VDM

Formation Développeur

17/06/2021

suez

Sommaire

1. Architecture générale

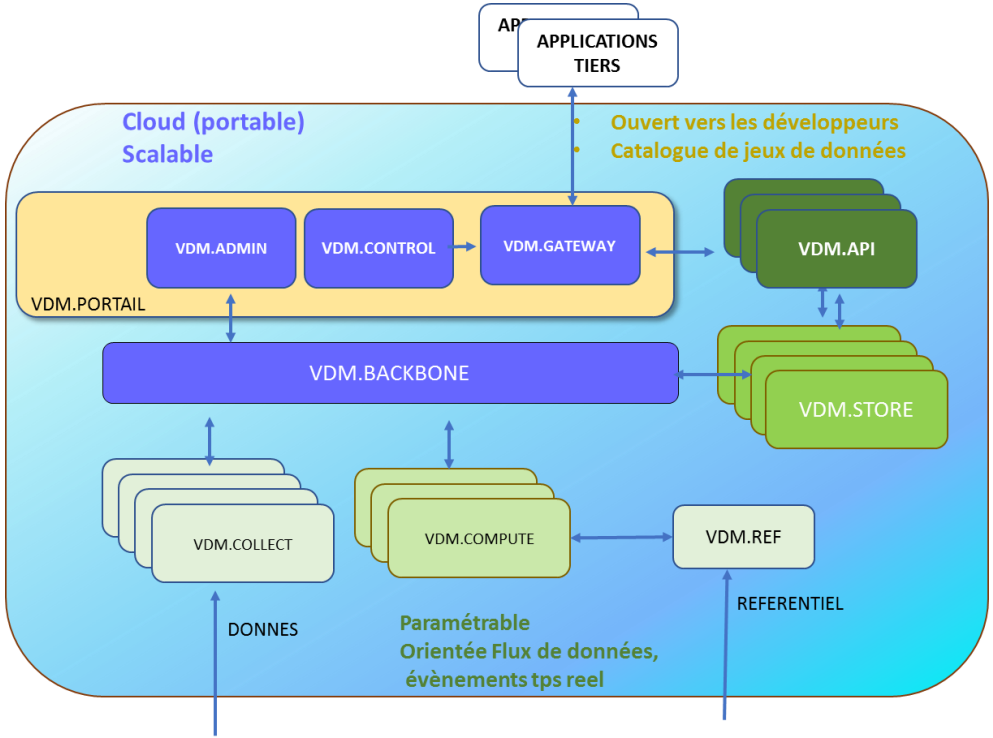
Concepts : architecture

La plateforme VDM est une plateforme d'exposition de données

Cette plateforme fournit les services suivant :

- **Agrégation** de données brutes hétérogènes – VDM.Collect
- **Routage** de message à travers la plateforme – VDM.Backbone
- **Enrichissement** des données brutes : qualification, metadata via un référentiel , … – VDM.Compute
- **Historisation** des données brutes et enrichies – VDM.Store
- **Exposition** de données via des API. VDM.API
- **Gestion des droits**, gestion des profils utilisateurs, de la sécurité, du catalogue de données via un portail internet : VMD.Portail dit « **portail Open API** »

Cette plateforme est hébergeable en cloud privé, public, et fournit des services en mode CaaS apportant une scalabilité par conception



Concepts : OpenSource

2 organisations Github :

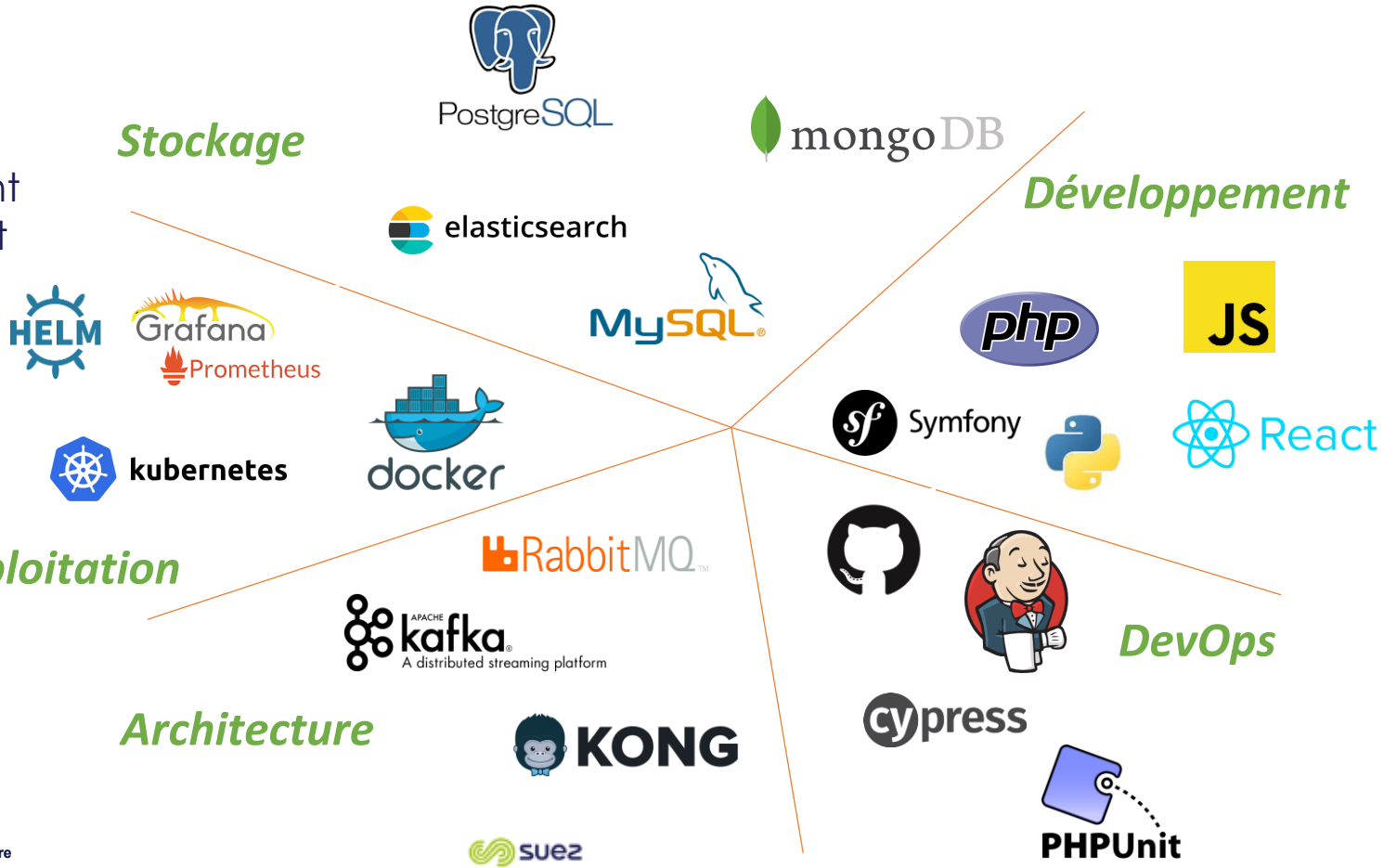
- Suezenv : https://github.com/Suezenv pour nos repos privés ou nos forks de repos OpenSource (bugfixes,enhancement,…)
- 3slab : https://github.com/3slab pour nos contributions OpenSource

Nos contributions OpenSource :

- Bundles VDM : VdmLibraryBundle, VdmPrometheusBundle, …
- Repository de charts helm : helm-charts
- Tutoriel déploiement monitoring prometheus : aks-prometheus-stack
- …

Technologies : Liste

Les modules sélectionnés sont éprouvés et font l'objet d'une communauté active



suez

Technologies : ETL ?

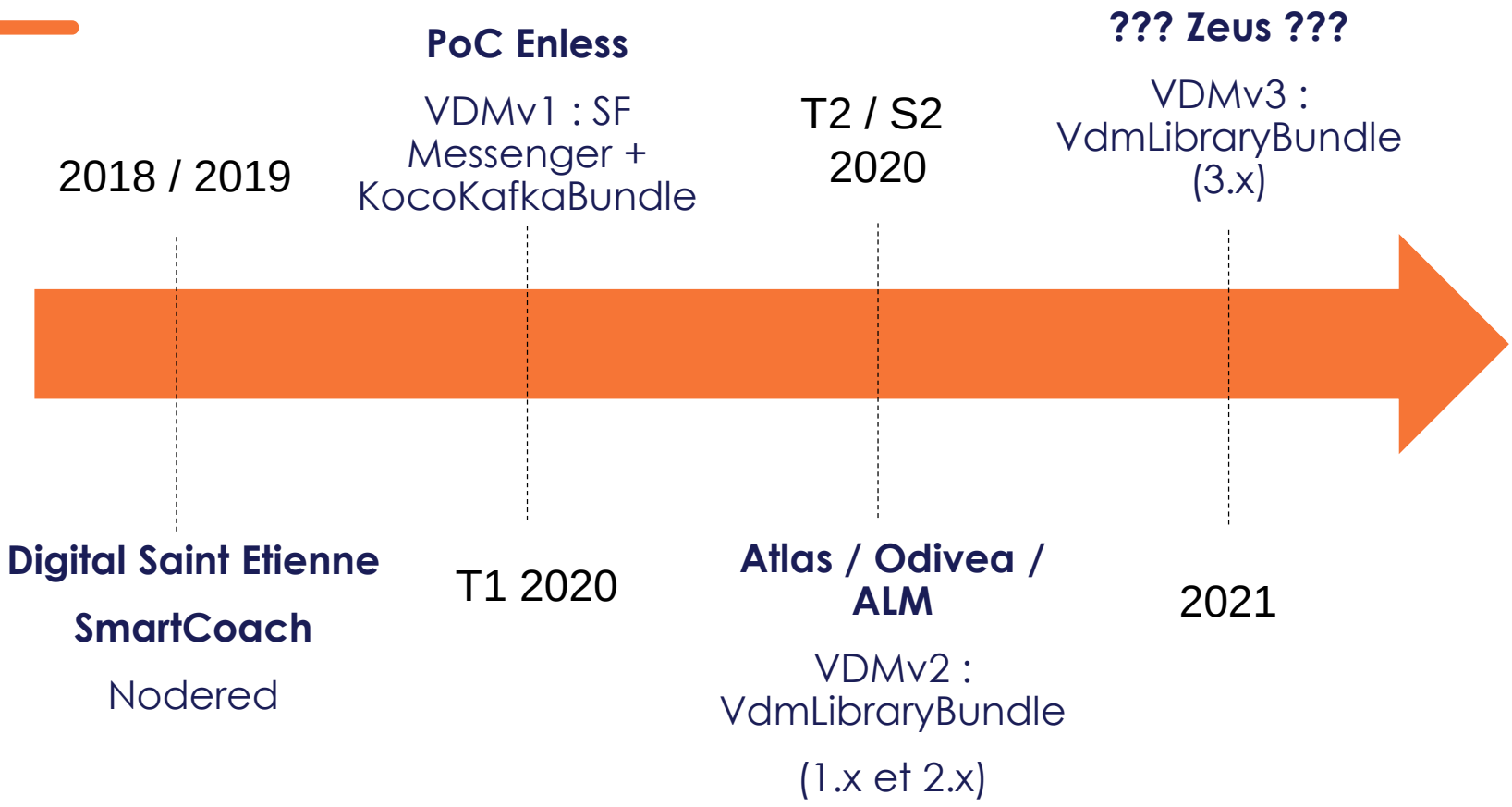
Pourquoi pas une solution spécialisée d'ETL OpenSource ?

Apache Nifi, Talend, Pentaho Kettle Data Integration, Apache Flink, …

- Intégration d'une nouvelle technologie :
  - Montée en compétence des développeurs
  - Montée en compétence des architectes
  - Montée en compétences des ops
  - Montée e compétences de l'infogérance
- Peu ou pas de « configuration as code » : dataflow « codé » en mode graphique
- Richesse des connecteurs : manquants, obsolètes, partiels, …

Conclusion : Utilisation d'une technologie maîtrisée, éprouvée et disposant d'un composant dédié à la communication avec des brokers => **SF Messenger**

Historique : Timeline



Historique : Pourquoi ces changements ?

Nodered :

- Avantage : RAD : Rapid Application Development (NoCode)
- Inconvénient : pas de maîtrise sur les flux (logs, métriques, gestion des erreurs, stabilité)

SF messenger / KocoKafkaBundle :

- Avantage : PHP/SF maîtrisé
- Inconvénient : Pas de standardisation

Historique : Pourquoi ces changements ?

VdmLibraryBundle v1 et v2 :

- Avantage : PHP/SF maîtrisé, standardisation des flux, monitoring intégré
- Inconvénient :
  - un projet SF par nœud du flux (1 collect 2 computes 1 store = 4 projets SF)
  - Plusieurs approches : projets SF séparés, bundle "common" séparé, bundle "common" monté par volumes
  - Maintenance des projets difficiles, environnements de développement complexes

VdmLibraryBundle v3 :

- Avantage : PHP/SF maîtrisé, SF messenger standard sans intrusion, plus de magie
- Inconvénient : A voir dans le temps ☺

2.

Messenger

Messenger : Concepts

RTFM ☺ :

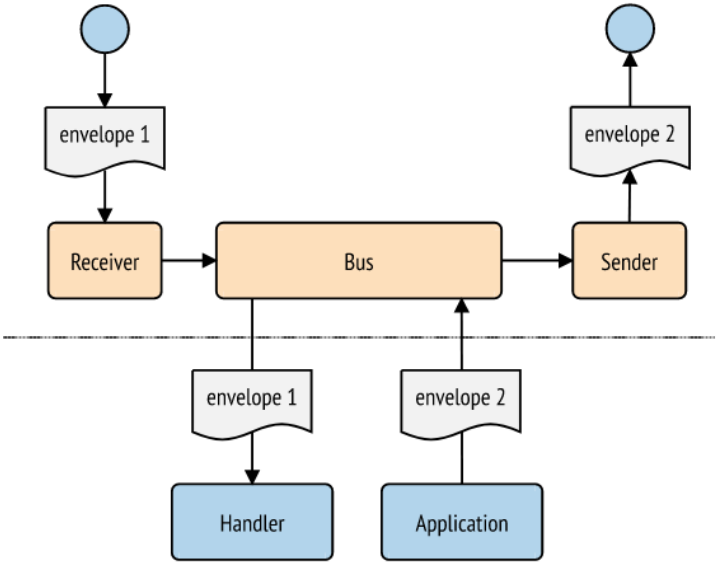
- Composant : https://symfony.com/doc/current/components/messenger.html
- Bundle SF : https://symfony.com/doc/current/messenger.html

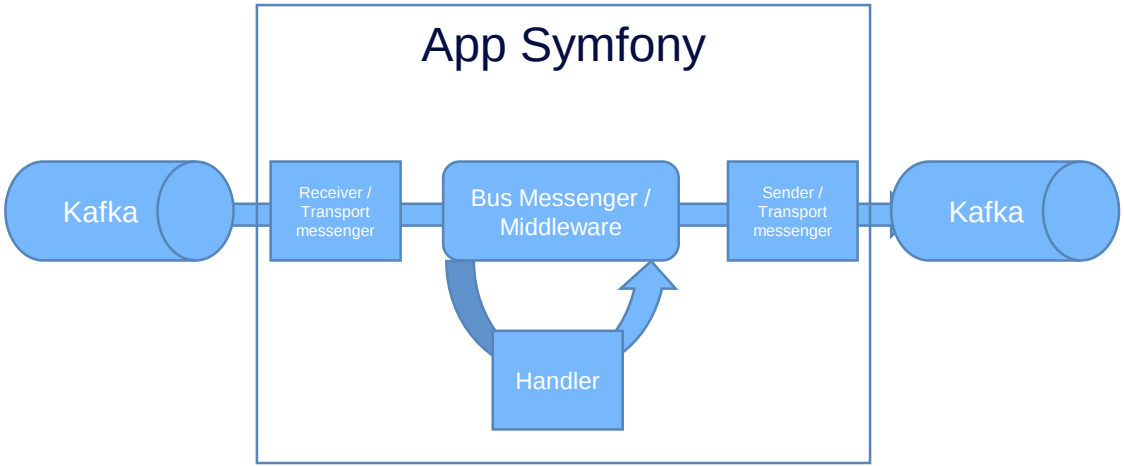
Ce composant du framework Symfony propose un **bus applicatif** de message capable de **recevoir, de traiter ou de transmettre des messages** à travers des transports (brokers de message) le plus souvent pour du traitement asynchrone.

Note : pour ceux qui connaissent, c'est l'équivalent PHP de Celery en Python

Messenger : Concepts

LE MESSAGE BUS DE MESSENGER EST UN BUS APPLICATIF !!!!!!





Messenger : dispatch message

Que se passe-t-il lorsqu'on dispatch un message dans un bus ?

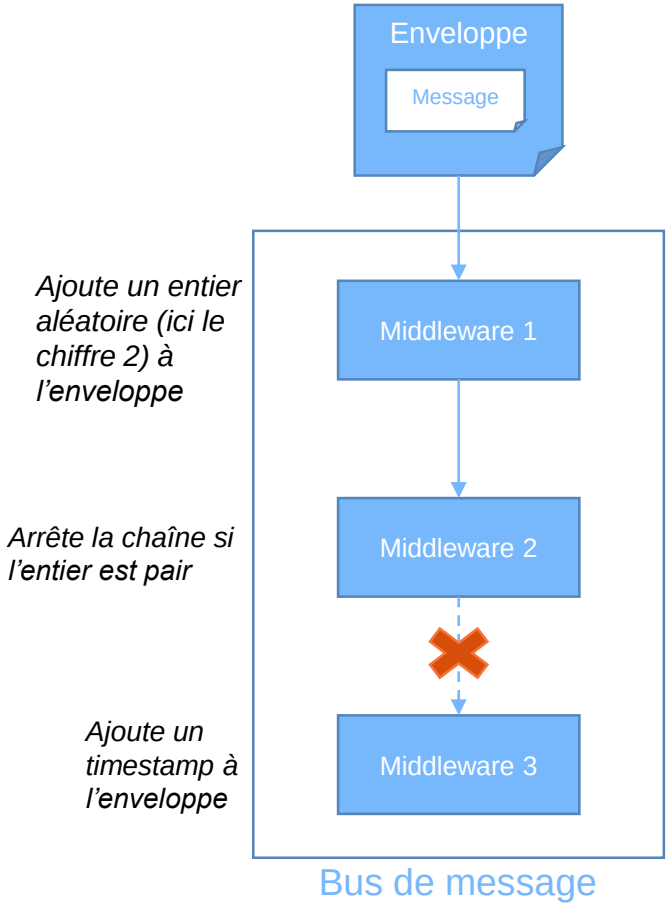
```
/** @var \Symfony\Component\Messenger\MessageBusInterface
$bus */

/** @var \App\Message\MyMessage $message */

$bus->dispatch($message);
```

Le bus de message effectue les opérations suivantes :

1. Wrap l'objet Message dans un objet Symfony\Component\Messenger\Envelope
2. Fait passer cet objet enveloppe parmi la collection de Middleware enregistré sur le bus



Messenger : 3 concepts issus du dispatch

- **Envelope** : spécifique à messenger. Permet à ce composant d'ajouter des informations au message sans modifier ce dernier

Note : au dispatch, tu peux laisser messenger créer l'enveloppe ou la créer toi même

- **Envelope Stamp** : une métadonnée ajoutée à l'enveloppe par un middleware (sans modifier le message)

- **Middleware** : une couche d'exécution du bus qui peut lire/modifier l'enveloppe et/ou le message, effectuer des actions, transmettre l'enveloppe (et donc le message) au middleware suivant ou arrêter la chaîne d'exécution

Messenger : Exemple d’utilisation Envelope et Stamps

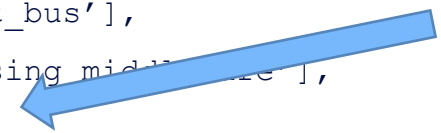
```
$envelope = $envelope->with(new SerializerStamp([
    'groups' => ['my_serialization_groups'],
]);

$bus->dispatch(new SmsNotification('...'), [
    new DelayStamp(5000),
]);

$bus->dispatch(new Envelope(new SmsNotification('...'), [
    new DelayStamp(5000),
]));
```

Messenger : quels middlewares dans le bus default ?

```
$defaultMiddleware = [
    'before' => [
        ['id' => 'add_bus_name_stamp_middleware'],
        ['id' => 'reject_redelivered_message_middleware'],
        ['id' => 'dispatch_after_current_bus'],
        ['id' => 'failed_message_processing_mid
    ],
    'after' => [
        ['id' => 'send_message'],
        ['id' => 'handle_message'],
    ],
];
```



Les middlewares déclarés manuellement (dans la conf Symfony) sont chargés entre before et after

```
framework:
    messenger:
        buses:
            messenger.bus.default:
                default_middleware: false
                middleware:
                    - 'App\Middleware\MyMiddleware'
                    - 'App\Middleware\AnotherMiddleware'
```

https://github.com/symfony/framework-bundle/blob/120e80e882debd7e705d53a3b054e1a0fae91fbc/DependencyInjection/FrameworkExtension.php#L1905

Messenger : quels middlewares dans le bus default ?

- **send_message** : en fonction du routing du message, si ce n'est pas un message reçu depuis un transport, il est transmis au « sender » du transport concerné par le traitement de ce message et la chaîne s'arrête

```
# config/packages/messenger.yaml

framework:

    messenger:

        transports:

            async: "%env(MESSENGER_TRANSPORT_DSN)%"


        routing:

            # async is whatever name you gave your transport above

            'App\Message\SmsNotification': async
```

Messenger : quels middlewares dans le bus default ?

- **handle_message** : cherche les classes MessageHandler capables de traiter le message contenu dans l'enveloppe, les trie par ordre de priorité et __invoke leur méthode sur le message

```
use Symfony\Component\Messenger\Handler\MessageHandlerInterface;


class NewUserWelcomeEmailHandler implements MessageHandlerInterface
{
    public function __invoke(NewUserWelcomeEmailMessage $welcomeEmail)
    {
        $user = $this->userRepository->find($welcomeEmail->getUserId());


        // ... send an email!
    }
}
```

Messenger : Exercice 1

Dispatch / Synchronous handler

https://github.com/3slab/vdm-training/01_messenger_sync

A retenir de cet exercice : le mode synchrone de messenger est inutile et devrait être remplacé si besoin par l'utilisation d'événement avec EventDispatcher.

Messenger : que se passe-t-il quand je lance messenger:consume ?

1. La commande SF instancie un objetSymfony\Component\Messenger\Worker et exécute sa méthode run()
   https://github.com/symfony/messenger/blob/5.3/Worker.php

2. La méthode lance une boucle infinie `while (false === $this->shouldStop)`

3. Pour chaque receiver/transport concerné par l'éxecution, elle appelle la méthode get() qui retourne un iterable (array, generator yield, …) d'enveloppes

4. Pour chacune de ces enveloppes, elle appelle la méthode handleMessage() du worker lui-même

5. Si rien n'empêche « d'handler » le message, elle dispatche dans le bus par défaut le message dans une nouvelle enveloppe avec en plus le ReceivedStamp et le ConsumedByWorkerStamp.

Messenger : que se passe-t-il quand je lance messenger:consume ?

6. Le dispatch est entouré d'un try/catch. Si une exception se produit, la méthode reject() du receiver/transport est appelée

7. Si le dispatch s'exécute correctement (soit tous les middlewares concernés par le traitement du message et de son enveloppe s'exécutent avec succès), la méthode ack() du receiver/transport est appelée.

8. La méthode handleMessage a fini de s'exécuter pour ce message, elle traite les autres messages du receiver/transport courant, puis ceux des potentiels autre receiver/transport impliqué dans l'exécution en cours et on revient finalement au début de la boucle infinie pour repartir au point (2) et recommencer.

Messenger : Exercice 2

Dispatch / Asynchronous handler

https://github.com/3slab/vdm-training/02_messenger_async

A retenir de cet exercice : attention à bien schématiser vos flux : Message, MessageHandler, Transport

Messenger : Pourquoi n’avons-nous pas une boucle infinie ?

Depuis un contrôleur, je dispatche un message de type MyNotificationMessage.

J’ai la configuration suivante dans messenger.yaml :

```
framework:
    messenger:
        transports:
            notification-producer: "%env(MESSENGER_TRANSPORT_DSN)%"
            notification-consumer: "%env(MESSENGER_TRANSPORT_DSN)%"
        routing:
            'App\Message\MyNotificationMessage': notification-producer
```

Je consomme mes messages MyNotificationMessage à l’aide de la commande :

```
bin/console messenger:consume notification-consumer
```

Messenger : Pourquoi n'avons-nous pas une boucle infinie ?

Rappel simplifié de ce qui se passe durant la commande `messenger:consume` :
https://github.com/symfony/messenger/blob/5.3/Worker.php

```
public function run()
{
    while (false === $this->shouldStop) {
        $envelopes = $receiver->get();
        foreach ($envelopes as $envelope) {
            $this->handleMessage($envelope);
        }
    }
}

public function handleMessage($envelope)
{
    $envelope = $this->bus->dispatch($envelope->with(new ReceivedStamp($transportName)));
}
```

Middleware send_message avant handle_message
=>
l'enveloppe avec le message MyNotificationMessage devrait théoriquement être renvoyé dans le transport notification-producer

Messenger : Pourquoi n'avons-nous pas une boucle infinie ?

Regardons le middleware `send_message`

https://github.com/symfony/messenger/blob/c61a3fe0a0de99e17390d7f23e0c0730c88ee558/Middleware/SendMessageMiddleware.php

```
public function handle(Envelope $envelope, StackInterface $stack): Envelope
{
    …
    if ($envelope->all(ReceivedStamp::class)) {
        $this->logger->info('Received message {class}', $context);
    } else {
        foreach ($this->sendersLocator->getSenders($envelope) as $alias => $sender) {
            …
            $envelope = $sender->send($envelope->with(new SentStamp(\get_class($sender), \is_string($alias) ? $alias : null)));
        }
    }
    …
}
```

Les enveloppes avec un stamp ReceivedStamp sont ignorées par ce middleware !!!!!!!!

Messenger : Exercice 3

Asynchronous dispatch / handler avec rabbitmq en mode fanout

https://github.com/3slab/vdm-training/03_messenger_rabbitmq_fanout

A retenir de cet exercice : le transport duquel on consomme n'a pas (ou peu) d'incidence sur les handlers de message exécutés.

Note : l'implémentation SF messenger du mode consommation rabbitmq ne fait pas apparaître de consumer dans l'UI d'admin rabbitmq

Messenger : comprendre les transports

Interface d'un TransportFactory :

```
interface TransportFactoryInterface
{
    public function createTransport(string $dsn, array $options, SerializerInterface $serializer): TransportInterface;

    public function supports(string $dsn, array $options): bool;
}
```

Interface d'un Transport :

Utilisé uniquement dans messenger:consumer (get => consome, ack => handled sans erreur, reject => exception pendant hander)

```
interface TransportInterface extends ReceiverInterface, SenderInterface
{
    public function get(): iterable;

    public function ack(Envelope $envelope): void;

    public function reject(Envelope $envelope): void;

    public function send(Envelope $envelope): Envelope;
}
```

Utilisé uniquement dans bus dispatch

Messenger : comprendre les transports

Exemple de transport factory :
https://github.com/symfony/amqp-messenger/blob/5.3/Transport/AmqpTransportFactory.php

Exemple de transport :
https://github.com/symfony/amqp-messenger/blob/5.3/Transport/AmqpTransport.php

Au besoin d'un sender (send middleware) / receiver (Worker::run dans messenger:consume), messenger appelle la factory du transport correspondant (identifié grâce à la méthode support) qui retourne un objet Transport capable de communiquer avec le broker via ses fonctions send, get, ack, reject

Noter le pattern dans le transport de création d'un objet dédié aux méthodes de « réception/consommation » et d'envoi « dispatch »

Messenger : Middlewares à activer/désactiver

1. Désactiver le middleware de retry en cas d'erreur pour quasiment tous les transports (sauf rabbitmq et encore, avec kubernetes, on préfère avoir un pod consumer en erreur et être prévenu par le monitoring)

https://symfony.com/doc/current/messenger.html#retries-failures

```
framework:
    messenger:
        consumer:
            dsn: "%env(MESSENGER_TRANSPORT_DSN)%"
            retry_strategy:
                max_retries: 0
```

Messenger : Middlewares à activer/désactiver

2. Ajouter des middlewares doctrine pour les « long-running processes » (comme des consumers avec peu de traffic) pour redéclencher automatiquement des connections au SGDB et éviter les timeouts

https://symfony.com/doc/current/messenger.html#middleware-for-doctrine

```
framework:
    messenger:
buses:
        messenger.bus.default:
            middleware:
                - doctrine_ping_connection
                - doctrine_close_connection
```

3.
VDM

VDM : kesako ?

VDM reprend les concepts d'un ETL (Extract = Collect/Get, Transform = Compute, Load = Store) sous la forme de micro services PHP s'appuyant sur le Framework Symfony et son composant Messenger. Chaque étape du processus ETL est exécutée dans un container dédié.

- AVANTAGE :
- Pod en erreur détecté automatiquement par K8S
- Pod en erreur redémarré automatiquement par K8S
- Langage de programmation et Framework maîtrisé par les équipes
- Robustesse du traitement (se prémunir contre la perte des données, fine-tuner la configuration, …)

- INCONVÉNIENT :
- Chaque flux est « codé » (dérives d'un projet à l'autre, CAPEX important, …) => *Solution : Passer par des formats « pivots » pour pouvoir réutiliser certains nœuds du flux ETL*

VDM : historique des versions

V1 : SF Messenger + KocoBundle

+ Pas de surcouche VDM

- Pas de monitoring

- Pas de standardisation

V2 : VdmLibraryBundle 1.x ou 2.x + SF Messenger + transports (KocoBundle,

+ Monitoring

+ Standardisation

- Un projet SF par processus VDM (core-bundle partagé dans le code ou dans un repo dédié, multiples projets SFs, …)

V3 : VdmLibraryBundle 3.x + SF Messenger + transports (KocoBundle, VdmLibrary*TransportBundle, amqp-messenger, …)

+ Monitoring

+ Standardisation

+ Plusieurs processus messenger dans un même projet

+ Plus de magie, on code du SF messenger standard (Transport, MessageHandler, Message)

VDM : v3 une simple surcouche à SF messenger

VDM est décomposé en plusieurs bundles :

- **VDMLIBRARYBUNDLE :**

https://github.com/3slab/VdmLibraryBundle

Le bundle de base qui étend les capacités de SF messenger en ajoutant les modèles de message standards à étendre, des listeners sur des events SF pour gérer les erreurs, collecter les métriques de monitoring, …

- **VDMLIBRARY*TRANSPORTBUNDLE :**

Des bundles pour enrichir les transports messenger pour nos besoins sur les sources et destination supportées par nos flux de données. A date (23/06), nous avons :

- HTTP (source) : https://github.com/3slab/VdmLibraryHttpTransportBundle
- Doctrine ORM et ODM (destination) : https://github.com/3slab/VdmLibraryDoctrineTransportBundle
- Flysystem (source WIP) : https://github.com/3slab/VdmLibraryFlysystemTransportBundle

VDM : v3 des bundles complémentaires

VDM propose également quelques bundles complémentaires pour répondre à certains cas d'usage et fonctionnalités :

- **VDMPROMETHEUSBUNDLE :**

https://github.com/3slab/VdmPrometheusBundle

Collecte de métriques sur des contrôleurs SF et exposition de ces dernières au format Prometheus (utile pour des portails ou API)

- **VDMHEALTHCHECKBUNDLE :**

https://github.com/3slab/VdmHealthcheckBundle

Ajoute 2 endpoints liveness et readiness pour paramétrer des sondes de santé utiles pour du blackblox uptime ou les probes K8S

- **VDMHEALTHCHECKBUNDLE :**

https://github.com/3slab/VdmVersionBundle

Ajoute 1 endpoint /version pour standardiser l'exposition d'une version du code d'une API ou d'un portail.

VDM : construire un flux

VdmLibraryBundle (v3.x) est non intrusif. Toutes les fonctionnalités qu'il propose se basent sur les concepts et les hooks d'extension (event, registry de services, …) fournis par SF messenger.

Donc, quand on construit un nœud VDM d'un dataflow (collect, compute, store), on définit toujours :

- Transport source
- Message source (et plus particulièrement sa classe PHP)
- MessageHandler
- Message produit (et plus particulièrement sa classe PHP)
- Transport destination

Néanmoins, pour certaines des fonctionnalités VDM, il est recommandé (si ce n'est imposé) de construire des Message sources ou destination qui héritent de Vdm\Bundle\LibraryBundle\Model\Message

https://github.com/3slab/VdmLibraryBundle/blob/3.x-dev/Model/Message.php

VDM : Model Message

```
{
    "payload": int / string / float / bool, array, object,
    "metadatas": [],
    "traces": []
}
```

- payload : le message « métier » transféré de nœud en nœud dans le flux de données
- Metadatas : un tableau d'objets Vdm\Bundle\LibraryBundle\Model\Metadata représentant des couples clé/valeur utiles pour transmettre de l'information aux nœuds suivants du flux de données
- Traces : un tableau d'objets Vdm\Bundle\LibraryBundle\Model\Trace représentant les timestamps d'entrée et de sortie d'un noeud VDM

Note : seule la payload est persistée en base de données lors d'une écriture en fin d'un flux de données dans un noeud de type stocker/store

VDM : exercice 4

Asynchronous dispatch avec VDM

https://github.com/3slab/vdm-training/04_vdm_async

A retenir de cet exercice : rien ne change dans l'approche VDM d'un flux qu'un développement sur messenger sans le bundle VdmLibraryBundle

VDM : les fonctionnalités : exit on error

config/packages/vdm_library.yaml

```
vdm_library:
    stop_on_error: true
```

Arrête le processus de consommation en cas d'erreur durant le traitement (MessageHandler) d'un message.

Utile pour faire qu'on container k8s s'arrête en cas d'erreur lors du traitement d'un message et soit détecté par le monitoring.

Note : ajouté en mode feature flag en v3 car messenger fournit maintenant en natif une option --failure-limit (non testée)

Use the --failure-limit=X option to stop the worker when the given number of failed messages is reached

VDM : les fonctionnalités : dev/debug

config/packages/vdm_library.yaml

```
vdm_library:
    print_msg: true
```

Effectue un dump() du message en sortie du transport receiver et en entrée du transport sender.

Utile en phase de dev pour déboguer

Note : peut être activé en production mais uniquement pour des nœuds VDM travaillant sur des messages de petite taille

VDM : les fonctionnalités : dev/debug

config/packages/messenger.yaml

```
framework:
    messenger:
        transports:
            wiki-collect-get:
              dsn: 'vdm+local://%kernel.project_dir%/local/wiki-collect-get.json?encodeBody=json'
              retry_strategy:
                max_retries: 0

            wiki-collect-produce:
              dsn: 'vdm+local://%kernel.project_dir%/local/wiki-collect-produce.json?encodeBody=json'
              retry_strategy:
                max_retries: 0
```

VDM : les fonctionnalités : dev/debug

.gitignore

```
/local/*-failed.json
/local/*-produce.json
```

%kernel.project_dir%/local/wiki-collect-get.json?encodeBody=json


```
{
    "body": {
        "payload": {
            "type": "edit",
            "ns": 0,
            "title": "Momodu Maligie",
            "pageid": 38268194,
            "revid": 1027536230,
            "old_revid": 1027533232,
            "rcid": 1391940415,
            "timestamp": "2021-06-08T13:54:42Z"
        },
        "metadatas": [],
        "traces": []
    },
   "headers": {
        "type": "App\\WikiRecentChanges\\Collect\\Message\\WikipediaMessage",
        "Content-Type": "application\/json"
    }
}
```

VDM : les fonctionnalités : dev/debug

Le transport vdm+local permet de construire manuellement un message et ainsi de développer son MessageHandler sans interaction avec le broker.

Les points importants sont :

- Le format JSON attendu du message
- Le paramètre encodeBody=json qui permet d'avoir une payload décodée dans le json au lieu d'une payload encodé

```
"body": {
    "payload": {
        "type": "edit",
        "ns": 0
    }
}
```

Au lieu de

```
"body": "{\"payload\":"{\"type\":\"edit\",\"ns\":0}\"}"
```

- En sortie, il écrit dans le fichier définit en DSN ce qui aurait été envoyé au transport de destination

VDM : exercice 5

Dev tools avec VDM

https://github.com/3slab/vdm-training/05_vdm_devtools

A retenir de cet exercice : l'utilisation des outils d'aide au développement et surtout du transport local permettent de développer un nœud d'un flux VDM sans interaction avec le broker ou les nœuds amonts

MAIS IL EST IMPORTANT DE BIEN CONNAÎTRE LES FORMATS DES MESSAGES EN ENTREE

VDM : les fonctionnalités : monitoring

Le monitoring est la fonctionnalité principale introduite par VDM. Elle est non-intrusive. Des listeners positionnés sur les différents événements messenger collectent des métriques et les poussent vers le système de monitoring. Du fait que la plupart des processus sont des commandes n'exposant pas d'API, on est sur un mécanisme de push des métriques vers une gateway ou un système de monitoring.

Les métriques collectées sont :

- Nombre de messages consommés
- Nombre de messages traités
- Nombre de messages produits
- Durée de traitement des messages
- Peak memory usage durant le traitement des messages
- Statut du worker 1/0 (redondant avec les métriques standards k8s. Privilégier ces dernières)

VDM : les fonctionnalités : monitoring

config/packages/messenger.yaml

```
vdm_library:
    app_name: itv-collect
    monitoring:
        type: null
        options: {}
```

Le type permet de définir la destination des métriques. Est supporté :

- null : métriques envoyées à un NullStorage (dans les limbes ….)
- statsd : métriques envoyées à un processus statsd_exporter pour les cacher et exposer un endpoint compatible prometheus. Paramétrer l'ip et el port de ce processus dans options

app_name ajoute un tag aux métriques générés par le processus en cours d'exécution avec la valeur app=<app_name value>.

Vu qu'un projet Symfony peut contenir plusieurs consumers, stockers, collecteurs. La bonne pratique est de positionner une variable d'environnement sur ce paramètre et de changer sa valeur au déploiement en fonction du processus qui est exécuté dans le pod.

VDM : exercice 6

Monitoring avec VDM

https://github.com/3slab/vdm-training/06_vdm_monitoring

A retenir de cet exercice : le monitoring est non intrusif et n'impacte pas le code messenger standard

VDM : Transports VDM

VDM fournit des transports qui enrichissent les transports natifs SF messenger (amqp) et ceux fournis par la communauté (KocoKafkaBundle).

Les transports fournis par VDM sont souvent liés à des sources ou des destinations où l'implémentation ou le protocole va changer d'un projet à un autre :

Exemple : je collecte depuis une source HTTP : une source va demander un appel login oauth2 avec gestion d'un token de rafraichissement puis passer le token d'autorisation dans un header Authorization Bearer. Une autre source sera peut être avec la génération d'un hashmac avec partage de secret ou tout simplement une clé d'api dans le header

Le point important à retenir est la notion d'executor.

VDM : Executor VDM

Un executor est un service injecté par VDM dans le transport pour personnaliser son fonctionnement. Chaque transport fournit une implémentation par défaut que le développeur peut remplacer par la sienne.

https://github.com/3slab/VdmLibraryHttpTransportBundle/blob/3.x-dev/Transport/HttpTransportFactory.php

https://github.com/3slab/VdmLibraryHttpTransportBundle/blob/3.x-dev/Transport/HttpTransport.php

https://github.com/3slab/VdmLibraryHttpTransportBundle/blob/3.x-dev/Executor/DefaultHttpExecutor.php

https://github.com/3slab/VdmLibraryHttpTransportBundle/blob/3.x-dev/Executor/AbstractHttpExecutor.php

```
framework:
    messenger:
        transports:
            wiki-collect-get:
              dsn: "https://myurl"
              retry_strategy:
                max_retries: 0
              options:
                http_executor: App\Executor\MyCustomHttpExecutor
```

VDM : collect vs consume

VDM fournit 2 commandes SF :

- vdm:consume
- vdm:collect

Qui sont de simples wrappers autour de la commande messenger:consume

vdm:consume : est identique à messenger:consume (juste pour l'appeler vdm ^^)

vdm:collect : est quasi identique à messenger:consume sauf qu'elle ne supporte que des sources messenger (receiver) qui ne sont pas des brokers soit à date http (et à venir flysystem). Il lève une exception si la source est du kafka ou rabbitmq.

Pourquoi ? J'ai eu des plaintes comme quoi ce n'était pas clair de « consume » une source http (ils se reconnaîtront ☺)

VDM : exercice 7

Nouveau projet VDM from scratch

https://github.com/3slab/vdm-training/07_vdm_new_project

A retenir de cet exercice : un flux VDM complet s'écrit en seulement quelques heures si on se pose correctement en amont pour définir les sources, traitements et destinations

Roadmap 3.x

- ~~Dépréciation VdmLibraryFtpTransportBundle pour un VdmLibraryFlysystemTransportBundle (S3, Blob, FTP, SFTP, …)~~
- Mise à jour des README des différents Bundle pour la v3.x
- Validation des messages en réception et en envoi
- Tests unitaires manquants : VdmLibraryDoctrineTransportBundle ~~et VdmLibraryHttpTransportBundle~~
- ~~Dashboard Grafana VDM dataflow basé sur les métriques VdmLibraryBundle~~
- Alert rules Prometheus basé sur les métriques VdmLibraryBundle
- Supporter le mode Sender sur le HttpTransport
- Supporter le mode Sender sur le FlysystemTransport
- Ajouter les behavior retry et monitoring sur FlysystemTransport
- Supporter Prometheus_push_gateway comme destination des métriques de monitoring VDM

- Faire un site projet public pour commencer à construire une communauté
- Peut être présenter ce projet dans un salon ? Une conférence PHP, Symfony, Data ?

Conclusion

Dire « VDM c'est trop compliqué »,

c'est comme dire « Symfony c'est trop compliqué » vu que c'est du Symfony Messenger.

Citation Jonathan B ☺

Si vous ne devez retenir qu'une chose : ma règle personnelle des 3 tiers pour un développeur (que je ne m'applique pas toujours ^^)

Quand je développe une fonctionnalité :

1/3 : compréhension : lecture du ticket / spécification, questions aux POs, aux architectes, …

1/3 : réflexion ← C'est ici que vous pensez à votre flux et son organisation : transports (receiver, sender), classes Message, classes MessageHandler

1/3 : développement (+ tests et c'est facile avec VDM : juste un MessageHandler et parfois des Executors ☺)


suez