Database evolution and migration

AMT

7 - Database evolution and migration

Résumé du document

Definition

Table des matières

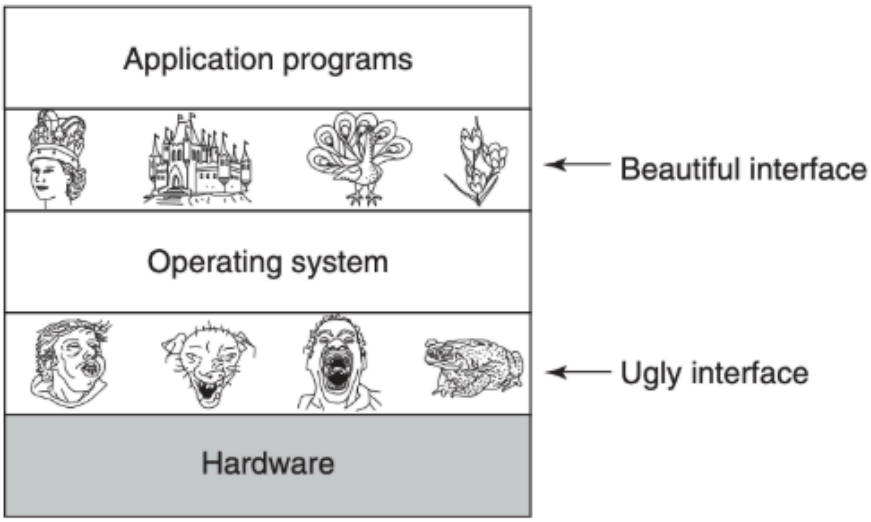
1. Abstraction dans l'ingénierie logicielle

L'abstraction est un skill et processus important en ingénierie logicielle. Elle permet de cacher les détails d'implémentation et de se concentrer sur les aspects importants du système. L'abstraction est utilisée pour simplifier la complexité du système et pour faciliter la maintenance et l'évolution du système. Cela permet notamment de :

- Généraliser (grouper, simplifier, catégoriser)
- Masquer les détails d'implémentation
- Se concentrer sur des aspects importants
- Interface ou modèle pour interagir avec les abstractions
- Designer des décisions

The job of the operating system is to create good abstractions and then implement and manage the abstract objects thus created. [Abstractions] are one of the keys to understanding operating systems.

Un concept d'abstraction que nous connaissons tous est l'OS. Il permet d'accéder à des ressources matérielles sans avoir à se soucier des détails d'implémentation.



1.1. Abstraction dans Java

En Java nous faisons de l'abstraction car le code que nous écrivons sera compatible avec n'importe quel OS qui supporte Java. Java fournit une interface pour interagir avec les abstractions. Par exemple, la classe `File` permet d'interagir avec les fichiers sans se soucier de la manière dont le système d'exploitation gère les fichiers.

1.2. Abstraction défaillante (Leaky Abstraction)

Le concept d'**abstraction défaillante** (ou "leaky abstraction") décrit le fait qu'aucune abstraction complexe n'est totalement étanche. Une abstraction est censée masquer les détails d'implémentation pour simplifier notre compréhension ou notre interaction avec un système. Cependant, dans la pratique, ces abstractions laissent parfois "fuir" des détails sous-jacents, ce qui signifie que des problèmes liés aux couches inférieures de l'implémentation peuvent remonter à la surface.

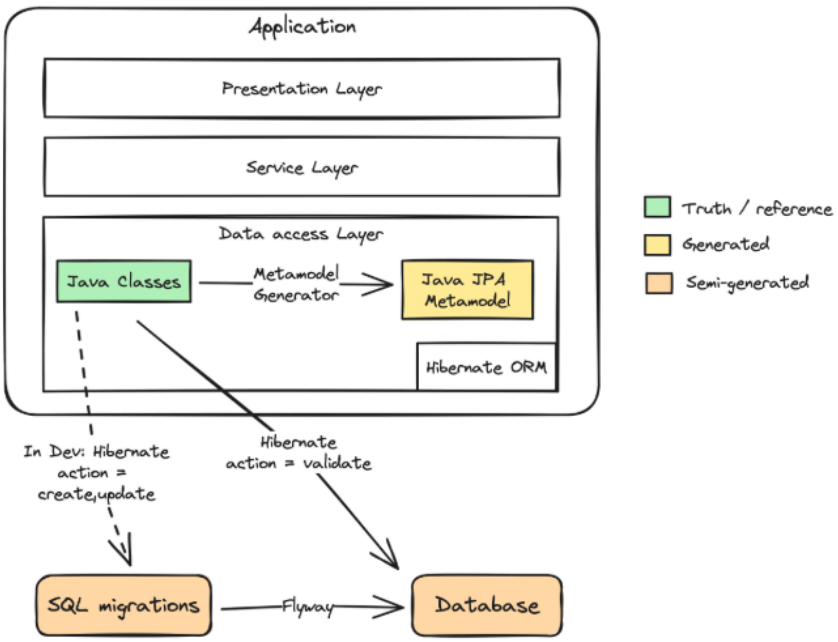
Selon Joel Spolsky, toutes les abstractions non triviales sont sujettes à ces fuites, dans une certaine mesure. Cela peut se manifester par des comportements inattendus, des erreurs ou des limitations qui nécessitent de comprendre les détails cachés sous l'abstraction. Cette situation réduit l'efficacité de l'abstraction, car elle ne simplifie pas autant qu'elle le devrait et force souvent les utilisateurs à plonger dans des couches de complexité qu'ils espéraient éviter.

2. Schema/Code génération

Grâce à Hibernate, nous avons vu qu'il était possible de générer un schéma relationnel en se basant sur les classes Java crée au préalable. Cela consiste en une abstraction très pratique mais qui peut dans certains cas être défaillante.

2.1. Migration ORM → SQL

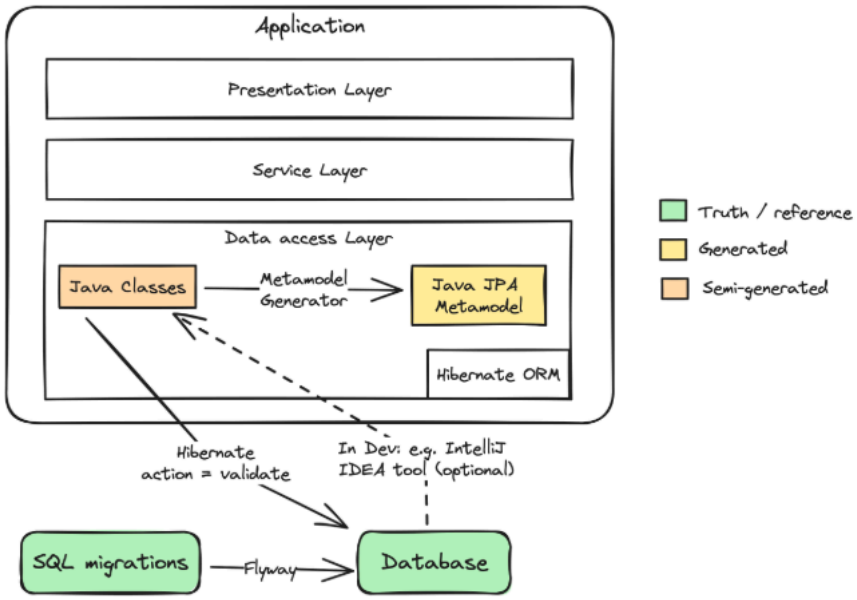
Dans le cas ou l'abstraction est défaillante, nous pouvons mettre en place un processus de migration **automatiquement généré**, **semi-automatique** ou **manuel**. Cela permet de garder les classes Java comme source principale de la structure et d'avoir la flexibilité de les adapter pour optimiser le schéma relationnel.



2.2. Migration SQL → ORM

Dans le cas ou le développement commence par l'adaptation du schéma SQL, il existe plusieurs outils permettant de générer automatiquement les classes Java correspondantes. Cela permet de garder le schéma relationnel comme source principale.

Cela permet notamment d'inclure directement les triggers, partitions et règles venant du schéma de la base de données.



2.3. Abstraction ORM

La génération de schéma est un outil vraiment intéressant et très puissant dans les premières phases d'un projet. Cependant, il est fort probable qu'une rupture ai lieu entre le schéma relationnel et les classes Java. Cela peut être dû à des contraintes de performance, de sécurité, de maintenance, etc. C'est pourquoi il est crucial de comprendre le processus d'abstraction dans sa globalité.

De manière générale il plusieurs critères sont important à prendre en compte pour mettre en place de l'abstraction:
- durée de vie de l'application
- skills de l'équipe
- complexité du domaine
- pertinence de l'abstraction

Lors-ce que l'on dépend d'une abstraction qui génére du code ou un schéma il est important de:
- en avoir le contrôle
- traquer les changements
- apprendre et comprendre l'abstraction

3. Migration de base de données

Lorsque l'on développe une application, il est fréquent que la structure de la base de données évolue au fil du temps. Nous retrouvons alors 2 situation bien distinctes:

- **En développement** → il est facile de vider la database, d'appliquer le nouveau schéma et de réeinserer les données de développement.
- **En production** → il est impossible de vider la database et de réeinserer les données. Il est donc nécessaire de mettre en place un processus de migration.

Flyway est un exemple d'outil de migration qui permet de gérer des scripts de migration. Il permet d'exécuter automatiquement ces scriptes dans le bon ordre et de tracker quels scritps ont déjà été exécutés.

3.1. Utilisation de PL/SQL

Dans certains cas il est nécessaire d'utiliser une logique plus complexe que le standard proposé par des requêtes SQL. Voici un exemple en utilisant `PL/pgSQL`:

```
CREATE OR REPLACE FUNCTION migrate_employees(min_salary INT)
RETURNS VOID AS $$
DECLARE
    emp_record RECORD;  -- Variable to hold employee data
BEGIN
    -- Loop through all employees from old_employees
    FOR emp_record IN
        SELECT id, name, salary FROM old_employees
    LOOP
        -- Check if salary is below the given minimum salary and apply a raise
        IF emp_record.salary < min_salary THEN
            -- Migrate with a 10% raise
            INSERT INTO new_employees (id, name, salary)
            VALUES (emp_record.id, emp_record.name, emp_record.salary * 1.10);
        ELSE
            -- Migrate without changes
            INSERT INTO new_employees (id, name, salary)
            VALUES (emp_record.id, emp_record.name, emp_record.salary);
        END IF;
    END LOOP;

    RAISE NOTICE 'Employee data migration completed.';
END;
$$ LANGUAGE plpgsql;
```

3.2. Risques

Lors d'une application de migration, il est crucial de suivre 3 points très important:

- **Backup** → il est important de faire un backup de la base de données avant de lancer les scripts de migration.
- **Test** → il est important de tester les scripts de migration sur un environnement de test avant de les appliquer en production.
- **Integrité** → il est important de ne **pas** éditer, effacer, renomer ou changer un script qui a déjà été éxécuté.

Pour éviter des probkèmes de migration, l'anticipation est clé.

3.3. Rollback

La fonctionnalité de rollback est très importante pour une migration, cependant il existe quelques limitations:

- Les changements destructifs tel que le destruction d'une table ou d'une colonne entrainent une perte de données qui est irréversible.
- Lors-ce que l'on doit rollback une migration, on se retrouve souvent à devoir créer nous même le script de rollback.

Là encore, il est essentiel d'anticiper lors de la rédaction des scripts de migration. Par exemple, si un script de migration supprime des données, il est judicieux de marquer d'abord les données comme étant obsolètes, puis de les supprimer dans un script de migration ultérieur.

3.4. Refactoring

Lors-ce que l'on doit faire un refactoring de la base de données, deux situations se présentent:

- **Backward compatible** → le refactoring est compatible avec les versions précédentes de l'application.
- **Non backward compatible** → le refactoring n'est pas compatible avec les versions précédentes de l'application et nécessite donc une mise à jour de l'application.

3.4.1. Non backward compatible

Dans le cas d'un refactoring non compatible avec la possibilité d'avoir un léger downtime, il est possible de mettre en place une nouvelle version de l'application en **coordonnant** le déploiement de la nouvelle version de l'application avec le refactoring de la base de données.

1. **Arrêter** toutes les instances de la version X de l'application, de sorte qu'aucune instance ne lise ou n'écrive dans la version Y du schéma de la base de données.
2. **Déployez** la version X+1 de l'application et exécutez les scripts de migration pour mettre à jour la base de données vers la version Y+1 du schéma, qui est compatible avec la nouvelle version de l'application.
3. La version d'application X+1 et le schéma de base de données Y+1 sont désormais opérationnels ensemble.

3.5. Expand et Contract

L'extension et la contraction est un modèle qui permet de transformer des changements incompatibles avec le passé en une chaîne de changements compatibles avec le passé, en trois phases distinctes. Les modifications du code et du schéma de la base de données sont découplées afin qu'elles puissent être développées, testées et déployées séparément.

- **Étendre**, introduire des modifications rétrocompatibles qui mettent en œuvre le remaniement du schéma de la base de données.
- **Migrer**, introduire des modifications de code qui s'adaptent aux nouvelles capacités du schéma de base de données remanié.
- **Contracter** (c'est-à-dire réduire la taille), consolidation en supprimant ou en nettoyant toutes les modifications non essentielles introduites dans la phase d'expansion pour soutenir la phase de migration et qui ne sont plus utilisées par le code.