SymfonyCon
VIENNA 2024

Who did that?
Auditing data changes at scale

SymfonyCon Vienna 2024, Sebastian Plagemann

Index

I‘m Sebastian…

… working for XING / New Work SE

… for a while now

→ https://xing.com
→ https://kununu.com
→ https://new-work.se

But…

… I‘m not a legal expert!

Why?

Storytime

One reason for why you should think about a systemic approach to logging data changes in your application

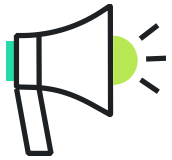
„The client reported that about 500 candidates were deleted by the system“

Support Engineer

„I am unable to determine why the account was changed and by whom.“

Support Engineer

Reasons for audit / activity logging



Debugging

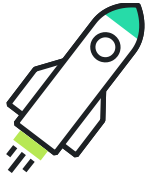
Understand where changes are coming from, when they happened, in what order they occurred and what was the trigger

Security

Find anomalies easier, support fraud detection and help to react to security incidents that might happen

Compliance

In some industries like finance or healthcare there are certain requirements about logging



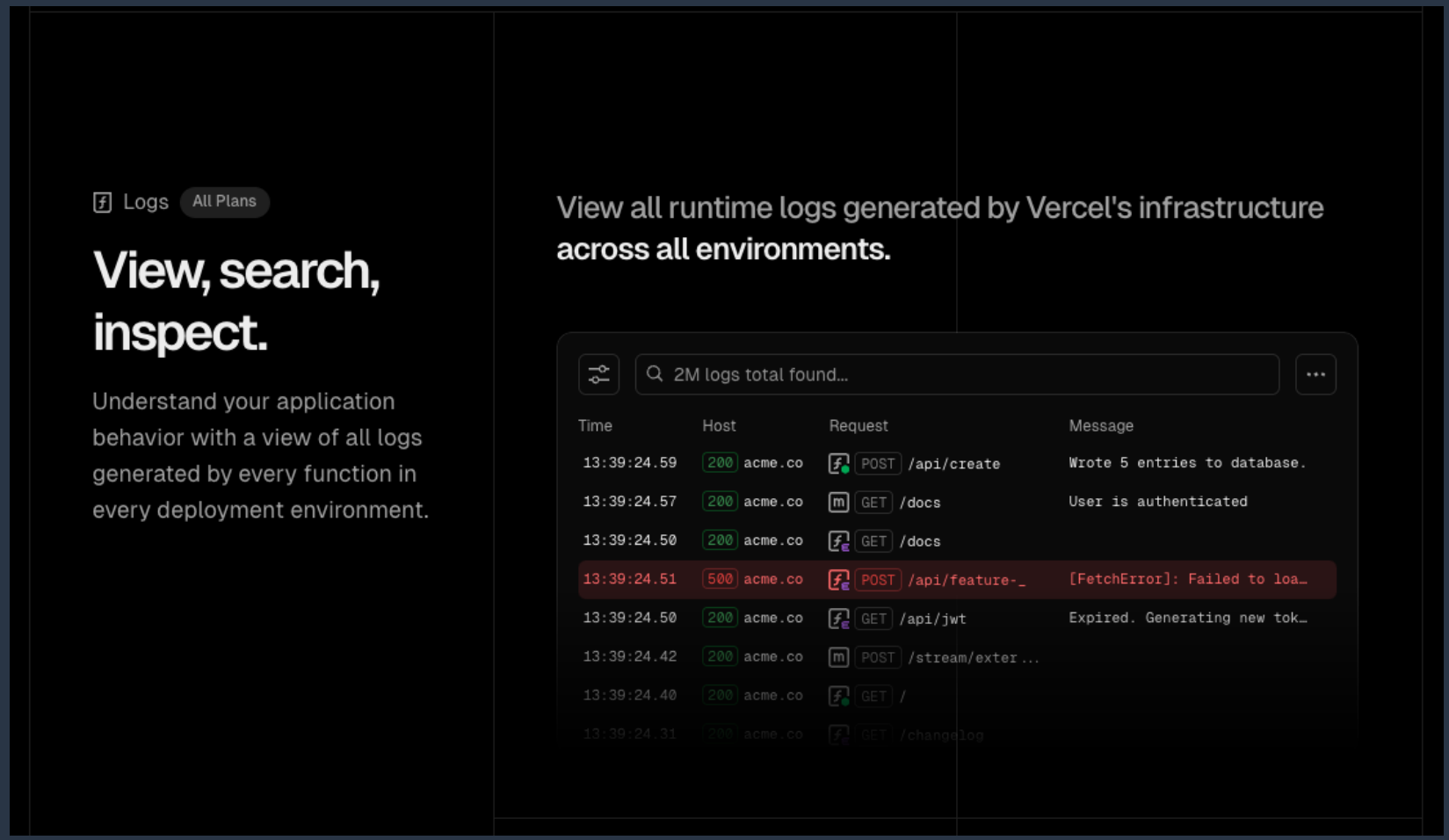
Performance

Can help you understand bottlenecks, redundant or unnecessary actions

Sidenote

The good news...

Usually there is some logging in place



What could be better?

How to Start? Ideas?

Logging in Controller

Problems

- Separation of Concerns
- Code duplication in many controllers
- Less flexibility when switching logging strategy
- Can have performance implications

```php
use Psr\Log\LoggerInterface;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;

// ...

  #[Route('/user', methods: ['PUT'])]
  public function update(Request $request, LoggerInterface $logger): Response
  {
    // do some processing

    $logger->info('User changed!');
  }

// ...
```

Logging in Entity

Problems

- Dependencies in entities
- Code duplication
- Creation and Removal of data not straight forward to log
- Can have performance implications

```php
use Psr\Log\LoggerInterface;

class User {

  // ...

  public function __construct(LoggerInterface $logger): void
  {
    $this->logger = $logger;
  }

  public function setEmail(string $email): void
  {
    $logger->info('Setting email to ' . $email);
    $this->email = $email;
  }

  // ...
}
```

Adding Audit Data to Entity

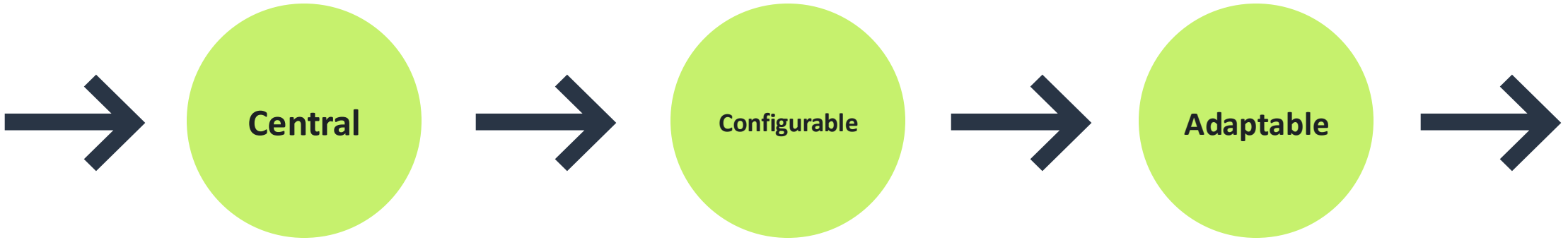
Problems

- Not suitable for auditing
- Not immutable
- Doesn‘t give you the full history
- Increases complexity
- Might get missed when data is updated

```php
class User {

  #[ORM\Column]
  private \DateTimeInterface $createdAt;

  #[ORM\Column]
  private User $createdBy;

  #[ORM\Column]
  private \DateTimeInterface $updatedAt;

  #[ORM\Column]
  private User $updatedBy;

  #[ORM\Column]
  private \DateTimeInterface $deletedAt;

  #[ORM\Column]
  private User $deletedBy;

  // ...
}
```

How to really Start?

The goal



Having **one central place** to capture changes

Easy to configure what should be logged

Having the possibility to **adapt your logging strategy** along the way

Concepts

We‘re looking at this

Change Data Capture (CDC)

Use database functionality to capture change events and propagate them to downstream consumers.

Pro
Captures all changes on database layer

Contra
Application context not available

Hook into ORM / DBAL

Use lifecycle events to hook into your application logic when changes are persisted to the database.

Pro
Easy to get started with

Contra
Changes outside of lifecycle events are not captures

Event stream

Use message queues or other forms of event bus systems to capture changes in microservice architectures

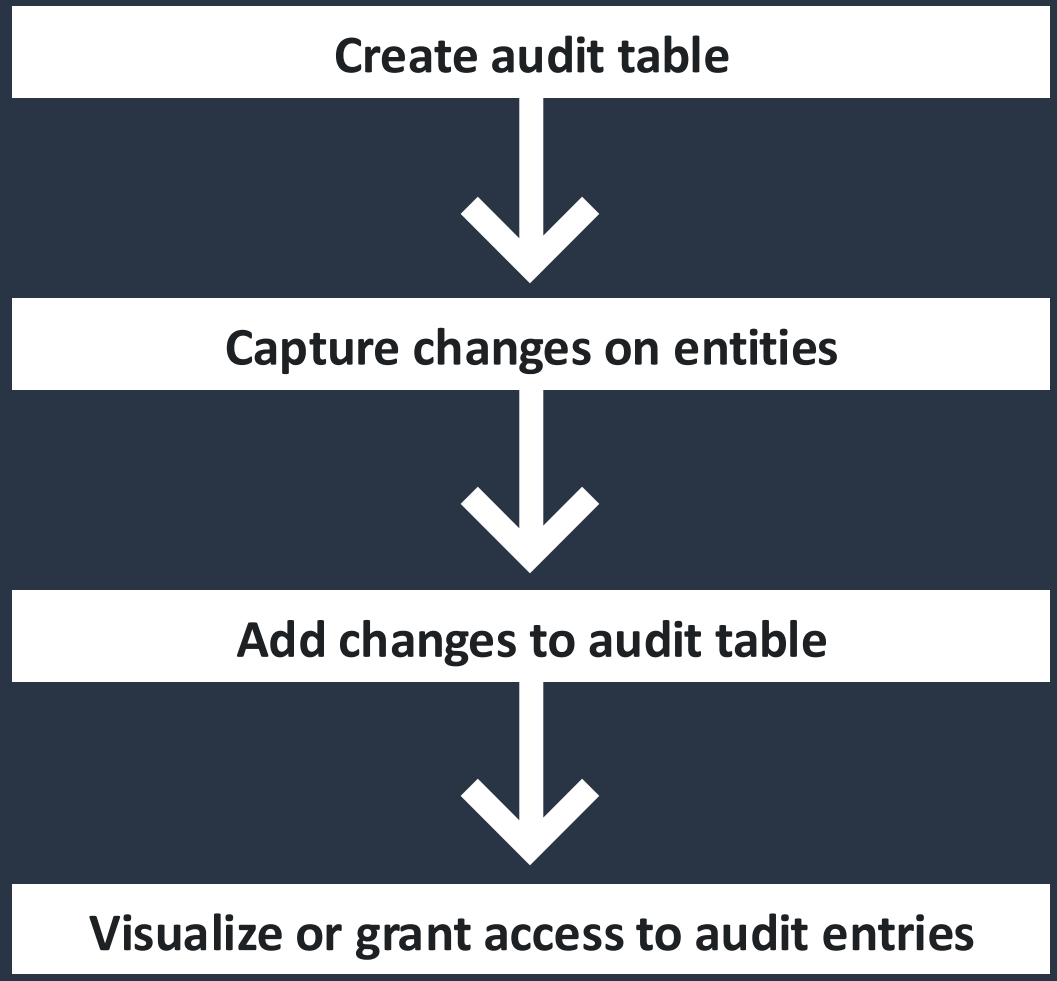
Pro
Unified view across different services

Contra
No service internal details available

Hook into ORM

There are bundles to get started quickly…

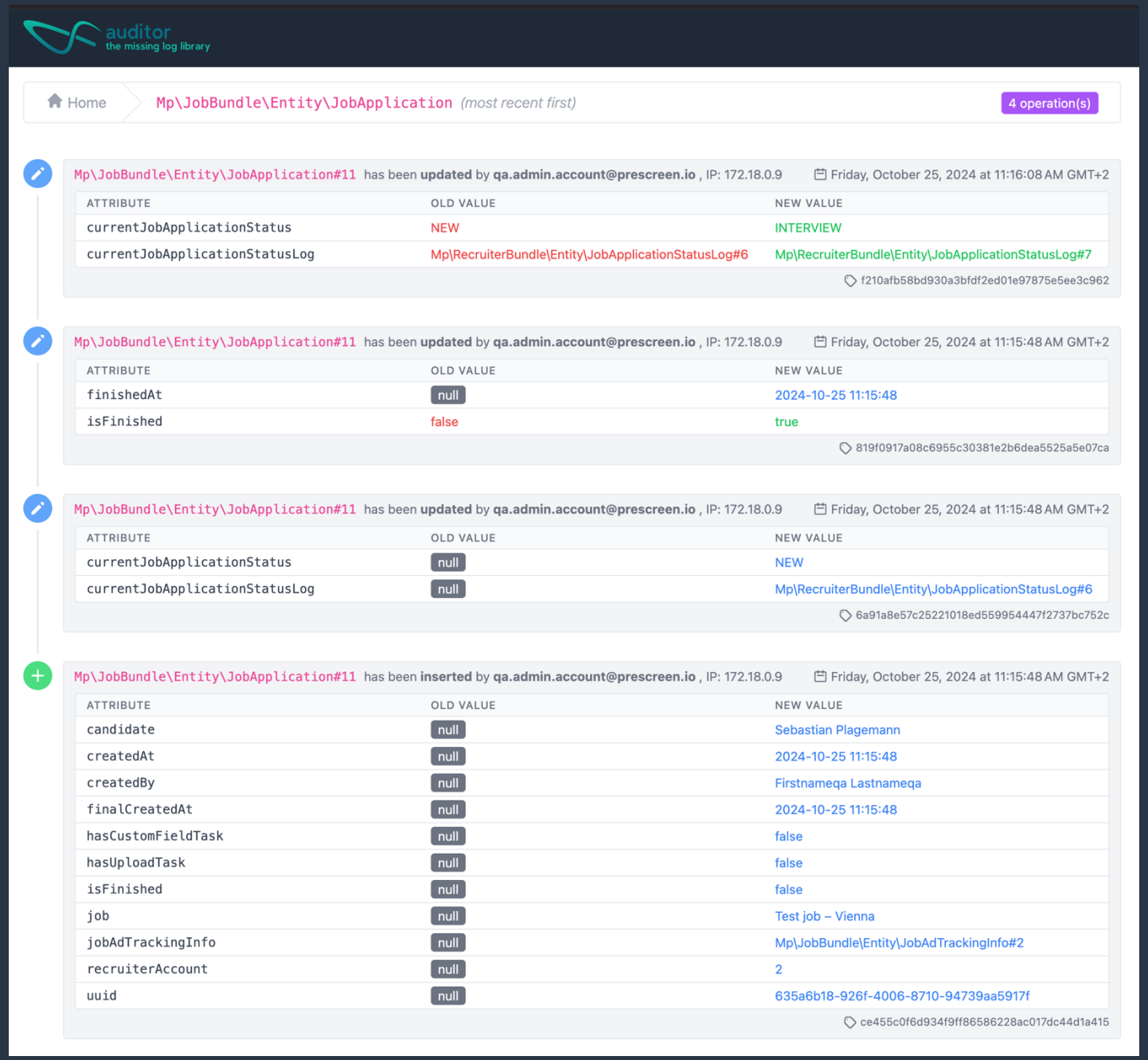
damienharper/auditor-bundle
sonata-project/entity-audit-bundle



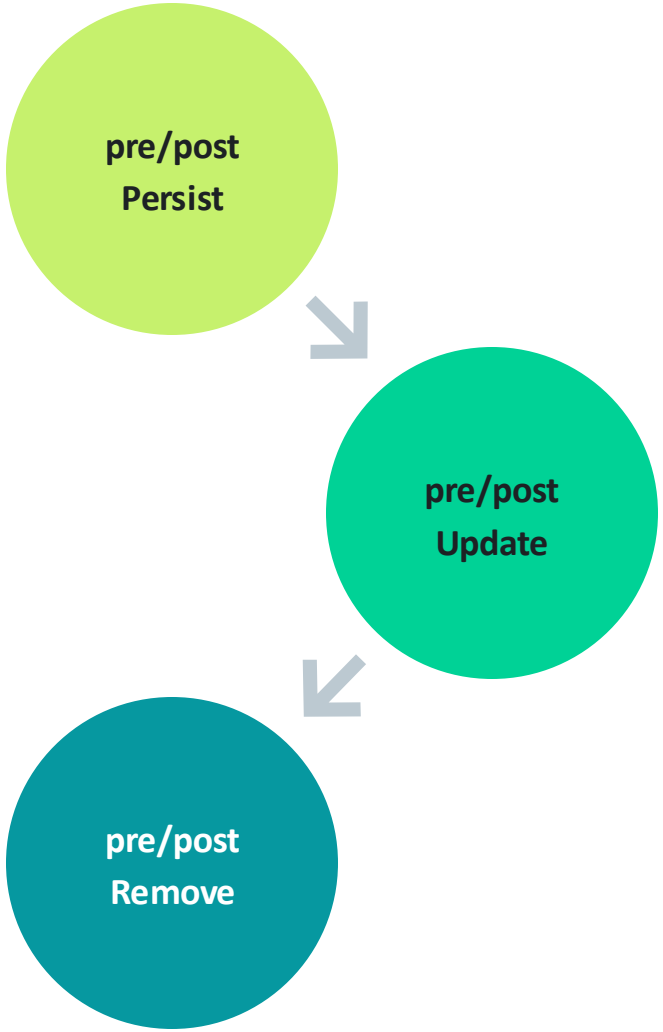
Hook into ORM

There are bundles to get started quickly…

damienharper/auditor-bundle
sonata-project/entity-audit-bundle



Doctrine Event System



- Centralized change detection
- Decoupled from business logic
- Granular control
- Transactions
- Extensibility
- Easy to implement

Note: DQL queries will not be captured

Doctrine EntityListener

```php
// src/EventListener/UserChangedNotifier.php
namespace App\EventListener;

use App\Entity\User;
use Doctrine\ORM\Event\PostUpdateEventArgs;
use Doctrine\Bundle\DoctrineBundle\Attribute\AsEntityListener;
use Doctrine\ORM\Events;

#[AsEntityListener(event: Events::postUpdate, method: 'postUpdate', entity: User::class)]
class UserChangedNotifier
{

    public function __construct(private readonly LoggerInterface $logger) {}

    // the entity listener methods receive two arguments:
    // the entity instance and the lifecycle event
    public function postUpdate(User $user, PostUpdateEventArgs $event): void
    {
        $this->logger->info('The user ' . $user->getId() . 'was changed!');
    }
}
```

Doctrine EventListener

```php
// src/EventListener/AuditLogger.php
namespace App\EventListener;

use App\Entity\User;
use Doctrine\Bundle\DoctrineBundle\Attribute\AsDoctrineListener;
use Doctrine\ORM\Event\PostPersistEventArgs;
use Doctrine\ORM\Events;
use Psr\Log\LoggerInterface;

#[AsDoctrineListener(event: Events::postPersist, priority: 500, connection: 'default')]
class AuditLogger
{

    public function __construct(private readonly LoggerInterface $logger) {}

    // the listener methods receive an argument which gives you access to
    // both the entity object of the event and the entity manager itself
    public function postPersist(PostPersistEventArgs $args): void
    {
        $entity = $args->getObject();

        // if this listener only applies to certain entity types,
        // add some code to check the entity type as early as possible
        if (!$entity instanceof User) {
            return;
        }

        $this->logger->info('The user ' . $entity->getId() . ' was created');
        // ...
    }
}
```

Scaling & Other Considerations

Storage
Performance

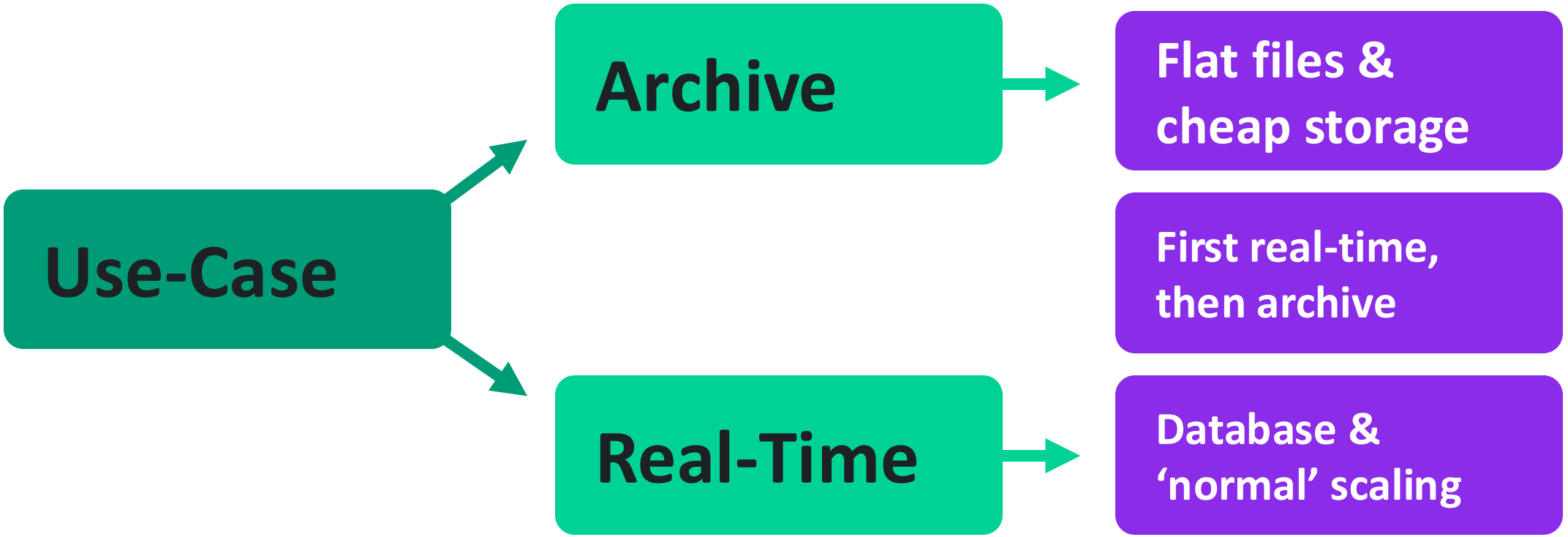
- **Storage Types:** High-throughput databases for real-time; object storage for archival.
- **Capacity & Costs:** Plan for growth; use tiered storage for cost efficiency.
- **Retention Policies:** Define retention periods based on compliance; automate archiving / deletion.
- **Partitioning & Sharding:** Partition by time or criteria; use sharding for high write throughput.

Storage

Performance

- **Write Throughput:** Estimate throughput, think about batching & buffering.
- **Asynchronous:** Use queues or background processing to decouple from main application.
- **Low Latency:** Maintain data locality and avoid slow network latency.
- **Log aggregation tools:** Tools like Datadog or Elastic Stack can help handling large amounts of data, but come at a price.

What's the main Use-Case?



Security

Data Privacy

- **Access Control:** Limit access with role-based permissions.
- **Log Integrity:** Store logs in a secure, centralized system where only authorized processes can write to them.
- **Encryption:** Encrypt logs at rest and in transit to prevent eavesdropping.
- **Anonymization:** Avoid storing sensitive or personally identifiable information (PII) in logs unless necessary.
- **Environment Segregation:** Ensure that data is segregated by environment (dev, staging, prod).

Security

Data Privacy

- **Data Minimization:** Collect only the data strictly necessary for auditing.
- **Guidelines for Accessing Logs:** Define who is allowed to access the data and in what circumstances.
- **Compliance with Regulations:** Ensure logging practices align with data privacy laws (e.g., GDPR).
- **User Consent and Transparency:** Inform users about logging practices (e.g. via privacy policy) and obtain consent when required.

Summary

Find a central place to capture data changes, e.g. Doctrine events

Define your use-cases and access pattern

Chose storage depending on your needs

Consider other factors like performance, security and data privacy

Thanks.

SymfonyCon
VIENNA 2024