Documentation technique AdopteUnDev

Architecture générale

L'architecture globale suivante sera utilisée pour notre projet :

1. Couche de présentation (Frontend avec Flask/Jinja2)
2. Couche d'application ou service (Logique métier de l'application)
3. Couche de domaine ou model (Entités et règles métier)
4. Couche de persistance des données (accès à la base de donnéesMongoDB)

Cette architecture permettra de maintenir un code propre et modulaire, facilitant la maintenance à long terme.

Fonctionnalités

Nous allons développer notre applications autour des fonctionnalités principales suivantes :

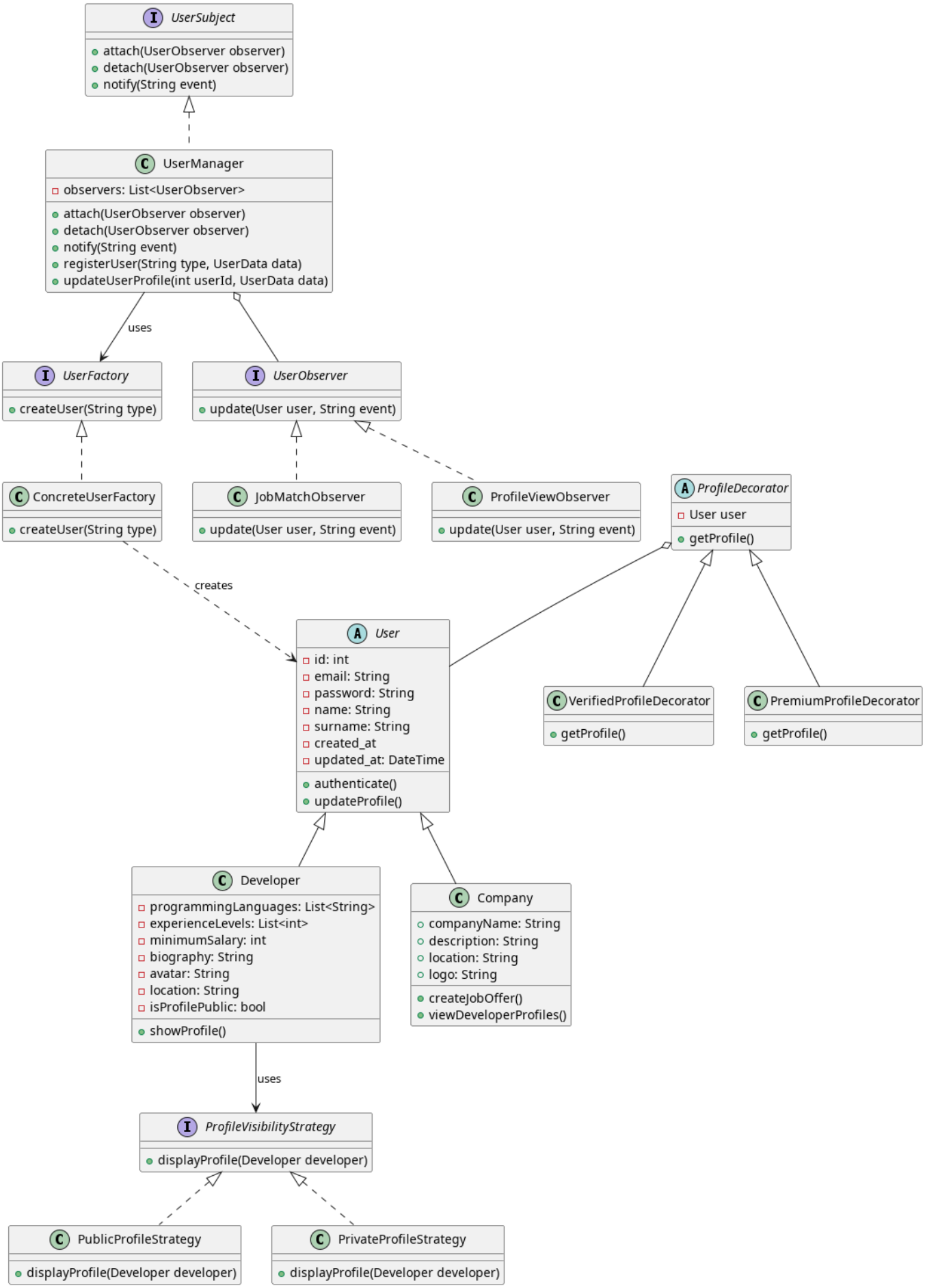
1. Gestion des utilisateurs

- Inscription distincte des utilisateurs (Développeur ou Entreprise)
- Profil public/privé pour les développeurs

Design patterns utilisés

- **Factory Method :** Utilisé pour créer différents types d'utilisateurs (Développeur/Entreprise)
- **Strategy :** Pour implémenter différentes stratégies d'affichage selon le type de profil du développeur (public/privé)
- **Decorator :** Pour ajouter des fonctionnalités supplémentaires aux profils utilisateurs
- **Observer :** Pour notifier les utilisateurs des mises à jour pertinentes

Diagramme UML



Explication du diagramme

User (Classe abstraite) Classe de base pour tous les utilisateurs Contient les attributs communs (id, email, password, etc.) Méthodes partagées comme authenticate() et updateProfile()

Developer et Company (Classes concrètes) Héritent de User Developer contient des attributs spécifiques comme programmingLanguages, experienceLevels, etc. Company contient des attributs spécifiques comme companyName, description, etc.

Design Pattern: Factory Method

UserFactory (Interface) Définit la méthode createUser() pour créer différents types d'utilisateurs

ConcreteUserFactory (Classe concrète) Implémente l'interface UserFactory Crée des instances de Developer ou Company selon le type demandé

Design Pattern: Strategy

ProfileVisibilityStrategy (Interface) Définit la stratégie d'affichage des profils

PublicProfileStrategy et PrivateProfileStrategy (Classes concrètes) Implémentent différentes stratégies d'affichage selon que le profil est public ou privé

Design Pattern: Decorator

ProfileDecorator (Classe abstraite) Décorateur de base pour ajouter des fonctionnalités aux profils

VerifiedProfileDecorator et PremiumProfileDecorator (Classes concrètes) Ajoutent des fonctionnalités supplémentaires comme la vérification ou les avantages premium

Design Pattern: Observer

UserObserver (Interface) Définit la méthode update() pour recevoir des notifications

JobMatchObserver et ProfileViewObserver (Classes concrètes) Implémentent différentes réactions aux événements (matchs d'emploi, vues de profil)

UserSubject (Interface) Définit les méthodes pour attacher/détacher des observateurs et les notifier

UserManager (Classe concrète) Gère les utilisateurs et implémente UserSubject Utilise UserFactory pour créer des utilisateurs Maintient une liste d'observateurs et les notifie des événements

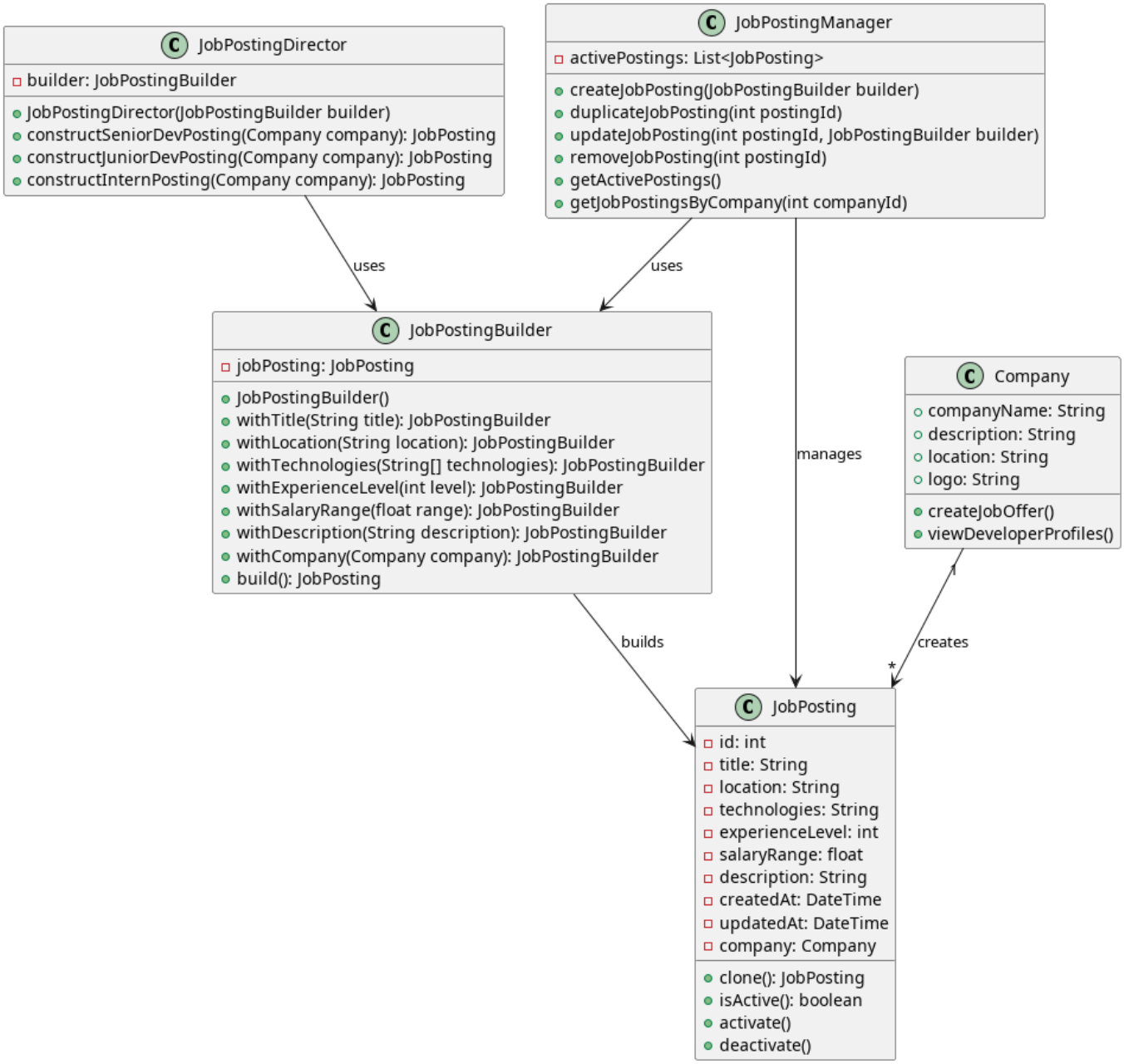
2. Gestion des fiches de postes

- Création de fiches par les entreprises
- Définition des critères (titre, localisation, technologies, etc.)

Design patterns utilisés :

Builder : Pour construire des fiches de poste complexes étape par étape **Prototype :** Pour cloner des fiches existantes comme base pour de nouvelles

Diagramme UML



Explication du diagramme

JobPosting Classe principale représentant une fiche de poste Contient tous les attributs nécessaires (titre, localisation, technologies, niveau d'expérience, salaire, description) Inclut des méthodes pour cloner une fiche (pattern Prototype) et gérer son état actif/inactif

JobPostingManager Gère l'ensemble des fiches de poste Fournit des méthodes pour créer, dupliquer, mettre à jour et supprimer des fiches Permet de récupérer les fiches actives ou par entreprise

Design Pattern: Builder

JobPostingBuilder Permet de construire des fiches de poste complexes étape par étape Utilise une interface fluide (méthodes chainables with...) Sépare la construction de la représentation

JobPostingDirector Utilise le builder pour construire des fiches de poste prédéfinies Fournit des méthodes pour créer rapidement des fiches spécifiques (senior, junior, stagiaire)

Design Pattern: Prototype

Le pattern Prototype est implémenté via la méthode clone() dans JobPosting, permettant de dupliquer facilement des fiches existantes comme base pour de nouvelles.

Relation entre les classes

- Company est associée à JobPosting (une entreprise peut créer plusieurs fiches de poste)
- JobPostingBuilder construit des instances de JobPosting
- JobPostingDirector utilise JobPostingBuilder pour construire des fiches prédéfinies
- JobPostingManager gère les instances de JobPosting et utilise JobPostingBuilder pour les mises à jour

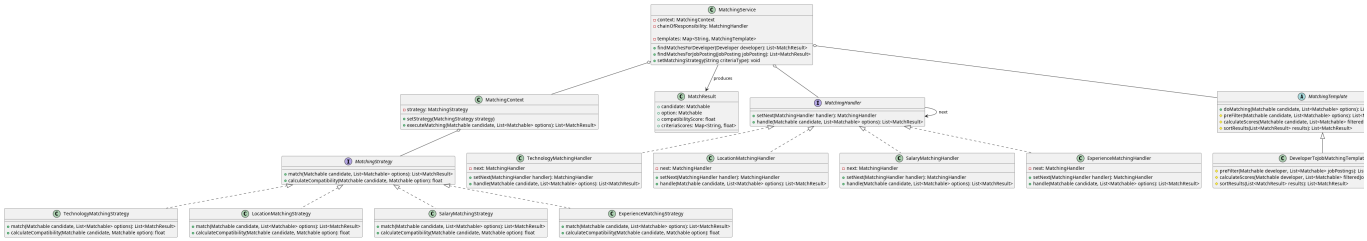
3. Système de matching

Suggestions basées sur des critères (langages, salaire, localisation, etc.)

Design patterns utilisés :

Strategy : Pour implémenter différents algorithmes de matching **Chain of Responsibility :** Pour filtrer les candidats selon différents critères **Template Method :** Pour définir le squelette d'un algorithme de matching

Diagramme UML



Explication du diagramme

Matchable (Interface)

- Interface que doivent implémenter les classes qui peuvent être mises en correspondance (Developer, JobPosting)
- Définit les méthodes communes pour accéder aux critères de matching (technologies, localisation, salaire, expérience)

Design Pattern: Strategy

MatchingStrategy (Interface)

- Définit l'interface pour les différentes stratégies de matching
- Permet de changer d'algorithme de matching à la volée

Stratégies concrètes

- TechnologyMatchingStrategy : Matching basé sur les technologies

- LocationMatchingStrategy : Matching basé sur la localisation
- SalaryMatchingStrategy : Matching basé sur le salaire
- ExperienceMatchingStrategy : Matching basé sur le niveau d'expérience

MatchingContext

- Maintient une référence à la stratégie actuelle
- Permet de changer de stratégie selon les besoins
- Délègue l'exécution du matching à la stratégie

Design Pattern: Chain of Responsibility

MatchingHandler (Interface)

- Définit l'interface pour les handlers de la chaîne
- Chaque handler peut traiter une partie du matching ou passer au suivant

Handlers concrets - TechnologyMatchingHandler : Filtre et évalue les technologies **- LocationMatchingHandler :** Filtre et évalue la localisation **- SalaryMatchingHandler :** Filtre et évalue le salaire **- ExperienceMatchingHandler :** Filtre et évalue l'expérience

Design Pattern: Template Method

MatchingTemplate (Classe abstraite)

- Définit le squelette de l'algorithme de matching
- Décompose le processus en étapes : préfiltrage, calcul des scores, tri des résultats
- Les sous-classes peuvent redéfinir certaines étapes

Templates concrets - DeveloperToJobMatchingTemplate : Matching d'un développeur vers des offres d'emploi **- JobToDeveloperMatchingTemplate :** Matching d'une offre d'emploi vers des développeurs

Classe de service

MatchingService

- Façade pour le système de matching
- Utilise les différents patterns pour effectuer le matching
- Expose des méthodes simples pour trouver des correspondances

MatchResult

- Représente le résultat d'un matching
- Contient les deux éléments mis en correspondance et leur score de compatibilité
- Inclut des scores détaillés par critère