

How to migrate to **Symfony Scheduler**

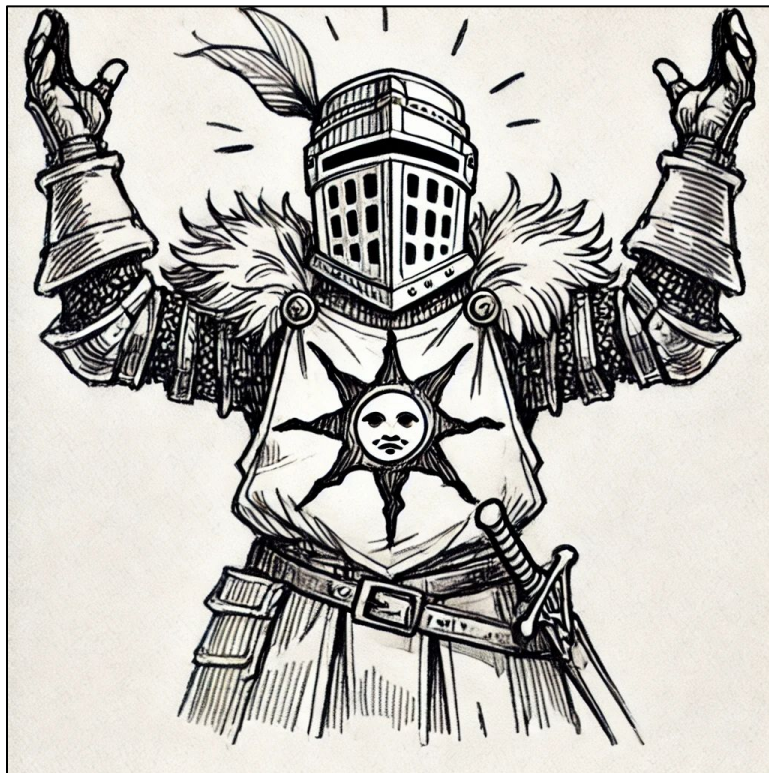


SymfonyCon
VIENNA 2024

by **Viktor Pikaev**

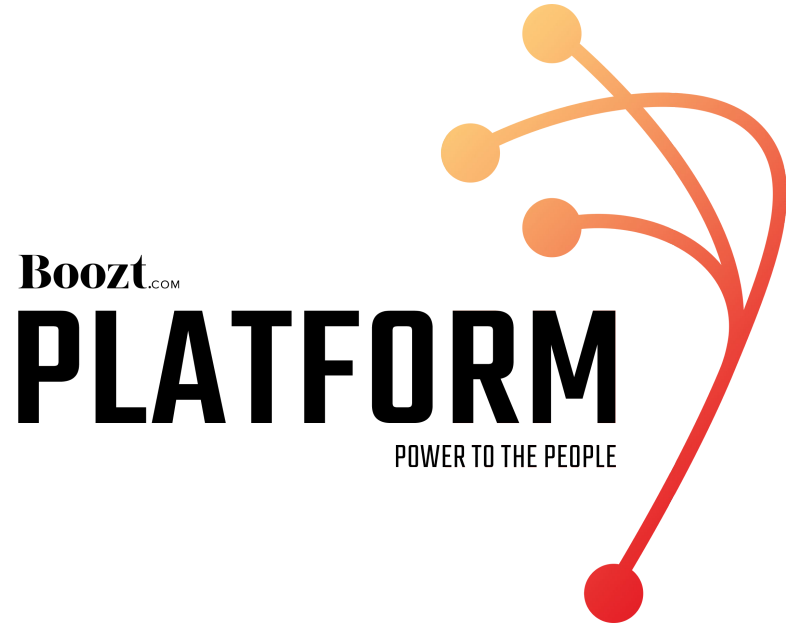
Viktor Pikaev

- Live in **Malmö**, Sweden 🇸🇪
- **16+ years** of experience
- Senior **Backend Developer** at **Boozt**.com
- A big fan of **X** and **®**
- **haru-atari.com**



Boozt Platform

- Boozt is a Nordic **technology** company
- More than **200 developers and engineers**
- More than **40 tailor-made systems**
(mostly using Symfony)
- Platform People post on LinkedIn
with **#boozttech**



Agenda

- Reasons
- Symfony Messenger
- Symfony Scheduler
- Pitfalls
- Migration process
(personal experience)
- Conclusion



Reasons

- Requires a **console command** to run
 - Sometimes **not flexible enough**
 - The schedule is a **part of business logic**, but it is stored **outside the project**
-

Symfony Scheduler

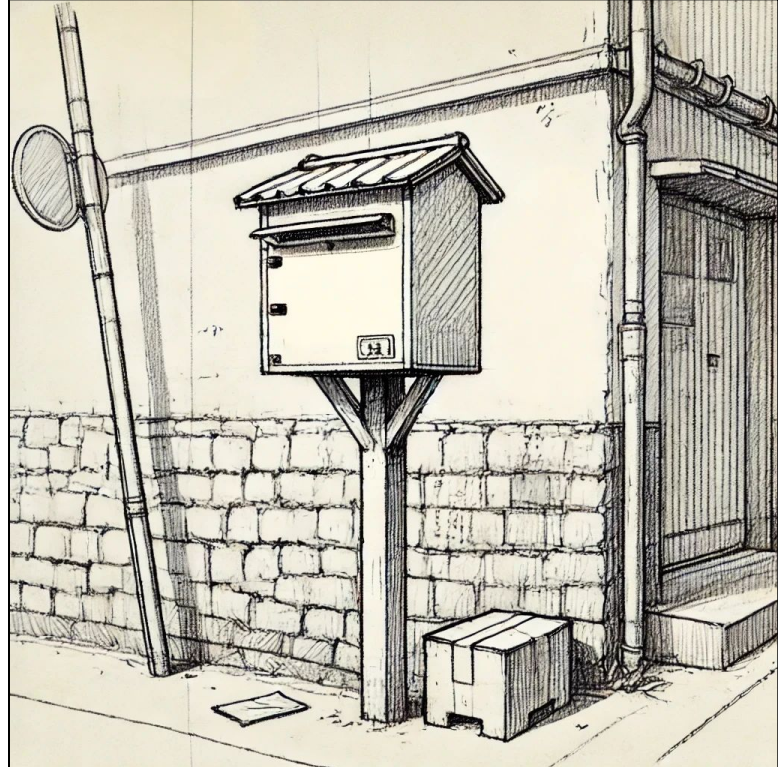
- Does not require a console command to run
 - Stores schedules as code within the project
 - Very simple if necessary
 - Very flexible if necessary
-

Symfony Scheduler

Symfony Scheduler can work as a standalone solution.

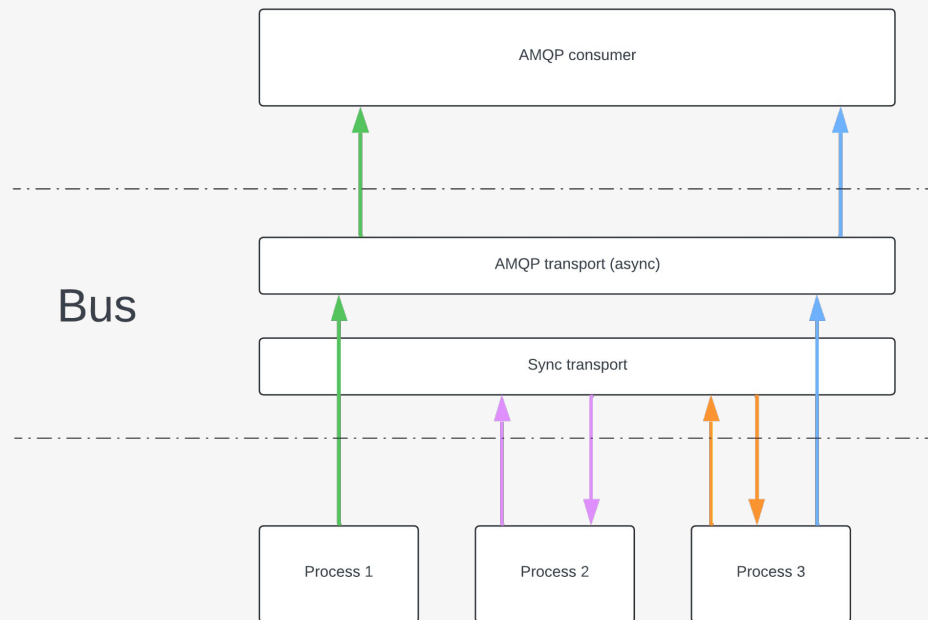
But the most common case is to use it in combination with **Symfony Messenger**.

Symfony Messenger



Symfony Messenger

Messenger provides a message bus with the ability to send messages and then handle them immediately in your application or send them through transports (e.g. queues) to be handled later.



Messages

- Represents data transferred through the bus
- Serializable DTO

```
class UserEmailConfirmationMessage
{
    public function __construct(
        public int $userId,
        public string $email
    ){ }
}
```

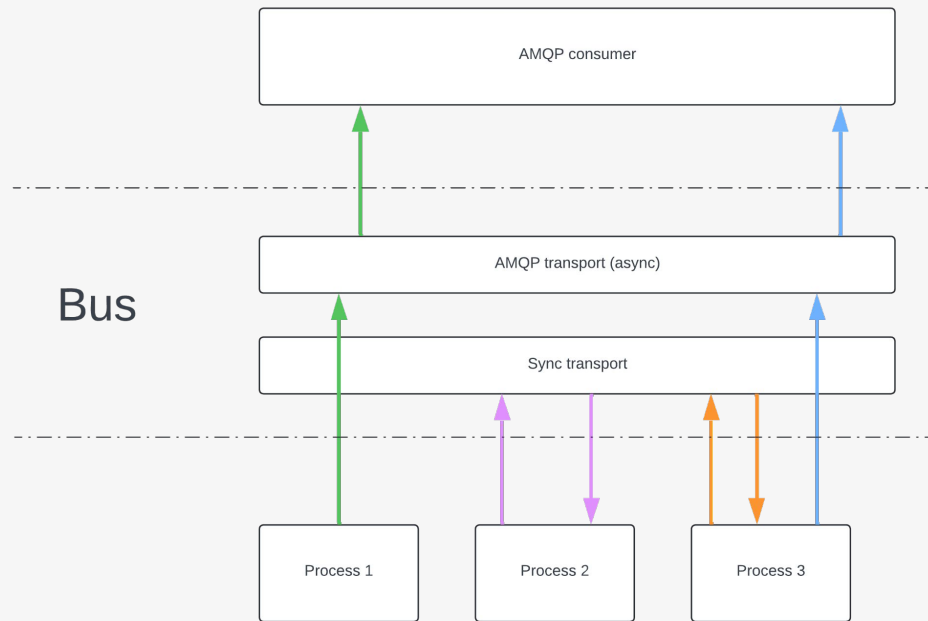
Message Handlers

- An invocable class or a method
- With an `#[AsMessageHandler]` attribute
- **Type-hinted** with the message class
- Can have several handlers for the same message

```
#[AsMessageHandler]
class UserEmailConfirmationHandler
{
    public function __invoke(
        UserEmailConfirmationMessage $message
    ): void {
        // TODO
    }
}
```

Transports and routing

- Can have **several transports**
- Transport can be **sync and async**
- Messages **can be routed** to different transports based on
 - Interface
 - Class (own or parent's)
 - Its metadata



Consuming

- Run a separate **background process** to consume **every transport**
- A single worker can process **several transports**

```
$ php bin/console messenger:consume {transportName}
```

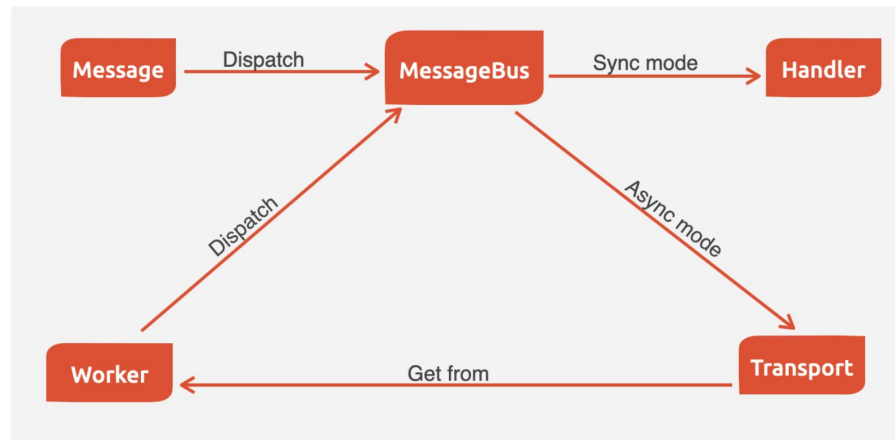
```
$ php bin/console messenger:consume {transport1}  
{transport2}
```

Symfony Scheduler

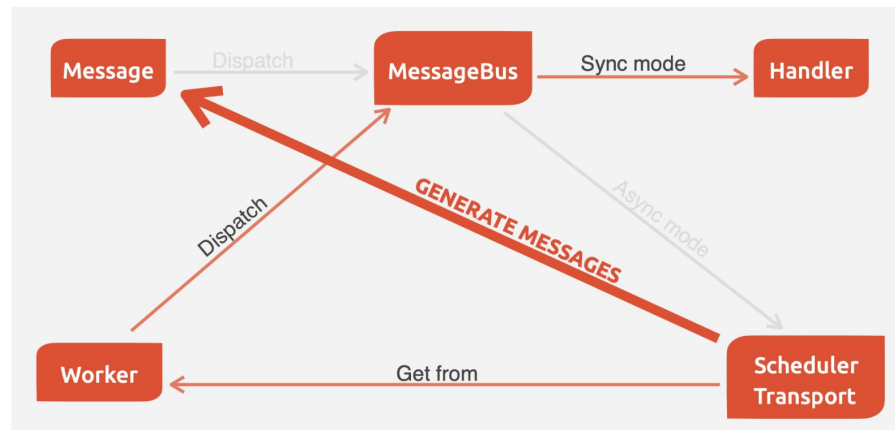


Symfony Scheduler

In Messenger:



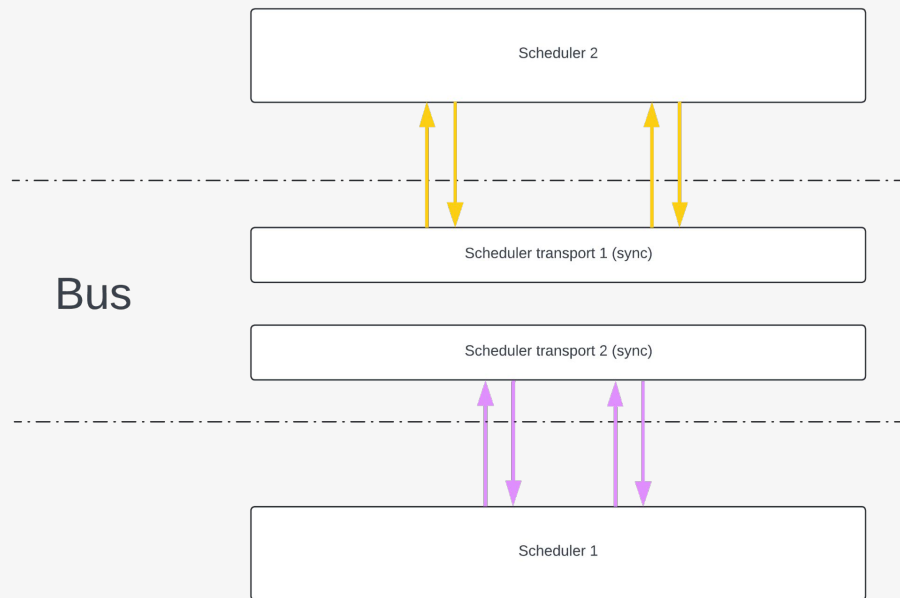
In Scheduler:



Symfony Scheduler

Scheduler can be treated like an extension of the messenger:

- Sends messages to the transport by a schedule
- To its own sync transport
- One scheduler - one transport
- Several schedulers can exist at the same time



Schedule providers

- Implements a `ScheduleProviderInterface`
- With an `#[AsSchedule]` attribute
- Several schedulers can work at the same time

```
#[AsSchedule] // A "default" name
class MainSheduleProvider
implements ScheduleProviderInterface
{
    public function getSchedule(): Schedule
    {
        return new Schedule(); // TODO
    }
}
```

```
#[AsSchedule("priority")] // A "priority" name
class PrioritySheduleProvider
implements ScheduleProviderInterface
{
    public function getSchedule(): Schedule
    {
        return new Schedule(); // TODO
    }
}
```

Scheduling

Cron expressions

- `RecurringMessage::cron()`
- Standard cron expressions
- Pseudorandom hash-expressions

```
return (new Schedule())->with(  
    RecurringMessage::cron(  
        '0 2 * * *',  
        new Message()  
    ),  
    RecurringMessage::cron(  
        '@hourly',  
        new Message()  
    ),  
    RecurringMessage::cron(  
        '#hourly',  
        new Message()  
    ),  
    RecurringMessage::cron(  
        '#(0-10) 2 * * *',  
        new Message()  
    )  
);
```

Scheduling

Frequency trigger

- `RecurringMessage::every()`
- Many ways to define a schedule
- Can set up a specific trigger period

```
return (new Schedule())->with(  
    RecurringMessage::every(  
        '30 minutes',  
        new Message()  
    ),  
    RecurringMessage::every(  
        new DateInterval('PT10M'),  
        new Message()  
    ),  
    RecurringMessage::every(  
        30, // seconds  
        new Message()  
    ),  
    RecurringMessage::every(  
        '2 hours',  
        new Message(),  
        from: '2000-01-01',  
        until: '2000-01-10'  
    )  
);
```

Scheduling

Custom triggers

- `RecurringMessage::trigger()`
- Implements a `TriggerInterface`
- Calculates the next trigger time
- Can be nested into the other

```
class MyTrigger implements TriggerInterface
{
    public function getNextRunDate(
        DateTimeImmutable $run
    ): ?DateTimeImmutable
    {
        return $this
            ->isWorkingHour($run)
                ? $run->modify('+10 minutes')
                : $run->modify('+1 hour');
    }

    // ...
}

// In a schedule provider
return (new Schedule())
    ->with(RecurringMessage::trigger(
        new MyTrigger(),
        new Message()
    ));
```

Scheduling

Custom triggers

- `RecurringMessage::trigger()`
- Implements a `TriggerInterface`
- Calculates the next trigger time
- Can be nested into the other

```
class MyTrigger implements TriggerInterface
{
    public function getNextRunDate(
        DateTimeImmutable $run
    ): ?DateTimeImmutable
    {
        $onlineUsersCount = $this
            ->usersRepository
            ->countOnlineUsers();

        return $onlineUsersCount > 1_000
            ? $run->modify('+10 minutes')
            : $run->modify('+1 hour');
    }
}

// In a schedule provider
return (new Schedule())
    ->with(RecurringMessage::trigger(
        $this->myTrigger,
        new Message()
    ));
```

Scheduling

Custom triggers

- `RecurringMessage::trigger()`
- Implements a `TriggerInterface`
- Calculates the next trigger time
- Can be nested into the other

```
return (new Schedule())
    ->with(RecurringMessage::trigger(
        new WorkingHoursTrigger(
            new SupportAgentsNumberTrigger(
                RecurringMessage::every(60)
            )
        ),
        new Message()
    ));
```

Pitfalls



Loss of schedule state on restart

Problem

- Schedules are **stateless**
- The next trigger time is calculated based on the previous one
- **It's reset** on each restart

```
return (new Schedule())
    ->with(RecurringMessage::every(
        '1 hour',
        new CalculateHourStatisticMessage()
    ));
```


Loss of schedule state on restart

Possible solution

Use a **cron expressions** trigger

```
return (new Schedule())
    ->with(RecurringMessage::cron(
        '@hourly',
        new CalculateHourStatisticMessage()
    ));
```

Loss of schedule state on restart

Possible solution

Use a **stateful** schedule

Will **store the last trigger time** in the cache

```
return (new Schedule())
    ->with(RecurringMessage::every(
        '1 hour',
        new CalculateHourStatisticMessage()
    ))
    ->stateful($this->cache); // CacheInterface
```

Trigger delays

Problem

- Scheduler triggers and processes messages **synchronously**
- Timetable intersections can cause **delays**

```
return (new Schedule()->with(
    RecurringMessage::every(
        '1 hour',
        new SlowMessage())
    ),
    RecurringMessage::every(
        '10 minutes',
        new AnotherSlowMessage()
    )
);
```

Trigger delays

Possible solution

- Split schedules
- Avoid mixing messages with slow handlers and intersected timetables in a single schedule

```
#[AsSchedule('high-prior')]
class HightPriorSheduleProvider
implements ScheduleProviderInterface
{
    public function getSchedule(): Schedule
    {
        return (new Schedule())
            ->with(RecurringMessage::every(
                '1 hour',
                new SlowMessage(),
            ));
    }
}
```

```
#[AsSchedule('default')]
class SheduleProvider
implements ScheduleProviderInterface
{
    public function getSchedule(): Schedule
    {
        return (new Schedule())
            ->with(RecurringMessage::every(
                '10 minutes',
                new AnotherSlowMessage(),
            ));
    }
}
```

Trigger delays

Possible solution

- **Redispatch** messages to **asynchronous** messenger transport
- It requires Symfony Messenger
- Can cause some delays in processing

```
return (new Schedule())->with(
    RecurringMessage::every(
        '1 hour',
        new RedispatchMessage(
            new SlowMessage(),
            'amqp'
        )
    ),
    RecurringMessage::every(
        '10 minutes',
        new RedispatchMessage(
            new AnotherSlowMessage(),
            'amqp'
        )
    )
);
```

Single point of failure

Problem

- Error in the trigger will break **the whole process**
 - Watchdog process will restart it
 - But there will be a **downtime**
 - Messages can be lost
-

Single point of failure

Possible solution

- Be careful
 - Develop **fail proof** triggers
 - **Catch and suppress** all errors
-

Messages duplication

Problem

- Sometimes we can have **several workers** for a single schedule
 - All messages will **be triggered twice**
-

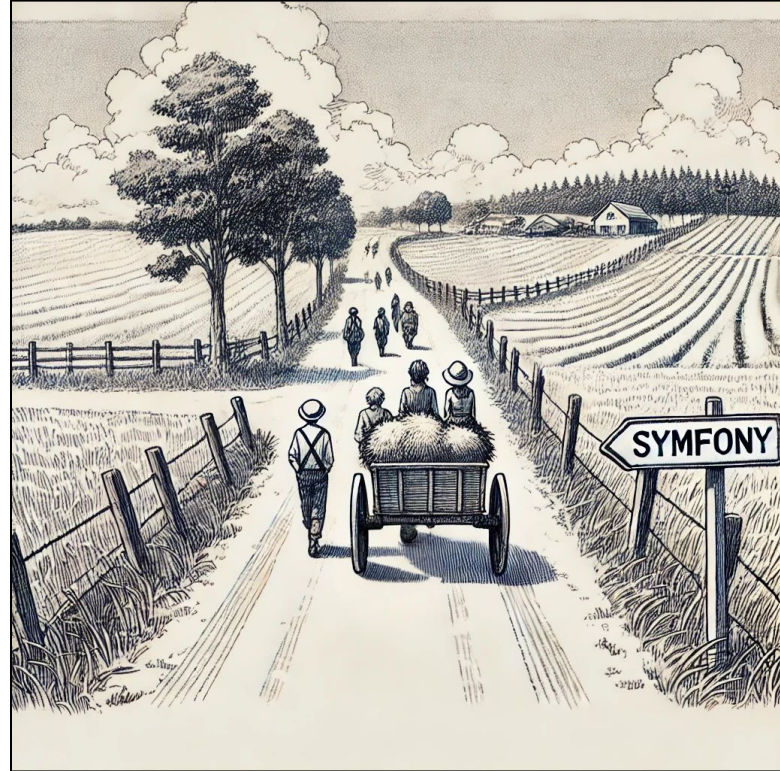
Messages duplication

Possible solution

- Use a lock to synchronize workers
- A symfony/lock package
- Guarantee that only one worker will be active at a time

```
return (new Schedule())->with(
    RecurringMessage::every(
        '1 minute',
        new Message()
    )
)->lock($this->lock); // LockInteface
```

Migration process
Personal experience



Migration process

- Collect all schedules from the application
 - Run a single worker for all of them
 - All messages go to the async transport
 - Make a plan for scaling
-

```
class RunSchedulerWorkerCommand extends Command
{
    public function __construct(
        #[AutowireLocator('scheduler.schedule_provider', 'name')]
        private readonly ServiceProviderInterface $schedules
    ) {
        parent::__construct();
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $names = array_map(
            static fn(string $name) => 'scheduler_' . $name,
            array_keys($this->schedules->getProvidedServices())
        );

        return $this->getApplication()?->doRun(new ArrayInput([
            'command' => 'messenger:consume',
            'receivers' => $names,
        ]), $output) ?? Command::FAILURE;
    }
}
```

Conclusion

- For us, the migration **was definitely worth it**
 - It **may not be appropriate** for your project
 - All potential issues can be solved
 - But at what cost?
 - It needs analysis and planning in advance
-

Questions?

