[IMAG] Projet ECOM - Documentation Système

Version	Changements	Rédaction
1	Struture + BD + Load Balancer	LEGRAS Mathieu
2	Elasticsearch	BAILLE Mathieu
3	Diagramme de classe + BD	BERTOLINI Alban
4	Schéma conception système	BERTOLINI Alban

Septembre 2017

Table des matières

Chapitre 1

Présentation du projet

Nous nous plaçons dans un contexte de développement d'un site d'*e-commerce* nommé *Pokeshop*. Le but de ce site commercial est de fournir une plateforme en ligne de vente de Pokémons et de produits secondaires (par exemple des objets) à nos utilisateurs. Nos utilisateurs peuvent être non connectés ou connectés. Les utilisateurs non connectés ne peuvent que consulter le site. Les utilisateurs connectés peuvent acheter et échanger des Pokémons ou objets. Il existe un rôle spécial d'utilisateurs connectés qui est le statut de vendeur ; ces utilisateurs peuvent mettre en vente des Pokémons ou objets.

Ce document présente les différentes étapes du projet, la réalisation de ces étapes et des justifications quant aux choix effectués.

Chapitre 2

Structure du projet

2.1 Structure des ejbs

Nous avons essayé de séparer autant que possible les différentes couches métiers. Nous distinguons quatre parties importantes concernant nos EJBs :

- /ejbs/src/main/java/database/entities contient les entités correspondantes à la base de données. Le mapping est effectué grâce aux annotations JPA (dont l'implémentation est fournie par Hibernate). La majorité des entités contiennent des requêtes nommées associées.
- /ejbs/src/main/java/database/dao contient les DAOs utilisés au sein du projet, qui sont les objets fournissant l'accès à la base de données (nous permettant d'effectuer des requêtes de recherche ou de persister nos données). Nous avons un DAO générique permettant d'effectuer la majorité du travail : insérer, mettre à jour ou supprimer des données ainsi que faire des recherches à partir de requêtes nommées.
- /ejbs/src/main/java/database/elasticsearch contient le code associé à ElasticSearch, détaillé ci-dessous.
- /web/src/main/java/API/resources contient les servlets qui exposent nos données.

2.2 ElasticSearch

La méthode *ElasticSearch* a été choisie pour plusieurs raisons :
- rapidité de recherche (Near Realtime NRT)
- compatibilité avec JDBC et autre type d'entrée
- scalabilité infinie
- gratuité (open source)
- documentation très fournie

La mise en place d'un *cluster* elasticsearch a été fait sur une machine et l'intégration des données de la base PostgreSQL a été fait via *Logstash* (fait parti de l'ElasticStack). Celui-ci recharge les données toutes les minutes mais ne permet pas la suppression des données (on doit donc effectuer un post traitement pour savoir si l'article est toujours accessible).

2.3 Structure Angular

La structure du code Angular respecte autant que possible les conventions, à savoir :

- Principe de responsabilité unique
- Une navigation aisée du code grâce à des dossiers qui correspondent aux features

2.4 Utilitaires

Le premier utilitaire développé a été un outil conçu pour, à partir d'une API déjà existante, récupérer des données générales sur les Pokémons existants. Ces données récupérées sont intégrées à des inserts SQL, visibles dans nos différents scripts SQL. Ils permettent aussi de rapidement re-peupler la base de données, au moins pour les données "fixes". Les données propres à l'utilisation (utilisateurs, pokémons

en vente, etc.) étaient quant à elle généralement insérées à la main pour effectuer de rapides tests (à défaut ici d'un script SQL insérant des données de test établies à l'avance).

Les autres utilitaires sont principalement des scripts permettant de déployer facilement sur les machines virtuelles, afin de tester à distance. Le script de base (pouvant être facilement adapté aux besoins de chacun) fait en sorte de repartir de 0 à chaque fois qu'il est lancé. Les actions effectuées sont :

— Tuer les instances des serveurs wildfly
— Supprimer l'installation actuelle de wildfly
— Cloner ou mettre à jour le projet à partir du dépôt git
— Build le projet en construisant un .war
— Réinstaller et reconfigurer wildfly
— Déployer le .war sur le serveur

Il manque le déployement de l'interface Web qui, pour le moment, est effectué à la main.
Un dernier utilitaire a été prévu en début de projet, qui permettait de crypter les mots de passe des utilisateurs, grâce à une combinaison de MD5 et SHA256, avant de les insérer en base de données. Néanmoins, cet utilitaire ayant malencontreusement été détruit suite à un reformatage, il n'est pas utilisé dans ce projet - à la place les mots de passe sont encryptés directement par la base de données.

Chapitre 3

Base de données

3.1 Choix de la base de données

Nous utilisons une base de données relationnelle pour persister nos données. Le choix s'est porté sur postgresql pour les raisons suivantes :

— Open-source, supporté par une grande communauté
— Licence MIT (contrairement à mySQL, qui est sous licence GPL)
— Familiarité des développeurs avec le système
— Peut être utilisé sur Linux, Unix, BSD et Windows servers.

3.1.1 Un système master/slave

Le schéma master/slave nous permet ici de répliquer les données de la base maître sur d'autres bases esclaves. Dans notre cas, la réplication n'a pas été mise en place avec pour principale raison les performances. La majorité des requêtes à effectuer sur la base de données sont des lectures - hors les lectures se feront sur la réplication effectuée sur ElasticSearch, ainsi notre base de données n'aura presque que des écritures à effectuer (et la charge devrait pouvoir être supportée par une seule instance). La base de données esclave est là afin de garantir l'intégrité et la sauvegarde (en tant que backup) des données dans le cas où la base de données maître tombe en panne. Nous pouvons ainsi utilser une base de données esclave afin de ne pas interrompre le service.

3.2 Mise en place du système master/slave

La première étape a été d'installer postgresql sur l'une des machines virtuelles. Nous avons ensuite dû installer un firewall (UFW) sur cette même machine virtuelle afin de pouvoir gérer les iptables, qui nous permettent de configurer le pare-feu de l'espace noyau. Les ports 22 (ssh) et 5432 (postgresql) devaient être rendu accessibles sur les machines (master comme slaves) afin de pouvoir communiquer avec elles (et qu'elles puissent communiquer entre elles).

La base de données master a été configurée de façon à autoriser les écritures et lectures, mais aussi d'accepter les requêtes provenant de la base de données esclave en ajoutant l'IP de la machine esclave dans les fichiers de configuration afin de l'autoriser à répliquer les données de la base maître.

La machine esclave a été configurée en suivant le même protocole, avec quelques modifications. La première modification est d'autoriser la machine esclave à seulement accepter les lectures (autrement il y aurait des complications pour reporter les écritures effectuées sur l'esclave au maître). Nous avons avons ensuite remplacé le répertoire principal de l'esclave par celui du master. La dernière étape a été d'exécuter la commande "pg_basebackup", qui autorise à l'utilisateur replicaUser de répliquer les données de la base maître via un trigger.

Avec le recul, une base de données mySQL aurait probablement simplifié la mise en place de la réplication, qui semble se faire de façon évidente contrairement à Postgresql qui demande un peu plus de configuration.

3.2.1 Administration de la base de données

Nous avons définis trois utilisateurs au sein de notre base de données.

- *replicaUser*, représentant la base de données esclave, ayant uniquement les droits de lecture.
- *pokemonUser*, représentant l'administrateur de la base de données, qui a les droits de lecture et d'écriture.
- *elasticUser*, représentant l'utilisateur d'ElasticSearch, possédant uniquement le droit de lecture.

3.2.2 Conception de la base de données

La conception de la base de données était l'étape clé du projet. Nous sommes partis dans l'idée d'avoir une base de données extensible et pouvant gérer plus de fonctionnalités que ce qui serait rendu en tant que version 0. Ci-dessous se trouve un diagramme UML correspondant au modèle que nous aurions aimé avoir :

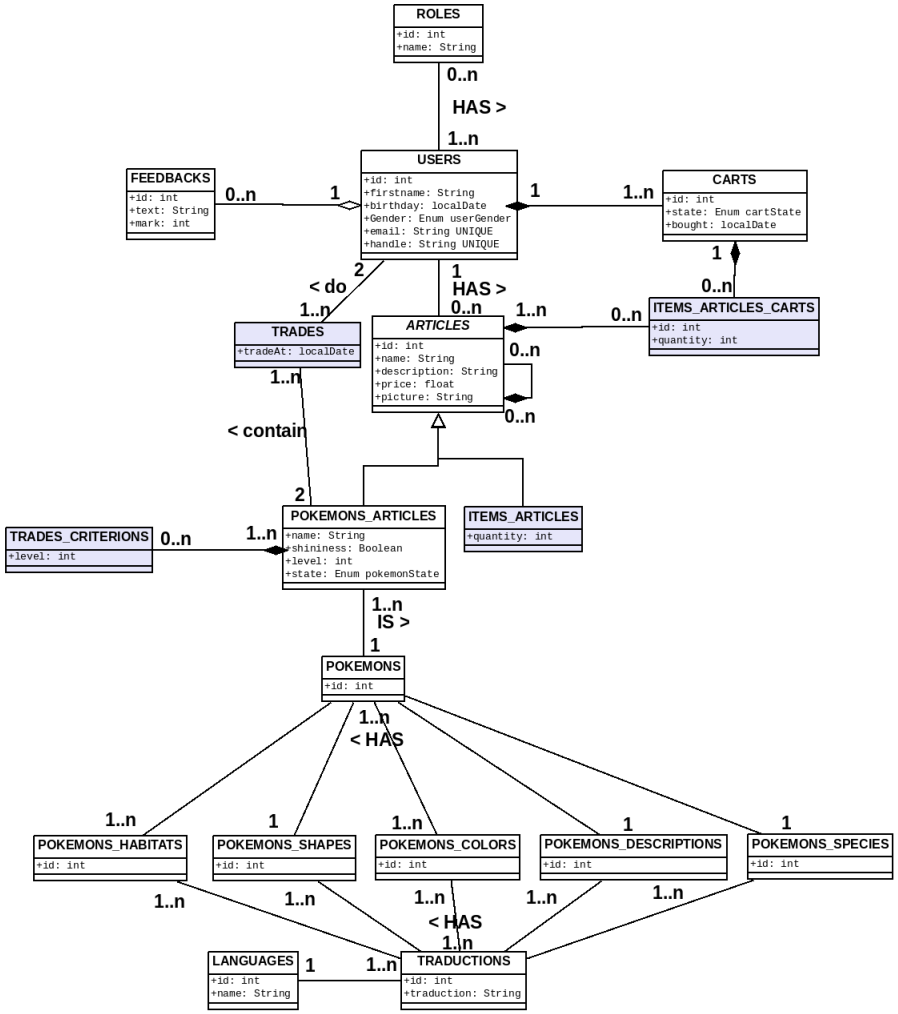


FIGURE 3.1 – Diagramme de classe

La base de données a beaucoup évolué avant la dernière semaine, sans pour autant être pafaite - on peut retrouver quelques problèmes de cohérence avec le diagramme ci-dessus (par exemple, la base de données n'est pas réellement en capacité de gérer les échanges suite à des oublis lors de l'implémentation). Quelques fautes de frappes se sont aussi glissées au sein de la base de données, mais elles n'ont pas été jugées importantes compte tenu du temps restant.

Avec le recul, il aurait peut-être été plus pertinent de partir sur une base de données la plus simple possible, afin de passer plus rapidement au développement.

Chapitre 4

Serveur

4.1 Choix du serveur

Deux options étaient considérées : Glassfish et Wildfly. Le choix s'est porté sur Wildfly pour les raisons suivantes :

- Open source avec une grande communauté autour (supportée par Red Hat)
- Meilleur temps de démarrage et une plus faible empreinte mémoire
- Configuration facilitée via les différents profils pré-existants
- Facilité de certaines opérations grâce à la CLI intégrée (redémarrer le serveur, ajouter des éléments de configuration, ...)

Le critère le plus important ayant été le côté plus user-friendly de Wildfly par rapport à Glassfish.

4.2 Mise en place du load balancer

Nous distinguons deux parties dans notre architecture : le load balancer (qui s'occupe de répartir la charge) et les workers (qui traitent les requêtes).

4.2.1 Configuration du load balancer

Depuis WildFly 10.1, un profil dédié au load-balancer est inclus dans les exemples. Nous utilisons donc le profil standalone-load-balancer.xml pour configurer notre serveur. La charge est répartie de façon égale. Dans notre cas, une requête sur deux partira vers le serveur 2.

4.2.2 Configuration des workers

Pour les workers, nous utilisons le fichier de configuration standalone.xml, modifié pour inclure notre base de données. La première idée était de communiquer avec la base de données uniquement grâce au fichier persistence.xml, qui ressemblait alors à :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence" version="2.0">

    <persistence-unit name="pokemonDB">
      <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
        <properties>
          <property name="hibernate.connection.url"
                    value="jdbc:postgresql://175.78.54.10:5432/pokemonDB"/>
          <property name="hibernate.connection.driver_class"
                    value="org.postgresql.Driver"/>
          <property name="hibernate.connection.username"
                    value="user"/>
          <property name="hibernate.connection.password"
                    value="password"/>

          <property name="hibernate.dialect"
                    value="org.hibernate.dialect.PostgreSQLDialect"/>
```

```
            <property name="hibernate.show_sql"
                      value="true"/>
            <property name="hibernate.format_sql"
                      value="true"/>

            <property name="hibernate.archive.autodetection"
                      value="class"/>
            <property name="hibernate.hbm2ddl.auto"
                      value="update"/>
        </properties>
    </persistence-unit>
</persistence>
```

Malgré plusieurs tentatives, impossible d'établir une connexion, le même message d'erreur apparaissant toujours (se résumant à des tables non trouvées). Il semblerait que malgré la prise en compte du fichier persistence.xml par le serveur wildfly lors du déploiement, celui-ci essayait toujours de se connecter à une base de données H2 - celle présente par défaut dans le fichier de configuration serveur standalone.xml. Est venue une seconde idée : utiliser ce fichier de configuration pour renseigner la base de données postgres.

Modification du fichier de configuration standalone.xml :

```
<subsystem xmlns="urn:jboss:domain:datasources:4.0">
  <datasources>
    <datasource jta="true" jdni-name="java:jboss/datasources/pokemonDB"
                pool-name="pokemonDB" enabled="true" use-ccm="true">
      <connection-url>jdbc:postgresql://175.78.54.10:5432/pokemonDB</connection-url>
      <driver-class>org.postgresql.Driver</driver-class>
      <driver>web.war_org.postgresql.Driver_42_1</driver>
      <security>
        <user-name>user</user-name>
        <password>password</password>
      </security>
      <pool>
        <min-pool-size>5</min-pool-size>
        <initial-pool-size>5</initial-pool-size>
        <max-pool-size>100</max-pool-size>
        <prefill>true</prefill>
      </pool>
    </datasource>
  </datasources>
</subsystem>
```

Modification du fichier persistence.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1"
    xmlns="http://xmlns.jcp.org/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
                        http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">

  <persistence-unit name="pokemonDB" transaction-type="JTA">
    <jta-data-source>java:jboss/datasources/pokemonDB</jta-data-source>
    <properties>
      <property name="hibernate.show_sql" value="true"/>
      <property name="hibernate.format_sql" value="true"/>
      <property name="hibernate.archive.autodetection" value="class"/>
      <property name="hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

Après ces modifications, le serveur fonctionne comme attendu.

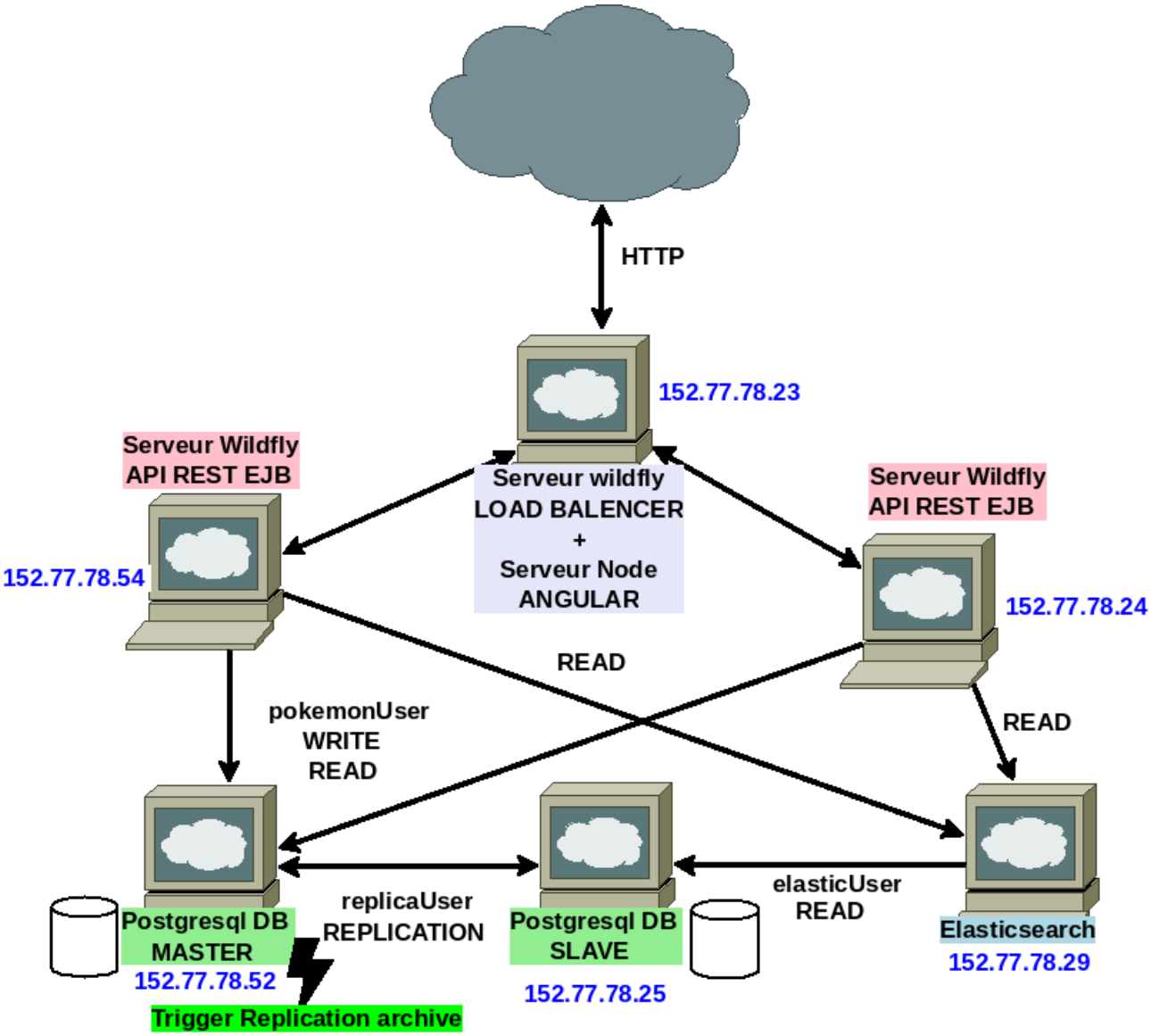
Note : nous avions dans un premier temps mis en place un système de cluster, et voulions partir sur un modèle load-balancer + clusters, mais les deux méthodes n'ont pas été combinées à temps.

4.3 Limites du load balancer

— Le nombre de requêtes gérées est limité par le load-balancer (celui-ci étant le point d'entrée). Toutes nos VMs ayant la même puissance, nous n'avons pas réellement la possibilité de gérer un plus grand nombre de requêtes en simultané - le load-balancer reste le goulot d'étranglement.

— Bien que nous ayons une meilleure résistance aux pannes (si un des workers tombe en panne, les autres peuvent prendre le relais), nous sommes toujours dépendants du fait de n'avoir qu'un seul load-balancer : si celui-ci tombe en panne, tout notre système sera en difficulté. Autrement dit, nous avons toujours un seul point de défaillance.

4.3.1 Principe schématique du sytème



Chapitre 5

Difficultés rencontrées au cours du projet

— Trop de temps passé sur les méthodes agiles, plus particulièrement pour définir les user stories. Nous avons voulu bien faire, mais compte tenu du temps accordé au projet, chercher à définir autant que possible les user stories n'était pas nécessaire.
— Trop de temps passé à la conception de la base de données, qui n'est toujours pas sans problèmes, mais qui bloquait relativement l'avancement des EJBs.
— Perte d'un week-end à travailler sur Spring Boot.
— La mise en place du load balancer, bien qu'évidente une fois effectuée, a pris plus de temps que prévu. La documentation manquait d'exemples simples à mettre en place.
— (résumé : trop peu de temps passer à coder, on a fait ça en une semaine sans rien connaître ...)

Chapitre 6

Annexe

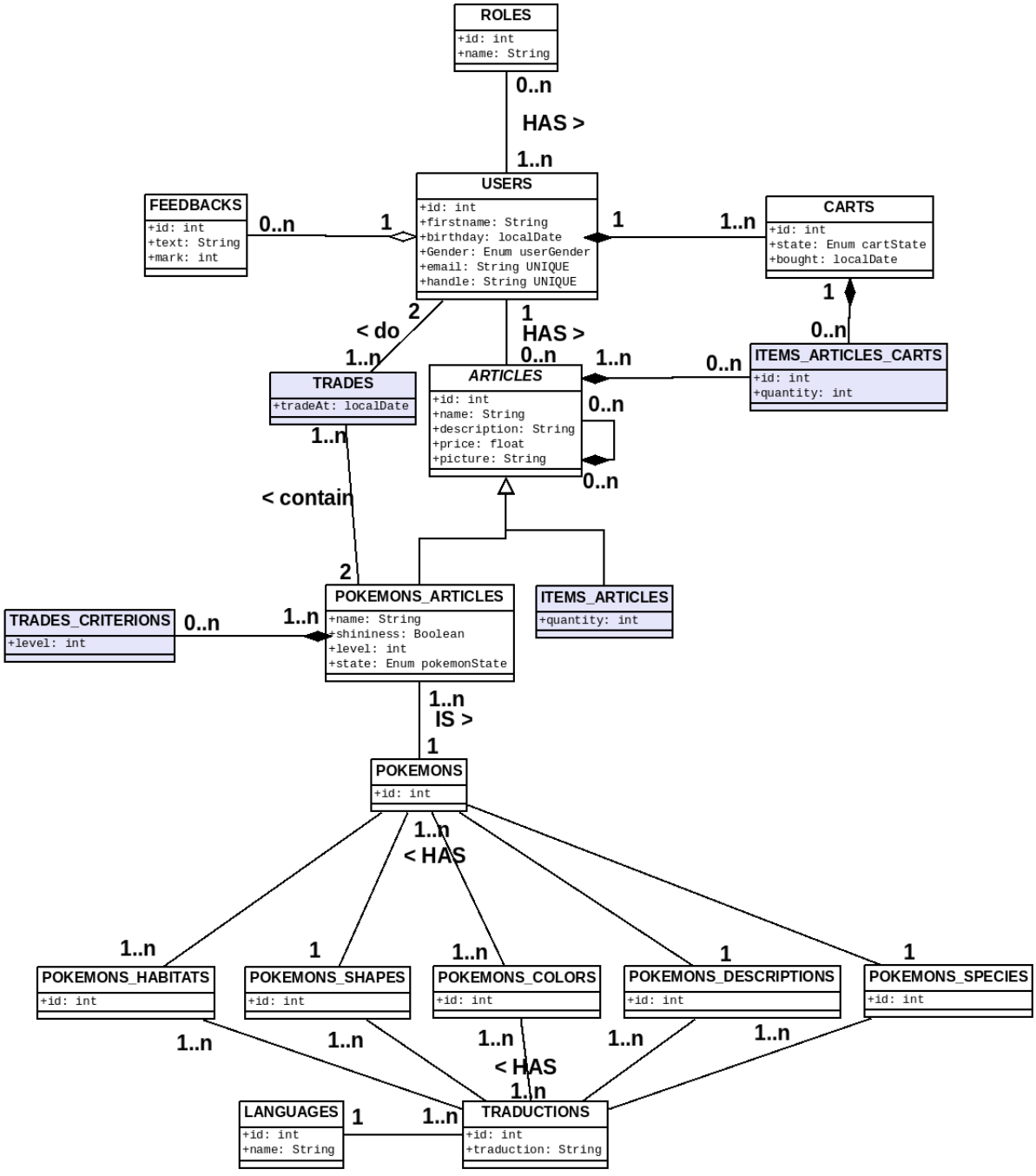


FIGURE 6.1 – Diagramme de classe