Utilisateur (

#login: string, nom: string, prenom: string, mdp: string, date_nais: Date, ~~droits[]:int(4)~~

) with mdp NOT NULL, (nom, prenom) KEY (je suppose. Pas obligatoire, étant donné que le login est unique, deux utilisateurs peuvent avoir le même prénom et le même nom)

Droit(

#id : int (ajout d'id pour faciliter les ajouts en BD),

intitulé : string

)with intitule UNIQUE

Droit_Utilisateur(

#login_utilisateur => Utilisateur(login), #id_droit=> Droit(id)

)

/*L'heritage etant complet, je suggere de le traduire par la classe mere utilisateur. Puisqu'on peut considerer que les

classes filles ne sont pas mutuellement exclusifs mais plutôt hierarchiques, voici ce que je propose :

1) on est administrateur, ou on ne l'est pas : premier digit 1 ou 0;

2) idem pour moderateur,

3) idem pour redacteur,

4) idem pour editeur,

5) on est lecteur par defaut (si on est enregistre dans la BDD, c'est bien qu'on a un compte).

Ce qui nous donne un code en binaire comme par exemple 1000 (seulement admin), 11 (redac + edit) ou encore 101 (mod +

edit).

Je suis d'accord sur le principe, mais ta méthode relève plus de la réalisation que de la représentation. Plus simplement pour le MLD on peut représenter les droits par une classe Droit et une associations N : M entre Droit et Utilisateur. Pour cela je suppose qu'un utilisateur peut cumuler

les droits. Un utilisateur peut être par exemple Lecteur Editeur et Admin. Le statut Lecteur étant le statut de base donnée à toutes les nouvelles entrées Utilisateur en BD.

Ainsi la base de données sera composée d'une table Droit, d'une table Utilisateur et d'une table qui fait la liaison entre les deux. Cette méthode permet de rajouter d'autres droits au besoin et d'ajouter des droits aux utilisateurs sans modifier les entrées dans la table Utilisateur.

*/

Pour la suite on utilise Article comme centre de notre modélisation

Article (

#id: int, honneur: bool = false, archive: bool = false, date_crea: Date, Créateur => Utilisateur(login), Publicateur =>Utilisateur(login)

) with date_crea,créateur NOT NULL

/*Alors, peut-etre que je reve et que je suis pas du tout concentre, mais il me semble qu'ils ont MAJ le sujet entre

temps... Donc apparemment les lecteurs peuvent aussi noter les articles, d'ou le rajout d'une table Note...

A part ca, ils ont encore modifie les conneries de statuts, alors j'en ai marre la classe enum de statuts redac ne

fonctionne plus, enfin bon faut que vous me disiez ce que vous en pensez...

J'ai bien cette impression aussi, pour palier à ça je propose de faire la même chose que pour les droits, c a d une table statut rédacteur avec tous les différents statut et une table statut_redacteur_article qui référence un statut, un article et un utilisateur. On ajoute un booléen dans cette table pour notifier le statut actif de l'article. On fera attention dans la réalisation à ne jamais avoir deux statuts actifs sur le même article sauf pour les statuts rédacteur ou plusieurs statuts peuvent être actif(TRIGGER)

*/

Comme ceci :

Statut rédacteur (

#id : int , intitulé : string

)with intitulé UNIQUE

Statut_rédacteur_article(

#id : int(ajout d'id car aucune clé primaire possible sur attributs ou tuples), login_rédac => Utilisateur(login), Statut => Statut rédacteur(id), Article => Article(id), Statut_actif : Bool

)

Même chose pour statut éditeur, en faisant bien attention au moment de la réalisation de restreindre les attributs modifiables selon les statuts (préconisations, justifications) :

Statut_éditeur(

#id : int, intitulé : string

)with intitulé NOT NULL UNIQUE

Statut_éditeur_article(

#id : int, login_édit=>Utilisateur(login), Statut => Statut_éditeur(id),Article=>Article(#id), Statut_actif : Bool, préconisations : string = NULL, justifications : string = NULL

)

Note (

#login=>Utilisateur, #id_art=>Article, note: int)

) with note <= 5

/*Concernant la normalisation, si l'on garde tags en multivalue (par exemple en tant que chaîne formatee de type <tag1,

tag2,..., tagn> (avec ',' comme separateur) on en sera pas en 1NF, donc autant creer une table Tags :

Tags (

#id=>Article, #mot: string

)

/* Heritage Complet et exclusif, modélisation par les classes filles */

Bloc_txt (

#id_art=>Article, #titre: string, texte: string

) with texte NOT NULL

Bloc_img (

#id_art=>Article, #titre: string, chemin: string

) with chemin NOT NULL

Contrainte : PROJ(Article, id) IN PROJ(UNION(PROJ(Bloc_txt, id_art), PROJ(Bloc_img, id_art)), id_art)

Rubrique (

#id: int, id_art=>Article, titre: string, Rubrique_mère => Rubrique(id)

) with (id_art, titre) KEY and NOT NULL

/*Je ne saisis pas l'attribut contenu de Rubrique, sachant qu'il est absent du sujet actuel, je suggere donc qu'on le

retire. Pour traduire la relation reflexive et recursive de possession (rubriques, sous-rubriques, etc.), nouvelle table.

Je propose du coup de creer une cle artificielle pour eviter de faire plein de cles etrangeres dans cette autre table.*/

~~Rubrique_tree (~~

~~#mere=>Rubrique, #fille=>Rubrique~~

~~) with fille != mere~~

A mon sens inutile, il suffit juste de garder la rubrique mère d'une rubrique car une sous-rubrique est associée à une seule rubrique mère.

Commentaire (

#id: int, login=>Utilisateur, id_art=>Article, date_crea: Date, texte: string, ~~statut: int(1)~~, Archivé : bool, masqué :bool,Masqué_par => Utilisateur(login),Archivé par=>Utilisateur(login) , (possibilité d'ajout de champ date tel que date_archivation : Date etc …)

) with statut<4

/*On peut creer une cle artificielle pour Commentaire, plus simple. Je suggere qu'on regroupe archive, masque, en avant

et le statut par defaut (visible) en un seul attribut, statut de type int qui est d'autant plus haut que le commentaire

est visible : 0 archive/supprime, 1 masque, 2 visible et 3 exergue.

On a besoin de savoir qui l'a masqué, qui la supprimé. Lors de l'implémentation il faudra bien faire attention de faire coïncider les boolean avec la complétion des bons champs. Par défaut les champs Masqué_par et archivé_par seront NULL

*/

/*Je n'avais pas vu ça avant d'écrire le reste. Du coup cette modélisation doit fonctionner.

Cependant je pense qu'il est plus simple de recenser ces actions directement dans les tables concernées comme je l'ai fait. Mais c'est mon avis.

Il est vrai que la méthode avec la table historique ne limite pas le nombre d'actions différentes recensables, ce qui peut être utile.

Historique (

#id: int, date: Date, login=>Utilisateur, rang: ('redac', 'mod', 'edit'), type: ('art', 'com'), id_obj: int, action:

string

) with (date, login, rang, type, id_obj, action) NOT NULL, PROJ(Historique, id_obj) IN PROJ(UNION(PROJ(Article, id),

PROJ(Commentaire, id)), id)

/*Meme principe pour action, on peut s'amuser a recenser toutes les actions possibles...*/