Part 05 - Aux élements multiples

A ce stade, vous devriez avoir le CRUD de votre perosnnage complètement fonctionnel. Il est temps de corser un peu le jeu pour le rendre plus réaliste !

Nous avons, durant la partie 2, défini des attributs à nos personnages comme UnitClass ou bien Element. Actuellement, nous stockons l'information sous forme de texte.

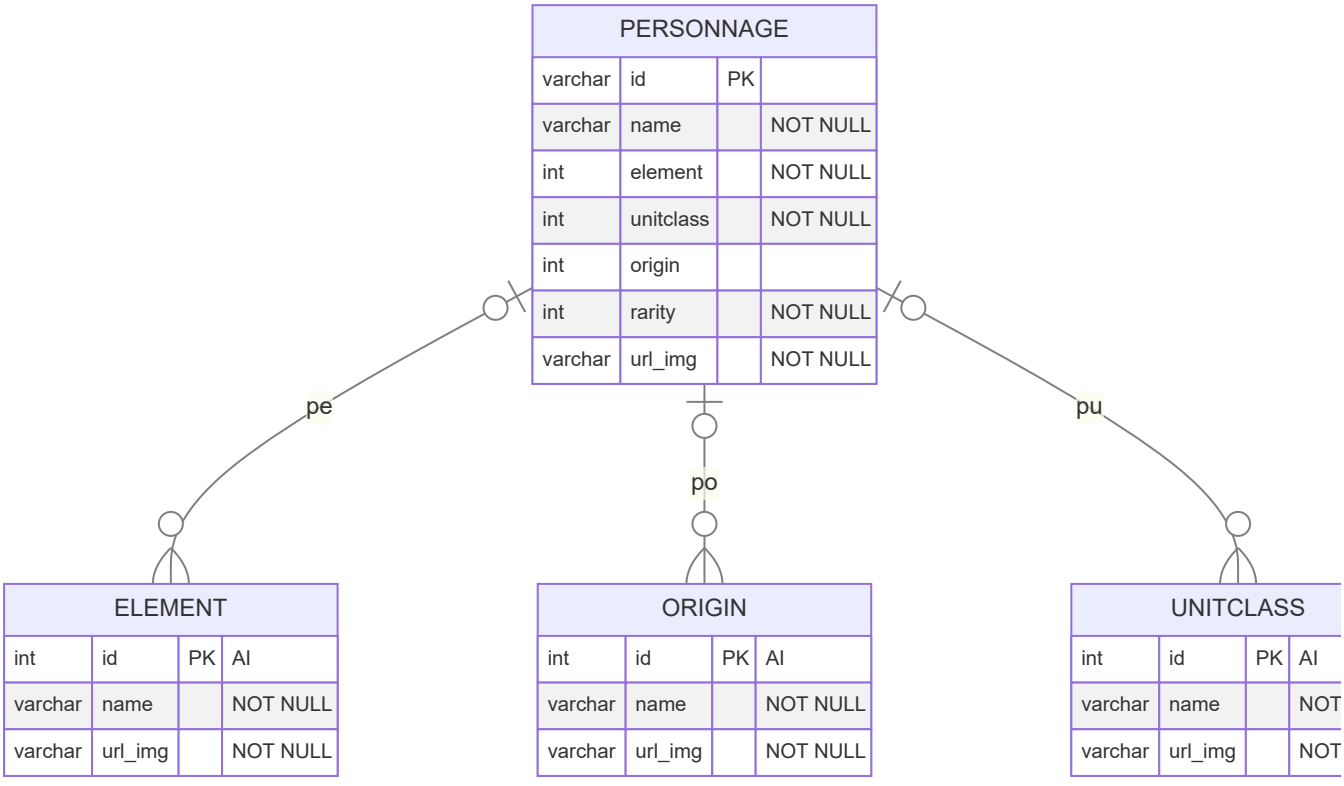
Hors, l'information est un peu plus complexe. Elle est souvent représenter par une icone et un texte. Il faudrait alors stocker ces infos dans des tables différentes et se servir des relations de note BD. Comme la structure est la même 2 choix s'offrent à nous :

- Avoir une table qui gère les 3 types d'attributs (1 seule table, mais besoin de trigger pour garder les contraintes d'intégrités)
- Avoir une table par type d'attribut (donc 3 table -> unitclass, element, origin)

Dans un soucis de facilité et de temps, nous allons aborder la 2ème option.

1 - Une origine en base de données

1.1 : Nous allons modifier notre BD pour nous aligner sur ce nouveau schéma :



Si vous avez des personnages en BD, je vous recommande de réinitialiser vos données. Vous pouvez faire la liaison entre les 2 tables via SQL ou votre logiciel pour visualiser votre

BD.

Avec la nouvelle contrainte entre PERSONNAGE et les autres tables, n'oubliez pas le ON DELETE/UPDATE CASCADE si vous voulez toujours pouvoir supprimer.

```
A ce stade, vous devriez avoir tout cassé, c'est normal, pas de panique.
C'est pour ca que je vous recommande de ne plus avoir de donnée en base

Nous verrons pour corriger cela dans la 2ème partie
```

1.2 : Vous avez pu remarquer que la structure est identique entre élément, origine ou unitclass. Dans le sujet, un seul sera traité mais il faut le faire pour les 3 attributs.

Pour commencer, attaquons notre modèle. Il ressemblera beaucoup à *Personnage* mais avec moins d'attribut.

```
La présence de l'hydratation est requise. Attention pour url_img ;)
```

1.3 : Maintenant, il faut mettre en œuvre la fonction qui permet de créer une *Origin* via le formulaire (sur la page `add-attribut`). Il faut alors :

1. Mettre à jour l'action du formulaire
2. Gérer l'action dans le routeur
3. Créer et appeler la bonne fonction du controller
4. Cette dernière crée l'attribut et affiche l'index avec un message

Vous devriez avoir l'habitude maintenant. A vous de jouer pour faire fonctionner l'ajout d'une origine.
L'application n'est pas forcément prévus pour modifier et supprimer une origine. Mais vous êtes libre de l'implémenter

Il ne vous reste plus qu'à reproduire ce schéma pour *UnitClass* et *Element*.

```
Pour éviter d'avoir trois formulaire. Vous pouvez utiliser un champs de type
select pour votre formulaire pour savoir quel type d'attribut nous ajoutons
```

Normalement vous devriez avoir vos 3 attributs ! Maitenant, il faut les associer à notre Personnage

```
Il est techniquement possible de n'avoir qu'un DAO pour gérer les 3. Mais je
ne vous le recommande pas. C'est une difficulté supplémentaire pour garantir
de respecter les principes SOLID
```

2 - Liaison

2.1 : Maintenant que nous avons des attributs. Il faut les lier à un personnage. Pour cela, quelques modifications sont nécessaires.

Pour commencer, il faut remplacer le type des variables qui stock nos attributs

N'oubliez pas de modifier le typage dans les getters/setters

```
Pour vous faciliter la vie, le null safe operator ?-> au lieu de -> permet d'appeler une fonction seulement si l'objet n'est pas null
```

A ce stade, nous avons un soucis. Pour hydrater une unité, actuellement, le *setter* attend un array. Hors la BD ne nous retourne plus que id d'attributs.

Il faut donc ajouter des requêtes en plus pour récupérer les attributs du personnages dans notre OriginDAO. C'est ici qu'avoir une couche Service prend tout son sens.

Elle a pour objectif de coordonner les DAO pour pouvoir créer un Personnage complet.

Si ce n'est pas déjà fait, créez un `PersonnageService`. Celui-ci aura les fonction de gestion (CRUD) d'un personnage.

Mais en plus de créer l'objet de la classe Personnage, les fonctions devront appeler les DAOs pour lui lier les différents attributs

```
function getAllPerso() : array {
      //Récupère tous les personnages
      //Pour chaque personnage
        // Appels les DAO des attributs pour récupérer les données
        // Instancie des Objets type Element,Origin,...
        // Instancie un Personnage avec les données complexes
        // Ajoute ce personnage à une liste de personnage
      //Retourne la liste de personnage
  }
```

Une fois que vous avez fait cela, il faudra modifier tous les endroits où vous créez vos unités, pour utiliser votre service

```
Petite invitation à la réflexion de qualité de code. Si vous avez eu le réflexe de faire un pattern comme la fabrique, alors cette modification devrait être simple et très localisé o/
```

Et pour finir, il faudra certainement modifier votre affichage sur la page d'accueil, pour gérer le getter des attributs comme un objet et non une string.

Si tout est bon, vous devriez pouvoir ajouter à la main dans la BD des unités, et les lier avec des attributs. La page d'accueil devrait fonctionner.

2.2 : Attaquons-nous à l'ajout d'un personnage. Une modification dans le formulaire s'impose. Avez-vous anticipé laquelle ?
...

Récupérons la liste des attributs depuis nos DAO, puis améliorons le formulaire pour avoir l'*id* en valeur et le nom en visuel.

Pour rappel, il nous faudra un select par attribut

Voici un exemple :

```
<option value="1">Fire</option>
<option value="2">Ether</option>
```

Il faut maintenant modifier la fonction post de notre route *add-perso*. Toujours une seule clé par attributs mais nous avons un intéger dans un string.

Exemple :

```
...
"element" =>intval(parent::getParam($params, "name-of-select", false)),
...
```

Il restera à utliser notre service qui appelera le DAO. Normalement, si c'est bien fait, votre requête SQL n'as pas à changer (sauf si vous avez typé votre paramètre PDO qui passe de string a int)

Si tout se déroule comme prévu, vous pouvez de nouveau ajouter un personnage mais cette fois-ci, lié à ses attributs.

2.3 : Finalisons ceci en corrigeant l'update. Comme pour l'ajout, il vous faudra mettre à jour :

- Le controler avec la fonction *displayEdit*
- Le formulaire de la vue si vous avez 2 fichiers différents
- La route *post EditPerso* comme la route *Addperso*

Attention à la gestion des attributs qui pourrait être optionnels !

Voila ! Comme d'habitue n'hésitez pas à consulter votre prof. Mais tout devrait être fonctionnel à présent. Si c'est le cas, félicitation !!

3 - Il est dans le journal

3.1 : Voici le dernier point manquant aux fonctionnalités attendues ! L'objectif sera de réaliser la fonctionnalité avec votre savoir (et pourquoi pas un peu d'aide du prof !).

Voici ce qui est déjà réalisé

```
Étant donné : Qu'un utilisateur veut consulter le log
Quand : Il clique sur le bouton log
Alors : Le page de log s'affiche avec un bloc de texte !
```

L'objectif ici sera que chaque action (comprendre *CREATE UPDATE DELETE*) de modification de base de données (réussie ou non) soit enregistrée.
Puis dans une nouvelle page, d'afficher chaque ligne de log pour avoir un historique.

Pour ajouter un peu de piquant, la contrainte sera que le système de log passe par plusieurs fichier.

Ce dernier sera daté pour éviter de n'avoir qu'un seul énorme fichier. Exemple : *MIHOYO_08_2025.log*.

Vous aurez un dossier *logs* pour contenir tous ces fichiers.
La page devra afficher la liste des mois/années disponibles pour afficher le log correspondant.
Et voilà, vous avec carte blanche pour réaliser la fonctionnalité ! Bon courage !

Une fois cela fini nous arrivons presque au terme de notre projet !

- [x] ~~Afficher la liste des personnages~~
- [x] ~~Ajouter des personnages à la BD~~
- [x] ~~Editer un personnage~~
- [x] ~~Supprimer un personnage~~
- [ ] Gérer une authentification
- [x] ~~Gérer un journal de log~~
- [x] ~~Gérer les données liés à un personnage~~
- [x] ~~Avoir un design simple et fonctionnel~~

4 - Help to 20/20

Vous êtes arrivé en vie jusque ici et tout est fonctionnel ? Félicitation ! Vous méritez une bonne note.

CEPENDANT (à lire avec la voix de dumbledore)

La qualité de code est tout aussi importante que les fonctionnalités (non, ce n'est pas vrai en termes de note, mais l'idée est la !)

- Bien typer attributs/méthodes/paramètres/...
- Bien respecter une architecture cohérente
- Avoir remarqué les super dossiers Exception et Service, donc tu t'es renseigné pour les utiliser de façon optimale
- Commentaires et PhPDoc sont bien présents
- Les noms de variables sont pertinents (non $bloubiboulga ni $poop ne sont pertinents ici)
- Les var_dump/print/echo qui n'ont raison d'être ont bien disparu