people first

1) Analyse et architecture

Afin de réaliser l'optimisation d'un monolithe vers une nouvelle architecture optimisée, fiable et éco-responsable, nous devons connaître nos choix disponibles, qui sont : le Monolithe lui-même, les Microservices et l'Architecture Hexagonale.

monolithique archi :

Une architecture monolithique est une application conçue et déployée comme une unité unique . Toutes les fonctionnalités, tous les modules et toutes les couches du système sont contenus dans la même base de code, partagent le même environnement d'exécution et, souvent, une seule base de données. L'application entière est construite, testée et déployée ensemble, et tous les composants internes interagissent directement au sein du même processus .

pros :

- Simple à développer, à comprendre et à déployer
- Tous les composants s'exécutent dans le même processus, ce qui permet des appels internes rapides
- Débogage et tests plus faciles grâce à une base de code unifiée
- Fonctionne bien pour les petites équipes et une complexité système modérée

cons :

- **Devient difficile à maintenir à mesure que l'application grandit**
- **Une modification dans un module nécessite le redéploiement de l'application entière**
- **La mise à l'échelle (scaling) doit être effectuée pour l'ensemble du système, même si une seule partie en a besoin**
- **Une défaillance dans un composant peut impacter l'ensemble du système**

coût :

(1-3 ECS/EKS instances - 1 RDS database - 1 S3 bucket - 1 load balancer - CloudWatcher logs)

environ : 300 - 1500 euro par mois / 18000 euro par an

microservices archi :

Une architecture de microservices organise le système en une collection de services petits et indépendants. Chaque microservice est responsable d'une capacité métier spécifique, possède sa propre base de code et gère généralement sa propre base de données. Les services communiquent entre eux via des API ou des systèmes de messagerie, et chaque service peut être développé, déployé et exploité indépendamment des autres.

Pros :

- **Les services indépendants permettent des cycles de déploiement et de développement séparés**
- **Chaque service peut être mis à l'échelle (scaler) individuellement en fonction de sa propre charge**
- **Meilleure isolation : une défaillance dans un service n'entraîne pas l'arrêt de l'ensemble du système**
- **Les équipes peuvent travailler en parallèle sur différents services**

Cons :

- **Complexité opérationnelle accrue (surveillance, mise en réseau, CI/CD par service)**
- **Plus d'infrastructure requise, ce qui entraîne souvent des coûts cloud plus élevés**
- **Nécessite une forte maturité DevOps**

Coût :
(ECS/EKS - RDS database - api gateway - cloudWatch - S3 - SNS/SQS)
environ : 2000 - 6000 euro par mois / 72000 euro par an (sans les salaires de dev ops, altering/on-call rotating, CI/CD infrastructure, ect)

La coûte totale = 80000 / 150000 euro par an !

hexagonal archi :

> **L'Architecture Hexagonale est un style architectural qui structure le code interne de l'application autour de la logique de domaine (Business Logic). Contrairement aux architectures traditionnelles en couches, elle isole strictement les règles métier fondamentales (le Domaine) de l'infrastructure technique.**
>
> **Le principe repose sur la définition d'interfaces claires, appelées Ports, par lesquelles le cœur de l'application communique avec le monde extérieur. Les technologies externes (bases de données, API, interfaces utilisateurs, messagerie) sont connectées via des Adaptateurs. Cette structure garantit que la logique métier ne dépend jamais de la technique, assurant une maintenabilité accrue et une indépendance vis-à-vis des frameworks.**

Pros :

- Elle sépare clairement la logique métier de l'infrastructure technique, rendant le cœur de l'application plus facile à comprendre et à maintenir.
- Elle améliore la testabilité en permettant de tester la logique de domaine sans dépendre de systèmes externes tels que les bases de données ou les API.
- Elle encourage une organisation du code claire alignée sur les concepts métier, améliorant la communication entre les équipes techniques et métier.
- Elle réduit l'effort de maintenance à long terme en imposant des limites strictes entre les couches.

Cons :

- Elle introduit des couches et des abstractions supplémentaires, ce qui augmente la complexité initiale pour les développeurs.
- Elle peut être excessive pour les modules ou services simples qui ont très peu de logique métier.
- Elle nécessite de la discipline de la part de l'équipe pour respecter les limites; sinon, la structure supplémentaire devient une surcharge inutile.
- Elle augmente le temps de développement initial par rapport à une architecture en couches plus simple.
- Elle peut nécessiter une formation ou une adaptation pour les développeurs peu familiers avec le modèle *Ports and Adapters*.

Coût :

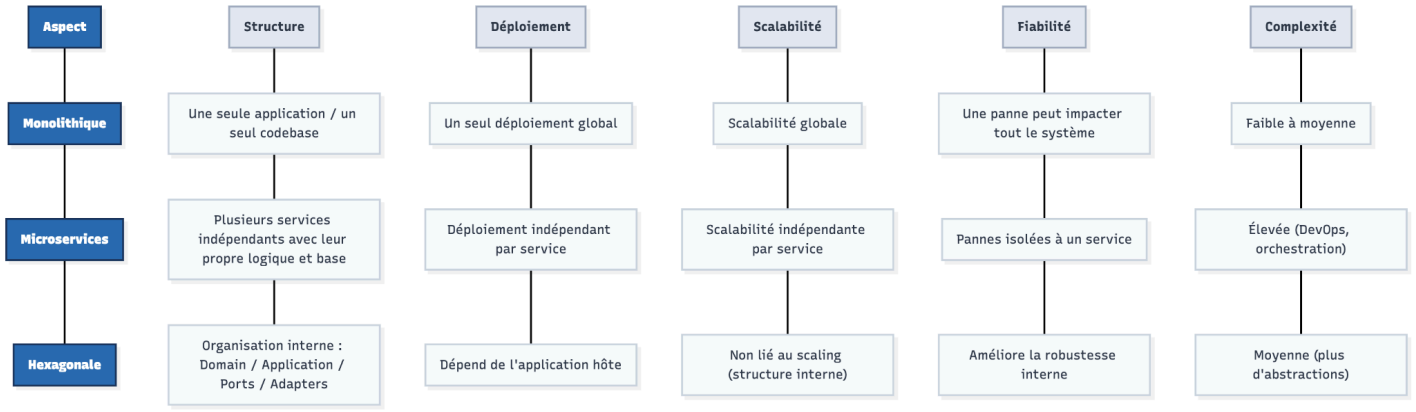
Coût : Surcoût humain initial (formation/complexité) mais économies sur la maintenance long terme

Recommandation pour le projet PeopleFirst :

Pour peopleFirst nous recommandons une architecture monolithique modulaire structurée de façon hexagonale, car elle permet de livrer rapidement une plateforme RH fiable tout en maîtrisant les coûts d'infrastructure et de maintenance
nous avons choisi cette architecture parce que une architecture monolithique classique n'a pas été retenue car elle limiterait fortement la capacité d'évolution du produit et reproduirait a moyen terme , les difficultés déjà rencontrées sur l'application actuelle (complecité de maintenance et évolutions risquées, il en va de même pour l'architecture micro services, car elle entraînerait des coûts d'infrastructure et d'explotation élevés ainsi qu'une complexité opérationnelle disproportionnée par rapport au volume actuel d'utlisateur et aux besions réels de PeopleFirs
C'est pour ca l'architecture monolithique modulaire hexagonale constitue ainsi le meilleur compromis business : elle sécurise les fonctionnalités critiques (paie, congés , documents RH) réduit les risques projet à court terme et préserve une capacité d'evolution progressive si la croissance du produit le justifie

Couts :
Pour assurer la haute disponibilité, l'infrastructure repose sur 2 conteneurs ECS Fargate (240 €/mois) couplés à une base de données RDS PostgreSQL unique de 100 Go gérée en multi-tenant (130 €/mois). Le stockage est délégué à S3 (15 €/mois) pour les fichiers et à Redis (30 €/mois) pour le cache de session. Les traitements asynchrones (paie, emails) sont orchestrés par SQS (10 €/mois). L'ensemble est sécurisé par un Load Balancer et monitoré via CloudWatch (50 €/mois cumulés), portant le budget total estimé à environ 475 € par mois.

Tableau comparatif des 3 architectures :

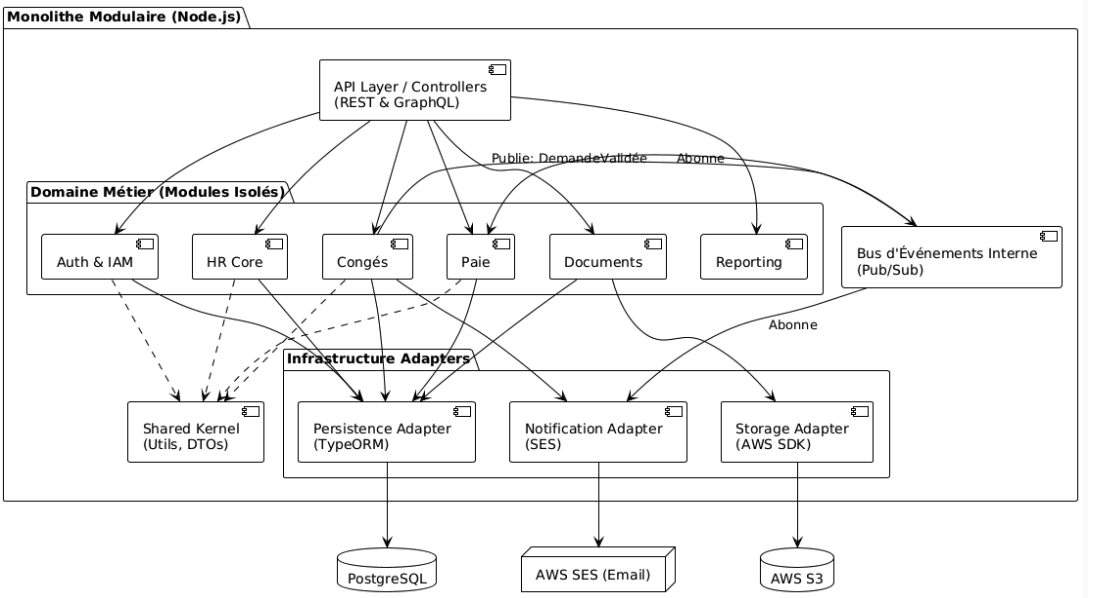


Architecture retenue :

Diagramme de Composants :

Context & synthese :

L'objectif est de remplacer le monolithe PHP/SQL vieillissant par une architecture modernisée , maintenable et évolutive , tout en garantissant une mise en production progressive sur 18 mois nous retenons une architecture : modulaire monolith (single deploy) avec hexagonal appliqué aux modules métiers critiques (HR core , Leave, Payslip, Auth) . Cette approache permet : clarté fonctionnelle , coûts opérationnelle maîtrisés , respect des NFR (SLA , temps de réponse , RGPD) , et possibilité d'extraction vers des microservices ultérieurement



Légende - composant :

API Gateway / load balancer : point d'entrée unique , authentification / throttling , canary .

Modular Monolith (app) : une seule application containerisée , organisée en modules :

- auth & security : gestion utilisateurs , roles , JWT/SSO,audit
- tenant / subscription : gestion société , plans , onboarding
- HR code : employés , contrats , structure orga
- Leave & absence : demandes demandes , soldes , workflow d'approbation
- Payslip / payroll : logique de paie , génération PDF, archivage légal
- Document Service : métadonnées , indexation stockage (s3)
- notification service : envoi email/sms
- Reporting & analytics : modéles les lecture , agrégatoins ;

RDS PostgreSQL : base primaire (multi-AZ) , schéma multi-tenant ou shared schéma selon décision de modéle

S3 : stockage chiffré des documents et fiches de paie

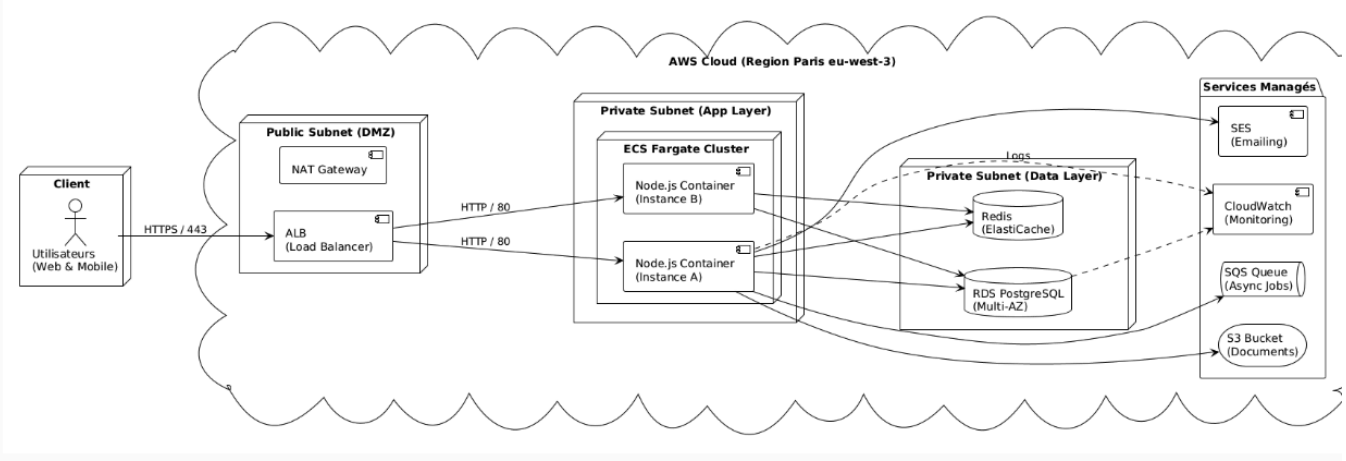
Redis : cache , sessions , accélération des lectures critiques (soldes congés…)

Queue (SQS/message bus) : découplage asynchrone (événements métier)

workers : génération des fichier , tâches batch , traitements lourds hors flux synchrones

CI/CD & logs : pipeline , monitoring traces (opentelematry) , alerting

2.3. Diagramme de Déploiement (Infrastructure AWS)



2.4. Justification des choix structurants

Voici la synthèse des décisions techniques, alignée avec les contraintes du cahier des charges

1. Persistance des Données (Stockage)

Pour répondre à la complexité des données RH et aux exigences légales, nous optons pour une persistance polyglotte :

Technologie	Usage	Justification du choix
Relationnel (SQL) AWS RDS PostgreSQL	Données Cœur (Employés, Contrats, Paie)	Nécessité absolue de transactions **ACID** pour garantir l'intégrité des données financières et contractuelles. Le mode **Multi-AZ** assure le SLA de 99,5%[2].
Object Storage AWS S3	Documents (PDF, Justificatifs)	Stockage moins coûteux et plus performant pour les fichiers binaires. Permet l'application de politiques de cycle de vie (ex: archivage légal 5 ans) pour l'éco-conception.
Cache In-Memory Redis (ElastiCache)	Sessions & Soldes Congés	Réduit la charge sur la base de données principale et garantit un temps de réponse **< 200ms** pour les données fréquemment consultées (soldes de congés).

2. Sécurité

L'architecture intègre la sécurité à tous les niveaux pour la conformité RGPD:

- **Chiffrement de bout en bout :**
    - **En transit :** HTTPS (TLS 1.3) forcé par le Load Balancer.
    - **Au repos :** Disques RDS et Buckets S3 chiffrés via AWS KMS (clés gérées).
- **Isolation Réseau :** L'application et les données résident dans des sous-réseaux privés (Private Subnets). Seul le Load Balancer est exposé publiquement.
- **Contrôle d'accès (RBAC) :** Gestion fine des droits (Admin, RH, Manager, Employé) implémentée au niveau du module d'Authentification et vérifiée à chaque appel API.

3. Communication

- **Synchrone (Client ↔ Serveur) : API REST**
    - Standard universel facilitant l'intégration avec le frontend (Web/Mobile) et les futurs partenaires (SaaS Paie, SIRH)
- **Asynchrone (Traitement de fond) : Message Queuing (SQS)**
    - Utilisé pour les tâches lourdes comme la **génération des fiches de paie PDF** ou l'envoi d'emails. Cela permet de "découpler" l'action utilisateur du traitement, évitant de bloquer l'interface et lissant les pics de charge en fin de mois

Conception Détaillée (Modélisation UML) :

diagramme de cas d'utilisation :

Le système s'articule autour de quatre acteurs hiérarchiques. L'**Employé** accède aux fonctions de base (congés, documents). Le **Manager** hérite des droits de l'employé mais possède en plus la capacité de valider les demandes de son équipe. Le **Gestionnaire RH** supervise l'ensemble des données administratives et la paie, tandis que l'**Administrateur** gère la configuration technique et les accès

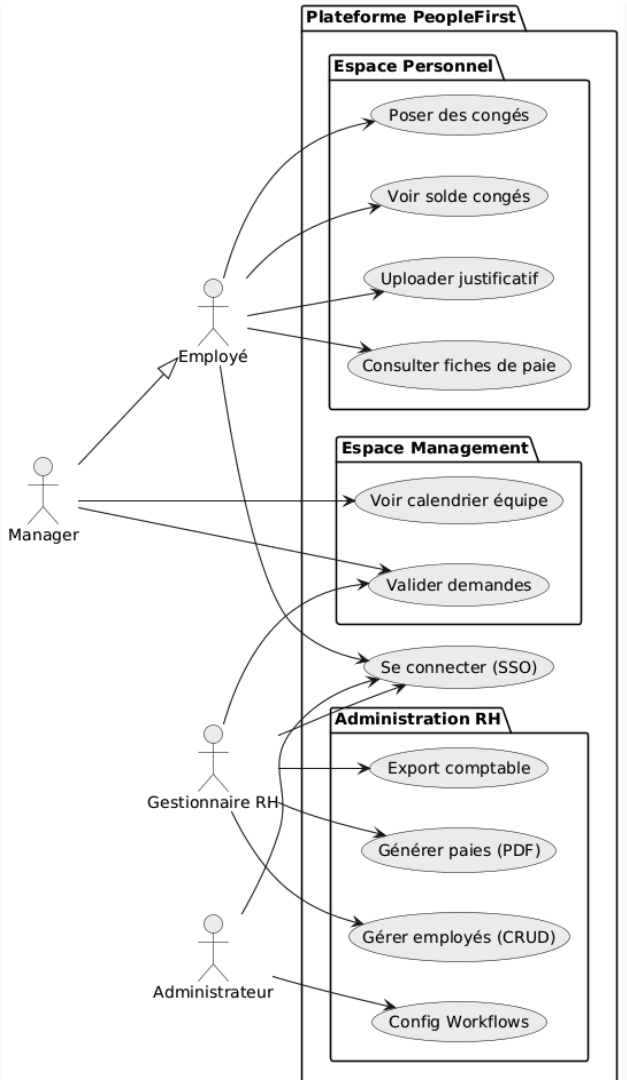


diagramme de classes :

de base (congés, documents). Le **Manager** hérite Le modèle de données est centré sur l'entité `Employee`, qui hérite de `User` pour la gestion des accès. Une attention particulière est portée à l'entité `LeaveRequest` (Demande de congé) qui porte les règles métier de validation, et aux entités `Document` et `Payslip` quistockent uniquement les métadonnées, le contenu binaire étant délégué au stockage S3 (attribut `s3Key`) pour des raisons de

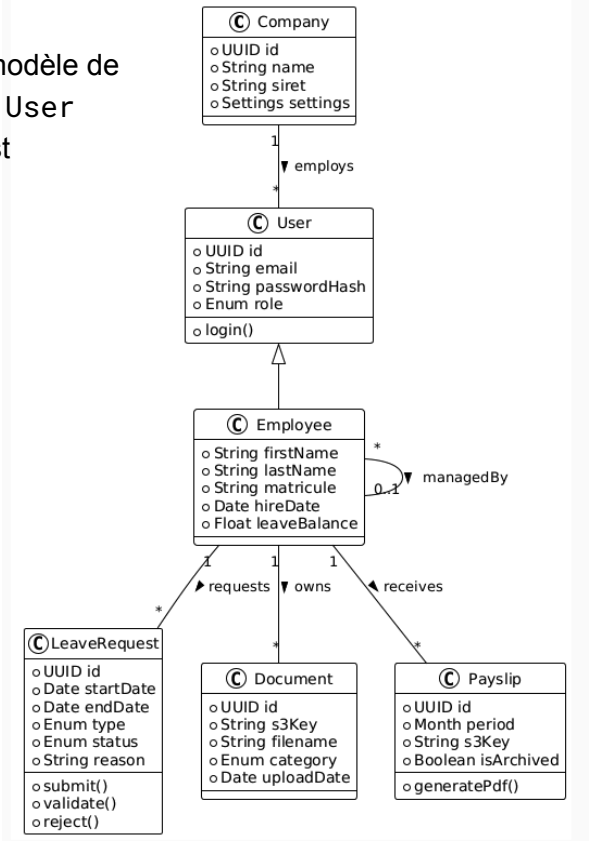
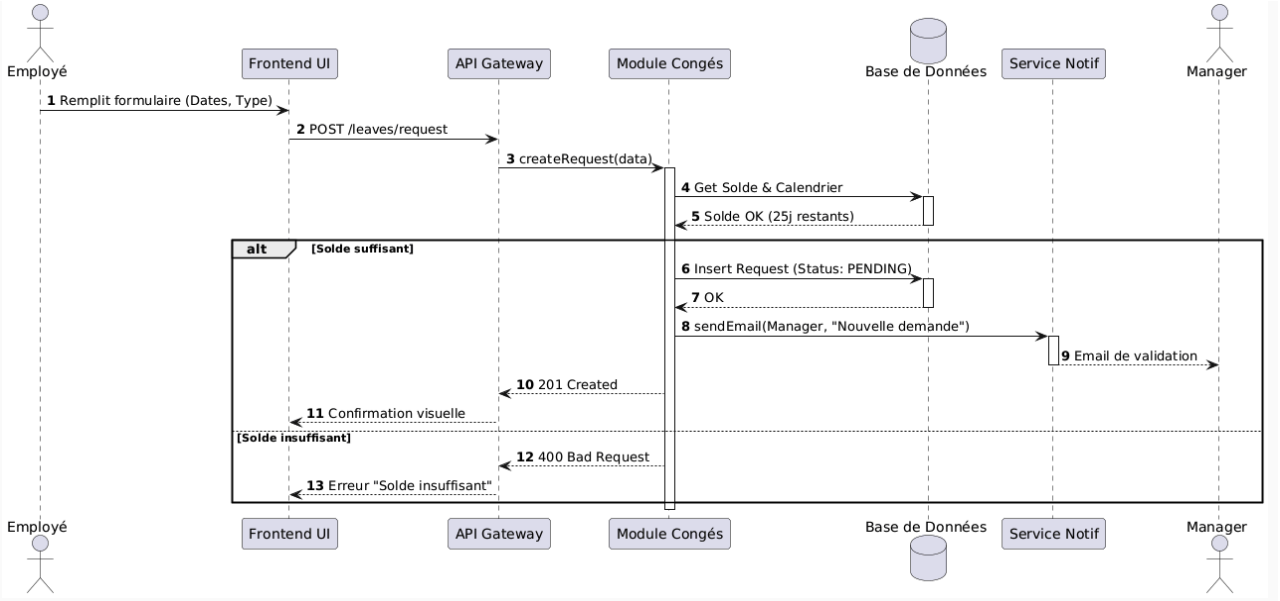


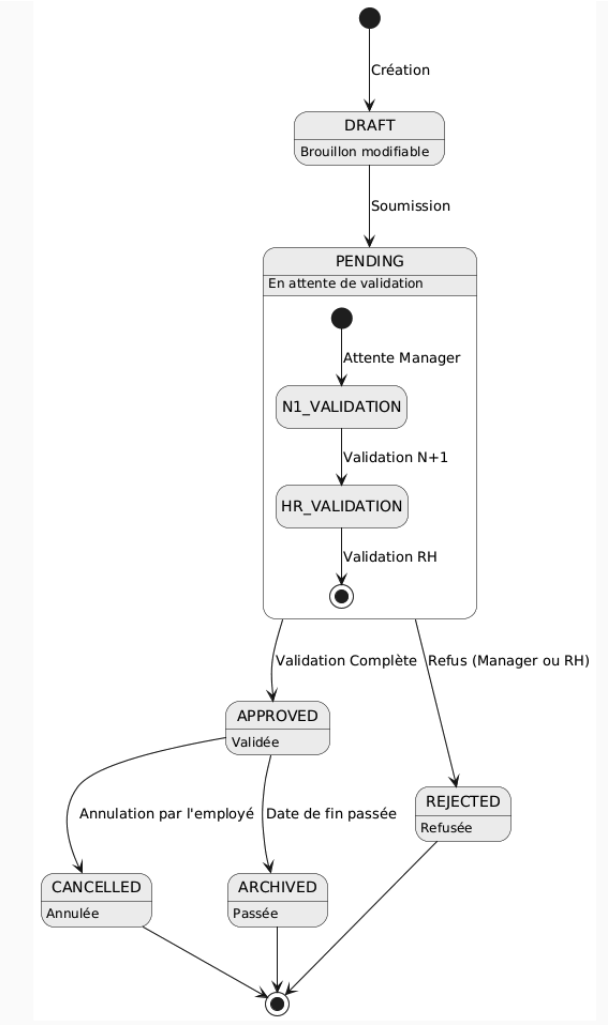
diagramme de Séquence : Workflow "Demande de Congé"



Ce diagramme détaille le flux nominal d'une demande de congé. La robustesse du système est assurée par une vérification synchrone du solde avant la création de la demande. Si le solde est suffisant, la demande passe en statut `PENDING` et une notification asynchrone est envoyée au Manager pour déclencher l'étape de validation

Diagramme d'États : Cycle de vie d'une demande

Ce diagramme modélise le cycle de vie complet d'une demande. Nous avons intégré un état `DRAFT` pour permettre à l'employé de préparer sa demande sans l'envoyer. L'état `PENDING` peut inclure une double validation (Manager puis RH) selon la configuration de l'entreprise. L'état `APPROVED` déclenche la décrémentation finale du solde



Principes de conception :

Pour garantir la maintenabilité et l'évolutivité de la plateforme PeopleFirst, nous avons appliqué des règles d'ingénierie logicielle strictes. Cette section détaille l'application concrète des principes SOLID et des Design Patterns dans notre architecture.

application de principes SOLID :

Nous avons sélectionné deux principes fondamentaux qui structurent notre approche "Monolithe Modulaire Hexagonal".

A. Dependency Inversion Principle (DIP) — L'Inversion de Dépendance

Définition : Les modules de haut niveau (le Domaine métier) ne doivent pas dépendre des modules de bas niveau (Infrastructure). Les deux doivent dépendre d'abstractions (Interfaces).

Application dans le projet : C'est le cœur de notre **Architecture Hexagonale**.

- **Sans DIP :** Le module *Congés* importerait directement la librairie **`node-mailer`** ou `aws-sdk` pour envoyer un email de validation. Si on change de fournisseur d'email, on doit modifier le code métier.
- **Avec DIP (Notre choix) :**
    1. Le Domaine *Congés* définit une interface (Port) : **`INotificationService`**.
    2. L'Infrastructure implémente cette interface via un adaptateur : **`SesNotificationAdapter`**.
    3. Au moment de l'exécution, on "injecte" l'adaptateur dans le service métier.

  Où le voir dans les diagrammes ? Sur le **Diagramme de Composants (Figure 1)** : Regardez les flèches entre les modules métier (Bleu) et les adaptateurs (Vert). Le sens des dépendances (flèches en pointillés) montre que l'infrastructure dépend des interfaces du domaine, et non l'inverse. Le Domaine reste "pur".

B. Single Responsibility Principle (SRP) — Responsabilité Unique

Définition : Une classe, un module ou une fonction ne doit avoir qu'une seule raison de changer.

Application dans le projet : Nous appliquons ce principe pour découper nos classes au sein de chaque module.

- **Entité (`LeaveRequest`) :** Ne gère que la logique métier et l'état (ex: transition de *PENDING* à *APPROVED*). Elle ne contient pas de SQL.
- **Repository (`LeaveRepository`) :** Ne gère que l'accès aux données (SQL `INSERT`, `SELECT`). Il ne contient aucune règle métier.
- **Controller (`LeaveController`) :** Ne gère que le protocole HTTP (Parsing JSON, Codes 200/400).

  Où le voir dans les diagrammes ? Sur le **Diagramme de Classes**, nous avons séparé les concepts. Par exemple, la classe `Payslip` contient la méthode `generatePdf()`, mais le stockage du fichier est délégué à une responsabilité distincte (via l'attribut `s3Key`), évitant de mélanger la logique de génération et la logique de stockage.

Design Patterns identifiés :

Pour résoudre des problèmes récurrents identifiés dans le cahier des charges, nous utilisons deux patrons de conception majeurs.

1. Strategy Pattern (Patron Stratégie)

Contexte : La gestion des absences implique différents types de règles selon le type de congé:

- **Congés Payés (CP) :** Décrémente un compteur annuel, nécessite validation N+1.
- **Maladie :** Ne décrémente pas les CP, nécessite un justificatif obligatoire.
- **RTT :** Règles d'acquisition spécifiques.

Justification du choix : Sans ce pattern, le code de validation serait rempli de conditions complexes (`if type == 'CP' ... else if type == 'Maladie' ...`). Le Pattern **Strategy** nous permet de définir une interface commune `LeaveCalculationStrategy` et de créer une classe concrète pour chaque type (`PaidLeaveStrategy`, `SickLeaveStrategy`).

- **Avantage :** Si la loi change ou si PeopleFirst veut ajouter un nouveau type d'absence (ex: "Congé Parental"), nous créons simplement une nouvelle classe "Stratégie" sans toucher au code existant (Respect de l'Open/Closed Principle).

2. Observer Pattern (Patron Observateur)

Contexte : Le workflow de demande de congé déclenche plusieurs actions collatérales qui ne sont pas liées directement :

1. Validation de la demande.
2. Envoi d'une notification email au manager.
3. Mise à jour du solde en temps réel.
4. Éventuelle notification au service Paie.

Justification du choix : Nous utilisons l'**Observer Pattern** via notre **Bus d'Événements Interne** (mentionné dans l'architecture). Lorsqu'une demande est validée, le module *Congés* publie un événement : `LeaveRequestValidated`.

- Le module *Notification* écoute cet événement -> Envoie l'email.
- Le module *Paie* écoute cet événement -> Prépare la variable de paie.
- **Avantage : Découplage total**. Le module *Congés* ne "connaît" pas le module *Notification*. On peut ajouter de nouvelles réactions (ex: logger l'action dans un audit) sans modifier le code de validation des congés.

  Note d'implémentation : Dans notre architecture Node.js, cela sera implémenté via un `EventEmitter` en interne (synchrone) ou via SQS (asynchrone) pour les tâches lourdes.

Stratégie d'Éco-conception (Green IT)

5.1. Les Enjeux pour PeopleFirst

Le numérique représente aujourd'hui 4% des émissions mondiales de gaz à effet de serre. Pour une plateforme SaaS comme PeopleFirst, l'impact environnemental se situe principalement à deux niveaux : la **consommation énergétique des centres de données** (hébergement AWS) et l'**obsolescence des terminaux utilisateurs** (si l'application est trop lourde, elle oblige les clients à changer de PC/Smartphone). Notre stratégie vise la sobriété numérique : fournir le service attendu avec le minimum de ressources informatiques, prolongeant ainsi la durée de vie du matériel et réduisant la facture énergétique.

5.2. Trois mesures concrètes intégrées à l'architecture

Nous avons transformé les contraintes techniques en opportunités écologiques directement dans le design de l'infrastructure.

A. Cycle de vie de la donnée (Stockage chaud vs froid)

Problème : Stocker 5 ans d'historique de fiches de paie 2 sur des disques SSD rapides (RDS/EBS) est un gaspillage énergétique et financier, car un bulletin de 2021 est très rarement consulté.

Mesure architecturale :

Nous implémentons une politique de Lifecycle Management sur AWS S3 (Object Storage).

1. **J0 à J+90 (Hot) :** Le document est stocké en standard (accès milliseconde).
2. **J+90 à 5 ans (Cold) :** Le document migre automatiquement vers la classe **S3 Glacier Deep Archive**.
3. Après 5 ans : Suppression automatique (Conformité RGPD "Droit à l'oubli").
   Gain : Réduction estimée de 40% de l'empreinte de stockage et division par 10 du coût.

B. Adaptation des ressources à la demande (Scaling dynamique)

Problème : Les serveurs traditionnels tournent souvent à vide la nuit et le week-end, consommant de l'électricité inutilement ("Zombie servers").

Mesure architecturale :

L'utilisation de AWS ECS Fargate (Conteneurs) couplé à l'Auto-Scaling.

- En journée (9h-18h) : Le cluster s'adapte au trafic (ex: 5 conteneurs).
- La nuit et le week-end : Le cluster réduit automatiquement la voilure au strict minimum (2 conteneurs pour la redondance).
  Gain : On évite de "chauffer le cloud" inutilement 60% du temps.

C. Minimisation des échanges de données (API Design)

Problème : Envoyer des données inutiles au navigateur (ex: tout le profil employé pour une simple liste) surcharge le réseau et la batterie des mobiles.

Mesure architecturale :

Application du pattern DTO (Data Transfer Object) strict sur l'API REST.

- L'API expose des endpoints spécifiques (ex: GetEmployeeSummary) qui ne renvoient que les champs affichés à l'écran (Nom, Prénom, Photo).
- Les données lourdes (Historique complet) ne sont chargées que sur demande explicite (Lazy Loading).
  Gain : Réduction de la bande passante réseau et de la charge CPU côté client (navigateur).

5.3. Application du référentiel RGESN

Nous nous appuyons sur le **Référentiel Général d'Écoconception de Services Numériques** (version 2024) pour guider nos développements. Voici 3 critères prioritaires appliqués au projet :

Critère RGESN	**Description**	**Application chez PeopleFirst**
8.1 - Stratégie de conservation	"Le service numérique définit-il une politique de conservation / suppression des données ?"	**Oui.** Configuration de règles d'expiration ("Time-To-Live") sur les logs techniques (30 jours) et les brouillons de demandes de congés abandonnés (90 jours) pour éviter l'obésité numérique.
1.3 - Terminaux supportés	"Le service est-il compatible avec des équipements anciens ?"	**Oui.** Le Frontend sera développé pour être compatible avec des navigateurs datant de 5 ans (ES5/ES6 target). Cela évite aux PME clientes de devoir renouveler leur parc informatique pour utiliser PeopleFirst.
7.3 - Compression des fichiers	"Les fichiers téléchargés sont-ils compressés ?"	**Oui.** Le module "Paie" intègre une librairie de compression PDF lors de la génération des bulletins avant leur envoi vers le stockage S3.

5.4. L'Arbitrage assumé (Trade-off)

L'éco-conception implique parfois de renoncer à des fonctionnalités de confort au profit de la sobriété.

La décision : Refus du "Temps Réel Permanent" (WebSocket) pour les notifications.

L'alternative écartée :

Nous aurions pu maintenir une connexion ouverte (WebSocket) avec chaque utilisateur pour afficher les notifications (ex: "Votre congé est validé") instantanément, à la milliseconde près.

L'impact écologique :

Maintenir 10 000 connexions ouvertes simultanément 3 consomme énormément de mémoire sur les serveurs et empêche la mise en veille des smartphones des utilisateurs (batterie drainée par le maintien du socket).

Notre choix (L'arbitrage) :

Nous avons choisi un modèle de Polling optimisé ou de notification par Email/Push Mobile standard. L'interface vérifie les nouvelles notifications uniquement lors d'une action de l'utilisateur ou toutes les 5 minutes.

- **Perte :** L'utilisateur voit sa validation avec quelques minutes de délai.
- **Gain :** Réduction drastique de la consommation serveur et préservation de la batterie des terminaux mobiles des employés.