tp4-genese-et-destruction

Project TFT - Part 04 - Génèse et Destruction !

Il est grand temps de pouvoir créer, modifier et supprimer des unités !

```
Durant tout le sujet, vous allez voir apparaitre la notion de message.
Ceci implique de retourner une information à l'utilisateur sur comment s'est déroulé le processus demandé.
Si cela vous bloque durant le sujet, vous pouvez y revenir plus tard.
Mieux vaut un CRUD qui marche sans message que bloquer jusqu'à la fin pour afficher un message !
```

Copy

1 - Et ainsi l'unité est

1.1 : Retournons sur notre formulaire d'ajout d'unités. Il est temps de déterminer la méthode et l'action dans notre balise *form*. Comme nous allons créer une donnée, les recommandations du protocole *HTTP* demandent d'utiliser *POST*. Cela permet d'utiliser la même route que l'affichage du formulaire. Nous n'aurons qu'à regarder si nous avons des données *$_POST* pour savoir si on doit gérer l'ajout.

```
<form action="index.php?action=add-unit" method="post">
```

```
Attention, le formulaire HTML ne peut gérer que les méthodes *POST* et *GET*.
```

Pour exploiter notre formulaire, chaque champs input devra posséder un attribut *name*. Sa valeur déterminera le nom de notre clé dans *$_POST*.

1.2 : Pour anticiper une erreur dans les données envoyées par le formulaire (donnée incorrect ou champ inexistant), nous allons préparer notre page à accueillir un message d'erreur.

Dans la fonction *displayAddUnit* (TP7), il faut ajouter un paramètre optionnel (pour ne pas casser notre code déjà en place) de type *?string* à valeur *null* par défaut.

Exemple :

```
function func(?type $varOptionnelle = valeurParDefaut)
```

Celui-ci sera passé à la fonction *render* dans l'array data avec une clé nommé *message* par exemple. Cela vous donnera accès à une variable *$message* dans votre vue *add-unit*. Si celle-ci existe, vous pourrez afficher la valeur de la variable en guise de message d'erreur.

```
Comme d'habitude, évitez de juste faire un appel à *echo* de votre message
et retourner du HTML/CSS.
```

1.3 : Il est temps de retravailler notre routeur pour gérer les données `$_POST` envoyé par notre formulaire. N'hésitez pas à utiliser la fonction *var_dump* sur la variable `$_POST` pour identifier comment celui-ci fonctionne.

Il est recommandé d'utiliser la fonction `getParam` de notre classe Route.
Il faudra faire un array avec les clés correspondant aux attribut de votre classe Unit et les remplir avec les informations de votre formulaire

```php
$data = [
  "name" => parent::getParam($params, "unit-nom",false),
  ...
]
```

Il est temps d'implémenter l'algorithme qui permettra de choisir ce que l'on affiche.
Nous seront directement dans la méthode post de notre route

```
Dans notre fonction post
  -> Récupérer toutes les clés nécessaires
  -> Si une exception est levée
    -> Afficher le formulaire avec un message
  -> Sinon
    -> Envoyer les données au contrôleur (avec la fonction créé juste après)
```

1.4 : Dans notre contrôleur *Unit*, nous allons créer une fonction *addUnit* qui aura pour but de :

1. Prendre les infos d'une unité en entrée (*Array* ou multi variable)
2. Lui ajouter un id généré par la fonction *uniqid*
3. Créer l'unité
   - Créer une fonction *createUnit(Unit)* qui insère une unité en BD dans notre *UnitManager*
4. Créer un message sur la réussite (ou non) de la création
5. Générer une page *Index* avec le message

```
Petite information qui pourrait vous éviter du soucis.
Votre hydratation ne sait pas gérer les _ dans les paramètres. Si vous lui donné en clé url_img, il cherchera setUrl_img.
Hors si votre fonction s'appelle setUrlImg, il ne la trouvera pas. Donc soit vous améliorez votre hydratation, vous ne passé pas des clé au format snake_case
```

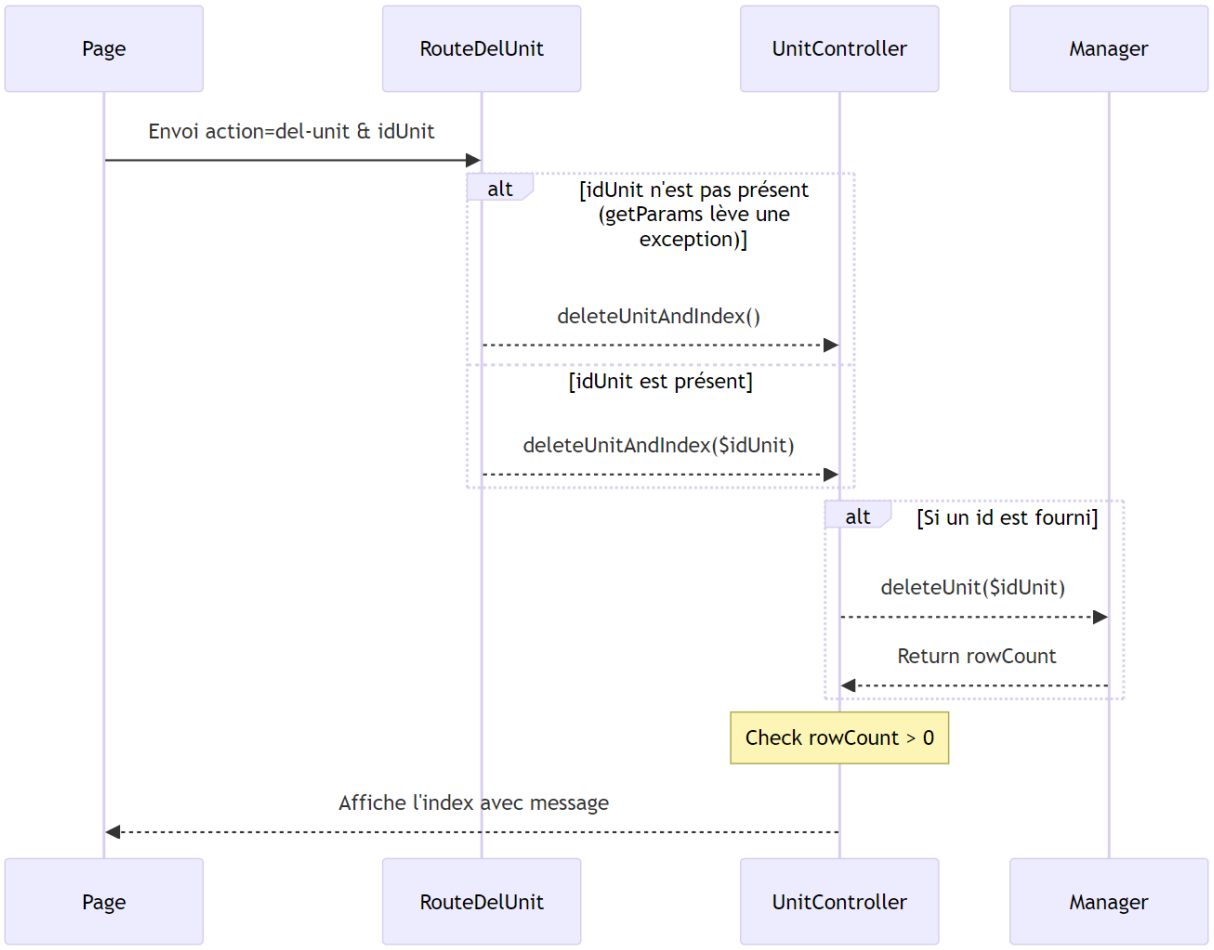
2 - Et maintenant l'unité changera ou ne sera point

2.1 : Si vous êtes un bon étudiant qui aime tester les choses pour vérifier que tout fonctionne, vous devriez avoir pléthore d'unité dans votre BD qui s'appellent Test.
On va donc préparer la suppression pour *clean up* un peu tout cela.

Voici en résumé notre fonctionnalité :

```
sequenceDiagram
  participant P as Page
  participant R as RouteDelUnit
  participant C as UnitController
  participant M as Manager
  P->>R: Envoi action=del-unit & idUnit
  alt idUnit n'est pas présent (getParams lève une exception)
    R-->>C : deleteUnitAndIndex()
  else idUnit est présent
    R-->>C : deleteUnitAndIndex($idUnit)
  end
  alt Si un id est fourni
    C-->>M: deleteUnit($idUnit)
    M-->>C: Return rowCount
  end
  Note over C: Check rowCount > 0
  C-->>P: Affiche l'index avec message
```

En version graphique :



Niveau modèle, rien de compliqué, une méthode *deleteUnit(int $idUnit = -1)* à implémenter dans le *manager*.

```
Envie de savoir si la suppression s'est bien passée ?
Regardez du côté de PDOStatement::rowCount() pour vous aider.
```

Niveau contrôleur, une méthode (soyons explicite) *deleteUnitAndIndex(int $idUnit)*. Comme son nom l'indique, on supprime l'unité (coucou le manager) puis on génère une vue *Index* avec un message (Suppression réussie ou non).

```
Si vous vous souvenez, cette fonction a déjà été créée au TP3 Q2.5.
Elle aura peut-être un nom différent.
Ce n'est pas un problème, tant que vous restez cohérent dans votre programme
(vous pouvez aussi la renommer dans tout votre projet !)
```

Puis niveau routeur, vous devriez avoir créé le bouton supprimer qui doit avoir un lien de cette forme :

```
index.php?action=del-unit&idUnit=1
```

A vous de jouer pour :

1. Traiter l'action get dans votre RouteDelUnit
2. Récupérer l'*id* depuis l'*url* (*hint:* Les infos de l'url sont passé par la méthode *GET*)
3. En cas d'erreur (l'*url* ne contient pas la donnée par exemple), appeler la fonction *deleteUnitAndIndex* mais sans paramètres
4. Appeler votre super méthode du contrôleur avec l'id en param.

```
Il est fort possible que, à ce stade du TP,
votre fonction index ne gère pas un message.
Si tel est le cas, pour éviter de casser votre code,
ajouter un paramètre optionnel à votre méthode index.
Puis passez ce paramètre à la fonction 'generer'.
```

2.2 : Marre de supprimer tous ces unités tests ? Peut-être qu'il est temps de voir pour mettre à jour nos données. Voici le process que l'on voudrait :

```
flowchart LR
  A(Click sur le bouton edit) --> B(Affiche un formulaire pré rempli)
  B --> C(Modifie les données)
  C --> D(Update dans la BD)
  D --> E(Retour sur Index avec un message sur le statut de l'update)
```

Cette fonction étant plus complexe, nous allons la couper en 2. Pour le moment, l'objectif est d'afficher le formulaire *add-unit* rempli des infos de l'unité que l'on veut modifier.

Dans le routeur, le procédé se déroule comme la fonction de suppression.
Si le paramètre *idUnit* n'existe pas, nous pouvons utiliser la méthode *displayAddUnit("id not found")* au lieu de *displayEditUnit($idPkmn)*

Dans le contrôleur, nous avons une méthode *editUnit*. Nous allons la renommer en *displayEditUnit* vu que celle-ci ne fera qu'afficher le formulaire rempli. Elle aura besoin de l'*id* de l'unité en paramètre.

Il ne manquera plus qu'à récupérer l'unité, et générer une vue *addUnit* avec l'unité en paramètre

C'est au niveau de la vue que cela devient plus complexe.

0. Pour chacune des actions ci-dessous => Vérifier si une unité existe
1. Préremplir chacun des champs avec sa valeur correspondante.
2. Ajouter un champs caché contenant l'*ID*.
3. Changer l'action du formulaire en *edit-unit*
4. Changer le titre de la page
5. Changer le texte du bouton

```
Votre code html parsemé de PhP peut vite devenir illisible !
N'hésitez pas à utiliser l'outil de formatage de votre IDE
et de bien indenter votre code !
```

2.3 : Maintenant que nous avons préparé le terrain, il est temps de faire l'*update* a proprement parler.

Pour ne pas trop compliquer la tâche, nous allons update tous les champs d'un coup sans se soucier s'ils ont été modifiés ou non (à l'exception de l'*id* bien évidemment).

Pour le *Manager*, la fonction *editUnitAndIndex(array $dataUnit)* se chargera de mettre à jour la base de donnée.

Pour le contrôleur, le processus est similaire à ce que l'on a vu avant :

0. On crée notre méthode *editUnitAndIndex(array $dataUnit)*
1. On crée notre unité
2. On l'envoi au *manager* qui fait l'*Update*
3. On vérifie si les 2 types sont les même, si oui, alors le 2ème type peut être null
4. On génère un message en fonction du résultat
5. On génère notre vue *Index* avec le message

```
Nous générons beaucoup de fois une page Index.
Or ce code existe déjà dans notre MainController.
Il serait bon de s'en servir.
N'hésitez pas à utiliser un paramètre de votre UnitController que vous
instanciez dans sa méthode __construct().
Vous pouvez ainsi disposer de ses méthodes et invoquer l'index.
```

Pour le routeur, après avoir vérifié que nous possédons bien des données *POST*, nous récupérons ce qui est nécessaire via *getParam*.
Puis, on transmet sous forme d'un *array* à notre contrôleur.

Et si tout fonctionne, nous devrions maintenant avoir un processus fonctionnel. N'oubliez pas de gérer l'exception si un paramètre obligatoire est manquant

3 - Récap

Nous avons déjà bien avancé à ce stade. Si tout est fonctionnel, bien codé (`#RevoirSonModuleQualité`), et avec une pointe de *design* qui permet de ressembler plus à un site web qu'à une expérimentation d'un doctorant, vous pouvez espérer une note très correcte !

Il est temps de faire le point sur l'avancée. Au niveau de l'architecture du projet, cela devrait ressembler à cela (Le bonus décrit après est inclus.).

```
📦 TonSuperProjet
├ 📂 Config
│ ├ 📜 Config.php
│ └ 📜 dev.ini
├ 📂 Controllers
│ ├ 📜 UnitController.php
│ ├ 📜 MainController.php
│ └ 📜 OriginController.php (optionnel)
├ 📂 Exceptions
├ 📂 Helpers
│ └ 📜 Message.php
├ 📂 Models
│ ├ 📜 Unit.php
│ ├ 📜 UnitDAO.php
│ └ 📜 BasePDODAO.php
├ 📂 public
│ ├ 📂 css
│ │ └ 📜 main.css
│ └ 📂 img
├ 📂 Vendors
│  └ 📂 Plates
├ 📂 Views
│ ├ 📜 template.php
│ ├ 📜 message.php
│ ├ 📜 add-unit.php
│ ├ 📜 add-origin.php
│ ├ 📜 home.php
│ ├ 📜 error.php
│ └ 📜 search.php
└ 📜 index.php
```

Evidemment, certains fichiers peuvent différer, comme les noms des fonctions/classes.

Faisons un récap de ce que l'on attend de notre application.

- [x] ~~Afficher la liste des unités~~
- [x] ~~Ajouter des unités à la BD~~
- [x] ~~Editer une unité~~
- [x] ~~Supprimer une unité~~
- [ ] Rechercher une unité particulière
- [ ] Affecter des origines à une unité
- [x] ~~Avoir un design simple et fonctionnel~~
- [ ] Plein de bonus

On a bien avancé et l'objectif du prochain TP sera de cocher 2 points de plus !

4 - Bonus

Il serait agréable de gérer nos messages de façons plus détaillé. Effectivement, nous envoyons un texte et ... puis c'est tout. Ajouter peut-être un titre au message et changer sa couleur (via des classes CSS) suivant son contenu (Bleu pour les infos, Rouge pour les erreurs, Vert pour les succès).

Pour éviter la duplication de code, je vous invite à créer un fichier */views/message.php* qui se chargera du *template* du message.

Il ne manquera plus qu'à inclure ses fichiers dans vos *templates* de page à l'aide d'un simple :

```
<?=$this->insert('message',['message' => $message])?>
```

Puis dans vos pages, lorsque vous appelez le template à l'aide de la fonction layout, transmettez la variable message.

```
$this->layout('template', ['title' => 'TP TFT', 'message' => $message]) ?>
```

Si l'on veut pousser encore plus loin, au lieu de gérer plusieurs variables, il serait temps de créer une classe *Message* dans un dossier *helpers* par exemple ;)

Vous pouvez donner des valeurs par défaut surtout pour *color* si vous utilisez un framework css. Mais aussi utiliser des constantes pour ne pas avoir à taper le nom de vos couleurs à chaque fois.

Exemple pour *materialize*:

```
const MESSAGE_COLOR_SUCCESS = "green lighten-2";
const MESSAGE_COLOR_ERROR = "red lighten-2";

public function __construct(string $message, string $color="light-blue
lighten-1", string $title="Message")
```