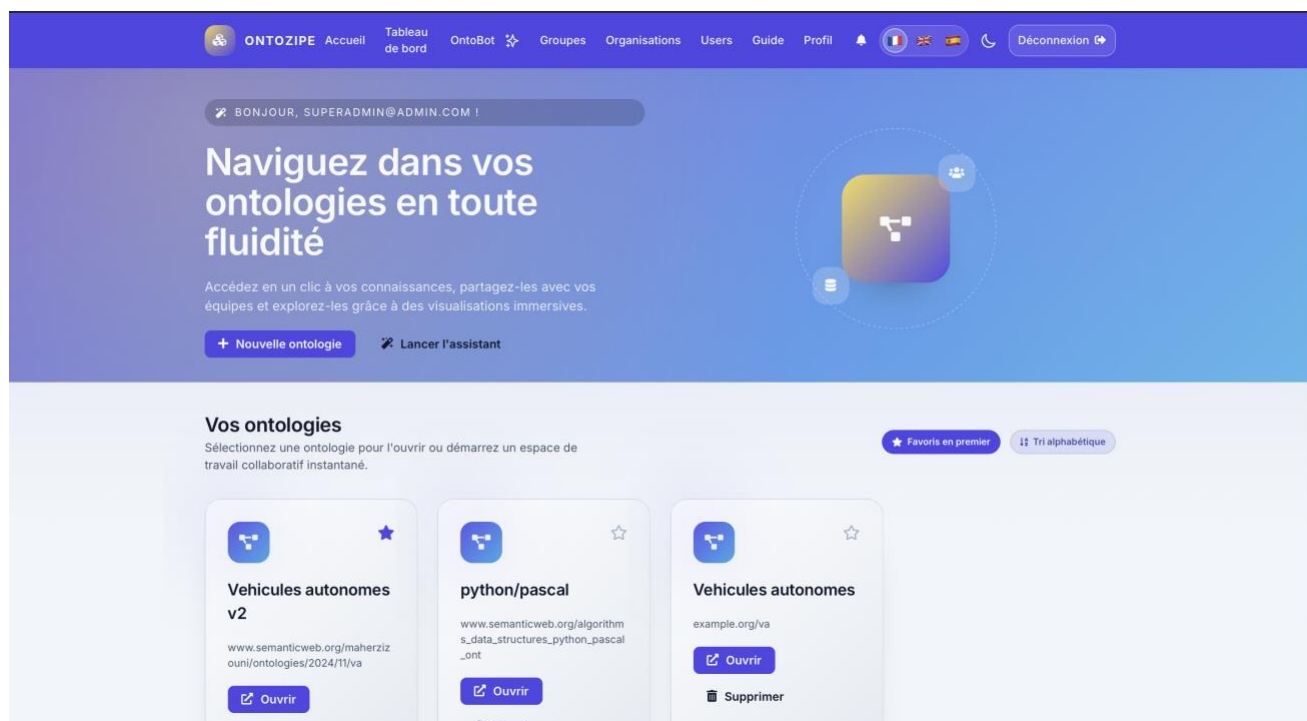


TX Ontozipe

Rapport de projet

Maher Zizouni · Saad Treyaoui

20/01/2026



Résumé exécutif

Ce rapport présente la transformation majeure de la TX sur Ontozipe : nous sommes passés d'une base POC à une plateforme robuste, professionnelle et maintenable.

Le travail réalisé ne se limite pas à un simple ajout de fonctionnalités. Il comprend une refonte complète de l'expérience utilisateur, une architecture backend plus claire, une meilleure gestion des performances, et un socle ontologique structuré qui permet de capitaliser la connaissance de manière collaborative.

Le résultat est une application web moderne, adaptée à un usage réel en entreprise : multi-organisation, sécurisée, documentée, et capable d'évoluer rapidement.

1. Contexte et point de départ

Le projet Ontozipe a été repris à partir d'une première version orientée POC (preuve de concept). Cette version validait la faisabilité technique (authentification, gestion d'ontologies, interaction), mais restait limitée sur l'ergonomie, la scalabilité et la structuration du code.

Le défi a été de transformer ce POC en une application réellement exploitable par des équipes, avec des besoins concrets : multi-utilisateurs, gestion de rôles, documentation intégrée, et performances adaptées à de gros volumes de données.

Cette transformation a impliqué des décisions techniques fortes : refonte du front, modularisation du backend, mise en place d'une ontologie centrale, et une stratégie de documentation vivante plutôt qu'un rapport statique.

2. Refonte UI/UX et standardisation Frontend

L'interface a été totalement réorganisée pour gagner en lisibilité, cohérence et professionnalisme. Nous avons quitté une approche utilitaire (Tailwind) pour structurer une vraie identité visuelle en CSS, avec des tokens, des composants et des pages cohérentes.

Tailwind est conservé uniquement pour le préflight (reset), tandis que l'ensemble des styles et composants est désormais géré par un CSS maison structuré.

Le système de design est centralisé via des variables CSS. Cela rend le thème clair/sombre fiable et cohérent sur tout le produit.

```
:root {
  --color-primary: #4f46e5;
  --surface-card: #ffffff;
  --border-color: #e2e8f0;
}

.dark {
  --color-text: #f1f5f9;
  --surface-card: #111c2f;
  --border-color: #334155;
}
```

L'internationalisation a été poussée jusqu'à la granularité des composants. Toutes les chaînes de texte sont stockées dans un dictionnaire multilingue et exposées via des clés :

```
const definitions = {
  "common.pagination.previous": { fr: "Précédent", en: "Previous", es:
  "Anterior" },
  "navbar.notifications": { fr: "Notifications", en: "Notifications", es:
  "Notificaciones" },
};
```

Le responsive a été un point critique : l'objectif était d'offrir une expérience aussi fluide sur mobile que sur desktop. Cela se traduit par des layouts réactifs, des drawers et des ajustements fins dans les media queries.

```
@media (max-width: 960px) {
  .guide-layout { grid-template-columns: 1fr; }
  .guide-sidebar {
    position: fixed;
    width: min(90vw, 340px);
    transform: translateX(-100%);
  }
}
```

Cette refonte UI/UX a changé la perception du produit : l'application n'est plus un prototype, mais un outil professionnel, agréable à utiliser et crédible auprès d'utilisateurs non techniques.

3. Optimisations de performance

La performance a été travaillée sur deux niveaux : le réseau (réduction des appels) et l'affichage (réduction des re-rendus).

Côté frontend, React Query est utilisé comme couche de cache. Les requêtes sont structurées par clés, avec des fenêtres de fraîcheur adaptées, ce qui évite de recharger la même donnée inutilement.

```
return useQuery({
  queryKey: ["dashboard-summary", input?.section, input?.payload,
    language],
  enabled: Boolean(token) && Boolean(input),
  staleTime: 30 * 1000,
  queryFn: async () => { /* ... */ },
});
```

La logique d'invalidation est fine et ciblée pour éviter les recharges globales :

```
const invalidate = () => {
  client.invalidateQueries({ queryKey: ["notifications"] });
  client.invalidateQueries({ queryKey: ["notifications", "unreadCount"] });
  client.invalidateQueries({ queryKey: ["notifications", "preview"] });
};
```

Côté backend, un cache en mémoire est implémenté pour éviter des requêtes SPARQL répétitives (rôles, propriétaires, groupes). Cela sécurise et accélère l'accès aux données tout en réduisant la charge sur Fuseki.

```
private static readonly CACHE_TTL_MS = Number(
  process.env.ONTOLLOGY_CACHE_TTL_MS ?? 10_000
);

private getCachedValue<T>(cache: Map<string, CacheEntry<T>>, key: string) {
  const entry = cache.get(key);
  if (!entry) return null;
  if (entry.expiresAt < Date.now()) { cache.delete(key); return null; }
  return entry.value;
}
```

Pour limiter les re-rendus inutiles, les calculs lourds sont mémorisés côté UI (ex. filtrage du guide, préparation du menu, etc.).

```
const accessibleContent = useMemo(
  () => guideContent.filter((entry) => canAccess(entry.access, roles)),
  [roles]
);
```

4. Nettoyage et modularisation du code

L'un des problèmes majeurs du POC initial était la centralisation excessive : un gros contrôleur backend et un gros composant React. Cette structure rendait l'évolution risquée et la maintenance coûteuse.

Nous avons découpé l'application en domaines clairs (auth, ontologie, organisations, notifications, LLM, dashboard). Chaque domaine dispose de son contrôleur, ses DTOs, et ses services. Le front suit la même logique avec des pages et des hooks spécialisés.

```
@ApiTags("Organizations")
@ApiBearerAuth()
@UseGuards(JwtAuthGuard)
@Controller("organizations")
export class OrganizationsController {
  constructor(private readonly organizationsService: OrganizationsService)
  {}
}
```

Le même principe s'applique côté frontend avec des hooks dédiés, notamment pour la recherche et la pagination qui sont devenues une base de toutes les pages professionnelles.

```
export function useSearchPagination<T>(items: T[], { filter, pageSize = 8
}) {
  const [searchTerm, setSearchTerm] = useState("");
  const filteredItems = useMemo(() => {
    if (!searchTerm) return items;
    return items.filter((item) => filter(item, searchTerm.toLowerCase()));
  }, [filter, items, searchTerm]);
  /* pagination ... */
}
```

5. Conception et architecture (diagrammes)

Les diagrammes suivants synthétisent la structure technique et les choix d'architecture. Chaque diagramme répond à un objectif précis : rendre lisible la complexité, clarifier les responsabilités et faciliter la maintenance.

5.1 Contexte global

Ce diagramme montre l'écosystème complet : utilisateur, frontend React, backend NestJS, stockage RDF (Fuseki), stockage PDF et LLM. Il positionne clairement les flux et les dépendances externes.

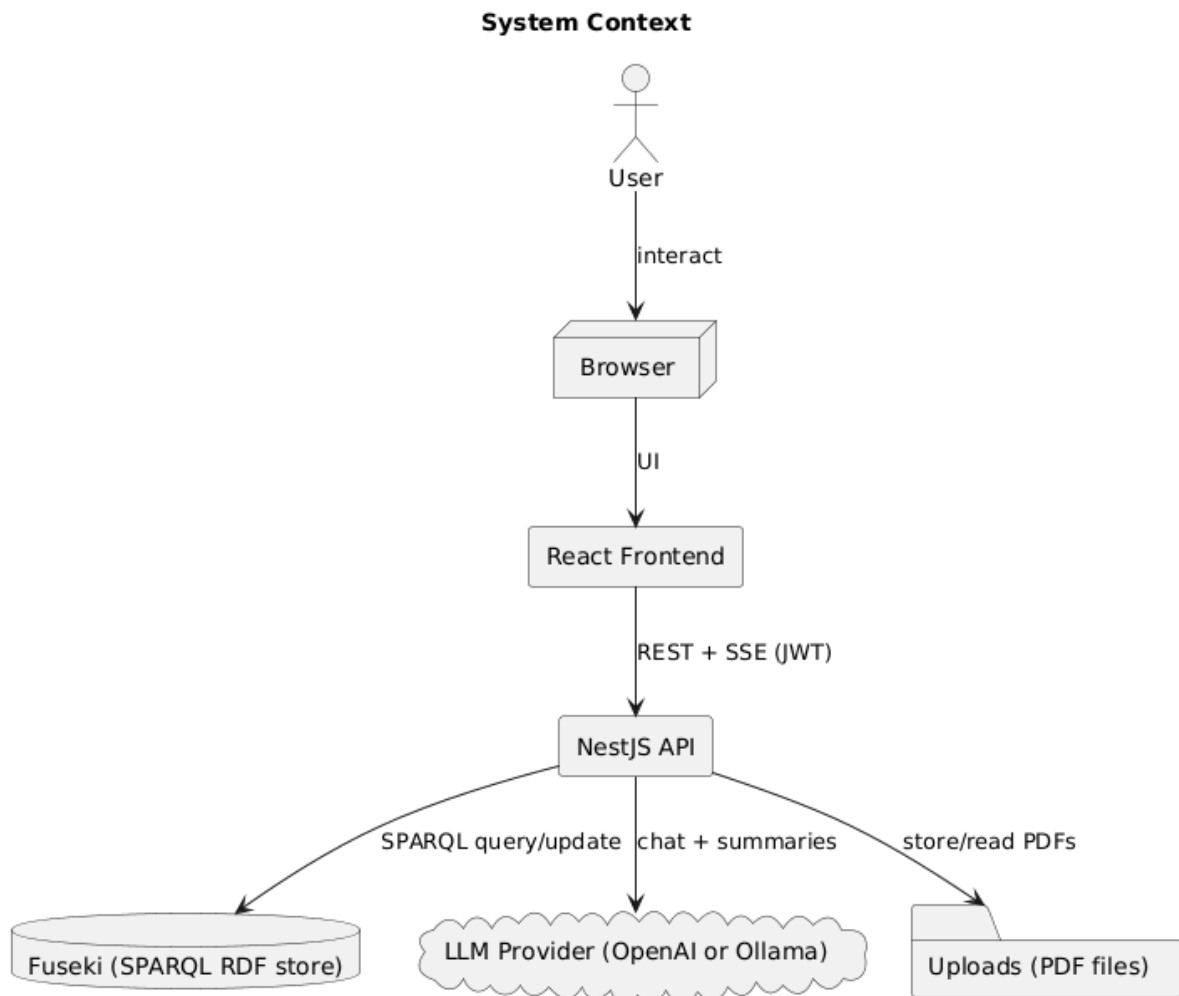


Figure 1 — System Context

5.2 Backend

Le découpage en modules backend rend chaque domaine autonome et permet d'industrialiser les évolutions sans effet domino.

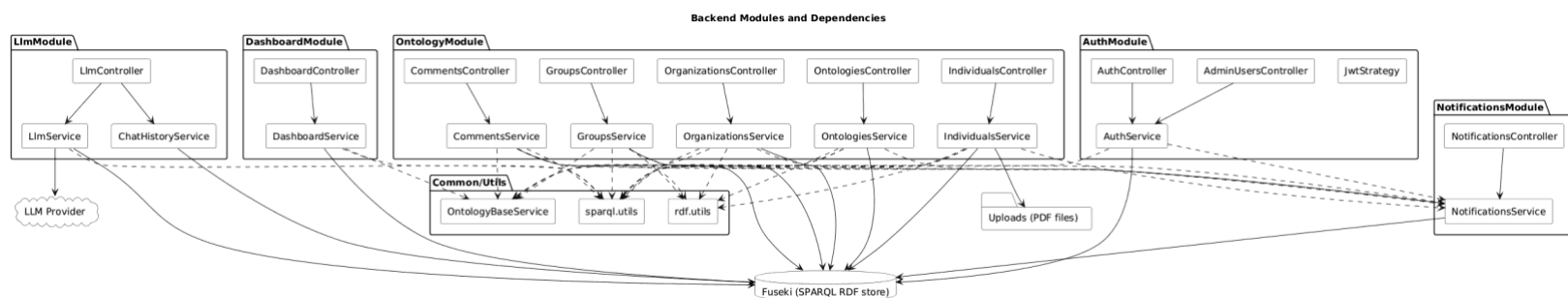


Figure 2 — Backend Modules

La surface API clarifie l'ensemble des routes exposées et facilite l'intégration côté frontend ou par des tiers.

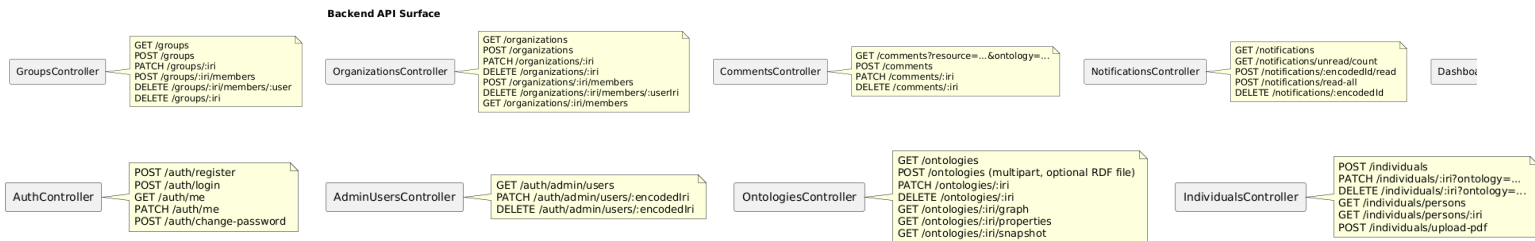


Figure 3 — Backend API Surface

Le flux d'authentification illustre comment les rôles, les comptes et les notifications sont enchaînés lors des connexions et inscriptions.

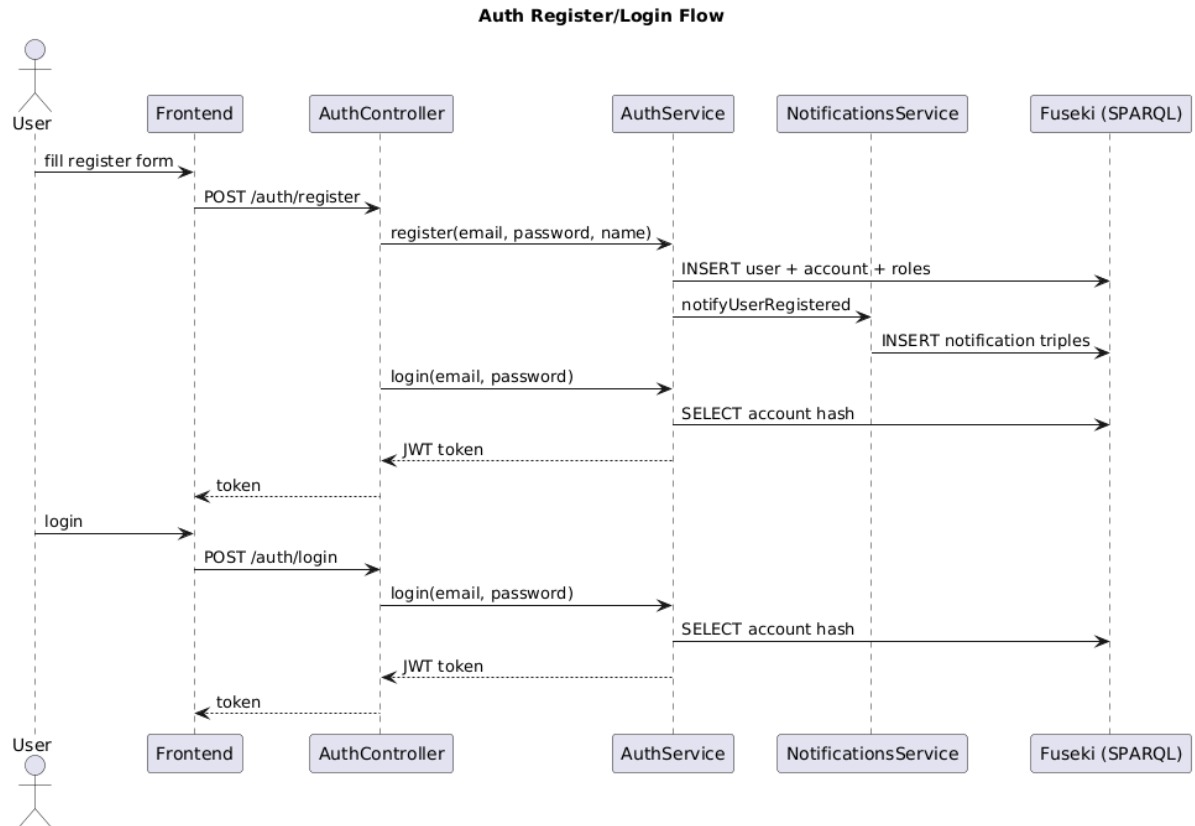


Figure 4 — Auth Flow

Le flux ontologie + PDF explique la création d'individus, l'ajout de documents et la propagation d'événements de notification.

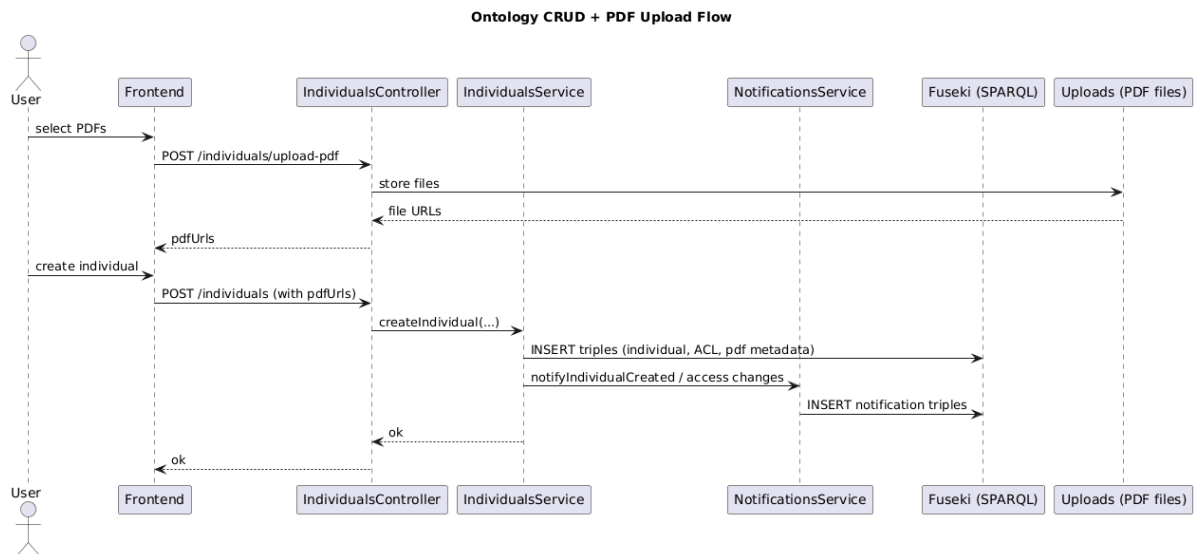


Figure 5 — Ontology Flow

Le flux LLM met en évidence la logique SSE et l'accès aux outils SPARQL, illustrant la connexion entre IA et ontologies.

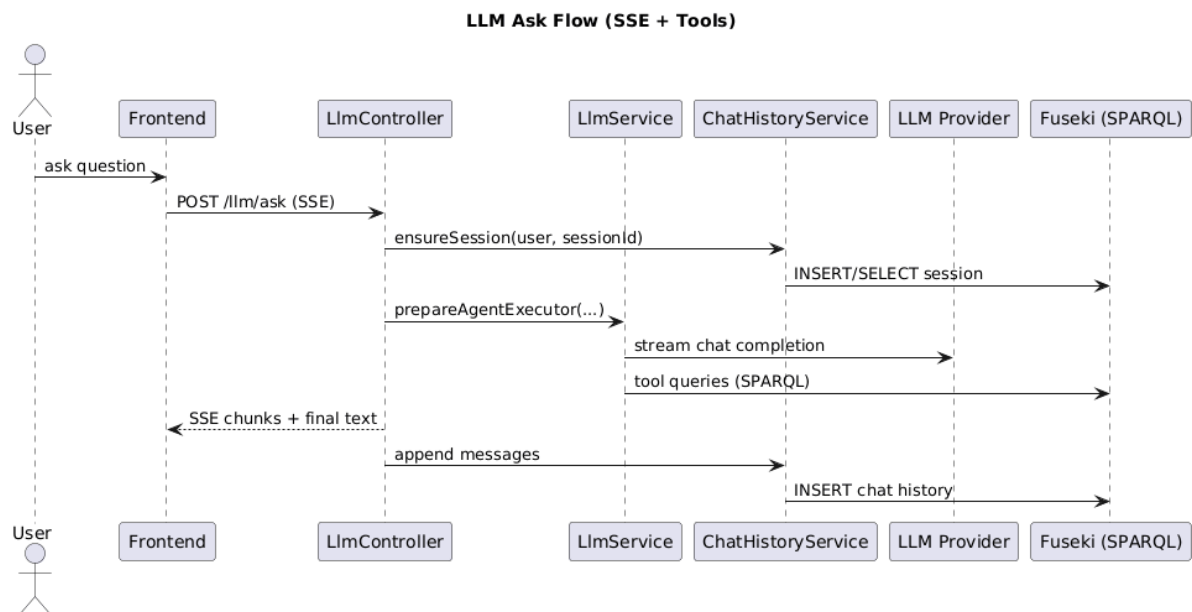


Figure 6 — LLM Flow

Le flux notifications montre comment les événements sont stockés en RDF puis exposés au frontend avec filtres et statuts.

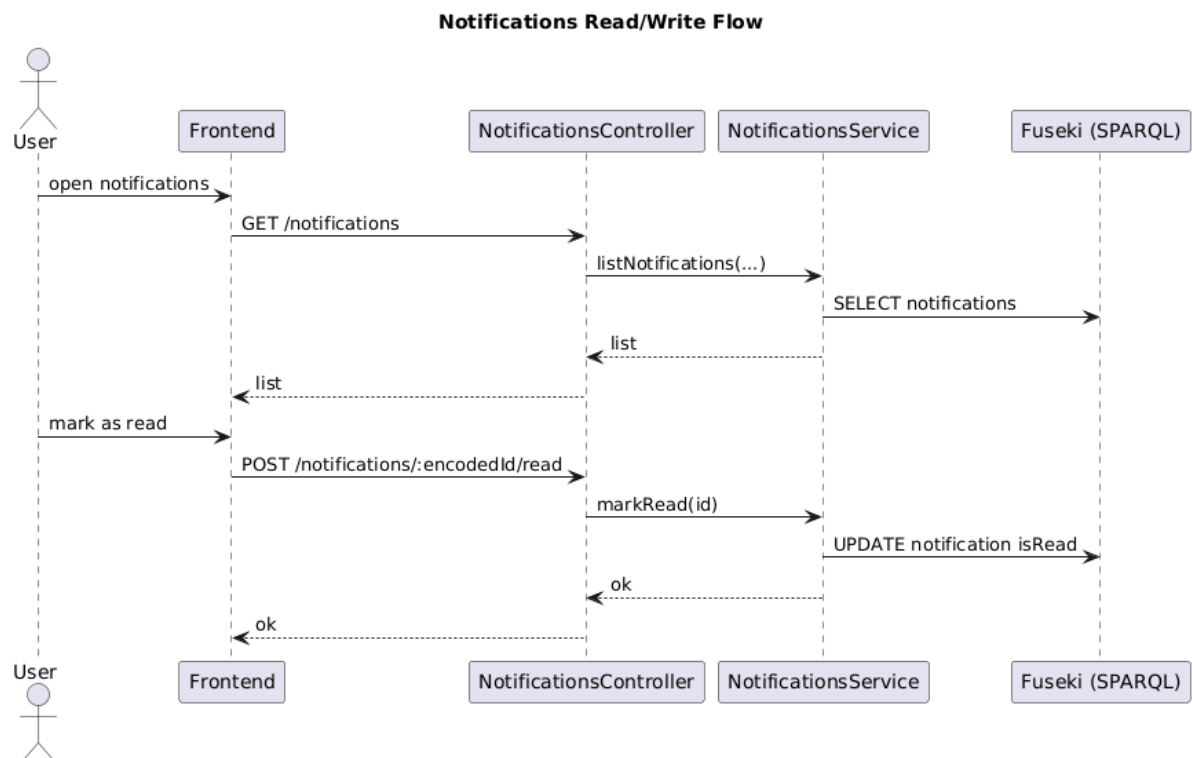


Figure 7 — Notifications Flow

5.3 Frontend

L'architecture applicative front montre les providers (langue, auth) et le routing sécurisé, garantissant une base claire pour la navigation.

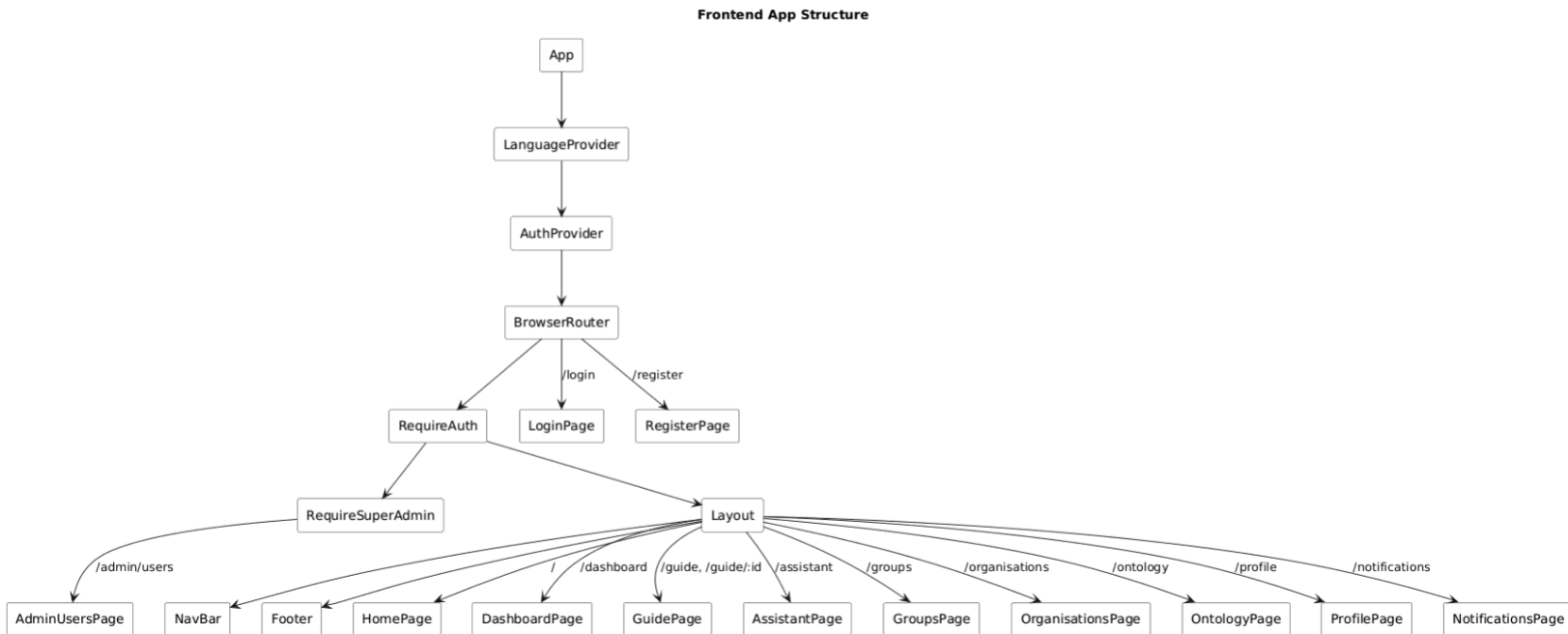


Figure 8 — App Structure

Le flux de données met en lumière l'utilisation systématique de hooks et de React Query pour un accès cohérent aux APIs.

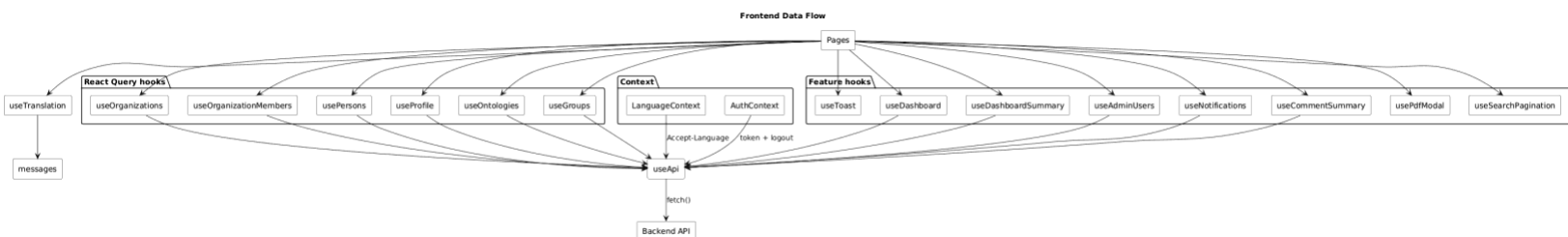


Figure 9 — Data Flow

La page Ontologie est un point critique de l'app : son découpage en composants démontre la volonté de rendre une interface complexe lisible.

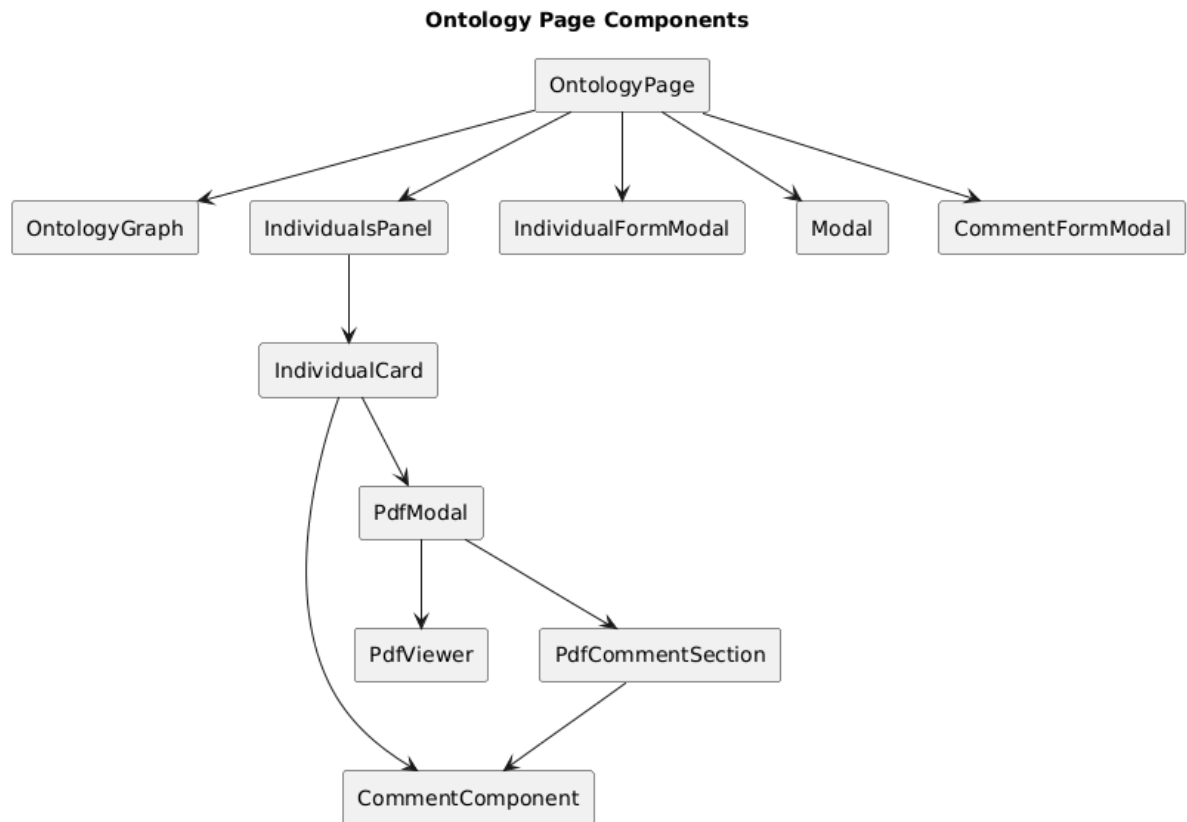
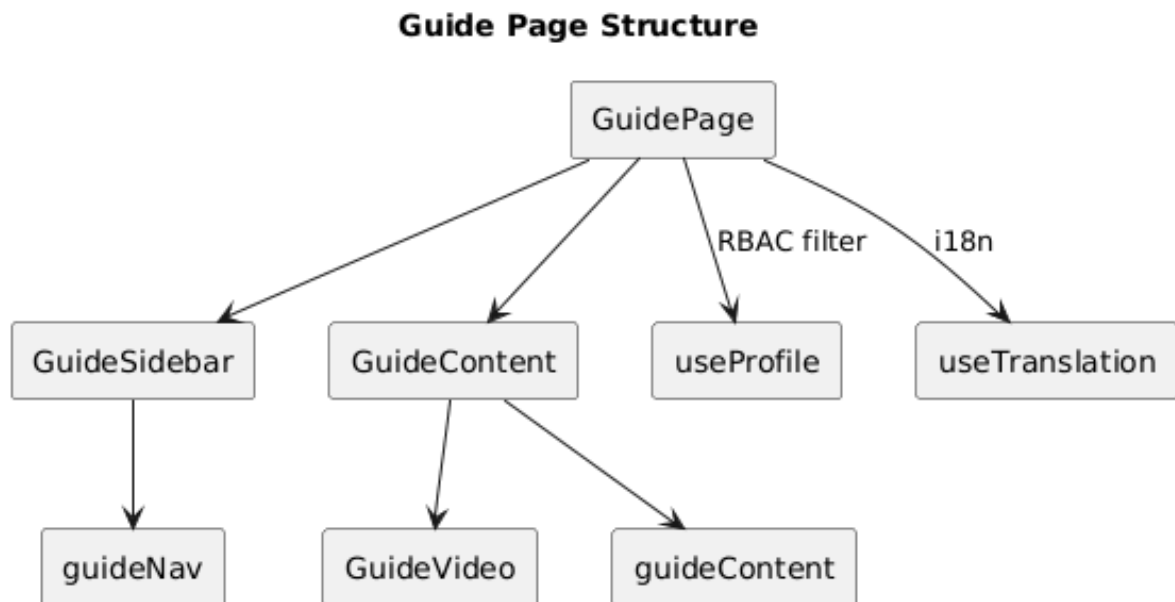


Figure 10 — Ontology Page

La page Guide illustre le modèle data-driven et l'intégration de RBAC, i18n et vidéo dans un seul flux.



6. Nouvelles fonctionnalités clés

Notifications : cette fonctionnalité apporte un vrai confort utilisateur et une traçabilité métier. Chaque action importante génère un événement consultable, filtrable et marquable comme lu.

```
@ApiTags("Notifications")
@UseGuards(JwtAuthGuard)
@Controller("notifications")
export class NotificationsController {
  @Get()
  @ApiQuery({ name: "status", enum: ["all", "unread"] })
  async list(...) { /* ... */ }
}
```

Le front consomme ces endpoints via des hooks dédiés, avec pagination et filtres :

```
const qs = new URLSearchParams();
qs.set("limit", String(limit));
qs.set("offset", String(offset));
if (params.status) qs.set("status", params.status);
const res = await api(`/notifications?${qs.toString()}`);
```

Groupes et organisations : nous avons sécurisé la gestion multi-organisation grâce à des vérifications côté backend (rôles, ownership, appartenance) et une structure ontologique explicite. Cela rend l'outil puissant pour la capitalisation de connaissance en environnement collaboratif.

```
protected async isOrganizationOwner(userIri: string, organizationIri:
string) {
  const result = await this.runAsk(`
    PREFIX core: <${this.CORE}>
    ASK { GRAPH <${this.PROJECTS_GRAPH}> { <${organizationIri}>
core:ownedBy <${userIri}> } }
  `);
  return result;
}
```

Professionnalisation des pages : filtres, recherche et pagination sont systématisés pour supporter des volumes importants. Cela change l'échelle d'utilisation de l'application.

```
const totalPages = useMemo(
  () => Math.max(1, Math.ceil(filteredItems.length / pageSize)),
  [filteredItems, pageSize]
);
```

Hooks personnalisés : la logique métier est centralisée dans des hooks réutilisables (useNotifications, useDashboardSummary, useSearchPagination, etc.). Cette approche facilite l'évolution sans duplication de code.

7. Guide utilisateur intégré

Le guide utilisateur est intégré directement dans l'application. Ce choix évite les PDF obsolètes et garantit une documentation vivante, maintenable et contextuelle.

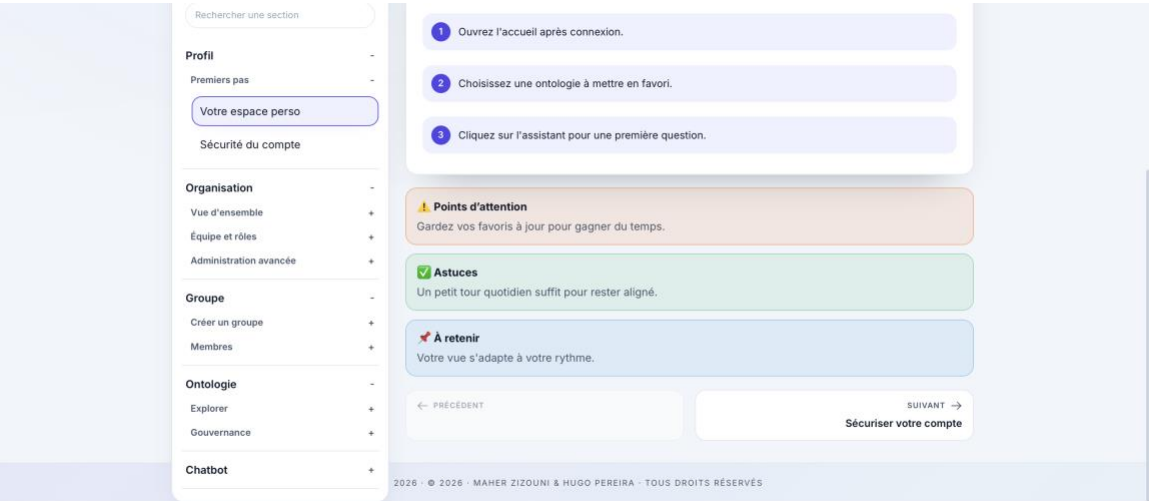
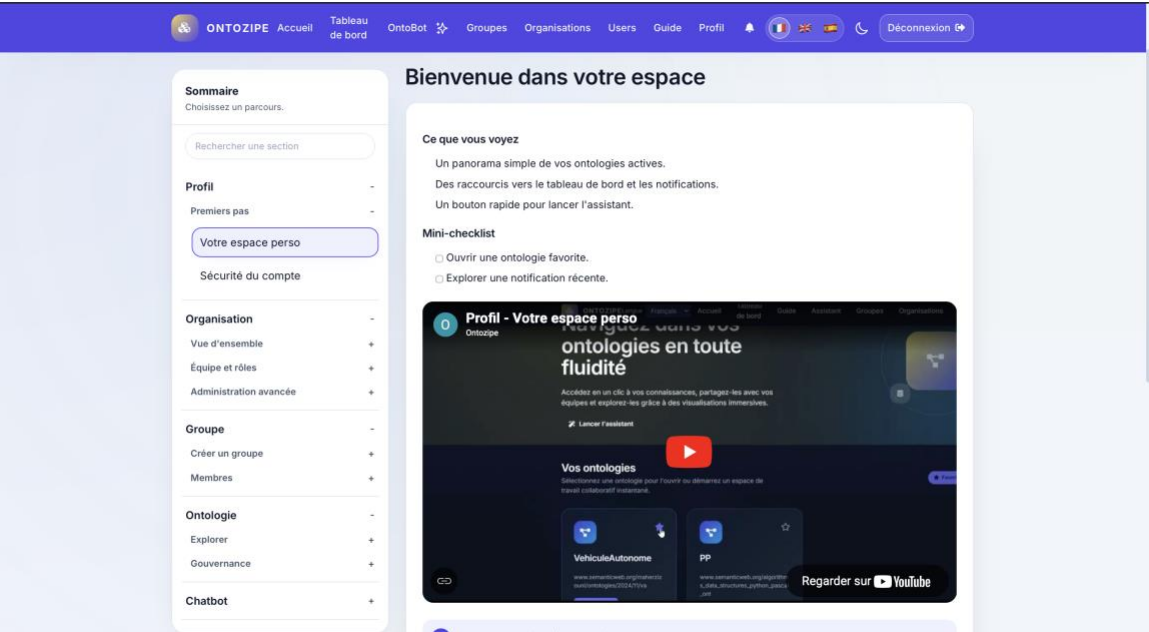
Ajouter un nouveau tutoriel se résume à une entrée structurée dans la configuration :

```
{
  id: "organization.members",
  category: "organization",
  titleKey: "guide.content.organization.members.title",
  markdownKey: "guide.content.organization.members.markdown",
  youtubeVideoId: "cb4Lci5oAq0",
  access: "admin",
  callouts: [ { type: "warning", textKey: "... " } ],
  steps: [ { textKey: "... " } ]
}
```

Le rendu est automatique : Markdown stylé, vidéo intégrée, liste d'étapes et callouts visuels. Cela permet de produire des guides riches sans effort de développement.

Le RBAC est intégré au guide : certaines sections ne sont visibles que par les admins ou superadmins.

```
const canAccess = (access, roles) => {
  const isSuperAdmin = roles.some((role) =>
    role.endsWith("SuperAdminRole"));
  const isAdmin = isSuperAdmin || roles.some((role) =>
    role.endsWith("AdminRole"));
  if (!access || access === "all") return true;
  if (access === "superadmin") return isSuperAdmin;
  if (access === "admin") return isAdmin;
  return false;
};
```



8. Documentation API (Swagger / OpenAPI)

La documentation Swagger est intégrée directement au backend, exposée via /docs et /docs-json. Elle est activable ou non par variable d'environnement pour la production.

```
const swaggerEnabled = String(process.env.SWAGGER_ENABLED ??
"true").toLowerCase() === "true";
if (swaggerEnabled) {
  const config = new DocumentBuilder()
    .setTitle("Ontozipe API")
    .addBearerAuth({ type: "http", scheme: "bearer", bearerFormat: "JWT" },
"bearer")
    .build();
  SwaggerModule.setup("docs", app, document, { jsonDocumentUrl: "/docs-
json" });
}
```

Chaque route est documentée via des décorateurs : DTOs, exemples, statut HTTP, et contraintes. Cela rend la doc interactive et bien plus utile qu'un PDF statique.

```
@ApiOperation({ summary: "Creer une organisation" })
@ApiCreatedResponse({ schema: { type: "string", example:
"http://example.org/org/acme" } })
@Post()
createOrganization(@Body() dto: CreateOrganizationDto) { /* ... */ }
```

Ontozipe API 1.0.0 OAS 3.0







Documentation OpenAPI pour l'API Ontozipe.

Authorize 

Auth

POST	/auth/register	Inscription classique	
POST	/auth/login	Connexion	
GET	/auth/me	Profil courant	 
PATCH	/auth/me	Mise à jour du profil	 
POST	/auth/change-password	Changement de mot de passe	 

Users

GET	/auth/admin/users	Lister les utilisateurs (admin)	 
PATCH	/auth/admin/users/{encodedId}	Mettre à jour un utilisateur (admin)	 
DELETE	/auth/admin/users/{encodedId}	Supprimer un utilisateur (admin)	 

9. Caractère innovant

L'innovation majeure repose sur l'intégration d'une ontologie centrale (core.ttl) comme pivot du produit. Cette ontologie n'est pas une simple documentation : elle structure les données, les droits d'accès et les interactions IA.

Le modèle formalise des entités clés (User, Organization, Group, OntologyProject, Resource, Notification) et leurs relations. Cela garantit une cohérence sémantique forte et une capacité unique à capitaliser la connaissance.

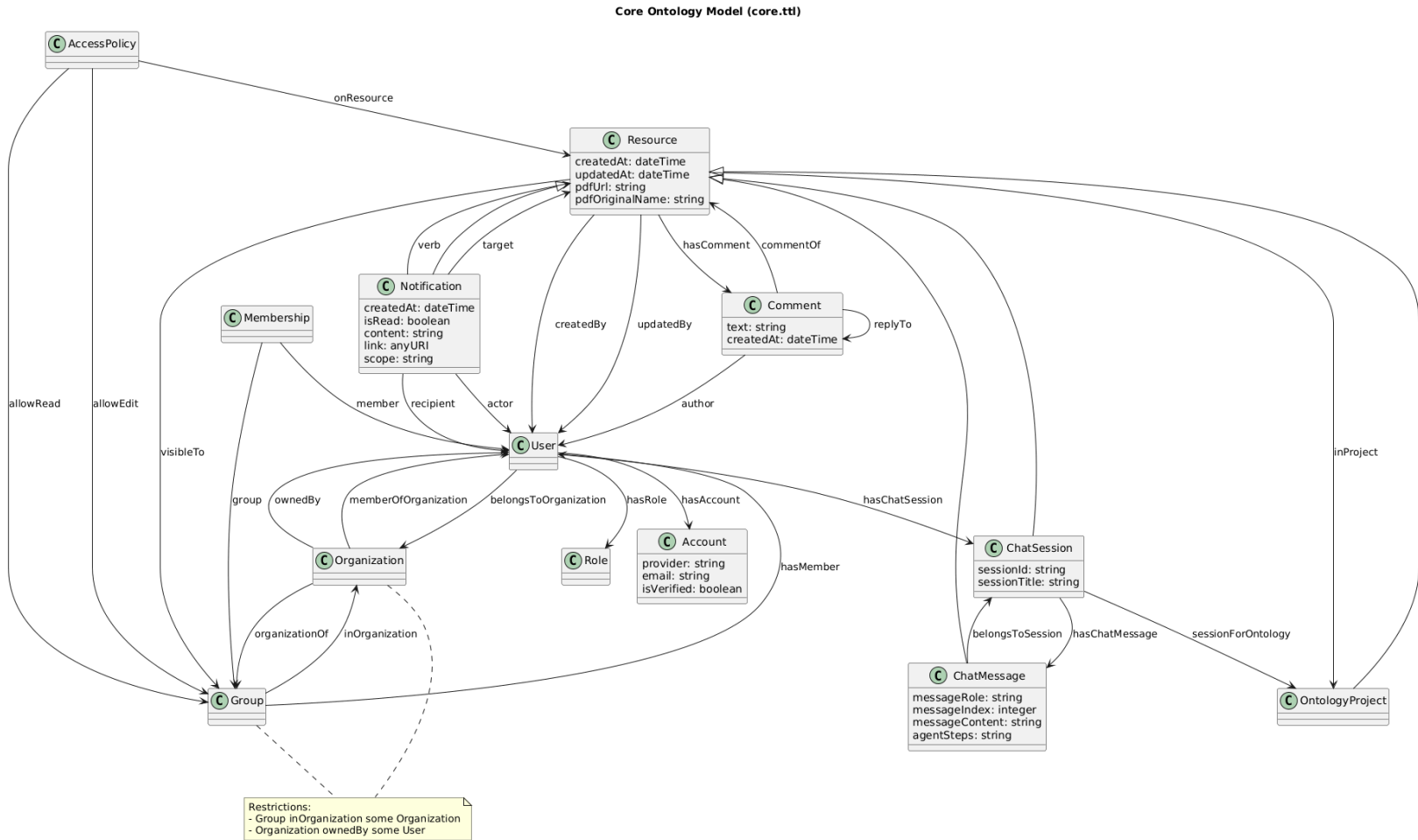


Figure 12 — Structure core.ttl

Extrait du cœur ontologique :

```
core:User                a owl:Class ; rdfs:subClassOf foaf:Person .
core:Organization        a owl:Class ; rdfs:subClassOf foaf:Organization .
core:Group               a owl:Class ; rdfs:subClassOf foaf:Group .
core:OntologyProject      a owl:Class ; rdfs:subClassOf void:Dataset,
core:Resource .
core:Notification        a owl:Class ; rdfs:subClassOf core:Resource .

core:hasRole              a owl:ObjectProperty ; rdfs:domain core:User ;
rdfs:range core:Role .
core:belongsToOrganization a owl:ObjectProperty ; rdfs:domain core:User ;
rdfs:range core:Organization .
core:inOrganization       a owl:ObjectProperty ; rdfs:domain core:Group ;
rdfs:range core:Organization .
```

La définition des rôles et des verbes de notification est explicite, ce qui permet d'ajouter de nouveaux cas d'usage sans refactor majeur :



Figure 13 — Instances (rôles et verbes)

Choix techniques clés :

- Les rôles sont modélisés comme des instances de core:Role, ce qui permet un RBAC extensible sans modifier le code.
- Les relations Organization/Group/User sont explicites, ce qui sécurise la collaboration et limite les fuites de données.
- Les notifications sont aussi modélisées dans l'ontologie, assurant traçabilité et auditabilité.
- Les ressources sont liées aux projets ontologiques et aux documents, offrant une vraie capitalisation de connaissance.

Cette approche est innovante car elle combine une base RDF, une gouvernance multi-organisation, une interface moderne, et une IA connectée à la connaissance métier. Peu d'outils proposent ce niveau d'intégration de bout en bout.

10. Pistes d'évolution et mini-projets TX

Plusieurs pistes peuvent prolonger ce travail et servir de mini-projets TX :

- Optimisation performance avancée : audit des re-renders avec React DevTools, réduction des fetchs inutiles, et exploration de solutions type Redux/Zustand pour la gestion d'état complexe.
- Mise en place d'un serveur SMTP : notifications email, préférences utilisateur, réinitialisation de mot de passe, sécurisation des flux et journaux d'envoi.
- Audit Lighthouse complet : objectif 100/100 en accessibilité, SEO et performance, avec corrections ciblées (images, scripts, contrastes).
- Observabilité et logs : centralisation des erreurs, dashboards de suivi et alerting en production.
- Extension IA : suggestion automatique d'ontologies, résumés multi-langues, et détection proactive d'incohérences dans les données.

11. Conclusion

Le travail réalisé sur TX Ontozipe va bien au-delà d'une amélioration cosmétique : l'application a été reconstruite comme un produit solide, sécurisé et scalable. La refonte UI/UX, la modularisation backend, l'intégration du guide, et la documentation Swagger rendent la plateforme immédiatement exploitable.

L'innovation provient de la combinaison unique entre ontologie structurée, collaboration multi-organisation, IA intégrée et traçabilité via notifications. Ce socle constitue une vraie valeur différenciante et ouvre des perspectives fortes pour les futurs projets TX.