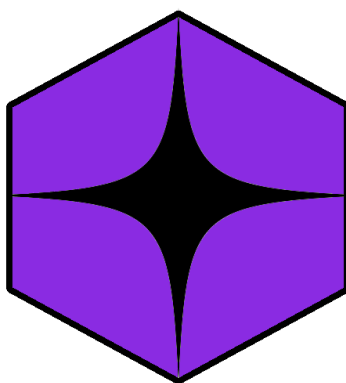


HEXAPLANNING

Carte interactive de gestion de tâches



Réalisé par Antoine Simper

2024 - 2025

Table des matières

I. Introduction

1. Présentation du projet
2. Objectifs et contexte

II. Spécifications du projet

1. Spécifications fonctionnelles
2. Spécifications techniques

III. Fonctionnalités principales

1. Page d'accueil
2. Gestion des quêtes (tâches)
3. Affichage visuel en hexagones (carte)
4. Système d'authentification et gestion des utilisateurs
5. Navigation et ergonomie

IV. Travail en équipe & méthodologie

1. Méthode Agile / Scrum
2. Workflow & branches stratégie
3. Outils collaboratifs

V. Modélisation des données

1. MCD (Modèle Conceptuel de Données)
2. MLD (Modèle Logique de Données)
3. Description des entités et relations

VI. Architecture technique et technologies

1. Vue d'ensemble
2. Frontend : Angular et PrimeNG
3. Backend : .NET Core
4. Base de données : PostgreSQL
5. Communication API REST
6. Infrastructure et DevOps
7. Services externes

VII. Qualité logicielle et tests

1. Tests unitaires (backend)
2. Tests d'intégration
3. Tests de charge et fixtures
4. Stratégie de validation

VIII. CI / CD

1. Intégration continue (CI) de l'API
2. Déploiement continu (CD) du backend
3. Déploiement continu (CD) du frontend
4. Conteneurisation et orchestration
5. Hébergement et reverse proxy

IX. Sécurité

1. Authentification et gestion des accès
2. Validation et intégrité des données
3. Protection contre les attaques
4. Sécurité de la conteneurisation et du déploiement

X. Accessibilité et conformité RGAA

1. Conformité RGAA et standards d'accessibilité
2. Accessibilité des formulaires
3. Navigation au clavier et focus management
4. Technologies d'assistance et lecteurs d'écran

XI. Conclusion et perspectives

1. Bilan du projet
2. Perspectives d'évolution
3. Ce que ce projet m'a apporté
4. Remerciements

I. Introduction

1. Présentation du projet

Hexapanning est une application web de gestion de tâches, pensée pour transformer la to-do list classique en une expérience visuelle et ludique. Reprenant la nomenclature des jeux-vidéos, les tâches sont appelées "quêtes". Chacune d'entre elles peut être placée sur une carte d'hexagones, permettant à l'utilisateur de visualiser ses objectifs comme un parcours à accomplir.

Cette approche vise à rendre la planification plus motivante et interactive, en s'inspirant des mécaniques de jeu et de la gamification. Hexapanning est destinée tout particulièrement aux personnes sujetes à un trouble de l'attention et ayant de la difficulté à se concentrer sur une tâche à la fois.

L'application a été développée en mobile-first, favorisant une utilisation quotidienne permettant à l'utilisateur d'avoir un aperçu de sa progression et de la mettre à jour régulièrement. Elle est bien entendu accessible également sur ordinateur, et l'utilisateur pourra se créer un compte pour accéder à sa progression depuis n'importe quel appareil.

2. Objectifs et contexte

Le projet est né du constat que la gestion des tâches peut rapidement devenir monotone et décourageante, surtout lorsqu'elle se limite à une simple liste. Hexapanning propose une alternative visuelle et dynamique, où chaque utilisateur peut organiser ses quêtes selon ses priorités et ses envies, tout en bénéficiant d'un suivi clair de sa progression. L'application s'adresse à toute personne souhaitant mieux organiser son temps, que ce soit dans un cadre personnel, scolaire ou professionnel, et met l'accent sur l'ergonomie, la sécurité et la personnalisation de l'expérience.

II. Spécifications du projet

1. Spécifications fonctionnelles

Fonctionnalités principales

- **Gestion des utilisateurs** : Inscription, connexion, changement et réinitialisation de mot de passe.
- **Gestion des quêtes** : Création, modification, suppression, changement rapide de statut.
- **Système de priorités** : Classification en trois niveaux (primaire, secondaire, tertiaire).
- **Visualisation hexagonale** : Attribution des quêtes sur une carte d'hexagones.
- **Suivi de progression** : Barre de progression et pourcentage d'avancement.

Cas d'usage (User Stories)



User Stories en tant qu'utilisateur, réalisées avec Trello.

Diagramme de cas d'usage

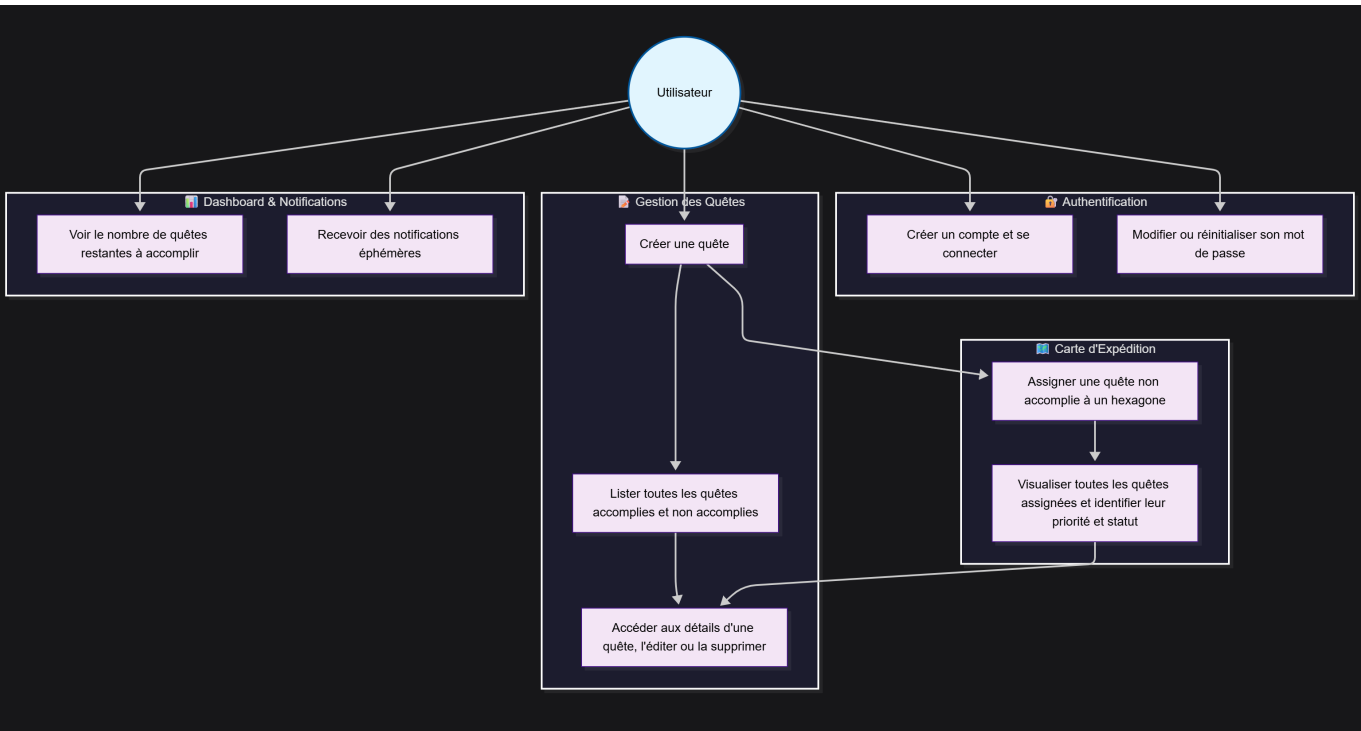
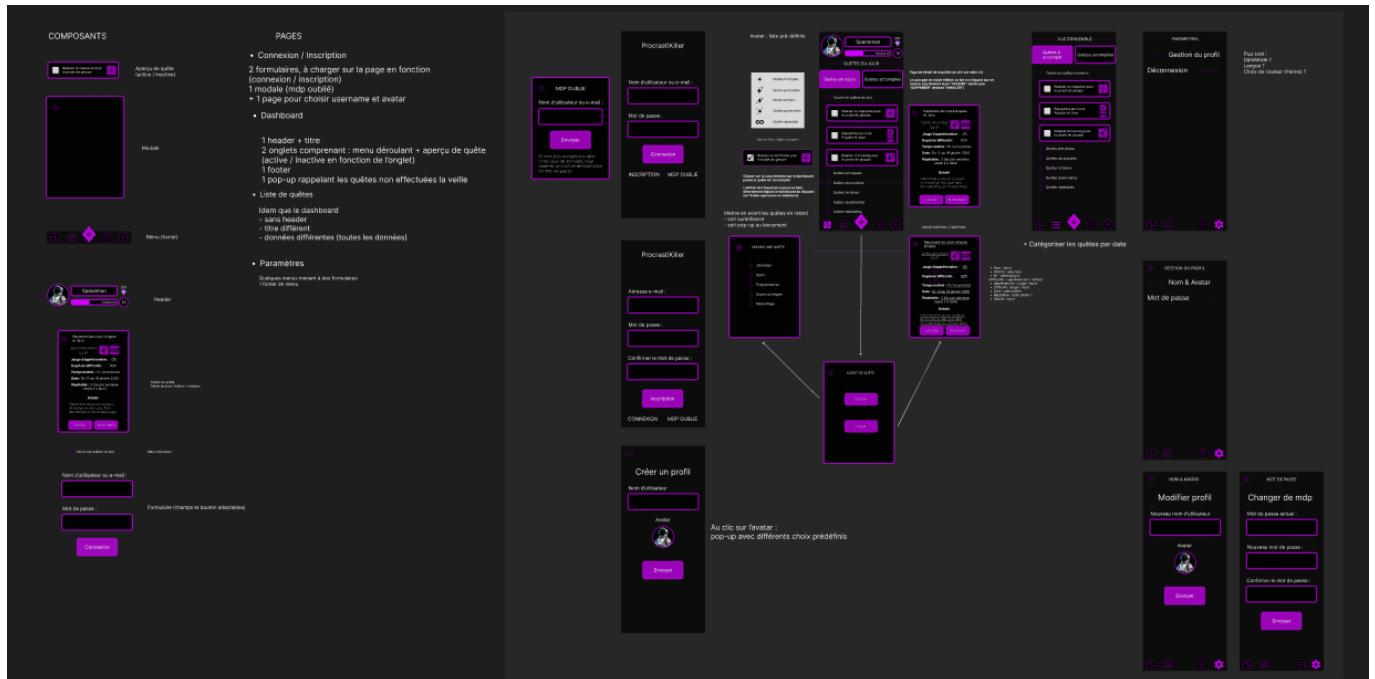


Diagramme de cas d'usage, réalisé avec Mermaid.

Analyse des cas d'usage :

1. **Authentification** (🔑) : Gestion complète de l'accès utilisateur avec sécurisation des mots de passe
2. **Gestion des Quêtes** (📋) : CRUD complet sur les tâches avec gestion des statuts et priorités
3. **Carte d'Expédition** (🗺️) : Visualisation sur une carte composée d'hexagones, avec assignation interactive
4. **Dashboard & Notifications** (📊) : Vue d'ensemble et notifications éphémères après chaque action

Maquette (Figma)



Version de départ de la maquette, réalisée avec Figma.

Objectifs pédagogiques du projet

- Développement d'une application web complète (frontend/backend)
- Mise en pratique des technologies modernes (Angular, .NET Core, PostgreSQL)
- Intégration de bonnes pratiques de développement (tests, CI/CD, sécurité)

2. Spécifications techniques

Technologies et frameworks utilisés

Frontend :

- Angular 18 avec TypeScript
- PrimeNG pour les composants UI
- SCSS pour le styling responsive

Backend :

- ASP.NET Core 8 avec C#
- Entity Framework Core pour l'ORM
- PostgreSQL comme base de données
- ASP.NET Identity pour l'authentification

Choix des langages et frameworks

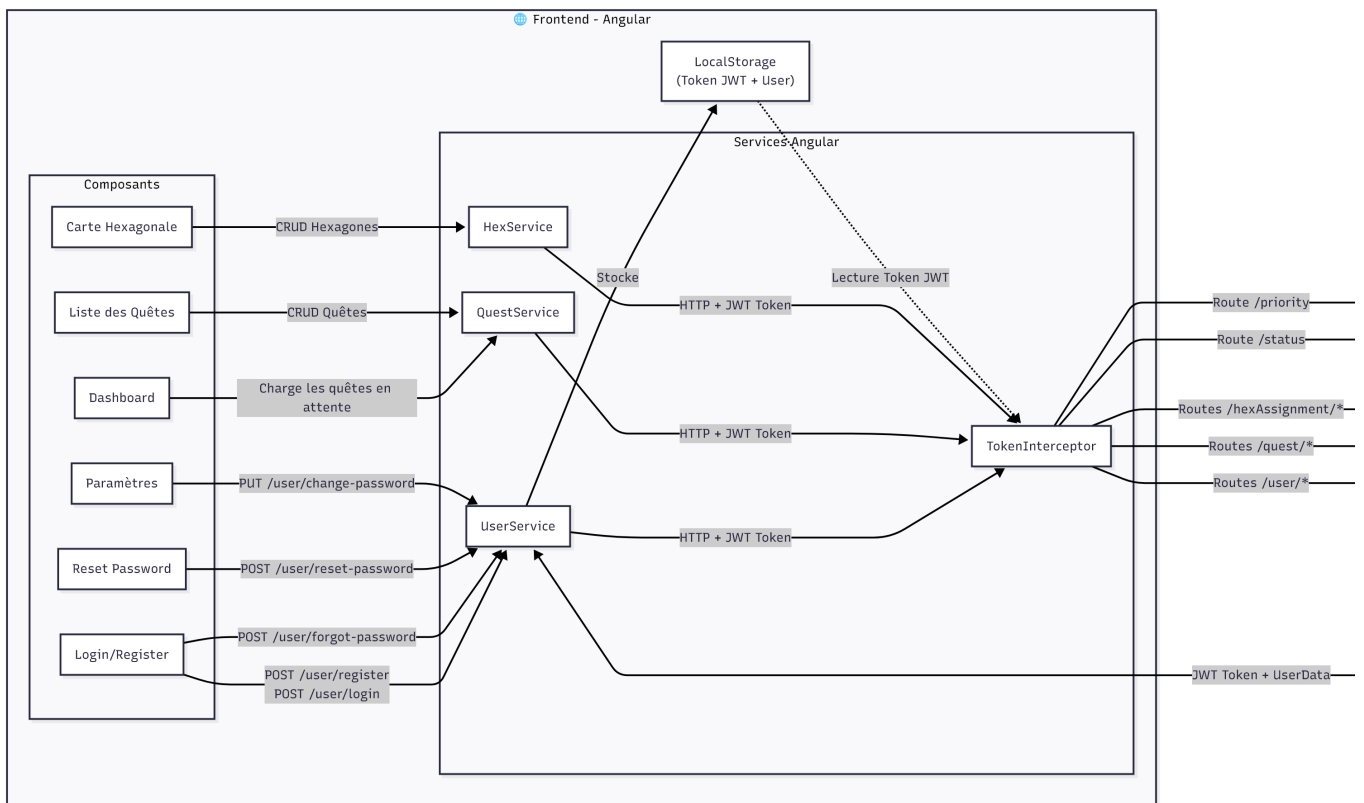
- **Angular** : Framework mature avec une large communauté, TypeScript intégré
- **ASP.NET Core** : Performance élevée, sécurité intégrée, cross-platform
- **PostgreSQL** : SGBD relationnel open-source, robuste et performant

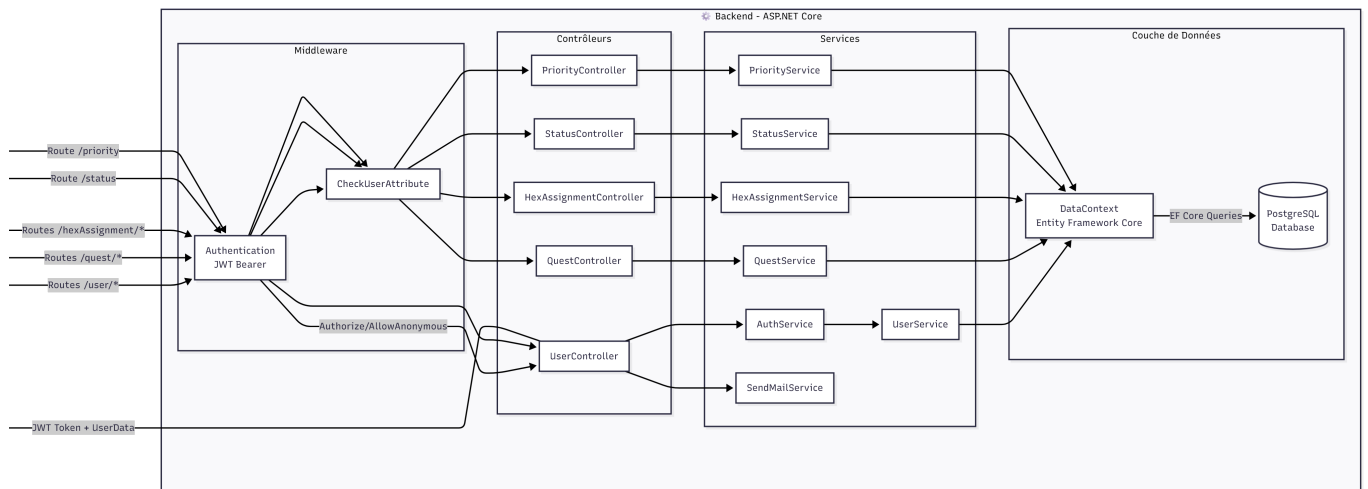
Outils de développement

- **Visual Studio Code** : IDE pour le front-end avec extensions spécialisées
- **Visual Studio** : IDE pour le back-end
- **GitKraken** : Interface graphique Git intuitive pour la gestion des branches et l'historique des commits
- **Swagger** : Documentation et accessibilité des endpoints de l'API

Outils d'environnement (CI, Git, GitHub, Jest, Docker, Maven, Node.js, Navigateurs...)

- **Git/GitHub** : Contrôle de version et collaboration
- **GitHub Actions** : Intégration et déploiement continus
- **Docker** : Conteneurisation des services (frontend, backend, base de données)
- **Node.js** : Runtime pour les outils de build Angular
- **npm** : Gestionnaire de packages JavaScript
- **Navigateurs** : Chrome et Firefox pour les tests cross-browser

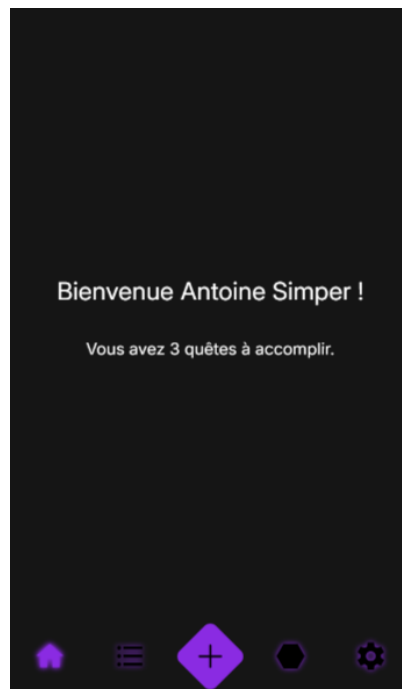




Organigramme Front / Back.

III. Fonctionnalités principales

1. Page d'accueil



Page d'accueil d'Hexaplanning.

La page d'accueil apparaît dès la connexion de l'utilisateur, et affiche le nombre de quêtes qu'il lui reste à accomplir

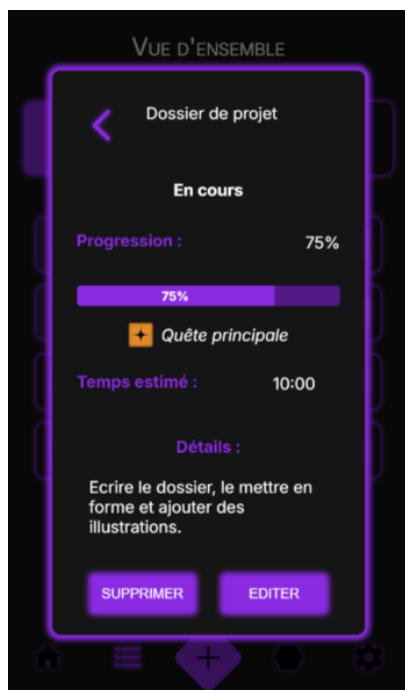
```
<h1>Bienvenue {{ user()?.firstName | titlecase }} {{ user()?.lastName | titlecase }} !</h1>
<p>Vous avez <span class="quest-count">{{ pending_questions_number() }}</span>
quête{{ pending_questions_number() > 1 ? 's' : '' }} à accomplir.</p>
```

```
export class DashboardPageComponent {
  private readonly _userService = inject(UserService);
  private readonly _questService = inject(QuestService);
  user = this._userService.user;
  pending_questions_number = computed(() =>
    this._questService.pendingQuests().length);

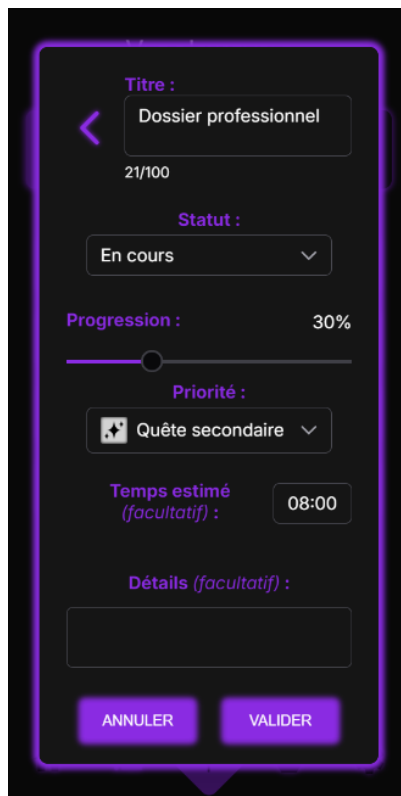
  constructor() {
    this._questService.getAllPendingQuests().subscribe();
  }
}
```

2. Gestion des quêtes (tâches)

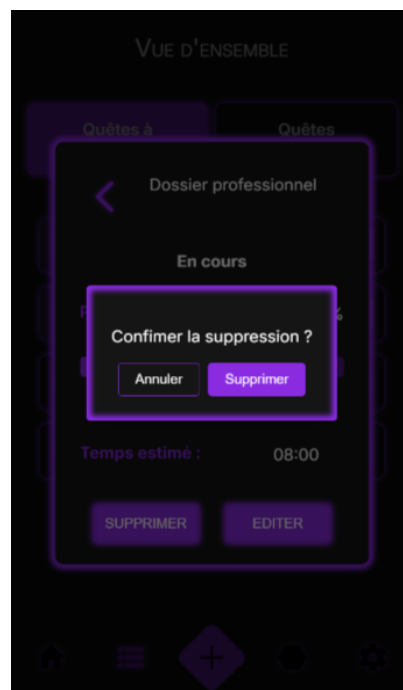
Les tâches, appelées "quêtes", sont au cœur de l'application. Chaque quête possède un titre, un statut (en attente, en cours et terminée) et une priorité (primaire, secondaire ou tertiaire, avec une icône et un code couleur associés), ainsi qu'une description et un temps estimé en option, ainsi qu'un pourcentage de progression (associé à une barre de progression) dans le cas des quêtes en cours. L'utilisateur peut créer, éditer ou supprimer une quête, la marquer rapidement comme terminée ou la remettre en attente, et l'associer à un hexagone sur la carte prévue à cet effet.



Modale de détails d'une quête.



Edition d'une quête existante.



Modale de suppression d'une quête.

```
type QuestModalData = {
  quest: QuestUpdatedTO | QuestCreatedTO;
  isNew: boolean;
};

@Injectable({
  providedIn: 'root',
})
export class QuestModalService {
  private readonly _DEFAULT_QUESTION: QuestCreatedTO = {
    title: '',
    estimatedTime: 0,
    description: '',
    statusId: '17c07323-d5b4-4568-b773-de3487ff30b1',
    priorityId: '17c07323-d5b4-4568-b773-de3487ff30b1',
  };

  private _questModalVisible = signal<boolean>(false);
  private _questModalData = signal<QuestModalData>({
    quest: this._DEFAULT_QUESTION,
    isNew: false,
  });
```



```

});

public questModalVisible = this._questModalVisible.asReadOnly();
public questModalData = this._questModalData.asReadOnly();

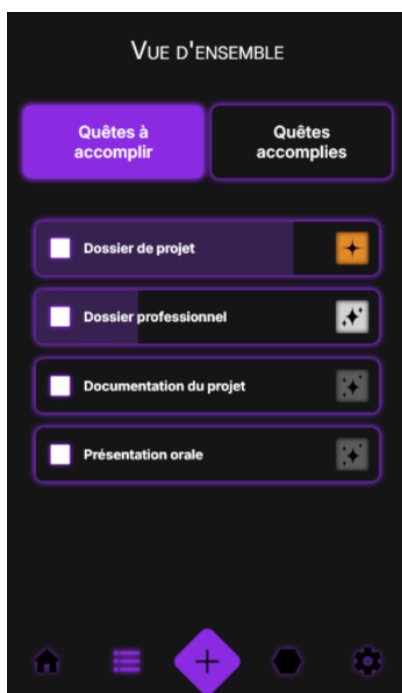
openQuestDetails(quest: QuestUpdateDTO, isNew = false): void {
    this._questModalData.set({ quest, isNew });
    this._questModalVisible.set(true);
}

openNewQuest(): void {
    this.openQuestDetails(this._DEFAULT_QUEST as QuestUpdateDTO, true);
}

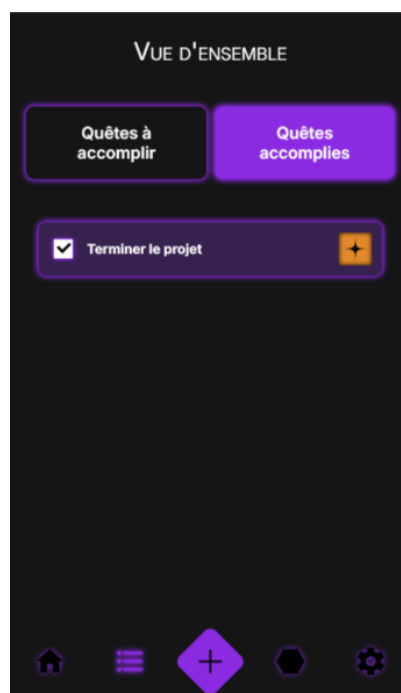
closeQuestModal(): void {
    this._questModalVisible.set(false);
    this._questModalData.set({ quest: this._DEFAULT_QUEST, isNew: false });
}
}

```

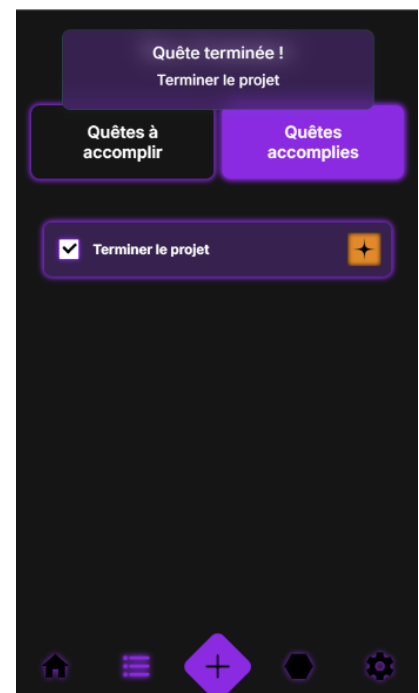
Un affichage standard des quêtes est proposé aux utilisateurs, sous forme de deux listes : l'une pour les quêtes à accomplir, l'autre pour les quêtes accomplies. La navigation se fait via un menu composé de deux onglets. Les quêtes à accomplir sont triées par ordre de priorité.



Liste des quêtes non accomplies.



Listes des quêtes accomplies.



Toast de succès : quête accomplie.

Sur ces listes, l'utilisateur peut voir d'un coup d'oeil le titre de chaque quête ainsi qu'une icône représentant sa priorité, doublée d'un code couleur (orangé pour les principales, argenté pour les secondaires, gris foncé pour les tertiaires). Il dispose également d'un bouton à cocher pour aisément marquer une quête comme accomplie - ce qui déclenche un toast de succès - ou au contraire réhabiliter une quête terminée. Si la quête est indiquée comme "en cours", la barre de progression s'affiche directement sur l'aperçu de la quête, la remplissant progressivement d'une couleur plus sombre. Les quêtes terminées sont entièrement remplies.

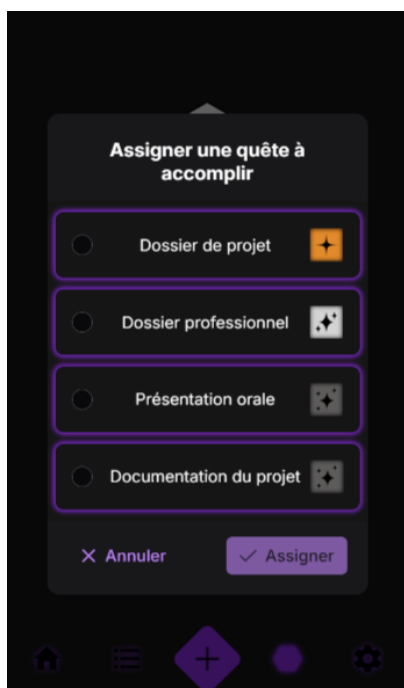
3. Affichage visuel en hexagones (carte)

L'originalité d'Hexapanning réside dans sa représentation visuelle : une carte d'hexagones sur laquelle l'utilisateur peut placer ses quêtes. Chaque hexagone peut accueillir une quête, et un code couleur sur le liseré permet d'identifier rapidement sa priorité (orangé pour les principales, argenté pour les secondaires, et aucun liseré pour les tertiaires). Les quêtes terminées apparaissent avec un fond plus sombre, et les quêtes en cours disposent d'une barre de progression radiale qui remplit progressivement l'hexagone avec cette couleur, à la manière d'une horloge.



Page de la carte d'hexagones.

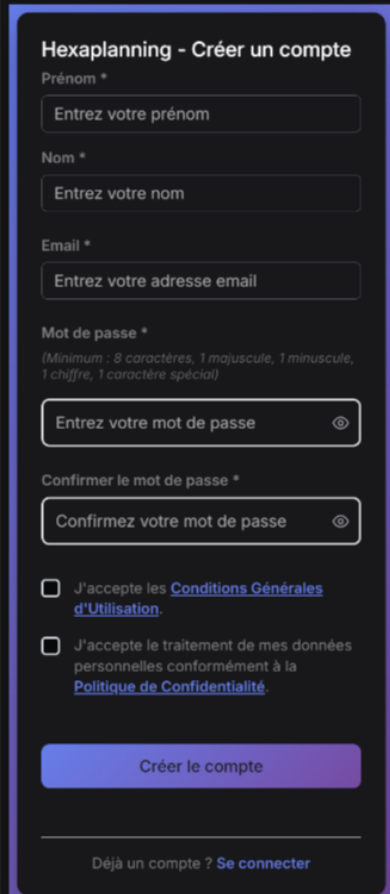
L'utilisateur peut assigner une quête en cliquant ou appuyant sur un hexagone vide, faisant apparaître une modale contenant la liste de toutes les quêtes non accomplies, et en sélectionnant la quête de son choix. Il pourra ensuite la désassigner d'un simple clic sur l'icône de croix au-dessus du titre de la quête, ce qui déclenchera une modale de confirmation.



Tout comme sur les listes des quêtes, il suffit de cliquer ou d'appuyer sur un hexagone associé à une quête pour afficher les détails de la quête en question, et éventuellement modifier ou supprimer la quête (ce qui la fera disparaître de la carte et des listes).

4. Système d'authentification et gestion des utilisateurs

L'accès à l'application nécessite la création d'un compte et une authentification sécurisée. L'utilisateur devra accepter les CGU et la politique de confidentialité, accessibles via des liens sur le formulaire de création de compte. Le mot de passe choisi devra respecter les normes standard : au minimum 8 caractères dont 1 lettre majuscule, 1 lettre minuscule, 1 chiffre et 1 caractère spécial. Après son enregistrement, l'utilisateur sera redirigé vers la page de connexion, et il peut aisément naviguer entre la connexion et la création de compte via un lien en bas de page.

The image shows a mobile application screen for creating an account. The title is 'Hexaplanning - Créer un compte'. It contains several input fields: 'Prénom *' with a placeholder 'Entrez votre prénom', 'Nom *' with 'Entrez votre nom', 'Email *' with 'Entrez votre adresse email', 'Mot de passe *' with a placeholder 'Entrez votre mot de passe' and a password strength note '(Minimum : 8 caractères, 1 majuscule, 1 minuscule, 1 chiffre, 1 caractère spécial)', and 'Confirmer le mot de passe *' with a placeholder 'Confirmez votre mot de passe'. Below these are two checkboxes: 'J'accepte les Conditions Générales d'Utilisation.' and 'J'accepte le traitement de mes données personnelles conformément à la Politique de Confidentialité.'. At the bottom is a large blue button labeled 'Créer le compte' and a link 'Déjà un compte ? Se connecter'.

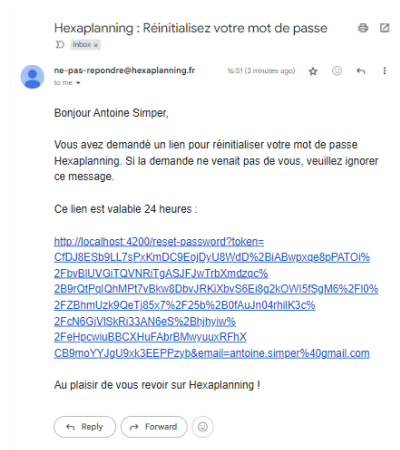
Page de création de compte.

Page de connexion.

Un système de gestion des mots de passe oubliés est en place, avec envoi d'email pour la réinitialisation. Lorsque l'utilisateur clique sur "mot de passe oublié", une modale s'ouvre. Si l'utilisateur avait déjà rentré son adresse e-mail dans le champ de connexion, il sera automatiquement reporté dans le champ de la modale. Au clic sur le bouton d'envoi, un toast informe l'utilisateur qu'un mail a été délivré à l'adresse indiquée, si elle existe. En effet, il s'agit de ne pas confirmer ou infirmer la présence de cette adresse e-mail dans la base de données. De plus, il ne peut y avoir qu'une seule requête vers la même adresse toutes les 5 minutes, afin d'éviter le spam d'une adresse e-mail et la saturation du service de mail.

Modale de mot de passe oublié.

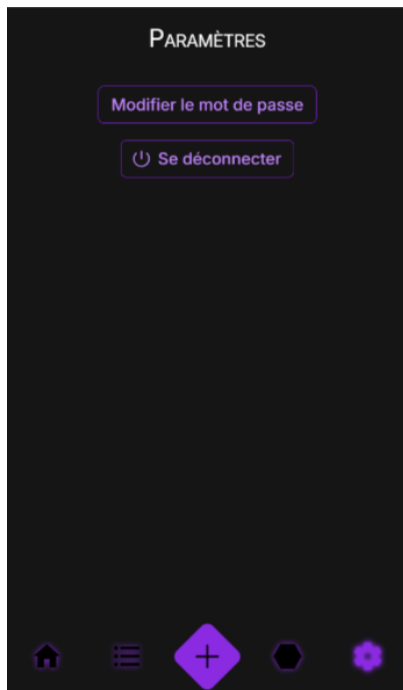
Le destinataire recevra un mail contenant un lien de réinitialisation de mot de passe. Ce lien le redirigera vers la page prévue à cet effet, avec dans l'url un token valable une heure, et l'adresse e-mail du compte à modifier. Sans ces deux éléments valides, la requête ne pourra être acceptée. L'utilisateur n'a plus qu'à rentrer son nouveau mot de passe et à le confirmer, avant d'être redirigé vers la page de connexion.



Mail de réinitialisation de mot de passe.

Page de réinitialisation de mot de passe.

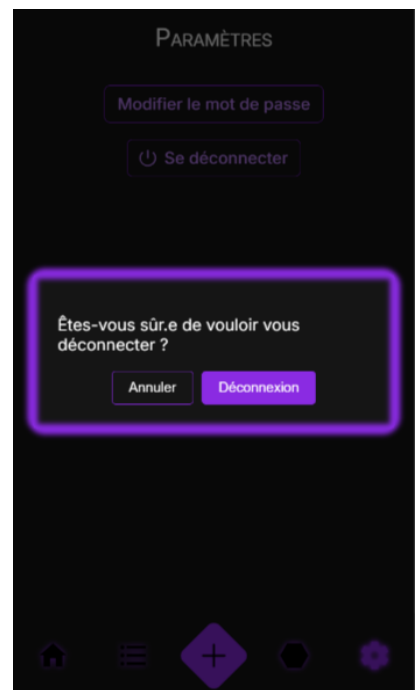
L'utilisateur peut également changer son mot de passe depuis l'interface : en accédant au menu des paramètres, il aura la possibilité d'ouvrir une modale lui demandant son mot de passe actuel ainsi que le nouveau. Depuis ce même menu, il pourra se déconnecter de l'application.



Page de paramètres.



Modale de changement de mot de passe.




Modale de déconnexion.

La sécurité des données et la protection contre les accès non autorisés sont assurées par des mécanismes robustes côté backend.

5. Navigation et ergonomie

L'application propose un menu apparaissant en permanence en bas de page, et permettant de naviguer entre l'accueil, les listes de quêtes, la carte des hexagones et les paramètres. Un bouton dédié au centre du menu permet de créer rapidement une nouvelle quête, qui viendra s'insérer dans la liste qui lui correspond, et sera accessible dans la modale d'assignation à un hexagone.



Modale de création de quête.

```
<div [routerLink]="['/']">
  <i class="material-icons icon" [ngClass]="{ active: activeIcon === 'home'
}">home</i>
</div>
```

```
export class MenuComponent implements OnInit, OnDestroy {
  _questModalService = inject(QuestModalService);
  _router = inject(Router);
  _routerSubscription!: Subscription;

  ngOnInit(): void {
    this.setActiveBasedOnUrl(this._router.url);
    this._routerSubscription = this._router.events.pipe(filter(event => event
instanceof NavigationEnd)).subscribe((event: NavigationEnd) => {
      this.setActiveBasedOnUrl(event.url);
    });
  }

  ngOnDestroy(): void {
    if (this._routerSubscription) {
      this._routerSubscription.unsubscribe();
    }
  }
}
```



```

}

setActiveBasedOnUrl(url: string): void {
  if (url === '/') {
    this.activeIcon = 'home';
  } else if (url.includes('/quest-list')) {
    this.activeIcon = 'quest-list';
  } [else if (url.includes('/map')) {
    this.activeIcon = 'map';
  } else if (url.includes('/settings')) {
    this.activeIcon = 'settings';
  } else {
    this.activeIcon = '';
  }]
}

navigateToRoute(route: string): void {
  this._router.navigate([route]);
}

showNewQuestDialog(): void {
  this._questModalService.openNewQuest();
}
}

```

L'interface est pensée pour être intuitive, responsive et agréable à utiliser, afin de maximiser l'engagement et la productivité de l'utilisateur.

IV. Travail en équipe & méthodologie

1. Méthode Agile / Scrum

Méthodologie adoptée

Le projet Hexapanning a été développé tout d'abord en collaboration, puis en solo. La méthode Agile a été adoptée au fil du projet :

- **Sprints de 2 semaines** : Cycles de développement courts et itératifs
- **User Stories** : Fonctionnalités définies du point de vue utilisateur
- **Backlog Product** : Priorisation des fonctionnalités selon la valeur métier
- **Daily Standup** : Points quotidiens sur l'avancement (adaptés selon disponibilité)

Découpage du projet

Sprint 1 : Fondations

- Réalisation du wireframe et de la maquette
- Mise en place des User Stories
- Configuration de l'environnement de développement
- Architecture de base (frontend Angular + backend .NET)
- Authentification et gestion des utilisateurs

Sprint 2 : Fonctionnalités core

- CRUD des quêtes
- Système de priorités et statuts
- Interface de liste des quêtes

Sprint 3 : Visualisation

- Développement de la carte hexagonale
- Assignation des quêtes aux hexagones
- Interactions et animations

Sprint 4 : Finalisation

- Tests et corrections de bugs
- Documentation
- Déploiement et mise en production

2. Workflow & branches stratégie

Git Workflow adopté

Stratégie de branching :

- **main** : Branche de production, code stable
- **develop** : Branche de développement, intégration des features

- **feature/** : Branches pour chaque nouvelle fonctionnalité
- **hotfix/** : Corrections urgentes sur la production

Processus de développement

```
# Création d'une nouvelle feature
git checkout develop
git pull origin develop
git checkout -b feature/quest-management

# Développement et commits
git add .
git commit -m "feat: add quest creation functionality"

# Push et Pull Request
git push origin feature/quest-management
# Création PR sur GitHub : feature/quest-management -> develop
```

Code Review

- **Pull Requests obligatoires** : Aucun code ne merge sans review
- **Critères de validation** : Tests passants, documentation, respect des conventions
- **Reviewers** : Au moins un autre développeur valide les modifications

3. Outils collaboratifs

Gestion de projet

- **Jira** : Tableau Kanban pour le suivi des tâches

Communication

- **Discord** : Communication instantanée de l'équipe

Documentation partagée

- **Figma** : Maquettes et schémas d'architecture collaboratifs
- **Confluence/Notion** : Spécifications fonctionnelles et notes de réunion

V. Modélisation des données

1. MCD (Modèle Conceptuel de Données)



Schéma de la base de données relationnelle d'Hexaplanning, réalisé avec dbdiagram.io.

2. MLD (Modèle Logique de Données)

- Table **UserApp** (Id PK, FirstName, LastName, Email, PasswordHash, CreatedAt, UpdatedAt, IsArchived, ...)
- Table **Quest** (Id PK, Title, Description, EstimatedTime, Advancement, UserId FK, PriorityId FK, StatusId FK, HexAssignmentId FK, CreatedAt, UpdatedAt, IsArchived)
- Table **Priority** (Id PK, Name, Color, BorderColor, Icon, CreatedAt, UpdatedAt, IsArchived)
- Table **Status** (Id PK, Name, Color, Icon, CreatedAt, UpdatedAt, IsArchived)
- Table **HexAssignment** (Id PK, Q, R, S, QuestId FK, CreatedAt, UpdatedAt, IsArchived)
- Table **Mail** (MailTo, MailSubject, MailBody, MailFrom, Receiver)

3. Description des entités et relations

UserApp (Utilisateur)

- Un utilisateur peut créer plusieurs quêtes. Il possède :
 - Un nom et un prénom.
 - Une adresse e-mail unique.
 - Un mot de passe (hashé dans la base de données).
 - Une liste de quêtes.
 - Des métadonnées : date de création, date de mise à jour, statut d'archivage.

Quest (Quête)

- Une quête appartient à un seul utilisateur. Elle possède :
 - Un titre (limité à 100 caractères).
 - Optionnellement, une description.
 - Optionnellement, un temps estimé.
 - Un pourcentage d'avancement (Advancement) pour les quêtes en cours.
 - Un UserId pour la rattacher à son utilisateur.
 - Un PriorityId pour définir sa priorité.
 - Un StatusId pour définir son statut.
 - Optionnellement, un HexAssignmentId pour l'assigner à un hexagone.
 - Des métadonnées : date de création, date de mise à jour, statut d'archivage.

Priority (Priorité)

- Une priorité peut être associée à plusieurs quêtes. Elle possède :
 - Un nom (PRIMARY, SECONDARY, TERTIARY).
 - Une couleur principale.
 - Une couleur de bordure (BorderColor) pour l'affichage sur la carte.
 - Optionnellement, une icône.
 - Des métadonnées : date de création, date de mise à jour, statut d'archivage.

Status (Statut)

- Un statut peut être associé à plusieurs quêtes. Il possède :
 - Un nom (en attente, en cours, terminée).
 - Une couleur pour l'affichage.
 - Optionnellement, une icône.
 - Des métadonnées : date de création, date de mise à jour, statut d'archivage.

HexAssignment (Assignation d'hexagone)

- Un hexagone (HexAssignment) est lié à une seule quête. Il possède :
 - Un jeu de coordonnées q, r, s qui lui est unique (système de coordonnées hexagonales).
 - Un QuestId pour la quête assignée.
 - Des métadonnées : date de création, date de mise à jour, statut d'archivage.

Mail

- Un mail est indépendant et permet d'envoyer des communications (réinitialisation de mot de passe, bienvenue, etc.). Il possède :
 - Un destinataire (MailTo).
 - Un sujet (MailSubject).
 - Un corps de message (MailBody).
 - Un expéditeur (MailFrom).
 - Un destinataire lié à un utilisateur (Receiver).

Relations principales :

- **UserApp 1:N Quest** : Un utilisateur possède plusieurs quêtes.
- **Quest N:1 Priority** : Une quête a une priorité.
- **Quest N:1 Status** : Une quête a un statut.
- **Quest 1:1 HexAssignment** : Une quête peut être assignée à un hexagone (optionnel).

VI. Architecture technique et technologies

1. Vue d'ensemble

Hexaplaning adopte une architecture moderne en trois couches avec une séparation claire des responsabilités. Le choix des technologies s'est fait en privilégiant la robustesse, la maintenabilité et l'écosystème de chaque solution. L'architecture repose sur :

- **Frontend** : Angular 18 avec PrimeNG pour une interface utilisateur moderne et responsive
- **Backend** : ASP.NET Core 8 pour une API REST performante et sécurisée
- **Base de données** : PostgreSQL pour la persistance des données
- **Infrastructure** : Docker et GitHub Actions pour le déploiement et l'intégration continue
- **Services externes** : Brevo pour l'envoi d'e-mails transactionnels

Cette approche modulaire facilite la maintenance, l'évolutivité et la sécurité de l'application. La communication entre les couches s'effectue via une API REST sécurisée par JWT.

Schéma global

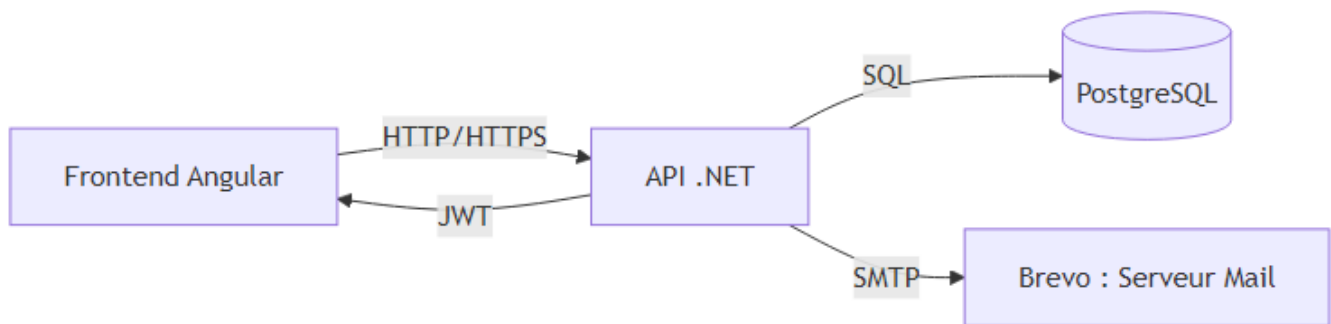


Schéma global de l'architecture d'Hexaplaning.

2. Frontend : Angular et PrimeNG

Choix technologiques et justifications

- **Angular 18** : Framework SPA reconnu pour sa structure modulaire, sa maintenabilité et sa communauté active. Il facilite la création d'interfaces dynamiques, responsives et testables. Le choix de la version 18 apporte les dernières optimisations de performance et les nouveautés du framework. L'utilisation de Typescript facilite la maintenance et réduit les erreurs de développement.
- **PrimeNG** : Bibliothèque de composants UI riche et moderne pour Angular, fournissant les éléments d'interface (modales, formulaires, boutons, toasts) avec un design cohérent et professionnel. Alternative considérée : Angular Material, mais PrimeNG est plus intuitif et facilement personnalisable.

Architecture et organisation

- **Structure modulaire** : Organisation en pages faisant appel à des composants réutilisables, des services, des pipes et des modèles de DTO. Routes avec guards et interceptors.
- **Approche mobile-first** : Interface responsive optimisée pour les appareils mobiles.

Responsabilités principales

- Gestion de l'interface utilisateur et de l'expérience utilisateur
- Navigation entre les différentes pages et modales
- Appels API vers le backend et gestion des réponses
- Gestion du token JWT pour l'authentification
- Affichage dynamique de la carte d'hexagones avec coordonnées hexagonales
- CRUD des quêtes avec validation côté client

Sécurité

- **Intercepteur HTTP** : Ajout automatique du JWT dans toutes les requêtes API
- **Guards de navigation** : Protection des routes sensibles (authentification requise)
- **Validation des formulaires** : Contrôles côté client avant envoi au backend

Tests et qualité

- **ESLint** : Analyse statique du code pour maintenir la qualité

Conventions Angular respectées

- **kebab-case** pour les sélecteurs : `app-quest-card`
- **camelCase** pour les propriétés : `questTitle`, `isCompleted`
- **PascalCase** pour les classes : `QuestComponent`, `QuestService`

3. Backend : .NET Core

Choix technologiques et justifications

- **ASP.NET Core 8** : Framework backend performant, sécurisé et multiplateforme, idéal pour exposer une API REST robuste et scalable. La version 8 LTS garantit la stabilité et le support à long terme.
- **Entity Framework Core** : ORM facilitant la gestion et la migration de la base de données, tout en assurant la cohérence des modèles.
- **ASP.NET Identity** : Système d'authentification et d'autorisation intégré, robuste et éprouvé pour la gestion des utilisateurs et des mots de passe.

Architecture en couches

L'API suit une architecture en couches claire pour séparer les responsabilités :

- **Controllers** : Points d'entrée API, gestion des requêtes HTTP et des réponses
- **Services** : Logique métier, règles de gestion et orchestration des opérations
- **Models** : Entités de domaine et DTOs pour le transfert de données
- **DataContext** : Couche d'accès aux données avec Entity Framework
- **Utilities** : Classes utilitaires et helpers transversaux

Modèle générique BaseModel

L'architecture utilise un **modèle générique d'héritage** pour standardiser les entités et éviter la duplication de code :

BaseModel - Classe abstraite commune :

```
public abstract class BaseModel
{
    public int Id { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime UpdatedAt { get; set; }
    public bool IsArchived { get; set; }
}
```

BaseModelOption - Pour les options de priorité et de statut des quêtes :

```
public abstract class BaseModelOption : BaseModel
{
    public string Name { get; set; }
    public string Color { get; set; }
    public string Icon { get; set; }
}
```

Utilisation dans les entités métier (exemples avec une partie de la classe Quest, et avec la classe Priority) :

```
public class Quest : BaseModel
{
    public string Title { get; set; }
    public string Description { get; set; }
    public int UserId { get; set; }
    public int PriorityId { get; set; }
    public int StatusId { get; set; }
    // Propriétés héritées automatiquement : Id, CreatedAt, UpdatedAt, IsArchived
}

public class Priority : BaseModelOption
{
    public string BorderColor { get; set; }
    // Propriétés héritées : Id, Name, Color, Icon, CreatedAt, UpdatedAt, IsArchived
}
```


Avantages de cette approche :

- **Cohérence** : Toutes les entités partagent les mêmes métadonnées
- **Maintenance** : Modifications centralisées dans BaseModel
- **Audit** : Traçabilité automatique (CreatedAt, UpdatedAt)
- **Préparation aux améliorations futures** : avec une option d'archivage.

Sécurité intégrée

- **Middleware JWT** : Authentification automatique sur tous les endpoints protégés

Exemple dans le contrôleur de quêtes, avec l'attribut Authorize :

```
/// <summary>
/// Gestion des quêtes
/// </summary>
[Route("[controller]")]
[Authorize]
[ApiController]
[CheckUser]
public class QuestController : ControllerBase
{}
```

- **Validation des entrées** : Contrôles sur toutes les données reçues

Exemple dans une partie du DTO de création de quête avec un titre qui doit être une chaîne de caractères limitée à 100 caractères :

```
public class QuestCreateDTO
{
    [Required]
    [StringLength(100)]
    public string Title { get; set; }

    public string? Description { get; set; }
}
```


- **Protection anti-attaques** : Protection contre l'injection SQL en utilisant l'ORM Entity Framework

Exemple avec la méthode de récupération de toutes les quêtes d'un utilisateur donnée, dans le QuestService :

```
/// <summary>
/// Récupère toutes les quêtes d'un utilisateur spécifique.
/// </summary>
/// <param name="userId">L'identifiant unique (GUID) de l'utilisateur.</param>
/// <returns>
/// Une liste de <see cref="QuestDTO"/> contenant toutes les quêtes associées à
/// l'utilisateur.
/// Retourne une liste vide si l'utilisateur n'a aucune quête.
/// </returns>
public async Task<List<QuestDTO>> GetAllQuestsAsync(Guid userId)
{
    var quests = await context.Quests.Where(x => x.UserId ==
userId).ToListAsync();
    return quests.Select(QuestDTO.ToQuestDTO).ToList();
}
```

- **Rate limiting** : Protection contre les tentatives de force brute

On utilise pour cela les options de Lockout fournies par ASP .NET Identity, qu'on configure dans Program.cs :

```
services.Configure<IdentityOptions>(options =>
{
    options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(2);
    options.Lockout.MaxFailedAccessAttempts = 5;
    options.Lockout.AllowedForNewUsers = true;
});
```


- **Gestion des droits** : Chaque utilisateur n'accède qu'à ses propres données

Mécanisme CheckUser - Isolation des données utilisateur

L'API implémente un **système de vérification automatique** (**CheckUser**) de façon à ce que chaque utilisateur ne puisse accéder qu'à ses propres ressources :

```
public class CheckUserAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(ActionExecutingContext context)
    {
        var user = context.HttpContext.User;
        var userId = CheckUser.GetUserIdFromClaim(user);

        if (!userId.HasValue)
        {
            context.Result = new UnauthorizedResult();
            return;
        }

        context.HttpContext.Items["UserId"] = userId.Value;

        base.OnActionExecuting(context);
    }
}
```

On récupère l'identifiant de l'utilisateur qui est un des claims du token JWT transmis avec la requête HTTP. On le stocke alors dans le contexte HTTP, pour pouvoir y accéder dans les méthodes qui utilisent l'attribut CheckUser, et vérifier que l'identifiant correspond aux données auxquelles on souhaite accéder.

Exemple d'application dans le contrôleur de quêtes :

```
[HttpGet]
public async Task<IActionResult> GetAllQuests()
{
    if (HttpContext.Items["UserId"] is Guid userId)
    {
        var quests = await questService.GetAllQuestsAsync(userId);
        return Ok(quests);
    }
    return Unauthorized();
}
```

Chaque contrôleur qui comporte des méthodes nécessitant de vérifier l'utilisateur dont émane la requête est décoré par l'attribut [CheckUser].

Avantages du système CheckUser :

- **Sécurité renforcée** : Impossible d'accéder aux données d'autres utilisateurs
- **Validation automatique** : Contrôle systématique sur toutes les opérations
- **Code centralisé** : Logique de vérification réutilisable dans tous les contrôleurs
- **Performance** : Vérification rapide basée sur les claims JWT

Tests et qualité

- **xUnit** : Framework de tests unitaires moderne et flexible, intégré à l'écosystème .NET
- **Tests d'intégration** : Validation complète des endpoints avec base de données de test, qui utilisent des Testcontainers pour générer une base de données PostgreSQL et effectuer une validation réelle.

Standards de développement et qualité du code

Programmation orientée objet respectée :

L'architecture backend .NET Core respecte les principes OOP :

- **Encapsulation** : Propriétés privées avec validation dans les setters
- **Héritage** : Classes `BaseModel` et `BaseModelOption` pour standardiser les entités

Conventions de nommage C# :

- **PascalCase** pour classes et méthodes : `QuestService`, `GetQuestById`
- **camelCase** pour variables locales : `questDto`, `userId`
- **Constantes en UPPER_CASE** : `MAX_QUEST_TITLE_LENGTH`

Documentation XML pour .NET :

Chaque méthode de contrôleur est documentée, ce qui ajoute une description aux endpoints sur Swagger.

Exemple avec la méthode de création de quête :

```
/// <summary>
/// Crée une nouvelle quête pour l'utilisateur authentifié.
/// </summary>
/// <param name="questDto">Les données de la quête à créer.</param>
/// <returns>
/// Une réponse HTTP 201 Created contenant la quête créée avec son identifiant
unique,
/// ainsi qu'un en-tête Location pointant vers l'endpoint de récupération de la
quête.
/// </returns>
/// <response code="201">La quête a été créée avec succès. Retourne la quête avec
son ID généré.</response>
/// <response code="400">Les données fournies sont invalides (validation échouée).
</response>
/// <response code="401">L'utilisateur n'est pas authentifié ou le token JWT est
invalidé.</response>
/// <response code="409">Une quête est déjà associée à cet hexagone (contrainte
d'unicité violée).</response>
[HttpPost]
public async Task<IActionResult> CreateQuest([FromBody] QuestCreatedDTO questDto)
{ }
```

4. Base de données : PostgreSQL

Choix technologique et justifications

- **PostgreSQL** : SGBD open source reconnu pour sa fiabilité et ses performances, plus robuste que MySQL par exemple, si l'application continue d'évoluer.

Modélisation et structure

- **Respect du MCD/MLD** : Implementation fidèle du modèle conceptuel présenté au chapitre III
- **Contraintes d'intégrité** : Clés étrangères, contraintes CHECK et UNIQUE pour la cohérence des données

Gestion et évolution

- **Migrations Entity Framework Core** : Versioning automatique du schéma de base de données
- **Code-First approach** : Génération du schéma à partir des modèles C#
- **Seeding** : Données initiales (priorités, statuts) injectées automatiquement

Sécurité

- **Accès restreint** : Connexion uniquement via l'API backend.
- **Isolation des données** : Chaque utilisateur accède uniquement à ses propres données
- **Mots de passe sécurisés** : Hashés avec ASP.NET Identity

5. Communication API REST

Architecture RESTful

- **Format de données** : JSON via HTTP(S) pour tous les échanges
- **Verbes HTTP** : Utilisation sémantique (GET, POST, PUT, DELETE)
- **Codes de réponse** : Status codes HTTP appropriés (200, 201, 400, 401, 404, 500)
- **Structure des URLs** : Routes RESTful cohérentes (`/quests`, `/users/{id}`)

Endpoints principaux

- **Authentification** : `/auth/login`, `/auth/register`, `/auth/reset-password`
- **Gestion des quêtes** : CRUD complet sur `/quests` avec filtrage par utilisateur
- **Gestion des hexagones** : `/hexassignments` pour l'assignation des quêtes
- **Gestion des utilisateurs** : `/users` pour les profils et paramètres
- **Données de référence** : `/priorities`, `/statuses` pour les listes déroulantes

Sécurité et authentification

- **JWT Bearer Token** : Toutes les routes sensibles protégées par authentification
- **CORS configuré** : Origines autorisées limitées aux domaines de l'application
- **Validation des données** : Contrôles stricts sur tous les inputs API

Gestion des erreurs

- **Réponses structurées** : Format JSON consistant pour les erreurs
- **Messages explicites** : Informations claires pour le débogage côté frontend

6. Infrastructure et DevOps

Conteneurisation

- **Docker** : Conteneurisation de chaque composant pour garantir la portabilité, l'isolation et la reproductibilité des environnements. Chaque service (frontend, backend, base de données) dispose de son propre Dockerfile optimisé.
- **docker-compose** : Orchestration simplifiée du déploiement multi-conteneurs. Gestion des dépendances entre services, des variables d'environnement et des volumes persistants.

Intégration et déploiement continu

- **GitHub Actions** : Automatisation des pipelines CI/CD. Pipelines séparés pour le frontend et le backend avec tests automatisés.
- **Workflow CI** : Tests unitaires et d'intégration automatiques avant chaque déploiement
- **Workflow CD** : Build, push vers Docker Hub et déploiement automatique sur le VPS

Hébergement et infrastructure

- **OVH VPS** : Hébergement flexible et sécurisé, adapté à la montée en charge. Serveur Linux Ubuntu avec Docker et docker-compose installés.
- **Nginx Proxy Manager** : Gestion centralisée des domaines, des certificats SSL et du reverse proxy. Interface web pour la configuration des routes et des certificats Let's Encrypt automatiques.

7. Services externes

Brevo (ex-Sendinblue)

- **Service d'emailing transactionnel** : Solution cloud fiable et simple à intégrer pour l'envoi d'e-mails automatisés
- **Utilisation** : Envoi de mails de réinitialisation de mot de passe
- **Avantages** : API simple et tarification adaptée aux petits volumes, plus simple et plus économique qu'un serveur mail à héberger

Le service d'envoi de mail est pour le moment utilisé uniquement pour la réinitialisation de mot de passe.

Une vérification est effectuée avant l'envoi, pour s'assurer que la requête n'est pas envoyée à la même adresse plus d'une fois toutes les 5 minutes. Pour cela on crée une variable :

```
private static readonly Dictionary<string, DateTime> _lastResetRequest = new();
```

Au début de la méthode, on vérifie que la dernière requête n'a pas été effectuée il y a peu, puis on configure le client SMTP, on cherche l'utilisateur associé à l'adresse e-mail renseignée (s'il n'est pas trouvé, on renvoie tout de même un message de succès pour ne pas révéler si l'adresse existe en base de données), puis on génère un token de réinitialisation de mot de passe qu'on insère dans un lien avec l'adresse e-mail, de façon à le transmettre dans le template de mail.


```

public async Task<bool> SendPasswordResetEmail(string emailAddress)
{
    try
    {
        if (_lastResetRequest.TryGetValue(emailAddress, out var lastRequest))
        {
            if (DateTime.UtcNow - lastRequest < TimeSpan.FromMinutes(5)) return
true;
        }
        _lastResetRequest[emailAddress] = DateTime.UtcNow;

        var smtpClient = new SmtpClient(Env.SMTP_HOST)
        {
            Port = Int32.Parse(Env.SMTP_PORT),
            Credentials = new NetworkCredential(
                Env.SMTP_EMAILFROM,
                Env.SMTP_PASSWORD
            ),
            EnableSsl = true
        };

        var user = await _userManager.FindByEmailAsync(emailAddress);
        if (user == null)
        {
            // Pour des raisons de sécurité, on ne révèle pas si l'email existe
            return true;
        }

        var resetToken = await _userManager.GeneratePasswordResetTokenAsync(user);
        var encodedToken = Uri.EscapeDataString(resetToken);
        var resetLink = $"{Env.API_FRONT_URL}/reset-password?token=
{encodedToken}&email={Uri.EscapeDataString(emailAddress)}";

        var mailMessage = new MailMessage
        {
            // Template du mail
        };

        mailMessage.To.Add(user.Email);

        await smtpClient.SendMailAsync(mailMessage);

        return true;
    }
    catch
    {
        throw;
    }
}

```


VII. Qualité logicielle et tests

La qualité logicielle d'Hexapanning repose sur une stratégie de tests complète, principalement concentrée sur l'API .NET, afin de garantir la robustesse, la fiabilité et la maintenabilité du backend.

1. Tests unitaires (backend)

Les tests unitaires sont réalisés avec xUnit et couvrent les principaux services métiers, notamment le service de gestion des quêtes (`QuestService`). Ces tests vérifient le bon fonctionnement des méthodes de création, lecture, mise à jour et suppression de quêtes, ainsi que la gestion des cas limites (identifiants invalides, absence de données, etc.).

Exemples de méthodes testées :

- Création d'une quête (`CreateQuestAsync`)
- Récupération d'une quête par ID (`GetQuestByIdAsync`)
- Mise à jour et suppression de quêtes (`UpdateQuestAsync`, `DeleteQuestAsync`)
- Récupération des quêtes selon leur statut (en attente, terminées, non assignées)

2. Tests d'intégration

Des tests d'intégration automatisés valident l'ensemble du pipeline API, de la couche HTTP jusqu'à la base de données PostgreSQL (via Testcontainers). Ils simulent des scénarios réels, comme la récupération de quêtes via des requêtes authentifiées, la gestion des droits d'accès, et la cohérence des données persistées.

Caractéristiques :

- Utilisation de `WebApplicationFactory` pour lancer l'API en environnement de test
- Base de données PostgreSQL éphémère (Testcontainers)
- Données de test injectées automatiquement (utilisateur, quêtes)

3. Tests de charge et fixtures

Des fixtures de données sont utilisées pour simuler des volumes importants de quêtes et d'utilisateurs, grâce à la librairie Bogus. Cela permet de valider la tenue en charge de l'API et la stabilité des traitements sur de grands ensembles de données. Les tests ont été réalisés avec 100000 utilisateurs et 1000000 de quêtes pour s'assurer de la robustesse de la base de données.

4. Stratégie de validation

Chaque nouvelle fonctionnalité ou correction de bug s'accompagne de tests dédiés. Les tests sont exécutés automatiquement lors des pipelines CI/CD (GitHub Actions), garantissant l'absence de régressions avant chaque déploiement.

Cette démarche assure un haut niveau de confiance dans la qualité logicielle du backend, tout en facilitant l'évolution continue du projet.

VIII. CI / CD

L'automatisation du déploiement et de l'intégration continue est assurée par des pipelines GitHub Actions distincts pour le frontend Angular et l'API .NET.

1. Intégration continue (CI) de l'API

Un pipeline CI dédié à l'API .NET s'exécute à chaque push sur la branche `main`:

- **Tests unitaires**: Compilation et exécution des tests unitaires (`dotnet test ./TestsUnitaires`)
- **Tests d'intégration**: Lancement des tests d'intégration sur une base PostgreSQL éphémère (`dotnet test ./TestsIntegration`)
- **Vérification de la qualité**: Toute régression ou échec bloque la suite du pipeline

Extrait du workflow:

```
jobs:
  test-unitaire:
    ...
    - run: dotnet test --no-build --verbosity normal ./TestsUnitaires
  test-integration:
    ...
    - run: dotnet test --no-build --verbosity detailed ./TestsIntegration
```

2. Déploiement continu (CD) du backend

Le backend .NET dispose également d'un pipeline CD automatisé. Celui-ci ne se déclenche que si le pipeline CI de l'API s'est terminé avec succès (`workflow_run`). Il effectue les étapes suivantes:

- **Build Docker**: Construction de l'image Docker de l'API
- **Push Docker**: Publication de l'image sur Docker Hub
- **Déploiement VPS**: Connexion SSH au serveur OVH, pull de la nouvelle image et redémarrage du conteneur backend via `docker compose`

Extrait du workflow:

```
on:
  workflow_run:
    workflows: ["CI pipeline for the API"]
    types:
      - completed
jobs:
  build-and-deploy:
    ...
    - run: docker build -t antoinespr/hexaplanning-api:dev1 .
    - run: docker push antoinespr/hexaplanning-api:dev1
    - uses: appleboy/ssh-action@v1.0.0
      with:
        script: |
          docker pull antoinespr/hexaplanning-api:dev1
          docker compose -f /home/ubuntu/backend/docker-compose.yml up -d --
force-recreate
```

3. Déploiement continu (CD) du frontend

Le frontend Angular dispose d'un pipeline CD qui automatise la construction, la publication et le déploiement sur le serveur de production:

- **Build Docker:** Construction de l'image Docker de l'application Angular
- **Push Docker:** Publication de l'image sur Docker Hub
- **Déploiement VPS:** Connexion SSH au serveur OVH, pull de la nouvelle image et redémarrage du conteneur via `docker compose`

Extrait du workflow:

```
jobs:
  deploy:
    ...
    - run: docker build --target prod-runtime -t antoinespr/hexaplanning-
front:dev1 .
    - run: docker push antoinespr/hexaplanning-front:dev1
    - uses: appleboy/ssh-action@v1.0.0
      with:
        script: |
          docker pull antoinespr/hexaplanning-front:dev1
          docker compose -f /home/ubuntu/frontend/docker-compose.yml up -d --
force-recreate
```

4. Conteneurisation et orchestration

Chaque composant (frontend, backend, base de données) dispose de son propre Dockerfile. Le déploiement s'effectue via `docker compose`, facilitant la gestion, la montée en charge et la maintenance.

5. Hébergement et reverse proxy

L'application est hébergée sur un VPS OVH, avec Nginx Proxy Manager pour la gestion des domaines et des certificats SSL. Cette architecture assure la sécurité, la disponibilité et la scalabilité du service.

Cette chaîne CI/CD garantit des livraisons rapides, sûres et automatisées, tout en limitant les interventions manuelles et les risques d'erreur.

Le résultat final est disponible sous le nom de domaine hexaplanning.fr.

Outils qualité et automatisation

ESLint et Prettier (Frontend) :

```
{
  "scripts": {
    "lint": "ng lint",
    "lint:fix": "ng lint --fix",
    "format": "prettier --write \"src/**/*.{ts,html,scss}\"",
    "format:check": "prettier --check \"src/**/*.{ts,html,scss}\""
  }
}
```


IX. Sécurité

L'application implémente une stratégie de sécurité multicouche couvrant l'authentification, la protection des données et la sécurisation de l'infrastructure.

1. Authentification et gestion des accès

Framework d'authentification

- **ASP.NET Identity** : Framework robuste intégré à .NET Core pour la gestion complète des utilisateurs
- **JWT (JSON Web Tokens)** : Authentification stateless sécurisée avec signature cryptographique. Toutes les opérations sensibles nécessitent un token JWT, généré lors de la connexion et vérifié à chaque requête côté backend
- **Guards et Intercepteurs** : Le frontend Angular utilise des guards pour protéger les routes et un intercepteur HTTP pour injecter automatiquement le token dans les requêtes API

Exemple de guards sur la route principale qui renvoie au dashboard (accessible uniquement si l'utilisateur est authentifié), et sur la page de login qui n'est accessible que si l'utilisateur n'est pas authentifié :

```
export const routes: Routes = [  
  {  
    canActivate: [isLoggedInGuard],  
    path: '',  
    component: DashboardPageComponent,  
  },  
  {  
    canActivate: [isLoggedInGuard],  
    path: 'login',  
    component: LoginComponent,  
  },  
]
```

Et le code du premier guard :

```
export const isLoggedInGuard: CanActivateFn = (route, state) => {  
  const _router = inject(Router);  
  
  if (localStorage.getItem('user') !== null && localStorage.getItem('token') !==  
null) {  
    return true;  
  }  
  _router.navigate(['/login']);  
  return false;  
};
```


Intercepteur pour insérer le token d'authentification dans toutes les requêtes HTTP :

```
export const tokenInterceptor: HttpInterceptorFn = (req, next) => {
  const token = localStorage.getItem('token');
  if (token) {
    req = req.clone({
      headers: {
        Authorization: `Bearer ${token}`,
      },
    });
  }
  return next(req);
};
```

Gestion sécurisée des mots de passe

- **Hachage des mots de passe** : Utilisation d'algorithmes sécurisés (PBKDF2) avec salage automatique
- **Politique de complexité** : Validation des mots de passe selon les standards de sécurité : au minimum 8 caractères, 1 lettre majuscule, 1 lettre minuscule, 1 chiffre, 1 caractère spécial
- **Réinitialisation sécurisée** : Tokens temporaires à usage unique pour la récupération de mot de passe via email (Brevo)
- **Protection contre la force brute** : Limitation du nombre de tentatives de connexion

ASP .NET Identity est configuré dans Program.cs avec le code suivant, assurant la validation des mots de passe dans le backend et la protection contre le brute force :

```
services.Configure<IdentityOptions>(options =>
{
  // Password settings
  options.Password.RequireDigit = true;
  options.Password.RequireLowercase = true;
  options.Password.RequireNonAlphanumeric = true;
  options.Password.RequireUppercase = true;
  options.Password.RequiredLength = 8;
  options.Password.RequiredUniqueChars = 1;

  // Lockout settings
  options.Lockout.DefaultLockoutTimeSpan = TimeSpan.FromMinutes(2);
  options.Lockout.MaxFailedAccessAttempts = 5;
  options.Lockout.AllowedForNewUsers = true;

  // User settings
  options.User.AllowedUserNameCharacters =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789-._@+";
  options.User.RequireUniqueEmail = true;
});
```


2. Validation et intégrité des données

Validation des entrées

- **Validation systématique** : Toutes les entrées utilisateur sont validées côté backend (.NET) pour éviter les injections, incohérences ou données malformées
- **Gestion des erreurs** : Messages d'erreur génériques pour éviter la fuite d'informations sensibles, notamment lors de la réinitialisation de mot de passe

Isolation des données utilisateur

- **Mécanisme CheckUser** : Système de vérification automatique garantissant que chaque utilisateur ne peut accéder qu'à ses propres ressources
- **Principe du moindre privilège** : Accès limité aux ressources strictement nécessaires

Exemple d'endpoint spécialisé : récupération des quêtes en attente et non assignées de l'utilisateur ayant effectué la requête (pour les afficher dans la liste des quêtes à assigner à un hexagone), ce qui évite de refaire un tri en front-end.

```
[HttpGet("unassigned_pending")]
public async Task<IActionResult> GetAllUnassignedPendingQuests()
{
    if (HttpContext.Items["UserId"] is Guid userId)
    {
        var unassigned_pending_quests = await
questService.GetAllUnassignedPendingQuestsAsync(userId);
        return Ok(unassigned_pending_quests);
    }
    return Unauthorized();
}
```

3. Protection contre les attaques

Attaques web courantes

- **SQL Injection** : Utilisation d'Entity Framework avec requêtes paramétrées exclusivement

Configuration sécurisée

- **CORS restrictif** : Configuration précise des origines autorisées pour les requêtes cross-origin

```
static void ConfigureCors(IServiceCollection services)
{
    services.AddCors(options =>
    {
        options.AddDefaultPolicy(builder =>
        {
            builder
                .SetIsOriginAllowed(IsOriginAllowed)
                .AllowAnyMethod()
                .AllowAnyHeader()
                .AllowCredentials();
        });
    });
}
```

```
static bool IsOriginAllowed(string origin)
{
    List<string> localUrls =
        new()
        {
            "http://hexaplanning.fr",
            "https://hexaplanning.fr",
            "http://api.hexaplanning.fr",
            "https://api.hexaplanning.fr",
            Env.API_BACK_URL,
            Env.API_FRONT_URL,
        };
    return localUrls.Contains(origin);
}
```

4. Sécurité de la conteneurisation et du déploiement

Infrastructure sécurisée

- **HTTPS obligatoire** : Chiffrement TLS 1.2+ en production avec redirection automatique
- **Reverse proxy** : Nginx Proxy Manager gère les certificats SSL et protège l'accès aux services
- **Isolation des conteneurs** : Docker avec utilisateurs non-privilégiés et réseaux isolés

Gestion des secrets

- **Variables d'environnement** : Secrets stockés de manière sécurisée, jamais dans le code source

La sécurité est intégrée à tous les niveaux de l'architecture d'Hexaplanning pour garantir la confidentialité, l'intégrité et la disponibilité des données utilisateurs.

X. Accessibilité et conformité RGAA

L'accessibilité numérique est un enjeu majeur pour Hexapanning, permettant à tous les utilisateurs, y compris ceux en situation de handicap, d'accéder pleinement aux fonctionnalités de l'application. Ce chapitre détaille les mesures d'accessibilité implémentées dans l'application.

1. Conformité RGAA et standards d'accessibilité

Standards respectés

L'application Hexapanning a été développée en tenant compte des recommandations d'accessibilité suivantes :

- **RGAA 4.1** : Référentiel français d'accessibilité numérique
- **WCAG 2.1** : Web Content Accessibility Guidelines
- Contraste de couleurs suffisant
- Navigation au clavier
- Structure sémantique HTML

Focus management global

Une gestion globale du focus a été implémentée pour améliorer la navigation au clavier :

```
*:focus {  
  outline: 2px solid #667eea;  
  outline-offset: 2px;  
  box-shadow: 0 0 0 2px #667eea !important;  
}
```

Cette règle CSS garantit que tous les éléments focalisables ont un indicateur visuel clair et visible.

2. Accessibilité des formulaires

Formulaire de connexion

Le formulaire de connexion implémente plusieurs bonnes pratiques d'accessibilité :

Attributs sémantiques et ARIA :

- **role="form"** : Identification claire du formulaire
- **aria-label** : Description accessible du formulaire
- **autocomplete** : Assistance à la saisie pour les champs email et mot de passe
- **aria-describedby** : Association avec les messages d'erreur
- **aria-invalid** : État de validation dynamique
- **aria-live="polite"** : Annonce des erreurs de validation

Exemple d'implémentation :

```
<form [formGroup]="loginForm" (ngSubmit)="onSubmit()" role="form" aria-label="Formulaire de connexion">
  <div class="form-field">
    <label for="email" class="form-label">Email *</label>
    <input
      id="email"
      type="email"
      formControlName="email"
      [attr.aria-describedby]="hasEmailError ? 'email-error' : null"
      [attr.aria-invalid]="hasEmailError"
      autocomplete="email" />
    @if (hasEmailError) {
      <small class="p-error" id="email-error" role="alert" aria-live="polite"> {{
emailError }} </small>
    }
  </div>
</form>
```

Formulaire d'inscription

Le formulaire d'inscription étend les fonctionnalités d'accessibilité :

Structure sémantique avancée :

- **<fieldset> et <legend>** : Regroupement sémantique des conditions d'utilisation
- **Classes .visually-hidden** : Labels cachés visuellement mais accessibles aux lecteurs d'écran
- **Descriptions détaillées** : Exigences de mot de passe clairement indiquées

Implémentation des conditions d'utilisation :

```
<fieldset class="checkbox-fieldset">
  <legend class="visually-hidden">Acceptation des conditions</legend>

  <div class="form-field">
    <div class="checkbox-container">
      <p-checkbox formControlName="acceptCgu" inputId="acceptCgu" [attr.aria-
describedby]="hasAcceptCguError ? 'acceptCgu-error' : null">
      </p-checkbox>
      <label for="acceptCgu" class="checkbox-label">
        J'accepte les
        <a routerLink="/cgu" target="_blank" aria-label="Conditions Générales
d'Utilisation (ouvre dans un nouvel onglet)">
          Conditions Générales d'Utilisation </a>
        >.
      </label>
    </div>
  </div>
</fieldset>
```

Modale de mot de passe oublié

La modale de récupération de mot de passe utilise des attributs ARIA appropriés :

```
<p-dialog
[(visible)]="forgotPasswordModalVisible"
[modal]="true"
role="dialog"
aria-labelledby="forgot-password-title"
aria-describedby="forgot-password-description">
  <ng-template pTemplate="header">
    <h3 id="forgot-password-title">Mot de passe oublié</h3>
  </ng-template>

  <div class="forgot-password-container">
    <p id="forgot-password-description">Entrez votre adresse email et nous vous
enverrons un lien...</p>
  </div>
</p-dialog>
```


3. Navigation au clavier et focus management

Navigation dans la carte hexagonale

La carte hexagonale supporte la navigation au clavier avec des attributs ARIA appropriés :

```
<polygon
  [attr.points]="getHexPoints(h.cx, h.cy)"
  (click)="handleHexClick(h)"
  (keydown)="handleHexKeydown($event, h)"
  tabindex="0"
  role="button"
  [attr.aria-label]="getHexAriaLabel(h)"
  class="hex-polygon" />
```

Fonctionnalités implémentées :

- **tabindex="0"** : Navigation séquentielle au clavier
- **role="button"** : Indication du rôle interactif
- **aria-label** : Description dynamique du contenu de l'hexagone
- **keydown** : Support de l'activation au clavier (Enter/Space)

Liste de quêtes

Les quêtes sont accessibles au clavier et incluent des labels appropriés :

```
<button type="button" class="quest-card" (click)="openDetails()"
  (keydown.enter)="openDetails()">
  <input
    type="checkbox"
    class="quest-checkbox"
    [checked]="quest.statusId === _questService.statusDoneId"
    (change)="toggleStatus()"
    aria-label="Marquer cette tâche comme terminée" />

  <span class="quest-title">{{ quest.title }}</span>
</button>
```


Gestion du focus dans les menus

Le menu de navigation a une gestion spécialisée du focus pour améliorer l'expérience utilisateur :

```
.menu-item:focus {  
  outline: none;  
}  
  
.menu-item:focus .icon {  
  box-shadow: 0 0 0 2px #667eea !important;  
  border-radius: 50%;  
}  
  
.losange:focus {  
  box-shadow: 0 0 0 2px #667eea !important;  
}
```

4. Technologies d'assistance et lecteurs d'écran

Classes pour lecteurs d'écran

Une classe `.visually-hidden` a été implémentée pour masquer visuellement du contenu tout en le gardant accessible aux lecteurs d'écran :

```
.visually-hidden {  
  border: 0;  
  padding: 0;  
  margin: 0;  
  position: absolute !important;  
  height: 1px;  
  width: 1px;  
  overflow: hidden;  
  clip: rect(1px 1px 1px 1px);  
  clip: rect(1px, 1px, 1px, 1px);  
  clip-path: inset(50%);  
  white-space: nowrap;  
}
```


Utilisation dans les formulaires

Cette classe est utilisée pour les légendes de fieldset et les labels contextuels :

```
<fieldset class="checkbox-fieldset">
  <legend class="visually-hidden">Acceptation des conditions</legend>
  <!-- Contenu du fieldset -->
</fieldset>

<label for="title" [class]="!isEdit && !isNew ? 'visually-hidden' : ''"> Titre :
</label>
```

Messages d'erreur dynamiques

Les messages d'erreur utilisent `role="alert"` et `aria-live="polite"` pour être annoncés automatiquement :

```
@if (hasEmailError) {
  <small class="p-error" id="email-error" role="alert" aria-live="polite"> {{
    emailError }} </small>
}
```

Boutons avec descriptions contextuelles

Les boutons incluent des descriptions appropriées selon leur état :

```
<p-button type="submit" label="Se connecter" [loading]="isLoading" [attr.aria-label]="isLoading ? 'Connexion en cours...' : 'Se connecter'">
</p-button>

<button type="button" class="return pi pi-chevron-left" (click)="onReturn()" aria-label="bouton retour"></button>
```

Cette approche d'accessibilité garantit qu'Hexapanning peut être utilisé efficacement par tous les utilisateurs, y compris ceux qui utilisent des technologies d'assistance, tout en respectant les standards d'accessibilité web modernes.

XI. Conclusion et perspectives

1. Bilan du projet

Hexapanning a permis de concevoir et de mettre en production une application web moderne, robuste et sécurisée, centrée sur l'expérience utilisateur et la gamification de la gestion de tâches, avec un découpage clair entre frontend Angular et backend .NET, la modélisation des entités (quêtes, utilisateurs, hexagones), ainsi que l'automatisation des tests et du déploiement.

Les fonctionnalités principales sont opérationnelles : création et gestion de quêtes, affichage visuel sur carte hexagonale, authentification sécurisée, gestion des mots de passe, et notifications par email. L'architecture modulaire et la conteneurisation facilitent la maintenance et l'évolutivité.

2. Perspectives d'évolution

Les évolutions futures d'Hexapanning s'articulent autour de plusieurs axes fonctionnels et techniques, en lien direct avec les besoins utilisateurs et la structure du code :

- **Sécurité et gestion des comptes**

- Ajout d'un système de refresh token (stocké localement ou en cookies) pour renforcer la sécurité et la gestion de session.
- Envoi d'un email de bienvenue et de confirmation à la création du compte.
- Création d'un dashboard administrateur pour gérer les utilisateurs.

- **Liste de quêtes**

- Ajout d'options de tri pour l'ordre d'affichage des quêtes : par date de création, par priorité, par temps estimé, ou selon un ordre personnalisé.
- Ajout du drag & drop pour réorganiser les quêtes dans un ordre personnalisé.
- Ajout de catégories personnalisables pour faciliter l'organisation.
- Ajout d'un sélecteur de priorité directement depuis la liste.
- Implémentation du glissement tactile sur mobile pour naviguer d'un menu à l'autre (quêtes à accomplir / quêtes accomplies).
- Implémentation d'un bouton pour ouvrir une quête au hasard dans une catégorie ou un niveau de priorité déterminé, afin de permettre à l'utilisateur d'agir sur une des tâches s'il ne sait pas par où commencer.

- **Détails des quêtes**

- Ajout d'une option pour rendre les quêtes répétables et permettre de les placer plusieurs fois sur la carte d'hexagones.
- Ajout d'options dates pour organiser les quêtes dans le temps (date d'exécution prévue ou deadline).
- Regroupement de quêtes en "expédition" avec un objectif final, chaque quête devenant une étape de la progression.
- Archivage manuel des quêtes, ou automatique après avoir été marquées comme terminées depuis un certain temps, afin de désencombrer la liste des quêtes accomplies.

- **Carte d'hexagones**

- Extension de la carte pour ajouter davantage de quêtes, voire extension automatique dès qu'un hexagone proche du bord est rempli.
- Multiplication des cartes pour séparer les quêtes par catégorie.
- Implémentations d'une navigation plus intuitive avec option de zoom et navigation à la souris ou au doigt.
- Ajout de filtres pour masquer les quêtes par état (accomplies / non accomplies) et par priorité, pour permettre à l'utilisateur de se concentrer sur les tâches les plus urgentes sans être distrait par les suivantes, ou simplement de personnaliser son affichage.
- Amélioration du système d'assignation des quêtes aux hexagones en permettant de déplacer une quête en drag & drop.
- Ajout de flèches pour indiquer le sens de progression d'une quête à l'autre.
- Ajout d'une mécanique de personnages se déployant sur la carte comme des soldats conquérant un territoire hexagonal après l'autre, ou d'un personnage seul progressant de façon linéaire jusqu'à un objectif.

- **Personnalisation**

- Ajout d'un avatar pour l'utilisateur.
- Personnalisation des couleurs (thème de l'application, texte des priorités dans les détails de quêtes, liseré des priorités dans la carte d'hexagones).
- Personnalisation des unités déployables sur la carte (une fois faite l'implémentation d'un ou plusieurs personnages évoluant sur la carte).
- Sélection possible de la langue, en stockant tous les textes affichés dans des constantes répertoriées dans différents fichiers (en, fr, etc.) et récupérées via un service de traduction.

- **Déploiement futur**

- Création d'une application mobile en utilisant Ionic, et déploiement sur les stores Android et iOS.
- Système de notifications.
- Persistance des données utilisateurs en les stockant sur l'AsyncStorage de l'appareil afin d'éviter d'avoir à se reconnecter à chaque ouverture de l'app.

L'architecture actuelle, modulaire et évolutive, permet d'intégrer ces améliorations de façon progressive, tout en maintenant la stabilité et la sécurité de la plateforme.

3. Ce que ce projet m'a apporté

Ce projet d'application web complète a été une expérience formatrice, me permettant d'acquérir et de consolider des compétences techniques et méthodologiques essentielles au développement moderne.

Compétences techniques acquises :

- Maîtrise de l'écosystème Angular et de l'architecture frontend moderne
- Développement d'API REST robustes avec .NET Core et Entity Framework
- Intégration et optimisation de bases de données PostgreSQL
- Mise en place de pipelines CI/CD complets avec Docker et automatisation
- Application des principes de sécurité web et de protection des données

Méthodologies et bonnes pratiques :

- Conception d'architecture logicielle modulaire et maintenable
- Implémentation de tests automatisés à tous les niveaux
- Respect des standards d'accessibilité et d'inclusion numérique
- Gestion de projet agile avec documentation technique complète
- Déploiement et monitoring d'applications en production

Ce projet représente une synthèse complète des compétences attendues d'un développeur full-stack, de la conception à la mise en production, en passant par l'optimisation et la maintenance.

4. Remerciements et crédits

Je tiens à remercier mon formateur Pierre-Louis Bastin pour son soutien et sa confiance malgré les difficultés. Egalemeht Mahdi Mcheik, mon camarade de formation et ami, pour ses précieux conseils et son aide pour terminer mon projet. De même, Jade Jolivet, ma camarade de formation, pour ses notes de cours. Enfin, Thomas Pied, mon collègue d'entreprise, qui a pris de son temps pour m'aider à appréhender des concepts complexes afin que je puisse régler des bugs sur une application mobile.

Crédits des icônes utilisées dans l'application : étoiles par GOWI, kornkun2 et meaicon sur flaticon.com Autres icônes par PrimeNG.