Implementing Repository Pattern in PHP

The Repository Pattern is used in PHP to separate database logic from business logic, making code more maintainable, testable, and scalable.

Step 1: Create a Model (Entity Layer)

This represents a database table. If using Laravel, you can use Eloquent:

```
class Product extends Model {
    protected $fillable = ['name', 'price'];
}
```

For pure PHP, you can create a class:

```
class Product {
    public $id;
    public $name;
    public $price;
}
```

Step 2: Create a Repository Interface

```
interface ProductRepositoryInterface {
    public function getAll();
    public function getById($id);
    public function create(array $data);
    public function update($id, array $data);
    public function delete($id);
}
```

Step 3: Implement the Repository (Database Logic)

For Laravel (Eloquent ORM):

```
class ProductRepository implements ProductRepositoryInterface {
    public function getAll() { return Product::all(); }
    public function getById($id) { return Product::find($id); }
    public function create(array $data) { return Product::create($data); }
```

```
    public function update($id, array $data) {
        $product = Product::find($id);
        if ($product) { $product->update($data); return $product; }
        return null;
    }
    public function delete($id) { return Product::destroy($id); }
}
```

For Pure PHP (PDO Approach):

```
class ProductRepository implements ProductRepositoryInterface {
    private $db;
    public function __construct($pdo) { $this->db = $pdo; }
    public function getAll() { return $this->db->query("SELECT * FROM products")->fetchAll(PDO::FETCH_ASSOC); }
    public function getById($id) {
        $stmt = $this->db->prepare("SELECT * FROM products WHERE id = ?");
        $stmt->execute([$id]);
        return $stmt->fetch(PDO::FETCH_ASSOC);
    }
    public function create(array $data) {
        $stmt = $this->db->prepare("INSERT INTO products (name, price) VALUES (?, ?)");
        return $stmt->execute([$data['name'], $data['price']]);
    }
    public function update($id, array $data) {
        $stmt = $this->db->prepare("UPDATE products SET name = ?, price = ? WHERE id = ?");
        return $stmt->execute([$data['name'], $data['price'], $id]);
    }
    public function delete($id) {
        $stmt = $this->db->prepare("DELETE FROM products WHERE id = ?");
        return $stmt->execute([$id]);
    }
}
```

Step 4: Use the Repository in a Controller

For Laravel:

```
class ProductController extends Controller {
    private $productRepository;
    public function __construct(ProductRepositoryInterface $productRepository) {
```

```
        $this->productRepository = $productRepository;
    }
    public function index() { return response()->json($this->productRepository->getAll()); }
    public function store(Request $request) { return response()->json($this->productRepository->create($request->all())); }
}
```

For Pure PHP:

```
$pdo = new PDO("mysql:host=localhost;dbname=mydb", "root", "");
$productRepo = new ProductRepository($pdo);
$products = $productRepo->getAll();
print_r($products);
```

Step 5: Binding in Laravel (Dependency Injection)

In AppServiceProvider.php:

```
public function register() {
    $this->app->bind(ProductRepositoryInterface::class, ProductRepository::class);
}
```

Advantages of Using the Repository Pattern

- Separation of concerns (business logic and data access are separate).
- Easier testing (mock repository for unit tests).
- Flexibility (switch from MySQL to MongoDB without changing business logic).
- Improved maintainability (cleaner code).