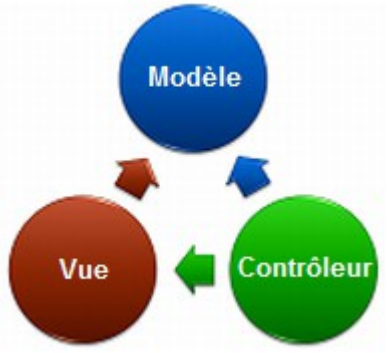
Architecture MVC

Table des matières

Le patron modèle-vue-contrôleur (en abrégé MVC, de l'anglais model-view-controller), tout comme les patrons modèle-vue-présentation ou Présentation, abstraction, contrôle, est un modèle destiné à répondre aux besoins des applications interactives en séparant les problématiques liées aux différents composants au sein de leur architecture respective.

Ce paradigme regroupe les fonctions nécessaires en trois catégories :

- un modèle (modèle de données) ;
- une vue (présentation, interface utilisateur) ;
- un contrôleur (logique de contrôle, gestion des événements, synchronisation).



1. Introduction

Il y a des problèmes en programmation qui reviennent tellement souvent qu'on a créé toute une série de bonnes pratiques que l'on a réunies sous le nom de design patterns.

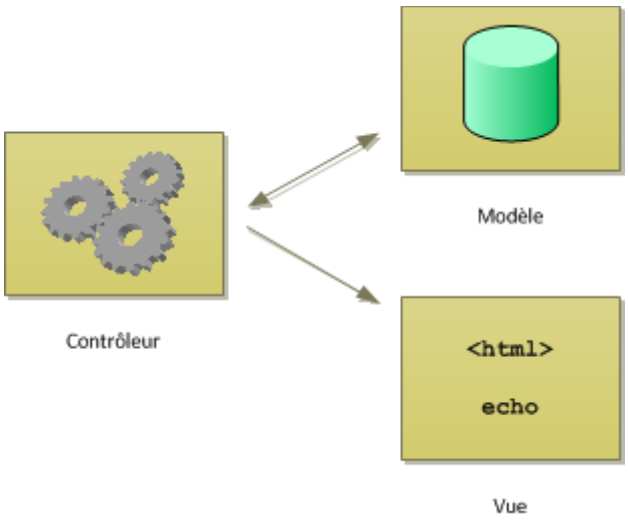
Un des plus célèbres design patterns s'appelle MVC (Modèle - Vue - Contrôleur). Le pattern MVC permet de bien organiser son code source. Il va nous aider à savoir quels fichiers créer, mais surtout à définir leur rôle. Le but de MVC est justement de séparer la logique du code en trois parties que l'on retrouve dans des fichiers distincts :

- Modèle : cette partie gère les données du site. Son rôle est d'aller récupérer les informations « brutes » dans la base de données, de les organiser et de les assembler pour qu'elles puissent ensuite être traitées par le contrôleur. On y trouve donc les requêtes SQL.

  Parfois, les données ne sont pas stockées dans une base de données. C'est plus rare, mais on peut être amené à aller chercher des données dans des fichiers. Dans ce cas, le rôle du modèle est de faire les opérations d'ouverture, de lecture et d'écriture de fichiers (fonctions fopen, fgets, etc.).
- Vue : cette partie se concentre sur l'affichage. Elle ne fait presque aucun calcul et se contente de récupérer des variables pour savoir ce qu'elle doit afficher. On y trouve essentiellement du code HTML mais aussi quelques boucles et conditions PHP très simples, pour afficher par exemple la liste des messages des forums.
- Contrôleur : cette partie gère la logique du code qui prend des décisions. C'est en quelque sorte l'intermédiaire entre le modèle et la vue : le contrôleur va demander au modèle les données, les analyser, prendre des décisions et renvoyer le texte à afficher à la vue. Le contrôleur contient exclusivement du PHP. C'est notamment lui qui détermine si le visiteur a le droit de voir la page ou non (gestion des droits d'accès).

La figure suivante schématise le rôle de chacun de ces éléments.

Il est important de bien comprendre comment ces éléments s'agencent et communiquent entre eux.



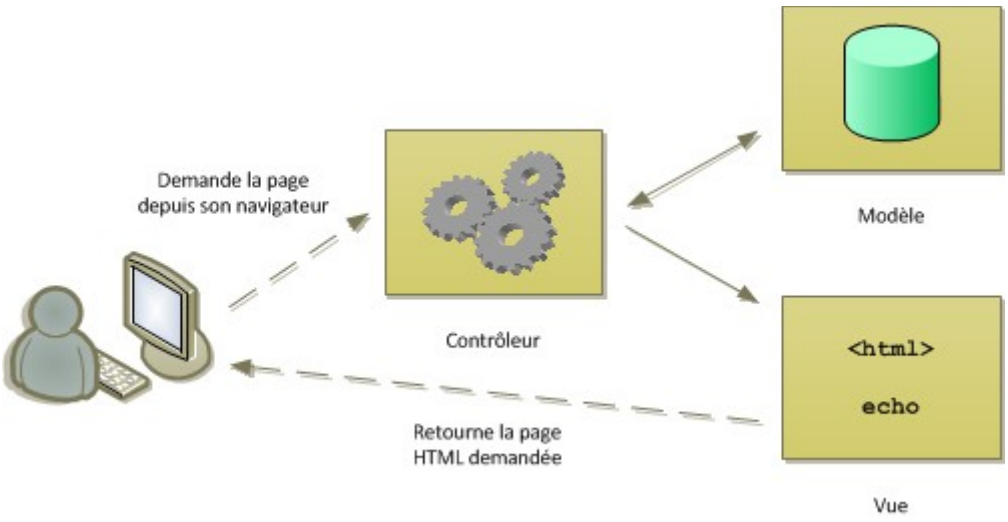
Il faut tout d'abord retenir que le contrôleur est le chef d'orchestre : c'est lui qui reçoit la requête du visiteur et qui contacte d'autres fichiers (le modèle et la vue) pour échanger des informations avec eux.

Le fichier du contrôleur demande les données au modèle sans se soucier de la façon dont celui-ci va les récupérer. Par exemple : « Donne-moi la liste des 30 derniers messages du forum no 5 ». Le modèle traduit cette demande en une requête SQL, récupère les informations et les renvoie au contrôleur.

Une fois les données récupérées, le contrôleur les transmet à la vue qui se chargera d'afficher la liste des messages.

Dans les cas les plus simples, le contrôleur sert seulement à faire la jonction entre le modèle et la vue. Mais le rôle du contrôleur ne se limite pas à cela : s'il y a des calculs ou des vérifications d'autorisations à faire, des images à miniaturiser, c'est lui qui s'en chargera.

Concrètement, le visiteur demandera la page au contrôleur et c'est la vue qui lui sera retournée, comme schématisé sur la figure suivante. Bien entendu, tout cela est transparent pour lui, il ne voit pas tout ce qui se passe sur le serveur.



La requête du client arrive au contrôleur et celui-ci lui retourne la vue.

2. Exemple de code

Le code ci-dessous présente un blog écrit en PHP de façon intuitive sans trop se soucier de son organisation. Résultat : le code est un joyeux mélange d'instructions PHP, de requêtes SQL et de HTML. Sur une page simple comme celle-là, cela ne pose pas trop de problèmes car il n'y a pas beaucoup de lignes de code. En revanche, si on veut ajouter plus tard des fonctionnalités et des requêtes SQL à cette page… cela va vite devenir un vrai capharnaüm !

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Mon blog</title>
        <link href="style.css" rel="stylesheet" />
    </head>

    <body>
        <h1>Mon blog !</h1>
        <p>Derniers billets du blog :</p>

<?php
// Connexion à la base de données
try
{
    $bdd = new PDO('mysql:host=localhost;dbname=test;charset=utf8', 'root', '');
}

catch(Exception $e)
{
        die('Erreur : '.$e->getMessage());
}

// On récupère les 5 derniers billets
$req = $bdd->query(
       'SELECT id, titre, contenu, DATE_FORMAT(date_creation, \'%d/%m/%Y à %Hh%imin%ss\')
       AS date_creation_fr
       FROM billets
       ORDER BY date_creation DESC
       LIMIT 0, 5');

while ($donnees = $req->fetch())
{
?>

<div class="news">
    <h3>
        <?php echo htmlspecialchars($donnees['titre']); ?>
        <em>le <?php echo $donnees['date_creation_fr']; ?></em>
    </h3>

    <p>

    <?php
    // On affiche le contenu du billet
    echo nl2br(htmlspecialchars($donnees['contenu']));
```

```
    ?>

    <br />

    <em>
       <a href="commentaires.php?billet=<?php echo $donnees['id']; ?>">Commentaires</a>
    </em>
    </p>
</div>

<?php
} // Fin de la boucle des billets
$req->closeCursor();
?>

</body>
</html>
```

Si on prend la peine de construire cette page en respectant l'architecture MVC, cela va prendre un peu plus de temps : il y aura probablement trois fichiers au lieu d'un seul, mais au final le code sera beaucoup plus clair et mieux découpé.

L'intérêt de respecter l'architecture MVC ne se voit pas forcément de suite. En revanche, si on modifie le site plus tard, on sera bien heureux d'avoir pris la peine de l'organiser avant ! De plus, si on travaille à plusieurs, cela permet de séparer les tâches : une personne peut s'occuper du contrôleur, une autre de la vue et la dernière du modèle. Si vous vous mettez d'accord au préalable sur les noms des variables et fonctions à appeler entre les fichiers, vous pouvez travailler ensemble en parallèle et avancer ainsi beaucoup plus vite !

3. Amélioration en respectant l'architecture MVC

On peut respecter l'architecture MVC de différentes manières ; tout dépend de la façon dont on l'interprète. D'ailleurs, la théorie « pure » de MVC est bien souvent inapplicable en pratique. Il faut faire des concessions, savoir être pragmatique : on prend les bonnes idées et on met de côté celles qui se révèlent trop contraignantes.

À la racine du site, on va créer trois répertoires :

- modele ;
- vue ;
- controleur.

Dans chacun d'eux, on va créer un sous-répertoire pour chaque « module » du site : forums, blog, minichat, etc. Pour le moment, créez un répertoire blog dans chacun de ces dossiers. On aura ainsi l'architecture suivante :

- modele/blog : contient les fichiers gérant l'accès à la base de données du blog ;
- vue/blog : contient les fichiers gérant l'affichage du blog ;
- controleur/blog : contient les fichiers contrôlant le fonctionnement global du blog.

On peut commencer par travailler sur l'élément que l'on veut ; il n'y a pas d'ordre particulier à suivre (on peut travailler à trois en parallèle sur chacun de ces éléments). On va commencer par le modèle, puis de voir le contrôleur et enfin la vue.

3.1. Le modèle

Créez un fichier get_billets.php dans modele/blog. Ce fichier contiendra une fonction dont le rôle sera de retourner un certain nombre de billets depuis la base de données. C'est tout ce qu'elle fera.

```
<?php
function get_billets($offset, $limit)
{
    global $bdd;

    $offset = (int) $offset;
    $limit = (int) $limit;

    $req = $bdd->prepare(
       'SELECT id, titre, contenu, DATE_FORMAT(date_creation, \'%d/%m/%Y à %Hh%imin%ss\')
       AS date_creation_fr
       FROM billets
       ORDER BY date_creation DESC
       LIMIT :offset, :limit');

    $req->bindParam(':offset', $offset, PDO::PARAM_INT);
    $req->bindParam(':limit', $limit, PDO::PARAM_INT);
    $req->execute();

    $billets = $req->fetchAll();

    return $billets;
}
```

Ce code source ne contient pas de réelles nouveautés. Il s'agit d'une fonction qui prend en paramètre un offset et une limite. Elle retourne le nombre de billets demandés à partir du billet nooffset. Ces paramètres sont transmis à MySQL via une requête préparée.

On utilise la fonction bindParam qui permet de préciser que le paramètre est un entier (PDO$\colon\colon$PARAM_INT). Cette méthode alternative est obligatoire dans le cas où les paramètres sont situés dans la clause LIMIT car il faut préciser qu'il s'agit d'entiers.

La connexion à la base de données aura été faite précédemment. On récupère l'objet $bdd global représentant la connexion à la base et on l'utilise pour effectuer notre requête SQL. Cela nous évite d'avoir à recréer une connexion à la base de données dans chaque fonction, ce qui serait très mauvais pour les performances du site (et très laid dans le code !).

Il existe une meilleure méthode pour récupérer l'objet $bdd et qui est basée sur le design pattern singleton. Elle consiste à créer une classe qui retourne toujours le même objet.

Plutôt que de faire une boucle de fetch(), on appelle ici la fonction fetchAll() qui assemble toutes les données dans un grand array. La fonction retourne donc un array contenant les billets demandés.

Vous noterez que ce fichier PHP ne contient pas la balise de fermeture ?>. Celle-ci n'est en effet pas obligatoire. Je vous recommande de ne pas l'écrire surtout dans le modèle et le contrôleur d'une architecture MVC. Cela permet d'éviter de fâcheux problèmes liés à l'envoi de HTML avant l'utilisation de fonctions comme setCookie qui nécessitent d'être appelées avant tout code HTML.

3.2. Le contrôleur

Le contrôleur est le chef d'orchestre de l'application. Il demande au modèle les données, les traite et appelle la vue qui utilisera ces données pour afficher la page.

Dans controleur/blog, créez un fichier index.php qui représentera la page d'accueil du blog.

```php
<?php
// On demande les 5 derniers billets (modèle)
include_once('modele/blog/get_billets.php');

$billets = get_billets(0, 5);

// On effectue du traitement sur les données (contrôleur)
// Ici, on doit surtout sécuriser l'affichage
foreach($billets as $cle => $billet)
{
    $billets[$cle]['titre'] = htmlspecialchars($billet['titre']);
    $billets[$cle]['contenu'] = nl2br(htmlspecialchars($billet['contenu']));
}

// On affiche la page (vue)
include_once('vue/blog/index.php');
```

Le rôle de la page d'accueil du blog est d'afficher les cinq derniers billets. On appelle donc la fonction get_billets() du modèle, on récupère la liste « brute » de ces billets que l'on traite dans un foreach pour protéger l'affichage avec htmlspecialchars() et créer les retours à la ligne du contenu avec nl2br().

S'il y avait d'autres opérations à faire avant l'appel de la vue, comme la gestion des droits d'accès, ce serait le bon moment. En l'occurrence, il n'est pas nécessaire d'effectuer d'autres opérations dans le cas présent.

Notez que la présence des fonctions de sécurisation des données dans le contrôleur est discutable. On pourrait laisser cette tâche à la vue, qui ferait donc les htmlspecialchars, ou bien à une couche intermédiaire entre le contrôleur et la vue (c'est d'ailleurs ce que proposent certains frameworks). Comme vous le voyez, il n'y a pas une seule bonne approche mais plusieurs, chacune ayant ses avantages et inconvénients.

Vous noterez qu'on opère sur les clés du tableau plutôt que sur $billet (sans s) directement. En effet, $billet est une copie du tableau $billets créée par le foreach. $billet n'existe qu'à l'intérieur du foreach, il est ensuite supprimé. Pour éviter les failles XSS, il faut agir sur le tableau utilisé à l'affichage, c'est-à-dire $billets.

Ce qui est intéressant, c'est que la fonction get_billets() pourra être réutilisée à d'autres occasions. Ici, on l'appelle toujours avec les mêmes paramètres, mais on pourrait en avoir besoin dans d'autres contrôleurs. On pourrait aussi améliorer ce contrôleur-ci pour gérer

la pagination des billets du blog et afficher un nombre différent de billets par page en fonction des préférences du visiteur.

3.3. La vue

Il ne reste plus qu'à créer la vue correspondant à la page d'accueil des derniers billets du blog. Le fichier de la vue devra simplement afficher le contenu de l'array $billets sans se soucier de la sécurité ou des requêtes SQL. Tout aura été préparé avant.

Vous pouvez créer un fichier index.php dans vue/blog et y insérer le code suivant :

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8" />
        <title>Mon blog</title>
      <link href="vue/blog/style.css" rel="stylesheet" />
    </head>

    <body>
        <h1>Mon blog !</h1>
        <p>Derniers billets du blog :</p>

<?php
foreach($billets as $billet)
{
?>
<div class="news">
    <h3>
        <?php echo $billet['titre']; ?>
        <em>le <?php echo $billet['date_creation_fr']; ?></em>
    </h3>

    <p>
    <?php echo $billet['contenu']; ?>
    <br />
    <em>
      <a href="commentaires.php?billet=<?php echo $billet['id']; ?>">Commentaires</a>
    </em>
    </p>
</div>
<?php
}
?>
</body>
</html>
```

Ce code source contient essentiellement du HTML et quelques morceaux de PHP pour afficher le contenu des variables et effectuer la boucle nécessaire.

L'intérêt est que ce fichier est débarrassé de toute « logique » du code : vous pouvez aisément le donner à un spécialiste de la mise en page web pour qu'il améliore la présentation. Celui-ci n'aura pas besoin de connaître le PHP pour travailler sur la mise en page du blog. Il suffit de lui expliquer le principe de la boucle et comment on affiche une variable.

3.4. Le contrôleur global du blog

Bien qu'en théorie ce ne soit pas obligatoire, je vous recommande de créer un fichier contrôleur « global » par module à la racine de votre site. Son rôle sera essentiellement de traiter les paramètres $_GET et d'appeler le contrôleur correspondant en fonction de la page demandée. On peut aussi profiter de ce point « central » pour créer la connexion à la base de données.

On a choisi d'inclure un fichier connexion_sql.php qui crée l'objet $bdd. Ce fichier pourra ainsi être partagé entre les différents modules du site.

Vous pouvez donc créer ce fichier blog.php à la racine du site :

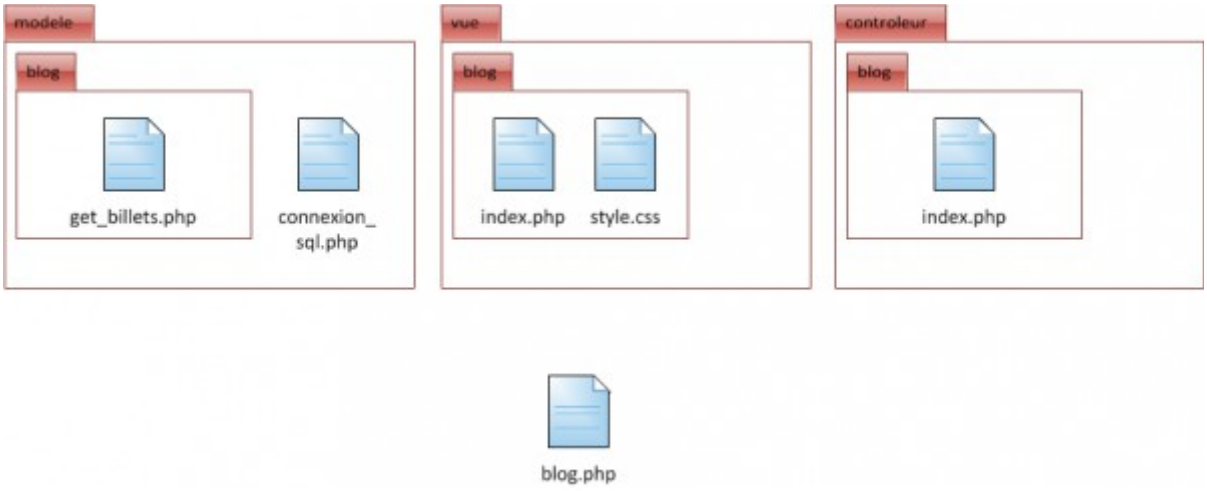
```
<?php
include_once('modele/connexion_sql.php');

if ( !isset($_GET['section']) OR $_GET['section'] == 'index' )
{
    include_once('controleur/blog/index.php');
}
```

Pour accéder à la page d'accueil du blog, il suffit maintenant d'ouvrir la page blog.php !

L'avantage de cette page est aussi qu'elle permet de masquer l'architecture du site au visiteur. Sans elle, ce dernier aurait dû aller sur la page controleur/blog/index.php pour afficher votre blog. Cela aurait pu marcher, mais aurait probablement créé de la confusion chez les visiteurs, en plus de complexifier inutilement l'URL.

3.5. Résumé des fichiers créés

Voici l'architecture des fichiers créés pour adapter le blog en MVC :



Cela fait plus de fichiers qu'auparavant. La construction du site selon une architecture MVC est à ce prix. Cela multiplie les fichiers, mais clarifie dans le même temps leur rôle.

Désormais, si vous avez un problème de requête SQL, vous savez précisément quel fichier ouvrir : le modèle correspondant. Si c'est un problème d'affichage, vous ouvrirez la vue. Tout cela rend le site plus facile à maintenir et à faire évoluer !

4. Les frameworks MVC

L'organisation en fichiers proposée n'est qu'une façon de faire parmi beaucoup d'autres. L'idéal serait de créer le site en se basant sur un framework PHP de qualité (mais cela demandera du temps, car il faut apprendre à se servir du framework en question !).

Un framework est un ensemble de bibliothèques, une sorte de kit prêt à l'emploi pour créer plus rapidement un site web, tout en respectant des règles de qualité.

Voici quelques frameworks PHP célèbres :

- CodeIgniter ;
- CakePHP ;
- Symfony ;
- Jelix ;
- Zend Framework.

Ils sont tous basés sur une architecture MVC et proposent en outre de nombreux outils pour faciliter le développement d'un site web.

Ces frameworks ont prouvé leur robustesse et leur sérieux, ce qui en fait des outils de choix pour concevoir des sites de qualité professionnelle.

5. Différence avec l'architecture trois tiers

L'architecture trois tiers est un modèle en couches, c'est-à-dire, que chaque couche communique seulement avec ses couches adjacentes (supérieures et inférieures) et le flux de contrôle traverse le système de haut en bas; les couches supérieures contrôlent les couches inférieures, c'est-à-dire, que les couches supérieures sont toujours sources d'interaction (clients) alors que les couches inférieures ne font que répondre à des requêtes (serveurs).

Dans le modèle MVC, il est généralement admis que la vue puisse consulter directement le modèle (lecture) sans passer par le contrôleur. Par contre, elle doit nécessairement passer par le contrôleur pour effectuer une modification (écriture). Ici, le flux de contrôle est inversé par rapport au modèle en couches, le contrôleur peut alors envoyer des requêtes à toutes les vues de manière qu'elles se mettent à jour.

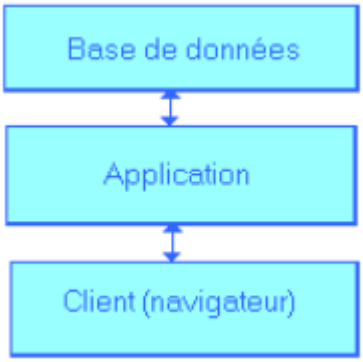
Dans l'architecture trois tiers, si une vue modifie les données, toutes les vues concernées par la modification doivent être mises à jour, d'où l'utilité de l'utilisation du MVC au niveau de la couche de présentation. La couche de présentation permet donc d'établir des règles du type « mettre à jour les vues concernant X si Y ou Z sont modifiés ». Mais ces règles deviennent rapidement trop nombreuses et ingérables si les relations logiques sont trop élevées. Dans ce cas, un simple rafraîchissement des vues à intervalle régulier permet de surmonter aisément ce problème. Il s'agit d'ailleurs de la solution la plus répandue en architecture trois tiers, l'utilisation du MVC étant moderne et encore marginale.

5.1. Définition et concepts

Son nom provient de l'anglais tier signifiant étage ou niveau. Il s'agit d'un modèle logique d'architecture applicative qui vise à modéliser une application comme un empilement de

trois couches logicielles (étages, niveaux, tiers ou strates) dont le rôle est clairement défini :

- la présentation des données : correspondant à l'affichage, la restitution sur le poste de travail, le dialogue avec l'utilisateur ;
- le traitement métier des données : correspondant à la mise en œuvre de l'ensemble des règles de gestion et de la logique applicative ;
- et enfin l'accès aux données persistantes : correspondant aux données qui sont destinées à être conservées sur la durée, voire de manière définitive.



Dans cette approche, les couches communiquent entre elles au travers d'un « modèle d'échange », et chacune d'entre elles propose un ensemble de services rendus. Les services d'une couche sont mis à disposition de la couche supérieure. On s'interdit par conséquent qu'une couche invoque les services d'une couche plus basse que la couche immédiatement inférieure ou plus haute que la couche immédiatement supérieure (chaque couche ne communique qu'avec ses voisins immédiats).

Le rôle de chacune des couches et leur interface de communication étant bien définis, les fonctionnalités de chacune d'entre elles peuvent évoluer sans induire de changement dans les autres couches. Cependant, une nouvelle fonctionnalité de l'application peut avoir des répercussions dans plusieurs d'entre elles. Il est donc essentiel de définir un modèle d'échange assez souple, pour permettre une maintenance aisée de l'application.

Ce modèle d'architecture 3-tiers a pour objectif de répondre aux préoccupations suivantes :

- allègement du poste de travail client (notamment vis-à-vis des architectures classiques client-serveur de données – typiques des applications dans un contexte Oracle/Unix) ;
- prise en compte de l'hétérogénéité des plates-formes (serveurs, clients, langages, etc.) ;
- introduction de clients dits « légers » (plus liée aux technologies Intranet/HTML qu'au 3-tiers proprement dit) ;
- amélioration de la sécurité des données, en supprimant le lien entre le client et les données. Le serveur a pour tâche, en plus des traitements purement métiers, de vérifier l'intégrité et la validité des données avant de les envoyer dans la couche de données.
- rupture du lien de propriété exclusive entre application et données. Dans ce modèle, la base de données peut être plus facilement normalisée et intégrée à un entrepôt de données.
- et enfin, meilleure répartition de la charge entre différents serveurs d'application.

Précédemment, dans les architectures client-serveur classiques, les couches présentation et traitement étaient trop souvent imbriquées. Ce qui posait des problèmes à chaque fois que l'on voulait modifier l'interface homme-machine du système.

5.2. Les trois couches

5.2.1. Couche présentation (premier niveau)

Elle correspond à la partie de l'application visible et interactive avec les utilisateurs. On parle d'interface homme machine. En informatique, elle peut être réalisée par une application graphique ou textuelle (WPF). Elle peut aussi être représentée en HTML pour être exploitée par un navigateur web ou en WML pour être utilisée par un téléphone portable.

On conçoit facilement que cette interface peut prendre de multiples facettes sans changer la finalité de l'application. Dans le cas d'un système de distributeurs de billets, l'automate peut être différent d'une banque à l'autre, mais les fonctionnalités offertes sont similaires et les services identiques (fournir des billets, donner un extrait de compte, etc.).

Toujours dans le secteur bancaire, une même fonctionnalité métier (par exemple, la commande d'un nouveau chéquier) pourra prendre différentes formes de présentation selon qu'elle se déroule sur Internet, sur un distributeur automatique de billets ou sur l'écran d'un chargé de clientèle en agence.

La couche présentation relaie les requêtes de l'utilisateur à destination de la couche métier, et en retour lui présente les informations renvoyées par les traitements de cette couche. Il s'agit donc ici d'un assemblage de services métiers et applicatifs offerts par la couche inférieure.

5.2.2. Couche métier ou business (second niveau)

Elle correspond à la partie fonctionnelle de l'application, celle qui implémente la « logique », et qui décrit les opérations que l'application opère sur les données en fonction des requêtes des utilisateurs, effectuées au travers de la couche présentation.

Les différentes règles de gestion et de contrôle du système sont mises en œuvre dans cette couche.

La couche métier offre des services applicatifs et métier2 à la couche présentation. Pour fournir ces services, elle s'appuie, le cas échéant, sur les données du système, accessibles au travers des services de la couche inférieure. En retour, elle renvoie à la couche présentation les résultats qu'elle a calculés.

5.2.3. Couche accès aux données (troisième niveau)

Elle consiste en la partie gérant l'accès aux données du système. Ces données peuvent être propres au système, ou gérées par un autre système. La couche métier n'a pas à s'adapter à ces deux cas, ils sont transparents pour elle, et elle accède aux données de manière uniforme (couplage faible).