



# DOSSIER PROFESSIONNEL (DP)

*Nom de naissance* ▶ JEAN-LOUIS  
*Nom d'usage* ▶ JEAN-LOUIS  
*Prénom* ▶ Jules  
*Adresse* ▶ 4 montée de la belle france, 13015 Marseille

## Titre professionnel visé

Concepteur, Développeur d'Applications

### MODALITÉ D'ACCÈS :

- Parcours de formation
- Validation des Acquis de l'Expérience (VAE)

# DOSSIER PROFESSIONNEL (DP)

## Présentation du dossier

Le dossier professionnel (DP) constitue un élément du système de validation du titre professionnel.  
**Ce titre est délivré par le Ministère chargé de l'emploi.**

Le DP appartient au candidat. Il le conserve, l'actualise durant son parcours et le présente **obligatoirement à chaque session d'examen.**

Pour rédiger le DP, le candidat peut être aidé par un formateur ou par un accompagnateur VAE.  
Il est consulté par le jury au moment de la session d'examen.

### Pour prendre sa décision, le jury dispose :

1. des résultats de la mise en situation professionnelle complétés, éventuellement, du questionnaire professionnel ou de l'entretien professionnel ou de l'entretien technique ou du questionnement à partir de productions.
2. du **Dossier Professionnel (DP)** dans lequel le candidat a consigné les preuves de sa pratique professionnelle.
3. des résultats des évaluations passées en cours de formation lorsque le candidat évalué est issu d'un parcours de formation
4. de l'entretien final (dans le cadre de la session titre).

*[Arrêté du 22 décembre 2015, relatif aux conditions de délivrance des titres professionnels du ministère chargé de l'Emploi]*

### Ce dossier comporte :

- pour chaque activité-type du titre visé, un à trois exemples de pratique professionnelle ;
- un tableau à renseigner si le candidat souhaite porter à la connaissance du jury la détention d'un titre, d'un diplôme, d'un certificat de qualification professionnelle (CQP) ou des attestations de formation ;
- une déclaration sur l'honneur à compléter et à signer ;
- des documents illustrant la pratique professionnelle du candidat (facultatif)
- des annexes, si nécessaire.

# DOSSIER PROFESSIONNEL (DP)

Pour compléter ce dossier, le candidat dispose d'un site web en accès libre sur le site.

 <http://travail-emploi.gouv.fr/titres-professionnels>

## Sommaire

### Exemples de pratique professionnelle

<b>Développer une application sécurisée</b>	<b>p.</b>	<b>5</b>
▸ SafeBase - sauvegarde de base de données	p.	p.
▸ Intitulé de l'exemple n° 2	p.	p.
▸ Intitulé de l'exemple n° 3	p	p.
<b>Intitulé de l'activité-type n° 2</b>	<b>p.</b>	
▸ Intitulé de l'exemple n° 1	p.	p.
▸ Intitulé de l'exemple n° 2	p.	p.
▸ Intitulé de l'exemple n° 3	p	p.
<b>Intitulé de l'activité-type n° 3</b>	<b>p.</b>	
▸ Intitulé de l'exemple n° 1	p.	p.
▸ Intitulé de l'exemple n° 2	p.	p.
▸ Intitulé de l'exemple n° 3	p	p.
<b>Titres, diplômes, CQP, attestations de formation (facultatif)</b>	<b>p.</b>	
<b>Déclaration sur l'honneur</b>	<b>p.</b>	
<b>Documents illustrant la pratique professionnelle (facultatif)</b>	<b>p.</b>	
<b>Annexes (Si le RC le prévoit)</b>	<b>p.</b>	

**DOSSIER PROFESSIONNEL** <sup>(DP)</sup>

---

# **EXEMPLES DE PRATIQUE PROFESSIONNELLE**

# DOSSIER PROFESSIONNEL (DP)

## Activité-type 1 Développer une application sécurisée

Exemple n°1 - SafeBase - sauvegarde de base de données

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre de ma formation CDA, j'ai participé au projet SafeBase, une plateforme de sauvegarde et de restauration automatisée de bases de données. L'architecture du projet est organisée en trois niveaux : un backend en Go, un frontend en Angular, et des bases de données MySQL et PostgreSQL conteneurisées via Docker.

Le projet a été développé en binôme sur une période de trois semaines. Je me suis principalement occupé de la mise en place de l'environnement de développement, de la configuration des conteneurs, et de la documentation technique liée au lancement de l'application.

J'ai tout d'abord mis en place l'environnement de développement backend avec Go 1.21 et les modules de gestion de dépendances (go.mod/go.sum) ainsi qu'Air configuré via air.toml pour le hot reload (rechargement à chaud).

Configuration Air (air.toml) :

```
[build]
cmd = "go build -o ./tmp/main ."
bin = "./tmp/main"
include_ext = ["go", "tpl", "tmpl", "html"]
exclude_regex = ["_test.go"]
stop_on_error = true

[color]
build = "yellow"
main = "magenta"
runner = "green"
```

Côté frontend, j'ai configuré Angular CLI avec un écosystème NPM complet incluant TypeScript, PrimeNG pour l'UI, et Prettier pour le formatage automatique du code.

Nous avons également utilisé Git pour le contrôle de version du code, et GitHub pour héberger le code versionné. Nous avons suivi une procédure de Git Flow avec la branche dev protégée et toutes les fonctionnalités mergées sur dev via des Pull Requests, tandis que main reste à jour et correspond à la dernière release de version.

# DOSSIER PROFESSIONNEL (DP)

J'ai ensuite pu mettre en place les différents services de mon docker-compose. Le backend et le frontend ont des fichiers Dockerfile qui sont une suite d'instructions permettant de construire mes conteneurs. J'ai notamment implémenté un Dockerfile multi-stage pour avoir une version développement avec toutes les dépendances nécessaires, mais dispensables en production. L'image est construite à partir de la configuration de développement et ne copie que l'indispensable pour la production, rendant l'image moins lourde en vue d'un déploiement sur serveur. (voir le Dockerfile en annexe)

Puis mon docker-compose avec mes différents services :

```
services:
  backend:
    build:
      context: ../backend
      dockerfile: Dockerfile
      target: ${NODE_ENV:-development}
    ports:
      - "${BACKEND_PORT}:8080"
    environment:
      - DATABASE_URL=${DATABASE_URL}
    depends_on:
      safebase_db:
        condition: service_healthy

  frontend:
    build:
      context: ../frontend
      dockerfile: Dockerfile
      target: ${NODE_ENV:-development}
    ports:
      - "${FRONTEND_PORT}:4200"

  safebase_db:
    image: postgres:16
    ports:
      - "${DB_PORT}:5432"
```

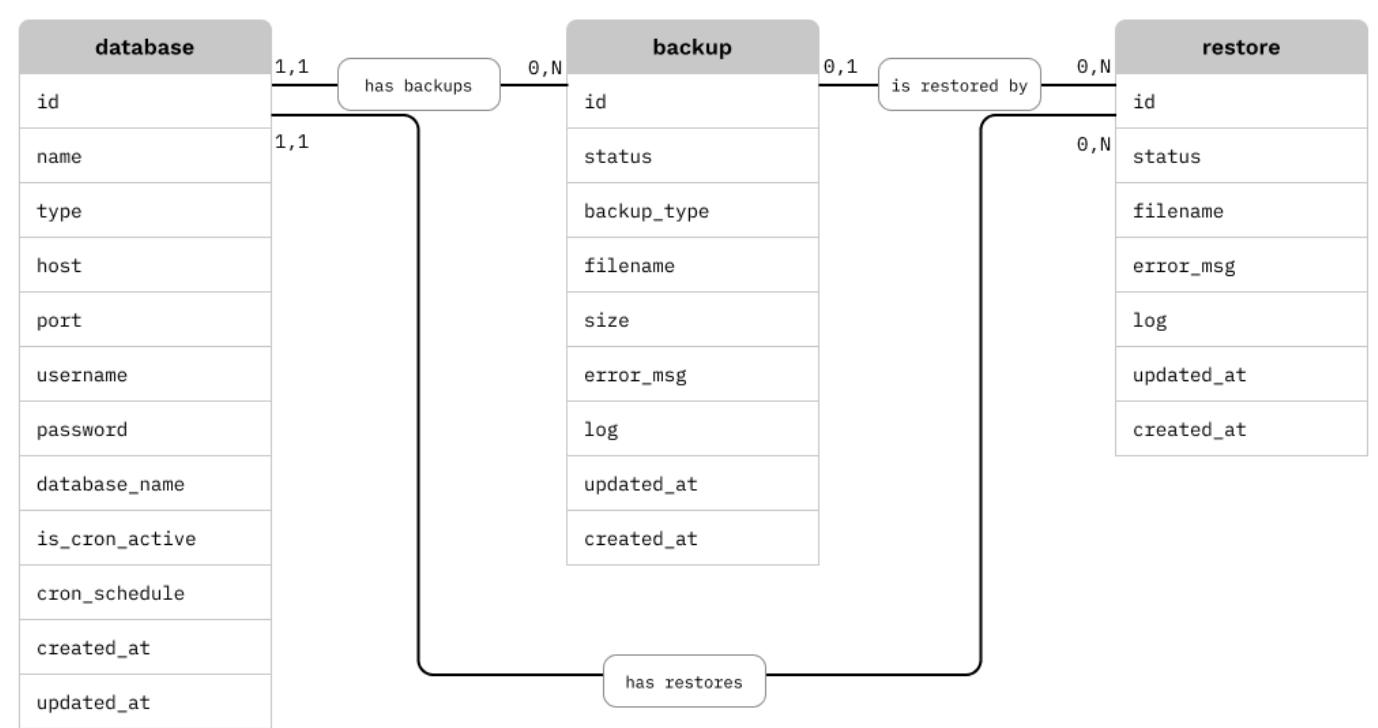
# DOSSIER PROFESSIONNEL (DP)

```
environment:  
  POSTGRES_USER: ${DB_USER}  
  POSTGRES_PASSWORD: ${DB_PASSWORD}  
  POSTGRES_DB: ${DB_NAME}  
  
healthcheck:  
  test: ["CMD=READY", "pg_isready"]
```

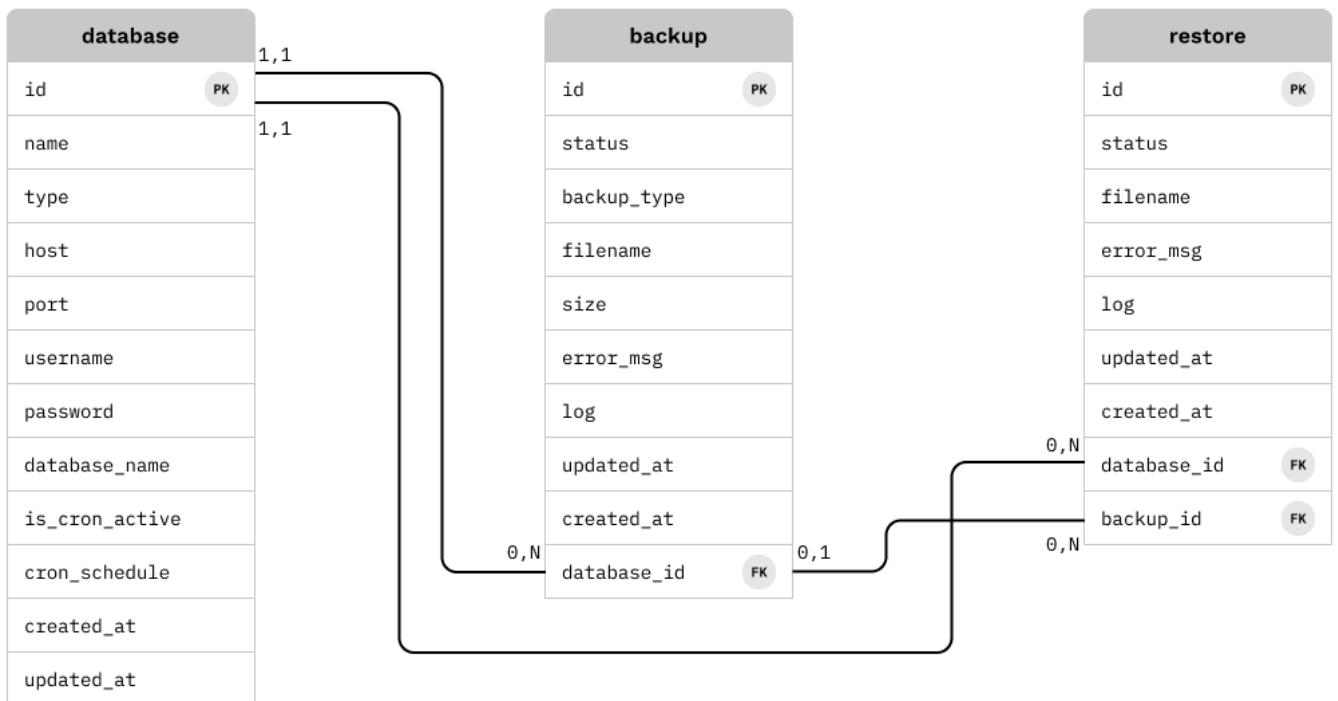
Pour les configurations, j'ai tout centralisé dans un fichier **.env**. C'est là que je stocke toutes les variables d'environnement nécessaires. Ce fichier n'est **pas versionné**, ce qui est crucial pour ne pas exposer de données sensibles comme des clés API ou des identifiants de base de donnée. À la racine du projet, j'ai inclus un fichier **.env.example** qui sert de modèle. J'a aussi documenté l'installation du projet et son utilisation dans un fichier [README.md](#).

Avant de me lancer dans le développement du backend, j'ai d'abord posé les bases de ma **base de données**. Mon but, c'était de pouvoir y **stocker les infos de mes différentes bases de données**, mais aussi d'avoir une **liste de tous les backups faits** et des **restaurations lancées** (qu'elles viennent d'une sauvegarde interne ou externe). Pour ça, j'ai commencé par faire le **MCD (Modèle Conceptuel de Données)**. J'ai défini les **entités principales de l'application**, leurs **relations**, les **cardinalités** et les **verbes** qui décrivent ces interactions :

# DOSSIER PROFESSIONNEL (DP)



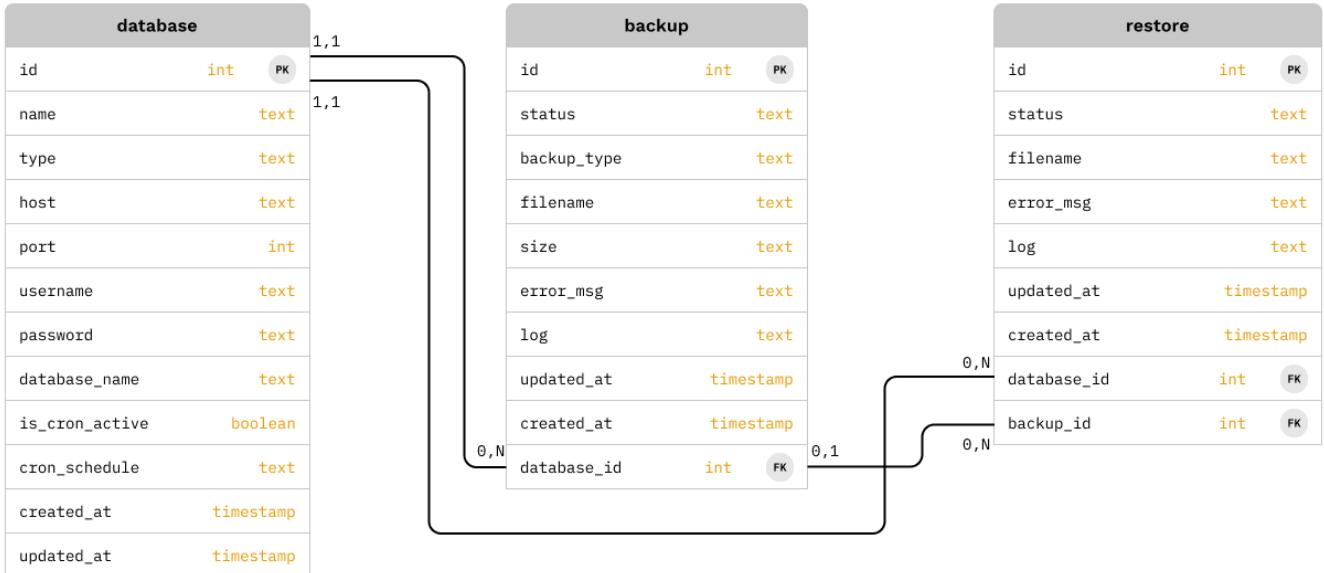
Ensuite, j'ai transformé le MCD en MLD. Ici, les entités deviennent des **tables** et les relations sont matérialisées par des **clés étrangères**.



Enfin, j'ai élaboré le MPD. C'est la représentation la plus proche du schéma réel qui serait implémenté

# DOSSIER PROFESSIONNEL (DP)

dans ma base de données PostgreSQL. À ce stade, j'ai détaillé le **type de données**.



Pour le développement de la logique métier du backend, j'ai implémenté une architecture respectant les principes SOLID et les bonnes pratiques de la programmation orientée objet en Go. Bien que Go ne soit pas strictement orienté objet, j'ai appliqué des concepts similaires via les structures et interfaces pour créer un code maintenable.

Pattern Repository avec Injection de Dépendance :

```

type DatabaseService struct {
    DB *gorm.DB
}

// Factory pattern pour l'initialisation
func NewDatabaseService() *DatabaseService {
    return &DatabaseService{
        DB: db.GetDB(),
    }
}

// CreateDatabase permet de créer une nouvelle entrée dans la table Database
// Méthodes métier avec responsabilité unique (Single Responsibility Principle)
func (s *DatabaseService) CreateDatabase(
    name string,
    dbType string,
    host string,
    port string,
)
    
```

# DOSSIER PROFESSIONNEL (DP)

```
username string,
password string,
databaseName string,
isCronActive bool,
cronSchedule string,
) (*model.Database, error) {
    // Validation et création de l'entité métier
    database := &model.Database{
        Name:           name,
        Type:          dbType,
        Host:          host,
        Port:          port,
        Username:      username,
        Password:      password,
        DatabaseName: databaseName,
        IsCronActive:  isCronActive,
        CronSchedule: cronSchedule,
    }

    // Persistance avec gestion d'erreur
    result := s.DB.Create(database)
    if result.Error != nil {
        return nil, result.Error
    }
    // Retourne l'objet Database créé
    return database, nil
}
```

J'ai mis en place plusieurs couches de sécurité pour protéger les composants serveur. La configuration CORS sécurise les communications cross-origin, la validation des connexions base de données empêche les configurations erronées, et la gestion centralisée des erreurs évite l'exposition d'informations sensibles.

```
// Middleware CORS avec contrôle strict des origines
router.Use(cors.New(cors.Config{
    AllowOrigins:     []string{"http://localhost:4200", "http://frontend:4200"},
    AllowMethods:     []string{"GET", "POST", "PUT", "DELETE", "OPTIONS"},
    AllowHeaders:     []string{"Origin", "Content-Type", "Accept", "Authorization"},
    ExposeHeaders:    []string{"Content-Length"},
    AllowCredentials: true,
    MaxAge:          12 * time.Hour,
}))
```

# DOSSIER PROFESSIONNEL (DP)

J'ai aussi implémenté une méthode permettant de vérifier qu'une connexion à une base de données est valide, avant de l'ajouter.

J'ai implémenté des tests unitaires complets pour valider le comportement des composants métier. Les tests couvrent les scénarios nominaux et d'erreur, avec une approche TDD (Test-Driven Development) pour garantir la fiabilité du code (voir l'annexe). De plus, j'ai utilisé le pattern Factory pour l'initialisation des services, respectant le principe d'inversion de dépendance (DIP) du SOLID. Cela me permet de créer des services réutilisables que ce soit dans mes fonctions ou dans les tests.

Pour la sécurité, Go intègre un typage fort, que j'utilise que ce soit au niveau de l'ORM, ou des fonctions au niveau des controllers, ce qui me permet de m'assurer que des données erronées ou malveillantes ne soient pas insérées.

Voici un exemple concret d'une route API qui illustre l'intégration de tous ces principes :

```
api := router.Group("/api")
{
    api.POST("/database", func(c *gin.Context) {
        database.AddDatabase(c, cronService)
    })

    api.GET("/databases", func(c *gin.Context) {
        database.GetAllDatabases(c)
    })

    api.PUT("/database", func(c *gin.Context) {
        database.UpdateDatabase(c, cronService)
    })

    api.DELETE("/database/:id", func(c *gin.Context) {
        database.DeleteDatabase(c, cronService)
    })
}
```

Cette route, les controllers, l'accès à la base de donnée, illustre les principes SOLID mis en place :

- S (Single Responsibility) : chaque fonction a une responsabilité unique
- O (Open/Closed) : extensible via les interfaces
- L (Liskov Substitution) : les services respectent leurs contrats
- I (Interface Segregation) : interfaces spécialisées par domaine
- D (Dependency Inversion) : injection de dépendances via les factories

# DOSSIER PROFESSIONNEL (DP)

L'ensemble garantit une API robuste, sécurisée et maintenable respectant les standards de l'industrie.

Pour le développement du dashboard, j'ai utilisé angular 17 est la bibliothèque de composant PrimeNG.

Les composant son standalone, ce qui facilte les test est la maintenance.

J'ai pu ensuite structure et map les différentes routes de mon applications :

```
export const routes: Routes = [
  { path: '', component: DashboardComponent },
  { path: 'backups', component: BackupComponent },
  { path: 'databases', component: DatabaseComponent },
  { path: 'executions', component: ExecutionComponent },
];
```

J'ai pu m'attelait au composant Dashboard pour afficher les différentes métrique sur la page d'accueil, en utilisant les observables RxJS pour la gestion asynchrone des données et le binding bidirectionnel pour une mise à jour en temps réel.

Composant Dashboard (dashboard.component.ts) :

```
@Component({
  selector: 'app-dashboard',
  standalone: true,
  imports: [CardModule, NgIf],
  templateUrl: './dashboard.component.html',
  providers: [DashboardService]
})
export class DashboardComponent implements OnInit {
  dashboardData: any;

  constructor(private dashboardService: DashboardService) {}

  ngOnInit(): void {
    this.dashboardService.getDashboardData().subscribe({
      next: (data) => {
        this.dashboardData = data;
      },
      error: (error) => {
        console.error('Erreur lors du chargement du dashboard:', error);
      }
    });
  }
}
```

Ensuite j'ai puis créé la structure HTML de cette page

```
<div class="flex justify-content-between gap-3">
```

# DOSSIER PROFESSIONNEL (DP)

```
<p-card header="Databases :" class="dashboard-card">
  <ng-template pTemplate="title">
    {{ dashboardData?.Databases?.Total }}
  </ng-template>
  <div class="p-card-content">
    <p class="m-0">MySQL: {{ dashboardData?.Databases?.Mysql }}</p>
    <p class="m-0">Postgres: {{ dashboardData?.Databases?.Postgres }}</p>
  </div>
</p-card>
<p-card header="Backups :" class="dashboard-card">
  <ng-template pTemplate="title">
    {{ dashboardData?.Backups?.Total }}
  </ng-template>
  <div class="p-card-content">
    <p class="m-0">Successful: {{ dashboardData?.Backups?.Successful }}</p>
    <p class="m-0">Failed: {{ dashboardData?.Backups?.Failed }}</p>
  </div>
</p-card>
</div>
```

Pour l'ajouté de base de données dans l'application, j'ai créé un composant modal avec un formulaire avec tous les paramètres nécessaire pour se connecter à la base, en plus de pouvoir teste la connexion au préalable. J'utilise aussi les validateurs d'Angular pour afficher des messages d'erreur et de confirmation contextuels.

```
@Component({
  selector: 'app-add-database-dialog',
  standalone: true,
  imports: [ReactiveFormsModule, MatDialogModule, MatButtonModule, InputTextModule, ToastModule],
  providers: [MessageService]
})
export class AddDatabaseDialogComponent implements OnInit {
  @Output() databaseAdded = new EventEmitter<void>();
  databaseForm: FormGroup = new FormGroup({ });
  visible: boolean = false;

  constructor(
    private databaseService: DatabaseService,
    private messageService: MessageService
  ) {}
```

# DOSSIER PROFESSIONNEL (DP)

```
ngOnInit() {
  this.databaseForm = new FormGroup({
    name: new FormControl('', Validators.required),
    type: new FormControl('', Validators.required),
    host: new FormControl('', Validators.required),
    port: new FormControl('', [Validators.required, Validators.pattern(/^\d+$/)]),
    username: new FormControl('', Validators.required),
    password: new FormControl(''),
    database_name: new FormControl('', Validators.required),
  });
}

onSubmit() {
  if (this.databaseForm.valid) {
    this.databaseService.addDatabase(this.databaseForm.value).subscribe({
      next: (data) => {
        this.databaseAdded.emit();
        this.messageService.add({
          severity: 'success',
          summary: 'Succès',
          detail: 'Base de données ajoutée avec succès'
        });
        this.visible = false;
      },
      error: (error) => {
        this.messageService.add({
          severity: 'error',
          summary: 'Erreur',
          detail: error.error.error
        });
      }
    });
  } else {
    this.markFormGroupTouched();
  }
}
}
```

Pour la communication avec l'API du backend en Go, j'ai implémenté une couche de services Angular utilisant HttpClient pour la communication avec l'API backend. Chaque service encapsule la logique d'accès aux données avec gestion d'erreurs centralisée.

Service Database (database.service.ts) :

```
@Injectable({
  providedIn: 'root'
```

# DOSSIER PROFESSIONNEL (DP)

```
} )

export class DatabaseService {
  constructor(private http: HttpClient) {}

  getDatabases(): Observable<any> {
    return this.http.get<any>('/api/databases');
  }

  addDatabase(database: any): Observable<any> {
    return this.http.post<any>('/api/database', database);
  }

  testConnection(database: any): Observable<any> {
    const params = new HttpParams()
      .set('host', database.host)
      .set('port', database.port)
      .set('username', database.username)
      .set('password', database.password)
      .set('dbName', database.database_name)
      .set('dbType', database.type);

    return this.http.get<any>('/api/database/test', { params });
  }
}
```

Pour la navigation et l'amélioration de l'expérience utilisateur, j'ai développé un composant sidebar responsive avec navigation par icônes utilisant Lucide Angular. La navigation s'adapte automatiquement selon la taille d'écran et fournit un feedback visuel sur la page active.

Composant Sidebar (sidebar.component.html) :

```
<nav class="sidebar">
  <ul class="nav_top">
    <li class="nav_items">
      <a routerLink="/" routerLinkActive="active"
         [routerLinkActiveOptions]="{ exact: true }"
         class="nav_links">
        <lucide-icon name="home"></lucide-icon>
        <span>Dashboard</span>
      </a>
    </li>
```

# DOSSIER PROFESSIONNEL (DP)

```
<li class="nav_items">
  <a routerLink="/database"
     routerLinkActive="active"
     class="nav_links">
    <lucide-icon name="database"></lucide-icon>
    <span>Databases</span>
  </a>
</li>
</ul>
</nav>
```

Ce projet m'a permis d'acquérir la compétence de l'activité type 1 :

- Installer et configurer son environnement de travail en fonction du projet
- Développer des interfaces utilisateur (composants Angular, formulaires)
- Développer des composants métier
- Concevoir et mettre en place une base de données relationnelle
- Développer des composants d'accès aux données SQL

## 2. Précisez les moyens utilisés :

Les moyens utilisés pour ce projet sont :

**VS Code** : Cet environnement de développement intégré (IDE) a été l'outil principal pour l'écriture et le débogage du code.

**Figma** : Cet outil de conception graphique a servi à la réalisation du logo et, plus largement, à la conception des maquettes et prototypes de l'interface utilisateur (UI) de l'application mobile.

**Go 1.23** : Ce langage de programmation a été choisi pour le développement du backend de l'application, notamment pour la création des APIs et des services.

**Git et GitHub** : **Git** a été utilisé comme système de contrôle de version distribué pour le suivi des modifications du code. **GitHub** a servi de plateforme de collaboration et d'hébergement pour les dépôts de code source, facilitant la gestion et le partage du projet.

**PostgreSQL et MySQL** : Ces systèmes de gestion de base de données relationnelle (SGBDR) ont été employés pour la persistance et l'organisation des données de l'application.

**Docker** : Cet outil de conteneurisation a permis de créer des environnements isolés et portables pour l'application et ses services (base de données, backend). Il a grandement facilité le développement, le test et le déploiement du projet.

## 3. Avec qui avez-vous travaillé ?

J'ai travail seul sur ce projet.

## 4. Contexte

# DOSSIER PROFESSIONNEL (DP)

Nom de l'entreprise, organisme ou association ▶ *Centre de formation - La Plateforme*

Chantier, atelier, service ▶

Période d'exercice ▶ Du : 02/04/2024 au : 21/05/2024

## 5. Informations complémentaires (*facultatif*)

# DOSSIER PROFESSIONNEL (DP)

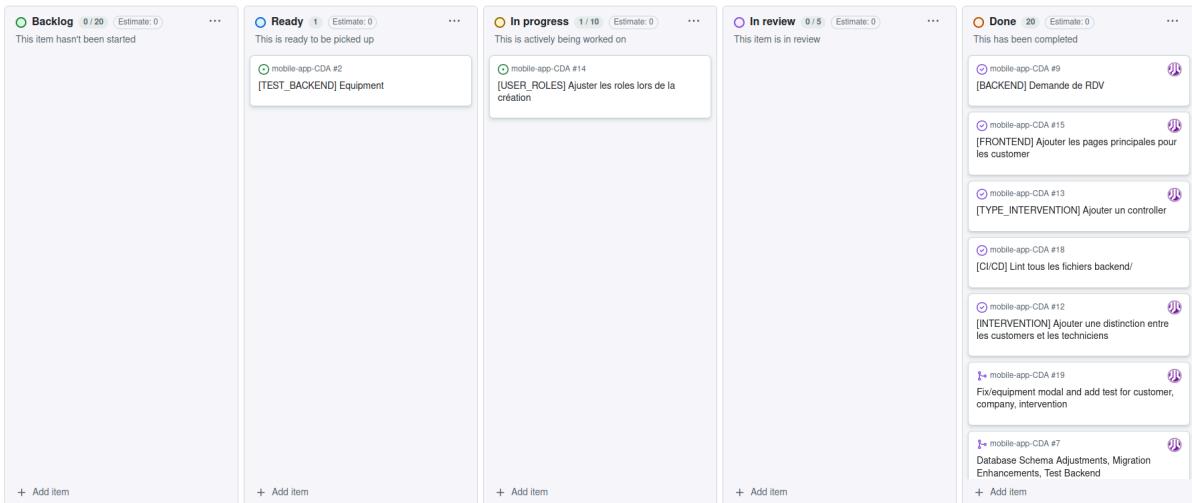
## Activité-type 1 Développer une application sécurisée

*Exemple n° 2 - Contribuer à la gestion d'un projet informatique - Proxifix*

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre du développement de notre projet, j'ai activement participé à la gestion et au suivi en adoptant une approche **Kanban**, inspirée du Manifeste Agile. Plutôt que d'utiliser des cycles de développement rigides, cette méthode nous a permis de maintenir un flux de travail continu et flexible.

Pour organiser le projet, nous avons utilisé un tableau **Kanban sur GitHub**. Ce tableau était structuré en colonnes représentant les étapes clés de notre processus de développement :

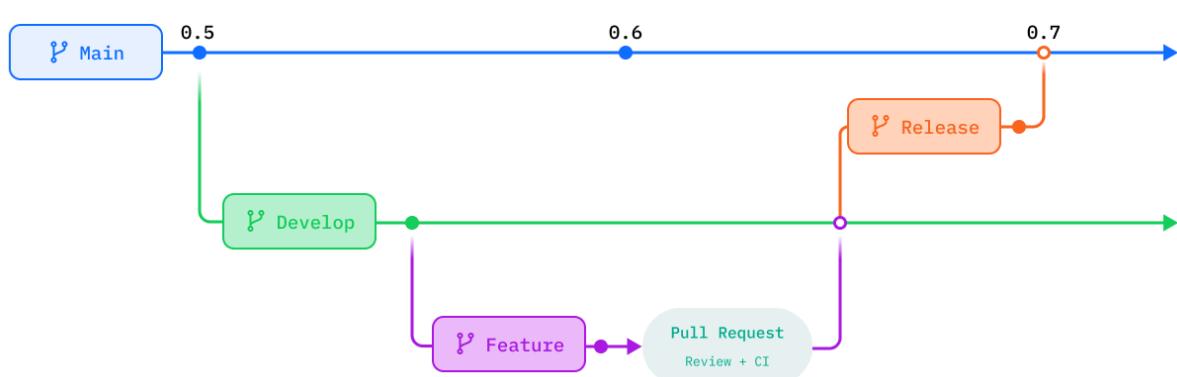


- **Backlog** : Le réservoir d'idées et de tâches à prioriser.
- **À faire (Ready)** : Les tâches prêtes à être démarrées, après une phase d'analyse et de conception.
- **En cours (In Progress)** : Les tâches en cours de développement, avec une limite de travail pour maintenir notre concentration.
- **Révision de code (In Review)** : Une étape de relecture et de tests pour garantir la qualité du code.
- **Terminé (Done)** : Les tâches validées et finalisées.

Cette approche a favorisé une collaboration transparente et nous a permis de réajuster rapidement nos priorités face aux imprévus, tout en garantissant le respect de nos délais.

### Stratégie de versionnement Git

# DOSSIER PROFESSIONNEL (DP)



Pour la gestion du code source, nous avons mis en place une stratégie de versionnement basée sur **Git Flow**. Nous utilisions deux branches principales et protégées : **main** pour les versions stables et **develop** pour le travail en cours. Les nouvelles fonctionnalités et les correctifs étaient développés sur des branches dédiées avant d'être fusionnés dans **develop**, assurant ainsi une séparation claire entre le développement et les versions livrables.

Cette gestion rigoureuse a permis d'assurer le bon déroulement du projet, de maintenir la qualité du code et de favoriser une collaboration efficace au sein de l'équipe.

## 2. Précisez les moyens utilisés :

**GitHub Issues, GitHub Projects, Git et GitHub**

## 3. Avec qui avez-vous travaillé ?

J'ai travaillé avec Axel Vair, un camarade de formation.

## 4. Contexte

Nom de l'entreprise, organisme ou association ▶

Centre de formation - La Plateforme

Chantier, atelier, service ▶

Période d'exercice ▶ Du : 06/01/25 au : 01/07/25

## 5. Informations complémentaires (*facultatif*)

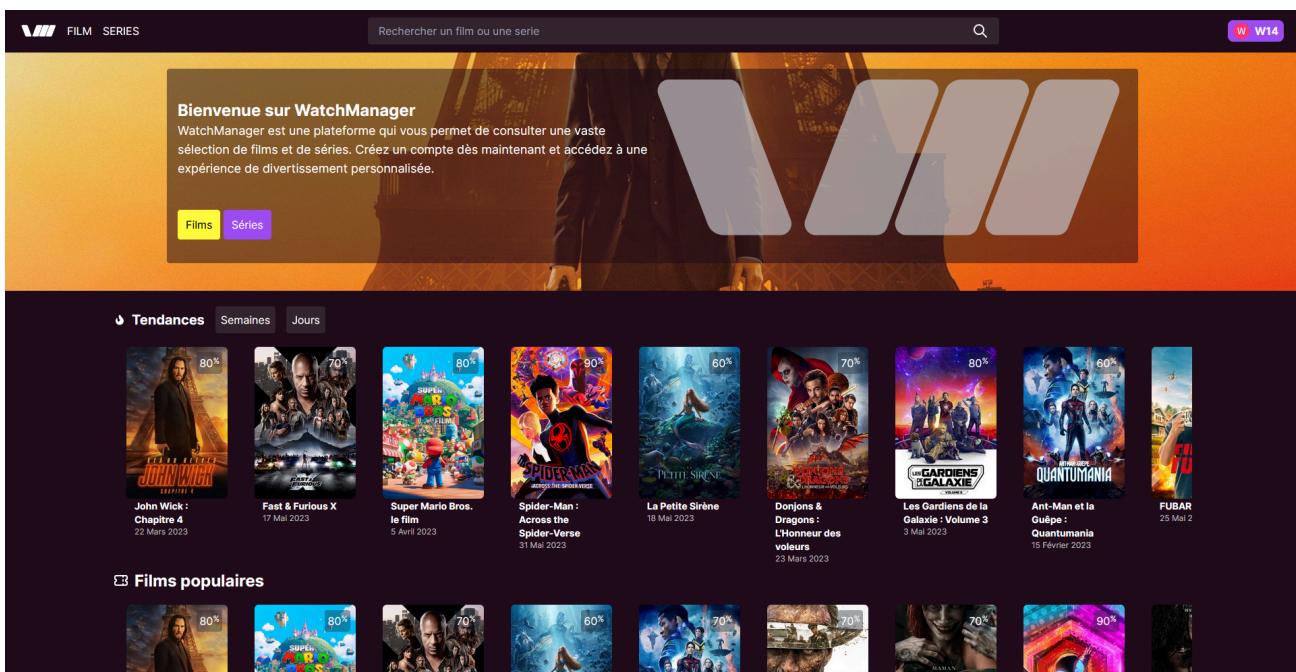
# DOSSIER PROFESSIONNEL (DP)

## Activité-type 1 Cliquez ici pour entrer l'intitulé de l'activité

Exemple n° 3 - Cinetech - API The Movie Database

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre de ma formation, il m'a été demandé de réaliser un projet de développement web en deux semaines. J'ai choisi de créer **Cinetech**, un site internet dynamique qui interroge une API (Application Programming Interface) publique pour cataloguer des films et des séries.



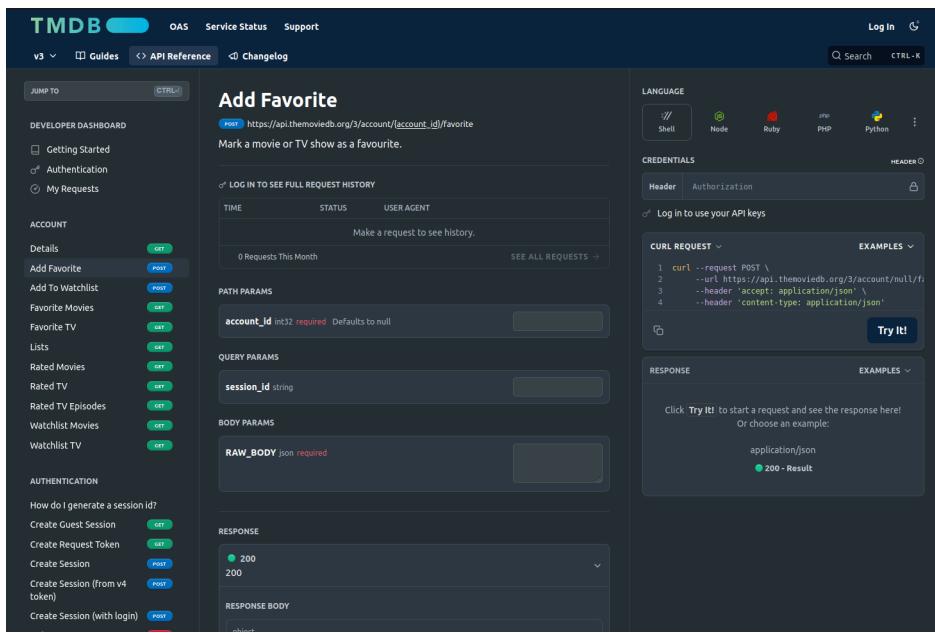
L'objectif était double :

- Mettre en pratique les concepts de l'architecture **MVC (Modèle-Vue-Contrôleur)**, l'utilisation d'un routeur (**AltoRouter**) et d'un gestionnaire de dépendances (**Composer**).
- Appréhender concrètement l'utilisation d'une API externe pour enrichir une application web, en s'inspirant de plateformes populaires comme Allociné.

Pour synthétiser, une API est une interface, une sorte de pont entre deux applications qui permet un échange d'informations. Le développeur utilise l'API sans avoir à coder ni à stocker les informations lui-même. Il utilise des "portes d'entrée" (endpoints) pour récupérer les données dont il a besoin.

### The Movie Database (TMDB)

# DOSSIER PROFESSIONNEL (DP)



Pour ce projet, j'ai utilisé l'API publique de **The Movie Database (TMDB)**. C'est une ressource extrêmement riche qui fournit des informations sur les films, les séries, les acteurs et l'ensemble des métiers du cinéma.

Bien que publique, l'API TMDB nécessite une authentification via un **Token Bearer**. Ce jeton d'authentification permet à l'API d'identifier l'application qui effectue la requête et d'appliquer des limites d'utilisation pour éviter une surcharge de leurs services.

L'architecture back-end repose sur le modèle **MVC** pour garantir une séparation claire des responsabilités et une meilleure maintenabilité du code.

- **Modèle** : Contient la logique métier et les interactions avec la base de données (**UserManager**, **CommentManager**, etc.). Une classe abstraite **AbstractDatabase** centralise la connexion à la base de données MySQL.
- **Vue** : Gère l'affichage et la présentation des données à l'utilisateur (fichiers **.php** dans **src/View/**).
- **Contrôleur** : Fait le lien entre le modèle et la vue. Il reçoit les requêtes de l'utilisateur, interagit avec le modèle et renvoie la vue appropriée (**HomeController**, **MovieController**, etc.).

Le routage est géré par la bibliothèque **AltoRouter**, qui associe les URLs à des méthodes spécifiques dans les contrôleurs. **Composer** est utilisé pour gérer les dépendances du projet et pour l'autoloading des classes, ce qui simplifie l'organisation du code.

La partie front-end est rendue dynamique grâce à **JavaScript**. J'ai utilisé l'**API Fetch** native pour effectuer des requêtes asynchrones vers les différentes APIs (TMDB et Cinetech), ce qui a permis

# DOSSIER PROFESSIONNEL (DP)

d'éviter les rechargements de pages et d'offrir une meilleure expérience utilisateur.

Le design a été réalisé avec **Tailwind CSS**, un framework "utility-first" qui a permis de construire une interface moderne et entièrement responsive de manière rapide et efficace.

## 2. Précisez les moyens utilisés :

**Langages** : PHP, JavaScript, HTML, CSS, SQL.

**Frameworks & Bibliothèques** : Tailwind CSS, AltoRouter.

**Outils de développement** : PHPStorm, Git, GitHub, Composer, Hopscotch (pour le test des APIs).

**API** : The Movie Database (TMDB).

**Base de données** : MySQL.

## 3. Avec qui avez-vous travaillé ?

J'ai travaillé seul sur ce projet.

## 4. Contexte

Nom de l'entreprise, organisme ou association ▶ *Centre de formation - La plateforme*

Chantier, atelier, service ▶

Période d'exercice ▶ Du : 15/05/2023 au : 29/05/2023

## 5. Informations complémentaires (*facultatif*)

# DOSSIER PROFESSIONNEL (DP)

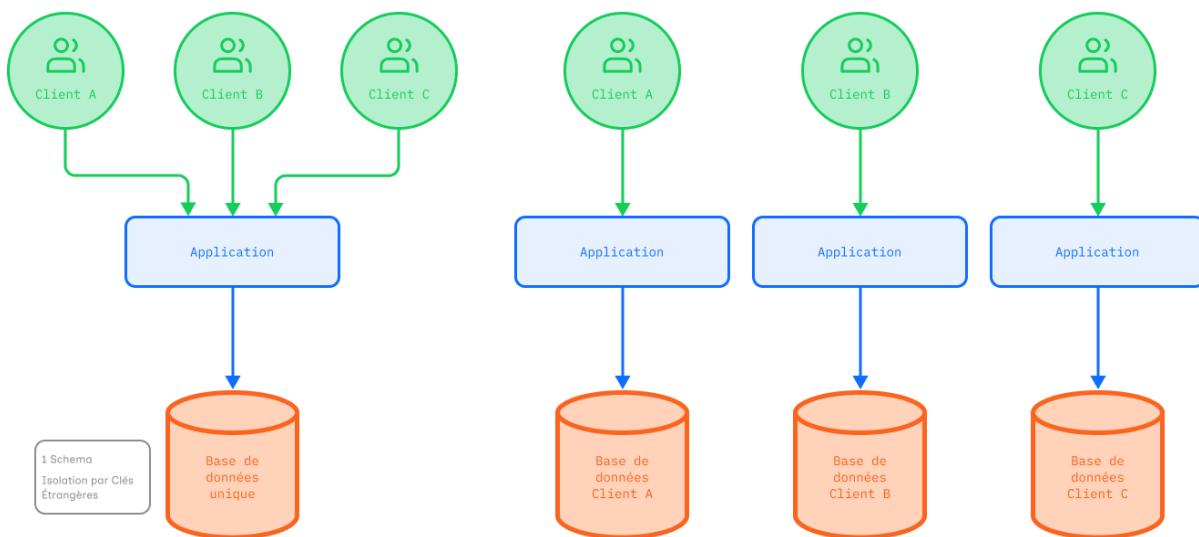
## Activité-type 2 Concevoir et développer une application sécurisée organisée en couches

Exemple n° 1 - Application mobile - **Proxifix**

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Proxifix est une application complète de gestion d'interventions techniques, développée sur une architecture multi-couches robuste. Elle offre aux entreprises la capacité de gérer efficacement leurs clients, équipements, techniciens et interventions grâce à un workflow sécurisé et optimisé. Le backend est bâti avec **Symfony 6** et **PHP 8.3**, en s'appuyant sur **API Platform** pour la création des APIs et **PostgreSQL** comme système de gestion de base de données. Pour le frontend, j'ai choisi **React Native** et **Expo**, garantissant une expérience mobile fluide et cross-plateforme.

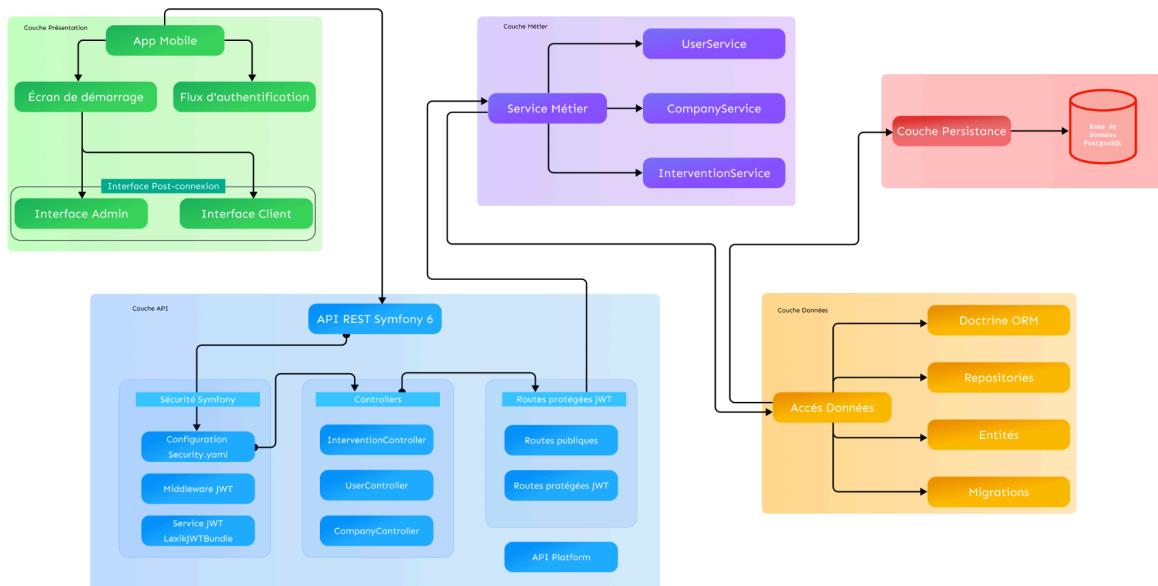
### Architecture SaaS Multi-Tenant



On a fait le choix d'une **architecture multi-tenant**, plutôt qu'une solution avec une instance par entreprise. Ça veut dire que toutes les entreprises partagent la **même base de données (shared-database)**, avec un seul schéma. Mais attention, les données de chaque entreprise sont bien **séparées et sécurisées** grâce à des clés étrangères dans les tables. Le gros avantage, c'est que ça **optimise les coûts** en mutualisant les ressources, et ça nous permet de garder la plateforme **toujours à jour** et de proposer rapidement de **nouvelles fonctionnalités** à tout le monde.

# DOSSIER PROFESSIONNEL (DP)

## Architecture par couche

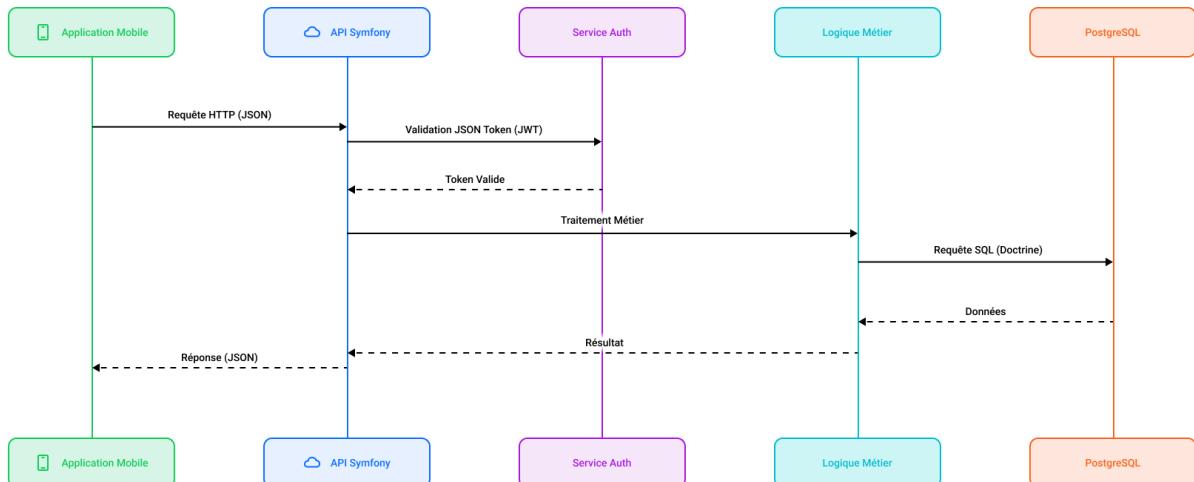


L'architecture de notre application est divisée en plusieurs couches distinctes, chacune ayant une responsabilité claire et bien définie. Cette approche garantit la modularité, la maintenabilité et la bonne organisation de notre code.

La **Couche Présentation** représente l'interface utilisateur, accessible via une application mobile. C'est elle qui gère l'écran de démarrage, le flux d'authentification et les différentes interfaces utilisateur (administrateur, client, et point-concessionnaire), permettant l'interaction directe avec l'utilisateur. En dessous se trouve la **Couche API (API REST Symfony 6)**, qui sert de pont vers notre backend. Elle reçoit les requêtes de la couche de présentation et est responsable de la sécurité (avec la configuration `security.yaml` et un middleware JWT), du routage et de la logique métier de haut niveau. C'est à ce niveau qu'**API Platform** est intégré pour faciliter la création des endpoints REST. La **Couche Métier**, quant à elle, contient la logique métier complexe de l'application, organisée en services comme le **UserService** pour les utilisateurs, le **CompanyService** pour les entreprises ou l'**InterventionService** pour les interventions. Les contrôleurs de l'API délèguent les tâches les plus complexes à ces services pour maintenir la séparation des responsabilités. Ces services interagissent avec la **Couche Données**, qui est responsable de l'accès et de la gestion de la base de données. Elle inclut **Doctrine ORM**, qui sert d'interface entre l'application et la base de données, ainsi que les **Repositories** (pour les requêtes complexes), les **Entités** (qui représentent nos tables) et les **Migrations** (pour gérer l'évolution du schéma). Enfin, la **Couche Persistance** représente la base de données elle-même, une base de données SQL dans notre cas.

# DOSSIER PROFESSIONNEL (DP)

## Architecture Globale du Système



L'architecture globale de notre système s'articule autour d'un flux de communication précis et sécurisé entre les différentes couches. Le processus débute par l'**Application Mobile** qui envoie une requête HTTP (au format JSON) à notre **API Symfony**. Cette requête est d'abord traitée par un service d'authentification qui se charge de valider le **JSON Web Token (JWT)**. Si le token est valide, la requête est transmise pour son **Traitement Métier**. Ce traitement interagit avec notre couche de **Logique Métier** qui, à son tour, exécute une requête SQL (via Doctrine) vers la base de données **PostgreSQL**. Une fois que les données sont récupérées et la logique métier appliquée, le résultat est renvoyé à l'**API Symfony**, qui la convertit en une réponse JSON. C'est cette réponse qui est finalement transmise à l'**Application Mobile** pour être affichée à l'utilisateur. Ce circuit garantit que chaque requête est authentifiée, que la logique est gérée de manière centralisée et que l'accès aux données est sécurisé.

Ce projet m'a permis d'acquérir la compétence de l'activité type 2 :

- Définir l'architecture logicielle d'une application

### 2. Précisez les moyens utilisés :

Outils de Conception et de Modélisation :

**Mermaid, Figma**

Sources d'Information et de Veille Technologique :

**Tutoriels en ligne, blogs techniques**

### 3. Avec qui avez-vous travaillé ?

J'ai avec Axel Vair. Étudiant dans la même formation.

### 4. Contexte

# DOSSIER PROFESSIONNEL (DP)

Nom de l'entreprise, organisme ou association ➤ Centre de formation - La Plateforme

Chantier, atelier, service ➤

Période d'exercice ➤ Du : 06/01/25 au : 15/02/25

## 5. Informations complémentaires (*facultatif*)

# DOSSIER PROFESSIONNEL (DP)

## Activité-type 2 Développer une application sécurisée

*Exemple n° 2 - Livre d'or JS - NoSQL*

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Dans le cadre de la formation, j'ai pu développer un accès aux données en NoSQL pour un projet de livre d'or permettant de se connecter et d'ajouter des commentaires sur une page. Le sujet initial proposait la création d'un livre d'or traditionnel avec MySQL/PHPMyAdmin, mais j'ai choisi d'adapter le projet pour utiliser MongoDB afin d'explorer les technologies NoSQL.

Le projet original consistait à développer un livre d'or permettant aux utilisateurs de laisser leurs avis, avec une structure comprenant des pages d'accueil, d'inscription/connexion, de profil, de visualisation des commentaires et d'ajout de commentaires. L'objectif était de créer une application complète avec validation asynchrone des formulaires et une interface utilisateur soignée.

Les bases de données NoSQL (Not Only SQL) représentent une approche moderne de la gestion des données qui s'écarte du modèle relationnel traditionnel. Contrairement aux bases de données relationnelles qui organisent les informations dans des tables avec des relations strictes entre elles, les bases de données NoSQL adoptent une structure plus flexible basée sur des documents. Cette flexibilité permet de stocker des données de manière plus naturelle, en conservant la structure hiérarchique des objets tels qu'ils sont manipulés dans le code applicatif.

MongoDB, l'une des bases de données NoSQL les plus populaires, utilise un modèle de stockage orienté document où les données sont organisées en collections de documents JSON (BSON dans le cas de MongoDB). Cette approche présente l'avantage considérable de permettre une évolution du schéma de données sans nécessiter de migrations complexes, contrairement aux bases de données relationnelles où toute modification de structure implique des opérations d'ALTER TABLE.

J'ai implémenté une couche d'accès aux données avec le **driver officiel MongoDB pour PHP**. La connexion à la base de données s'effectue via une URI standardisée, assurant une configuration simple et flexible.

Voici l'extrait de code pour la connexion à la base de données :

```
<?php  
public function __construct()  
{
```

# DOSSIER PROFESSIONNEL (DP)

```
try {
    $this→client = new Client($this→mongoUri);
    $this→db = $this→client→selectDatabase($this→database);
    $this→utilisateursCollection =
$this→db→selectCollection('utilisateurs');
    $this→commentairesCollection =
$this→db→selectCollection('commentaires');

} catch (Exception $e) {
    echo "ERROR: " . $e→getMessage();
}
}
```

Cette approche garantit que chaque instance de la classe User dispose d'un accès direct aux collections MongoDB nécessaires, éliminant ainsi le besoin de préparer des requêtes SQL comme c'était le cas avec l'approche relationnelle précédente.

La flexibilité de MongoDB a simplifié la gestion des utilisateurs. Chaque utilisateur est représenté par un document unique, ce qui rend les requêtes de recherche et d'insertion beaucoup plus lisibles et directes, en remplaçant les requêtes SQL par des méthodes intuitives comme `findOne()`.

```
public function check_DB($login)
{
    $user = $this→utilisateursCollection→findOne(['login' => $login]);
    return $user ? $user→toArray() : null;
}
```

Cette implémentation remplace avantageusement les requêtes SQL complexes par une syntaxe intuitive qui reflète directement la structure des données. L'utilisation de la méthode `findOne()` avec un critère de recherche simple rend le code plus lisible et maintenable.

L'enregistrement de nouveaux utilisateurs bénéficie également de cette simplicité, avec une insertion directe du document sans nécessiter de définition préalable de schéma :

```
public function register($email, $login, $password)
{
    $existingUser = $this→check_DB($login);

    if (empty($existingUser)) {
        try {

```

# DOSSIER PROFESSIONNEL (DP)

```
$result = $this->utilisateursCollection->insertOne([
    'email' => $email,
    'login' => $login,
    'password' => $password,
]);

if ($result->getInsertedCount() > 0) {
    return header("http/1.1 201 created");
}

} catch (Exception $e) {
    error_log("Erreur d'insertion: " . $e->getMessage());
}

header("http/1.1 400 Bad Request");
}
```

La récupération des commentaires avec les informations utilisateur associées démontre la puissance du pipeline d'agrégation MongoDB. Cette approche remplace les JOIN SQL par une séquence d'opérations de transformation des données :

```
public function livrOr(): array
{
    try {
        $pipeline = [
            [
                '$lookup' => [
                    'from' => 'utilisateurs',
                    'localField' => 'id_utilisateur',
                    'foreignField' => '_id',
                    'as' => 'user'
                ]
            ],
            [
                '$unwind' => '$user'
            ]
        ];
    } catch (Exception $e) {
        error_log("Erreur d'agrégation: " . $e->getMessage());
    }
    return $pipeline;
}
```

# DOSSIER PROFESSIONNEL (DP)

```
        ],
        [
            '$project' => [
                'login' => '$user.login',
                'commentaire' => 1,
                'date' => 1
            ]
        ],
        [
            '$sort' => ['date' => -1]
        ]
    ];

$cursor = $this->commentairesCollection->aggregate($pipeline);
$results = [];

foreach ($cursor as $document) {
    $results[] = [
        'login' => $document['login'],
        'commentaire' => $document['commentaire'],
        'date' => $document['date']->toDateTime()->format('Y-m-d H:i:s')
    ];
}

return $results;
} catch (Exception $e) {
    error_log("Erreur lors de la récupération des commentaires: " .
$e->getMessage());
    return [];
}
}
```

# DOSSIER PROFESSIONNEL (DP)

Ce projet m'a permis d'acquérir la compétence de l'activité type 1 :

Développer des composants d'accès aux données NoSQL

## 2. Précisez les moyens utilisés :

PHP, mongoDB, JavaScript, Vanilla, VScode

## 3. Avec qui avez-vous travaillé ?

J'ai travail en autonomie sur ce projet

## 4. Contexte

Nom de l'entreprise, organisme ou association ▶

*Centre de formation - La plateforme*

Chantier, atelier, service ▶

Période d'exercice

▶ Du : 14/02/2023 au : 19/02/2023

## 5. Informations complémentaires (facultatif)

# **DOSSIER PROFESSIONNEL** <sup>(DP)</sup>

---

# DOSSIER PROFESSIONNEL (DP)

## Activité-type 2 Concevoir et développer une application sécurisée organisée en couches

*Exemple n° 1 - Application mobile - Proxifix*

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Après une analyse approfondie des outils existants sur le marché, j'ai identifié un **manque notable pour une solution mobile de gestion d'interventions** intégrant à la fois une partie administrateur et une partie client. Les solutions desktop traditionnelles ne répondaient pas pleinement à la flexibilité et l'accessibilité requises par les opérations sur le terrain.

Cette analyse des besoins a mis en lumière les exigences clés suivantes pour le projet :

- **Gestion multi-rôles** : Prise en charge des profils Client, Technicien, Administrateur et Super Administrateur.
- **Workflow d'intervention complet** : Un cycle de vie clair allant de la demande à l'assignation, l'exécution et la clôture de l'intervention.
- **Interface mobile intuitive** : Une application facile à utiliser pour les techniciens directement sur le terrain.
- **Système intégré** : Gestion des équipements et planification des interventions.

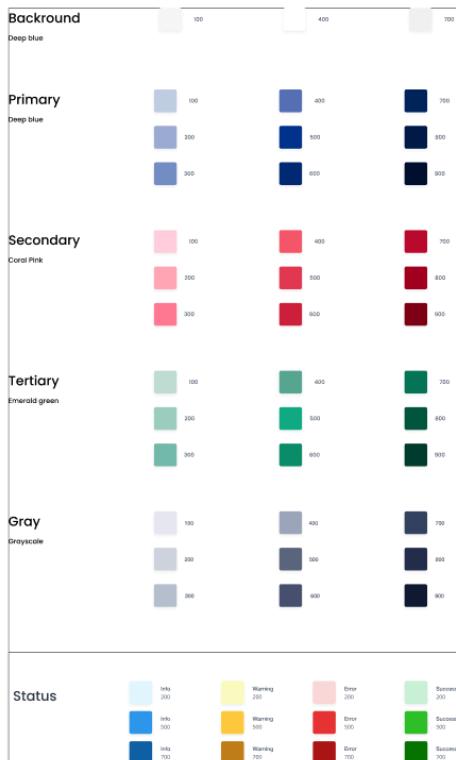
À partir de cette analyse des besoins, j'ai structuré la conception de l'interface utilisateur (UI) et l'expérience utilisateur (UX) en plusieurs étapes clés, en suivant un plan méthodique :

Charte graphique → Design System → Maquettes basse fidélité → Maquettes haute fidélité → Prototypage

Charte graphique : J'ai d'abord défini toutes les couleurs, typographies (polices, tailles pour les titres, corps de texte, etc.) qui seraient utilisées, afin d'établir une identité visuelle cohérente.

# DOSSIER PROFESSIONNEL (DP)

## Couleurs



## Polices

### Typefaces

**Aa**  
**Outfit**  
abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
0123456789

**Bold**  
Medium  
Regular

**Aa**  
**Rubik**  
abcdefghijklmnopqrstuvwxyz  
ABCDEFGHIJKLMNOPQRSTUVWXYZ  
0123456789

**Bold**  
Medium  
Regular

### Mobile

#### a) Main styles

**titre-1**  
**titre-2**  
**titre-3**  
**titre-4**  
**titre-5**  
**titre-6**

**paragraph/bold**  
paragraph/regular  
paragraph/italic

text-sm

text-xl

Pour assurer une uniformité visuelle et fonctionnelle de l'application, j'ai créé un Design System. Il inclut la définition des composants Figma réutilisables (boutons d'action, éléments de retour, etc.) garantissant une expérience utilisateur intuitive et compréhensible.

## Buttons

### Buttons

Cancel

Accept

## Inputs

### Input-text

Default

Focus

Error

Success

Disable

## Navbar components

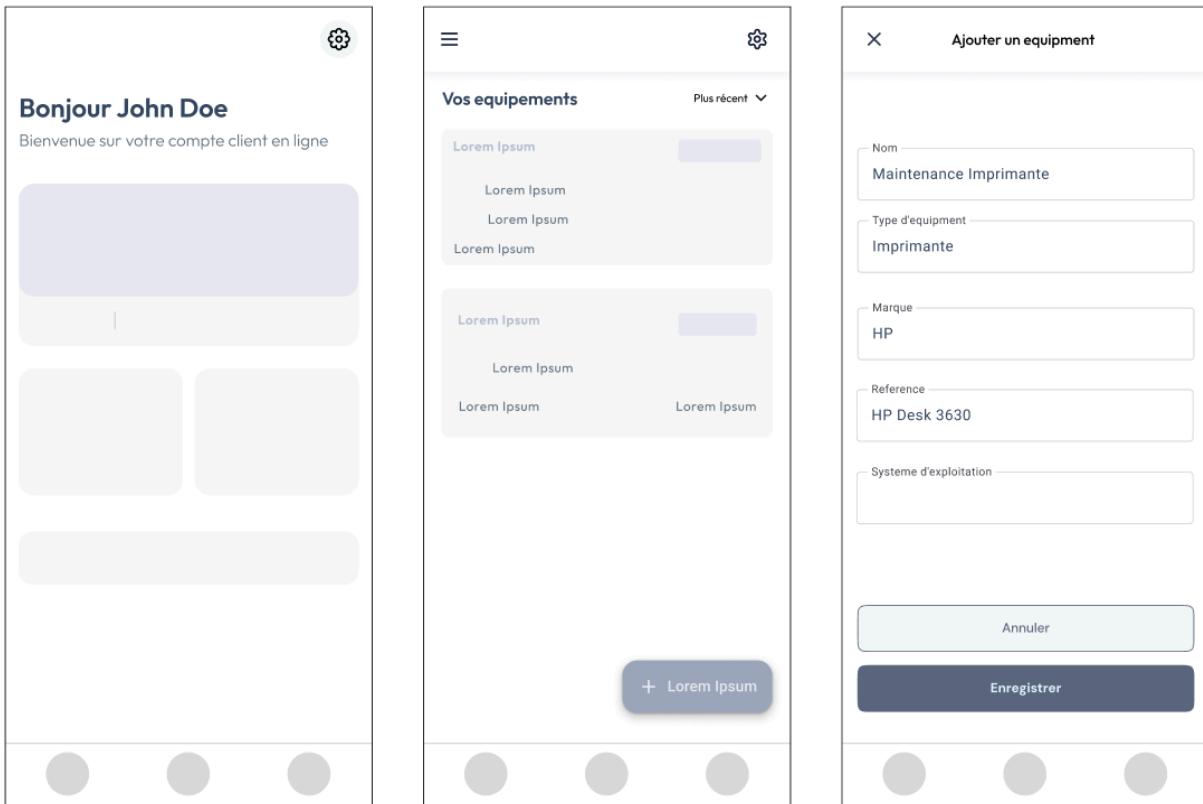
### Buttons

Nouvelles interventions



Cette étape a consisté à concevoir des wireframes en niveaux de gris. L'objectif était de définir l'agencement des pages et des éléments interactifs, en se concentrant sur la structure de l'information sans se soucier des détails visuels.

# DOSSIER PROFESSIONNEL (DP)



Enfin, j'ai appliqué la charte graphique et le Design System aux wireframes. Cela m'a permis de créer des prototypes visuellement finalisés, entièrement interactifs, intégrant les couleurs, typographies et composants finaux pour les sections **Client** et **Administrateur**.

## Partie Client :

# DOSSIER PROFESSIONNEL (DP)

The screenshots illustrate the user interface of the professional dossier application. The left panel shows a list of equipment (PC Portable, Imprimante hp) and interventions (Reparation Lenovo). The middle panel is a form for adding new equipment, requiring fields like Name, Type d'équipement, Marque, Reference, and Système d'exploitation. The right panel shows the details of a specific intervention, including information about the company (it-informatique), equipment (Lenovo IdeaPad 115IAU7), and tasks (Formatage HDD, Installation des logiciels basiques, Restauration sauvegarde HDD vers SSD).

## Partie Administrateur

The administrator interface includes several key components: a login panel for Proxave; a central dashboard showing client (Julien Morel, Chloé Lemoine, Lucas Dupuis) and equipment (Dépannage pour it-informatique) details; forms for creating new clients and enterprises; and a detailed view of an intervention (Dépannage par it-informatique, Christian BERGER, Print Laser).

J'ai conçu la base de données relationnelle en utilisant **PostgreSQL**, en respectant scrupuleusement les **règles de normalisation** pour garantir l'intégrité et l'efficacité des données.

```
CREATE TABLE "user" (
    id SERIAL PRIMARY KEY,
    email VARCHAR(180) UNIQUE NOT NULL,
    roles JSON NOT NULL,
```

# DOSSIER PROFESSIONNEL (DP)

```
password VARCHAR(255) NOT NULL,  
first_name VARCHAR(255) NOT NULL,  
last_name VARCHAR(255) NOT NULL,  
company_id INTEGER REFERENCES company(id) ON DELETE SET NULL,  
created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP  
) ;
```

Avec Symfony, j'ai exploité l'**ORM Doctrine** pour gérer les évolutions du schéma de la base de données de manière contrôlée, via des migrations.

```
public function up (Schema $schema) : void {  
    $this->addSql ('CREATE TABLE intervention (  
        id SERIAL NOT NULL,  
        company_id INT NOT NULL,  
        user_id INT NOT NULL,  
        PRIMARY KEY (id)  
    )' );  
    $this->addSql ('ALTER TABLE intervention  
        ADD CONSTRAINT FK_D11814AB979B1AD6  
        FOREIGN KEY (company_id) REFERENCES company (id)  
        NOT DEFERRABLE INITIALLY IMMEDIATE'  
);  
}
```

J'ai pu développer d'après les maquettes et la définition des besoins une **architecture en couches stricte**, conçue pour garantir une **séparation claire des responsabilités** et une **sécurité robuste**. Chaque couche possède un rôle bien défini, intégrant des considérations de sécurité et, dans une certaine mesure, d'éco-conception.

Mobile :

Cette couche est l'interface directe avec l'utilisateur. Elle gère l'affichage et les interactions, en s'appuyant sur **React Native et Expo Router** pour une navigation structurée et performante. La séparation des interfaces (authentification, client, administrateur) garantit que chaque utilisateur n'accède qu'aux fonctionnalités qui lui sont dédiées, renforçant ainsi la sécurité par la conception.

// Structure du routing avec Expo Router

```
app/  
  └── (auth)/      // Authentification  
  └── (main)/      // Application principale  
      └── customer/ // Interface client  
          └── admin/  // Interface administrateur  
      └── components/ // Composants réutilisables
```

# DOSSIER PROFESSIONNEL (DP)

## Couche Métier (Backend):

Cœur de l'application, cette couche encapsule toute la **logique métier et les règles de gestion**. Elle est responsable de la validité des opérations et de la cohérence des données. La sécurité est intégrée directement dans le comportement des entités, par exemple en contrôlant les transitions d'état.

```
// Entité Intervention avec logique métier
class Intervention {
    public const PENDING = 'pending';
    public const ASSIGNED = 'assigned';
    public const IN_PROGRESS = 'in_progress';
    public const COMPLETED = 'completed';
    // Workflow métier encapsulé
    public function assignTechnician(User $technician): void {
        if ($this->status !== self::PENDING) {
            throw new InvalidStateException(
                'Intervention must be pending to assign technician'
            );
        }
        $this->technician = $technician;
        $this->status = self::ASSIGNED;
    }
}
```

## Couche Données :

Cette couche est responsable de la **persistence et de la récupération des données**. Elle interagit directement avec la base de données via l'ORM Doctrine et est exposée de manière contrôlée via **API Platform**. Pour faciliter le développement de l'API, j'ai utilisé les **attributs natifs de PHP 8** directement sur mes entités pour définir les différents paramètres des colonnes et des comportements de l'API.

```
class InterventionRepository extends ServiceEntityRepository {
    public function findPendingByCompany(Company $company): array {
        return $this->createQueryBuilder('i')
            ->andWhere('i.company = :company')
            ->andWhere('i.status = :status')
            ->setParameter('company', $company)
            ->setParameter('status', Intervention::PENDING)
            ->getQuery()
            ->getResult();
    }
}

// Configuration API Platform avec groupes de sérialisation
#[ApiResource(
    operations: [
        new Get(security: "is_granted('ROLE_USER')"),
        new Post(security: "is_granted('ROLE_ADMIN')")
    ]
)]
```

# DOSSIER PROFESSIONNEL (DP)

```
],  
normalizationContext: ['groups' => ['intervention:read']],  
denormalizationContext: ['groups' => ['intervention:write']]  
)]  
class Intervention {  
#[Groups(['intervention:read', 'intervention:write'])]  
private ?string $title = null;  
}
```

Pour la **sécurité des données**, j'ai opté pour l'utilisation de **JSON Web Tokens (JWT)** pour l'authentification des utilisateurs. Ce token est généré dans le service d'authentification, à l'aide d'une dépendance Composer dédiée. J'ai mis en place une **sécurisation globale des ressources** via le fichier **security.yaml** de Symfony, en définissant une **hiérarchie de rôles** et un **contrôle plus fin au niveau des routes et des contrôleurs** pour les accès qui demandent une sécurité accrue. L'avantage du JWT, c'est que sa **partie "payload"** peut être personnalisée pour y inclure les données nécessaires, tout en étant **inaltérable** et doté d'une **date d'expiration** pour renforcer la sécurité.

Ce projet m'a permis d'acquérir la compétence de l'activité type 2 :

- Définir l'architecture logicielle d'une application
- Développer des composants d'accès aux données SQL et NoSQL
- Concevoir et mettre en place une base de données relationnelle

## 2. Précisez les moyens utilisés :

Symfony 6, PHP 8.3, API plateforme, Figma

## 3. Avec qui avez-vous travaillé ?

## 4. Contexte

Nom de l'entreprise, organisme ou association ➤ *LaPlateforme*

Chantier, atelier, service ➤ *LaPlateforme*

Période d'exercice ➤ Du : *02/12/2024* au : *01/07/2025*

# DOSSIER PROFESSIONNEL<sup>(DP)</sup>

## 5. Informations complémentaires (*facultatif*)

# DOSSIER PROFESSIONNEL (DP)

## Activité-type 3 Préparer le déploiement d'une application sécurisée

*Exemple n° 1 - Safefbase et Application Mobile*

### 1. Décrivez les tâches ou opérations que vous avez effectuées, et dans quelles conditions :

Pour le projet **Safefbase**, j'ai attaché à respecter une de **qualité code**, la **fiabilité des fonctionnalités** et la **sécurité du déploiement**, et ce, dès le début du développement.

J'ai adopté une approche Test-Driven-Development (TDD) pour le backend, c'est-à-dire que j'ai d'abord écrit un test qui ne passait pas (Red) puis le minimum de code pour le faire passer (Green), avant de réaliser des amélioration et réécriture (Refactor). J'ai utilisé cette méthode notamment pour l'ajout d'une nouvelle base de données.

J'ai ensuite détaillé dans un cahier de recette toutes les fonctionnalités attendues, avec le données en entrée et le résultat attendu.

Objectif	Vérifier l'ajout d'une nouvelle base de données avec validation de connexion
Prérequis	Serveur backend démarré, base SafeBase initialisée
Données d'entrée	{"name": "TestDB", "type": "postgres", "host": "localhost", "port": "5432", "username": "postgres", "password": "password", "database_name": "testdb"}
Résultat attendu	Status 201, retour de l'objet database avec ID généré
Critères d'acceptation	- Validation des champs obligatoires - Test de connectivité - Persistance en base - Retour JSON structuré

Cas de test n° 2 : Backup automatique d'une base

Objectif	Vérifier la création d'un backup avec génération de fichier
Prérequis	Base de données configurée et accessible
Données d'entrée	{"database_id": "uuid-valid"}
Résultat attendu	Status 201, fichier .sql créé, entrée backup en base

# DOSSIER PROFESSIONNEL (DP)

Critères d'acceptation	- Validation UUID - Génération fichier dump - Calcul taille fichier - Log des opérations
------------------------	---

La mise en place des différentes étapes (Red, Green, Refactor) se trouve dans l'annexe.

Pour faciliter le déploiement et garantir une bonne maintenance du code, j'ai mis en place des pipelines **GitHub Actions**. Pour l'**Intégration Continue (CI)**, un pipeline **lint** s'exécute automatiquement sur les branches **dev** et **main** pour assurer la qualité du code poussé ou fusionné. Un autre pipeline, **dev\_ci.yaml**, crée une base de données dans Docker avec le schéma nécessaire pour l'exécution des différents tests. Côté **Déploiement Continu (CD)**, j'ai utilisé des images Docker multi-stage pour optimiser leur taille. L'ensemble est déployé automatiquement sur le **registry** une fois que tous les tests ont été validés (le pipeline complet est détaillé en annexe).

The screenshot shows a GitHub Actions build status page. It starts with a green circular icon containing a checkmark and the text "All checks have passed". Below it, it says "5 successful checks". A list follows, each item with a green checkmark and a GitHub icon: "Dev Build / build (pull\_request)" (Successful in 43s), "lint / backend (pull\_request)" (Successful in 1m), "lint / backend (push)" (Successful in 1m), "lint / frontend (pull\_request)" (Successful in 25s), and "lint / frontend (push)" (Successful in 57s). At the bottom, there's another green circular icon with a checkmark and the text "No conflicts with base branch", followed by the note "Merging can be performed automatically." A "Squash and merge" button is at the bottom left, and a note "You can also merge this with the command line. [View command line instructions.](#)" is at the bottom right.

J'ai également créé un fichier **README.md** qui regroupe toutes les commandes utiles, que ce soit pour modifier les variables d'environnement ou pour lancer les tests en local.

Ce projet m'a permis d'acquérir la compétence de l'activité type 3 :

- Préparer et exécuter les plans de tests d'une application
- Préparer et documenter le déploiement d'une application
- Contribuer à la mise en production dans une démarche DevOps

## 2. Précisez les moyens utilisés :

# DOSSIER PROFESSIONNEL (DP)

VScode, Insomina, GitHub Action, Dbeaver, Docker.

## 3. Avec qui avez-vous travaillé ?

J'ai travail seul sur ce projet

## 4. Contexte

Nom de l'entreprise, organisme ou association ▶ *LaPlateforme*

Chantier, atelier, service ▶ *LaPlateforme*

Période d'exercice ▶ Du : *05/09/2024* au : *20/09/2024*

## 5. Informations complémentaires (*facultatif*)

## Titres, diplômes, CQP, attestations de formation

(*facultatif*)

Intitulé	Autorité ou organisme	Date
Cliquez ici.	Cliquez ici pour taper du texte.	Cliquez ici pour sélectionner une date.

# **DOSSIER PROFESSIONNEL** <sup>(DP)</sup>


# DOSSIER PROFESSIONNEL (DP)

## Déclaration sur l'honneur

Je soussigné(e) Jules JEAN-LOUIS

,

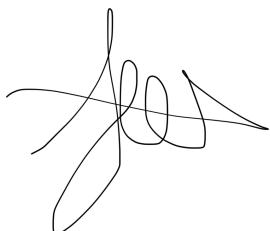
déclare sur l'honneur que les renseignements fournis dans ce dossier sont exacts et que je suis  
l'auteur(e) des réalisations jointes.

Fait à Marseille

le 01/08/2025

pour faire valoir ce que de droit.

Signature :



# DOSSIER PROFESSIONNEL <sup>(DP)</sup>

## Documents illustrant la pratique professionnelle

*(facultatif)*

### Intitulé

Cliquez ici pour taper du texte.

# DOSSIER PROFESSIONNEL (DP)

## ANNEXES

(Si le RC le prévoit)

./frontend/Dockerfile

```
FROM node:18 AS development
WORKDIR /usr/src/app
COPY package*.json ./
RUN npm install
COPY . .
EXPOSE 4200
# Commande pour démarrer le serveur Angular en mode développement
CMD ["npm", "start"]

FROM node:18 AS production
WORKDIR /app
COPY . .
RUN npm install
RUN npm run build -- --configuration build
# Utiliser une image Nginx pour servir l'application Angular
FROM nginx:alpine
COPY --from=production /app/dist/ /usr/share/nginx/html
COPY nginx.conf /etc/nginx/conf.d/default.conf
EXPOSE 80
# Démarrer Nginx
CMD ["nginx", "-g", "daemon off;"]
```

Test unitaire backend safebase backend:

```
func TestInsertDatabase(t *testing.T) {
    // Vérifier que les variables d'environnement sont présentes
    dbURL := os.Getenv("DATABASE_URL")
    if dbURL == "" {
        t.Fatal("DATABASE_URL environment variable is not set")
    }

    // Étape 1 : Initialiser la connexion à la base de données
    err := db.Connect()
    if err != nil {
        t.Fatalf("Erreur lors de la connexion à Safebase : %v", err)
    }
```

# DOSSIER PROFESSIONNEL (DP)

```
// Étape 2 : Créer une instance de base de données pour le test
database := model.Database{
    Name:          "test_database_safebase",
    Type:          "postgres",
    Host:          "localhost",
    Port:          os.Getenv("DB_PORT"),
    Username:      os.Getenv("DB_USER"),
    Password:      os.Getenv("DB_PASSWORD"),
    DatabaseName:  os.Getenv("DB_NAME"),
    IsCronActive:  false,
    CronSchedule:  "",
}

// Log des informations de connexion pour le débogage
t.Logf("Tentative de connexion avec:")
t.Logf("Host: %s", database.Host)
t.Logf("Port: %s", database.Port)
t.Logf("Database: %s", database.DatabaseName)

databaseService := services.NewDatabaseService()

// Test d'insertion
insertedDatabase, err := databaseService.CreateDatabase(
    database.Name, database.Type, database.Host, database.Port,
    database.Username, database.Password, database.DatabaseName,
    database.IsCronActive, database.CronSchedule,
)

if err != nil {
    t.Fatalf("Erreur lors de l'insertion de la base de données : %v", err)
}
t.Logf("Base de données insérée avec succès : %v", insertedDatabase.Name)

// Test de récupération
result, err := databaseService.GetDatabaseBy("name", database.Name)
if err != nil {
    t.Fatalf("Erreur lors de la récupération de la base de données : %v", err)
}

// Assertion
if result.Name != database.Name {
    t.Errorf("Nom de la base de données attendu : %v, mais obtenu : %v",
        database.Name, result.Name)
} else {
    correctly.t.Logf("Test réussi : la base de données a été insérée et récupérée")
}

// Nettoyage
err = databaseService.DeleteDatabase(result.ID.String())
if err != nil {
    t.Fatalf("Erreur lors de la suppression de la base de données : %v", err)
} else {
    t.Logf("Base de données supprimée avec succès.")
}
```

Test Unitaire : Validation Database Service  
Étape 1 : RED - Test écrit en premier

# DOSSIER PROFESSIONNEL (DP)

```
func TestCreateDatabase_ShouldReturnError_WhenNameIsEmpty(t *testing.T) {
    // Arrange
    databaseService := services.NewDatabaseService()

    // Act
    result, err := databaseService.CreateDatabase(
        "", // nom vide = cas d'erreur
        "postgres",
        "localhost",
        "5432",
        "postgres",
        "password",
        "testdb",
        false,
        "",
    )

    // Assert
    assert.Nil(t, result, "Le résultat devrait être nil en cas d'erreur")
    assert.NotNil(t, err, "Une erreur devrait être retournée")
    assert.Contains(t, err.Error(), "name is required",
                    "Le message d'erreur devrait mentionner le champ name")
}

func TestCreateDatabase_ShouldSuccess_WhenAllFieldsValid(t *testing.T) {
    // Arrange
    err := db.Connect()
    require.NoError(t, err, "La connexion DB doit réussir")

    databaseService := services.NewDatabaseService()
    expectedName := "test_database_" + uuid.New().String()

    // Act
    result, err := databaseService.CreateDatabase(
        expectedName,
        "postgres",
        "localhost",
        os.Getenv("DB_PORT"),
        os.Getenv("DB_USER"),
        os.Getenv("DB_PASSWORD"),
        os.Getenv("DB_NAME"),
        false,
    )
}
```

# DOSSIER PROFESSIONNEL (DP)

```
    "",  
)  
  
// Assert  
assert.NoError(t, err, "Aucune erreur ne devrait survenir")  
assert.NotNil(t, result, "Le résultat ne devrait pas être nil")  
assert.Equal(t, expectedName, result.Name, "Le nom doit correspondre")  
assert.NotEmpty(t, result.ID, "L'ID doit être généré")  
assert.True(t, result.CreatedAt.After(time.Now().Add(-time.Minute)),  
           "CreatedAt doit être récent")  
  
// Cleanup  
defer func() {  
    cleanupErr := databaseService.DeleteDatabase(result.ID.String())  
    assert.NoError(t, cleanupErr, "Le nettoyage doit réussir")  
}()  
}  
}
```

Étape 2 : GREEN - Code minimal pour faire passer le test

```
func (s *DatabaseService) CreateDatabase(  
    name string,  
    dbType string,  
    host string,  
    port string,  
    username string,  
    password string,  
    databaseName string,  
    isCronActive bool,  
    cronSchedule string,  
) (*model.Database, error) {  
  
    // Validation TDD : les tests guident l'implémentation  
    if name == "" {  
        return nil, fmt.Errorf("name is required")  
    }  
    if dbType == "" {  
        return nil, fmt.Errorf("type is required")  
    }  
    if host == "" {  
        return nil, fmt.Errorf("host is required")  
    }  
}
```

# DOSSIER PROFESSIONNEL (DP)

```
}

// Création de l'entité
database := &model.Database{
    Name:         name,
    Type:        dbType,
    Host:        host,
    Port:        port,
    Username:   username,
    Password:   password,
    DatabaseName: databaseName,
    IsCronActive: isCronActive,
    CronSchedule: cronSchedule,
}

// Persistance
result := s.DB.Create(database)
if result.Error != nil {
    return nil, result.Error
}

return database, nil
}
```

## Étape 3 : REFACTOR - Amélioration du code

```
// Ajout de la validation de connexion et logging
func (s *DatabaseService) CreateDatabase(...) (*model.Database, error) {
    // Validation des règles métier
    if err := s.validateDatabaseParams(name, dbType, host, port); err != nil {
        return nil, err
    }

    // Test de connectivité (TDD nous a guidé vers cette nécessité)
    if err := s.testConnection(host, port, username, password, databaseName,
dbType); err != nil {
        return nil, fmt.Errorf("connection test failed: %w", err)
    }
}
```

# DOSSIER PROFESSIONNEL (DP)

```
// Création et persistance
database := s.buildDatabaseEntity(name, dbType, host, port, username,
password, databaseName, isCronActive, cronSchedule)

if err := s.persistDatabase(database); err != nil {
    return nil, err
}

return database, nil
}
```

## TESTS D'INTÉGRATION

Test d'Intégration : API + Database + Validation

```
func TestAddDatabaseAPI_Integration(t *testing.T) {
    // Setup environnement d'intégration
    gin.SetMode(gin.TestMode)
    router := gin.Default()

    // Initialisation des services
    err := db.Connect()
    require.NoError(t, err)

    cronService, err := services.NewCronService()
    require.NoError(t, err)

    // Configuration de la route
    router.POST("/api/database", func(c *gin.Context) {
        database.AddDatabase(c, cronService)
    })

    // Test Case 1 : Succès avec données valides
    t.Run("Success_ValidData", func(t *testing.T) {
        // Arrange
        requestBody := `{
            "name": "integration_test_db",
            "type": "postgres",
            "host": "localhost",
            "port": "` + os.Getenv("DB_PORT") + `,
            "username": "` + os.Getenv("DB_USER") + `,
            "password": "test"
        }`
```

# DOSSIER PROFESSIONNEL (DP)

```
    "password": "` + os.Getenv("DB_PASSWORD") + `",
    "database_name": "` + os.Getenv("DB_NAME") + `"
}

req, _ := http.NewRequest("POST", "/api/database",
strings.NewReader(requestBody))
req.Header.Set("Content-Type", "application/json")
w := httptest.NewRecorder()

// Act
router.ServeHTTP(w, req)

// Assert HTTP Response
assert.Equal(t, 201, w.Code, "Status code doit être 201")

var response model.Database
err := json.Unmarshal(w.Body.Bytes(), &response)
assert.NoError(t, err, "La réponse doit être un JSON valide")
assert.Equal(t, "integration_test_db", response.Name)
assert.NotEmpty(t, response.ID, "L'ID doit être généré")

// Assert Database State
databaseService := services.NewDatabaseService()
dbFromDB, err :=
databaseService.GetDatabaseByID(response.ID.String())
assert.NoError(t, err, "La database doit être trouvée en base")
assert.Equal(t, "integration_test_db", dbFromDB.Name)

// Cleanup
defer func() {
    cleanupErr :=
databaseService.DeleteDatabase(response.ID.String())
    assert.NoError(t, cleanupErr)
}()

// Test Case 2 : Échec avec connexion invalide
t.Run("Failure_InvalidConnection", func(t *testing.T) {
```

# DOSSIER PROFESSIONNEL (DP)

```
// Arrange
requestBody := `{
    "name": "invalid_db",
    "type": "postgres",
    "host": "invalid-host",
    "port": "9999",
    "username": "invalid",
    "password": "invalid",
    "database_name": "invalid"
}`

req, _ := http.NewRequest("POST", "/api/database",
strings.NewReader(requestBody))
req.Header.Set("Content-Type", "application/json")
w := httptest.NewRecorder()

// Act
router.ServeHTTP(w, req)

// Assert
assert.Equal(t, 500, w.Code, "Status code doit être 500 pour
connexion invalide")

var errorResponse map[string]string
json.Unmarshal(w.Body.Bytes(), &errorResponse)
assert.Contains(t, errorResponse["error"], "connection",
    "Le message d'erreur doit mentionner la connexion")
})

// Test Case 3 : Validation des champs obligatoires
t.Run("Failure_MissingRequiredFields", func(t *testing.T) {
    // Arrange
    requestBody := `{"name": "", "type": "postgres"}` // champs manquants

    req, _ := http.NewRequest("POST", "/api/database",
strings.NewReader(requestBody))
    req.Header.Set("Content-Type", "application/json")
    w := httptest.NewRecorder()
```

# DOSSIER PROFESSIONNEL (DP)

```
// Act
router.ServeHTTP(w, req)

// Assert
assert.Equal(t, 400, w.Code, "Status code doit être 400 pour champs manquants")

var errorResponse map[string]string
json.Unmarshal(w.Body.Bytes(), &errorResponse)
assert.Contains(t, errorResponse["error"], "Missing required fields")
})

}
```

## Continuous Integration (CI)

```
name: Dev Build
on:
  push:
    branches: [dev]
  pull_request:
    branches: [dev]

jobs:
  lint:
    name: Code Quality Check
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      # Linting Backend Go
      - name: Setup Go
        uses: actions/setup-go@v5
        with:
          go-version: '1.23'

      - name: Install golangci-lint
        run: go install github.com/golangci/golangci-lint/cmd/golangci-lint@latest

      - name: Run golangci-lint
        run: |
```

# DOSSIER PROFESSIONNEL (DP)

```
cd backend
golangci-lint run ./... --timeout=5m
echo "✅ Backend code quality passed"

# Linting Frontend Angular
- name: Setup Node.js
  uses: actions/setup-node@v3
  with:
    node-version: '18'

- name: Frontend format check
  run: |
    cd frontend
    npm install prettier@~3.3.3
    npm run format:check
    echo "✅ Frontend code formatting passed"

test:
  name: Run Tests
  runs-on: ubuntu-latest
  needs: lint
  steps:
    - uses: actions/checkout@v4

    # Setup infrastructure de test
    - name: Setup PostgreSQL Test Environment
      run: |
        docker run -d \
          --name safebase_test_db \
          -e POSTGRES_USER=${{ secrets.DB_USER }} \
          -e POSTGRES_PASSWORD=${{ secrets.DB_PASSWORD }} \
          -e POSTGRES_DB=${{ secrets.DB_NAME }} \
          -p 5434:5432 \
          ${github.workspace}/sql/safebase_db/safebase.sql:/docker-entrypoint-initdb.d/safebase.sql \
          --health-cmd "pg_isready -U postgres" \
          --health-interval 10s \
          --health-timeout 5s \
          --health-retries 5 \
          postgres:16

    - name: Wait for database
      run: |
```

# DOSSIER PROFESSIONNEL (DP)

```
until pg_isready -h localhost -p 5434; do
  echo "⌚ Waiting for PostgreSQL..."
  sleep 2
done
echo "✅ Database ready"

# Exécution des tests avec couverture
- name: Run Unit Tests
  env:
    DATABASE_URL: postgres://$(secrets.DB_NAME):$(secrets.DB_PASSWORD}@localhost:5434/${
      secrets.DB_NAME}?sslmode=disable
  run: |
    cd backend
    go test ./... -v -cover -coverprofile=coverage.out
    go tool cover -html=coverage.out -o coverage.html
    echo "✅ Tests unitaires passed"

# Upload des résultats
- name: Upload Coverage Reports
  uses: actions/upload-artifact@v3
  with:
    name: coverage-report
    path: backend/coverage.html
```

## Continuous Deployment (CD)

```
name: Production CD
on:
  workflow_run:
    workflows: ["Dev Build"]
    types: [completed]
    branches: [dev]

jobs:
  deploy:
    name: Build and Deploy
    runs-on: ubuntu-latest
    if: ${{ github.event.workflow_run.conclusion == 'success' }}

    steps:
      - uses: actions/checkout@v4
```

# DOSSIER PROFESSIONNEL (DP)

```
# Setup Docker Build
- name: Set up Docker Buildx
  uses: docker/setup-buildx-action@v2

- name: Login to GitHub Container Registry
  uses: docker/login-action@v2
  with:
    registry: ghcr.io
    username: ${{ secrets.GHCR_USERNAME }}
    password: ${{ secrets.GHCR_TOKEN }}

# Build Backend Multi-Stage
- name: Build and Push Backend
  uses: docker/build-push-action@v4
  with:
    context: ./backend
    file: ./backend/Dockerfile
    push: true
    target: production
    tags: |
      ghcr.io/${{ github.repository }}/backend:latest
      ghcr.io/${{ github.repository }}/backend:${{ github.sha }}
    cache-from: type=gha
    cache-to: type=gha,mode=max

# Build Frontend Multi-Stage
- name: Build and Push Frontend
  uses: docker/build-push-action@v4
  with:
    context: ./frontend
    file: ./frontend/Dockerfile
    push: true
    target: production
    tags: |
      ghcr.io/${{ github.repository }}/frontend:latest
      ghcr.io/${{ github.repository }}/frontend:${{ github.sha }}
    cache-from: type=gha
    cache-to: type=gha,mode=max

# Notification de déploiement
- name: Deploy Success Notification
  uses: SimonScholz/google-chat-action@main
```

# DOSSIER PROFESSIONNEL (DP)

```
with:  
  webhookUrl: ${ secrets.GOOGLE_CHAT_WEBHOOK_URL }  
  jobStatus: ${ job.status }  
  title: "🚀 SafeBase Deployment"  
  subtitle: "Production deployment completed"
```