Architecture d'un site web

Alexandre Niveau

GREYC — Université de Caen

En partie adapté du cours de Jean-Marc Lecarpentier

Séparation traitement des données & affichage

- Objectifs : factorisation, modularisation, séparation des responsabilités

  ➤ faciliter la maintenance et les évolutions futures
- Séparer au maximum le PHP du HTML
- PHP génère le contenu **sans l'afficher**
- Fichier HTML ne contient que des instructions `echo` pour afficher le contenu

Exemple simplissime

Page PHP : `hello.php`

```php
<?php
$titre = "Une page PHP simple";
$info = "Bonjour le monde !";

include("squelette.php");
?>
```

Squelette : `squelette.php`

```php
<!DOCTYPE html>
<html lang="fr">
<head>
  <title><?php echo $titre; ?></title>
</head>
<body>
<h1><?php echo $titre; ?></h1>
<p>L'information à délivrer est simple :</p>
<div>
  <?php echo $info; ?>
</div>
<p>Voilà.</p>
</body>
</html>
```

Résultat [demo1/hello.php]

Exemple à peine moins simple

Page PHP : `hello.php`

```php
<?php
if (key_exists('toto', $_GET)) {
  $titre = "Une page sur toto";
  $info = "toto est une variable " .
    "métasyntaxique utilisée " .
    "dans les exemples de programmes.";
} else {
  $titre = "Une page PHP simple";
  $info = "Bonjour le monde !";
}
$menu =
  '<a href="hello.php">Accueil</a> |' .
  ' <a href="hello.php?toto">toto</a>';

include("squelette.php");
?>
```

Squelette : `squelette.php`

```php
<!DOCTYPE html>
<html lang="fr">
<head>
  <title><?php echo $titre; ?></title>
</head>
<body>
<nav>Navigation : <?php echo $menu; ?></nav>
<h1><?php echo $titre; ?></h1>
<p>L'information à délivrer est simple :</p>
<div>
  <?php echo $info; ?>
</div>
<p>Voilà.</p>
</body>
</html>
```

Résultat [demo2/hello.php]

Avantages de cette architecture

- Le script PHP gère toute la logique à un seul endroit, pas au milieu du code HTML
  - plus facile à maintenir
  - plus propre, notamment aucun risque d'erreur *headers already sent* si on manipule des en-têtes HTTP (cookies, redirections…)
- Squelette : fichier avec du HTML « à trous » (*template*), facile à maintenir séparément (potentiellement par une personne différente)
- Un problème de notre exemple ? C'est le script PHP qui décide comment afficher le menu

Gestion du menu

- L'affichage du menu doit être géré par le squelette…

- … mais on ne peut pas simplement tout passer de l'autre côté, car le nom des paramètres GET sera dupliqué
- C'est le PHP qui gère l'analyse des URL, il doit donc garder la main sur leur forme
- Solution : le script PHP donne un tableau avec les liens au squelette, qui les affiche comme il l'entend

Exemple avec gestion du menu

Page PHP : `hello.php`

```php
<?php
if (key_exists('toto', $_GET)) {
  $titre = "Une page sur toto";
  $info = "toto est une variable " .
    "métasyntaxique utilisée " .
    "dans les exemples de programmes.";
} else {
  $titre = "Une page PHP simple";
  $info = "Bonjour le monde !";
}
$menu = array(
  "Accueil" => "hello.php",
  "toto" => "hello.php?toto",
);

include("squelette.php");
?>
```

Squelette : `squelette.php`

```php
<!DOCTYPE html>
<html lang="fr">
<head>
  <title><?php echo $titre; ?></title>
</head>
<body>
<nav>Navigation : <ul>
<?php
foreach ($menu as $texte => $lien) {
  echo "<li>";
  echo "<a href=\"$lien\">$texte</a>";
  echo "</li>\n";
} ?>
</ul></nav>
<h1><?php echo $titre; ?></h1>
<p>L'information à délivrer est simple :</p>
<div>
  <?php echo $info; ?>
</div>
<p>Voilà.</p>
</body>
</html>
```

Résultat [demo3/hello.php]

Modularité

- Le script ne transmet *que* l'information nécessaire au squelette, il ne fait aucun choix d'affichage
- Le squelette n'a pas besoin de connaître le nombre de liens à créer, ni la forme des liens
- Rajouter une nouvelle page sur le même modèle ne nécessite que de rajouter les contenus dans le script PHP
- C'est ce type de *modularité* que l'on recherche, *a fortiori* pour des applications complexes

Problématique

- Application moins triviale, où l'internaute peut faire des actions autres qu'afficher telle ou telle page
- On va préciser notre organisation modulaire
- Il faut séparer :
  - le fonctionnement de l'application
  - les affichages qui sont produits
  - le pilotage de l'application
- On va utiliser une architecture inspirée de MVC

Architecture MVC

- MVC : modèle-vue-contrôleur
- De façon générale :
  - le modèle décrit le fonctionnement de l'application
  - la vue est l'affichage produit en fonction de l'état du modèle
  - le contrôleur capte les actions de l'utilisateur et les transmet au modèle
- À partir de là : dizaines de variantes et de conceptions différentes
- Objectif : modularité, séparation des composants
- Règle générale : faire en sorte de pouvoir au maximum modifier chaque composant indépendamment
- Principes indépendants de l'utilisation de la programmation objet !

MVC sur le web ?

- Notre architecture est destinée à être utilisée dans le cadre du web

  ➤ choix techniques spécifiques
- Contrairement aux interfaces graphiques habituelles, sur le web les vues ne permettent pas d'interaction en temps réel
  - les vues permettent de faire des actions…

    - … mais qui doivent passer par une requête HTTP
    - requête HTTP = nouvelle page : on exécute à nouveau tout le programme depuis le début

      ➤ le contrôleur ne peut pas « écouter » la vue, ni la vue « écouter » le modèle !

« MVCR »

- Une autre différence essentielle est la nature de l'interaction :
    - le serveur ne sait pas où l'internaute a cliqué
    - il voit seulement ce qu'il y a dans la requête HTTP

      ➤ concrètement, une action est constituée
        - d'une méthode HTTP (GET, POST, ou autre)
        - d'une URL (paramétrée ou non)
        - éventuellement, de données dans le corps de la requête (si POST)

  ➤ On va utiliser un *routeur* : quatrième composant, qui s'occupe d'analyser la requête HTTP pour décider quoi faire (c'est-à-dire quelle méthode du contrôleur doit être exécutée)
- On parlera dans ce cours de « MVCR » pour insister sur les différences
- L'architecture que l'on va utiliser est une implémentation possible parmi d'autres

Routeur

- Le routeur (ou *dispatcher*) est le point d'entrée de l'application
- C'est le fichier principal qui va être appelé par l'internaute
- Les actions de l'internaute sont transmises par des clics sur des liens ou des soumissions de formulaires

  ➤ Le travail du routeur est donc d'analyser le contenu de la requête HTTP :
    - méthode utilisée
    - URL
    - contenu des tableaux `$_GET` et `$_POST`
- En fonction des choix de l'internaute :
    - il choisit le contrôleur et la vue à utiliser
    - il en crée des instances (les précédentes ont disparu, puisqu'on recommence le programme depuis le début à chaque requête)
    - il appelle la bonne méthode du contrôleur, en lui passant les bons paramètres
    - à la fin, il affiche la vue
- Il fait l'interface entre l'action de l'internaute et le contrôleur : en quelque sorte

il remplace le *listener* entre le contrôleur et la vue dans le MVC habituel

- Il ne connaît pas le modèle, et il ne « sait » pas ce qui se passe dans les contrôleurs et les vues

Contrôleur

- Comme dans le cadre classique, le contrôleur est le composant qui *effectue* les actions demandées
- En fonction des choix de l'internaute, qui ont été transmis par le routeur :
  - il met à jour le modèle
  - il appelle la bonne méthode de la vue, avec en paramètre des éléments du modèle
- Il fait le lien entre modèle et vue, mais n'utilise pas le routeur.

Modèle

- Le modèle est le cœur de l'application
- Données, algorithmes, traitements… *logique métier* de l'application
- L'idée est que le site web n'est qu'une interface possible : on veut pouvoir réutiliser le modèle dans un autre contexte
- Par exemple le modèle pourrait être manipulé en ligne de commande, ou alors par un programme plutôt que par un humain

  ➤ le modèle ne connaît ni le routeur, ni le contrôleur, ni la vue

Vue

- La vue correspond à l'affichage
- Elle utilise l'état du modèle pour générer du HTML en fonctions des demandes du contrôleur
- Elle ne doit pas manipuler le modèle : c'est le travail du contrôleur !

  ➤ elle se contente d'afficher son état courant
- La vue ne connaît pas le contrôleur, elle se contente de lui obéir
- Elle construit des liens et des formulaires pour que l'internaute puisse effectuer des actions

  ➤ elle a donc besoin de savoir comment le routeur interprète les requêtes HTTP :
  - quelle URL pour aller à telle page ?
  - quel nom pour les clefs dans les données ?
  - méthode GET ou POST dans le formulaire ?
- Pour cela, elle va interroger le routeur, qui va lui donner ces informations

Détails sur les choix de conception

- Le modèle sera généralement un ensemble de classes autonomes, dont certaines correspondront plus ou moins directement aux objets stockés en BD (attributs = champs de la table)
- Une vue = un squelette + une classe dont les attributs sont les « trous » du squelette, avec des méthodes pour remplir les trous de diverses manières, et une méthode `render` qui affiche le HTML final
- Le routeur contient principalement un switch (ou plusieurs) qui branche sur le contenu de l'URL pour appeler une méthode d'un contrôleur
- Chaque méthode d'un contrôleur va
  - exécuter du code sur le modèle (par exemple créer un nouvel objet, etc.)
  - appeler une méthode de construction de la vue
- Finalement, le routeur appelle la méthode `render` de la vue.

Modèle et BD

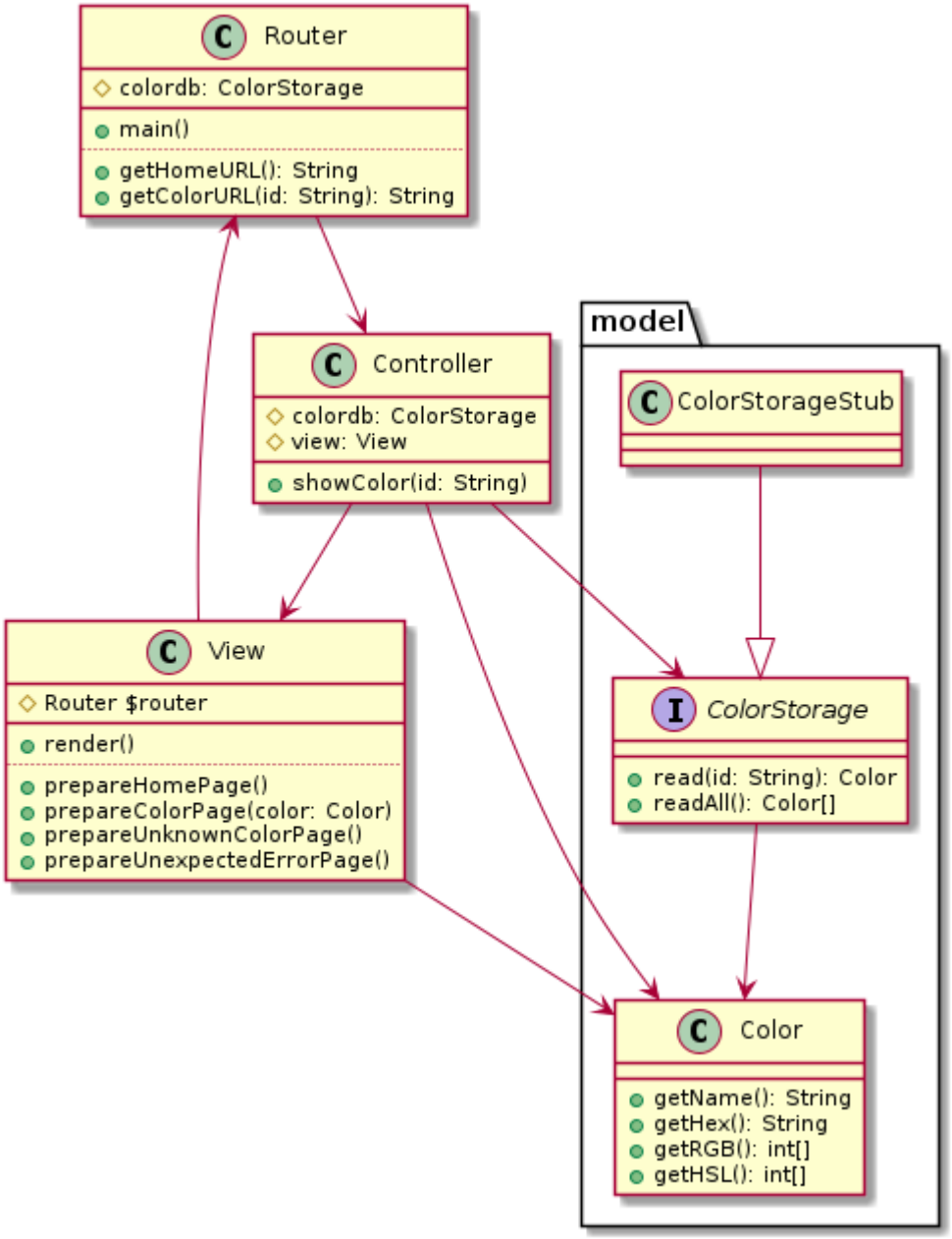
- Si on vise la modularité, il est important que le modèle soit indépendant du mode de stockage utilisé
- Divers SGBD, mais aussi modes de stockage autre que BD : fichiers, XML, etc.
- Le contrôleur va manipuler des *interfaces* de stockage, que l'on pourra implémenter de diverses façons sans toucher le reste du code

Modèle et BD

- Si on vise la modularité, il est important que le modèle soit indépendant du mode de stockage utilisé
- Divers SGBD, mais aussi modes de stockage autre que BD : fichiers, XML, etc.
- Le contrôleur va manipuler des *interfaces* de stockage, que l'on pourra implémenter de diverses façons sans toucher le reste du code

Démo

- Construction d'un site web sur des couleurs



[class_diagram.png]