DEEPer

Further Databases In PHP

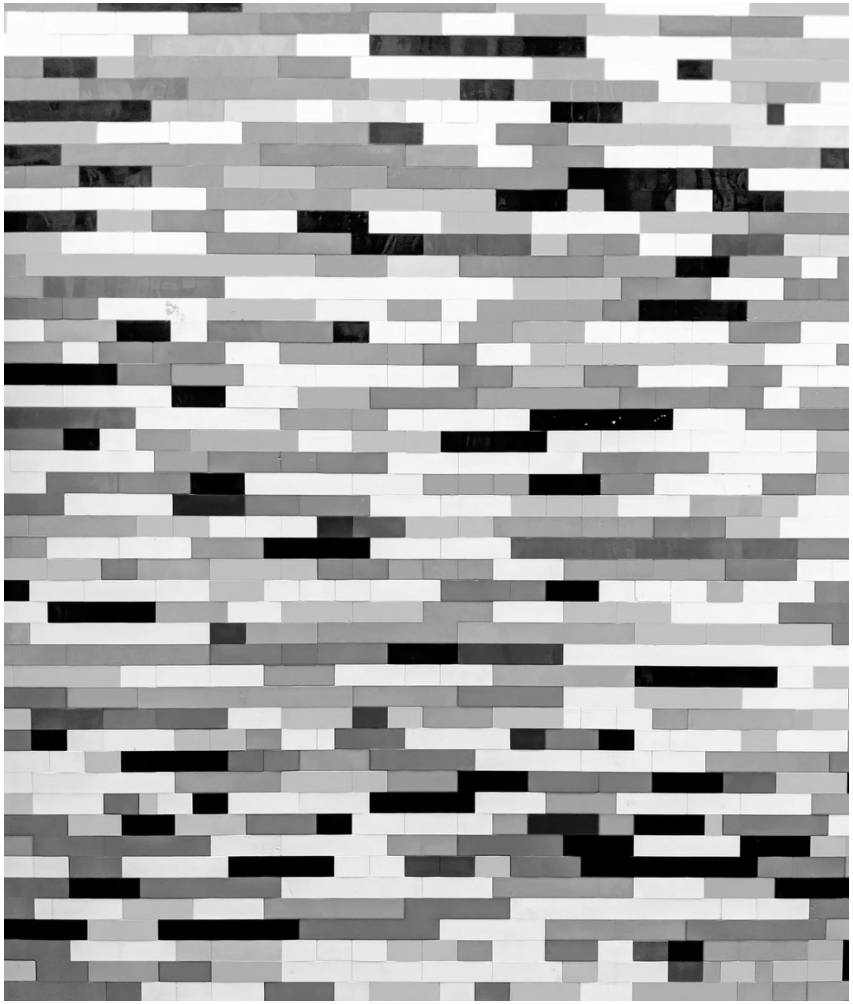
Week 6 Session 2

Committing & Pushing

- It’s important to push work!
- Pushing means we can see your code on BitBucket, and we can track your progress and help when you’re stuck
- Think of it like a backup of your work – we can see a history of changes and easily revert if something breaks

```
git add . && git commit -m "Task complete" && git push
```

TODAY'S SESSION



01 Manual Hydration

02 Architecture Principles

Manual Hydration

Beyond PDO

01

PDO Hydration Summary


```php
class Product
{
    public int $id;
    public string $title;
}

$db = new PDO('mysql:host=mysql;dbname=lecture', 'root', 'root');
$db->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);

$stmt = $db->prepare('SELECT * FROM `product` WHERE `id` = :id');
$stmt->execute(['id' => $_GET['id']]);

// A single Product instance
$product = $stmt->fetchObject(Product::class);
```

- PDO can hydrate instances of classes we provide
- PDO maps column name to property name
- If data from related tables is returned through joins, this data cannot be hydrated in this way

Table Joins - Aliasing

- When referencing tables in queries, we can provide aliases
- Aliases can then be used in selects, joins etc
- An alias is defined with a string after a table name
- Using the first character of the table is common, but the alias must be unique in the query

```
SELECT p.id
FROM product p
INNER JOIN checkin c
  ON c.product_id = p.id
```

Fetching Join Data Summary


```php
$stmt = $db->prepare('
  SELECT * FROM product p
  LEFT JOIN checkin c ON c.product_id = p.id
  WHERE p.id = :id
');
$stmt->execute(['id' => $_GET['id']]);

$productData = $stmt->fetchAll(PDO::FETCH_ASSOC);

var_dump($productData);
```

```
array (size=2)
  0 =>
    array (size=7)
      'id' => string '1' (length=1)
      'title' => string 'Macbook Pro' (length=11)
      'product_id' => string '1' (length=1)
      'name' => string 'John Smith' (length=10)
      'rating' => string '2' (length=1)
      'review' => string 'Not keen on this' (length=16)
      'posted' => string '2020-09-07 09:10:51' (length=19)
  1 =>
    array (size=7)
      'id' => string '2' (length=1)
      'title' => string 'Macbook Pro' (length=11)
      'product_id' => string '1' (length=1)
      'name' => string 'Jane Doe' (length=8)
      'rating' => string '5' (length=1)
      'review' => string 'Love it!' (length=8)
      'posted' => string '2020-09-07 09:10:51' (length=19)
```

- All data selected, regardless of table, is returned in a flat array
- The join means we can can have multiple results returned, so we must use `$stmt->fetchAll()`
- PDO can't automatically determine which data belongs to the **product**, and which data belongs to the **check-in**
- We therefore need to handle this process manually

Manual Hydration

- **Hydration** is the process of converting data in an array format into a populated object structure
- A hydrator in its simplest form is a function accepting an array, manually creating objects, assigning properties and returning them
- There are many libraries available to automate this process

Product Hydration


```php
class Product
{
    public int $id;
    public string $title;
}

function hydrateProduct(array $data): Product
{
    $product = new Product();
    $product->id = $data['id'];
    $product->title = $data['title'];

    return $product;
}

$stmt = $db->prepare('SELECT id, title FROM product WHERE id = :id');
$stmt->execute(['id' => $_GET['id']]);

$productData = $stmt->fetch(PDO::FETCH_ASSOC);
$product = hydrateProduct($productData);

var_dump($product);
```

- This code effectively emulates the functionality PDO provided

```
object(Product)[3]
  public int 'id' => int 1
  public string 'title' => string 'Macbook Pro' (length=11)
```

Check-In Hydration


```php
class CheckIn
{
    public int $id;
    public int $productId;
    public string $review;
}

function hydrateCheckin(array $data): CheckIn
{
    $checkIn = new CheckIn();
    $checkIn->id = $data['id'];
    $checkIn->productId = $data['product_id']; // Opportunity to rename
    $checkIn->review = $data['review'];

    return $checkIn;
}

$stmt = $db->prepare('SELECT id, product_id, review FROM checkin WHERE product_id = :id');
$stmt->execute(['id' => $_GET['id']]);

$checkins = $stmt->fetchAll(PDO::FETCH_ASSOC);

$hydratedCheckins = array_map(static function (array $checkin): CheckIn {
    return hydrateCheckin($checkin);
}, $checkins);

var_dump($hydratedCheckins);
```

```
array (size=2)
  0 =>
    object(CheckIn)[4]
      public int 'id' => int 1
      public int 'productId' => int 1
      public string 'review' => string 'Not keen on this' (length=16)
  1 =>
    object(CheckIn)[5]
      public int 'id' => int 2
      public int 'productId' => int 1
      public string 'review' => string 'Love it!' (length=8)
```

Hydrating Product With Check-Ins

- So far, we have seen how to define functions that hydrate a product or list of check-ins when loaded separately
- For performance reasons, we likely want to load both the product and its check-ins in a single query
- As mentioned, PDO cannot hydrate related entities - that is our job

Why Hydrate Objects?

- Arrays have no defined types – we cannot enforce that a particular property is of an expected type
- The array structure returned by PDO is flat – we want to build a logical data hierarchy
- Object instances can be type-hinted in methods, both as parameter and return types

Defining Our Classes


```php
class CheckIn
{
    public int $id;
    public int $productId;
    public string $review;
}

class Product
{
    public int $id;
    public string $title;
    /** @var CheckIn[] */
    public array $checkins = [];
}

$product = new Product();
$checkin = new CheckIn();
$product->checkins[] = $checkin;
$product->checkins[] = 12345;
$product->checkins[] = 'This is not a check-in?';

var_dump($product);
```

- Each Product instance can now contain multiple CheckIn instances
- There is however an issue – any type can be added to `Product::checkins`

```
object(Product)[1]
  public int 'id' => *uninitialized*
  public string 'title' => *uninitialized*
  public array 'checkins' =>
    array (size=3)
      0 =>
        object(CheckIn)[2]
          public int 'id' => *uninitialized*
          public int 'productId' => *uninitialized*
          public string 'review' => *uninitialized*
      1 => int 12345
      2 => string 'This is not a check-in?' (length=23)
```

Defining Our Classes


```php
class Product
{
    public int $id;
    public string $title;
    /** @var CheckIn[] */
    private array $checkins = [];

    public function addCheckin(CheckIn $checkIn): void
    {
        $this->checkins[] = $checkIn;
    }

    public function getCheckins(): array
    {
        return $this->checkins;
    }
}
```

- To prevent this, we make `Product::checkins` **`private`**, and use methods to populate and retrieve the values
- The method's parameter can be type hinted to only allow CheckIn instances to be provided

Defining Our Classes


```php
$product = new Product();
$checkin = new CheckIn();
$product->addCheckin($checkin);
$product->addCheckin(12345);
$product->addCheckin('This is not a check-in?');
```

Fatal error: Uncaught TypeError: Argument 1 passed to Product::addCheckin() must be an instance of CheckIn, int given, called in /var/www/sites/deeper.local/exercises/week-06/session-2/typed-check-ins.php on line 31 and defined in /var/www/sites/deeper.local/exercises/week-06/session-2/typed-check-ins.php on line *17*

TypeError: Argument 1 passed to Product::addCheckin() must be an instance of CheckIn, int given, called in /var/www/sites/deeper.local/exercises/week-06/session-2/typed-check-ins.php on line 31 in /var/www/sites/deeper.local/exercises/week-06/session-2/typed-check-ins.php on line *17*

Call Stack

#	Time	Memory	Function	Location
1	0.0076	363040	{main}( )	.../typed-check-ins.php:0
2	0.0076	363608	Product->addCheckin( )	.../typed-check-ins.php:31

Defining Our Classes – Finished Structure


```php
class CheckIn
{
    public int $id;
    public int $productId;
    public string $review;
}

class Product
{
    public int $id;
    public string $title;
    /** @var CheckIn[] */
    private array $checkins = [];

    public function addCheckin(CheckIn $checkIn): void
    {
        $this->checkins[] = $checkIn;
    }

    public function getCheckins(): array
    {
        return $this->checkins;
    }
}
```

Hydrating Product With Check-Ins


```php
$stmt = $db->prepare('
  SELECT * FROM product p
  LEFT JOIN checkin c ON c.product_id = p.id
  WHERE p.id = :id
');
```

```
array (size=2)
  0 =>
    array (size=7)
      'id' => string '1' (length=1)
      'title' => string 'Macbook Pro' (length=11)
      'product_id' => string '1' (length=1)
      'name' => string 'John Smith' (length=10)
      'rating' => string '2' (length=1)
      'review' => string 'Not keen on this' (length=16)
      'posted' => string '2020-09-07 09:10:51' (length=19)
  1 =>
    array (size=7)
      'id' => string '2' (length=1)
      'title' => string 'Macbook Pro' (length=11)
      'product_id' => string '1' (length=1)
      'name' => string 'Jane Doe' (length=8)
      'rating' => string '5' (length=1)
      'review' => string 'Love it!' (length=8)
      'posted' => string '2020-09-07 09:10:51' (length=19)
```

- We need a hydrator which converts the above structure into a single Product instance, which contains two CheckIn instances

Hydrating Products With Check-Ins


```php
function hydrateProductWithCheckins(array $data): Product
{
    // Since the Product data exists in each returned row, pull it out of the first
    $product = new Product();
    $product->id = $data[0]['product_id'];
    $product->title = $data[0]['title'];

    // Each individual row returned represents a single check-in
    foreach ($data as $checkinRow) {
        $checkIn = new CheckIn();
        $checkIn->id = $checkinRow['id'];
        $checkIn->review = $checkinRow['review'];
        $checkIn->productId = $checkinRow['product_id'];

        $product->addCheckin($checkIn);
    }

    return $product;
}
```

```
array (size=2)
  0 =>
    array (size=7)
      'id' => string '1' (length=1)
      'title' => string 'Macbook Pro' (length=11)
      'product_id' => string '1' (length=1)
      'name' => string 'John Smith' (length=10)
      'rating' => string '2' (length=1)
      'review' => string 'Not keen on this' (length=16)
      'posted' => string '2020-09-07 09:10:51' (length=19)
  1 =>
    array (size=7)
      'id' => string '2' (length=1)
      'title' => string 'Macbook Pro' (length=11)
      'product_id' => string '1' (length=1)
      'name' => string 'Jane Doe' (length=8)
      'rating' => string '5' (length=1)
      'review' => string 'Love it!' (length=8)
      'posted' => string '2020-09-07 09:10:51' (length=19)
```

```
object(Product)[3]
  public int 'id' => int 1
  public string 'title' => string 'Macbook Pro' (length=11)
  private array 'checkins' =>
    array (size=2)
      0 =>
        object(CheckIn)[4]
          public int 'id' => int 1
          public int 'productId' => int 1
          public string 'review' => string 'Not keen on this' (length=16)
      1 =>
        object(CheckIn)[5]
          public int 'id' => int 2
          public int 'productId' => int 1
          public string 'review' => string 'Love it!' (length=8)
```

02

Architecture Principles

How to write better code

Code Architecture Principles

- There are several principles in software development which, when followed, produces code that is
  - Easier to read and understand
  - Easier to update and refactor
  - Easier to test (more on this in a future session)
- Five of these principles are grouped under the acronym S.O.L.I.D
- Today we will look at S – Single-Responsibility Principle
- https://en.wikipedia.org/wiki/SOLID

SRP – Single Responsibility Principle

- The Single Responsibility Principle states that a single unit of code should have one single job, or responsibility
- Each class should handle one piece of application functionality, and should have **one reason to change**
- The class should encapsulate the logic required to perform that task and only expose what is needed for its use

SRP – Basic Example


```php
class Calculator
{
    public function add(float $a, float $b): void
    {
        $total = $a + $b;

        echo 'The total is ' . $total;
    }

    public function multiply(float $a, float $b): void
    {
        $total = $a * $b;

        echo 'The total is ' . $total;
    }
}
```

- This class, although basic, does not adhere to SRP
- It is currently responsible for both
  - Performing mathematical operations
  - Generating output
- This means the code is less flexible – we may want to use the result in further calculations
- We may want to return the result in a different format

SRP – Basic Example


```php
class Calculator
{
    public function add(float $a, float $b): float
    {
        return $a + $b;
    }

    public function multiply(float $a, float $b): float
    {
        return $a * $b;
    }
}
```

- This class better adheres to SRP
- The class is only responsible for mathematical calculations
- The data returned from it can be used in whatever way the application requires

SRP – Hydrators


```
function hydrateProductWithCheckins(array $data): Product
{
    $product = new Product();
    $product->id = $data[0]['product_id'];
    $product->title = $data[0]['title'];

    foreach ($data as $checkinRow) {
        $checkIn = new CheckIn();
        $checkIn->id = $checkinRow['id'];
        $checkIn->review = $checkinRow['review'];
        $checkIn->productId = $checkinRow['product_id'];

        $product->addCheckin($checkIn);
    }

    return $product;
}
```

- This hydrator function currently handles two relatively distinct tasks
  - Hydrating a Product instance
  - Hydrating a Checkin instance
- Although linked, both tasks are independent and useful without the other in our application
- We can therefore break out the individual tasks into separate functions
- This also leads into another principle - DRY

DRY – Don’t Repeat Yourself!

- DRY is a principle that simply aims to reduce the duplication of logic in applications
- Whenever you find **logic** has been duplicated, it should be abstracted to a shared location, e.g. a class
- The abstracted element can then be included in all locations that requires its logic
- By keeping SRP and DRY in mind when writing code, we start to produce a larger number of smaller components
- Violations of DRY are sometimes referred to as WET – We Enjoy Typing! (or Wasting Everyone’s Time)

Premature Abstraction

- Attempting to be too DRY can however have side effects
- If too much effort is put into abstracting every piece of logic **before it is duplicated anywhere**, the application can become overly complex
- As a rule, only abstract logic to a shared location **once it becomes duplicated**
- For example, we only abstracted the header bar in the project to its own file **when it was needed on 2 pages**

Updating Our Hydrators


```php
function hydrateProduct(array $data): Product
{
    $product = new Product();
    $product->id = $data['id'];
    $product->title = $data['title'];

    return $product;
}

function hydrateCheckin(array $data): CheckIn
{
    $checkIn = new CheckIn();
    $checkIn->id = $data['id'];
    $checkIn->productId = $data['product_id'];
    $checkIn->review = $data['review'];

    return $checkIn;
}

function hydrateProductWithCheckins(array $data): Product
{
    $productData = [
        'id' => $data[0]['product_id'],
        'title' => $data[0]['title'],
    ];
    $product = hydrateProduct($productData);

    foreach ($data as $checkinRow) {
        $checkIn = hydrateCheckin($checkinRow);
        $product->addCheckin($checkIn);
    }

    return $product;
}
```

```php
$stmt = $db->prepare('
  SELECT * FROM product p
  LEFT JOIN checkin c ON c.product_id = p.id
  WHERE p.id = :id
');
$stmt->execute(['id' => $_GET['id']]);

$productData = $stmt->fetchAll(PDO::FETCH_ASSOC);
$product = hydrateProductWithCheckins($productData);

var_dump($product);
```

```
object(Product)[3]
  public int 'id' => int 1
  public string 'title' => string 'Macbook Pro' (length=11)
  private array 'checkins' =>
    array (size=2)
      0 =>
        object(CheckIn)[4]
          public int 'id' => int 1
          public int 'productId' => int 1
          public string 'review' => string 'Not keen on this' (length=16)
      1 =>
        object(CheckIn)[5]
          public int 'id' => int 2
          public int 'productId' => int 1
          public string 'review' => string 'Love it!' (length=8)
```

Hydrator Class

- The previous example split the logic of hydrating Products and CheckIns into separate functions
- We don't however want to pollute our application with global functions – we should create one or more classes
- We could validly create a separate class for each entity we wish to hydrate
- Since for now we only have two, we will create a single class for simplicity

Hydrator Class


```php
class EntityHydrator
{
    public function hydrateProduct(): Product
    {
        // ...
    }

    public function hydrateCheckIn(): CheckIn
    {
        // ...
    }

    public function hydrateProductWithCheckIns(): Product
    {
        // ...
    }
}
```

```php
$productAndCheckInData = []; // i.e. from query...

$hydrator = new EntityHydrator();
$product = $hydrator->hydrateProductWithCheckIns(
  $productAndCheckInData
);
```

KISS – Keep It Simple Stupid!

- KISS is more of a general mindset to maintain than a concrete principle with examples
- Code should always be as simple as possible to understand
- This can mean;
  - Keep code well formatted
  - Use meaningful naming – you don't get points for unnecessarily concise variable/method names
  - Don't always use short-handed alternatives if they increase difficulty of understanding – e.g. ternary statements or combining function calls in a single line