Projet *PetBoarding* : Analyse V 1.0

Version initiale de l'analyse du système PetBoarding - Plateforme de gestion de pension et services pour animaux domestiques.

Cette version couvre :

- L'architecture Clean Architecture en .NET avec Angular 19
- Les fonctionnalités de gestion des utilisateurs, prestations, réservations et animaux
- L'implémentation CQRS avec authentification JWT
- L'interface utilisateur moderne basée sur Angular Material

1. Table des matières

2. Objectif du document

Ce document contient une analyse orientée objet du système PetBoarding, une plateforme de gestion de pension et services pour animaux domestiques. L'analyse suit les principes de la Clean Architecture et de la

méthodologie Domain-Driven Design (DDD).

Le système utilise une architecture moderne avec :

- **Backend** : .NET 8+ avec Clean Architecture et CQRS
- **Frontend** : Angular 19 avec architecture standalone et signals
- **Base de données** : PostgreSQL
- **Authentification** : JWT avec refresh tokens

L'objectif est d'établir la liste des classes métier et d'application nécessaires, ainsi que leurs interactions dans un contexte d'architecture en couches.

3. Sélection des cas d'utilisation

Évaluation des cas d'utilisation selon les critères de risques et pertinence :

Échelle des risques :

1. Fonctionnalité standard déjà maîtrisée
2. Patterns architecturaux connus mais nécessitant adaptation
3. Intégration de technologies modernes (Angular 19, .NET 8)
4. Logique métier complexe nécessitant analyse approfondie
5. Fonctionnalités innovantes ou non documentées

Échelle de pertinence :

1. Fonctionnalité cosmétique
2. Amélioration de l'expérience utilisateur
3. Fonctionnalité importante mais non critique
4. Fonctionnalité essentielle au métier
5. Fonctionnalité critique pour le fonctionnement du système

Cas d'utilisation	Risques	Pertinence	Prioritaire
S'inscrire au système	2	5	oui
Se connecter	1	5	oui
Gérer son profil	2	4	oui
Consulter les prestations	2	5	oui
Créer une prestation	2	5	oui
Modifier une prestation	2	4	oui
Enregistrer un animal	3	5	oui
Gérer les vaccinations	3	4	oui
Créer une réservation	4	5	oui
Consulter ses réservations	2	5	oui
Gérer le panier	3	4	oui

Cas d'utilisation	Risques	Pertinence	Prioritaire
Gestion des notifications	4	3	non
Rapports et statistiques	3	3	non

Les cas d'utilisation sélectionnés couvrent le cycle de vie complet d'un utilisateur dans le système : inscription, gestion de profil, enregistrement d'animaux, consultation de prestations, création de réservations et gestion du panier.

4. Cas d'utilisation

4.1. Groupe 1 : Gestion des utilisateurs

4.1.1. S'inscrire au système

L'utilisateur s'inscrit en fournissant ses informations personnelles et crée un compte dans le système.

Entités et classes candidates

```
@startuml
skin rose

class User <<entity>> {
  id: UserId
  firstname: Firstname
  lastname: Lastname
  email: Email
  phoneNumber: PhoneNumber
  passwordHash: string
  profileType: UserProfileType
  status: UserStatus
  address: Address
  createAt: DateTime
  updatedAt: DateTime
}

class RegisterRequestDto <<dto>> {
  firstName: string
  lastName: string
  email: string
  password: string
  phoneNumber: string
}

enum UserProfileType {
  CLIENT
  ADMIN
  EMPLOYEE
}
```

```
enum UserStatus {
  CREATED
  CONFIRMED
  INACTIVE
  DELETED
}

class RegisterComponent <<boundary>> {
  registerForm: FormGroup
  onSubmit()
  validateForm()
}

class AuthService <<control>> {
  register(): Observable<RegisterResponseDto>
}

class CreateUserHandler <<control>> {
  handle(): Result<User>
}

class UserRepository <<lifecycle>> {
  saveAsync(): Task<User>
  findByEmailAsync(): Task<User?>
}

RegisterComponent ..> AuthService
AuthService ..> CreateUserHandler
CreateUserHandler ..> UserRepository
CreateUserHandler ..> User

User --> UserProfileType
User --> UserStatus
@enduml
```

Diagramme de séquence (cas nominal)

```
@startuml
skin rose

actor Client as c
boundary RegisterComponent as ui
control AuthService as auth
control CreateUserHandler as handler
entity User as u
participant UserRepository as repo

c -> ui : saisir informations
c -> ui : valider formulaire
ui -> ui : validateForm()
ui -> auth : register(registerData)
```

```
auth -> handler : handle(CreateUserCommand)

alt email disponible
  handler -> repo : findByEmailAsync(email)
  repo --> handler : null
  handler -> u : new User()
  handler -> repo : saveAsync(user)
  repo --> handler : user
  handler --> auth : Result.Ok(user)
  auth --> ui : RegisterResponseDto(success: true)
  ui --> c : redirection profil
else email déjà utilisé
  handler -> repo : findByEmailAsync(email)
  repo --> handler : existingUser
  handler --> auth : Result.Fail("Email déjà utilisé")
  auth --> ui : RegisterResponseDto(success: false)
  ui --> c : message d'erreur
end

@enduml
```

Classes consolidées

```
@startuml
skin rose

class User <<entity>> {
  id: UserId
  firstname: Firstname
  lastname: Lastname
  email: Email
  phoneNumber: PhoneNumber
  passwordHash: string
  profileType: UserProfileType
  status: UserStatus
  changeForConfirmedStatus(): Result
  updateProfile(): Result
}

class RegisterComponent <<boundary>> {
  registerForm: FormGroup
  isLoading: signal<boolean>
  onSubmit(): void
  validateForm(): boolean
}

class AuthService <<control>> {
  register(): Observable<RegisterResponseDto>
  _currentUser: signal<User | null>
  _isAuthenticated: signal<boolean>
}
```

```
class CreateUserHandler <<control>> {
  handle(): Task<Result<User>>
}

class UserRepository <<lifecycle>> {
  saveAsync(): Task<User>
  findByEmailAsync(): Task<User?>
}

RegisterComponent ..> AuthService
AuthService ..> CreateUserHandler
CreateUserHandler ..> UserRepository
CreateUserHandler ..> User
UserRepository ..> User

@enduml
```

4.1.2. Se connecter

L'utilisateur se connecte avec son email et mot de passe pour accéder aux fonctionnalités du système.

Classes candidates

```
@startuml
skin rose

class LoginComponent <<boundary>> {
  loginForm: FormGroup
  rememberMe: boolean
  onSubmit(): void
}

class AuthService <<control>> {
  login(): Observable<LoginResponseDto>
  _isAuthenticated: signal<boolean>
  _currentUser: signal<User | null>
}

class LoginHandler <<control>> {
  handle(): Task<Result<LoginResponseDto>>
}

class TokenService <<service>> {
  setToken(): void
  setRefreshToken(): void
  hasValidSession(): boolean
}

class JwtService <<infrastructure>> {
  generateToken(): string
```

```
  validateToken(): boolean
}

LoginComponent ..> AuthService
AuthService ..> LoginHandler
AuthService ..> TokenService
LoginHandler ..> JwtService

@enduml
```

Diagramme de séquence (cas nominal)

```
@startuml
skin rose

actor Client as c
boundary LoginComponent as ui
control AuthService as auth
control LoginHandler as handler
service TokenService as token
entity User as u
participant UserRepository as repo

c -> ui : saisir email/password
c -> ui : cliquer connexion
ui -> auth : login(email, password, rememberMe)
auth -> handler : handle(LoginCommand)
handler -> repo : findByEmailAsync(email)
repo --> handler : user

alt utilisateur trouvé et password correct
  handler -> u : checkPassword(password)
  u --> handler : true
  handler -> token : generateToken(user)
  token --> handler : jwtToken
  handler --> auth : LoginResponseDto(success: true, token, user)
  auth -> token : setToken(jwtToken)
  auth -> auth : _currentUser.set(user)
  auth -> auth : _isAuthenticated.set(true)
  auth --> ui : success response
  ui --> c : redirection dashboard
else échec authentification
  handler --> auth : LoginResponseDto(success: false)
  auth --> ui : error response
  ui --> c : message d'erreur
end

@enduml
```

4.1.3. Gérer son profil

L'utilisateur authentifié peut consulter et modifier ses informations personnelles.

Classes candidates

```
@startuml
skin rose

class ProfileComponent <<boundary>> {
  currentUser: signal<User>
  profileForm: FormGroup
  onUpdateProfile(): void
}

class ProfileService <<control>> {
  getUserProfile(): Observable<GetProfileResponseDto>
  updateProfile(): Observable<UpdateProfileResponseDto>
}

class UpdateUserProfileHandler <<control>> {
  handle(): Task<Result<User>>
}

class GetUserProfileHandler <<control>> {
  handle(): Task<Result<User>>
}

ProfileComponent ..> ProfileService
ProfileService ..> UpdateUserProfileHandler
ProfileService ..> GetUserProfileHandler
UpdateUserProfileHandler ..> UserRepository
GetUserProfileHandler ..> UserRepository

@enduml
```

4.2. Groupe 2 : Gestion des prestations

4.2.1. Consulter les prestations disponibles

Les utilisateurs peuvent consulter la liste des prestations offertes par l'établissement.

Classes candidates

```
@startuml
skin rose

class Prestation <<entity>> {
  id: PrestationId
  libelle: string
  description: string
```

```
  categorieAnimal: TypeAnimal
  prix: decimal
  dureeEnMinutes: int
  estDisponible: boolean
  dateCreation: DateTime
}

enum TypeAnimal {
  CHIEN
  CHAT
  AUTRES
}

class PrestationsComponent <<boundary>> {
  prestations: signal<Prestation[]>
  filteredPrestations: signal<Prestation[]>
  filterByType(): void
}

class PrestationsService <<control>> {
  getPrestations(): Observable<Prestation[]>
  filterPrestations(): Prestation[]
}

class GetPrestationsHandler <<control>> {
  handle(): Task<Result<List<Prestation>>>
}

class PrestationRepository <<lifecycle>> {
  findAllActiveAsync(): Task<List<Prestation>>
  findByCategoryAsync(): Task<List<Prestation>>
}

PrestationsComponent ..> PrestationsService
PrestationsService ..> GetPrestationsHandler
GetPrestationsHandler ..> PrestationRepository
PrestationRepository ..> Prestation

@enduml
```

Diagramme de séquence (cas nominal)

```
@startuml
skin rose

actor Client as c
boundary PrestationsComponent as ui
control PrestationsService as service
control GetPrestationsHandler as handler
participant PrestationRepository as repo
entity Prestation as p
```

```
c -> ui : accéder page prestations
ui -> service : getPrestations()
service -> handler : handle(GetPrestationsQuery)
handler -> repo : findAllActiveAsync()
repo --> handler : List<Prestation>
handler --> service : Result.Ok(prestations)
service --> ui : Observable<Prestation[]>
ui -> ui : prestations.set(prestations)
ui --> c : affichage liste prestations

c -> ui : filtrer par type animal
ui -> service : filterPrestations(TypeAnimal.CHIEN)
service --> ui : prestations filtrées
ui -> ui : filteredPrestations.set(filtered)
ui --> c : affichage prestations filtrées

@enduml
```

4.2.2. Créer une prestation

Un administrateur peut créer de nouvelles prestations dans le système.

Classes candidates

```
@startuml
skin rose

class CreatePrestationComponent <<boundary>> {
  prestationForm: FormGroup
  onSubmit(): void
  validateForm(): boolean
}

class PrestationService <<control>> {
  createPrestation(): Observable<CreatePrestationResponseDto>
}

class CreatePrestationHandler <<control>> {
  handle(): Task<Result<Prestation>>
}

CreatePrestationComponent ..> PrestationService
PrestationService ..> CreatePrestationHandler
CreatePrestationHandler ..> PrestationRepository
CreatePrestationHandler ..> Prestation

@enduml
```

4.2.3. Modifier une prestation

Un administrateur peut modifier les caractéristiques d'une prestation existante.

Diagramme de séquence

```
@startuml
skin rose

actor Administrateur as a
boundary EditPrestationComponent as ui
control PrestationService as service
control UpdatePrestationHandler as handler
participant PrestationRepository as repo
entity Prestation as p

a -> ui : modifier prestation
ui -> service : updatePrestation(prestationId, updateData)
service -> handler : handle(UpdatePrestationCommand)
handler -> repo : findByIdAsync(prestationId)
repo --> handler : prestation

alt prestation trouvée
  handler -> p : modifierLibelle(nouveauLibelle)
  handler -> p : modifierPrix(nouveauPrix)
  handler -> repo : saveAsync(prestation)
  repo --> handler : prestation mise à jour
  handler --> service : Result.Ok(prestation)
  service --> ui : success response
  ui --> a : confirmation modification
else prestation non trouvée
  handler --> service : Result.Fail("Prestation introuvable")
  service --> ui : error response
  ui --> a : message d'erreur
end

@enduml
```

4.3. Groupe 3 : Gestion des animaux

4.3.1. Enregistrer un animal

Un client peut enregistrer les informations de ses animaux dans le système.

Classes candidates

```
@startuml
skin rose
```

```
class Pet <<entity>> {
  id: PetId
  name: string
  species: string
  breed: string
  birthDate: DateTime
  weight: decimal
  owner: User
  vaccinations: List<Vaccination>
}

class PetFormComponent <<boundary>> {
  petForm: FormGroup
  onSubmit(): void
}

class PetService <<control>> {
  createPet(): Observable<CreatePetResponseDto>
  getUserPets(): Observable<Pet[]>
}

class CreatePetHandler <<control>> {
  handle(): Task<Result<Pet>>
}

class PetRepository <<lifecycle>> {
  saveAsync(): Task<Pet>
  findByOwnerAsync(): Task<List<Pet>>
}

PetFormComponent ..> PetService
PetService ..> CreatePetHandler
CreatePetHandler ..> PetRepository
CreatePetHandler ..> Pet
Pet --> User

@enduml
```

4.3.2. Gérer les vaccinations

Le système permet d'enregistrer et suivre les vaccinations des animaux.

Classes candidates

```
@startuml
skin rose

class Vaccination <<entity>> {
  id: VaccinationId
  pet: Pet
  vaccineName: string
```

```
  vaccinationDate: DateTime
  expirationDate: DateTime
  veterinarian: string
}

class VaccinationFormComponent <<boundary>> {
  vaccinationForm: FormGroup
  selectedPet: Pet
  onSubmit(): void
}

class VaccinationService <<control>> {
  addVaccination(): Observable<AddVaccinationResponseDto>
  getPetVaccinations(): Observable<Vaccination[]>
}

Vaccination --> Pet
VaccinationFormComponent ..> VaccinationService

@enduml
```

4.4. Groupe 4 : Gestion des réservations

4.4.1. Créer une réservation

Un client peut créer une réservation pour une prestation donnée.

Classes candidates

```
@startuml
skin rose

class Reservation <<entity>> {
  id: ReservationId
  client: User
  prestation: Prestation
  pet: Pet
  dateDebut: DateTime
  dateFin: DateTime
  status: ReservationStatus
  prixTotal: decimal
  notes: string
}

enum ReservationStatus {
  PENDING
  CONFIRMED
  IN_PROGRESS
  COMPLETED
  CANCELLED
}
```

```
class ReservationFormComponent <<boundary>> {
  reservationForm: FormGroup
  selectedPrestation: Prestation
  selectedPet: Pet
  onSubmit(): void
}

class ReservationService <<control>> {
  createReservation(): Observable<CreateReservationResponseDto>
  getUserReservations(): Observable<Reservation[]>
}

class CreateReservationHandler <<control>> {
  handle(): Task<Result<Reservation>>
}

class ReservationRepository <<lifecycle>> {
  saveAsync(): Task<Reservation>
  findByClientAsync(): Task<List<Reservation>>
}

Reservation --> User
Reservation --> Prestation
Reservation --> Pet
Reservation --> ReservationStatus

ReservationFormComponent ..> ReservationService
ReservationService ..> CreateReservationHandler
CreateReservationHandler ..> ReservationRepository

@enduml
```

Diagramme de séquence (cas nominal)

```
@startuml
skin rose

actor Client as c
boundary ReservationFormComponent as ui
control ReservationService as service
control CreateReservationHandler as handler
participant ReservationRepository as repo
entity Reservation as r
entity Prestation as p
entity Pet as pet

c -> ui : sélectionner prestation
ui -> ui : selectedPrestation.set(prestation)
c -> ui : sélectionner animal
ui -> ui : selectedPet.set(pet)
```

```
c -> ui : choisir dates
c -> ui : valider réservation

ui -> service : createReservation(reservationData)
service -> handler : handle(CreateReservationCommand)

alt créneaux disponibles
  handler -> repo : checkAvailability(dateDebut, dateFin)
  repo --> handler : true
  handler -> r : new Reservation()
  handler -> r : calculateTotalPrice()
  handler -> repo : saveAsync(reservation)
  repo --> handler : reservation
  handler --> service : Result.Ok(reservation)
  service --> ui : success response
  ui --> c : confirmation réservation
else créneaux indisponibles
  handler -> repo : checkAvailability(dateDebut, dateFin)
  repo --> handler : false
  handler --> service : Result.Fail("Créneaux indisponibles")
  service --> ui : error response
  ui --> c : message d'erreur
end

@enduml
```

4.4.2. Consulter ses réservations

Un client peut consulter l'historique de ses réservations.

Classes candidates

```
@startuml
skin rose

class ReservationsComponent <<boundary>> {
  reservations: signal<Reservation[]>
  filteredReservations: signal<Reservation[]>
  filterByStatus(): void
}

class ReservationsService <<control>> {
  getUserReservations(): Observable<Reservation[]>
  filterReservations(): Reservation[]
}

class GetUserReservationsHandler <<control>> {
  handle(): Task<Result<List<Reservation>>>
}

ReservationsComponent ..> ReservationsService
```

```
ReservationsService ..> GetUserReservationsHandler
GetUserReservationsHandler ..> ReservationRepository

@enduml
```

4.4.3. Gérer le panier

Le système permet d'ajouter des prestations au panier avant de finaliser les réservations.

Classes candidates

```
@startuml
skin rose

class BasketItem <<entity>> {
  id: BasketItemId
  prestation: Prestation
  pet: Pet
  dateDebut: DateTime
  dateFin: DateTime
  quantity: int
  unitPrice: decimal
  totalPrice: decimal
}

class Basket <<entity>> {
  id: BasketId
  client: User
  items: List<BasketItem>
  totalAmount: decimal
  createdAt: DateTime
}

class BasketComponent <<boundary>> {
  basketItems: signal<BasketItem[]>
  totalAmount: signal<decimal>
  onRemoveItem(): void
  onCheckout(): void
}

class BasketService <<control>> {
  addToBasket(): Observable<BasketItem>
  removeFromBasket(): Observable<boolean>
  getBasketItems(): Observable<BasketItem[]>
  calculateTotal(): decimal
}

Basket --> User
Basket --> BasketItem
BasketItem --> Prestation
BasketItem --> Pet
```

```
BasketComponent ..> BasketService
BasketService ..> Basket

@enduml
```

5. Regroupement des classes

5.1. Groupe Domaine

```
@startuml
skin rose
hide empty members

' Entités principales
class User <<entity>> {
  id: UserId
  firstname: Firstname
  lastname: Lastname
  email: Email
  phoneNumber: PhoneNumber
  passwordHash: string
  profileType: UserProfileType
  status: UserStatus
  address: Address
  changeForConfirmedStatus(): Result
  updateProfile(): Result
}

class Prestation <<entity>> {
  id: PrestationId
  libelle: string
  description: string
  categorieAnimal: TypeAnimal
  prix: decimal
  dureeEnMinutes: int
  estDisponible: boolean
  modifierLibelle(): void
  modifierPrix(): void
  rendreDisponible(): void
}

class Pet <<entity>> {
  id: PetId
  name: string
  species: string
  breed: string
  birthDate: DateTime
  weight: decimal
  owner: User
}
```

```
class Reservation <<entity>> {
  id: ReservationId
  client: User
  prestation: Prestation
  pet: Pet
  dateDebut: DateTime
  dateFin: DateTime
  status: ReservationStatus
  prixTotal: decimal
  calculateTotalPrice(): decimal
}

class Vaccination <<entity>> {
  id: VaccinationId
  pet: Pet
  vaccineName: string
  vaccinationDate: DateTime
  expirationDate: DateTime
}

class Basket <<entity>> {
  id: BasketId
  client: User
  items: List<BasketItem>
  totalAmount: decimal
  addItem(): void
  removeItem(): void
  calculateTotal(): decimal
}

class BasketItem <<entity>> {
  id: BasketItemId
  prestation: Prestation
  pet: Pet
  dateDebut: DateTime
  dateFin: DateTime
  totalPrice: decimal
}

' Value Objects
class UserId <<value object>>
class PrestationId <<value object>>
class PetId <<value object>>
class ReservationId <<value object>>
class Firstname <<value object>>
class Lastname <<value object>>
class Email <<value object>>

' Enums
enum UserProfileType {
  CLIENT
  ADMIN
  EMPLOYEE
```

```
}

enum UserStatus {
  CREATED
  CONFIRMED
  INACTIVE
  DELETED
}

enum TypeAnimal {
  CHIEN
  CHAT
  AUTRES
}

enum ReservationStatus {
  PENDING
  CONFIRMED
  IN_PROGRESS
  COMPLETED
  CANCELLED
}

' Relations
User --> UserProfileType
User --> UserStatus
Pet --> User : owner
Prestation --> TypeAnimal
Reservation --> User : client
Reservation --> Prestation
Reservation --> Pet
Reservation --> ReservationStatus
Vaccination --> Pet
Basket --> User : client
Basket --> BasketItem
BasketItem --> Prestation
BasketItem --> Pet

@enduml
```

5.2. Groupe Application (CQRS)

```
@startuml
skin rose

' Command Handlers
class CreateUserHandler <<command handler>> {
  handle(CreateUserCommand): Task<Result<User>>
}

class LoginHandler <<command handler>> {
```

```
  handle(LoginCommand): Task<Result<LoginResponseDto>>
}

class UpdateUserProfileHandler <<command handler>> {
  handle(UpdateUserProfileCommand): Task<Result<User>>
}

class CreatePrestationHandler <<command handler>> {
  handle(CreatePrestationCommand): Task<Result<Prestation>>
}

class UpdatePrestationHandler <<command handler>> {
  handle(UpdatePrestationCommand): Task<Result<Prestation>>
}

class CreatePetHandler <<command handler>> {
  handle(CreatePetCommand): Task<Result<Pet>>
}

class CreateReservationHandler <<command handler>> {
  handle(CreateReservationCommand): Task<Result<Reservation>>
}

class AddVaccinationHandler <<command handler>> {
  handle(AddVaccinationCommand): Task<Result<Vaccination>>
}

' Query Handlers
class GetPrestationsHandler <<query handler>> {
  handle(GetPrestationsQuery): Task<Result<List<Prestation>>>
}

class GetUserProfileHandler <<query handler>> {
  handle(GetUserProfileQuery): Task<Result<User>>
}

class GetUserPetsHandler <<query handler>> {
  handle(GetUserPetsQuery): Task<Result<List<Pet>>>
}

class GetUserReservationsHandler <<query handler>> {
  handle(GetUserReservationsQuery): Task<Result<List<Reservation>>>
}

class GetPetVaccinationsHandler <<query handler>> {
  handle(GetPetVaccinationsQuery): Task<Result<List<Vaccination>>>
}

' Commands
class CreateUserCommand <<command>> {
  firstName: string
  lastName: string
  email: string
  password: string
```

```
  phoneNumber: string
}

class LoginCommand <<command>> {
  email: string
  password: string
  rememberMe: boolean
}

class CreateReservationCommand <<command>> {
  prestationId: PrestationId
  petId: PetId
  dateDebut: DateTime
  dateFin: DateTime
}

' Queries
class GetPrestationsQuery <<query>> {
  categorieAnimal?: TypeAnimal
}

class GetUserReservationsQuery <<query>> {
  userId: UserId
  status?: ReservationStatus
}

@enduml
```

5.3. Groupe Infrastructure

```
@startuml
skin rose

' Repositories
class UserRepository <<lifecycle>> {
  findByIdAsync(UserId): Task<User?>
  findByEmailAsync(string): Task<User?>
  saveAsync(User): Task<User>
  updateAsync(User): Task<User>
}

class PrestationRepository <<lifecycle>> {
  findByIdAsync(PrestationId): Task<Prestation?>
  findAllActiveAsync(): Task<List<Prestation>>
  findByCategoryAsync(TypeAnimal): Task<List<Prestation>>
  saveAsync(Prestation): Task<Prestation>
}

class PetRepository <<lifecycle>> {
  findByIdAsync(PetId): Task<Pet?>
  findByOwnerAsync(UserId): Task<List<Pet>>
```

```
  saveAsync(Pet): Task<Pet>
}

class ReservationRepository <<lifecycle>> {
  findByIdAsync(ReservationId): Task<Reservation?>
  findByClientAsync(UserId): Task<List<Reservation>>
  checkAvailabilityAsync(DateTime, DateTime): Task<boolean>
  saveAsync(Reservation): Task<Reservation>
}

class VaccinationRepository <<lifecycle>> {
  findByPetAsync(PetId): Task<List<Vaccination>>
  saveAsync(Vaccination): Task<Vaccination>
}

' Services Infrastructure
class JwtService <<service>> {
  generateToken(User): string
  validateToken(string): boolean
  extractUserId(string): UserId
}

class PasswordHashService <<service>> {
  hashPassword(string): string
  verifyPassword(string, string): boolean
}

class EmailService <<service>> {
  sendWelcomeEmailAsync(User): Task
  sendReservationConfirmationAsync(Reservation): Task
}

class UnitOfWork <<service>> {
  saveChangesAsync(): Task<int>
  beginTransactionAsync(): Task<IDbTransaction>
}

@enduml
```

5.4. Groupe Présentation (API + Frontend)

```
@startuml
skin rose

' Backend - API Endpoints
class AuthenticationEndpoints <<boundary>> {
  registerAsync(RegisterRequestDto): Task<IResult>
  loginAsync(LoginRequestDto): Task<IResult>
  refreshTokenAsync(): Task<IResult>
}
```

```
class UsersEndpoints <<boundary>> {
  createUserAsync(CreateUserDto): Task<IResult>
  getUserProfileAsync(): Task<IResult>
  updateUserProfileAsync(UpdateUserDto): Task<IResult>
}

class PrestationsEndpoints <<boundary>> {
  getPrestationsAsync(): Task<IResult>
  createPrestationAsync(CreatePrestationRequest): Task<IResult>
  updatePrestationAsync(UpdatePrestationRequest): Task<IResult>
}

class ReservationsEndpoints <<boundary>> {
  createReservationAsync(CreateReservationRequest): Task<IResult>
  getUserReservationsAsync(): Task<IResult>
}

' Frontend - Components
class LoginComponent <<boundary>> {
  loginForm: FormGroup
  isLoading: signal<boolean>
  onSubmit(): void
}

class RegisterComponent <<boundary>> {
  registerForm: FormGroup
  onSubmit(): void
}

class PrestationsComponent <<boundary>> {
  prestations: signal<Prestation[]>
  filteredPrestations: signal<Prestation[]>
  filterByType(): void
}

class PetFormComponent <<boundary>> {
  petForm: FormGroup
  onSubmit(): void
}

class ReservationFormComponent <<boundary>> {
  reservationForm: FormGroup
  selectedPrestation: Prestation
  selectedPet: Pet
  onSubmit(): void
}

class BasketComponent <<boundary>> {
  basketItems: signal<BasketItem[]>
  totalAmount: signal<decimal>
  onCheckout(): void
}

' Frontend - Services
```

```
  class AuthService <<control>> {
    register(): Observable<RegisterResponseDto>
    login(): Observable<LoginResponseDto>
    _currentUser: signal<User | null>
    _isAuthenticated: signal<boolean>
  }

  class PrestationsService <<control>> {
    getPrestations(): Observable<Prestation[]>
    createPrestation(): Observable<CreatePrestationResponseDto>
  }

  class ReservationService <<control>> {
    createReservation(): Observable<CreateReservationResponseDto>
    getUserReservations(): Observable<Reservation[]>
  }

  class BasketService <<control>> {
    addToBasket(): Observable<BasketItem>
    getBasketItems(): Observable<BasketItem[]>
  }

  ' Relations Frontend
  LoginComponent ..> AuthService
  RegisterComponent ..> AuthService
  PrestationsComponent ..> PrestationsService
  ReservationFormComponent ..> ReservationService
  BasketComponent ..> BasketService

  @enduml
```

6. Annexes

6.1. Terminologie

Clean Architecture : Architecture en couches avec inversion de dépendance, où le domaine métier est au centre et ne dépend d'aucune couche externe.

CQRS (Command Query Responsibility Segregation) : Séparation des responsabilités entre les commandes (écriture) et les requêtes (lecture).

Value Object : Objet immuable défini par ses valeurs plutôt que par son identité.

Domain Event : Événement métier qui se produit dans le domaine et peut déclencher des actions dans d'autres parties du système.

JWT (JSON Web Token) : Standard de token d'authentification encodé en JSON, utilisé pour l'authentification stateless.

Signals (Angular) : Primitive réactive d'Angular pour la gestion d'état, alternative moderne aux Observables pour certains cas.

Standalone Components : Composants Angular 19 qui ne nécessitent pas de modules et peuvent être utilisés de manière autonome.

6.2. Architecture technique

Stack technique :

- **Backend** : .NET 8, Entity Framework Core, PostgreSQL
- **Frontend** : Angular 19, TypeScript, Angular Material, Bootstrap
- **Authentification** : JWT avec refresh tokens
- **Containerisation** : Docker, docker-compose
- **Tests** : Architecture tests avec NetArchTest

Patterns architecturaux :

- Clean Architecture avec DDD
- CQRS avec MediatR
- Repository + Unit of Work
- Dependency Injection
- Event Sourcing (partiel avec Domain Events)

Organisation frontend :

- Feature-based architecture
- Signals pour la gestion d'état
- Injection function pattern
- Standalone components sans NgModules
- Contracts pattern pour les DTOs

Ce système PetBoarding illustre une architecture moderne et robuste adaptée aux besoins métier de gestion de pension pour animaux, avec une séparation claire des responsabilités et une approche orientée domaine.