Jobs, queues & events

Anatomie des erreurs courantes et pistes de résolutions

Salut!

Jonathan Van Belle aka Grummfy

🇧🇪

@grummfy@mamot.fr

@grummfy.bsky.social

Gitlab

Github

🌐 grummfy.be

👨‍💼 itlink

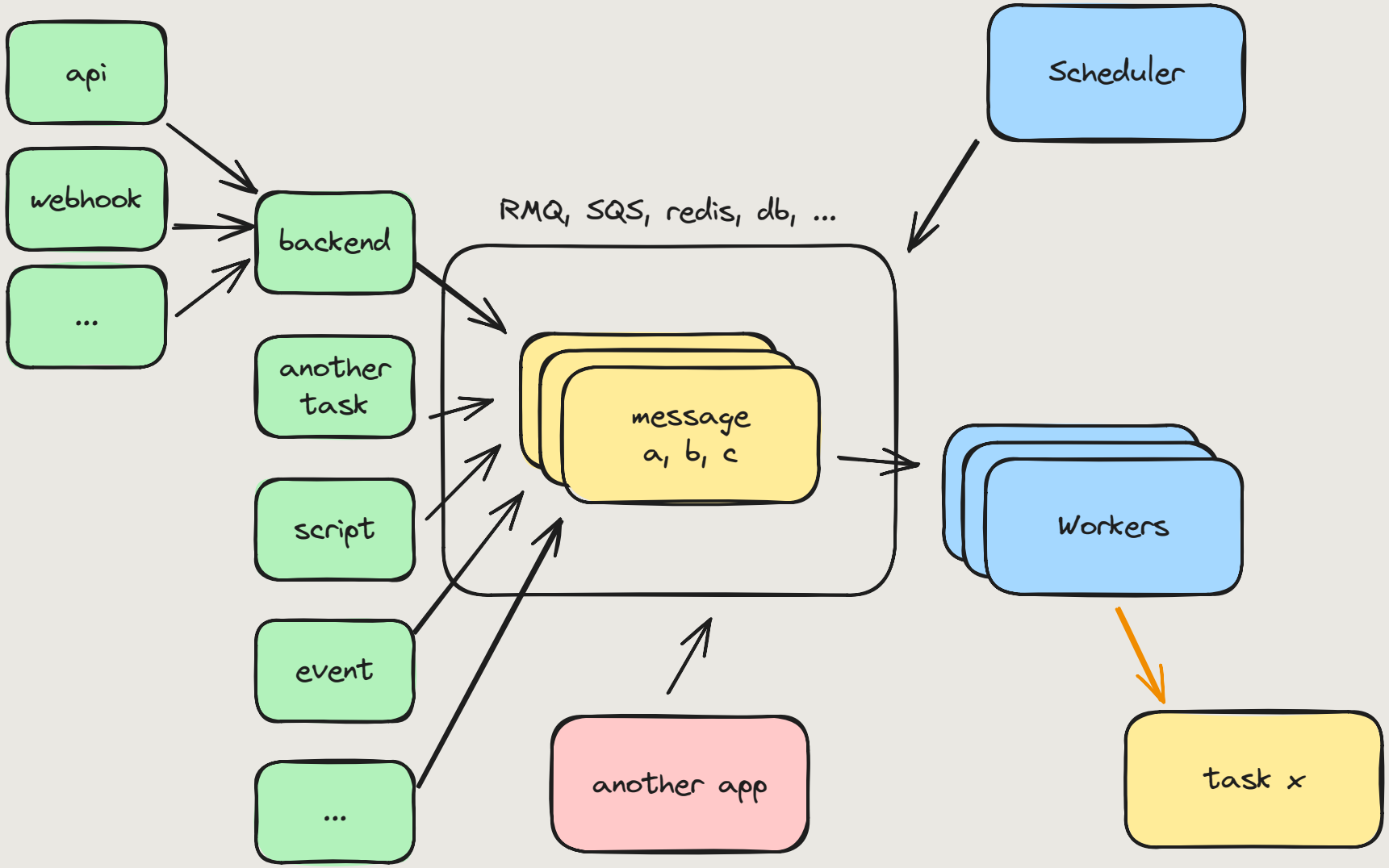
@atoum@phpc.social

Pourquoi ce talk?

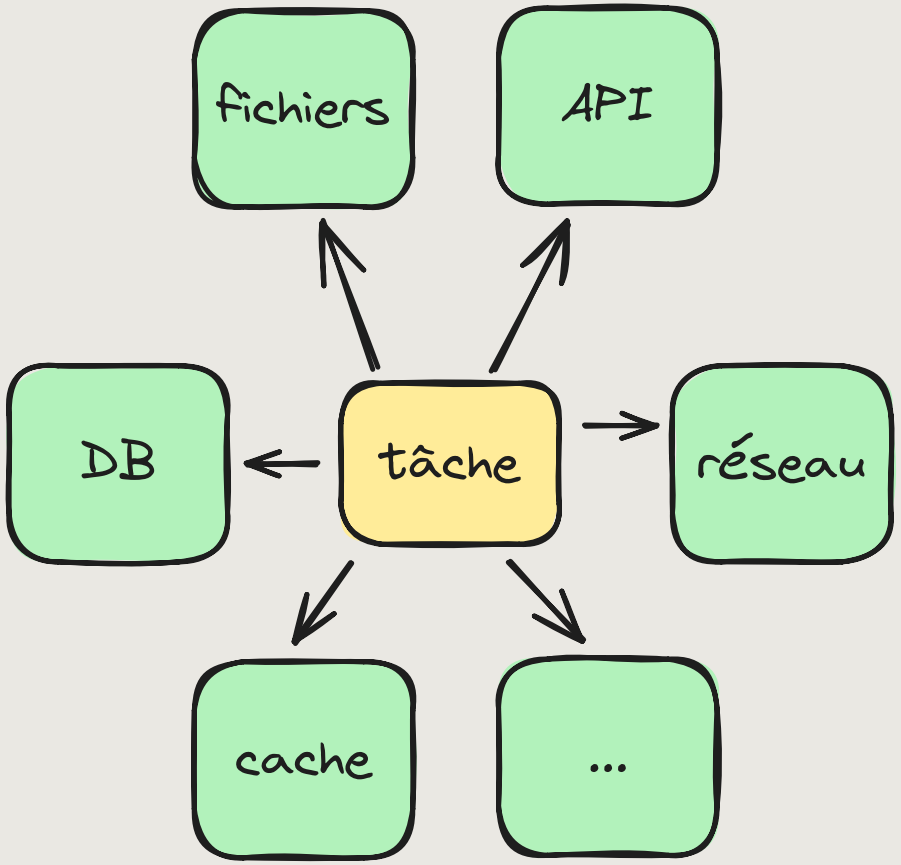
> Je voix toujours la même chose, en PHP ou d'autres langages

On commence un projet,
On ajoute 1, 2, ... 10, ... 50 jobs / workers,
Et bardaf, c'est l'embardée!

Jobs, Queues, Events, Worker, ...



Tâche ?!?



Tout va bien … puis KABOUM

- Saturation mémoire
- DB ou cache qui crash
- Disque plein
- Bus message qui ne répond plus
- API 404, 500, …
- Instabilité réseau
- Des erreurs non gérées qui font crashé vos workers.
- …

⇒ Bref, c'est la cata!
Et bien souvent au plus mauvais moment!

La tâche

Votre exécution

Gestion de la mémoire - cas pratique

Rapport `DB` ⟶ `map/filter/enrich` ⟶ `write`

Import `file` ⟶ `read` ⟶ `map/filter/enrich` ⟶ `insert/update DB`

- 1000 lignes
- + 1000 enregistrements dans un tableau
- + 1000 enregistrements dans un deuxième tableau
- + 100 lignes d'enrichissement via des systèmes externes
- ...
- 3000+ lignes vs 1000 de départs

Gestion de la mémoire - Mitigation des soucis

- Base de données
  - Query avec des **limit**
  - Batch insert
- Générateur (**yield**)
- **stream_*** && buffer en lecture/écriture

- Caches de certaines libs (exemple : phpspreadsheet)
- Optimisation mémoire globale
  - *objet* vs *array of ... of array*
  - Version de PHP

Gestion de la mémoire - générateur

```php
function getResults() {
  $page = 0;
  while ($rows = $db->giveRecords($page)) {
    yield $rows;
        $page++;
} }

function mapResultToObj($rows) {
  foreach ($rows as $row) {
    yield resultToObj($row);
} }

function enrichData($rows) {
  foreach ($rows as $row) {
    yield enrichFromApi($newRow);
} }
```

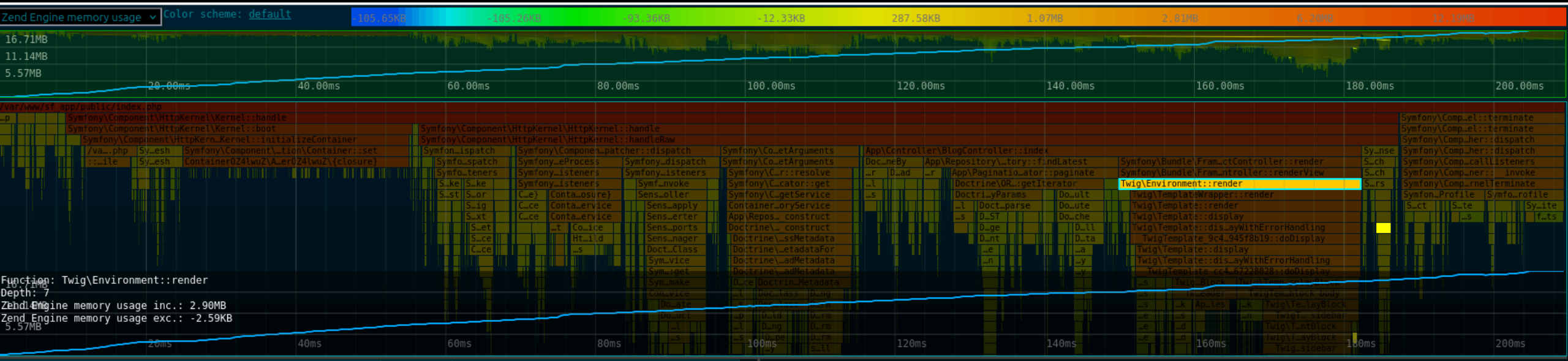
```php
$mapAndGetResult = enrichData(
  mapResultToObj(
    getResults()
 ) );
foreach (mapAndGetResult as $i => $row) {
  $data []= $row;
  if (0 === ($i % BATCH_SIZE)) {
    writePartial($data);
    $data = [];
} }
writePartial($data);
```

* *pseudo-code d'illustration*

Gestion de la mémoire - Outils

- Raw : **memory_get_usage()** & **memory_get_peak_usage(true)**
- Framework : **Profiler** & outils du framework
  - DebugBar fournie
  - **StopWatch** sous symfony
- **Profiler** externe : XDebug, PHP-SPX, Ecimer, Blackfire, Tideways, ...
- Outils de tracing : jaggear, opentelemetry, ...

PHP-SPX



Function	Called	Zend Engine memory usage			
		Percentage		Value	
		Inc.	Exc.	Inc.	Exc.
{closure}	8	0.00%	0.00%	712B	-200B
urlencode	2	0.00%	0.00%	104B	104B
unserialize@2	9	0.90%	0.33%	345.72KB	127.60KB

Erreur de ressources : API, DB, cache, ...

- Activer les modes d'exceptions/erreurs dans les libs
- Activer des timeouts cohérents
- Utiliser les requêtes parallèles (*curl_multi_ **)
- Utiliser les node readonly si multi-nœuds
- **Déconnexion** des ressources (exemple : fichier, db)
  - 🤔 worker ou job
- Utiliser des circuits-breakers ✋

Ressource : ORM/db connexion

Exemple, d'un cas courant, soucis d'ORM (ici doctrine)

```
Doctrine\ORM\ORMException
The EntityManager is closed.
```

⇒ Utiliser

```php
// vérifier si ouvert
$entityManager->isOpen();
// obtenir un nouvel entity manager
$managerRegistry->resetManager();
```

Arrive en cas de soucis, de problème de transaction, de problème de concurrence ...

⚡ Bien comprendre les libs utilisées

Gestion des erreurs

- try/catch :
  - ✅ **Throwable**
  - ❌ **Error** & **Exception**
- log, log, log
  - visibilité!
  - continue, skiped ou stop ?
  - id ?
  - raisons ?

```php
try
{
  // your execution
}
catch (Throwable $t)
{
  $this->logger
    ->warn(
      'Error or exeption: {message}.
           Skipped #{id}.',
      [
        'message' => $t->getMessage(),
        'id' => $item->id,
      ]
    );
  continue;
}
```

Tâche : en résumé

- Gestion de la mémoire
- Gestion des erreurs
- Gestion des ressources : ouverture/fermeture
- Log

Workers

Votre orchestrateur longue durée

Gestion du cycle de vie du worker

- Discuter avec vos sysadmin & devops
- VM/bare : systemd, supervisord, ...
- Container : Container, k8s, ...

🧠 Questions à se poser !

- Comment détecter un souci ?
- Relance automatique ?
- Directement ou après un délai ?
- Maximum de relance ?

Exécution : configuration de PHP

- **opcache**
  - par défaut (cli), inactif ⇒ 💡 actif
  - plus rapide (🥳 *JIT*)
- **extension** inutile! ⇒ 🚪
  - ~ gain de temps d'exécution
  - ~ évite la fuite mémoire
- 🪠🧹🪣 configuration de PHP

Exécution : Effet de bord de framework

- ✅ **APP_DEBUG=false**
- Désactivation de certains automatismes
  - *Tracing interne* de certain framework/lib
    - symfony 6.4 vs 7 😇
  - Exécution automatique, mais vous les gérés manuellement

Worker maison : Gestion des erreurs

Comme pour les tâches

- try/catch
- cycle de vie des ressources
- ...

Ici, c'est obligatoire !

Même un simple : *"Il y a eu une erreur, je m'arrête"*

Worker maison : Arrêt du worker ?

- Sortie du worker : 0 = OK, 1-255 = erreur

- Signaux POSIX (donc pas sous windows)
- **pcntl_signal** & **pcntl_async_signals(true)**
- Signaux : *SIGINT*, *SIGTERM*, *SIGQUIT*, ...
- Gestion
  - Finir la tâche courante
  - Log
  - Envoi de metric final
  - Clôture correcte (exemple : déconnexion, fclose, ...)

Worker maison : 🦺 mode d'exécution des tâches

- Exécution courante
  - 😐 *Une boucle et puis on est parti* 👷
- Utiliser les coroutines telles que **Fiber** ou **yield**
- Sous-processus : **pcntl_fork**
- ✅ Utiliser des libs tels que **react**, **Amp**, **Roadrunner**, …
- Processus indépendant : **shell_exec**, **exec**, …
- Appel externe : *FaaS*, …
- …

Worker maison : quelques astuces

- Système de queue : *ack / nack / lock / retry*
- 📝 Log de démarrage/arrêt
- 🧠 Limite
  - Temps d'exécution
  - Nombre d'éléments traité

Worker : en résumé

- Il en va de même pour les tâches
  - Gestion des erreurs
  - Gestion des ressources : ouverture/fermeture
- Gestion des signaux in/out
- Penser au cycle de vie
- *Les workers des frameworks sont bien*
- Log

Consolidation

Comprendre ce qui se passe

Gestionnaire d'erreurs

- *sentry*
- *rollbar*
- *newrelic*
- ...

⇒ permet de **corréler** & **agréger**

Comprendre ce qui se passe = observabilités

Logs 📝

- Identifiants de tâches
- Statut de ce qui se passe
- Démarrage et arrêt

Metric 📊

worker

- nombre d'éléments traité
- nombre par statut : erreur, ok, skipped, retry, ...

job : dépends de vote cas

statsd, **openmetric**, ...

Pensez aussi aux métriques métier

Trace 🕵

- **opentelemetry**, dynatrace, ... → outils variés
- ⚠️ longueur des traces : trace de la *tâche*, pas du *worker*

Gestion des tâches ingérables

- Dead letter queue (DLQ)
- Failed job queue

- ⇒ métriques
- ⇒ alerting
- ⇒ analyse régulière

Bonnes pratiques

- 1 job = 1 tâche précise
  - ne pas hésiter a subdivisé
    - petit ≃ rapide ≃ plus facile à rendre résilient
  - think unix
- 🚨 taille des messages
- Idempotence des jobs
  - ⇒ facilite le scaling horizontal (cf. autre conf)
  - ⇒ permet la réexécution

En résumé

- Gestion de la mémoire, des erreurs et des ressources
- Gestion des signaux in/out & cycle de vie
- Ne pas hésiter à subdiviser
- Log & metric

Merci

Avez-vous des questions ?

@Grummfy sur les réseaux sociaux

Les slides sont disponibles en ligne
https://grummfy.be/blog/546

Vos feedbacks !