CLINIQUE MONTVERT

Dossier Technique

Cyril Ponsan

1 YAGNI

1.1 Exemples d'applications des principes SOLID :

- Single Responsability :

```python
class PatientsService:
    """Service gérant les opérations liées aux patients"""

    # Repository pour accéder aux données des patients
    patients_repository: PgPatientsRepository

    def __init__(self, patients_repository: PgPatientsRepository):
        """
        Initialise le service avec un repository de patients
        Args:
            patients_repository: Repository pour accéder aux données des patients
        """
        self.patients_repository = patients_repository

    async def read_all_patients(
        self,
        db: Session,
        page: int,
        limit: int,
        field: str,
        order: str,
    ) -> dict:
        """
        Récupère la liste paginée de tous les patients
        Args:
            db: Session de base de données
            page: Numéro de la page
            limit: Nombre d'éléments par page
            field: Champ sur lequel trier
            order: Ordre de tri (asc/desc)
            user_id: ID de l'utilisateur faisant la requête
            role: Rôle de l'utilisateur
            request: Requête HTTP
        Returns:
            dict: Dictionnaire contenant les patients et leur nombre total
        """
        # Appel au repository pour récupérer les patients avec pagination et tri
        return await self.patients_repository.read_all_patients(
            db=db, page=page, limit=limit, field=field, order=order
        )

    async def detail_patient(self, db: Session, patient_id: int):
        """
        Récupère les détails d'un patient spécifique
        Args:
```

```
            db: Session de base de données
            patient_id: ID du patient à récupérer
            user_id: ID de l'utilisateur faisant la requête
            role: Rôle de l'utilisateur
            request: Requête HTTP
        Returns:
            Patient: Les détails du patient demandé
        """
        # Appel au repository pour récupérer les détails d'un patient par son ID
        return await self.patients_repository.read_patient_by_id(
            db=db, patient_id=patient_id
        )

    async def search_patients(
        self,
        db: Session,
        search: str,
        page: int,
        limit: int,
        field: str,
        order: str,
    ) -> dict:
        """
        Recherche des patients selon des critères
        Args:
            db: Session de base de données
            search: Terme de recherche
            page: Numéro de la page
            limit: Nombre d'éléments par page
            field: Champ sur lequel trier
            order: Ordre de tri (asc/desc)
            user_id: ID de l'utilisateur faisant la requête
            role: Rôle de l'utilisateur
            request: Requête HTTP
        Returns:
            dict: Dictionnaire contenant les patients trouvés et leur nombre total
        """
        # Appel au repository pour rechercher des patients avec pagination et tri
        return await self.patients_repository.search_patients(
            db=db, search=search, page=page, limit=limit, field=field, order=order
        )

    async def create_patient(self, db: Session, data: Patient) -> Patient:
        """
        Crée un nouveau patient
        Args:
            db: Session de base de données
            patient: Données du patient à créer
        Returns:
            Patient: Le patient créé
        Raises:
```

```
            HTTPException: Si le patient existe déjà
        """
        # Vérification si le patient existe déjà
        if await self.patients_repository.check_patient_exists(db, data):
            raise HTTPException(
                status_code=status.HTTP_400_BAD_REQUEST, detail="patient_already_exists"
            )
        # Création du patient via le repository
        return await self.patients_repository.create_patient(db, data)

    async def update_patient(
        self, db: Session, patient_id: int, data: Patient
    ) -> Patient:
        """
        Met à jour les données d'un patient existant
        """

        if not await self.patients_repository.read_patient_by_id(
            db=db, patient_id=patient_id
        ):
            raise HTTPException(
                status_code=status.HTTP_404_NOT_FOUND, detail="patient_not_found"
            )
        return await self.patients_repository.update_patient(db, patient_id, data)

```

- Cette classe présente dans le micro-service « Patients » a une seule responsabilité : gérer les opérations liées aux patients.
- Chaque méthode a une responsabilité unique et bien définie.
- La logique métier est séparée de la persistance des données.
-

- Open/Closed Principle :

```
# Interface pour les opérations liées aux documents
class DocumentsRead(ABC):
    @abstractmethod
    async def create_document(
        self, db: Session, file_name: str, type_document: DocumentType, patient_id: int
    ) -> dict:
        pass

# Interface pour les opérations liées aux documents
class DocumentsRepository(DocumentsRead):
    @abstractmethod
    async def create_document(
        self, db: Session, file_name: str, type_document: DocumentType, patient_id: int
    ) -> dict:
        pass

```

- L'interface « DocumentRead » est ouverte à l'extension mais fermée aux modifications.
- De nouvelles implémentations peuvent être ajoutées sans modifier le code existant.

- Liskov Substitution Principle :

```python
# Repository pour accéder aux données des documents
class PgDocumentsRepository(DocumentsRepository):
    # Méthode pour créer un document
    async def create_document(
        self, db: Session, file_name: str, type_document: DocumentType, patient_id: int
    ) -> Document:
        document = Document(
            nom_fichier=file_name, type_document=type_document, patient_id=patient_id
        )
        db.add(document)
        db.commit()
        db.refresh(document)
        return {"message": "document_created"}

    # Méthode pour récupérer un document par son ID
    async def get_document_by_id(self, db: Session, document_id: int) -> Document:
        return db.query(Document).filter(Document.id_document == document_id).first()

    # Méthode pour supprimer un document par son ID
    async def delete_document_by_id(self, db: Session, document_id: int) -> Document:
        db.query(Document).filter(Document.id_document == document_id).delete()
        db.commit()
        return {"success": True, "message": "document_deleted"}
```

- La classe « PgDocumentRepository » peut être substituée à sa classe parente « DocumentRepository ».
- Elle respecte le contrat défini par l'interface sans modifier le comportement attendu.

- Interface Segregation Principle

```python
# Interface pour les opérations liées aux documents
class DocumentsRead(ABC):
    @abstractmethod
    async def create_document(
        self, db: Session, file_name: str, type_document: DocumentType, patient_id: int
    ) -> dict:
        pass
```

- L'interface « DocumentRead » est petite et spécifique.
- Les clients ne sont pas forcés d'implémenter des méthodes dont ils n'ont pas besoin.

- Dependency Inversion Principle coté micro-services :

```python
# Endpoint pour créer un nouveau patient
@router.post("/", response_model=PostPatientResponse, status_code=201)
async def create_patient(
    request: Request,
    payload: Annotated[InternalPayload, Depends(check_authorization)],
    data: Annotated[CreatePatient, Body()],
    db: Session = Depends(get_db),
    patients_service=Depends(get_patients_service),
):
    """
    Crée un nouveau patient dans la base de données.

    Args:
        data (Patient): Les données du patient à créer
        db (Session): La session de base de données

    Returns:
        Patient: Les données du patient créé

    Raises:
        HTTPException: En cas d'erreur lors de la création
    """
    logger.write_log(
        f"{payload['role']} - {payload['user_id']} - {request.method} - create patient",
        request,
    )
    new_patient = await patients_service.create_patient(db=db, data=data)
    return {
        "success": True,
        "message": "Patient créé avec succès",
        "id_patient": new_patient.id_patient,
    }

```

  - Le router dépend d’abstractions (services) et non d’implémentations concrètes.
  - L’injection de dépendances est utilisée via « FastAPI Depends ».
  -

- Dependency Inversion Principle coté interface utilisateur :

```vue
<script setup lang="ts">
/**
 * @file PatientView.vue
 * @description Vue détaillée d'un patient permettant de voir et modifier ses informations et documents
 * @author [@CyrilPonsan](https://github.com/CyrilPonsan)
 */

// Import des composants
```

```vue
import DocumentsList from '@/components/documents/DocumentsList.vue'
import DocumentUpload from '@/components/documents/DocumentUploadDialog.vue'
import PageHeader from '@/components/PageHeader.vue'
import PatientDetail from '@/components/PatientDetail.vue'
import PatientActions from '@/components/patient/PatientActions.vue'
import PatientForm from '@/components/create-update-patient/PatientForm.vue'

// Import des composables et utilitaires
import usePatient from '@/composables/usePatient'
import useDocuments from '@/composables/useDocuments'
import usePatientForm from '@/composables/usePatientForm'

// Import des composables Vue
import { computed, onBeforeMount } from 'vue'
import { useI18n } from 'vue-i18n'
import { useRoute } from 'vue-router'

// Initialisation des composables
const { t } = useI18n()
const route = useRoute()
```

- Les composables sont injectés comme des dépendances.
- La vue dépend d'abstractions (interfaces des composables) plutôt que d'implémentations concrètes.

1.2 Conclusion

Ces exemples montrent une bonne application des principes SOLID :

- Séparation des responsabilités.
- Utilisation d'interfaces et d'abstractions.
- Injection de dépendances.
- Modules faiblement couplés et hautement cohésif.

Le code est ainsi plus :

- Maintenables.
- Testable.
- Extensible.
- Réutilisable.

2 Revue de code

2.1 Interface utilisateur

Utilisation du linter ESLint installé par défaut lors de la création du projet avec les options par défaut.

Utilisation de l'extension Prettier pour Visual Studio Code pour le formatage du code pour la lisibilité.

- Architecture :
  - Séparation des composants, vues et composables.
  - Structure claire avec des fichiers nommés en « upper case » et organisés.
  - Utilisation cohérente des composable Vue.js pour la logique métier.
  - Internationalisation.
  - Implémentation robuste de vue-i18n.
- Sécurité :
  - Validation des données de formulaires avec les bibliothèques Zod et Vee-Validate dans leur dernière version stable.
- Qualité du code :
  - Utilisation de TypeScript.
  - Types bien définis pour les props, variables, émissions d'évènements et les fonctions.
  - Messages de traduction bien organisés pour une maintenabilité améliorée.
  - Composants réutilisables bien structurés.
  - Gestion propre des états et des erreurs.
  - Utilisation efficace de PrimeVue.
  - Code documenté avec des commentaires pertinents expliquant les logiques complexes.
- Accessibilité :
  - Utilisation « d'aria-label » pour les éléments interactifs de l'interface utilisateur qui ne sont pas dotés de texte.
  - Implémentation d'info-bulles pour les éléments de l'interface utilisateur les plus pertinents.
- Tests :
  - Couverture presque complète pour les tests unitaires.
  - Utilisation appropriée des mocks pour les dépendances externes.
  - Tests bien structurés avec des descriptions claires.

Points D'amélioration :

- Erreurs :

    - Ajouter des types pour les messages d'erreurs.
- Tests :
    - Atteindre un ratio de 100% pour la couverture des différents composants par les tests unitaires.

2.2 Micro-services

Utilisation du linter Ruff pour l'écriture de code en Python avec les options par défaut. Il s'agit d'une extension pour l'IDE Visual Studio Code.

- Architecture :
    - Tous les micro-services suivent une structure de fichiers uniformes et appliquent les mêmes conventions de nommage, assurant ainsi la cohérence et la facilité de maintenance.
    - Séparation du routage pour les différentes ressources (patients, documents, etc.).
    - Séparation des responsabilités (routage, services, repositories).
    - Utilisation des dépendances FastAPI.
    - Structure de code modulaire.
- Sécurité :
    - Gestion dynamique des permissions.
    - Journalisation des actions.
    - Authentification des utilisateurs.
    - Validation des tokens et blacklistage des tokens.
    - Utilisation de schémas de validations pour les requêtes et les réponses.
- Qualité du code :
    - Utilisation de la casse « snake case » habituelle du langage de programmation Python pour le nommage des variables, propriétés, fonctions et méthodes.
    - Utilisation de la casse « upper case » pour le nommage des classes.
    - Code documenté.
    - Endpoints documentés avec OpenAPI (anciennement Swagger).
    - Utilisation de types pour les paramètres.
    - Gestions des erreurs grâce à des middlewares.
- Tests :
    - Les tests unitaires couvrent une grande partie des différents endpoints des micro-services.

Points d'amélioration :

- Sécurité :
    - Implémenter une limite de taille pour les documents téléversés.
    - Ajouter des validations MIME type plus strictes.

- Performances :
  - Implémenter une mise en cache pour les requêtes fréquentes avec Redis.
  - Ajouter des métriques de performances avec Prometheus et Grafana.
- Maintenabilité :
  - Centraliser la configuration dans un fichier dédié.
- Tests :
  - Atteindre un ratio de 100% pour la couverture des micro-services par les tests unitaires.

2.3 Conclusion

Globalement le code est de bonne qualité et bien structuré. Les points à améliorer seront pris en compte au fil des prochains sprints.

Une nouvelle revue de code sera effectuée lorsque les améliorations auront été effectuées.

Des audits de performances ont été réalisés avec « Lighthouse », disponible dans les outils pour développeur du navigateur Google Chrome. Les retours de ces audits sont globalement positifs, à part pour les indicateurs du référencement, ce qui dans le cadre de cette application ne pose pas de problème.

3 Stratégie de tests

3.1 Tests implémentés

A l'heure où ce document est rédigé seuls des tests unitaires ont été implémentés.

- Interface utilisateur :
  La quasi-totalité des composants et composables implémentés dans l'interface utilisateur sont couverts par des tests unitaires. Ces derniers ont été écrits avec la bibliothèque Vitest.

```ts
/**
 * Série de tests pour la fonctionnalité de recherche de patients
 */

import useLazyLoad from '@/composables/useLazyLoad'
import { describe, it, expect, vi, beforeEach } from 'vitest'
import { ref } from 'vue'
import { createPinia, setActivePinia } from 'pinia'

// Mock du composable useHttp pour simuler les appels API
vi.mock('@/composables/useHttp', () => ({
  default: () => ({
    sendRequest: vi.fn(), // Mock de la fonction d'envoi de requête
    isLoading: ref(false) // État de chargement initial
  })
}))

describe('useLazyLoad composable', () => {
  // Déclaration du type pour le composable qui sera testé
  let composable: ReturnType<typeof useLazyLoad<any>>

  // Avant chaque test, on réinitialise le store Pinia et le composable
  beforeEach(() => {
    setActivePinia(createPinia())
    composable = useLazyLoad<any>('/api/patients')
  })

  // Test de la mise à jour de la valeur de recherche et réinitialisation de la pagination
  it('should update search value and reset pagination', () => {
    // Définition d'une valeur de recherche
    composable.search.value = 'test'

    // Vérification que la pagination est réinitialisée à 0
    expect(composable.lazyState.value.first).toBe(0)
    // Vérification que l'état de chargement est false
    expect(composable.loading.value).toBe(false)
  })

  // Test de la réinitialisation du filtre de recherche
  it('should reset search value', () => {
    // Définition d'une valeur de recherche
    composable.search.value = 'test'
    // Appel de la fonction de réinitialisation
    composable.onResetFilter()

    // Vérification que la valeur de recherche est vide
    expect(composable.search.value).toBe('')
    // Vérification que la pagination est réinitialisée à 0
    expect(composable.lazyState.value.first).toBe(0)
  })
})
```

Les tests sont exécutés avec la commande :