TO-DO-LIST

1. Description du Projet

L'objectif principal du projet est de développer une application web permettant aux utilisateurs de gérer efficacement leurs listes de tâches ("To-Do List"). Cette application doit offrir une interface moderne, intuitive et réactive, permettant de créer, organiser, modifier et supprimer des tâches. Le projet sera réalisé en utilisant React pour le front-end et Flask pour le back-end, avec une architecture basée sur une API REST.

2. Analyse Concurrentielle

2.1. Analyse des concurrents directs

- **Todoist** : Plateforme connue pour sa simplicité et ses fonctionnalités avancées comme les rappels et la gestion collaborative des tâches.
    - Points forts : Interface claire, intégrations avec d'autres outils.
    - Points faibles : Certaines fonctionnalités nécessitent un abonnement payant.
- **Microsoft To-Do** : Intégré au sein de l'écosystème Microsoft.
    - Points forts : Synchronisation facile avec les produits Microsoft.
    - Points faibles : Fonctionnalités limitées comparées à d'autres solutions.

2.2. Analyse des concurrents indirects

- **Google Keep** : Simple mais limité à des fonctions basiques de prise de notes.
- **Trello** : Plus axé sur la gestion de projets que sur la gestion de listes personnelles.

2.3. Opportunités du projet

- Offrir une solution 100% gratuite et personnalisable.
- Proposer une interface simple et intuitive avec des fonctionnalités adaptées aux besoins individuels.

3. Analyse des Besoins

3.1. Besoins Fonctionnels

1. **Gestion des tâches :**
    - Créer une tâche avec un titre, une description, une priorité, une date limite et un état (complété/non complété).
    - Modifier et supprimer une tâche.
    - Filtrer et trier les tâches par priorité, date ou état.
2. **Gestion des utilisateurs :**
    - Inscription et connexion via e-mail.
    - Réinitialisation de mot de passe.
3. **Notifications :**
    - Alertes pour les tâches en retard ou imminentes.

3.2. Besoins Non Fonctionnels

TO-DO-LIST

1. **Performance :**
    - Temps de réponse pour chaque requête : inférieur à 1 seconde.
2. **Scalabilité :**
    - Gestion simultanée de plusieurs milliers d'utilisateurs.
3. **Sécurité :**
    - Protection des données utilisateur avec des protocoles SSL.
    - Hachage des mots de passe.
4. **Compatibilité :**
    - Responsive design pour s'adapter aux différents écrans

Conception

4.1. Architecture

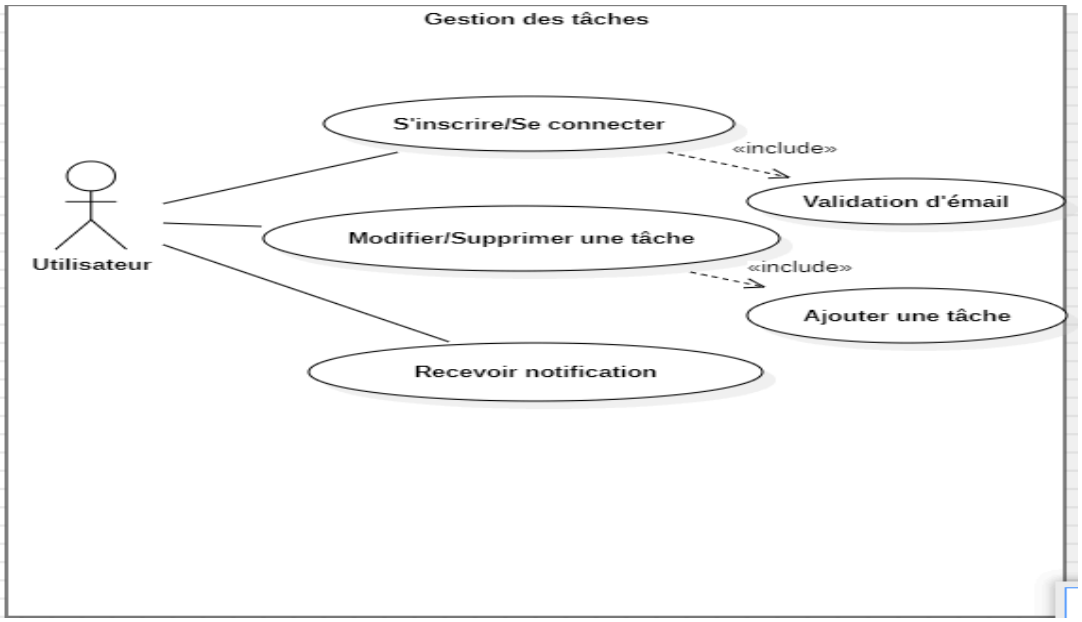
- **Front-end :** React.js pour la création de composants réactifs et dynamiques.
- **Back-end :** Flask pour la gestion des API REST.
- **Base de données :** PostgreSQL pour une gestion fiable des données relationnelles.

4.2. Diagrammes UML

4.2.1. Diagramme de Cas d'Utilisation

Ce diagramme décrit les principales interactions entre l'utilisateur et le système :

- S'inscrire/se connecter.
- Ajouter/modifier/supprimer une tâche.
- Recevoir des notifications.

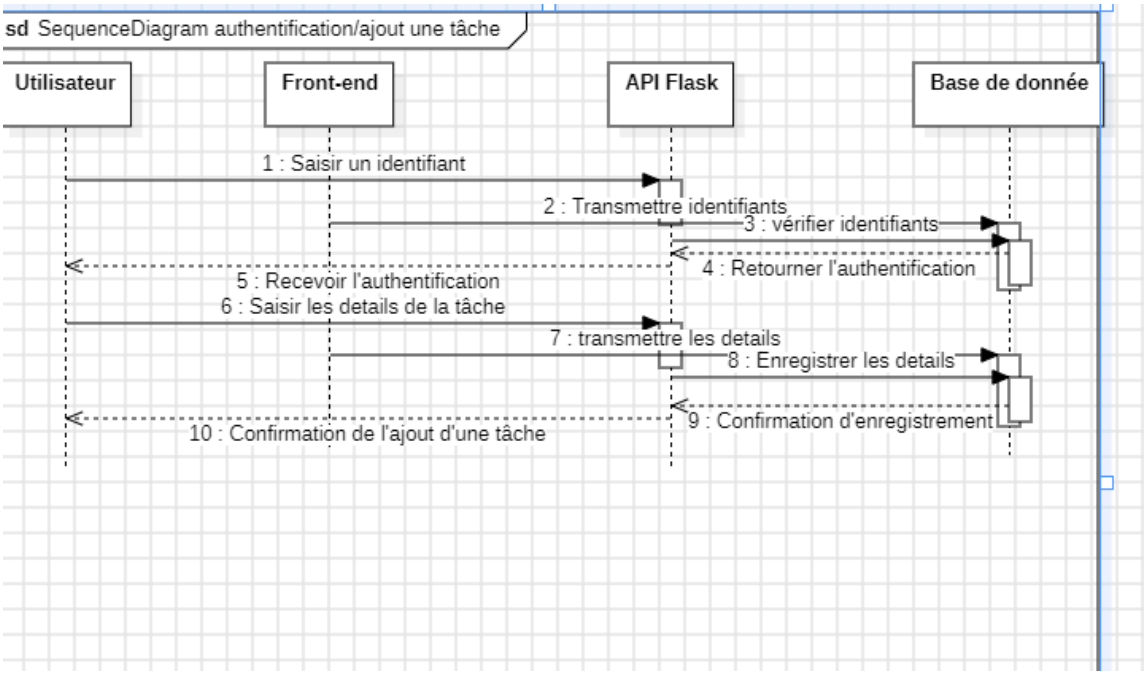


4.2.2. Le système vérifie et retourne un token d'authentification.

- **Ajout d'une tâche :**
    - L'utilisateur saisit les détails de la tâche.

TO-DO-LIST

- **Diagramme de Séquence**
  Exemples de scénarios :
  - **Connexion utilisateur :**
    1. L'utilisateur saisit ses identifiants.
  - Le front-end transmet les données à l'API Flask.
  - La tâche est enregistrée dans la base de données.

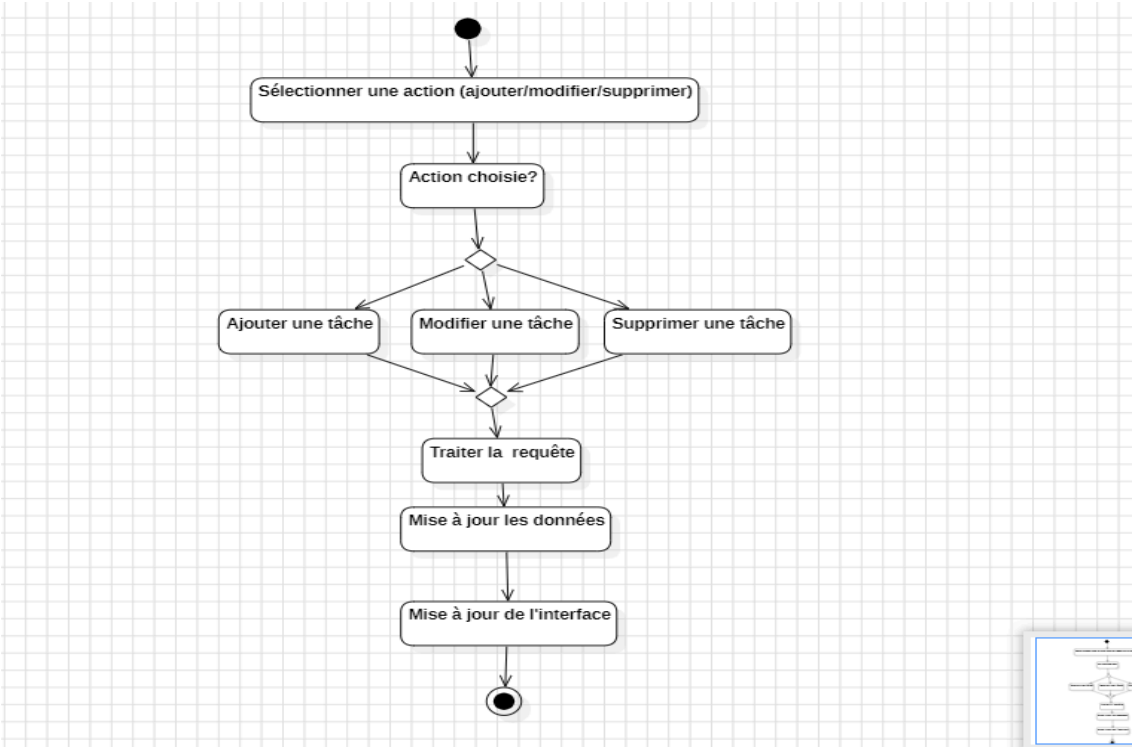


4.2.3. Diagramme d'Activité

Exemple de scénario :

- Gestion des tâches :
  1. L'utilisateur sélectionne une action (ajouter, modifier, supprimer).
  2. Le système traite la requête et met à jour les données.
  3. Mise à jour de l'interface utilisateur.

TO-DO-LIST



5. Exigences Techniques

- **Front-end :** React.js avec Material-UI pour les composants.
- **Back-end :** Flask avec SQLAlchemy pour la gestion des données.
- **API :** RESTful pour une communication claire entre client et serveur.
- **Base de données :** PostgreSQL avec hébergement sur un service cloud.

6. Livrables

- Code source complet.
- Documentation utilisateur et technique.
- Diagrammes UML finalisés.
- Version déployée sur un environnement cloud.