

TP MVC

Version : 1

A rendre un fichier compressé (zip, 7z, rar) contenant :

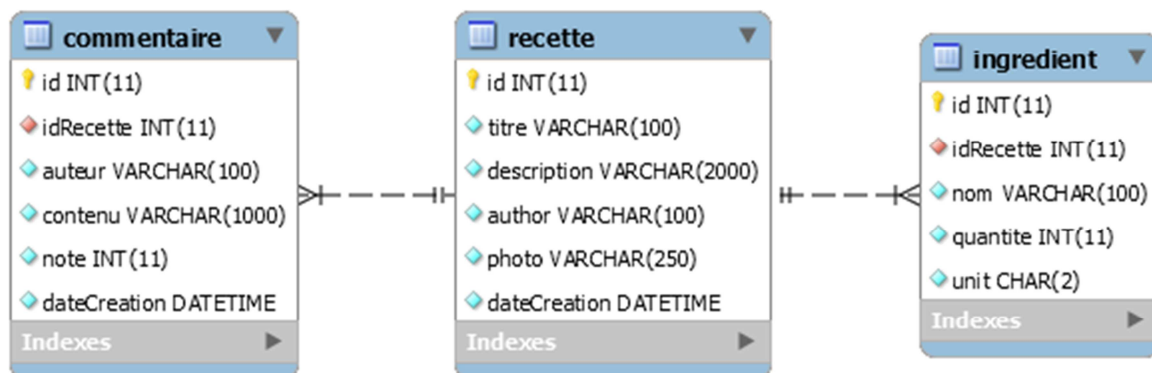
- Le projet contenant les développements de l'application Web et la base de données

Nommer le fichier nom.b3.tpmvc.zip

Envoyer le fichier à l'adresse nicolas.chevalier@epsi.fr

L'objectif de ce TP est de créer un site implémentant une architecture logicielle en utilisant le modèle MVC.





























La base de données utilisée est très simple. Elle se compose de trois tables : deux tables stockent les recettes avec les ingrédients du blog et l'autre les commentaires associés aux recettes.



Le script de création de la base de données est contenu dans l'archive avec le sujet.

Travail à réaliser :

La structure du site à réaliser est présentée ci-dessous :

- ▲  config
 -  dev.ini
- ▲  controllers
 -  controllerAccueil.php
 -  controllerRecette.php
- ▲  css
 -  style.css
- ▲  framework
 -  configuration.php
 -  controller.php
 -  modele.php
 -  requete.php
 -  router.php
 -  vue.php
- ▲  img
 -  tartiflette.jpg
 -  veloute-de-carotte-au-cumin.jpg
- ▲  modeles
 -  commentaire.php
 -  recette.php
- ▲  vues
 - ▲  accueil
 -  index.php
 - ▲  recette
 -  recette.php
 -  erreur.php
 -  gabarit.php
 -  index.php

Les pages à réaliser sont les suivantes :

Page d'accueil :



Mon Blog de Recettes

Bienvenue sur mon blog de recettes



Tartiflette

07/01/2019

La tartiflette savoyarde est un gratin de pommes de terre avec du Reblochon fondu dessus

Ingrédients

- 750 g Pommes de terre
- 1 u Reblochon
- 200 g Lardons
- 3 cs Crème fraîche épaisse
- 2 u Oignons
- 20 g Beurre
- 1 cc Sel
- 1 p Poivre

Commentaires

Nicolas : Super recette

Note : 5/5

09/01/2019

Votre Nom

Votre commentaire

Note

1 ▼

Commenter

Erreurs

Le fichier style.css permettra de spécifier le design du site :

```
html, body {
    height: 100%;
}

body {
    color: #bfbfbf;
    background: black;
    font-family: 'Futura-Medium', 'Futura', 'Trebuchet MS', sans-serif;
}

h1 {
    color: white;
}

.titreRecette {
    margin-bottom : 0px;
}

#global {
    min-height: 100%;
    background: #333534;
    width: 70%;
    margin: auto;
    text-align: justify;
    padding: 5px 20px;
}

#contenu {
    margin-bottom : 30px;
}

#titreBlog, #piedBlog {
    text-align: center;
}

#txtCommentaire {
    width: 50%;
}
```

Le modèle MVC :

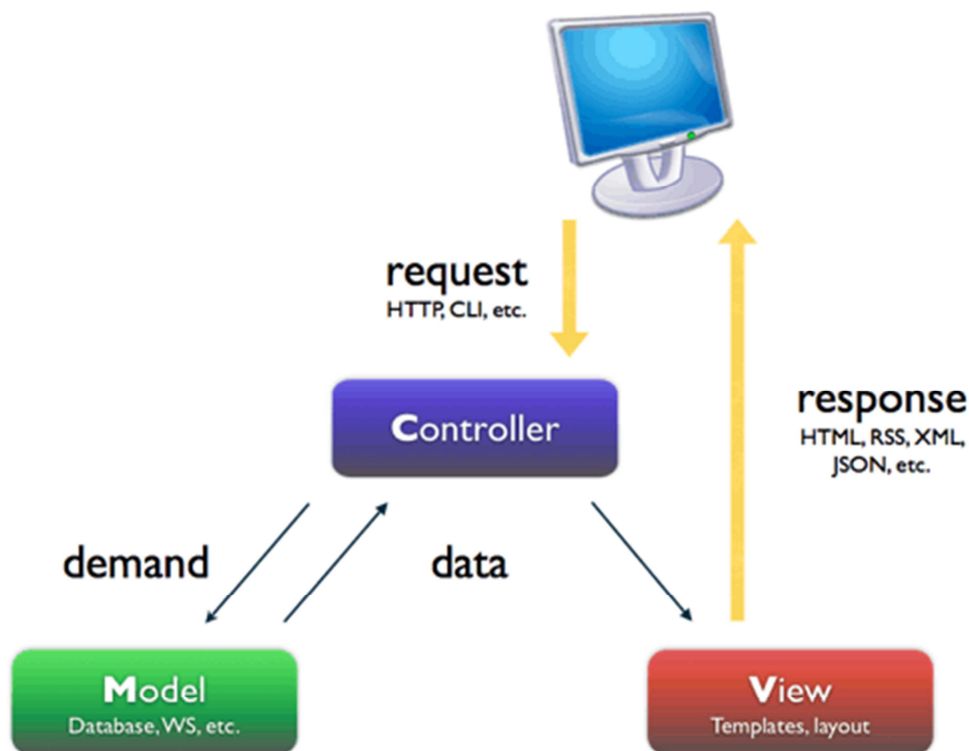
Le modèle MVC décrit la manière d'architecturer une application en la décomposant en trois sous-parties :

- Partie Modèle ;
- Partie Vue ;
- Partie Contrôleur.

La partie Modèle d'une architecture MVC encapsule la logique métier (« business logic ») ainsi que l'accès aux données. Il peut s'agir d'un ensemble de fonctions (Modèle procédural) ou de classes (Modèle orienté objet).

La partie Vue s'occupe des interactions avec l'utilisateur : présentation, saisie et validation des données.

La partie Contrôleur gère la dynamique de l'application. Elle fait le lien entre l'utilisateur et le reste de l'application.



La demande de l'utilisateur (exemple : requête HTTP) est reçue et interprétée par le Contrôleur.

Celui-ci utilise les services du Modèle afin de préparer les données à afficher.

Ensuite, le Contrôleur fournit ces données à la Vue, qui les présente à l'utilisateur.

Une application construite sur le principe du MVC se compose toujours de trois parties distinctes. Cependant, il est fréquent que chaque partie soit elle-même décomposée en plusieurs éléments. On peut ainsi trouver plusieurs modèles, plusieurs vues ou plusieurs contrôleurs à l'intérieur d'une application MVC.

Mise en place d'un framework :

Un framework fournit un ensemble de services de base, généralement sous la forme de classes en interaction. À condition de respecter l'architecture qu'il préconise (pratiquement toujours une déclinaison du modèle MVC), un framework PHP libère le développeur de nombreuses tâches techniques comme le routage des requêtes, la sécurité, la gestion du cache, etc. Cela lui permet de se concentrer sur l'essentiel, c'est-à-dire ses tâches métier. Il existe une grande quantité de frameworks PHP. Parmi les plus connus, citons Symfony, Zend Framework, CakePHP ou encore CodeIgniter.

Notre petit framework n'aura bien pas la richesse fonctionnelle et le niveau de qualité des exemples précédents. Son but est d'illustrer « de l'intérieur » et aussi simplement que possible les bases du fonctionnement d'un framework PHP.

Notre framework va nous permettre de mettre en place les fonctionnalités suivantes :

- Rendre configurable les paramètres d'accès à la base de données (serveur, nom de la BD, identifiant de connexion, mot de passe) ;
- Instancier un seul objet PDO d'accès à la BD afin d'optimiser les performances ;
- Rendre le routage de la requête (choix de l'action à exécuter) automatique ;
- Filtrer les paramètres des requêtes en entrée ;
- Nettoyer les données insérées dans les vues pour éviter le risque de failles XSS.

Les fichiers du framework seront ajoutés au projet dans un répertoire framework.

Accès générique aux données

Comme vous le verrez plus loin nous allons créer une classe abstraite *Modele* qui sera la classe mère de nos classes modèle. Cette classe aura pour fonction d'effectuer la connexion à la base de données et de créer l'objet PDO.

Afin de rendre paramétrable les valeurs pour accéder à la base de données nous allons mettre en place un fichier de configuration. Ce fichier de configuration sera ajouté dans un répertoire config.

Afin de lire les informations contenues dans ce fichier de configuration nous allons créer un composant dont le rôle sera de gérer la configuration du site. Ce composant prend la forme d'une classe appelée logiquement *Configuration*.

Nous allons ajouter le fichier `configuration.php` à notre répertoire framework de notre projet :

`configuration.php` :

```
class Configuration {  
  
    private static $parametres;  
  
    // Renvoie la valeur d'un paramètre de configuration  
    public static function get($nom, $valeurParDefaut = null) {  
        if (isset(self::getParametres()[$nom])) {
```



```

        $valeur = self::getParametres()[$nom];
    }
    else {
        $valeur = $valeurParDefaut;
    }
    return $valeur;
}

// Renvoie le tableau des paramètres en le chargeant au besoin
private static function getParametres() {
    if (self::$parametres == null) {
        $cheminFichier = "config/dev.ini";
        if (!file_exists($cheminFichier)) {
            $cheminFichier = "config/prod.ini";
        }
        if (!file_exists($cheminFichier)) {
            throw new Exception("Aucun fichier de configuration trouvé");
        }
        else {
            self::$parametres = parse_ini_file($cheminFichier);
        }
    }
    return self::$parametres;
}
}

```

Cette classe encapsule un tableau associatif clés/valeurs (attribut \$parametres) stockant les valeurs des paramètres de configuration. Ce tableau est statique (un seul exemplaire par classe), ce qui permet de l'utiliser sans instancier d'objet Configuration.

La classe dispose d'une méthode statique publique nommée get() qui permet de rechercher la valeur d'un paramètre à partir de son nom. Si le paramètre en question est trouvé dans le tableau associatif, sa valeur est renvoyée. Sinon, une valeur par défaut est renvoyée. On rencontre au passage le mot-clé PHP self qui permet de faire référence à un membre statique.

Enfin, la méthode statique privée getParametres() effectue le chargement tardif du fichier contenant les paramètres de configuration. Afin de faire cohabiter sur un même serveur une configuration de développement et une configuration de production, deux fichiers sont recherchés dans le répertoire config du site : dev.ini (cherché en premier) et prod.ini. La lecture du fichier de configuration utilise la fonction PHP parse_ini_file(). Celle-ci instancie et renvoie un tableau associatif immédiatement attribué à l'attribut \$parametres.

Grâce à cette classe, on peut externaliser la configuration d'un site en dehors de son code source. Voici par exemple le fichier de configuration correspondant à notre blog d'exemple :

dev.ini :

```

; Configuration pour le développement

[BD]
dsn = 'mysql:host=localhost;dbname=blog_recette;charset=utf8'
login = root
mdp =

```

Un changement de paramètres de connexion, par exemple pour employer un autre utilisateur que root, nécessite uniquement une mise à jour de ce fichier de configuration.

Modèle orienté objet :

Nous allons utiliser le modèle objet de PHP.

Nous allons créer un modèle par entité du domaine, tout en regroupant les services communs dans une classe commune.

La classe de base Modele va permettre d'implémenter les méthodes permettant de se connecter à la base de données et d'exécuter une requête. Elle sera abstract car elle sert uniquement à fournir des méthodes de base aux autres modèles (Recette et Commentaire).

modele.php :

```
require_once 'configuration.php';

abstract class Modele {

    /** Objet PDO d'accès à la BD
     * Statique donc partagé par toutes les instances des classes dérivées */
    private static $bdd;

    // Exécute une requête SQL éventuellement paramétrée
    protected function executerRequete($sql, $params = null) {
        if ($params == null) {
            $resultat = $this->getBdd()->query($sql);    // exécution directe
        }
        else {
            $resultat = $this->getBdd()->prepare($sql); // requête préparée
            $resultat->execute($params);
        }
        return $resultat;
    }

    // Renvoie un objet de connexion à la BD en initialisant la connexion au besoin
    private function getBdd() {
        if (self::$bdd == null) {
            // Récupération des paramètres de configuration BDD
            $dsn = Configuration::get("dsn");
            $login = Configuration::get("login");
            $mdp = Configuration::get("mdp");

            // Création de la connexion
            self::$bdd = new PDO($dsn, $login, $mdp,
                array(PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION));
        }
        return self::$bdd;
    }
}
```

Notre classe Modele utilise les fonctionnalités de Configuration afin de récupérer les paramètres de configuration de la BDD.

L'attribut \$bdd est static afin de ne créer qu'une seule instance de la classe PDO partagée par toutes les classes dérivées de Modele. Ainsi, l'opération de connexion à la base de données ne sera réalisée qu'une seule fois.

Pour effectuer des requêtes vous aurez besoin d'utiliser la méthode `executerRequete` en spécifiant :

- La requête SQL
- Les paramètres de la requête sous forme de tableau (array)

Les requêtes sont des requêtes préparées. Il faut donc spécifier des `?` dans vos requêtes à l'emplacement des données qui seront envoyées lors de l'exécution.

Exemple pour récupérer une recette :

```
SELECT * from recette WHERE id = ?
```

La classe `Recette` hérite de la classe `Modele` et implémente les méthodes permettant de récupérer la liste des recettes et les informations sur une recette :

`recette.php` :

```
require_once 'framework/modele.php';

class Recette extends Modele {

    // Renvoie la liste des recettes du blog
    public function getRecettes() {
        // code à implémenter
        // retourne la liste des recettes
        // utiliser pour cela executerRequete avec la requête SQL
    }

    // Renvoie les informations sur une recette
    public function getRecette($idRecette) {
        // code à implémenter
        // retourne la recette ou génère un message d'erreur via une exception : Aucune
recette ne correspond à l'identifiant '$idRecette'
        // utiliser pour cela executerRequete avec la requête SQL et $idRecette en
paramètre (attention les paramètres sont sous forme de tableau)
    }

    // Renvoie les ingrédients d'une recette
    public function getIngredients($idRecette) {
        // code à implémenter
        // retourne la liste des ingrédientscode
        // utiliser pour cela executerRequete avec la requête SQL et $idRecette en
paramètre (attention les paramètres sont sous forme de tableau)
    }
}
```

La classe `Commentaire` hérite de la classe `Modele` et implémente les méthodes permettant de récupérer la liste des commentaires associés à une recette et d'ajouter un commentaire dans la base

`commentaire.php` :

```
require_once 'framework/modele.php';

class Commentaire extends Modele {
```

```

// Renvoie la liste des commentaires associés à une recette
public function getCommentaires($idRecette) {
    // code à implémenter
    // retourne la liste des commentaires
    // utiliser pour cela executerRequete avec la requête SQL
}

// Ajoute un commentaire dans la base
public function ajouterCommentaire($idRecette, $auteur, $contenu, $note) {
    // code à implémenter
    // requête d'insert pour ajouter un commentaire
    // utiliser pour cela executerRequete avec la requête SQL et $idRecette,
    $auteur, $contenu, $note et $date en paramètre (attention les paramètres sont sous
    forme de tableau)
}
}

```

Routage automatique de la requête

Dans les framework basés sur la requête il faut pouvoir déterminer quel contrôleur appeler et quelle action à exécuter suivant la requête demandée. Cela permet également de choisir quelle vue afficher.

Afin de dispatcher les demandes nous allons mettre en place un contrôleur frontal. Celui-ci sera modélisé à l'aide d'une classe Router dont la méthode principale analyse la requête entrante pour déterminer l'action à entreprendre. On parle souvent de routage de la requête.

Le Router que nous allons mettre en place sera générique et intégré au framework.

Nous allons placer les classes suivantes dans notre framework :

- requete.php
- router.php
- vue.php

Pour atteindre l'objectif décrit précédemment, nous allons commencer par ajouter une classe Requete dans notre framework, dont le rôle est de modéliser une requête. Pour l'instant, le seul attribut de cette classe est un tableau rassemblant les paramètres de la requête. Par la suite, on pourrait y ajouter d'autres informations sur la requête : en-têtes HTTP, session, etc.

requete.php :

```
class Requete {  
  
    // paramètres de la requête  
    private $parametres;  
  
    public function __construct($parametres) {  
        $this->parametres = $parametres;  
    }  
  
    // Renvoie vrai si le paramètre existe dans la requête  
    public function existeParametre($nom) {  
        return (isset($this->parametres[$nom]) && $this->parametres[$nom] != "");  
    }  
  
    // Renvoie la valeur du paramètre demandé  
    // Lève une exception si le paramètre est introuvable  
    public function getParametre($nom) {  
        if ($this->existeParametre($nom)) {  
            return $this->parametres[$nom];  
        }  
        else  
            throw new Exception("Paramètre '$nom' absent de la requête");  
    }  
}
```

Au début du routage, un objet Requete sera instancié afin de stocker les paramètres de la requête reçue.

Afin de gérer les vues nous allons créer une classe Vue dont le rôle sera de gérer la génération des vues.

vue.php :

```
class Vue {

    // Nom du fichier associé à la vue
    private $fichier;
    // Titre de la vue (défini dans le fichier vue)
    private $titre;

    public function __construct($action, $controller = "") {
        // Détermination du nom du fichier vue à partir de l'action et du constructeur
        $fichier = "vues/";
        if ($controller != "") {
            $fichier = $fichier . $controller . "/";
        }
        $this->fichier = $fichier . $action . ".php";
    }

    // Génère et affiche la vue
    public function generer($donnees) {
        // Génération de la partie spécifique de la vue
        $contenu = $this->genererFichier($this->fichier, $donnees);
        // Génération du gabarit commun utilisant la partie spécifique
        $vue = $this->genererFichier('vues/gabarit.php',
            array('titre' => $this->titre, 'contenu' => $contenu));
        // Renvoi de la vue au navigateur
        echo $vue;
    }

    // Génère un fichier vue et renvoie le résultat produit
    private function genererFichier($fichier, $donnees) {
        if (file_exists($fichier)) {
            // Rend les éléments du tableau $donnees accessibles dans la vue
            extract($donnees);
            // Démarrage de la temporisation de sortie
            ob_start();
            // Inclut le fichier vue
            // Son résultat est placé dans le tampon de sortie
            require $fichier;
            // Arrêt de la temporisation et renvoi du tampon de sortie
            return ob_get_clean();
        }
        else {
            throw new Exception("Fichier '$fichier' introuvable");
        }
    }
}
```

Le constructeur de Vue prend en paramètre une action et un controller, qui détermine le fichier vue utilisé. Sa méthode generer() génère d'abord la partie spécifique de la vue afin de définir son titre (attribut \$titre) et son contenu (variable locale \$contenu). Ensuite, le gabarit est généré en y incluant les éléments spécifiques de la vue. Sa méthode interne genererFichier() encapsule l'utilisation de require et permet en outre de vérifier l'existence du fichier vue à afficher. Elle utilise la fonction extract pour que la vue puisse accéder aux variables PHP requises, rassemblées dans le tableau associatif \$donnees.

On impose une convention de nommage aux vues :

- chaque vue doit résider dans le sous-répertoire **vues/** ;
- dans ce répertoire, chaque vue est stockée dans un sous-répertoire portant le nom du contrôleur associé à la vue, exemple **recette/** ;
- chaque fichier porte directement le nom de l'action aboutissant à l'affichage de cette vue, exemple **recette.php**.
-

Un routage générique consistera à déduire automatiquement le contrôleur et la méthode d'action en fonction de la requête. Pour atteindre cet objectif, toutes les URL de notre site devront intégrer au moins 2 paramètres afin de spécifier le controller et l'action. Par exemple :
index.php?controller=xxx&action=yyy&id=zzz.

On peut à présent créer le code du routeur afin de rendre le routage automatique et donc générique.

router.php :

```
require_once 'requete.php';
require_once 'vue.php';

class Routeur {

    // Route une requête entrante : exécute l'action associée
    public function routerRequete() {
        try {
            // Fusion des paramètres GET et POST de la requête
            $requete = new Requete(array_merge($_GET, $_POST));

            $controller = $this->creerController($requete);
            $action = $this->creerAction($requete);

            $controller->executerAction($action);
        }
        catch (Exception $e) {
            $this->genererErreur($e);
        }
    }

    // Crée le contrôleur approprié en fonction de la requête reçue
    private function creerController(Requete $requete) {
        $controller = "Accueil"; // Contrôleur par défaut
        if ($requete->existeParametre('controller')) {
            $controller = $requete->getParametre('controller');
            // Première lettre en majuscule
            $controller = ucfirst(strtolower($controller));
        }
        // Création du nom du fichier du contrôleur
        $classeController = "Controller" . $controller;
        $fichierController = "controllers/" . $classeController . ".php";
        if (file_exists($fichierController)) {
            // Instanciation du contrôleur adapté à la requête
            require($fichierController);
            $controller = new $classeController();
            $controller->setRequete($requete);
            return $controller;
        }
    }
}
```

```

        else
            throw new Exception("Fichier '$fichierController' introuvable");
    }

    // Détermine l'action à exécuter en fonction de la requête reçue
    private function creerAction(Requete $requete) {
        $action = "index"; // Action par défaut
        if ($requete->existeParametre('action')) {
            $action = $requete->getParametre('action');
        }
        return $action;
    }

    // Gère une erreur d'exécution (exception)
    private function gererErreur(Exception $exception) {
        $vue = new Vue('erreur');
        $vue->generer(array('msgErreur' => $exception->getMessage()));
    }
}

```

La méthode principale **routerRequete()** de cette classe instancie un objet *Requete* en fusionnant les données des variables `$_GET` et `$_POST`, afin de pouvoir analyser toute requête issue soit d'une commande HTTP GET, soit d'une commande HTTP POST.

Ensuite, cette méthode fait appel à deux méthodes internes afin d'instancier le contrôleur approprié et d'exécuter l'action correspondant à la requête reçue.

La méthode **creerController()** récupère le paramètre **controller** de la requête reçue et le concatène pour construire le nom du fichier contrôleur (celui qui contient la classe associée) et renvoyer une instance de la classe associée. En l'absence de ce paramètre, elle cherche à instancier la classe **controllerAccueil** qui correspond au contrôleur par défaut.

La méthode **creerAction()** récupère le paramètre **action** de la requête reçue et le renvoie. En l'absence de ce paramètre, elle renvoie la valeur « **index** » qui correspond à l'action par défaut.

Cela n'est possible qu'en imposant à tous les contrôleurs des contraintes de nommage strictes : chaque contrôleur doit résider dans le sous-répertoire **controllers/** sous la forme d'un fichier définissant une classe nommée **ControllerXXX** (**XXX** correspondant à la valeur du paramètre **controller** dans la requête). Le fichier doit porter le même nom que la classe.

Enfin, la méthode privée **gererErreur()** permet d'afficher la vue d'erreur.

Contrôleur orienté objet :

Notre routeur fait appel aux méthodes **setRequete()** et **executerAction()** de l'objet contrôleur instancié. Comme il serait maladroite de dupliquer la définition de cette méthode dans tous nos contrôleurs. Nous allons donc définir une classe abstraite **Controller** regroupant les services communs aux contrôleurs et l'intégrer dans notre framework.

controller.php :

```
require_once 'requete.php';
require_once 'vue.php';

abstract class Controller {

    // Action à réaliser
    private $action;

    // Requête entrante
    protected $requete;

    // Définit la requête entrante
    public function setRequete(Requete $requete) {
        $this->requete = $requete;
    }

    // Exécute l'action à réaliser
    public function executerAction($action) {
        if (method_exists($this, $action)) {
            $this->action = $action;
            $this->{$this->action}();
        }
        else {
            $classeController = get_class($this);
            throw new Exception("Action '$action' non définie dans la classe $classeController");
        }
    }

    // Méthode abstraite correspondant à l'action par défaut
    // Oblige les classes dérivées à implémenter cette action par défaut
    public abstract function index();

    // Génère la vue associée au contrôleur courant
    protected function genererVue($donneesVue = array()) {
        // Détermination du nom du fichier vue à partir du nom du contrôleur actuel
        $classeController = get_class($this);
        $controller = str_replace("Controller", "", $classeController);
        // Instanciation et génération de la vue
        $vue = new Vue($this->action, $controller);
        $vue->generer($donneesVue);
    }
}
```

Cette classe a pour attributs l'action à réaliser et la requête. Sa méthode **executerAction()** met en œuvre le concept de **réflexion** : elle utilise les fonctions PHP **method_exists** et **get_class** afin de faire appel la méthode ayant pour nom l'action à réaliser.

La méthode **index()** est abstraite. Cela signifie que tous nos contrôleurs, qui hériteront de **Controller**, devront obligatoirement définir une méthode **index()** qui correspond à l'action par défaut (quand le paramètre **action** n'est pas défini dans la requête) **même si celle-ci ne fait rien**.

Enfin, la méthode **genererVue()** permet d'automatiser le lien entre contrôleur et vue : les paramètres de création de la vue sont déduits du nom du contrôleur et de l'action à réaliser.

Les contrôleurs devront donc :

- Hériter de la classe Controller
- implémenter la méthode index même si elle n'effectue aucune opération ;
- utiliser la méthode genererVue pour la création d'une vue ;
- utiliser getParametre de la classe requete pour récupérer des paramètres (get ou post) ;
- utiliser executerAction pour l'exécution d'une action si besoin.

Nous allons maintenant créer une classe ControllerAccueil pour gérer l'accueil et une classe ControllerRecette pour gérer l'affichage d'une recette.

Chaque contrôleur va utiliser les services des classes des parties Modèle et Vue définies précédemment.

controllerAccueil.php :

```
require_once 'modeles/recette.php';
require_once 'framework/controller.php';
require_once 'framework/vue.php';

class ControllerAccueil extends Controller {

    private $recette;

    public function __construct() {
        $this->recette = new Recette();
    }

    // Affiche la liste de toutes les recettes du blog
    public function index() {
        $recettes = $this->recette->getRecettes();
        $this->genererVue(array('recette' => $recettes));
        // code à implémenter
        // récupérer la liste des recettes
        // générer la vue
    }
}
```

controllerRecette.php :

```
require_once 'modeles/recette.php';
require_once 'modeles/commentaire.php';
require_once 'framework/controller.php';
require_once 'framework/vue.php';

class controllerRecette extends Controller {

    private $recette;
```

```

private $commentaire;

public function __construct() {
    $this->recette = new Recette();
    $this->commentaire = new Commentaire();
}

public function index() {

// Affiche les détails sur une recette
public function recette() {
    // code à implémenter
    // récupérer la recette
    // récupérer la liste des ingrédients
    // récupérer la liste des commentaires
    // générer la vue
}

// Ajoute un commentaire à une recette
public function commenter() {
    // récupérer les paramètres (idRecette, auteur, contenu, note)
    // Sauvegarde du commentaire
    // Actualisation de l'affichage de la recette
}
}

```

Chaque classe contrôleur instancie les classes modèles requises, puis utilise leurs méthodes (getRecette, getRecettes, getIngredients, getCommentaires, ajouterCommentaire) pour récupérer les données nécessaires aux vues. La méthode genererVue de la classe Controller définie plus haut est utilisée en lui passant en paramètre un tableau associatif contenant l'ensemble des données nécessaires à la génération de la vue. Exemple pour la vue Accueil :

```

$this->genererVue(array('recette' => $recettes));

```

Vue orientée objet :

Nous allons mettre en place un gabarit afin de centraliser les éléments d'affichage communs (header et footer). Deux variables devront être renseignées avant d'appeler une vue afin de renseigner les éléments spécifiques qui sont le titre et le contenu : variables \$titre et \$contenu.

gabarit.php :

```
<!doctype html>
<html lang="fr">
  <head>
    <meta charset="UTF-8" />
    <link rel="stylesheet" href="css/style.css" />
    <title><?= $titre ?></title>
  </head>
  <body>
    <div id="global">
      <header>
        <a href="index.php"><h1 id="titreBlog">Mon Blog</h1></a>
        <p>Je vous souhaite la bienvenue sur ce blog.</p>
      </header>
      <div id="contenu">
        <?= $contenu ?>
      </div> <!-- #contenu -->
      <footer id="piedBlog">
        Blog réalisé avec le modèle MVC.
      </footer>
    </div> <!-- #global -->
  </body>
</html>
```

Comme décrit précédemment chaque vue doit respecter un formalisme en terme de nommage et d'emplacement.

La vue de l'accueil doit donc s'appeler index.php et doit se trouver dans le répertoire accueil. Elle permet d'afficher la liste des recettes.

Son contenu statique sera le suivant (Exemple avec la Tartiflette) :

```
<div id="global">
  <article>
    <header>
      
      <a href="index.php?controller=recette&action=recette&id=id tartiflette">
        <h1 class="titreRecette">
          Tartiflette
        </h1>
      </a>
      <time>
        07/01/2019
      </time>
    </header>
    <p>
      La tartiflette savoyarde est un gratin de pommes de terre avec du
      Reblochon fondu dessus
    </p>
```

```

        </article>
        <hr />
    </div>

```

Afin de rendre dynamique la vue, il faudra lui passer une liste de recettes dans une variable « recettes » et remplacer les informations statiques « id tartiflette », « Tartiflette », « 07/01/2019 » et « La tartiflette savoyarde est un gratin de pommes de terre avec du Reblochon fondu dessus » par du code PHP correspondant aux valeurs de chaque recette contenu dans la liste. Utiliser pour parcourir la liste une boucle foreach.

Il faudra également renseigner la variable titre en lui donnant par exemple la valeur :

```
$this->titre = "Mon Blog";
```

La vue de la recette doit s'appeler recette.php et doit se trouver dans le répertoire recette. Elle permet d'afficher le détail d'une recette, les ingrédients, les commentaires et d'ajouter un commentaire.

Son contenu statique sera le suivant (Exemple avec la Tartiflette) :

```

<div id="global">
    <article>
        <header>
            
            <h1 class="titreRecette">
                Tartiflette
            </h1>
            <time>
                07/01/2019
            </time>
        </header>
        <p>
            La tartiflette savoyarde est un gratin de pommes de terre avec du
            Reblochon fondu dessus
        </p>
    </article>
    <hr />
    <header>
        <h2 id="titreIngredient">
            Ingrédients
        </h2>
        <ul>
            <li>750 g Pommes de terre</li>
            <li>1 u Reblochon</li>
            <li>200 g Lardons</li>
            <li>3 cs Crème fraîche épaisse</li>
            <li>2 u Oignons</li>
            <li>20 g Beurre</li>
            <li>1 cc Sel</li>
            <li>1 p Poivre</li>
        </ul>
    </header>

    <h2 id="titreCommentaire">
        Commentaires
    </h2>

```

```

<div class="divCommentaire">
    <p>Nicolas : Super recette </p>
    <p> Note : 5/5 </p>
    <p>
        09/01/2019
    </p>
    <hr>
</div>

<form method="post" action="index.php?controller=recette&action=commenter&id=id
recette" >
    <input id="auteur" name="auteur" type="text" placeholder="Votre Nom"
/><br />
    <textarea id="txtCommentaire" name="contenu" rows="4" placeholder="Votre
commentaire" ></textarea><br/>
    <label for="note">Note</label><br />
    <select name="note" id="note">
        <option value="1">1</option>
        <option value="2">2</option>
        <option value="3">3</option>
        <option value="4">4</option>
        <option value="5">5</option>
    </select>
    <br />
    <input type="submit" value="Commenter" />
</form>
<div id="erreur">
    <p> Erreurs </p>
</div>
</div>

```

Afin de rendre dynamique la vue, il faudra passer à cette vue la recette, une liste d'ingrédients et de commentaires dans 3 variables « recette », « ingredients », « commentaires » qui contiendront les informations d'une recette, la liste des ingrédients et la listes des commentaires. Remplacer les informations statiques « id tartiflette », « Tartiflette », « 07/01/2019 » et « La tartiflette savoyarde est un gratin de pommes de terre avec du Reblochon fondu dessus » par les valeurs contenues dans la variable « recette ». Effectuer une boucle foreach pour remplacer la liste des ingrédients par un texte contenant la quantité, l'unité et le nom de l'ingrédient. Effectuer une boucle foreach pour remplacer la liste des commentaires par l'auteur du commentaire et le contenu du commentaire.

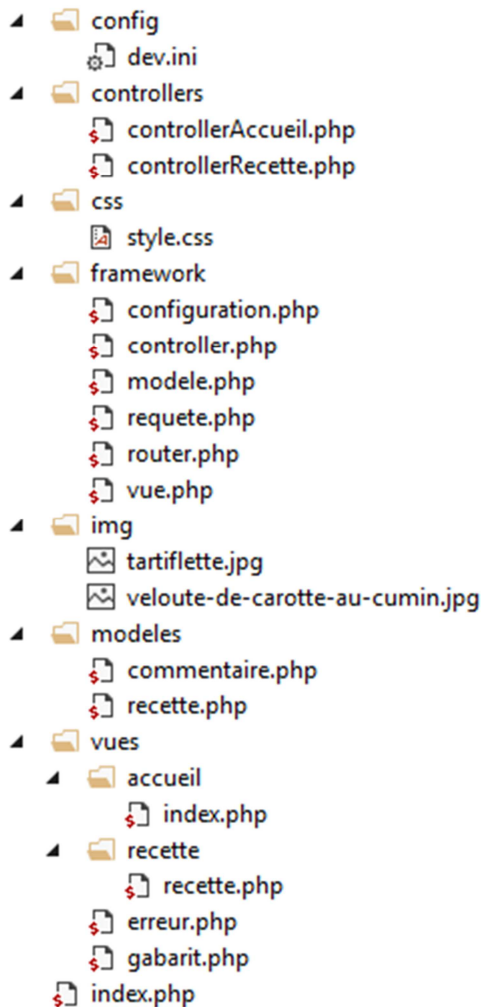
La vue de l'erreur doit s'appeler erreur.php et doit se trouver à la racine du répertoire vues. Elle permet d'afficher le message d'erreur :

```
<p>Une erreur est survenue : <?= $msgErreur ?></p>
```

Cette vue est générée par le fichier router.php défini un peu plus haut avec la méthode erreur. Il n'y a donc pas de code à rajouter pour le fonctionnement de cette vue.

Architecture finale du site :

Comme indiqué précédemment nous avons créé un répertoire **framework/** qui regroupera les éléments qui le composent. Il sera constitué des fichiers source des six classes qui le composent.



Ce *framework* impose des contraintes sur l'architecture du site qui l'utilise :

- un répertoire **config/** contient le ou les fichiers de configuration (**dev.ini** et **prod.ini**) lues par la classe **Configuration** ;
- un répertoire **controllers/** contient tous les contrôleurs ;
- chaque classe contrôleur (ainsi que le fichier PHP associé) commence par le terme « Controller » (exemple : **controllerAccueil**).
- à chaque action réalisable correspond une méthode publique dans la classe contrôleur associée ;
- une méthode **index()** (action par défaut) est définie dans chaque contrôleur ;
- un répertoire **vues/** contient toutes les vues ;
- chaque vue est définie dans un sous-répertoire de Vues/ portant le même nom que le contrôleur associé à la vue (exemple : **vues/accueil/index.php** pour la vue associée à l'action par défaut du contrôleur d'accueil) ;

- les fichiers **gabarit.php** (gabarit commun à toutes les vues) et **erreur.php** (affichage du message d'erreur) sont stockés dans le répertoire **vues/** ;