Contexte

FEEDIIE est un site de rencontre jouant sur l'humour autour de la nourriture. Des applications de rencontre, il en existe beaucoup sur Internet, mais tous ne correspondent pas à ce que vous recherchez. Tinder, AdopteUnMec, Happn… Passage en revue des incontournable de la dragueconnectée pour trouver l'amour.

Nous souhaitons nous différencier et faire mieux que nos concurrents. Pourquoi quelqu'un de végétarien ne pourrait-elle pas indiquer que c'est pour elle un critère important de recherche ? Pourquoi ne pouvons-nous pas indiquer (ou indiquer qu'un seul) ce qu'on recherche comme type de relation ? Plutôt qu'une description de type texte, pourquoi ne pas, en plus de cela proposer une description plus détaillée en choisissant par exemple une personnalité ou un hobby ? Ou bien un grand sportif recherche sans doute quelqu'un avec qui partager ses séances. Donner une ambiance chaleureuse et conviviale au site est important pour nous.

Nous voulons que chaque utilisateur puisse s'identifier autour d'un ou plusieurs plats qui sont des représentations subjectives de personnalités et d'attentes. Notre slogan « **Trouve ton plat** » est une symbolique humoristique pour trouver sa moitié, une amie et tant d'autres choses qui nous correspondent.

FEEDIIE, venant du mot anglais « **FEED** » qui signifie s'alimenter, se nourrir et l'extension «**IIE»** en lien avec l'ENSIIE, le mot apporte une consonance agréable à entendre et facile à retenir.

L'équipe

Notre équipe de projet est composée de personnalités complémentaires provenant du même cursus :

- Léanna JI
- Valentin ELOY
- Théo SZATKOWSKI
- Mégane DJONGOUE
- Bastian PADIGLIONE

Description technique des pages

Connexion :

L'email doit respecter vérifier le regex [a-Z0-9]@[a-Z0-9].[a-Z]{2,5} (vrai pattern : ([a-zA-Z0-9_\-\.]+)@([a-zA-Z0-9_\-\.]+)\.([a-zA-Z]{2,5}))

Un appel en ajax est ensuite fait après la confirmation du formulaire. (Appel à connection dans UserRequest.php)

Nous récupérons les données des inputs ainsi que le mot de passe hashé de la base de données. Un appel au PasswordService est ensuite réalisé pour regarder si les mots de passe sont identiques.

Si les données ne match avec aucun utilisateur dans la base de données alors nous restons sur la page de connexion en affichant un message d'erreur de connexion. Sinon nous créons la session de connexion. Si l'utilisateur avait coché « Se rappeler de moi », nous récupérons le token de la base de données lié à l'utilisateur, et nous créons la session avec un temps de 30j. Sinon la session se détruit à la fermeture du navigateur.

Enfin connecté nous regardons si l'utilisateur possède une description ou une photo, si ce n'est pas le cas nous le redirigeons vers la page pour éditer cela. Sinon nous l'amenons sur la page de swipe

Il y a un bouton mot de passe oublié pré défini mais non fonctionnel car nous n'avons pas de serveur mail. Les fonctions existent mais pas codées.

Swipe :

Nous avons fait en sorte d'effectuer un double filtrage. Ainsi les préférences d'un utilisateur A doivent être respectées par un utilisateur B mais il faut également que les préférences de l'utilisateur B soient respectées par l'utilisateur A. En plus de ces catégories, nous vérifions également si l'utilisateur n'a pas déjà aimé ou pas aimé la personne. De plus, nous avons choisi que si un utilisateur B a dislike un utilisateur A, ce dernier va quand même pouvoir voir cet utilisateur B.

L'ensemble des animations ont été fait en Jquery. L'utilisateur peut sélectionner la tranche d'âge qu'il souhaite voir, entre 18 et 60 ans (on a prévu large). Cela se fait avec deux inputs range, une condition Jquery empêche l'inversion des bornes.

Page de profil :

La page de profil d'un utilisateur peut être accédée via l'url **/profile/[uniqID_user]**

Si la page de profil correspond à celui de l'utilisateur connecté, un bouton apparaît pour pouvoir effectuer des modifications sur son profil. Cela emmène l'utilisateur sur /profile/edit

Cette page regroupe l'ensemble des informations de l'utilisateur.

Etant donnée que nous avons besoin de certaines de ces informations sur 2 parties (La partie « voir plus » dans les swipe et sur la page de profile). Nous avons décidé de faire une partie à partie nommé UserDetails. Ce dernier sera une classe qui peut être appelé n'importe où tant qu'on include le fichier. Ainsi nous avons juste à instancier la classe en lui envoyant les données pour que les informations soient affichées de façon responsive.

Même principe pour les photos des utilisateurs, nous avons une classe UserPhoto

Chat :

La discussion instantanée est composée de deux parties. Sur la partie gauche se trouve la liste des matchs de l'utilisateur courant. Ceux-ci sont ordonnés par ordre de date de message. Les utilisateurs qui ont envoyés ou reçus les messages les plus récents sont placés en haut. Si aucun message n'a été envoyé ou reçu alors c'est la date du match qui compte. Le nombre de messages non lus envoyés par le contact est également affiché.

Sur la droite se trouve la discussion entre l'utilisateur courant et le contact sélectionné. Par défaut, seuls les 50 messages les plus récents sont chargés. Pour charger les messages précédents il faut remonter en haut de la discussion (ils seront chargés 50 par 50).

Chaque seconde, 3 requêtes ajax sont exécutées :

- Recherche des messages non lus de la conversation ouverte. S'il y a des messages non lus, ceci est marqué à lu puis retournés par la requête. Ils sont ensuite mis à la suite de la conversation.
- Recherche des matchs de l'utilisateur courant. Cela permet de mettre à jour l'état des notifications ainsi que l'ordre. L'ordre des matchs n'est mis à jour que si l'ordre courant est différent de celui de la réponse de la requête.
- Récupération du nombre total de messages non lus. Cela sera indiqué dans la notification du header.

Lorsque l'utilisateur courant envoie un message, la requête lui renvoie les éventuels messages non lus qui seraient apparu entre le dernier rafraîchissement et l'envoie du message, afin de garder l'ordre des messages en direct. Dès que la requête renvoie que l'insertion c'est bien passé, une nouvelle requête est effectuée pour mettre à jour l'ordre des matchs (l'utilisateur venant d'envoyé un message l'ordre peut avoir été modifié).

Upload d'une photo (administrateur):

Sur la page d'administrateur, vous avez la possibilité d'ajouter une personnalité/ plat et relation. Cependant, cela nécessite d'upload une photo et de valider la création.
Ainsi, l'administrateur peut télécharger une photo mais nous n'avons pas la possibilité de savoir si elle sera bien utilisée au final. Nous avons préféré éviter de créer un dossier pour les photos en attente et de déplacer la photo hors du dossier temporaire.
Nous avons alors récupéré la photo téléchargée du dossier temporaire avant qu'elle soit supprimée et de le transcrire en base64 pour pouvoir l'afficher lors de la création. Si l'utilisateur confirme la création, nous récupérons cette base64 et nous l'envoyons via une requête AJAX. Nous retranscrivons ensuite cette base64 en fichier réel et le plaçons au bon endroit. Pour finir, nous affichons l'image dont la source est le chemin de la photo.

Fonctionnement et architecture du site

L'architecture du projet suit un modèle MVC et nous nous sommes inspirés des API REST pour concevoir les requêtes AJAX.

Dans le dossier **data**, vous retrouverez tous les scripts SQL liés à la création et l'insertion des données. La création des tables a été généré par JMerise.

Dans le dossier **public**, nous avons placé toutes les images du site, scripts et fichiers CSS ainsi que 2 fichiers : index.php et ajax.php. Le premier fichier est le point d'entrée du site et servira de routage pour afficher la page correspondante à l'URL demandé. Le second fichier est le point d'entrée pour les requêtes AJAX. Selon la route, il effectuera différentes actions et renverra les données en format JSON. Nous détaillerons sur leurs fonctionnements plus tard.

Dans le dossier **src**, nous allons retrouver les dossiers correspondant à l'architecture MVC ainsi qu'un répertoire pour gérer les requêtes AJAX et un répertoire qui regroupe tous les services. Nous avons alors 5 dossiers :

-**Model** : Contient tous les fichiers qui gèrent les requêtes SQL

-**View** : Contient tous les fichiers qui génèrent la page à afficher

-**Controller** : Contient tous les fichiers qui gèrent la communication entre le modèle et la vue selon la demande utilisateur.

-**Request** : Contient tous les fichiers qui traitent les requêtes AJAX reçues et appellent le modèle si besoin.

-**Service** : Contiens tous les fichiers qui servent à faciliter certaines tâches comme la gestion des photos, générer un objet au format JSON, créer un PDO, récupérer l'utilisateur actuellement connecté etc…

Details dossier Image :

Dans ce dossier, il existe 5 dossiers dont 4 qui sont des images du site (par exemple : le logo, les images de plats, de relation etc…).

Le dernier nommé « UserUpload » contient toutes les photos des utilisateurs. Ce dossier étant public, nous avons voulu par sécurité, complexifier le chemin d'accès à ces images. Ainsi, dans le dossier UserUpload, chaque utilisateur aura son propre dossier de photos nommé par son identifiant unique et chaque photo aura un nom généré aléatoirement.

Les Services :

Les services n'ont que des fonctions static afin que l'on puisse les appeler sans avoir besoin de d'instancier un objet de la classe correspondante. Il existe plusieurs types de service :

-**AuthService** : Ce service entre en scène une fois que l'utilisateur se connecte. Il va permettre de récupérer toutes les informations de l'utilisateur connecté.

-**DBConnection :** Ce service permettra de simplifier toutes les requêtes SQL car il génère le PDO donnant ainsi accès à la base de données.

-**RequestService :** Permet de générer un objet au format JSON et de le renvoyer au client suite à un appel AJAX.

-**EmailService** et **DateService**: Contiennent une fonction qui permet de vérifier une chaine de caractères afin de savoir si elle respecte bien le format d'un mail et d'une date respectivement.

-**PasswordService** : Contient toutes les règles qu'un mot de passe doit respecter, il permet également de générer un texte pour montrer les règles à suivre et de vérifier la conformité d'un mot de passe.

-**PhotoService** : Permet de réaliser toutes les opérations que nous avons besoin pour manipuler une photo. La classe peut ainsi permettre de supprimer une photo, de gérer la base64, de nommer une photo aléatoirement etc…

La partie View :

La classe ViewModel va permettre d'afficher la page souhaitée grâce à sa fonction render(). Ainsi render() contiendra tout le code HTML de la page. De plus il va appeler tous les scripts et les styles CSS dont cette page a besoin. Lors de l'instanciation de l'objet, on peut passer jusqu'à 3 paramètres :

-Page : nom de la page à afficher. **Ce paramètre est obligatoire.**

-Data : Données qu'on souhaite afficher, par défaut il est initialisé à NULL

-Title : Permet de changer le nom de l'onglet, par défaut il est initialisé à « Feediie »

Le nommage des fichiers est très important. Par exemple, si on veut accéder à l'URL « feediie.fr/register », il faut impérativement nommer les fichiers comme cela :

- Register.php
- Register.css
- Register.js

Ainsi la fonction render() de ViewModel se résume à :

<html>

<head>

Appel du CSS Bootstrap + meta + les icones de font awesome

Inclusion du fichier Style nommé par [nom_de_la_page].css

</head>

<body>

Inclusion du header.php si l'utilisateur est connecté

Inclusion de la page à afficher nommé par [nom_de_la_page].php

</body>

Appel des scripts Jquery + Popper.js + Bootstrap JS + Jquery Google

Inclusion du fichier Script nommé par [nom_de_la_page].js

</html>

Cela permet alors d'afficher dynamiquement la page en fonction des paramètres envoyés

Les Controllers :

Chaque contrôleur va hériter la classe Controller. Cette classe est abstraite et contient la fonction abstraite execute. Ce qui permet à chaque controller de posséder cette methode.

Après avoir été instancié par l'index.php, le controller va exécuter la fonction execute et effectuera les actions nécessaires pour retourner la page correspondante. Un controller va forcément retourner une classe ViewModel

Les Models :

Chaque Model va hériter la classe DBConnection afin de récupérer le PDO généré et effectuer les requêtes SQL grâce à ce PDO. Dans le but d'éviter les injections SQL, nous utilisons les méthodes prepare() et execute() du PDO pour exécuter les requêtes SQL.

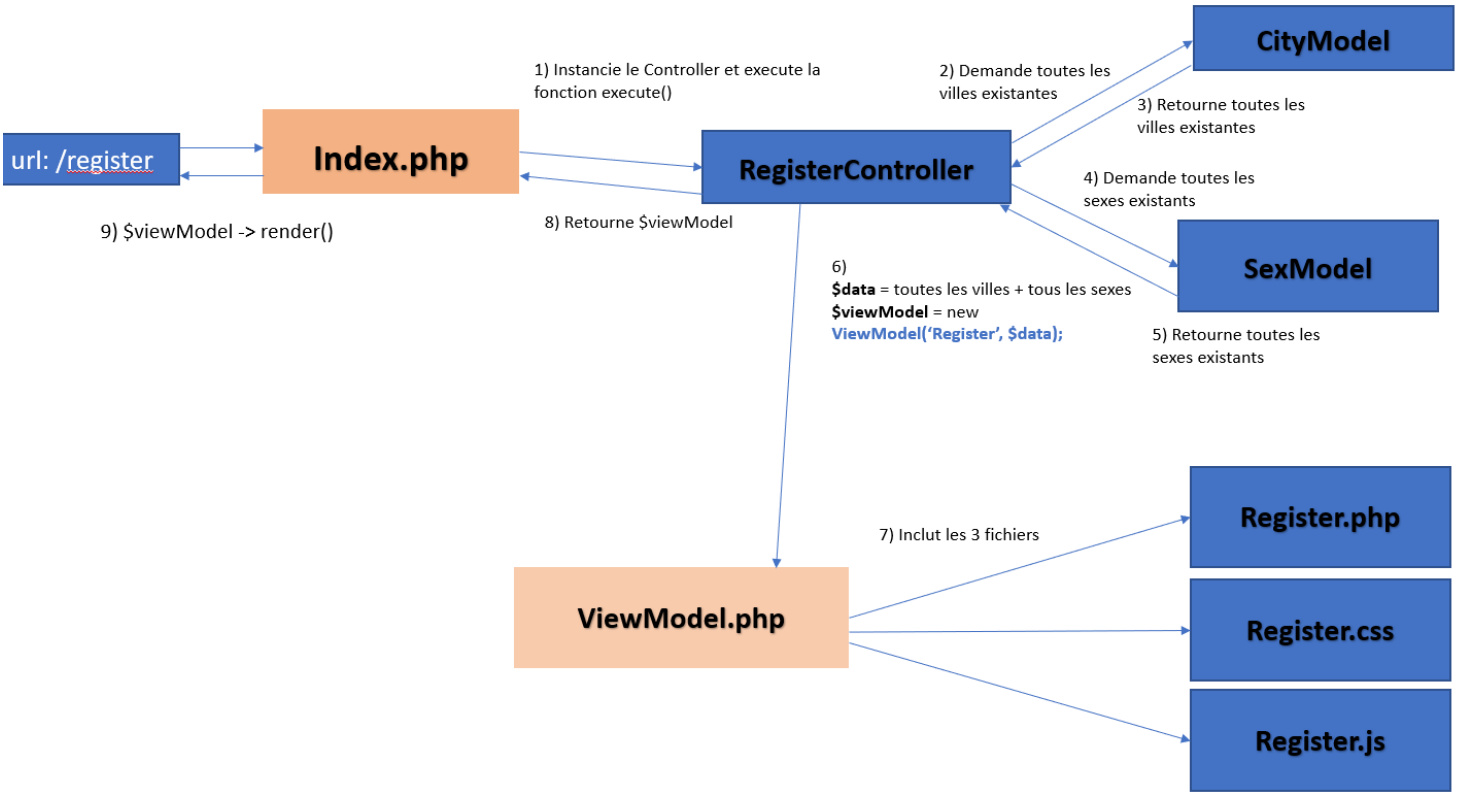
Les Request :

Chaque Request va hériter la classe RequestService afin de récupérer les fonctions de cette classe pour simplifier la création de l'objet JSON à retourner.

Après avoir été instancié par l'ajax.php, le Request va exécuter la fonction execute et effectuera les actions nécessaires pour retourner l'objet JSON demandé. Un Request va forcément retourner un objet JSON

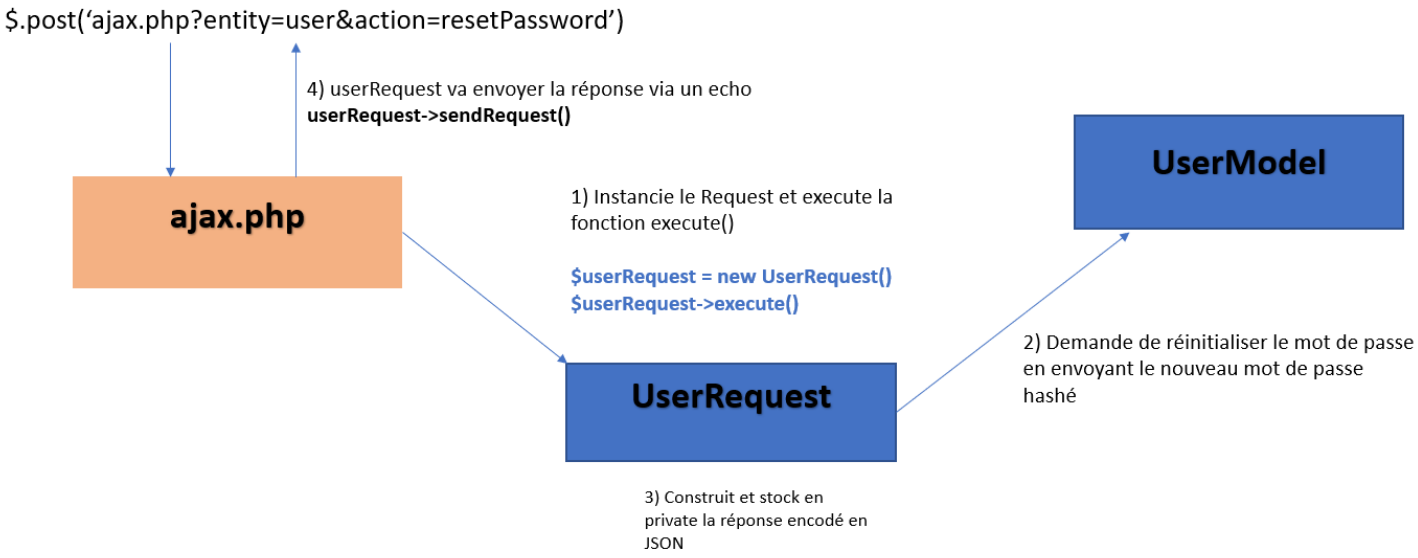
Déroulement de l'affichage d'une page

Prenons l'exemple de si l'url de la page est équivalent à /register

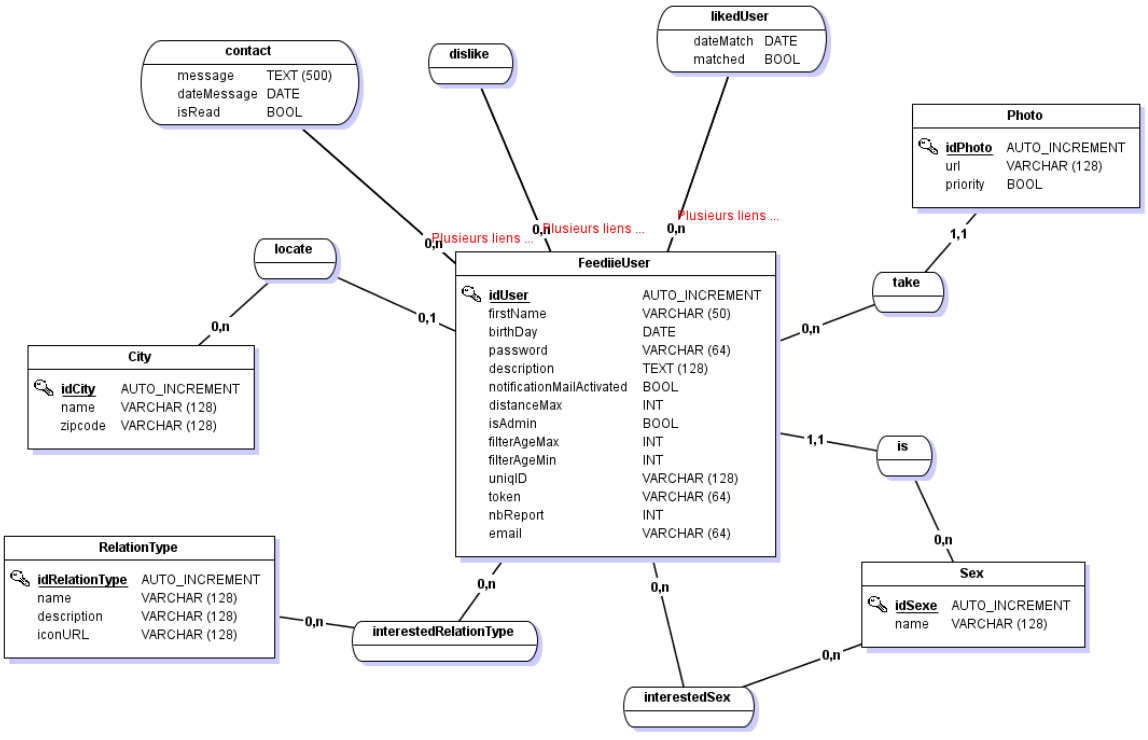


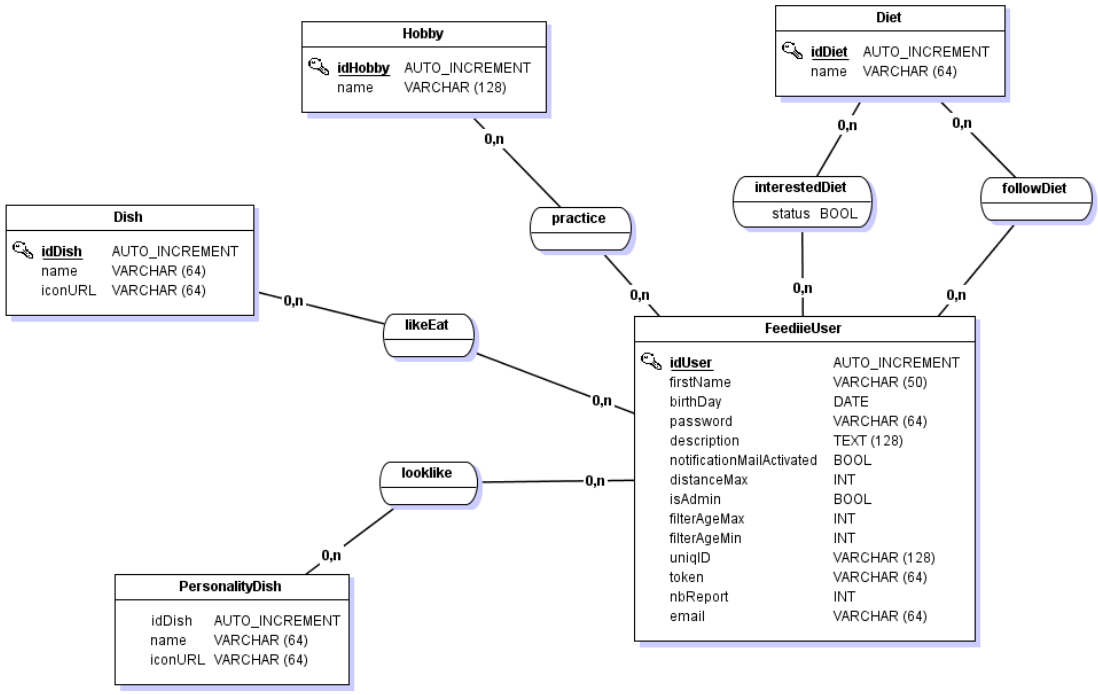
Déroulement d'une requête AJAX

Prenons l'exemple d'une requête AJAX qui demande la réinitialisation d'un mot de passe



Notre base de données




Hobby
idHobby AUTO_INCREMENT
name VARCHAR (128)
Diet
idDiet AUTO_INCREMENT
name VARCHAR (64)
0,n
0,n
0,n
interestedDiet
status BOOL
followDiet
practice
Dish
idDish AUTO_INCREMENT
name VARCHAR (64)
iconURL VARCHAR (64)
0,n
0,n
0,n
0,n
likeEat
FeediieUser
idUser AUTO_INCREMENT
firstName VARCHAR (50)
birthDay DATE
password VARCHAR (64)
description TEXT (128)
notificationMailActivated BOOL
distanceMax INT
isAdmin BOOL
filterAgeMax INT
filterAgeMin INT
uniqID VARCHAR (128)
token VARCHAR (64)
nbReport INT
email VARCHAR (64)
0,n
looklike
0,n
0,n
PersonalityDish
idDish AUTO_INCREMENT
name VARCHAR (64)
iconURL VARCHAR (64)