Proposition Architecturale : Création d'un crate oar metier (domain ou model) dédié

Résumé Exécutif

Dans le cadre de l'évolution de l'API OAR Scheduler, cette proposition détaille et justifie la création d'un crate oar-scheduler-model dédié. Ce crate centralisera les modèles de données et la logique métier de base, les séparant des couches d'accès aux données (oar-scheduler-db), des algorithmes de planification (oar-scheduler-core) et des futures interfaces de communication. Cette refonte est un prérequis technique à l'implémentation d'une API REST robuste et indépendante.

Analyse de l'architecture actuelle et points de friction

L'architecture actuelle répartit les responsabilités de manière sous-optimale entre les crates existants :

Problème 1 : couplage fort

Le graphe de dépendances actuel force des liens étroits entre les couches :

```
oar-scheduler-redox → oar-scheduler-db → oar-scheduler-core
oar-scheduler-redox → oar-scheduler-core
```

Mélanger les types liés aux requêtes SQL (via sea-query/sqlx) avec les structures de l'algorithme d'ordonnancement pur entraîne des temps de compilation plus longs et une dette technique importante. En Rust, maintenir un graphe de dépendances acyclique (DAG) strict et unidirectionnel est impératif pour la scalabilité du projet.

Problème 2 : Duplication et incohérence des modèles

Il existe actuellement deux définitions du modèle Job :

- **Job DB (oar-scheduler-db)** : Spécifique aux opérations SQL, inclut des enums liés à la base de données.
- **Job Core (oar-scheduler-core)** : Axé sur la logique métier et la planification.

L'absence d'une source de vérité unique rend la conversion complexe, propice aux erreurs et difficile à maintenir lors de l'ajout de nouvelles fonctionnalités.

Architecture Recommandée

La solution consiste à extraire les structures fondamentales dans un nouveau crate oar-scheduler-model situé à la racine du graphe de dépendances.

Nouveau flux de dépendances :

```
oar-scheduler-model (Source de vérité, aucune dépendance interne)
       ↑
       ├── oar-scheduler-core (Algorithmes de planification)
       ├── oar-scheduler-db   (Persistance et mapping SQL)
       ├── oar-scheduler-api  (NOUVEAU : Module REST API)
       └── oar-scheduler-redox (Bindings Python)
```

Nécessité pour le développement d'une API REST robuste

L'ajout d'un crate modèle dédié est la pierre angulaire de la future stratégie API du projet pour les raisons suivantes :

Stabilité du contrat d'interface

Une API REST doit offrir une stabilité à long terme pour les clients externes. En séparant les modèles métiers des structures internes de calcul (core) ou de stockage (db), nous créons une couche tampon. Cela permet de modifier l'algorithme d'ordonnancement ou le schéma de base de données sans provoquer de "breaking changes" sur les endpoints de l'API.

Gestion efficace des DTO

L'API REST ne doit pas nécessairement exposer l'intégralité des structures internes. Le crate model permet de définir proprement :

- **Les Internal Models** : Utilisés pour la logique complexe.
- **Les API Resources/DTOs** : Versions simplifiées ou agrégées destinées à la sérialisation JSON.

L'utilisation de traits comme serde::Serialize et Deserialize centralisés garantit que la représentation des données est identique, qu'elle soit consommée par le module Python ou via une requête HTTP.

Réduction de l'empreinte mémoire et binaire

Le module oar-scheduler-api n'a pas besoin de compiler l'intégralité du moteur de planification (core) ou les pilotes de base de données lourds pour valider une structure de données reçue en entrée. En dépendant uniquement de model, le microservice API reste léger, rapide à compiler et facile à conteneuriser.

Documentation automatisée (OpenAPI)

La centralisation des modèles permet d'intégrer des outils de génération de documentation (comme utoipa en Rust). En annotant les structures dans le crate model, l'API REST peut générer automatiquement un fichier openapi.json précis, reflétant fidèlement la logique métier du scheduler.