and send them to the users')
      ->addOption('maximum', ['help' => 'The maximum amount of things to recommend'])
  }

  public function main()
  {
    $this->log('Calculating recommendations', 'info');

    $recommender = new Recommender($this->loadModel('Users'));
    $things = $recommender->calculate($this->params['maximum']);

    $this->log('Sending recommendations', 'info');
    return (bool) (new Sender())->send($things);
  }
}
```

Add it to the cron tab (everyday at 8:30am)

```
30 8 * * * cd /my/app; bin/cake recommender --maximum 3
```

Things to keep in mind

Define a getOptionsParser()

- You want to understand what your app does from the command line without reading the code
- Learn about all the info you can present in the help for your command.
- Keep it up to date
- Great for auto-completion

```
Usage:
cake translations [subcommand] [-h] [-q] [-v]

Subcommands:

fetch     Fetch latest translations from API and save them to
          `app/Locale/`
push      Push new translation keys - it does not override existing
          translations
snapshot  Create project snapshot, that can be restored in project
          settings

To see help on a subcommand use `cake translations [subcommand] --help`
```

Things to keep in mind

Don't echo to the console

Bad

```
$this->out('Starting the calculation');
```

Good

```
$this->log('Starting the calculation', 'info');
```

Things to keep in mind

Remember to show progress

Otherwise you have no idea of what your process is doing!

```
  public function main()
  {
    $this->log('Calculating recommendations (step 1 of 2)', 'info');

    $recommender = new Recommender($this->loadModel('Users'));
    $helper = $this->helper('Progress')->output(['total' => $recommender->total]);
    $progress = function($completed) use ($helper) {
      $helper->increment($completed);
      $helper->draw();
    };

    $things = $recommender->calculate($this->params['maximum'], $progress);

    $this->log('Sending recommendations (step 2 of 2)', 'info');
    ...
}
```

Things to keep in mind

Return a boolean in your method

```
function main()
{
  ...
  return $success;
}
```

The return value is used to determine whether or not your job finished successfully.

Things to keep in mind

Commit your crontab

```
cat config/crontab

30 8 * * * cd /my/app; bin/cake recommender --maximum 3

git commit -a config/crontab
```

And load it on each deploy

```
crontab config/crontrab
```

There is no need to setup cron jobs as a privileged user.

Dealing with problems....

Two processes doing the same job

- Cron prevents this case (only when run in a single machine)

Knowing when a job fails

- Cron sends you an email with the errors (ugh, but still something)

Debugging a program

- Either run it locally with the same data or look at the logs (there should be plenty)

Dealing with problems...

Knowing how much time left

- Difficult to know

Handling dying processes

- Difficult to handle

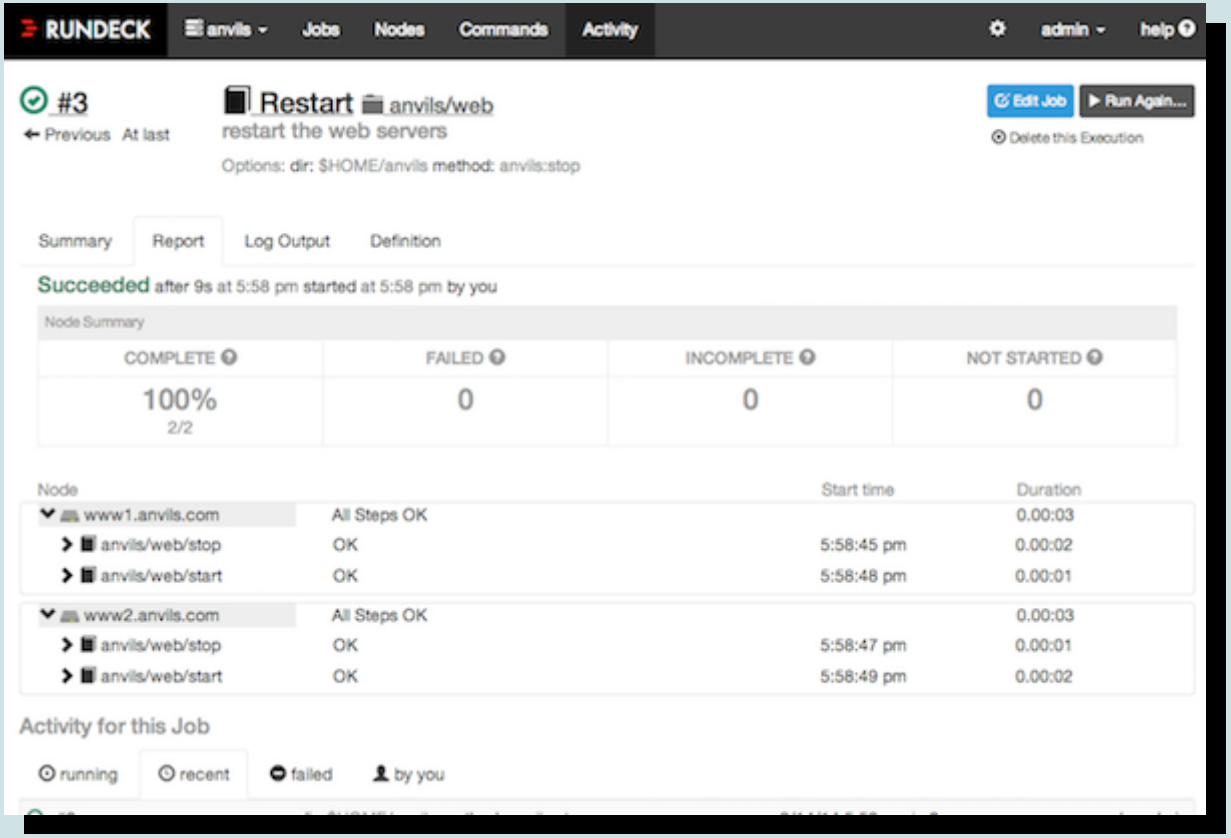
Going beyond Cron

We need something that:

- Is friendlier at configuring than crontabs
- Has better support for running jobs in multiple nodes
- Improves on the failure notification experience
- Reports time left for a job to finish

Use Rundeck

Rundeck is a cron replacement with a web interface capable of monitoring jobs across multiple machines.



Use Rundeck

With Rundeck you can:

- Define jobs using a the web interface or a rest API
- Monitor jobs live as they are run
- Get an overview of how much time until a job finishes
- Have better notification options (Slack, IRC, HipChat, PagerDuty...)
- Get insights on a job's hisotry (number of fails vs number of successful runs)

Installation

It is just a couple commands away!

```
apt-get install openjdk-7-jdk
dpkg -i rundeck-2.6.7-1-GA.deb
```

Types of background processing

Unscheduled

E.g. Job Queues

Unscheduled

- Require both queueing software and a process supervisor
- Can execute jobs on demand

Important questions to ask:

- ~~What happens is two or more processes are trying to execute the same work?~~
- ~~How do I know if the job was not executed successfully?~~
- How can I debug my program without breaking anything else?
- What happens if the process dies in the middle of its execution?
- How do I know how much time left until the process finishes once it starts?

Unscheduled

Advantages

- Allow to executed jobs immediately
- Automatically prevents workers doing the same job
- Better suited to workflow-based processing.
- Allows to stop a process and resume later
- More resilient to dying processes, if done correctly

Disadvantages

- If done incorrectly (wrong selection of tools) it can be a real nightmare.
- More difficult to debug.
- Requires you to install more software that needs to be monitored

Uscheduled

Doing it right

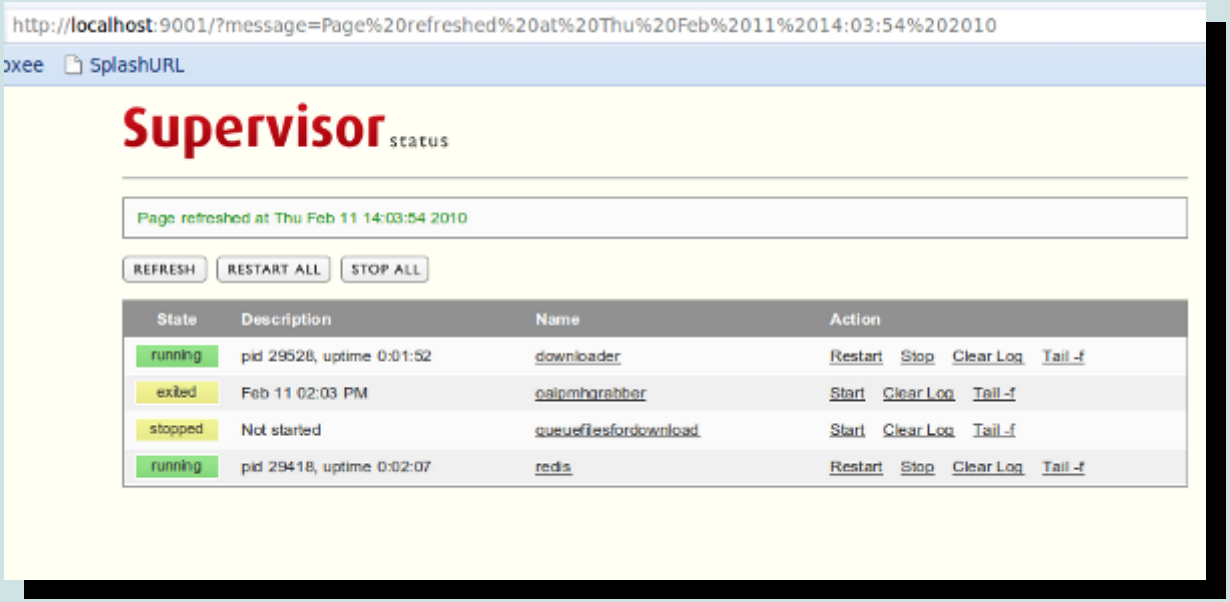
- Use a real process supervisor, for example supervisord
- Use a real queueing software, for example RabbitMQ

Doing it wrong

- Use a poor man's supervisor (see laravel's queue:listen)
- Use redis, mysql or zeromq as a queuing software

Supervisord

- Define the shell commands you want to use as queue workers in the config file
- Go to the admin interface and see how your jobs are doing!



Supervisord

Commit your supervisor.conf to your repo

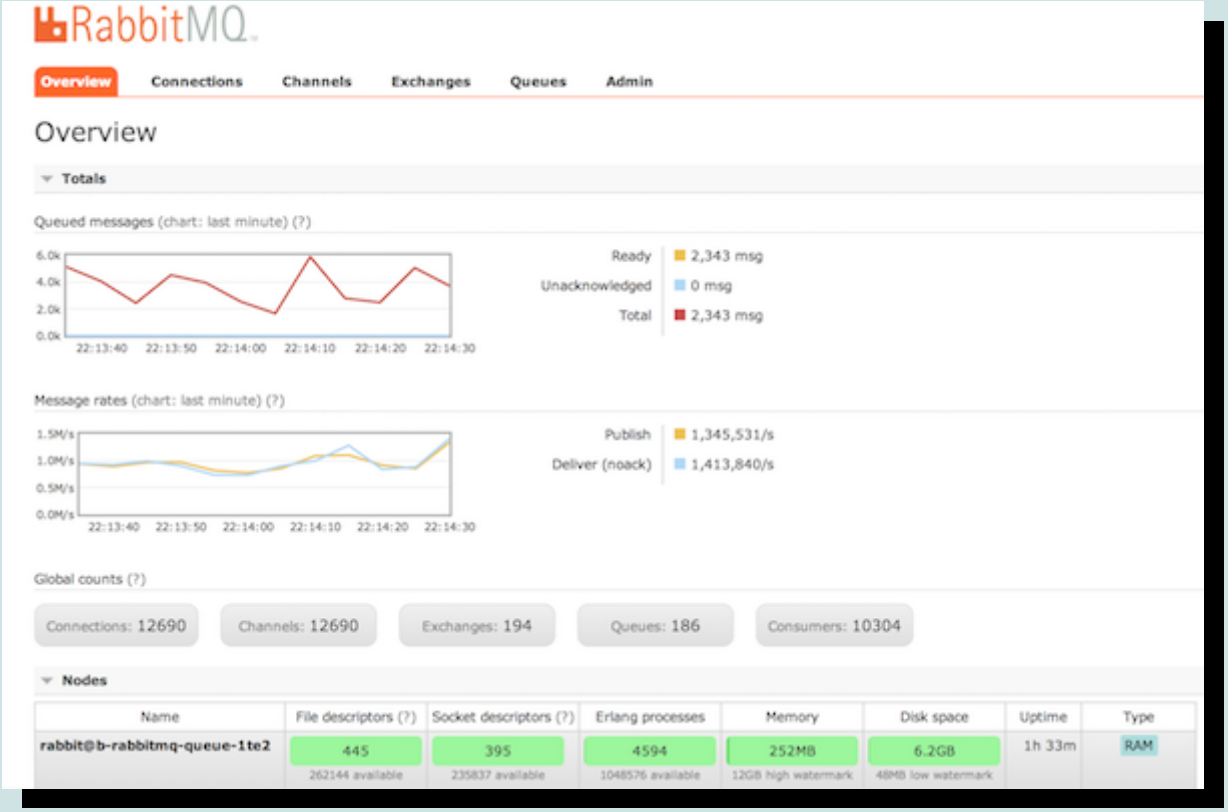
```
cat config/supervisor.conf

[program:send_welcome_email]
command = bin/cake welcome_emails
directory = /path/to/my/app
numprocs = 2
autostart = true
autorestart = true
```

Link it after every deploy

```
ln -s config/supervisor.conf /path/to/supervisor/conf.d/my_app.conf
```

RabbitMQ

A super stable mesage queue with an great interface:



Getting ready for our queueing system

```
pecl install amqp
composer require friendsofcake/process-mq; bin/cake plugin load ProcessMq
composer require friendsofcake/process-manager; bin/cake plugin load ProcessManager
```

```php
// config/app.php
...
'Queues' => [
  'send_welcome_email' => [
    'publish' => [
      'exchange' => 'emails',
      'routing' => 'welcome',
      'compress' => false,
      'delivery_mode' => 2, // Persist message on disk
    ],
    'consume' => [
      'exchange' => 'emails',
      'prefetchCount' => 3 // Optimize network activity
    ]
  ]
]
```

Creating our worker

```
class EmailSenderShell enxtends Shell
{
  public $tasks = ['ProcessMQ.RabbitMQWorker'];

  public function welcome()
  {
    $this->RabbitMQWorker->consume('send_welcome_email', [$this, 'doSendWelcome'])
  }

  public function doSendWelcome($userId)
  {
    // Send the email
    ...
    return true; // I'm done, remove the job from the queue
  }
}
```

Send a message to the queue

```
Queue::publish('send_welcome_email', $user->id);
```

Handling errors

```
public function doSendWelcome($userId)
{
  ...
  $success = (bool)$email->send();
  return $sucess; // If false, the message will be requeued
}
```

Handling exceptions

Messages are automatically requeued when exceptions happen, but the process will also die. That's a good thing.

Emergency stopping

In case of manually having to quite a worker, the job will gracefully wait until the job is done before exiting.

```
Sun, 22 May 2016 20:40:27 +0000 - info      -
Sun, 22 May 2016 20:40:27 +0000 - info      - Ready to serve requests.... (PID: 7930)
Sun, 22 May 2016 20:40:27 +0000 - info      -
Sun, 22 May 2016 20:40:27 +0000 - debug     - Successfully subscribed to signal SIGHUP (1)
Sun, 22 May 2016 20:40:27 +0000 - debug     - Successfully subscribed to signal SIGTERM (15)
Sun, 22 May 2016 20:40:27 +0000 - debug     - Successfully subscribed to signal SIGINT (2)
Sun, 22 May 2016 20:40:27 +0000 - debug     - Successfully subscribed to signal SIGQUIT (3)
Sun, 22 May 2016 20:40:27 +0000 - debug     - Successfully subscribed to signal SIGUSR1 (10)
Sun, 22 May 2016 20:40:27 +0000 - debug     - Successfully subscribed to signal SIGUSR2 (12)
Sun, 22 May 2016 20:40:50 +0000 - warning   - Got OS signal "SIGINT"
Sun, 22 May 2016 20:40:50 +0000 - warning   - Not doing any jobs, going to die now...
```

Time left for a process

- We now look at a list of jobs left in the queue
- Go to RabbitMQ's admin panel
- Devide total number of jobs in the queue by the message rate

```
$seconds = $messagesInQueue / messagesPerSecond;
```

Things to keep in mind

Break tasks in small parts

```
public function sendDailyEmails()
{
  $query = TableRegistry::get('Users')
    ->find('subscribed');

  $progress $this->helper('Progress')->output(['total' => $query->count()])

  $query
    ->bufferResults(false)
    ->each(function ($user) use ($progress) {
      Queue::publish('send_daily_email', $user->id);
      $progress->increment(1);
      $progress->draw();
    });
}
```

Never send php serialized objects to the queue

Use arrays or plain ids you can look up in the database instead.

Bad

```
Queue::publish('something', serialize($userObject));
```

Good

```
Queue::publish('something', $userObject->id);
```

Is it ok to lose a job?

If a process die in a weird way the message cannot be requeued. Or if the queueing server goes down... is it ok if the job is lost?

Yes

```
'deliveryMode' => 1
```

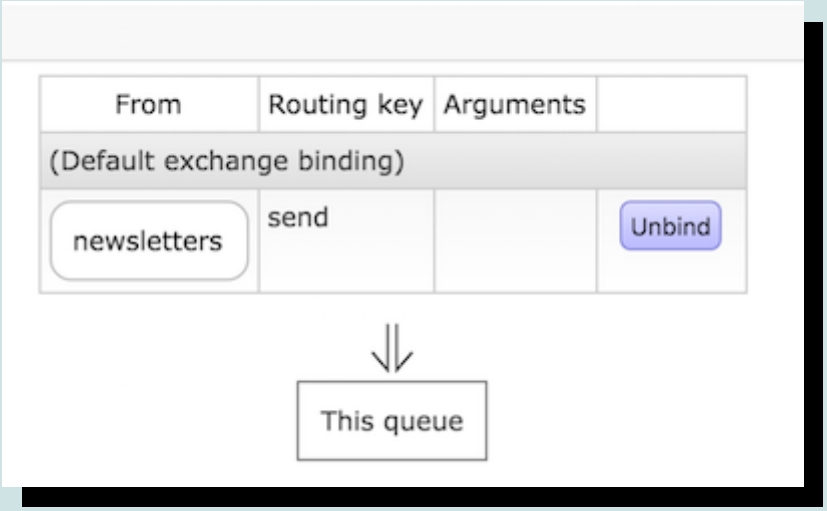
No

```
'deliveryMode' => 2
```

You can live-replay production jobs

If you need to debug a weird debug happening in production. Just connect to production...

1. Clone a queue with a different name.
2. Connect it to the same exchange and routing
3. Configure your local machine to conencto to production's RabbitMQ
4. Watch as live data comes in!



Don't trust machines, the are trying to take over

Use an auditing tool to figure out who's changing what in the background

```
composer intall lorenzo/audit-stash
```

```
class ProfitCalculatorShell extends AppShell
{

public function initialize()
{
  EventManager::instance()->on(new ApplicationMetadata('profit_calculator'));
}
```

Any changes to the tables having AuditStash enabled will now be tagged as being made by this shell.

Thanks!

Got questions?