

Structure MVC avec Repository et Services

📁 Architecture des dossiers

```
src/  
├── Controllers/      # Contrôleurs MVC  
│   ├── UserController.php  
│   └── ProductController.php  
├── Models/          # Entités/Modèles  
│   ├── User.php  
│   └── Product.php  
├── Repositories/     # Accès aux données  
│   ├── Interfaces/  
│   │   ├── UserRepositoryInterface.php  
│   │   └── ProductRepositoryInterface.php  
│   ├── UserRepository.php  
│   └── ProductRepository.php  
├── Services/         # Logique métier  
│   ├── UserService.php  
│   ├── ProductService.php  
│   └── EmailService.php  
├── Views/            # Templates/Vues  
│   ├── user/  
│   └── product/  
└── Config/           # Configuration  
    └── Database.php
```

🔗 Flux de données et responsabilités

```
Request → Controller → Service → Repository → Database  
      ↓  
Response ← View ← ──────────┐
```

Responsabilités par couche :

- **Controller** : Reçoit les requêtes, valide les données d'entrée, appelle les services
- **Service** : Contient la logique métier, orchestre les opérations
- **Repository** : Accès aux données uniquement (CRUD)
- **Model** : Représente les entités métier

📋 Exemple concret : Gestion d'utilisateurs

1. Model (Entité)

php

```
// src/Models/User.php
class User
{
    private int $id;
    private string $email;
    private string $password;
    private bool $active;
    private DateTime $createdAt;

    // Constructeur, getters, setters...

    public function setPassword(string $password): void
    {
        $this->password = password_hash($password, PASSWORD_ARGON2ID);
    }
}
```

2. Repository Interface

php

```
// src/Repositories/Interfaces/UserRepositoryInterface.php
interface UserRepositoryInterface
{
    public function findById(int $id): ?User;
    public function findByEmail(string $email): ?User;
    public function save(User $user): User;
    public function delete(int $id): bool;
    public function findAll(int $limit = 50, int $offset = 0): array;
}
```

3. Repository Implémentation

php

```
// src/Repositories/UserRepository.php
class UserRepository implements UserRepositoryInterface
{
    private PDO $pdo;

    public function __construct(PDO $pdo)
    {
        $this->pdo = $pdo;
    }

    public function findById(int $id): ?User
    {
        $stmt = $this->pdo->prepare("SELECT * FROM users WHERE id = ?");
        $stmt->execute([$id]);
        $data = $stmt->fetch(PDO::FETCH_ASSOC);

        return $data ? $this->hydrate($data) : null;
    }

    public function save(User $user): User
    {
        if ($user->getId()) {
            return $this->update($user);
        }
        return $this->create($user);
    }

    private function hydrate(array $data): User
    {
        $user = new User();
        $user->setId($data['id']);
        $user->setEmail($data['email']);
        // ... autres propriétés
        return $user;
    }
}
```

4. Service (Logique métier)

php

```
// src/Services/UserService.php
class UserService
{
    private UserRepositoryInterface $userRepository;
    private EmailService $emailService;

    public function __construct(
        UserRepositoryInterface $userRepository,
        EmailService $emailService
    ) {
        $this->userRepository = $userRepository;
        $this->emailService = $emailService;
    }

    public function registerUser(string $email, string $password): User
    {
        // Validation métier
        if ($this->userRepository->findByEmail($email)) {
            throw new UserAlreadyExistsException("Email déjà utilisé");
        }

        if (strlen($password) < 8) {
            throw new InvalidPasswordException("Mot de passe trop court");
        }

        // Création de l'utilisateur
        $user = new User();
        $user->setEmail($email);
        $user->setPassword($password);
        $user->setActive(false);
        $user->setCreatedAt(new DateTime());

        // Sauvegarde
        $user = $this->userRepository->save($user);

        // Actions complémentaires
        $this->emailService->sendWelcomeEmail($user);

        return $user;
    }

    public function authenticateUser(string $email, string $password): ?User
    {
        $user = $this->userRepository->findByEmail($email);

        if (!$user || !password_verify($password, $user->getPassword())) {
```

```
        return null;
    }

    if (!$user->isActive()) {
        throw new InactiveUserException("Compte non activé");
    }

    return $user;
}
}
```

5. Controller

php

```
// src/Controllers/UserController.php
class UserController
{
    private UserService $userService;

    public function __construct(UserService $userService)
    {
        $this->userService = $userService;
    }

    public function register(): void
    {
        try {
            $email = $_POST['email'] ?? '';
            $password = $_POST['password'] ?? '';

            // Validation des entrées
            if (empty($email) || empty($password)) {
                throw new InvalidInputException("Email et mot de passe requis");
            }

            // Appel du service
            $user = $this->userService->registerUser($email, $password);

            // Réponse
            $this->renderJson([
                'success' => true,
                'message' => 'Utilisateur créé avec succès',
                'user_id' => $user->getId()
            ]);

        } catch (Exception $e) {
            $this->renderJson([
                'success' => false,
                'message' => $e->getMessage()
            ], 400);
        }
    }
}
```

Injection de dépendances

Container simple

php

```
// src/Config/Container.php
class Container
{
    private array $services = [];

    public function register(): void
    {
        // Database
        $this->services[PDO::class] = function() {
            return new PDO(
                "mysql:host=localhost;dbname=myapp",
                "username",
                "password"
            );
        };

        // Repositories
        $this->services[UserRepositoryInterface::class] = function() {
            return new UserRepository($this->get(PDO::class));
        };

        // Services
        $this->services[EmailService::class] = function() {
            return new EmailService();
        };

        $this->services[UserService::class] = function() {
            return new UserService(
                $this->get(UserRepositoryInterface::class),
                $this->get(EmailService::class)
            );
        };

        // Controllers
        $this->services[UserController::class] = function() {
            return new UserController($this->get(UserService::class));
        };
    }

    public function get(string $class)
    {
        if (!isset($this->services[$class])) {
            throw new Exception("Service $class non trouvé");
        }

        return $this->services[$class]();
    }
}
```

✓ Avantages de cette architecture

Séparation des responsabilités

- **Repository** : Uniquement l'accès aux données
- **Service** : Logique métier complexe
- **Controller** : Gestion des requêtes/réponses

Testabilité

```
php

// Test unitaire du service
class UserServiceTest extends PHPUnit\Framework\TestCase
{
    public function testRegisterUser()
    {
        $mockRepo = $this->createMock(UserRepositoryInterface::class);
        $mockEmail = $this->createMock(EmailService::class);

        $userService = new UserService($mockRepo, $mockEmail);

        // Tests...
    }
}
```

Flexibilité

- Changement de base de données → modifier uniquement le Repository
- Ajout de logique métier → modifier uniquement le Service
- Nouvelle interface → modifier uniquement le Controller

🎯 Règles à respecter

1. **Controller** ne fait jamais de requêtes SQL directes
2. **Repository** ne contient aucune logique métier
3. **Service** ne connaît pas les détails de stockage
4. Utilisez les **interfaces** pour le découplage
5. **Injection de dépendances** partout
6. **Une responsabilité par classe**

Cette architecture garantit un code maintenable, testable et évolutif pour votre projet MVC.