

MSPR TPRE532

Mise en production d'une solution I.A

2025 - 2026

MSPR TPRE532 - Mise en production d'une solution I.A

MSPR : Mise en Situation Professionnelle Reconstituée

Bloc E6.3 - Produire et maintenir une solution I.A

Certification IA et Data Science - RNCP 36581

Objectif général : mettre en production une application web complète (backend et frontend) prête à accueillir un modèle d'IA.

1.1 Mise en route : qu'est-ce qu'on va faire ensemble ?

Aujourd'hui, on démarre la MSPR TPRE532.

- MSPR, ça veut dire Mise en Situation Professionnelle Reconstituée. C'est un format d'évaluation où on vous met dans une situation qui ressemble le plus possible à une vraie mission en entreprise, avec un client, un cahier des charges, des contraintes, des livrables.

- Cette MSPR appartient au Bloc de compétences E6.3 - Produire et maintenir une solution I.A de votre certification Développeur en Intelligence Artificielle et Data Science (RNCP 36581).

1.2 Le scénario : ObRail Europe

Le client de ce projet, c'est ObRail Europe.

ObRail est un observatoire indépendant de la mobilité ferroviaire en Europe.

- ❑ Sa mission collecter et analyser les données sur les dessertes ferroviaires,
- ❑ mesurer l'impact environnemental des différents modes de transport longue distance,
- ❑ produire des études pour les décideurs politiques, les opérateurs ferroviaires et les ONG,
- ❑ et, encourager l'usage du train, de jour comme de nuit, comme alternative à l'avion sur les trajets intra-européens.

ObRail travaille avec la Commission européenne, le Parlement européen, des ONG environnementales comme Transport et Environment ou Back-on-Track, et des opérateurs comme SNCF, ÖBB Nightjet, DB, Trenitalia.

ObRail est fictif.

ObRail Europe - Le client du projet

Observatoire indépendant des dessertes ferroviaires en Europe

Mesure de l'impact environnemental des transports longue distance

Partenaires :

Institutions européennes (Commission, Parlement)

ONG (Transport et Environment, Back-on-Track)

Opérateurs (SNCF, ÖBB Nightjet, DB, Trenitalia)

Besoin : un service applicatif fiable, industrialisé, pour ses partenaires.

1.3 Où on en est dans le scénario ?

Avec la [MSPR TPRE512](#), vous avez construit un entrepôt de données et un premier prototype d'API REST pour exposer les données ferroviaires.

Aujourd'hui, ce prototype fonctionne, mais il est encore au stade ça marche localment .

Le déploiement est manuel, les tests ne sont pas automatisés, et il n'y a pas de supervision.

L'objectif de [TPRE532](#), c'est de passer de ce prototype à une application prête pour la mise en production :

- une application web complète (backend et frontend),
- conteneurisée (Docker),
- testée (unitaires, intégration, E2E),
- monitorée (métriques, logs, dashboards),
- documentée, et prête à accueillir un modèle d'IA dans une future [MSPR \(TPRE622\)](#).

1.3 Où on en est dans le scénario ?

- Donc on est exactement au milieu du chemin :
- Derrière vous : la construction du socle de données.
- Devant vous : l'intégration d'un modèle d'IA plus avancé.
- Ici : industrialiser, exploitable en conditions réelles.

Mission TPRE532 - Résumé du cahier des charges

- ❑ Industrialiser la solution existante : backend, frontend, base, monitoring
- ❑ Conteneuriser (Docker, Docker Compose) et rendre le déploiement reproductible
- ❑ Mettre en place une chaîne CI/CD (tests; build Docker; déploiement)
- ❑ Garantir supervision, tests, conformité RGPD et accessibilité (RGAA)

1.4 Les notions clés (Séance 1)

API : Application Programming Interface

C'est l'interface par laquelle un programme (par exemple le frontend) parle à un autre programme (le backend). Dans notre cas, c'est une API REST, donc basée sur HTTP, avec des routes comme /trajets, /health, etc.

Backend

C'est la partie côté serveur de l'application : le code qui reçoit les requêtes, applique la logique métier, appelle la base de données, renvoie des réponses JSON. Ici, le backend est un service web qui donne accès aux données ferroviaires harmonisées.

Frontend

C'est l'interface utilisateur, dans le navigateur : pages, formulaires, tableaux, graphiques. ObRail veut un frontend ergonomique et accessible, pensé pour des utilisateurs non techniques mais exigeants : ONG, institutions, opérateurs.

CI/CD : Continuous Integration / Continuous Delivery

C'est la chaîne qui automatise les tests, la construction des images Docker, et la mise à disposition de versions testables. ObRail attend un pipeline fonctionnel (GitHub Actions, GitLab CI ou Jenkins) documenté et reproductible.

Docker / Docker Compose

Docker sert à embarquer une application dans un conteneur reproductible. Docker Compose permet de lancer plusieurs conteneurs ensemble : backend, frontend, base de données, outil de monitoring, en une seule commande. C'est une exigence forte de l'énoncé : l'évaluateur doit pouvoir lancer tout le système en une fois.

RGPD / RGAA

ObRail ne manipule pas de données personnelles dans vos jeux de données, mais le projet doit respecter le RGPD dans sa façon de gérer les données et les logs. Et le frontend doit respecter les bonnes pratiques d'accessibilité numérique (référentiel RGAA), parce que le public visé inclut des institutions publiques.

1.5 Ce que demande exactement ObRail

1. Une industrialisation de la solution existante :

stabiliser et renforcer le backend et son architecture,

conteneuriser l'ensemble (backend, frontend, base, monitoring),

garantir la reproductibilité du déploiement via Docker / Docker Compose.

2. Une interface utilisateur professionnelle :

un frontend ergonomique et accessible,

qui permet de consulter et filtrer les trajets ferroviaires,

d'afficher des statistiques (répartition jour/nuit, volumes par opérateur, etc.),

et de visualiser l'état de santé du système (monitoring).

3. Une stratégie de tests complète :

tests unitaires et d'intégration pour le backend, tests E2E sur le frontend (Cypress, Playwright...), intégrés dans la CI/CD.

4. Une supervision et une observabilité :

un outil de monitoring (Metabase, Grafana, autre), des métriques sur l'API (disponibilité, latence, taux d'erreurs),

des logs exploitables pour diagnostiquer les incidents.

5. Une documentation complète :

architecture, pipeline CI/CD, procédures d'installation,

mesures de conformité RGPD et RGAA, plan de maintenance et de rollback.

1.6 Motivation pour la séance 1

L'activité spécifique de cette séance 1 :

- Bien comprendre le scénario ObRail et ce que raconte le cahier des charges,
- Poser une architecture de référence simple,
- Créer un backend minimal FastAPI et PostgreSQL,
- Implémenter GET /health pour préparer la supervision.

Proposer une architecture technique de référence (backend / frontend / DB / monitoring / CI/CD)

- Poser le socle technique backend :
 - repo,
 - structure FastAPI,
 - modèle Trip et base PostgreSQL,
 - premier endpoint /health

MSPR TPRE532

Mise en production d'une solution I.A - Cas ObRail Europe

Mettre en production une solution complète :

- Backend (API REST)

- Frontend (interface web)

- Base de données

- Monitoring (Grafana, Metabase...)

Conteneurisation (Docker / Docker Compose) pour tout relancer en 1 commande

Chaîne CI/CD : tests, build Docker, déploiement sur env. de test

Supervision, logs, conformité RGPD et accessibilité (RGAA)

Évaluation : ce que le jury regarde

- ✓ Qualité du backend (endpoints, tests, robustesse)
- ✓ Qualité du frontend (ergonomie, accessibilité)
- ✓ CI/CD : automatisation des tests et du déploiement
- ✓ Monitoring et observabilité (métriques, logs, dashboard)
- ✓ Documentation et capacité à expliquer vos choix (soutenance)

Ce que demande le cahier des charges (dev)

Endpoints obligatoires :

GET /trajets, GET /trajets/{id}

GET /stats/volumes

GET /health

Backend : erreurs gérées, sécurité basique, doc OpenAPI

Frontend :

filtrage de trajets,

indicateurs jour/nuit et opérateurs,

état de santé du service,

accessibilité RGAA

Objectifs en équipe

En équipe : clarifier le contexte ObRail et la mission MSPR

Poser une architecture de référence (schéma global)

Créer le squelette backend :

- projet FastAPI

- connexion PostgreSQL

- modèle Trip (structure de la table)

Implémenter GET /health minimal

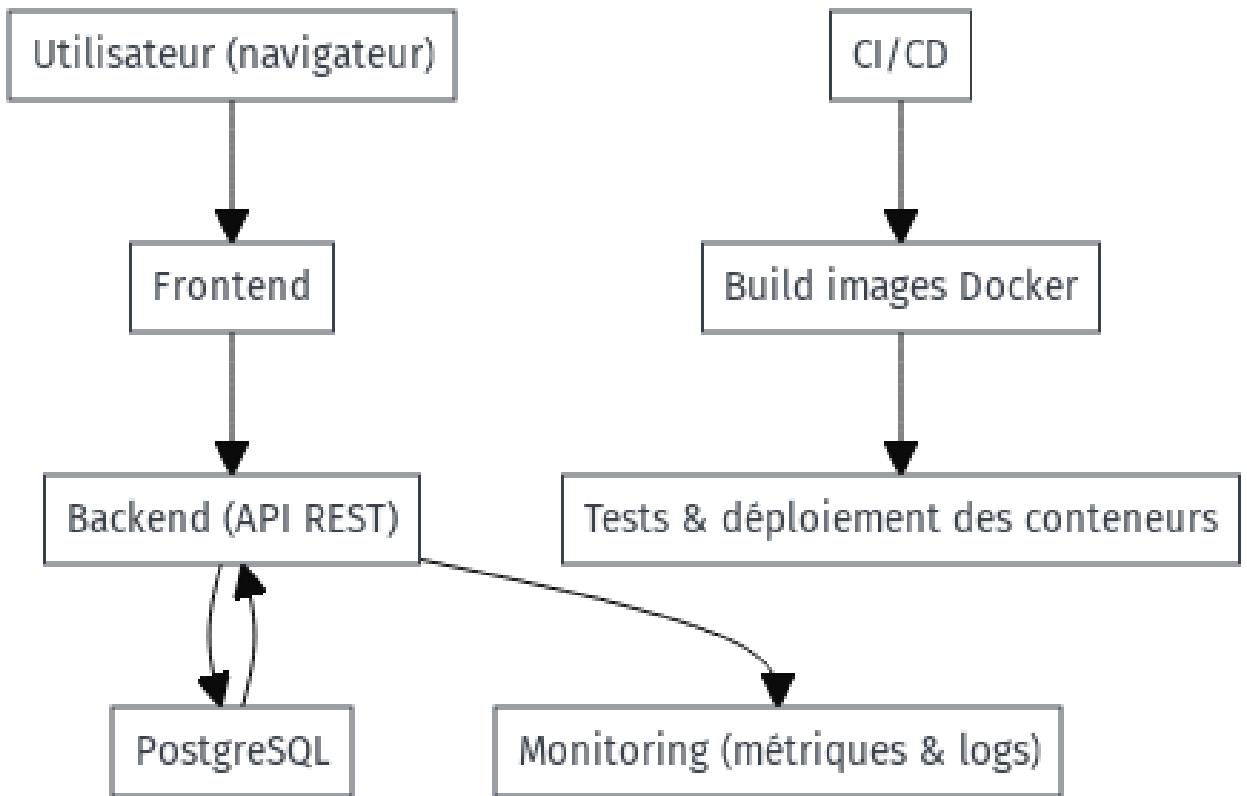
Architecture globale de la solution

- Backend API : FastAPI (Python)
- Base de données : PostgreSQL (référentiel de trajets)
- Frontend : SPA en React ou équivalent
- Monitoring : Grafana / Metabase sur métriques et DB
- CI/CD : GitHub Actions / GitLab CI, pipeline tests; build Docker → déploiement
- Orchestration : Docker ; Docker Compose

Schéma simplifié des flux

Schéma des flux (v1)

- Utilisateur (navigateur) parle avec Frontend
- Frontend parle à Backend (API REST)
- Backend discute avec PostgreSQL (lecture / écriture)
- Backend (API) envoie vers le Monitoring (logs et métriques)
- Pipeline CI/CD reconstruit les images Docker, lance les tests et déploie (Build Docker, Tests et déploiement).



Étapes backend (d'aujourd'hui)

- ✓ Créer le dépôt Git ObRail MSPR
- ✓ Initialiser un projet FastAPI minimal
- ✓ Configurer la connexion PostgreSQL
- ✓ Modéliser la table trips (en lien avec le dataset ObRail)
- ✓ Implémenter GET /health

Structure de base du backend

backend/

 main.py – point d'entrée FastAPI

 app/core/config.py – configuration (URL DB, etc.)

 app/db/ – base.py, models.py, session.py

 app/api/v1/ – routes (routes_health.py, routes_trajets.py)

Le modèle Trip (point de vue données)

Colonnes clés :

trip_id (id métier du trajet)

agency_name, train_type, service_type (Jour/Nuit)

origin_* / destination_* (gares + pays ISO-2)

departure_time, arrival_time

distance_km, duration_h

emission_gco2e_pkm, total_emission_kgco2e

frequency_per_week, source_dataset, last_update, traction

Endpoint /health minimal

Route GET /api/v1/health

Retour JSON minimal :

status : "ok" / "degraded"

api : true/false

db : true/false

version : version de l'API

Utilisé ensuite pour :

monitoring,

tableaux de bord,

tests automatiques.

Synthèse inter-séance

Checklist :

- Un dépôt Git initialisé pour le projet ObRail
- Un backend FastAPI qui démarre (uvicorn main:app --reload)
- Une base PostgreSQL accessible (via Docker ou locale)
- Un modèle Trip défini côté backend
- Un endpoint GET /api/v1/health fonctionnel et testé rapidement

Préparation Séance 2

Relire le cahier des charges (parties backend et stats)

Vérifier que votre /health est stable (même après redémarrage DB)

Optionnel :

Esquisser les filtres que vous voudriez sur /trajets

Réfléchir aux indicateurs qui vous semblent pertinents pour /stats/volumes

Conformité et contexte UE / France

1. App ObRail doit être conçue en gardant en tête le cadre européen (RGPD, accessibilité) et français (CNIL, RGAA).
2. Le jeu de données utilisé ne contient pas de données personnelles, ce qui limite les contraintes RGPD ; néanmoins, l'architecture respecte les principes de privacy by design (données minimales, séparation données métier et logs, secrets en variables d'environnement).
3. L'interface web applique quelques bonnes pratiques d'accessibilité inspirées de la norme EN 301 549 et du référentiel RGAA (structure sémantique, formulaires labellisés, navigation clavier).
4. Côté sécurité et disponibilité, nous isolons les composants (DB, API, frontend) via Docker, exposons un endpoint /health et prévoyons un rollback simple grâce au versionnage des images.
5. Enfin, le socle open source (FastAPI, PostgreSQL, React) et l'API documentée (OpenAPI/Swagger) facilitent un hébergement sur un cloud européen (« de confiance ») et l'intégration future dans des écosystèmes publics (points d'accès nationaux, portails open data).

Exigences front-end et accessibilité

Le front-end ObRail doit être :

Ergonomique (simple, clair, utilisable)

Accessible (RGAA / EN 301 549 / WCAG 2.1)

Accessibilité : rendre l'interface utilisable par :

personnes avec handicaps visuels, moteurs, cognitifs...

personnes avec équipements variés (clavier seul, lecteurs d'écran...)

Concrètement pour votre projet :

HTML sémantique (titres, listes, tableaux)

formulaires labellisés

navigation clavier possible

contrastes suffisants

Panorama des tests dans la MSPR

Back-end :

Tests unitaires

testent une fonction ou un endpoint isolé

ex. : vérifier que /health renvoie bien un JSON avec les bons champs

Tests d'intégration

testent plusieurs composants ensemble

ex. : /trajets , base PostgreSQL

Front-end :

Tests E2E (End-to-End)

simulent un vrai utilisateur dans le navigateur

ex. : ouvrir la page, remplir un filtre, vérifier que la liste de trajets se met à jour

Et aussi :

Tests de non-régression : vérifier qu'une nouvelle version n'a pas cassé ce qui marchait avant

Tests de sécurité : vérifier que certaines erreurs ne dévoilent pas d'infos sensibles

Surveiller la solution : monitoring et supervision

Superviser la solution IA, c'est :

Suivre son état en temps réel :

l'API répond-elle ?

la base de données est-elle joignable ?

Collecter des métriques :

disponibilité, temps de réponse

taux d'erreurs

Analyser et agir :

déetecter les incidents

corriger / redéployer si nécessaire

Dans ObRail :

/api/v1/health , première brique de supervision

plus tard : tableau de bord (Grafana, Metabase...) connecté à l'API et à la DB

Documentation : faire le récit de votre solution

Attendus de la MSPR :

Documentation technique claire et structurée :

architecture (schéma, choix techniques)

installation et déploiement (Docker, variables d'environnement)

endpoints API (exemples d'appels)

stratégie de tests et monitoring

aspects RGPD / accessibilité (front-end)

Dès maintenant, pensez à :

un README à la racine du projet

noter vos décisions techniques importantes

garder vos schémas d'architecture à jour

MSPR TPRE532

Mise en production d'une solution I.A

2025 – 2026

16/12/2025

MSPR TPRE532 - Contexte et objectif

Bloc : Produire et maintenir une solution I.A

Client fictif : ObRail Europe

Observatoire des trajets ferroviaires européens

Analyse jour - nuit, opérateurs, pays, émissions de CO₂

Votre mission : industrialiser une application web complète

Backend, frontend, base de données

CI/CD, monitoring, documentation

Architecture ObRail - Vue d'ensemble

Backend : API FastAPI (Python)

/health, /trajets, /stats...

Base de données : PostgreSQL

table principale trips

Frontend : React (ou équivalent)

interface utilisateur (tableaux, filtres, stats)

Autour :

Docker / Docker Compose pour lancer tous les services

CI/CD pour tests et déploiements

Monitoring (métriques, logs, /health)

Ce qui est déjà en place / démarré

Compréhension du cahier des charges ObRail

Mise en place du backend minimal :

- projet FastAPI structuré (core, db, api, schemas)

- connexion à PostgreSQL (DATABASE_URL, SessionLocal)

- modèle Trip défini dans models.py

Implémentation du endpoint /api/v1/health :

- vérifie l'API et la DB

- renvoie un JSON avec status, api, db, version

Objectifs

vérifier que le backend minimal fonctionne

vérifier que /api/v1/health répond correctement

Implémenter /trajets

lecture de vrais trajets dans la DB

filtres simples (origine, destination, jour/nuit, opérateur...)

pagination (page, taille de page)

Commencer les /stats

indicateurs globaux (nombre total de trajets, répartition jour/nuit, volumes par opérateur)

Introduire les tests

un premier test automatique sur /health ou /trajets

MSPR TPRE532

Mise en production d'une solution I.A

2025 – 2026

05/01/2026

A. Pour vérifier le travail (socle backend et /health)

Contexte et MSPR

C'est quoi ObRail Europe et à quoi sert notre application ?

À quoi sert la MSPR TPRE532, par rapport à un simple TP de code ?

Architecture générale

Quelles sont les trois briques principales de notre application ObRail ?
(frontend, backend/API, base de données PostgreSQL)

Modèle Trip

Quelles sont les colonnes les plus importantes de la table trips ?
Que représente le champ service_type ? Et emission_gco2e_pkm ?

API et /health

Qu'est-ce qu'un endpoint dans une API REST ? Donnez un exemple dans notre projet.
Qu'est-ce que retourne /api/v1/health et à quoi ça sert ?

Industrialisation

Pourquoi on dit que /health est une brique de monitoring ?
Quelle est la différence entre ça marche sur mon PC et application industrialisée ?

<https://fullstackopen.com/fr/>

intro à la dev web avec SPA React et REST APIs.

<https://cs50.harvard.edu/web/>

Couvre design et implémentation de web apps.

A. Pour vérifier le travail (socle backend + /health) – Pouvoir montrer:

Lancer l'API

Lancez le backend et montrez que FastAPI démarre sans erreur.

Ex. uvicorn main:app --reload fonctionne, /docs accessible.

Structure du projet

Ouvrez le projet et montrez où sont :

la config,

les modèles de base de données,

les routes.

Au moins une structure (core, db, api, schemas, ...).

Connexion DB

Dans le code, où est définie la DATABASE_URL ? Comment l' API se connecte à PostgreSQL ?
(montrer config.py, session.py ou équivalent).

Modèle Trip

Montrez la classe Trip dans le code. Est-ce qu'elle correspond à la table trips ?
(types, noms de champs cohérents).

Endpoint /health

Appelle /api/v1/health depuis ton navigateur ou /docs et montre-moi la réponse JSON.
(vérifiez que les champs status, api, db, version sont là et cohérents).

B. Pour vérifier le travail de Session 2 (/trips, stats, test)

Filtres et pagination

Pourquoi on a besoin de **pagination** sur /trips ?

Comment calcule-t-on l'offset à partir de page et page_size ?

Donnez un exemple de filtre métier utile sur /trips pour ObRail.

Réponse /trips

Quels champs doit contenir la réponse JSON de /api/v1/trips en plus de la liste des trajets ?

(Attendu : items, total, page, page_size)

Stats

Pourquoi ObRail ne veut pas seulement une liste brute de trajets, mais aussi des endpoints /stats ?

Citez au moins deux indicateurs qu'on a prévu de renvoyer dans /stats/summary.

Tests

À quoi sert un test automatique sur notre API ?

Quelle différence faites-vous entre un test unitaire et un test d'intégration, dans notre contexte ?

Quel endpoint avons-nous choisi pour écrire un premier test simple ?

B. Pour vérifier le travail de Session 2 (/trips, stats, test)

Endpoint /trips : base

Montrez dans /docs l'endpoint /api/v1/trips. Lancez-le sans paramètres et montrez :
qu'il renvoie bien du JSON, qu'il y a un tableau de trajets.
(l'endpoint existe et renvoie des données cohérentes).

Endpoint /trips : filtres

Appellez /trips avec au moins un filtre, par exemple :
origin_country=FR ou service_type=Nuit. Montrez que la liste renvoyée change.
(les filtres sont bien pris en compte)?

Endpoint /trips : pagination

Vérifiez que la pagination fonctionne (contenu items différent, page correct, page_size respecté).

Schémas Pydantic

Montrez le schéma de réponse utilisé pour /trips (ton TripListResponse ou équivalent).
(la séparation est la : TripOut et TripListResponse).

Endpoint /stats/summary

Appellez /api/v1/stats/summary et montrez le JSON.
Les indicateurs de base sont présents (total_trips, avg_distance_km, etc.).

Premier test automatique

Montrez un fichier de test (par ex. test_health.py) et lancez pytest. Est-ce que le test passe ?
(vous avez au moins 1 test, vous avez bien lancé la commande.)

Séance 3 - Frontend ObRail : React, API et Accessibilité

Consommer les endpoints existants :

/api/v1/health

/api/v1/trips

/api/v1/stats/summary

Construire une interface web qui permet de :

voir l'état du service,

explorer les trajets avec filtres et pagination,

afficher des statistiques globales.

Appliquer quelques règles :

d'ergonomie (interface claire et logique),

d'accessibilité (RGAA / WCAG),

et préparer l'interface à des tests automatiques plus tard.

Application React qui parle à l'API que vous avez déjà développée : /health, /trips, /stats/summary.
L'objectif est d'avoir une interface : compréhensible, utilisable, et accessible.

On va aussi poser des bases qui serviront plus tard pour les tests E2E et la mise en production.

Séance 3 - Frontend ObRail : React, API et Accessibilité

Consommer les endpoints existants :

/api/v1/health

/api/v1/trips

/api/v1/stats/summary

Construire une interface web qui permet de :

voir l'état du service,

explorer les trajets avec filtres et pagination,

afficher des statistiques globales.

Appliquer quelques règles :

d'ergonomie (interface claire et logique),

d'accessibilité (RGAA / WCAG),

et préparer l'interface à des tests automatiques plus tard.

Application React qui parle à l'API que vous avez déjà développée : /health, /trips, /stats/summary.
L'objectif est d'avoir une interface : compréhensible, utilisable, et accessible.

On va aussi poser des bases qui serviront plus tard pour les tests E2E et la mise en production.

Séance 3 - Frontend ObRail : React, API et Accessibilité

Documentation RGAA v4.1.2

DINUM (France)

Référentiel officiel français d'accessibilité numérique, aligné sur **EN 301 549** et **WCAG 2.1**.

Référence normative pour expliquer pourquoi l'interface ObRail respecte un minimum sérieux : structure HTML sémantique, formulaires labellisés, gestion des messages d'erreur et de chargement.

Séance 3 – Rappel backend

Endpoints disponibles :

GET /api/v1/health

(indique si l'API répond, si la base de données répond, et la version du service).

GET /api/v1/trips

renvoie une liste de trajets, avec :

filtres (pays d'origine / destination, service_type, agency_name...)

pagination (page, page_size)

format de réponse : { items, total, page, page_size }.

GET /api/v1/stats/summary

renvoie quelques indicateurs globaux :

total_trips,

avg_distance_km,

avg_emission_gco2e_pkkm.

Suite: Le frontend va consommer proprement ces trois endpoints et les présenter de façon claire à l'utilisateur.

Séance 3 - Vue utilisateur : scénarios ObRail

Un utilisateur ObRail doit pouvoir, depuis l'interface :

Vérifier rapidement si le service est disponible

ex. badge Service OK / Dégradé / Indisponible.

Explorer les trajets ferroviaires :

filtrer par pays d'origine / destination

filtrer par type de service (Jour / Nuit)

filtrer par opérateur

passer d'une page de résultats à l'autre.

Comprendre les statistiques globales de l'offre :

combien de trajets ?

quelle distance moyenne ?

quelles émissions moyennes par km ?

Avant de parler de composants React, on se met dans la peau d'un utilisateur ObRail.

Est-ce que l'outil marche ?

Quels trajets existent entre tel et tel pays ?

Quels chiffres clés je peux mettre dans mon rapport ?

Transformer vos endpoints techniques en une interface qui répond à ces questions-là (et d'autres encore).

Séance 3 - Architecture du frontend, Architecture de l'app React

On se fixe une petite architecture :

- ❑ un composant App qui contient un header commun et le contenu principal,
- ❑ une TripsPage dédiée aux trajets (filtres, tableau, pagination),
- ❑ un StatsPanel pour les chiffres clés,
- ❑ un HealthBadge pour l'état du service,
- ❑ et un fichier api/client.ts pour centraliser tous les appels à l'API.

Séance 3 - Architecture du frontend, Architecture de l'app React

Organisation proposée :

App

Header

 Titre: ObRail Europe – Explorateur de trajets

 HealthBadge (état du service)

Main

 TripsPage

 formulaire de filtres

 tableau de trajets

 pagination

 StatsPanel

 tuiles avec total_trips, avg_distance_km, avg_emission_gco2e_pk

api/client.ts

 fonctions fetchHealth, fetchTrips, fetchStatsSummary

Séance 3 - Pattern : centraliser les appels API

Ne pas mettre `fetch(...)` dans tous les composants.

Créer un fichier `api/client.ts` qui :

- connaît l'URL de base (`API_BASE_URL`),
- expose des fonctions utilitaires :
 - `fetchHealth()`
 - `fetchTrips(query)`
 - `fetchStatsSummary()`

Chaque fonction :

- construit l'URL et les query params,
- gère les erreurs de base (`res.ok`),
- renvoie les données typées (TypeScript) au composant.

Plutôt que d'écrire `fetch(http://localhost:8000/api/v1/trips?...)` un peu partout, on va faire ce que font les pros : un petit client API.

- Dans `api/client.ts`, on aura :
- les types TypeScript (`Trip`, `TripsResponse`, etc.),
- les fonctions `fetchHealth`, `fetchTrips`, `fetchStatsSummary`.
- Les composants React n'auront plus à se soucier de l'URL exacte ou de la gestion de `res.ok`. Ils appelleront juste `fetchTrips({ page: 1, origin_country: "FR", ... })`.
- Ça rend le code plus clair, plus testable et plus facile à faire évoluer.

Séance 3 - HealthBadge : état du service et accessibilité

Rôle du composant HealthBadge (votre application est en forme) :

- Appeler **/api/v1/health** au chargement.
- Gérer plusieurs états :
 - **chargement** : État du service : chargement...
 - **succès** : badge Service OK / Dégradé / Indisponible
 - **erreur** : message État du service : erreur (...)
- Accessibilité :
 - utiliser `aria-live="polite"` pour annoncer les changements,
 - fournir un texte lisible (pas seulement une couleur ou une icône).

HealthBadge doit :

- appeler `/health`,
- afficher un message pendant le chargement,
- afficher un texte clair quand tout va bien ou quand c'est dégradé,
- afficher une erreur si l'API ne répond pas.
- Pour l'accessibilité, on ajoute `aria-live` pour que les lecteurs d'écran soient informés du changement, et on évite de s'appuyer uniquement sur une couleur

Séance 3 - TripsPage : à quoi doit ressembler la page Trajets ?

Avant même de coder, on dessine mentalement la page Trajets.

On veut quelque chose de simple et logique :

- en haut, les filtres,
- en dessous, les résultats sous forme de tableau,
- en bas, la pagination.

Et :

- toujours un message de contexte : combien de trajets ? quelle page ?
- jamais un écran vide sans explication : si ça charge, on le dit ; s'il n'y a pas de résultat, on le dit ; s'il y a une erreur, on le dit.
- l'ergonomie de base fait toute la différence pour un utilisateur.

Séance 3 - TripsPage : à quoi doit ressembler la page Trajets ?

Organisation de la page (suggestion):

Bloc Filtres :

- pays d'origine (FR, DE, ...)
- pays de destination
- type de service (Jour / Nuit)
- opérateur (texte libre)

Bloc Résultats :

- texte : N trajets trouvés, page P / N
- tableau des trajets (opérateur, type, origine, destination, distance, durée)

Bloc Pagination :

- bouton "Page précédente"
- bouton "Page suivante"

Messages :

- Chargement des trajets...
- Aucun trajet trouvé.
- Erreur : impossible de récupérer les trajets.

Séance 3 - TripsPage : implémenter filtres & pagination

Logique principale :

État local :

- originCountry, destinationCountry, serviceType, agencyName
- page (page courante)
- data (réponse de l'API)
- loading, error

Fonction loadTrips(page) :

- appelle fetchTrips({ page, page_size, ...filtres })
- met à jour data et page
- gère loading et error

Pagination :

- totalPages = ceil(total / page_size)
- désactiver les boutons si page <= 1 ou page >= totalPages.

Séance 3 - TripsPage : à quoi doit ressembler la page Trajets ?

Côté code, TripsPage va :

- stocker les valeurs des filtres dans son state,
- stocker aussi la page courante, les données, l'état de chargement et les erreurs,
- avoir une fonction `loadTrips(page)` qui se charge d'appeler l'API avec les bons paramètres.
- Quand l'utilisateur clique sur Appliquer les filtres, on rappelle `loadTrips(1)` pour revenir à la première page.
Les boutons de pagination appellent `loadTrips(page - 1)` ou `loadTrips(page + 1)`, en vérifiant qu'on ne sort pas des bornes.

C'est une structure classique pour une page de recherche / résultats

StatsPanel : présenter les stats métier, quelles stats montrer, pourquoi ?

ObRail a besoin de chiffres synthétiques:

Avec /stats/summary, on va montrer au minimum :

- combien de trajets il y a dans la base (ordre de grandeur de l'offre),
- la distance moyenne,
- les émissions moyennes par km-passager.

Ces trois chiffres, présentés dans des petites cartes claires, sont bien pour un tableau de bord.

Vous pourrez les commenter facilement: l'offre est principalement composée de trajets de X km en moyenne, avec une intensité carbone de Y gCO₂e/pkm.

StatsPanel : présenter les stats métier, quelles stats montrer, pourquoi ?

Endpoint : GET /api/v1/stats/summary

Exemples d'indicateurs :

total_trips

combien de trajets en base (taille de l'offre).

avg_distance_km

distance moyenne d'un trajet (idée du type d'offre : plutôt courts, moyens ou longs trajets).

avg_emission_gco2e_pkkm

intensité carbone moyenne par passager-kilomètre.

Affichage dans l'UI :

3 blocs côte à côte :

Total de trajets

Distance moyenne

Émissions moyennes

Accessibilité (RGAA) : notre minimum

Quand on parle d'accessibilité pour le web en France, on se réfère au RGAA, qui s'appuie sur une norme européenne (EN 301 549) et sur les règles WCAG 2.1 du W3C.

- ObRail est un outil destiné à des acteurs publics et des ONG, donc on ne peut pas ignorer ces sujets. Ici, on ne va pas faire un audit RGAA complet, mais on va viser un minimum sérieux :
- utiliser une structure HTML logique (`<header>`, `<main>`, `<section>`, `<table>`...),
- relier des labels à chaque champ de formulaire,
- garder la navigation clavier possible,
- afficher des messages clairs pour les erreurs ou le chargement.

Ça suffit déjà pour entrer dans une démarche d'accessibilité cohérente avec ce qu'on attend en MSPr.

Accessibilité (RGAA) : notre minimum

En France, on parle d'accessibilité numérique via :

- RGAA : Référentiel Général d'Amélioration de l'Accessibilité
- basé sur la norme européenne EN 301 549
- et sur les règles WCAG 2.1 (W3C)

ObRail est pensé pour des institutions publiques / ONG, on doit respecter un minimum de ces règles.

- Dans ce projet, ça implique de soigner :
- la structure HTML (balises sémantiques)
- les formulaires (labels, aides)
- la navigation clavier et les messages (erreurs, chargement)

Erreurs, pas de données : comment réagir côté UI ?

UX et résilience

Différencier clairement :

Erreur technique (API down, réseau coupé) :

message type :

Impossible de joindre le serveur. Vérifiez votre connexion ou réessayez plus tard.

HealthBadge : Service indisponible.

Aucun résultat :

message type :

Aucun trajet ne correspond à ces critères.

Chargement :

Chargement des trajets...

Chargement des statistiques...

Bonne pratique :

toujours un message textuel explicite,
utiliser role="alert" pour les erreurs.

Erreurs, pas de données : comment réagir côté UI ? UX et résilience

Vous devez distinguer :

- une erreur technique (serveur éteint, API down); message type « Impossible de joindre le serveur... »,
- un cas sans résultat; message « Aucun trajet ne correspond à ces critères. »,
- et l'état de chargement.

L'utilisateur ne doit jamais se retrouver devant un écran vide sans savoir pourquoi.

Et pour l'accessibilité, on pense à utiliser role="alert" pour les erreurs, pour que les lecteurs d'écran les annoncent correctement.

Tests côté frontend : la suite

Vous avez déjà vu les tests côté backend.

Côté frontend, l'idée est la même :

on peut tester des composants en isolé (par exemple vérifier que le HealthBadge affiche « Service OK » si l'API renvoie status: « ok »),

on peut tester l'application de bout en bout : ouvrir un navigateur, simuler un vrai utilisateur, et vérifier le comportement.

Aujourd'hui, on ne va pas écrire tout de suite ces tests-là, mais on construit une interface qui s'y prête : texte clair, composants bien séparés, comportements prévisibles.

On y reviendra quand on parlera de CI/CD et de qualité globale.

Tests côté frontend : la suite

Ce que vous avez déjà :

Des tests backend (pytest, TestClient) sur /health et /trips.

Pour le frontend, on vise :

des tests de composants (ex. React Testing Library) :

- rendre <HealthBadge /> avec une API mockée
- vérifier l'affichage Service OK.

des tests E2E (Cypress, Playwright...) :

- ouvrir la page
- remplir des filtres
- cliquer sur Appliquer les filtres
- vérifier que le tableau et les stats se mettent à jour

Dans cette séance :

on conçoit l'UI pour qu'elle soit testable, l'écriture des tests sera intégrée plus tard (CI/CD).

Récap et checklist

Pour clôturer la séance, vérifiez que vous pouvez cocher ces cases :

- votre app React tourne,
- le HealthBadge affiche réellement l'état du service,
- la page des trajets permet de filtrer et de paginer,
- les statistiques globales sont visibles,
- et votre interface n'est ni un écran blanc, ni un truc qu'on ne peut utiliser qu'à la souris.

Si c'est le cas, vous avez un frontend ObRail aligné avec la grille de compétences et prêt pour la suite : Docker, CI/CD, monitoring et tests E2E.

Récap et checklist

Checklist frontend ObRail :

- Une application React qui démarre (npm run dev).
- Un **HealthBadge** connecté à /api/v1/health :
 - affiche chargement, état OK / dégradé / indisponible, et erreurs.
- Une **TripsPage** qui :
 - propose des filtres (pays, type, opérateur),
 - affiche un tableau de trajets,
 - gère la pagination (page précédente / suivante),
 - affiche les messages « chargement », « aucun trajet », « erreur ».

- Un **StatsPanel** qui consomme /api/v1/stats/summary et affiche au moins :
 - total de trajets,
 - distance moyenne,
 - émissions moyennes.

Une interface :

- structurée (header, main, section, table, ...),
- utilisable au clavier,
- avec messages clairs pour les états du système.