Doctrine ORM and NoSQL

Symfony Catalunya 2016

Benjamin Eberlei @beberlei

23.07.2016


tideways.io


Qafoo
passion for software quality

Before we begin

On the state of Doctrine

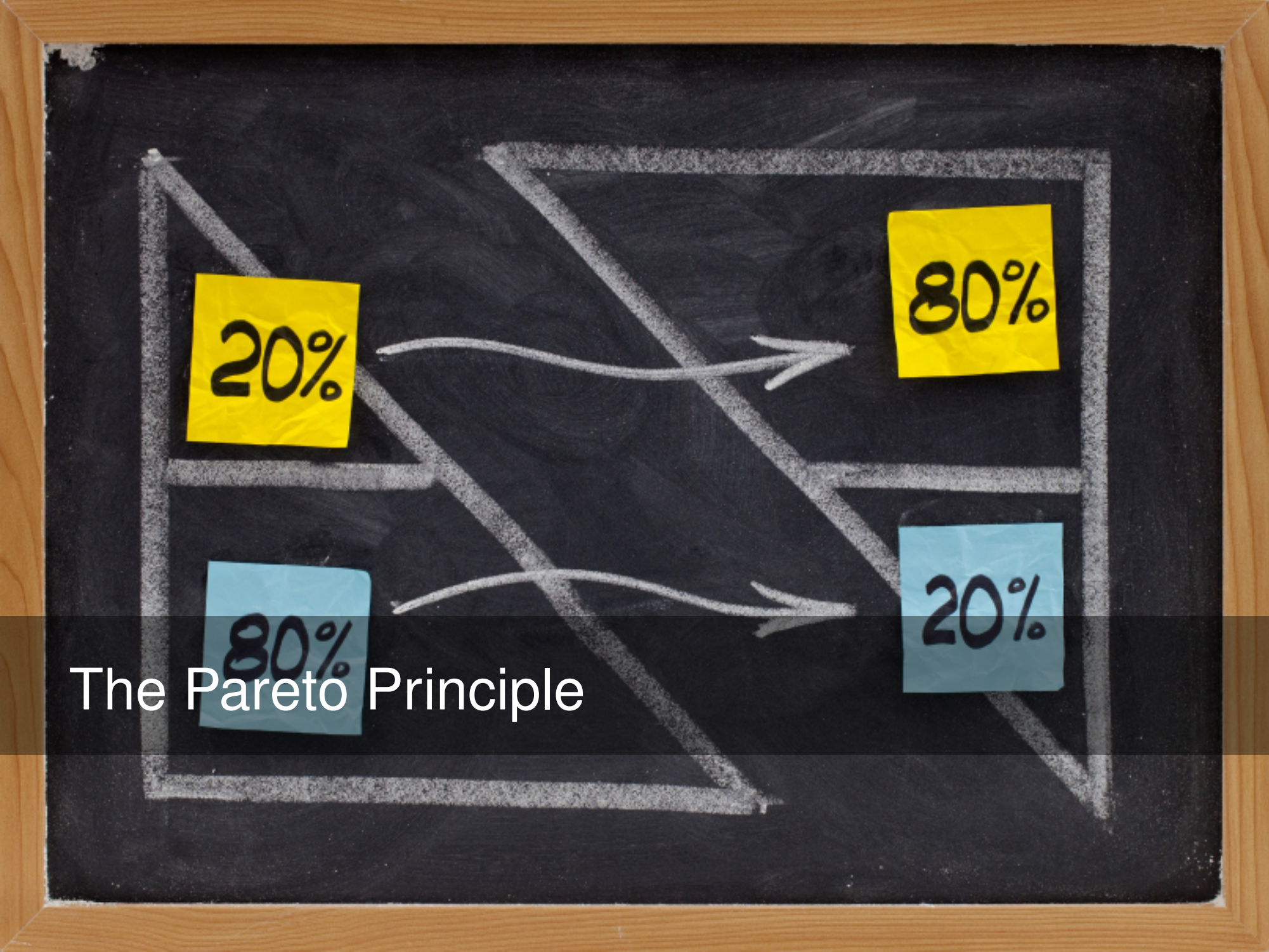
My condolences, you're now the maintainer of a popular open source project

@danielbachhuber - June 26, 2016

The following are my slides and notes from a talk I gave yesterday at WordCamp Europe. This is more or less what I said, but not exactly.

Vilfredo Pareto (1848-1923)

Licensed under CC BY-SA 3.0 via Commons https://commons.wikimedia.org/wiki/File:Vilfredo_Pareto.jpg



The Pareto Principle

Economics and Doctrine?

We built Doctrine for the 80% use case + `$some`.

Doctrine ORM and NoSQL

We are going to talk about:

Pragmatic Solutions to for non-relational data

Why NoSQL/Non-Relational?

- The web is mostly about documents
  - Products
  - Articles, Blog-Posts, ...
  - Emails, Messages, ...
  - Content

Starting with a Problem

Doctrine ORM and ODM are not particularly well suited for polygot persistence

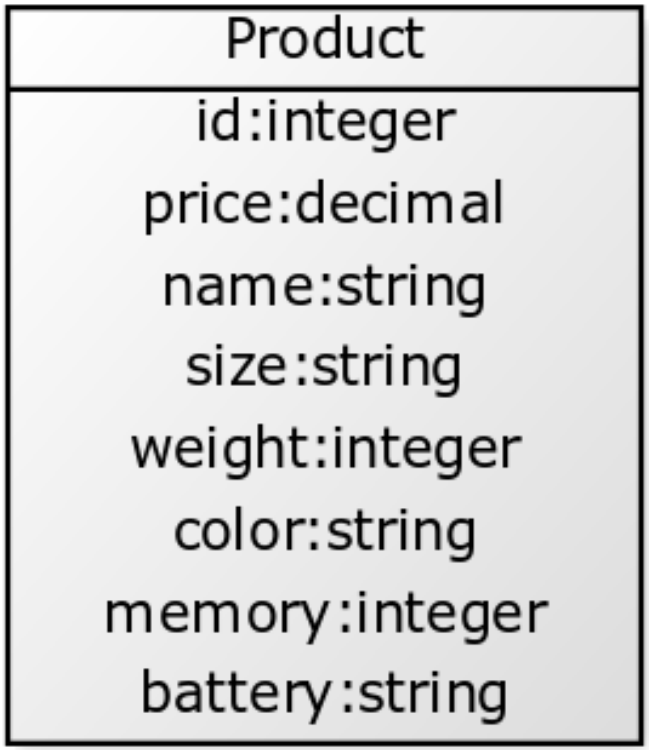
Doctrine Polygot Persistence

- No support for mixed storage columns
- No support for relations between different mappers
- no support for transactions across mappers

Tooling vs Aesthetics


Leaky Abstraction

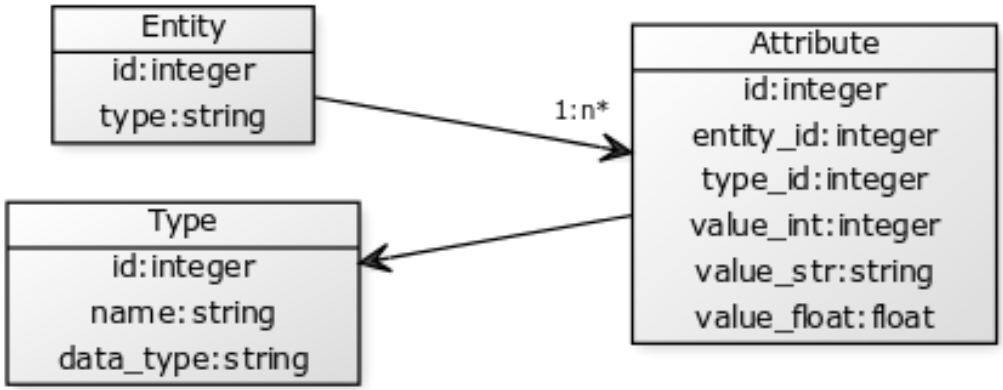
A product



Disadvantages

- Can lead to entities with **MANY** nullable properties
- Users cannot add properties add runtime
- Performance is degrading significantly:
  - Doctrine Hydration
  - Symfony Forms

Entity-Attribute-Value (EAV) Models



A data model to encode, in a space-efficient manner, entities where the number of attributes (properties, parameters) that can be used to describe them is potentially vast, but the number that will actually apply to a given entity is relatively modest.

Product Example

```php
<?php

use Doctrine\Common\Collections\ArrayCollection;

class Product
{
    /** @ORM\Id ... */
    private $id;
    /** @ORM\OneToMany(...) */
    private $attributes;

    public function __construct()
    {
        $this->attributes = new ArrayCollection();
    }
}
```

Product Example

```php
<?php

class ProductAttribute
{
    private $id;
    /** @ORM\ManyToOne(...) */
    private $product;
    private $name;
    private $value;

    public function __construct(Product $product, $name)
    {
        $this->product = $product;
        $this->name = $name;
    }
}
```

Product Example

```php
<?php
class Product
{
    // ...
    public function setAttribute($name, $value)
    {
        $this->attributes[$name] = new ProductAttribute(
            $this,
            $name
        );
        $this->attributes[$name]->setValue($value);
    }

    public function getAttribute($name)
    {
        return $this->attributes[$name];
    }
}
```

Disadvantages

- Complicated OneToMany/ManyToOne code
- Complicated Query logic
- Generic code, hard to encode business logic
- Slow Doctrine hydration
- Abysmal performance for standard use-cases:
  - Listing of many products
  - Batch updating of products

Fix #1: Storing JSON Attributes

```php
<?php
class Product
{
    /** @ORM\Column(type="json_array") */
    private $attributes = [];

    public function setAttribute($name, $value)
    {
        $this->attributes[$name] = $value;
    }
}
```

Fixed:

- No complicated one-to-many/many-to one code
- Hydration and Batch performance issues

Fix #2: Storing JSON in Relational Databases

- Doctrine `json_array` uses CLOB/text for storage by default
- MySQL 5.7.8+ and PostgreSQL 9.2+ have JSON datatype
- PostgreSQL 9.4+ with JSONB datatype

MySQL JSON Datatype support

- Store up to max_allowed_packet data
- Validation of JSON syntax
- Support for indexes via virtual columns
- New functions for efficient manipulation:
  - `JSON_EXTRACT` to fetch subvalues
  - `JSON_SET` to UPSERT JSON without full replace
  - `JSON_INSERT` like INSERT
  - `JSON_REMOVE` like DELETE
  - `JSON_REPLACE` only whe value exists
- Support for compressed InnoDB storage via Barracuda + `ROW_FORMAT=COMPRESSED`

MySQL JSON Datatype support

```
mysql> CREATE TABLE product (id INTEGER, attributes JSON);
Query OK, 0 rows affected (0.20 sec)

mysql> INSERT INTO product VALUES (1, '{"weight": "100kg",
    "color": "blue"}');
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM product;
+------+-----------------------------------------+
| id   | attributes                              |
+------+-----------------------------------------+
|    1 | {"color": "blue", "weight": "100kg"} |
+------+-----------------------------------------+
1 row in set (0,00 sec)
```

PostgreSQL JSON Datatype support

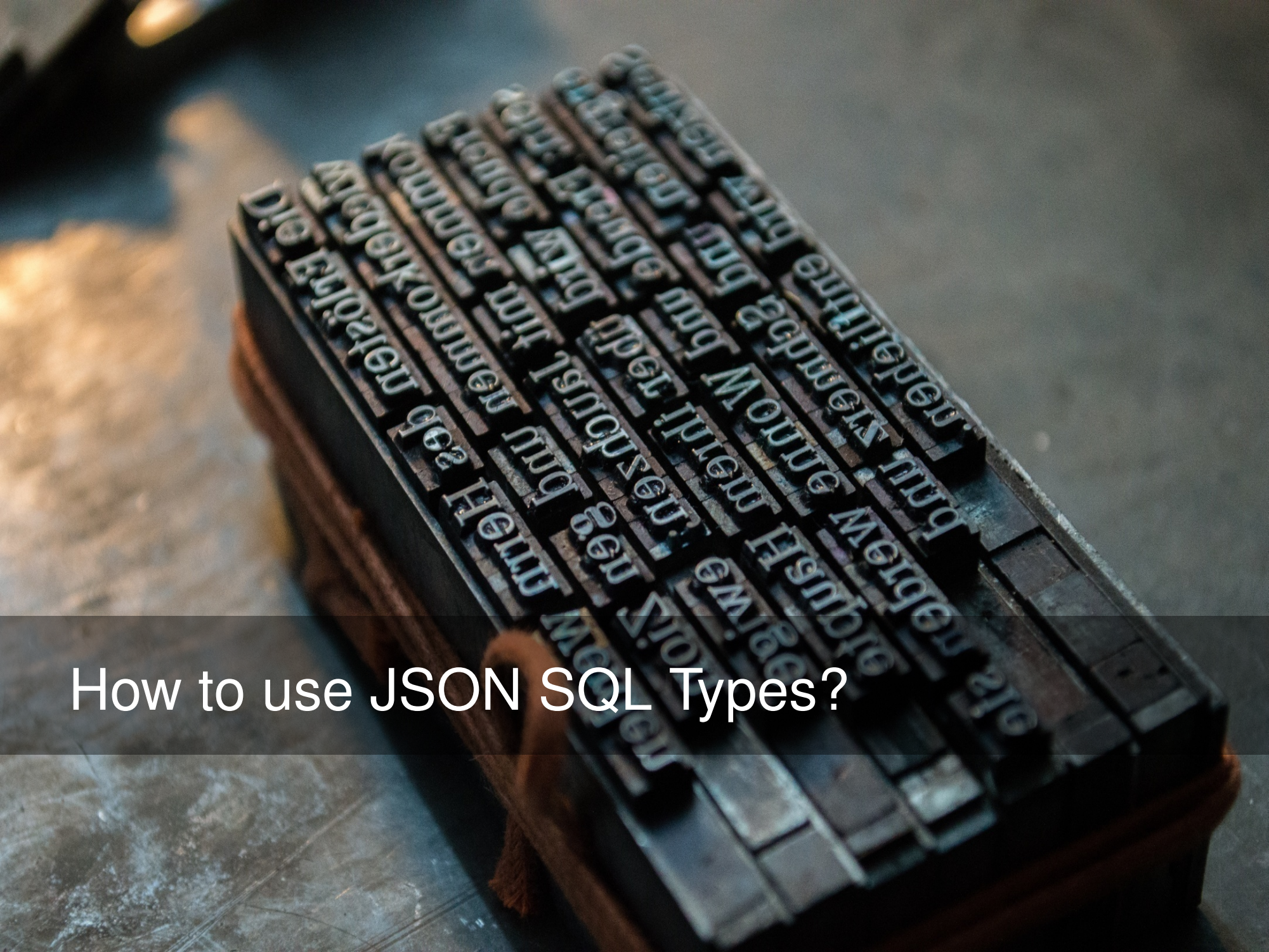
- Validation of JSON syntax
- Versions 9.3, 9.4 and 9.5 added plenty of operators/functions
- Caveats:
  - Server charset should be UTF-8
  - No indexing for JSON datatype

PostgreSQL JSONB Datatype support

- Binary variant of the `JSON` datatype
- Higher performance, but no preservation of order/whitespaces
- Support for indexing via GIN indexes

PostgreSQL JSONB Datatype support

```
1  postgres=# CREATE TABLE product (id SERIAL, attributes
       JSONB);
2  CREATE TABLE
3  postgres=# INSERT INTO product (attributes) VALUES ('{"
       color": "blue"}');
4  INSERT 0 1
5
6  postgres=# CREATE INDEX ON product ((attributes->'color'));
7  CREATE INDEX
8
9  postgres=# SELECT * FROM product
10   WHERE attributes -> 'color' ? 'blue';
11
12  id |     attributes
13 ----+--------------------
14   1 | {"color": "blue"}
15 (1 row)
```



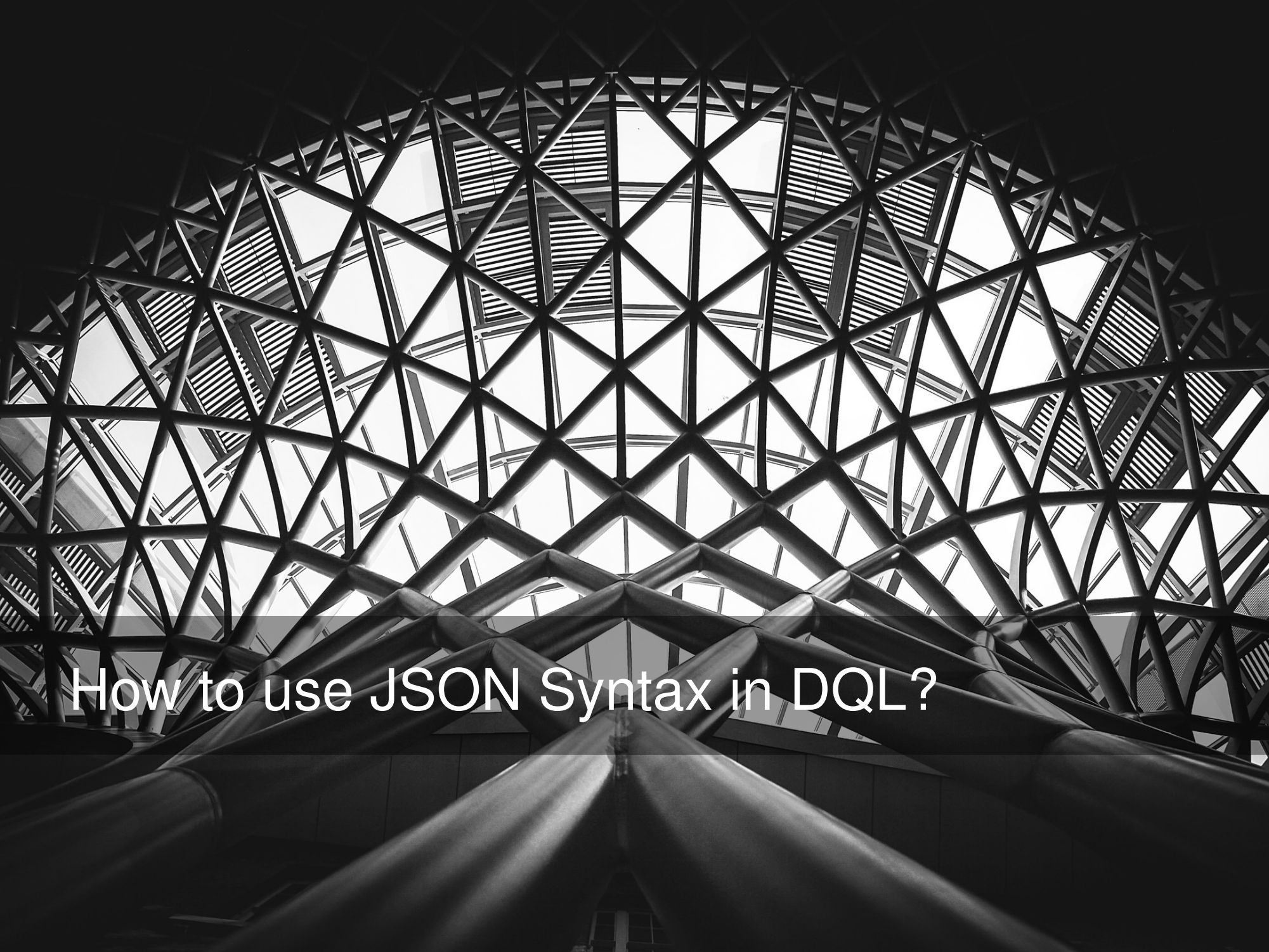
How to use JSON SQL Types?

DBAL Custom Type

```php
<?php

namespace AppBundle\Doctrine;

use Doctrine\DBAL\Types\JsonType;

class MySQLJsonType extends JsonType
{
    public function getSQLDeclaration(array $field,
        AbstractPlatform $platform)
    {
        return 'JSON';
    }
}
```

DBAL Custom Type

```
# app/config/config.yml
#
doctrine:
    dbal:
        types:
            json: "AppBundle\\Doctrine\\MySQLJsonType"
        mapping_types:
            json: "json"
```


How to use JSON Syntax in DQL?

JSONExtract DQL Function

```php
<?php
class JsonExtractFunction extends FunctionNode
{
    private $field;
    private $path;

    public function parse(Parser $parser)
    {
        $parser->match(Lexer::T_IDENTIFIER);
        $parser->match(Lexer::T_OPEN_PARENTHESIS);
        $this->field = $parser->StringPrimary();
        $parser->match(Lexer::T_COMMA);
        $this->path = $parser->StringPrimary();
        $parser->match(Lexer::T_CLOSE_PARENTHESIS);
    }
}
```

JSONExtract DQL Function

```php
<?php

class JsonExtractFunction extends FunctionNode
{
    public function parse(Parser $parser) { /** .. */ }

    public function getSQL(SqlWalker $walker)
    {
        return 'JSON_EXTRACT(' .
            $walker->walkStringPrimary($this->field)
            . ', $.' .
            $walker->walkStringPrimary($this->path) .
        ')';
    }
}
```

JSONExtract DQL Function

```
1 doctrine:
2   orm:
3     # ...
4     dql:
5       string_functions:
6         JSON_EXTRACT: AppBundle\\Doctrine\\
                  JsonExtractFunction
```

JSONExtract DQL Function

```php
<?php
$dql = <<<DQL
SELECT p
FROM Product p
WHERE JSON_EXTRACT(p.attributes, "color") = ?1
DQL;

$query = $entityManager->createQuery($dql);
$query->setParameter(1, "blue");

$blueProducts = $query->getResult();
```


How to avoid array soup?

A better JSON custom type

```php
<?php

class ProductAttributes extends Attributes
{
    public $color;
    public $size;
    public $memory;
    public $battery;
}
```

A better JSON custom type

```php
<?php

class ProductAttributeType extends Type
{
    public function convertToPHPValue($value,
        AbstractPlatform $platform)
    {
        if ($value === null) {
            return null;
        }

        return new ProductAttributes(
            json_decode($value, true)
        );
    }
}
```

Warning about DBAL Custom Types

When checking for fields to update, Doctrine ORM compares object-based DBAL types by reference and **NOT** by value.

Assumption: Object types are immutable values

One Problem Left

Nested JSON objects use simplistic (de-)serialization logic


Improving the JSON type

Improved JSON type

- Make use of serialization libraries in DBAL types
  - JMS Serializer
  - Symfony Serializer
- Disadvantages
  - Affects performance negatively again
  - Different hydration logic than Doctrine

JSON Serializer Type

```php
<?php
class JsonDocumentType extends JsonArrayType
{
    public function convertToDatabaseValue($value,
        AbstractPlatform $platform)
    {
        return $this->getSerializer()
            ->serialize($value, 'json', []);
    }

    public function convertToPHPValue($value,
        AbstractPlatform $platform)
    {
        return $this->getSerializer()
            ->deserialize($value, $this->type, 'json', []);
    }
}
```

JSON Serializer Type

```php
<?php
class JsonDocumentType extends JsonArrayType
{
    private $serializer;
    private $type;

    public function setSerializer($serializer)
    {
        $this->serializer = $serializer;
    }

    public function setType($type)
    {
        $this->type = $type;
    }
}
```

JSON Serializer Type

```php
<?php

class AppBundle extends Bundle
{
    public function boot()
    {
        Type::addType('product_attributes',
            JsonDocumentType::class);

        $type = Type::getType('product_attributes');
        $type->setSerializer(
            $this->container->get('serializer')
        );
        $type->setType(ProductAttributes::class);
    }
}
```

JSON Serializer Type

```php
<?php

class Product
{
    /**
     * @ORM\Column(type="product_attributes")
     */
    private $attributes;
}

class ProductAttributes
{
    private $color;
    private $size;
}
```

JSON Serializer Type

```php
<?php

$attributes = new ProductAttributes();
$attributes->color = 'blue';
$attributes->size = 'XL';

$product = new Product();
$product->setAttributes($attributes);

$entityManager->persist($product);
$entityManager->flush();
```

JSON Serializer Type

```
mysql> select * from Product;
+----+-------------------------------------+
| id | attributes                          |
+----+-------------------------------------+
|  1 | {"color":"blue","size":"XL"}        |
+----+-------------------------------------+
```

Doctrine JSON ODM

- `https://github.com/dunglas/doctrine-json-odm`
- One generic `json_document` type only
- Supports arbitrary objects in the same property
- Includes PostgreSQL JSON(b) types
- MySQL support forthcoming


How to integrate multiple databases?

Polygot Databases

Developers using Doctrine ORM have also been using:

- Redis
- Elasticsearch
- MongoDB

Polygot Databases

How to tackle multiple databases?

1. Use Doctrine
2. Write your own mapper
3. Use CQRS

Using Doctrine

- Advantages
  - Code already exists
  - NoSQL databases are simple to map
- Disadvantage: Working within limits of tool
  - No relation support
  - Multiple objects ”per entity” required

Relations through Repository

```php
<?php

/** @ORM\Entity */
class Product
{
    private $id;
}

/** @ODM\Document */
class ProductAttribute
{
    private $id;
    private $color;
    private $size;
}
```

Relations through Repository

```php
<?php

$product = $entityManager->find(Product::class, $id);
$attributes = $documentManager->find(
    ProductAttributes::class,
    $product->getId()
);
```

Searching with Elasticsearch

```php
<?php

$params = [
    'index' => 'products',
    'type' => 'product',
    'body' => ['query' => ['match' => ['color' => 'blue']]]
];

$response = $client->search($params);

$ids = array_map(function ($doc) {
    return $doc['_id'];
}, $response['hits']['hits']);

$products = $productRepository->find(array('id' => $ids));
```

Mapping KeyValue Stores

- Warning: Still under development
- Nearing first stable release
- Supported stores
  - Redis
  - Amazon DynamoDB
  - Riak
  - CouchDB
  - MongoDB
  - several more

`github.com/doctrine/KeyValueStore`

THANK YOU

Rent a quality expert
qafoo.com