Tutoriel CodeIgniter 3

Pierre Nerzic - octobre 2022

Présentation

CodeIgniter est un *framework serveur*. Il s'agit d'un outil complet pour créer un serveur web, c'est à dire un serveur HTTP capable de produire des pages HTML en fonction des demandes des clients. Il intègre des scripts permettant de développer sous le patron de conception MVC (modèle-vue-contrôleur). Ce patron sépare tout ce qui est affichage HTML (les vues), de ce qui est stockage persistant en base de données (modèle) et enfin des algorithmes de gestion de la navigation de l'utilisateur (les contrôleurs).

CodeIgniter est un *framework* léger et rapide, facile à comprendre et à apprendre.

Préparatifs

Installation

- Télécharger le fichier ZIP, dernière version sur https://codeigniter.com/userguide3/installation/downloads.html
- Extraire ses fichiers et renommer le dossier, par exemple en `"Tutoriel"`
- Exécuter le projet localement : se placer dans le dossier et lancer `php -S localhost:8000`
- Documentation : https://codeigniter.com/userguide3/, en particulier General Topics et ses chapitres (controllers, views, models, helpers...).

Configuration d'un nouveau projet

- Modifier `application/config/config.php` :
  - `$config['base_url'] = 'http://localhost:8000/';`
- Modifier `application/config/database.php` :
  - `'hostname' => 'servbdd.iutlan.etu.univ-rennes1.fr',`
  - `'username' => '<votre login>',`
  - `'password' => '<votre mot de passe>',`
  - `'database' => 'pg_<votre login>',`
  - `'dbdriver' => 'postgre',`
- Si les tests sont faits sur une BDD PostgresQL personnelle, il faut installer le paquet `php-pgsql`.
- Pour faire des tests avec SQLite3 (bdd locale légère sans serveur), nécessite le paquet `php-sqlite3` installé, et avoir créé le dossier `application/databases` :
  - `'hostname' => '',`
  - `'username' => '',`
  - `'password' => '',`
  - `'database' => 'application/databases/db.sqlite',`
  - `'dbdriver' => 'sqlite3',`

Exemple 1 : page totalement statique

On veut accéder à http://localhost:8000/test1. Dans cet URL, `test1` est une page dont le contenu est figé.

Dans les concepts CodeIgniter, cet URL est découpé en plusieurs *segments* : `test1` est considéré comme une classe. Il faut donc programmer une classe PHP de type contrôleur pour répondre à cet URL. Cette classe contient une méthode qui génère le code HTML demandé par le navigateur client.

Le minimum à faire est de créer un fichier `application/controllers/Test1.php`, attention à la majuscule obligatoire :

```
<?php
class Test1 extends CI_Controller {

    public function index()
    {
        echo '<p>Hello super World !</p>';
    }
}
```

NB: la balise de fin `?>` n'est pas obligatoire, voir la première note dans PHP: Instruction separation.

NB: tous ces sources sont inclus dans l'archive `TutorielCodeIgniter3.zip`.

- Ouvrir http://localhost:8000/test1. Changer le message et rafraîchir la page.

Ce qu'il faut retenir :

1) Un URL `http://serveur/classe` est séparé en une partie *serveur* et une partie *classe*.
2) La Classe (initiale en majuscule) est le nom à la fois d'un fichier `application/controllers/Classe.php` et d'une classe à l'intérieur de ce même fichier.
3) Cette classe doit hériter de `CI_Controller` et doit définir des méthodes, dont par exemple `index()`.
4) La méthode `index()` est appelée par défaut avec l'URL `http://localhost:8000/classe`
5) Les méthodes doivent produire du code HTML qui est retourné au client.

Exemple 2 : page paramétrée par l'URL

Il est possible de faire que l'URL d'une page fournisse des paramètres. Attention, ce ne sont pas des informations revenant d'un formulaire comme `$_GET` ou `$_POST` (voir plus loin), mais une sorte de complément de chemin, présent dans l'URL : `http://localhost:8000/test2/message/bonjour/Pierre`.

- Créer un fichier `application/controllers/Test2.php`, attention à la majuscule:

```
<?php
class Test2 extends CI_Controller {

    public function message($msg, $personne)
    {
        echo '<p>Il y a un message pour '.$personne.' : '.$msg.'</p>';
    }
}
```

- Ouvrir
  - http://localhost:8000/test2/message/bonjour/Pierre,
  - http://localhost:8000/test2/message/bonjour/Pierre/Nerzic,
  - http://localhost:8000/test2/message/bonjour.

Ce qu'il faut retenir :

1) Un URL `http://serveur/classe/methode/param1/param2/...` est séparé en une partie *serveur*, une partie *classe*, une partie *méthode* et des parties *paramètres*. Avec le serveur, seule la partie classe est obligatoire.
2) Les paramètres sont passés à la méthode (`public function`) définie dans la classe.
3) Des paramètres manquants engendrent une erreur. Les paramètres supplémentaires sont ignorés.
4) La classe doit être écrite avec l'initiale en majuscule, mais les méthodes seulement en minuscules.

En PHP, on peut donner des valeurs par défaut aux paramètres d'une fonction, et donc éventuellement éviter une erreur s'ils ne sont pas fournis.

```
<?php
class Test2 extends CI_Controller {

    public function message($msg='ràs', $personne='inconnu')
    {
```

```
        echo '<p>Il y a un message pour '.$personne.' : '.$msg.'</p>';
    }
}
```

Remarque : pour masquer une fonction qui serait dans un contrôleur, mais qu'on ne veut pas rendre accessible par un URL, il suffit de la préfixer par `private` au lieu de `public`.

```
<?php
class Test2 extends CI_Controller {

    public function message($msg='ràs', $personne='inconnu')
    {
        echo '<p>Il y a un message pour '.$personne.' : '.$msg.'</p>';
    }

    private function paspourtesyeux()
    {
        echo "<p>On a dit private, on n'a pas dit public !</p>";
    }
}
```

Ouvrir http://localhost:8000/test2/paspourtesyeux. Changer `private` en `public`, réouvrir.

Question ouverte : comment faire pour que l'accès à la méthode privée se traduise par un message d'erreur poli et non pas un code 404 ? Peut-être que toutes les erreurs de ce type, classe inconnue, méthode inconnue et méthode privée aboutissent à une erreur 404.

Exemple 3 : page définie par un template

Dans les exemples précédents, le contrôleur fait tout le travail dans la méthode désignées par l'URL. Normalement, on doit séparer ces deux aspects, car le contrôleur n'est pas fait pour générer du HTML, mais pour récupérer les paramètres, lancer des accès aux données d'une base, faire des vérifications, etc. Donc on va avoir un couple contrôleur et *vue* qui travailleront ensemble. La génération d'HTML est faite seulement par la vue.

Exemple 3a : vue simple

Au plus simple, c'est un fichier qui ne contient que du code HTML. Il faut quand même le nommer en `.php`.

- Créer `application/views/test3a.php` :

```
<html>
<head>
    <title>Tutoriel CodeIgniter</title>
</head>
<body>
    <p>On essaie de rendre les choses compréhensibles en allant étape par étape.</p>
</body>
</html>
```

Ce fichier n'est pas atteignable par un URL, car CodeIgniter intercepte les URLs et les envoie à des contrôleurs. Il faut donc un contrôleur pour afficher le document.

- Créer `application/controllers/Test3a.php` (majuscule) :

```
<?php
class Test3a extends CI_Controller {

    public function index()
    {
        $this->load->view('test3a');
    }
}
```

- Ouvrir http://localhost:8000/test3a.

Cet URL déclenche la méthode `index()` du contrôleur `Test3a`. Cette méthode retourne la vue `test3a.php`.

Ce qu'il faut retenir :

1) Le contrôleur délègue la génération d'HTML à une *vue* par l'instruction `$this->load->view('nom_vue');`
2) Une vue est un fichier PHP qui génère du HTML – il ne doit faire que ça. De tels fichiers sont appelés *templates* dans d'autres outils.
3) Dans le contrôleur, la vue est désignée seulement par son nom. Le fichier est dans `application/views/` avec l'extension `.php`.

Exemple 3b : plusieurs vues

Ce qui est intéressant, c'est qu'on peut décomposer une vue en plusieurs éléments. Il est fréquent d'avoir plusieurs choses qu'on retrouve d'une page à l'autre sur un site web, un bandeau de présentation en haut, un menu burger, un bandeau en bas, etc ou même plus finement, ça peut être des feuilles de style, des scripts JS, etc. CodeIgniter permet de placer ces choses chacune dans une vue spécifique, et le contrôleur les emploie selon les besoins.

- Créer le fichier `application/views/haut.php` :

```
<html>
<head>
    <title>Tutoriel CodeIgniter</title>
</head>
<body>
<h2>Tutoriels CodeIgniter</h2>
```

- Créer le fichier `application/views/bas.php` :

```
<hr/>
<p style="font-size: small;">CodeIgniter, c'est pas CI compliqué.</p>
</body>
</html>
```

- Créer le fichier `application/views/test3b.php` :

```
<p>Tutoriel montrant comment le contrôleur regroupe plusieurs vues.</p>
```

- Créer `application/controllers/Test3b.php` :

```
<?php
class Test3b extends CI_Controller {

    public function index()
    {
        $this->load->view('haut');
        $this->load->view('test3b');
        $this->load->view('bas');
    }
}
```

- Ouvrir http://localhost:8000/test3b.

Chaque `$this->load->view()` ajoute un contenu de vue au résultat final. Attention à bien concevoir le regroupement.

NB: par défaut, toutes les vues sont placées dans `application/views/`, mais si elles sont nombreuses, ça pourrait devenir ingérable. On peut faire des sous-dossiers, et l'ajouter au nom de la vue : `$this->load->view('commun/haut');`

Exemple 4 : vue paramétrée

Le plus intéressant est une vue qui change selon un paramètre calculé par le contrôleur. C'est à dire qu'une variable est initialisée par le contrôleur et fournie à la vue. Cette dernière met sa valeur dans le HTML final.

Exemple 4a : variables toutes simples

- Créer `application/views/test4a.php` :

```
<p>J'ai un message pour <?php echo $personne; ?> :</p>
<p>Il paraît que <?=$message;?>.</p>
```

NB: la syntaxe `<?=$var;?>` est un raccourci pour `<?php echo $var; ?>`. On l'utilise pour afficher une seule variable. On aurait pu faire pareil pour l'autre variable : `<?=$personne;?>`.

- Créer `application/controllers/Test4a.php` (majuscule) :

```
<?php
class Test4a extends CI_Controller {

    public function index()
    {
        $variables['personne'] = 'Pierre Nerzic';
        $variables['message'] = 'tout va bien';

        $this->load->view('haut');
        $this->load->view('test4a', $variables);
        $this->load->view('bas');
    }
}
```

- Ouvrir http://localhost:8000/test4a.

Ce qu'il faut retenir :

1) Le contrôleur définit un tableau associatif (clé => valeur) qui est fourni à la vue. **Attention au piège**, on ne peut pas fournir des variables séparées. Il faut qu'elles soient dans un même tableau associatif, sous forme de couples clé => valeur. En revanche, ce tableau peut avoir le nom qu'on veut.
2) Dans la vue, les différentes clés du tableau sont disponibles directement sous forme de variables.

Exemple 4b : paramètres complexes

On prend l'exemple d'un tableau de données, défini par le contrôleur et affiché par la vue sous forme d'une liste.

- Créer `application/views/test4b.php` :

```
<p>Voici la liste des <?=$concept?> :</p>
<ul>
    <?php
    foreach ($liste as $item) {
        echo '<li>'.$item.'</li>'.PHP_EOL;
    }
    ?>
</ul>
```

- Créer `application/controllers/Test4b.php` (majuscule) :

```
<?php
class Test4b extends CI_Controller {

    public function index()
    {
        $parametres['concept'] = 'langages de programmation';
        $parametres['liste'] = array("C", "Java", "Python", "JavaScript");

        //$parametres['concept'] = 'animaux';
        //$parametres['liste'] = array("Marmotte", "Chamois", "Bouquetin", "Tétras", "Gypaète");

        $this->load->view('haut');
        $this->load->view('test4b', $parametres);
        $this->load->view('bas');
```

```
    }
}
```

- Ouvrir http://localhost:8000/test4b.
- Échanger les commentaires sur la variable paramètres pour faire apparaître l'autre liste. C'est ce qui prouve l'indépendance entre contrôleur et vues.

Exemple 5 : formulaires

On s'intéresse maintenant à la gestion des formulaires HTML par CodeIgniter. Une vue contient un formulaire, et une autre vue affiche les informations saisies. C'est le même contrôleur qui organise le travail : d'abord afficher le formulaire, puis quand les données sont saisies, afficher le résultat.

- Créer `application/views/test5a.php` :

```
<p>Saisissez les informations d'un nouveau produit :</p>

<?php echo validation_errors(); ?>

<form action='/test5/valider' method='POST'>
  <table>
  <tbody>
    <tr>
      <td><label for="nom">Nom</label></td>
      <td><input type="text" name="nom" /></td>
    </tr>
    <tr>
      <td><label for="categ">Catégorie</label></td>
      <td><input type="text" name="categ" /></td>
    </tr>
    <tr>
      <td><label for="prix">Prix</label>
      <td><input type="text" name="prix" /></td>
    </tr>
  </tbody>
  </table>
  <br/>
  <input type="submit" name="submit" value="Valider" />
</form>
```

C'est un formulaire classique à part la ligne `<?php echo validation_errors(); ?>` qui rajoute de quoi afficher un message d'erreur si le formulaire est incomplet. On note l'URL qui est sollicité par le bouton `submit` : `test5/valider`.

- Créer `application/views/test5b.php` :

```
<p>Voici les informations saisies :</p>
<ul>
    <li>nom = "<?=$nom;?>"</li>
    <li>categ = "<?=$categ;?>"</li>
    <li>prix = "<?=$prix;?>"</li>
</ul>
<p>Faire une autre saisie ? <a href="/test5">cliquez ici</a>.</p>
```

- Créer `application/controllers/Test5.php` :

```
<?php
class Test5 extends CI_Controller {

    // constructeur
    function __construct()
    {
        // super()
```

```
        parent::__construct();

        // inclure des fonctions pour aider à gérer les formulaires
        $this->load->helper('form');
        $this->load->library('form_validation');
    }

    public function index()
    {
        $this->load->view('haut');
        $this->load->view('test5a');    // formulaire
        $this->load->view('bas');
    }

    public function valider()
    {
        // validité du formulaire
        $this->form_validation->set_rules('nom', 'Nom', 'required');
        $this->form_validation->set_rules('categ', 'Catégorie', 'required');

        // est-ce que c'est un retour du formulaire et est-il valide ?
        if ($this->form_validation->run() === FALSE) {
            // pas de formulaire ou champs invalides => réafficher le formulaire
            return $this->index();
        } else {
            // retour des données => afficher le produit
            $this->load->view('haut');
            $this->load->view('test5b', $_POST);    // valeurs saisies
            $this->load->view('bas');
        }
    }
}
```

- Ouvrir http://localhost:8000/test5.

C'est un peu compliqué. D'abord il y a un constructeur, écrit avec la syntaxe PHP. Il charge des ensembles de fonctions utiles appelés *helpers*.

Ensuite il y a la méthode `index()` qui affiche le formulaire. Enfin, il y a la méthode `valider()` qui reçoit les données du formulaire. Elle est chargée de vérifier que le formulaire est complet. On doit définir des critères de validité des données. Ensuite, si les données ne sont pas valides, elle ré-affiche le formulaire. Inversement, si tout est correct, c'est la deuxième vue qui est affichée avec les données.

Il y a une variante permettant de retrouver les valeurs déjà saisies dans le formulaire. Il faut modifier le contrôleur et la vue comme ceci.

- Modifier `application/views/test5a.php` :

```
<p>Saisissez les informations d'un nouveau produit :</p>

<?php echo validation_errors(); ?>

<form action='/test5/valider' method='POST'>
  <table>
  <tbody>
    <tr>
      <td><label for="nom">Nom</label></td>
      <td><input type="text" name="nom" value="<?=isset($nom)?$nom:''?>"/></td>
    </tr>
    <tr>
      <td><label for="categ">Catégorie</label></td>
      <td><input type="text" name="categ" value="<?=isset($categ)?$categ:''?>" /></td>
    </tr>
```

```
    <tr>
      <td><label for="prix">Prix</label>
      <td><input type="text" name="prix" value="<?=isset($prix)?$prix:''?>" /></td>
    </tr>
  </tbody>
  </table>
  <br/>
  <input type="submit" name="submit" value="Valider" />
</form>
```

Remarquer la manière dont on affiche les anciennes valeurs, c'est à dire avec un test au cas où elles ne soient pas définies dans `$_POST`.

- Modifier la fonction `valider` dans `application/controllers/Test5.php` :

```
    public function valider()
    {
        // validité du formulaire
        $this->form_validation->set_rules('nom', 'Nom', 'required');
        $this->form_validation->set_rules('categ', 'Catégorie', 'required');

        // est-ce que c'est un retour du formulaire et est-il valide ?
        if ($this->form_validation->run() === FALSE) {
            // pas de formulaire ou champs invalides => réafficher le formulaire
            $this->load->view('haut');
            $this->load->view('test5a', $_POST);  // formulaire avec les valeurs déjà saisies, ou pas
            $this->load->view('bas');
        } else {
            // retour des données => afficher le produit
            $this->load->view('haut');
            $this->load->view('test5b', $_POST); // valeurs saisies
            $this->load->view('bas');
        }
    }
```

Exemple 6 : modèles de données

Dans une vraie application, les données dynamiques ne sont pas stockées dans des variables des contrôleurs, mais dans une base de données. C'est le seul moyen de les rendre persistantes, c'est à dire qu'elles ne disparaissent pas dès qu'on change de page.

CodeIgniter emploie ce qui s'appelle un ORM *Object Relational Mapping*. C'est un dispositif qui relie une table d'une BDD à une classe : les colonnes de cette table deviennent des variables d'instance (membres) de la classe, et les n-uplets de la table deviennent des instances de cette classe. En simplifiant, quand on fait `$item = new Classe(...);`, ça va faire l'équivalent d'un `INSERT` dans la table, et si on fait `$item->colonne = valeur;`, ça va faire un `UPDATE` dans la base. Sauf que c'est un peu plus compliqué dans CodeIgniter.

Exemple 6a : modèle simple

Il faut d'abord créer une classe qui va représenter les n-uplets d'une table. Son constructeur doit créer la table si elle n'existe pas. Et cette classe contient des méthodes permettant de manipuler les données.

- Créer `application/models/Produits.php`, attention à la majuscule :

```
<?php

// nom de la table gérée par ce modèle
define("TABLE", "Produits");

class Produits extends CI_Model {

    // variables membres = colonnes de la table
    public $nom;
```

```
    public $categorie;
    public $prix;

    // constructeur
    public function __construct()
    {
        // super()
        parent::__construct();

        // création de la table si elle n'existe pas déjà
        $this->load->dbforge();
        $colonnes = array(
            'nom'       => array('type' => 'VARCHAR'),
            'categorie' => array('type' => 'VARCHAR'),
            'prix'      => array('type' => 'REAL', 'null' => TRUE)
        );
        $this->dbforge->add_field($colonnes);
        $this->dbforge->add_key('nom', TRUE);   // clé primaire
        $this->dbforge->create_table(TABLE, TRUE);

        // charger les données
        $this->load->database();
    }

    // retourne la liste de tous les produits, triés par nom
    public function get_all()
    {
        $this->db->order_by('nom');
        $query = $this->db->get(TABLE);
        return $query->result();
    }
}
```

La table est créée par un outil appelé *dbforge*, voir sa documentation. Chercher les instructions `add_field`, `add_key` et `create_table` dans la documentation afin de comprendre la logique, et voir d'autres possibilités (default, autoincrement, not null, unique, identifiant, ...).

La méthode `$this->db->get(nomtable);` effectue une sorte de `SELECT * FROM nomtable` qui est modulé par des méthodes comme `$this->db->order_by(nomchamp);` ou `$this->db->where(nomchamp, valeur);`, voir plus loin. Après avoir utilisé `$this->db->get`, on retourne la liste des n-uplets sous forme d'un tableau d'objets par `$query->result();`. Toutes ces méthodes sont décrites dans la documentation, dans *Query Builder Class* et *Generating Query Results*.

- Créer `application/controllers/Test6.php` (majuscule) :

```
<?php
class Test6 extends CI_Controller {

    // constructeur
    function __construct()
    {
        // super()
        parent::__construct();

        // inclure des fonctions comme redirect()
        $this->load->helper('url');

        // inclure des fonctions pour aider à gérer les formulaires
        $this->load->helper('form');
        $this->load->library('form_validation');

        // charger le modèle
```

```
        $this->load->model('produits', '', TRUE);
    }

    public function index()
    {
        // obtenir la liste de tous les produits
        $data['liste'] = $this->produits->get_all();

        // instancier la vue
        $this->load->view('haut');
        $this->load->view('test6a', $data);
        $this->load->view('bas');
    }
}
```

- Créer `application/views/test6a.php` :

```
<p>Voici la liste des produits :</p>
<table>
  <thead>
    <tr><th>nom</th><th>catégorie</th><th>prix</th></tr>
  </thead>
  <tbody>
    <?php
    foreach ($liste as $produit) {
        echo '<tr>';
        echo '<td>'.$produit->nom.'</td>';
        echo '<td>'.$produit->categorie.'</td>';
        echo '<td>'.$produit->prix.'</td>';
        echo '</tr>'.PHP_EOL;
    }
    ?>
  </tbody>
</table>
```

- Ouvrir http://localhost:8000/test6. La première fois affiche une liste vide puisque la table vient d'être créée. Ajouter manuellement quelques n-uplets dans la base, à l'aide de l'outil de gestion (SQLworkbench ou pgAdmin). Rafraîchir la page.

Ce qu'il faut retenir :

1) On n'écrit pas de requêtes SQL, mais à la place, on emploie des concepts de la programmation objet : table = classe, colonne = variable membre, n-uplet = instance `$this`.
2) C'est un peu compliqué de créer un modèle et d'ajouter des méthodes, mais ce n'est à faire qu'une fois.

Exemple 6b : ajout de n-uplets

Maintenant, on veut pouvoir ajouter des n-uplets à l'aide d'un formulaire.

- Ajouter ces lignes dans `application/models/Produits.php` :

```
    // ajoute un produit défini par un formulaire
    public function insert_POST()
    {
        // $this->input->post('var') retourne $_POST['var'] ou NULL si indéfini
        $data['nom']       = $this->input->post('nom');
        $data['categorie'] = $this->input->post('categ');
        $data['prix']      = $this->input->post('prix');

        $this->db->insert(TABLE, $data);
    }
```

- Ajouter cette ligne dans `application/views/test6a.php` :

```
<p>Ajouter un produit ? <a href="/test6/nouveau">cliquez ici</a>.</p>
```

Quand l'utilisateur cliquera sur le lien, ça ouvrira la méthode `nouveau` dans le contrôleur.

- Créer `application/views/test6b.php` :

```
<p>Saisissez les informations d'un nouveau produit :</p>

<?php echo validation_errors(); ?>

<form action='/test6/nouveau' method='POST'>
  <table>
  <tbody>
    <tr>
      <td><label for="nom">Nom</label></td>
      <td><input type="text" name="nom" value="<?=isset($nom)?$nom:''?>"/></td>
    </tr>
    <tr>
      <td><label for="categ">Catégorie</label></td>
      <td><input type="text" name="categ" value="<?=isset($categ)?$categ:''?>" /></td>
    </tr>
    <tr>
      <td><label for="prix">Prix</label>
      <td><input type="text" name="prix" value="<?=isset($prix)?$prix:''?>" /></td>
    </tr>
  </tbody>
  </table>
  <br/>
  <input type="submit" name="submit" value="Valider" />
</form>
```

NB: par rapport à l'exemple 5, le formulaire et la vue qui affiche les données sont inversées.

- Ajouter ces lignes dans `application/controllers/Test6.php` :

```
    public function nouveau()
    {
        // validité du formulaire
        $this->form_validation->set_rules('nom', 'Nom', 'required');
        $this->form_validation->set_rules('categ', 'Catégorie', 'required');

        // est-ce que c'est un retour du formulaire ?
        if ($this->form_validation->run() === FALSE) {
            // pas de retour ou champs invalides => afficher le formulaire
            $this->load->view('haut');
            $this->load->view('test6b', $_POST);
            $this->load->view('bas');
        } else {
            // retour des données => enregistrer le produit
            $this->produits->insert_POST();
            // afficher la liste
            redirect('/test6', 'refresh');
        }
    }
```

- Ouvrir http://localhost:8000/test6/nouveau ou en cliquant sur le lien dans la page `test6`. Saisissez des informations puis validez. La liste est affichée, avec le nouveau n-uplet.

Ce qu'il faut retenir :

1) Chaque classe est concernée : il faut une méthode d'insertion `insert_POST` dans le modèle, il faut une vue contenant un formulaire, il faut un contrôleur pour organiser le travail.
2) La fonction `insert_POST` affecte les champs de `$this` puis appelle `insert(table, $this);`.

3) Le formulaire de la vue doit permettre la saisie avec des input ayant les mêmes attributs `name` que ceux de `insert_POST`.
4) Le contrôleur vérifie les informations et appelle les méthodes appropriées.

Exemple 6c : sélection de n-uplets sur une condition

On veut n'afficher que certains produits, en choisissant leur catégorie. Il faut un formulaire pour choisir la catégorie, puis n'afficher que les n-uplets concernés.

- Créer `application/views/test6c.php` :

```
<p>Quelle est la catégorie à afficher ?</p>

<form action='/test6/categorie' method='POST'>
  <label for="categ-select">Faites un choix parmi les catégories existantes :</label>
  <select name="categ" id="categ-select">
    <option value="">--choisissez une catégorie--</option>
    <?php
    foreach ($liste as $row) {
        $categ = $row['categorie'];
        echo '<option value="'.$categ.'">'.$categ.'</option>'.PHP_EOL;
    }
    ?>
  </select>
  <br/>
  <input type="submit" name="submit" value="Valider" />
</form>
```

Ce formulaire reçoit une liste un peu bizarre, ce sont comme des n-uplets de la table `produits` mais avec seulement la colonne `categorie` et celle-ci en un seul exemplaire. Voir la requête plus bas. Noter qu'il appelle `/test6/categorie` dans le contrôleur.

- Ajouter cette ligne dans `application/views/test6a.php` :

```
<p>Ne lister qu'une catégorie ? <a href="/test6/categorie">cliquez ici</a>.</p>
```

- Ajouter ces lignes dans `application/controllers/Test6.php` :

```
    public function categorie()
    {
        // validité du formulaire
        $this->form_validation->set_rules('categ', 'Catégorie', 'required');

        // est-ce que c'est un retour du formulaire ?
        if ($this->form_validation->run() === FALSE) {
            // pas de retour ou champs invalides => afficher le formulaire
            $data['liste'] = $this->produits->get_categories();

            $this->load->view('haut');
            $this->load->view('test6c', $data);
            $this->load->view('bas');
        } else {
            // retour des données => afficher la liste avec uniquement la catégorie
            $data['liste'] = $this->produits->get_all_categorie($_POST['categ']);

            // instancier la vue
            $this->load->view('haut');
            $this->load->view('test6a', $data);
            $this->load->view('bas');
        }
    }
```

- Ajouter ces lignes dans `application/models/Produits.php` :

```php
    // retourne la liste de toutes les catégories connues
    public function get_categories()
    {
        $this->db->select('categorie');
        $this->db->distinct();
        $query = $this->db->get(TABLE);
        return $query->result();
    }

    // retourne la liste des produits d'une catégorie spécifique
    public function get_all_categorie($categ)
    {
        $this->db->where('categorie', $categ);
        $query = $this->db->get(TABLE);
        return $query->result();
    }
```

Ces deux nouvelles fonctions sont :

- `get_categories()` : elle revient à faire un `SELECT DISTINCT(categorie) FROM produits;` Il faut comprendre que les résultats sont des n-uplets qui ne comprennent qu'une seule colonne, `categorie`, mais dans la vue `test6c`, on fait comme si c'étaient des produits entiers.
- `get_all_categorie($categ)` : elle revient à faire un `SELECT * FROM produits WHERE categorie=$categ;`

La documentation de toutes ces méthodes sur le modèle se trouve dans Query Builder Class.

Exemple 6d : modification d'un n-uplet

Comment modifier un n-uplet existant ? Il faut utiliser la même sorte de formulaire que pour la création d'un n-uplet, `test6b`, mais avec le nom dans la liste des produits.

- Modifier ce groupe de lignes dans `application/views/test6a.php` :

```php
  <thead>
    <tr><th>nom</th><th>catégorie</th><th>prix</th><th>actions</th></tr>
  </thead>
  <tbody>
    <?php
    foreach ($liste as $produit) {
        echo '<tr>';
        echo '<td>'.$produit->nom.'</td>';
        echo '<td>'.$produit->categorie.'</td>';
        echo '<td>'.$produit->prix.'</td>';
        echo '<td>';
        echo '<a href="/test6/edition/'.$produit->nom.'">modifier</a>';
        echo '</td>';
        echo '</tr>'.PHP_EOL;
    }
    ?>
  </tbody>
```

Chaque produit a maintenant un lien pointant vers `/test6/edition/nom du produit`. Cet URL indique donc la classe, `Test6`, la méthode `edition` ainsi qu'un paramètre, qui est l'identifiant de l'item concerné.

- Créer `application/views/test6d.php` :

```php
<p>Éditez les informations du produit <?=$nom;?> :</p>

<?php echo validation_errors(); ?>

<form action="/test6/edition/<?=$nom;?>" method="POST">
  <table>
  <tbody>
    <tr>
```

```
      <td><label for="nom">Nom</label></td>
      <td><input type="text" name="nom" value="<?=$nom;?>" readonly/></td>
    </tr>
    <tr>
      <td><label for="categ">Catégorie</label></td>
      <td><input type="text" name="categ" value="<?=$categorie;?>"/></td>
    </tr>
    <tr>
      <td><label for="prix">Prix</label>
      <td><input type="text" name="prix" value="<?=$prix;?>"/></td>
    </tr>
  </tbody>
  </table>
  <br/>
  <input type="submit" name="submit" value="Valider" />
</form>
```

Le formulaire ouvre le même URL qui aboutit à la méthode `edition` suivante.

- Ajouter ces lignes dans `application/controllers/Test6.php` :

```
public function edition($nom)
{
    // validité du formulaire
    $this->form_validation->set_rules('nom', 'Nom', 'required');
    $this->form_validation->set_rules('categ', 'Catégorie', 'required');

    // est-ce que c'est un retour du formulaire ?
    if ($this->form_validation->run() === FALSE) {
        // pas de retour ou champs invalides => afficher le formulaire avec les infos du produit
        $produit = $this->produits->get($nom);
        $this->load->view('haut');
        $this->load->view('test6d', $produit);
        $this->load->view('bas');
    } else {
        // retour des données => enregistrer les modifications du produit
        $this->produits->update_POST();
        // afficher la liste
        redirect('/test6', 'refresh');
    }
}
```

- Ajouter ces lignes dans `application/models/Produits.php` :

```
// retourne l'un des produits identifié par son nom
public function get($nom)
{
    $this->db->where('nom', $nom);
    $query = $this->db->get(TABLE);
    return $query->row();
}

// modifie le produit désigné et défini par un formulaire
// NB: le nom ne peut pas être changé
public function update_POST()
{
    $data['categorie'] = $this->input->post('categ');
    $data['prix']      = $this->input->post('prix');

    $this->db->where('nom', $this->input->post('nom'));
    $this->db->update(TABLE, $data);
}
```

Exemple 6e : suppression d'un n-uplet

Pour finir, on souhaite pouvoir supprimer un produit. Il faut rajouter un lien de suppression dans la liste des produits, ce lien déclenche une méthode qui supprime l'élément. Il est hors propos de demander confirmation (script JS dans la vue).

- Modifier ce groupe de lignes dans `application/views/test6a.php` :

```
        echo '<td>';
        echo '<a href="/test6/edition/'.$produit->nom.'">modifier</a>';
        echo ' <a href="/test6/suppression/'.$produit->nom.'">supprimer</a>';
        echo '</td>';
```

- Ajouter ces lignes dans `application/controllers/Test6.php` :

```
public function suppression($nom)
{
    // suppression sans confirmation (à faire en amont dans la page HTML)
    $this->produits->delete($nom);
    // réafficher la liste
    redirect('/test6', 'refresh');
}
```

- Ajouter ces lignes dans `application/models/Produits.php` :

```
// supprime le produit identifié par son nom
public function delete($nom)
{
    $this->db->where('nom', $nom);
    $query = $this->db->delete(TABLE);
}
```