Université de Montréal

**Projet final : Jikiki**

Par  
Marc Laliberté, Kabore Élisée, Jérémy Camirand et Marc-André Piché

Bacc. en Informatique

Travail présenté à Maude Sabourin

Dans le cadre du cours IFT-2935

Bases de données

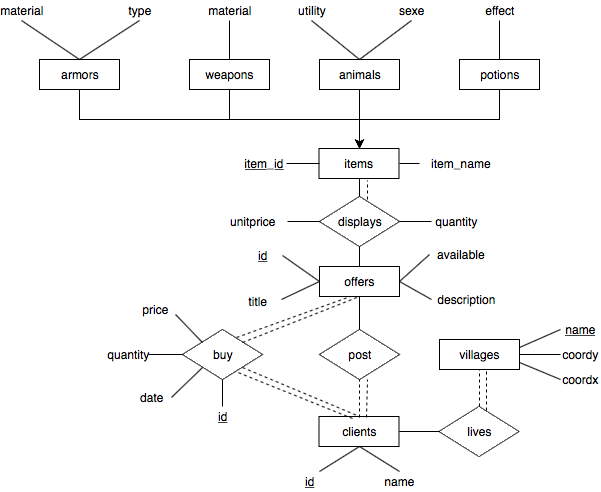
Mai 2018

**Liens utiles**

Lien vers le site du projet : http://108.61.78.227:8080/

Lien vers le repo Git : https://github.com/maPiche/Jikiki

**Représentation E-A de la base relationnelle**



Notre base de données représente une sorte d’hôtel des ventes d’un monde fantastique. L’idée est donc que des utilisateurs du site peuvent mettre en ligne une offre de vente pour leurs objets. Nous nous inspirons d’un hôtel de vente dans un jeu de rôle par exemple *World of Warcraft*.

Entités

Une offre a tout d’abord l’attribut « available » qui indique si l’offre est encore disponible. Chaque offre aura aussi une description, un titre et aussi sont id qui est sa clé primaire.

Un client a aussi son id qui est unique et aussi la clé primaire. Il a aussi son nom d’utilisateur.

Un village a un nom qui est sa clé primaire, une position géographique x et une position géographique y.

Un objet a un id unique qui est la clé primaire et aussi un nom. Nous voyons objet comme une classe abstraite, il sera donc impossible de créer un objet tout court, il faudra prendre une des classes qui hérite d’items, c’est-à-dire une des suivantes :

* Armure qui a pour attribut son type (casque, armure de jambe, etc.) et son matériau.
* Arme qui a pour attribut son matériau
* Animal qui a pour attribut son type (monture ou bétail) et son sexe.
* Potion qui a pour attribut son type (l’effet de la potion)

Relations

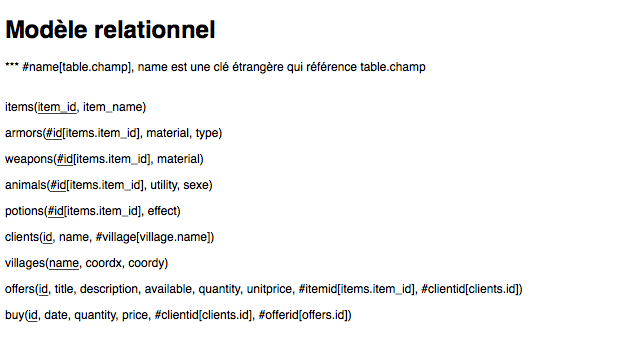
La relation displays fait le lien entre l’offre et l’objet vendu. Une offre est faite de sorte qu’un utilisateur peut vendre le même objet un nombre arbitraire de fois, c’est pourquoi displays a l’attribut quantity. Displays a aussi l’attribut unitprice symbolisant le prix unitaire de l’objet vendu.

La relation lives fait le lien entre un utilisateur et le village où il habite.

La relation post fait le lien entre un utilisateur et l’offre qu’il poste sur le site.

La relation buy fait le lien entre l’offre et un utilisateur qui a fait un achat. On a dans cette relation l’attribut price qui symbolise le prix acheté, quantity pour la quantité achetée, la date ainsi que l’id de l’achat.

**Le schéma relationnel de la base**

****

**Explications du code DDL**

Pour les clés primaires, nous avons décidé de mettre un ID unique pour chaque entité. Donc par exemple, chaque objet aura un « itemId » unique. Ceci fait en sorte que le choix de clé primaire est très simple.

Le seul cas de format standard à gérer dans notre base de données était la date de la vente. Nous avons décidé d’utiliser le types postgres Date pour représenter les dates.

Nous avons décidé de dupliquer l’attribut « material » qui se trouve dans l’entité armure et arme puisque, bien que les deux aient un matériau, la liste des matériaux possibles pour les armes n’est pas la même que celle pour les armures. On voulait aussi garder « Item » comme une classe abstraite, il serait donc impossible de créer un Item qui ne sera pas référencé dans un des sous-types de Item (armure, arme, animal ou potion).

La façon que nos id sont codés, nous ne pourrions pas avoir plus que 5000 items, 5000 users et 5000 offers, mais pour les besoins du projet nous pensions que c’était suffisant.

**Requêtes en SQL et explication du résultat attendu**

Tout d’abord, voici nos requêtes que nous jugeons plus complexe et qui font intervenir au moins 4 relations.

1.



Cette requête donne les objets les plus vendus, avec le nom des vendeurs, leur village et les coordonnées des villages.

2.



Cette requête donne, pour chaque type d’armure, l’objet le moins cher.

3.



Cette requête retourne l’arme de Mace la moins chère ainsi que le client de vente.

4.



Cette requête retourne toutes les offres et leurs vendeurs qui n’ont plus de stock.

5.

select title,item\_name,material,unitprice,village

from (select \* from armors, items,offers,clients

where armors.id=item\_id and item\_id=offers.itemid and offers.clientid=clients.id) as r1

Voici un exemple des requêtes qui sont utilisées dans la barre de navigation principale. Par exemple, celle-ci retourne le titre, le nom, le materiel, le prix et le village de ventes de toutes les armures. Nous utilisons ce gabarit pour les armes, potions, animaux.

Autres requêtes

1.****

Cette requête sort le nom, le type, le matériau, le prix unitaire et la quantité de toutes les armures fait de «Plate» qui sont en vente à Stormwind.

2.



Cette requête sort tous les objets en vente à Edoras.

3.



Cette requête donne toutes les potions de soins en vente à Theramore en sortant leur nom, leur effet, leur prix et la quantité en vente.

4.



Cette requête donne toutes les armures de corps en vente avec la ville où elles sont vendues.

5.



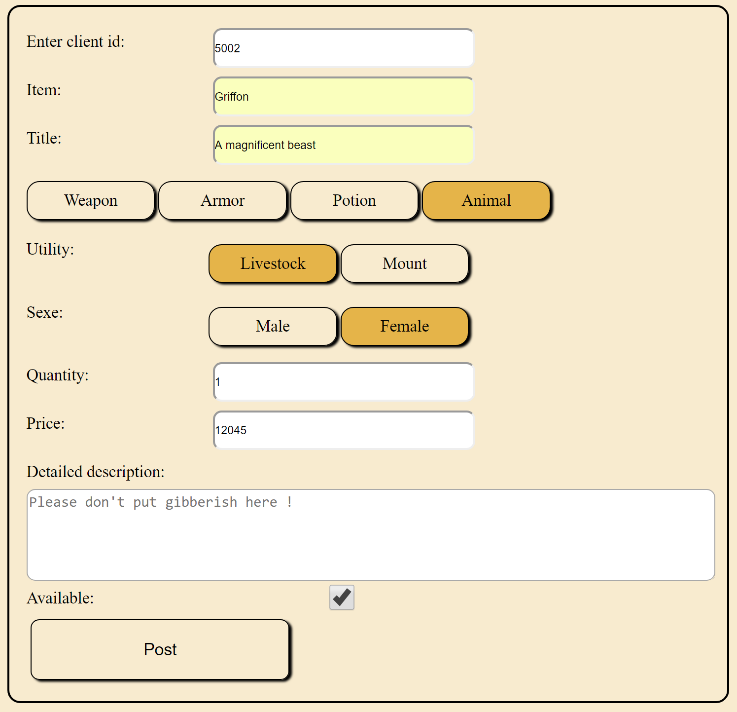
Cette requête retourne la distance entre 2 villages.

**Fonctionnement de l’application**

L’application est extrêmement simple à utiliser. Chaque requête préconfigurée a un bouton qui lui est désigné avec une étiquette très claire. Les requête les plus utiles se situent dans le menu du haut et les plus précises dans le menu de côté. De plus, si vous n’êtes pas satisfait des requêtes proposées, il est possible d’entrer sa propre requête.

Nous avons aussi ajouté un calculateur de distances entre les villages possibles.

Nous avons aussi opté d’utiliser un autre serveur pour la base de données que celui offert par l’école. Nous avons donc utilisé ElephantSQL. La raison principale est que c’était beaucoup plus simple de l’intégrer avec notre site web ensuite.

Vous souhaitez maintenant mettre un Griffon en vente, mais nous ne savez pas comment procédez. Voici donc la procédure à suivre :

La première étape est de vous referez à un administrateur du parchemin pour inscrire en tant que client, car sans un numéro client il vous sera impossible d’afficher une offre sur les babillards du royaume. Ensuite vous n’avez qu’à accéder à la page d’affichage grâce à « Post a new Item ».

Entrez votre no client - le nom de l’objet (gardez ça simple) - un titre pour votre affiche – choisissez le type d’objet à mettre en ventre remplissez paramètres ou choix correspondants – la quantité que vous avez en stock – le prix de vente – une bonne description (svp pas de texte n’importe quoi) – sa disponibilité. Révisez tous les champs pour être sûr de ce que vous voulez afficher.

Ici, nous avons limité la complexité à la simple fonctionalité. C’est à dire qu’il serait possible d’améliorer le porcessur de validité des champs. La validité est guidée uniquement par le formulaire html et il est bien sur possible de rentrer n’importe quoi dans la base de données. Chaque *input* est *required* à l’exception de eux reliés aux sous-types (armor -> utility) et leurs tailles correspond aux paramètres de base de données (ex : type VARCHAR(14)). Finnalement, l’autre aspect qu’il serait interessant d’avoir dans les prochaines versions c’est un peu de feedback lorsque que nous appuyons sur « Post », il est pour l’instant impossible de savoir si il y a eu succès ou echec ou par exemple de connaitre le *id* de notre nouvelle offre.

Au cœur du fonctionnement de la page il y a la requêtte SQL, qui est d’abord formattée dans le javascript.

WITH insert1 AS (

INSERT INTO items(item\_id, item\_name)

VALUES (DEFAULT, 'Mace')

ON CONFLICT DO NOTHING

RETURNING item\_id AS item\_id),

insert2 AS (

INSERT INTO weapons (id, material)

SELECT item\_id, 'Granite' FROM insert1

ON CONFLICT DO NOTHING)

INSERT INTO offers (title, itemid, clientid, quantity, available, unitprice, description)

SELECT 'Mace of tears', item\_id, 5024,1, true, 12716, 'Special request only' FROM insert1;

Pour insérer une offre directement dans la base de données, l’objet affiché doit être insérer dans 2 tables précédement ou ne rien insérer dans les 3 tables si pour une raison ou une autre il eu un conflit lors de la requêtte. On fait donc une une insertion en chaine dans les 3 tables, avec insert1 et insert2 qui revoient tous le item-id au suivant pour être utilisé dans sa requêtte. Les *id* sont des SERIAL ce qui permet d’inserer sans jamais specifier aucun id, c’est pour cette raison que nous avons DEFAULT dans la requette INSERT INTO items, cela nous permet autant de : 1. ne pas spécifier le item\_id, 2. recupérer ce nouvelle id pour les requêttes suivantes.

Du côté javascript, on procède en 4 étapes :

1. On récupère les informations du formulaire grâce à **$('form').serializeArray()**, qui est passé en requette. SerializeArray transfert ces informations sous la forme d’un objet {nom:valeur,…}, le nom du <*input>* et sa valeur. (dans script.js)
2. Du coté de bdserveur.js, on utilise en boucle **b[key].replace(/'/g,"''")**, qui traite tous les appostrophes pour le SQL, puisque c’est un charactère illegal. On transforme aussi la valuer du champs *available* en valeur booléene.
3. Ensuite avec un switch on génére la partie de la requette qui est différente pour les 4 possibiltés, c’est-à-dire une des 4 tables différentes dans lequelle on doit insérer.
4. On assemble la requette avec toute les valeurs passé en 1 et les 2 lignes en 3, puis en on l’envoie.

app.post('/postItem', **function** (req, res) {

**const** b = req.body;

**const** available = (b.available\_field.toString() === "on");

//difference entre les requettes selon le type

**function** switch\_lines(a){

**switch**(a){

**case** "weapons":

**return** ("INSERT INTO "+b.item\_subtype+" (id, material)\nSELECT item\_id, '"+b.we\_mat\_field+"' FROM insert1\n");

**case** "armors":

**return** "INSERT INTO "+b.item\_subtype+" (id, type, material)\nSELECT item\_id, '"+b.ar\_type\_field+"', '"+b.ar\_mat\_field+"' FROM insert1\n";

**case** "potions":

**return** "INSERT INTO "+b.item\_subtype+" (id, effect)\nSELECT item\_id, '"+b.potion\_effect\_field+"' FROM insert1\n";

**case** "animals":

**return** "INSERT INTO "+b.item\_subtype+" (id, utility, sexe)\nSELECT item\_id, '"+b.animal\_utility\_field+"', '"+b.animal\_sexe\_field+"' FROM insert1\n";

**default**:

**return** "input type error"; } }

//prend soin des ' dans chacune des entrées (' -> '')

**for** (**let** key **in** b) {

**if** (Object.prototype.hasOwnProperty.call(b,key)) {

b[key] = b[key].replace(/'/g,"''");}}

//requette d'insertion en chaine avec switch\_line selon le type d'item

lastRequest="WITH insert1 AS (\n"+

"INSERT INTO items(item\_id, item\_name)\n"+

"VALUES (DEFAULT, '"+b.item\_field+"')\n"+

"ON CONFLICT DO NOTHING\n"+

"RETURNING item\_id AS item\_id)\n"+

", insert2 AS (\n"+

switch\_lines(b.item\_subtype) +

"ON CONFLICT DO NOTHING)\n"+

"INSERT INTO offers (title, itemid, clientid, quantity, available, unitprice, description)\n"+

"SELECT '"+b.title\_field+"', item\_id, "+b.client\_field+", "+b.quantity\_field+", "+ available.toString()+", "+b.price\_field+", '"+b.description\_field+"' FROM insert1;";

pool.query(lastRequest, (err, response) => {

res.send(response)}) });

**Captures d’écran de l’application**



