Serveur Web, middleware côté serveur, MVC et Framework Symfony

Télécom SudParis

(19/09/2016)

CSC4101 - séance 2

Table des matières

1 Préface

1.1 Rappel séance précédente

— HTTP (1.1)
— Utiliser HTTP pour faire communiquer client et serveur Web
— Outils (inspecteur réseau navigateur, `curl`)

> Les concepts sur HTTP, le client-serveur ont été vus dans la séance précédente

1.2 Objectifs de cette séance

— Comprendre l'architecture de composants logiciels utilisés pour faire fonctionner des applications sur un serveur Web
— **Comment s'y prendre pour développer**
— Prise en main de l'environnement Symfony
— Programmer une application en PHP dans Symfony

> On met particulièrement en évidence le fait qu'on se préoccupe du développement d'applications, et qu'on prend directement le parti de sauter dans le grand bain avec un *framework*, sans passer par toutes les étapes des CGIs, du PHP traditionnel, ...

2 Architecture

Principes de conception des applications

> On va présenter les grandes architectures dans un ordre chronologique d'apparition des technologies :
>
> 1. pages statiques
> 2. applications BD + Web
> 3. multi-couches

2.1 Serveur de pages Web statiques

Historiquement, les premiers serveurs Web se contentent de fournir des pages Web **statiques** chargées depuis des fichiers (documents HTML).

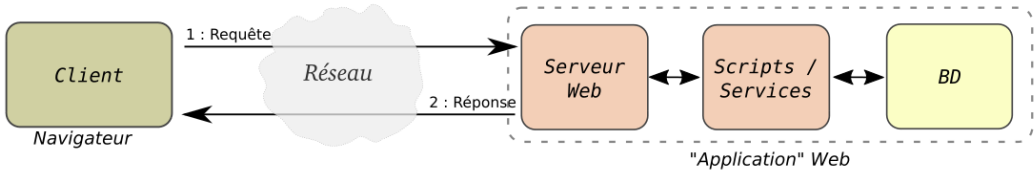
2.1.1 Caractéristiques d'un serveur de pages statiques

1. Avantages
   — Efficace (tient la charge)
2. Inconvénients
   — Ne permet pas facilement des pages qui se mettent à jour dynamiquement, en fonction des visites
   — Encore moins des *applications* interactives
   — Problème de cohérence des URLs (renommages, liens cassés, etc.)

> Le problème de cohérence des URLs vient du fait qu'un renommage du nom d'une page, ou son déplacement dans l'arborescence du système de fichiers sur le serveur, nécessitera de re-modifier le contenu de tous les fichiers des pages qui pointent vers celle-ci : l'arborescence logique des pages via les liens hypertextes est fortement liée à l'arborescence physique. Des solutions de configuration ou de compilateur de sites permettent de contourner ce problème, mais c'est peu confortable.

2.2 Applications BD + Web

Les serveurs servent à faire tourner des applications produisant des pages Web **dynamiques**, à partir de données issues de bases de données.



> BD : Bases de Données. Typiquement via l'interrogation d'un SGBDR
> ***Note** : on ne revient pas sur les technologies de bases de données dans ce cours, que l'on considère comme acquises (programme première année).*
> Les programmes qui produisent les pages peuvent aussi utiliser d'autres sources de données que les bases de données

2.2.1 Caractéristiques d'un serveur pour applications BD + Web

Age d'or du Web : des **applications** sont possibles : des programmes génèrent des pages Web en fonction des requêtes du client

1. Avantages
   — Base de données gère l'intégrité des données (transactions, accès multiples, intégrité référentielle, ...)

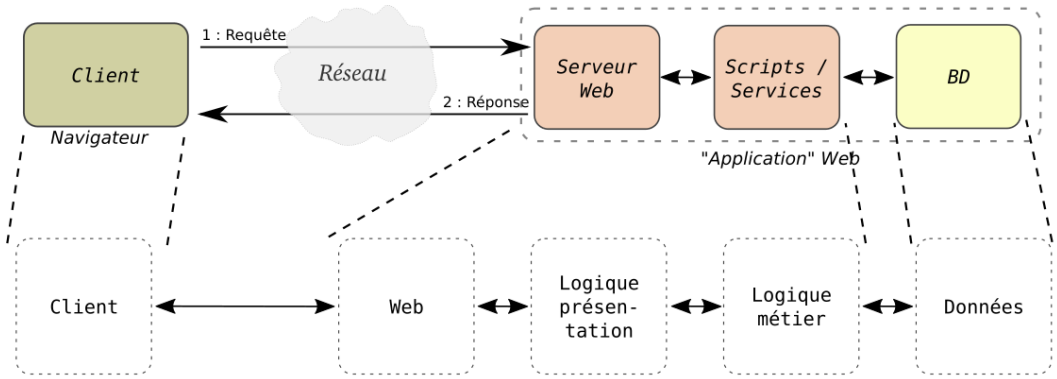
— Tient la charge : dupliquer l'exécution des scripts
— Large gamme de choix du langage de programmation

2. Inconvénients
— Invocation des scripts par le serveur
— Difficulté à programmer (+/-)

On ne raffine pas trop le modèle de couches avec l'adapteur de données qui est parfois introduit, pour ne pas trop complexifier
On mentionne des *scripts*, car les langages de scripts (PHP, Perl, Python, etc.) furent très populaire dans l'essor de cette technologie de pages Web dynamiques. Des programmes dans un langage compilé sont aussi possibles (Java par ex.) sans que cela change le modèle.
Différentes techniques d'invocation des programmes par les serveurs Webs (qui étaient initialement dédiés aux pages statiques) ont vu le jour. Un soucis général est les performances pour des consultations simultanées, sachant que tout n'est pas complètement dynamique dans une page Web, mais que si tout est regénéré à chaque consultation, le serveur peut vite s'écrouler. L'efficacité du mécanisme de déclencement de l'exécution du bon programme, et de récupération du contenu des pages généré est critique. Afin d'optimiser les performances, différentes générations de technologies ont émergé, qu'on verra plus loin.

2.3 Architecture multi-couches

On distingue les différents composants mis en œuvre dans une application, en différentes couches indépendantes.



Chaque couche s'appuie sur la couche de droite, les flèches illustrant le flux de données
On décrit cette architecture en parlant aussi d'architecture *multi-tier* ou *n-tier*.

2.3.1 Caractéristiques

— **Découpage en couches**
— Mieux décomposer l'architecture de l'application côté serveur pour mieux maîtriser le développement
— Découpler par exemple :
— la **logique métier** (pas nécessairement Web)
— la **présentation** (sous forme de pages Web)
— Modèle qui peut être raffiné en ajoutant d'autres couches

Cette approche d'architecture multi-couche n'est pas réservé aux applications Web, mais s'applique à d'autres types d'applications.
L'important est la modularisation, le découplage, qui facilite la maintenance, notamment l'évolutivité.
Un autre avantage consiste aussi à permettre de mieux gérer certains problèmes de performances, en scindant les composants d'une couche sur plusieurs exécutions en parallèle, ce qui multiplie les capacités de traitement (pourvu que la cohérence n'en souffre pas, ce qui n'est pas toujours possible).

3 Fonctionnement des serveurs Web

3.1 Serveur HTTP

1. Comprendre les requêtes HTTP des clients
   — URL
   — Verbe/méthode/action
   — En-têtes
   — Données transmises
2. Construire la réponse
   — Servir des documents
   — Ou **exécuter des programmes**
3. Renvoyer une réponse au client (HTML, etc.)

On ne détaillera pas ici la façon dont le serveur Web sert des données statiques depuis des documents, mais on se concentrera plutôt sur l'invocation des programmes, qui vont faire fonctionner les applications Web.

3.1.1 Mais aussi

— Vérifier la sécurité
— Gérer les performances
— ...

3.1.2 Exemples

— Apache
— Nginx
— PaaS (*Platform as a Service*) prêt à l'emploi sur un *Cloud*

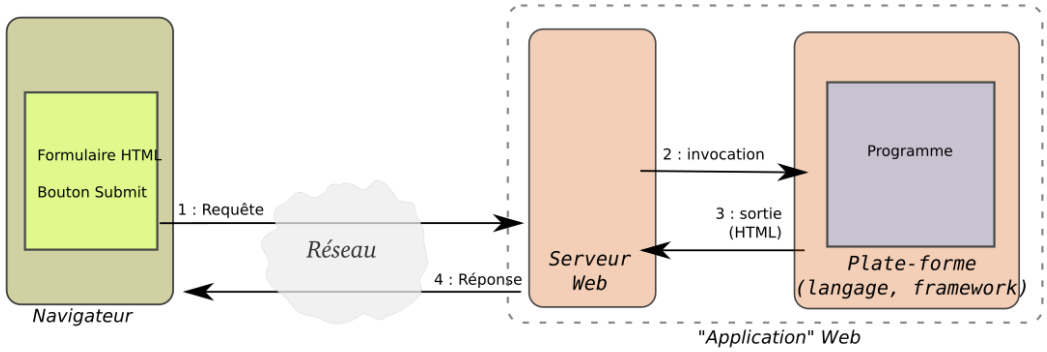
Dans le cours, on s'appuiera globalement sur le serveur fourni par php avec `php -S` y compris quand utilisé via environnement de tests de Symfony.

3.2 Programmes / Applications

— Exécution :
   — **Directe** sur le système d'exploitation
   ou
   — Dans le contexte d'un **serveur d'applications**
      — Un module du serveur Web
      — Un serveur d'application dédié

3.2.1 Invocation du programme

— Le serveur Web invoque l'exécution d'un programme
— Le programme s'exécute sur une "plate-forme" : langage, bibliothèques, moteurs d'exécution, ...
— Le programme fournit une sortie au format HTML (en général) : la page Web est transmise telle-quelle au client



Lien avec "*Platform as a Service*" (PaaS) : la plate-forme de développement et de déploiement est gérée de façon cohérente comme un tout intégré, disponible sur le Cloud. `https://fr.wikipedia.org/wiki/Plate-forme_en_tant_que_service`
Les plate-formes Cloud liées au développement d'applications ne sont pas détaillées dans le cadre de ce cours.

3.3 Technologies d'invocation d'un programme

— Différentes façons d'appeler un programme
— Globalement la même architecture :

1. CGI (*Common Gateway Interface*)
2. Exécution de programmes "au sein" du serveur HTTP
3. Serveur d'applications séparé

3.3.1 1. CGI (*Common Gateway Interface*)

— Requête sur une URL invoque une exécution d'un **processus** sur l'OS : shell script, exécutable compilé, script (Perl, Python, ...)
— Point d'entrée du programme unique : programmation impérative "classique"
— Problèmes :
  — **lourdeur** de l'invocation d'un nouveau processus système à chaque requête
  — gestion de session difficile
  — sécurité : celle de l'OS

Obsolète

Exemple de problème de sécurité : les processus des programmes s'exécutent tous dans le contexte du même utilisateur système, celui du serveur Web. Ainsi, une faille sur un des programmes permettant d'avoir, par exemple, accès au système de fichier, donnera à l'attaquant la maîtrise de tous les autres programmes et leurs données. Même si les fichiers "système" du serveur sont à l'abri (le serveur Web ne tourne pas en tant que `root`), les dégats peuvent quand même être conséquents.

3.3.2 2. Exécution de programmes "au sein" du serveur HTTP

— Le serveur HTTP "intègre" des fonctionnalités pour développer des applications (via des "*plugins*" / modules optionnels) :
  — Langage de programmation "embarqué" (**PHP**, Python, ...)
  — Gestion "native" de HTTP
  — Génération "facile" de HTML, etc.
— Exécution dans des *threads* dédiées (plus efficace que des processus à invoquer)
— Transfert des données immédiat : en mémoire, sans passer par l'OS

Ce type de technologies est toujours utilisé même s'il n'est pas idéal notamment en terme de souplesse pour le déploiement, pour la gestion avancée des performances ou de la sécurité.
Historiquement, pour PHP, on utilisait par exemple beaucoup `mod_php` qui est un *plugin* pour Apache, qui rendait l'interpréteur PHP et ses bibliothèques accessibles en direct.

3.3.3 3. Serveur d'applications séparé

- Architecture un peu différente
- Le serveur Web gère la charge HTTP et fournit des documents statiques (CSS, images, etc.)
- Un (ou plusieurs) serveur(s) d'applications gère(nt) l'exécution des réponses dynamiques aux requêtes
- Le serveur HTTP et les serveurs d'application sont connectés de façon efficace (si possible)
- Le serveur d'application gère des sessions
- On peut gérer de façon indépendante la partie statique et la partie dynamique, notamment pour s'adapter à la charge plus ou moins dynamiquement.

Aujourd'hui, dans le monde PHP, avec `PHP-FPM`, qui devient très répandu, on se rapproche un peu plus de ce modèle : serveur Web + serveur d'exécution PHP bien distincts (Apache + PHP-FPM ou NginX + PHP-FPM).
PHP-FPM implémente l'API FastGCI pour la communication entre le serveur Web et le serveur d'applications.

3.3.4 TODO Exemple : JSP NOEXPORT

- Programmation Java
  - HTML + balises dédiées -> Code Java
- Moteur de *servlets* (applications JAVA exécutées sur un serveur)
  - Apache Tomcat par ex.
- Processus de développement "qualité"

Sera vu en module d'ouverture par certains

3.4 Exemple : PHP

- Bibliothèques pour spécificités HTTP
- Facile à tester (?)
- Très populaire pour déploiement chez des hébergeurs
- Très permissif sur le style de programmation
- Conserve des mécanismes très proches de la façon d'écrire les programmes en scripts CGI
- Serveur d'exécution PHP dédié (PHP-FPM)

C'est le langage qu'on va utiliser dans ce cours.
On utilisera la plate-forme PHP fortement : on exécutera un serveur Web basique, directement dans PHP, car on sera dans l'environnement de développement et de tests fourni par défaut par Symfony.
Symfony permettra de faire mieux que certains mécanismes "archaiques" de PHP, pour monter en qualité.

3.5 Contexte d'invocation

- Données transmises au serveur Web via la requête HTTP
  - URL
    - Arguments (*URL-encoded*)
  - En-têtes
    - *Cookies*
  - Données du contenu de la requête (ex. données de formulaires pour un `POST`)
- Le choix du programme à invoquer est déterminé par une combinaison d'éléments (a minima via le chemin dans l'URL)
  - La configuration du serveur détermine les règles à appliquer
- Le serveur Web peut filtrer les données reçues dans la requête
- Il invoque un "processus" ou l'exécution d'une fonction sur l'API d'un module
- Il transmet les données au programme
  - arguments d'un processus (CGI)
  - variables d'environnement (CGI)
  - paramètres des fonctions des APIs, ou variables du langage de programmation (module intégré)

Quelque soit le langage, la plate-forme, on en revient aux données transmises au serveur via HTTP qui sont en général accessibles de façon plus ou moins facile à récupérer, via les outils du langage ou les bibliothèques.
Par exemple, en parcourant un tableau contenant des variables globales du programme (héritage historique de l'utilisation de variables d'environnement Unix au temps des CGI).

3.6 Résultats d'exécution

- Résultat d'un programme (POSIX)
  - Code de retour système (exécution d'un processus) :
    - `0` ou `!= 0`
  - Sortie standard du programme
  - Erreur standard ?
- Conversion en réponse HTTP
  - Code de retour -> Code de statut
    - `0` -> `200`
    - `!=0` -> `5xx`
  - Sorties
    - Standard (`stdout`) telle-quelle : **attention au format**
      - *Headers* : succès, échecs, redirections, ...
      - *Cookies* : sessions
      - *Ligne vide*
      - *Contenu du document* :
      - HTML
      - Autres formats (APIs : JSON, ...)
    - Erreur standard (`stderr`) : dans les logs du serveur ?

La plate-forme du langage peut intégrer la gestion de la génération de la réponse, sous forme de code de statut HTTP et de messages.
Attention à la façon dont est générée la sortie du programme. Si on mélange l'affichage des en-têtes et du message sans respecter le format attendu, les en-têtes seront ignorés.
Le PHP "basique" pose un certain nombre de problèmes de ce type, qui seront résolus avec un *framework* comme Symfony

4 Le programme en lui-même

4.1 Étapes

1. Récupération d'arguments (décodés depuis la requête HTTP)
   - ASCII -> objets d'un langage de programmation
   - facilité par le langage et les bibliothèques
2. Traitements
3. Code de retour
4. Affichages ?
   - en-têtes,
   - HTML,
   - etc.

Les traitements peuvent concerner n'importe quel type de contexte d'application, et pas nécessairement ce qu'on détaillera dans le cours.
Même si on illustrera ce cours par des développements sur des applications assez classiques de gestion de données présentes dans une base de données relationelle, les technos Web peuvent servir à faire plein d'autres choses.

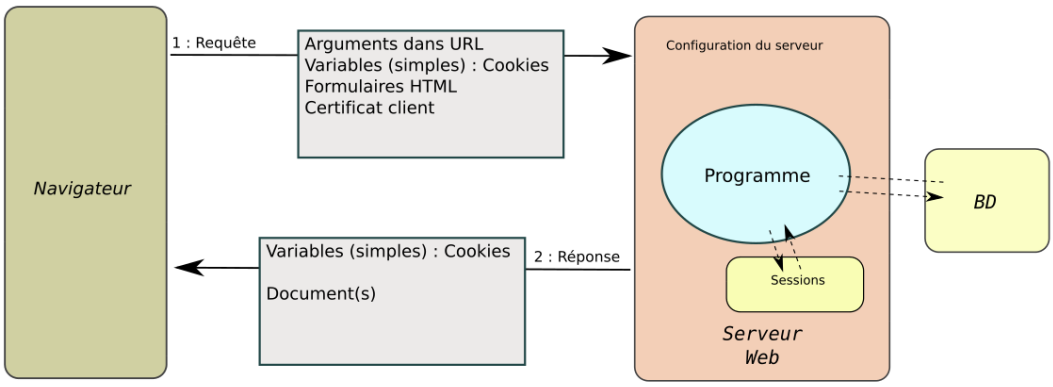
4.2 Exemple classique : affichage des résultats d'une requête en BD

1. Décodage des paramètres venant de la requête HTTP pour en faire des éléments d'une requêtes SQL (`... "WHERE ATTRIBUT = [argument transmis]" ...`)

   — (et récupération de la session ?)
2. Exécution du programme SQL (délégué au SGBD)
3. Habillage du résultat en HTML (écrit complètement avec des `"PRINT"`, boucle sur résultats, ...)
4. Écriture sur la "sortie standard"

> ATTENTION : sécurité => injections SQL

4.3 Données échangées



> Les certificats clients comportent aussi des informations potentiellement utiles au serveur et à l'application, mais les certificats TLS et HTTPS ne sont pas vus en détail dans ce cours.

4.4 En cas de panne ?

— Si rien de spécial n'est prévu : code de retour `5xx`
   — Erreur standard dans les *logs* non visibles par le client HTTP
— Si on souhaite faire mieux : mécanismes d'interception des crashes
   — Dépend du langage / *framework*
   — Gestion d'**exceptions**
      — Par exemple pour affichage d'une page HTML visualisable dans un navigateur, qui complète l'erreur `5xx`

4.5 Mise en place environnement TP

— Commencer les étapes du TP-B (cf. Moodle) :
   — Récupération archive code Symfony
   — Lancement Eclipse

> Les manipulation prennent du temps, si le réseau et les serveurs de fichiers NFS sont un peu chargés, donc on initie le tout pour que ça se déroule tranquilement pendant la première phase de cours intégré.

5 Sessions applicatives

5.1 Paradoxe : applications sur protocole sans état

— L'expérience utilisateur suppose une **instance unique** d'exécution d'application, comme dans un programme "sur le bureau"

— Pourtant, HTTP est *stateless* (sans état) : chaque requête recrée un nouveau contexte d'exécution

> Avec une application sur le bureau, entre deux actions de l'utilisateur, le programme ne change pas d'état et conserve la mémoire des actions (fonction défaire, etc.).
> Avec une application Web, chaque clic sur un lien (ou autres actions) réinitialiserait l'état de l'application à zéro ?
> L'enjeu a été de résoudre cette contradiction en ne réinitialisant pas l'état de l'application à chaque action de l'utilisateur.

5.2 Application peut garder la mémoire

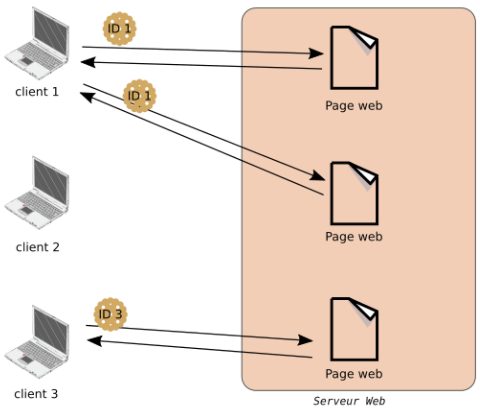
— Le programme Web peut stocker un état (par ex. en base de données) à la fin de chaque réponse à une requête
— Il peut le retrouver au début de la requête suivante
  — Le client doit pour cela se faire reconnaître
— Simule une session d'exécution unique comprenant une séquence d'actions de l'utilisateur

> Besoin de fonctionnalités côté serveur : pour purger le cache de sessions périodiquement, gérer la sécurité ...
> Attention aux boutons de retour en arrière du navigateur

5.3 Le client HTTP peut s'identifier

— Argument d'invocation dans URL
— **Cookie**
— ...

5.4 Identification du clients par cookie



— Identifie le client
— Commun à un ensemble d'URLs

> Identifiant fourni systématiquement au serveur, chaque fois que le même client HTTP se connecte
> Attention, pas nécessairement le même utilisateur final : plusieurs navigateurs ouverts en même temps sur le même site (sur plusieurs machines, ou avec plusieurs profils différents) => plusieurs cookies.
> Est-ce suffisant pour déterminer tout le contexte d'exécution de l'application ?

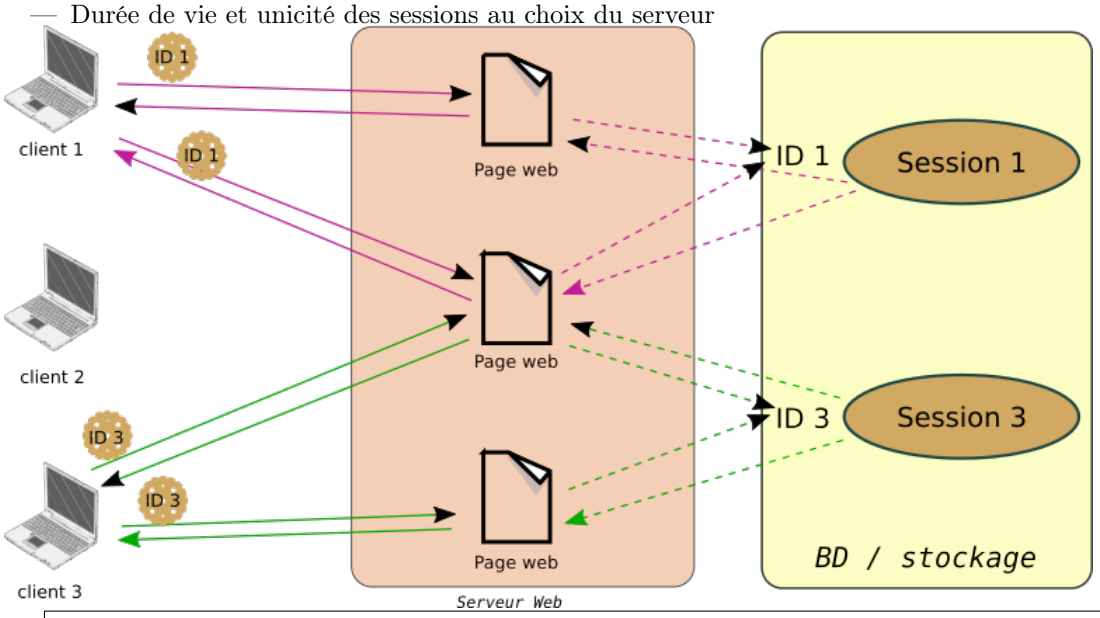
5.5 Cookies

— Juste un "jeton" unique sur un certain périmètre (serveur, chemin d'URL, application, ...)
— Format choisi par le serveur
— Peut contenir complètement un état de l'application : des données, mais **taille limitée**
— Le serveur peut trouver en base de données des données plus complètes sur présentation du jeton

Attention aux problématiques de sécurité : pas de mot de passe dans un Cookie, par exemple. Les cookies sont consultables par un attaquant ayant accès aux données stockées dans les données d'un profil du navigateur.

5.6 Stockage d'une session

- Session stockée sur le serveur (pas contrainte taille)
- Objets de l'application stockés dans la session
- Session retrouvée via un identifiant
  - Identifiant fourni par un *cookie*
- Durée de vie et unicité des sessions au choix du serveur



La session est potentiellement grosse, peut-être stockée en base de données, ...
Le "même" document accédé par deux clients présentant des ID différents, ne sera peut-être pas le même document, s'il est généré dynamiquement en fonction des informations présentes dans la session.
La plate-forme du langage de programmation rend l'utilisation de la session très facile pour le programmeur.
En pratique, la session n'est pas obligatoirement stockées dans un SGBD, mais plus souvent sur un système de fichiers ou dans une mémoire partagées (pour la gestion de cohérence si le serveur d'application est dans un environnement d'exécution distribuée).

5.6.1 Détails cookie

- Le serveur crée les cookies et les intégre dans la réponse HTTP
- Il utilise le champ "header" particulier `"Set-Cookie"` (sur une seule ligne)
  `Set-Cookie: <nom>=<valeur>;expires=<Date>;domain=<NomDeDomaine>; path=<Path>; secure`
  Exemple de réponse HTTP :
  ```
  HTTP/1.1 200 OK
  Server: Netscape-Entreprise/2.01
  Content-Type: text/html
  Content-Length: 100
  Set-Cookie: clientID=6969;domain=unsite.com; path=/jeux
  ```
- Ce cookie sera renvoyé avec chaque requête ayant comme URL de début `http://www.unsite.com/jeux/...`

Il peut y avoir plusieurs champs "Set-Cookie" dans le message HTTP, afin de représenter plusieurs cookies différents.

6 Développer des applis Web

Intéressons-nous à l'humain :
— L'utilisateur
— **Le programmeur**

> On se préoccupera de l'utilisateur dans la séance suivante.

6.1 Challenges

— Beaucoup de choses à faire
  — Comprendre HTTP (requêtes)
  — Programmer (la logique métier) dans un langage de haut niveau
  — Comment produire des pages Web comme interface de l'application
— **Maintenabilité**
— Qualité de l'expérience utilisateur
— Performances
— Sécurité

6.2 Méthode

— **Ne pas réinventer la roue**
— Plate-forme / *framework* de développement (Web)
  — Langage(s)
  — Outils
— Orienté Objet ?
— Patrons de conception
— Tests
— Aide de spécialistes (*web design*, performances)

> La plupart de ces aspects ne seront pas couvert en détail dans le présent cours.
> Approfondis dans les modules suivants du domaine informatique de la deuxième année (optionnels).
> Développer des applications Web, ça ne doit pas être vraiment différent de développer pour d'autres cas d'usage ou d'autres technologies.

7 Patron MVC

7.1 Patron de conception ?

— Ne pas réinventer la roue
— Appliquer un cadre méthodologique et technique standard

> Comment faire mieux que le PHP à papa : faire des applications Web maintenables.
> Comme en couture, un patron est un modèle "standard" qui correspond à une meilleure pratique du domaine.
> Le patron doit être adapté (à la taille du sujet, par exemple).

7.2 MVC

Patron architectural "MVC" :

M Modèle

V Vue

C Contrôleur

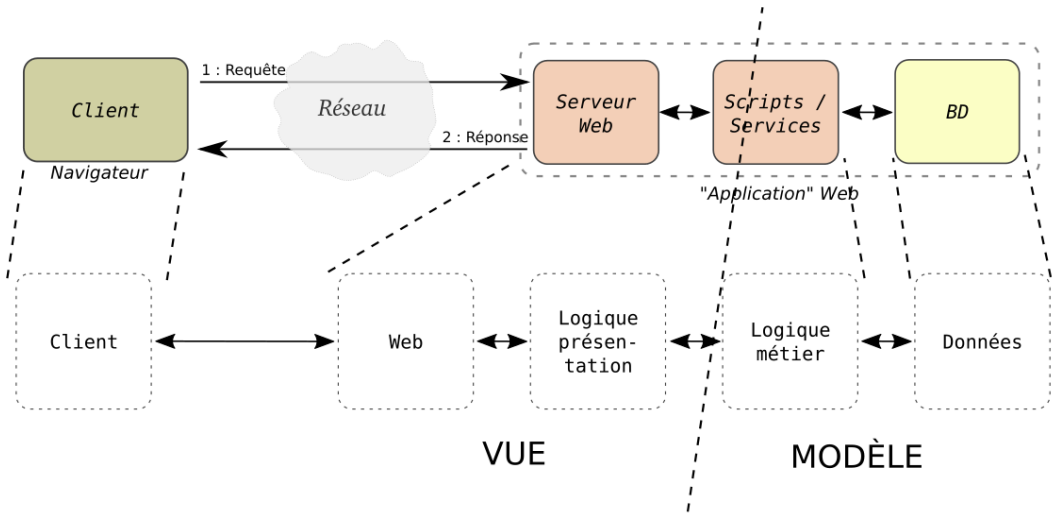
> Dans le cadre de ce cours, on ne détaille pas l'historique et toutes les subtilités de MVC ou ses variantes, mais on l'applique tel que proposé dans Symfony.

7.3 Modèle - Vue

Modulariser le code pour faciliter la maintenance.
Distinguons d'abord :

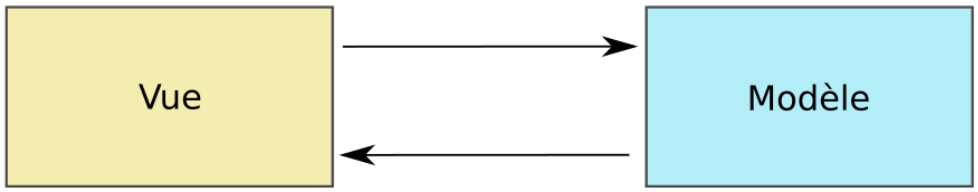
— Le **modèle** : les données applicatives, les objets, etc.
  — Côté serveur (PHP, SQL, . . .)
— La (les) **vue** (s) : une façon de présenter ces données dans une application (Web)
  — Côté serveur et client (PHP, HTML, CSS, JS)
  — Plusieurs représentations des mêmes données dans une même application

7.4 Frontière entre Modèle et Vue, côté serveur



7.5 Modèle - Vue

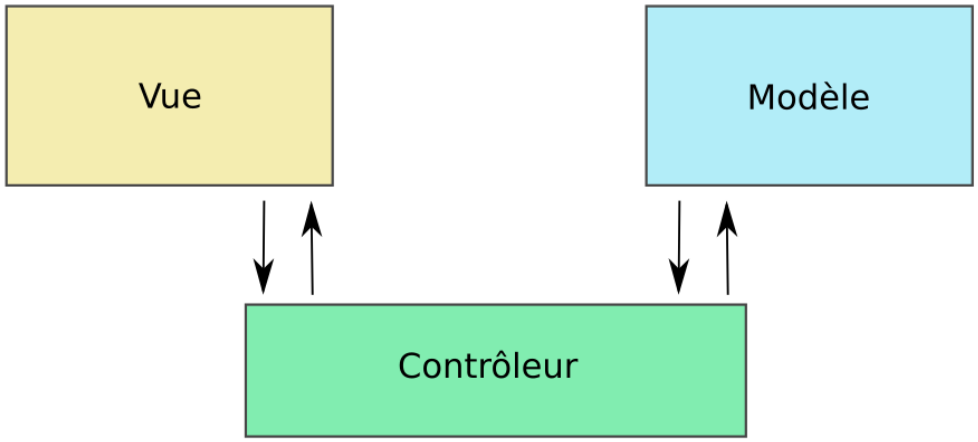
Première étape : séparer le code



On peut faire évoluer le modèle de façon transparente pour les utilisateurs.
Inversement, on peut faire évoluer l'ergonomie de l'application sans changer le modèle (passage d'interface texte ou *desktop* à pages Web, par exemple)

7.6 Modèle - Vue - Contrôleur

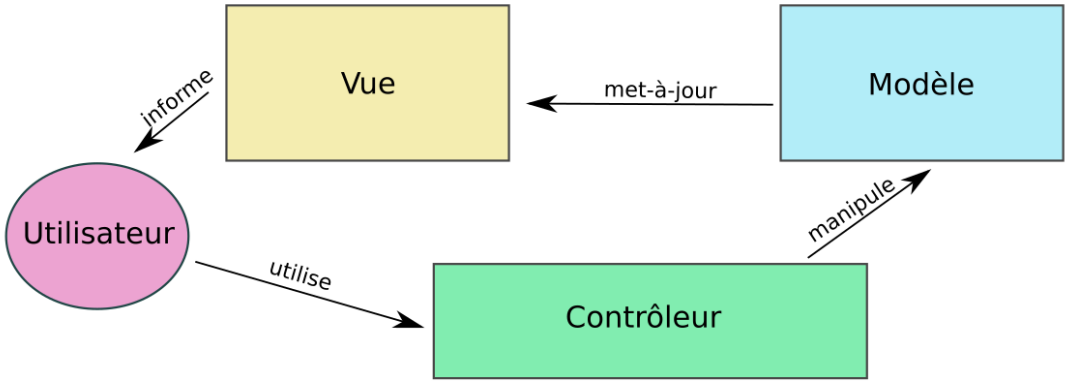
Deuxième étape : ajout d'un contrôleur qui gère la logique de l'application (enchainements dans l'interface graphique)



C'est ce Contrôleur qui est plus spécifiquement dépendant du choix des technos Web, et qui sera mis en oeuvre par un *Framework* de développement Web particulier (pour nous Symfony)
Le modèle peut être typiquement du SQL classique ou du PHP objet pas forcément lié à Symfony.
Idem pour les pages Web, qui peuvent être codées à l'ancienne ou avec un système de templates

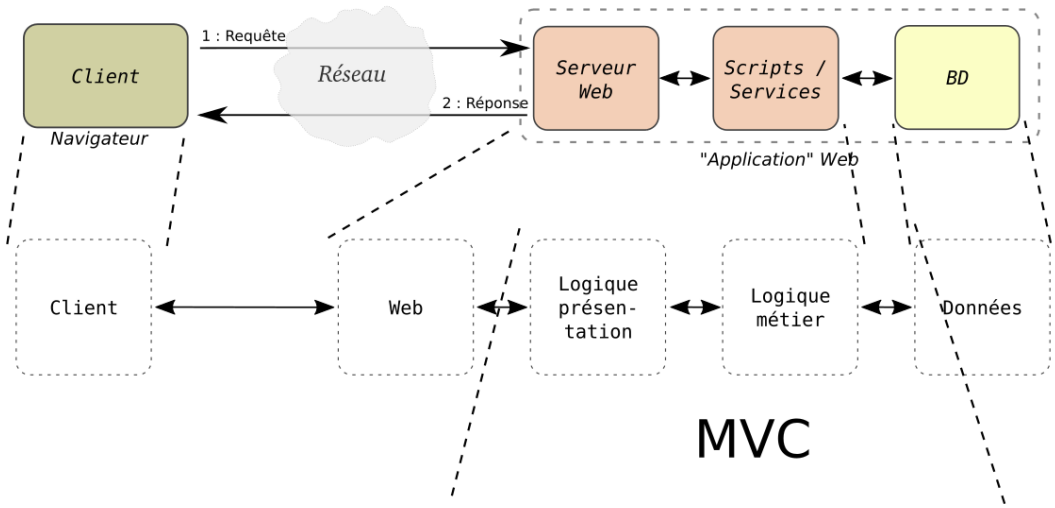
7.7 Rétroaction avec l'utilisateur

Boucle de rétroaction qui est au cœur du fonctionnement de l'application



Ce schéma est typiquement le même dans tous les systèmes d'interface utilisateur. Pas uniquement en application Web
Le contrôleur intègre une gestion événementielle : style de **programmation événementielle** : réflexes (*callbacks*) réagissant à des événements correspondant aux actions de l'utilisateur

7.8 MVC Web



8 *Framework* Symfony

8.1 Symfony

`https://symfony.com/`

— *Framework* PHP
— Logiciel libre
— Édité par SensioLabs (*Made in France*)
— MVC
— Populaire
— Souple
— Portable

Framework : ensemble de bibliothèques, mais aussi une façon de concevoir l'architecture de l'application
Extrait de `http://symfony.com/fr/explained-to-a-developer` :
Un framework pour simplifier les developpements
Un framework vous aide à mieux travailler (développements structurés) et à travailler plus rapidement (réutilisation de modules génériques). Un framework facilite la maintenance à long terme ainsi que l'évolutivité en se conformant aux normes de développement standard. Le respect des normes de développement simplifie également l'intégration et l'interfaçage de l'application avec le reste du système d'information.
Pourquoi Symfony ?
Symfony est un environnement stable de développement reconnu à l'international. Symfony est soutenu par SensioLabs, une entreprise avec plus de 13 ans d'expérience dans le développement web, ainsi que par une communauté internationale et dynamique qui assurent sa longévité. Un environnement innovant et simple d'utilisation grace à l'intégration de solutions créées dans d'autres environnements telle que l'injection de dépendance (repris de Java) et des solutions spécifiquement développées telles que la Web Debug Toolbar ou bien le Web Profiler. Enfin, en adoptant des normes de facto, Symfony ne vous limite pas à son environnement, mais vous permet de choisir les composants logiciels que vous souhaitez utiliser.

8.2 Documentation Symfony

RTFM

1. La documentation Symfony
2. StackExchange et autres forums

ATTENTION aux versions des logiciels et documentations

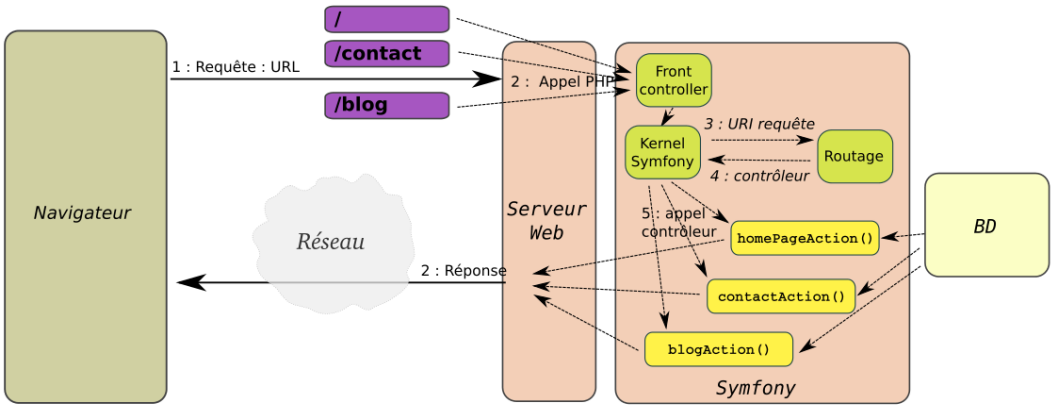
Réflexe : lire la doc, lire la doc, et encore lire la doc !
Pas la peine de l'imprimer, mais aussi possible (livres PDF / *ebook*)

8.3 MVC dans Symfony

Cf. documentation du *Symfony Book* :
Symfony and HTTP Fundamentals
(ou traduit en français ?)

Prendre le temps de lire la documentation.
Prendre le temps de poser des questions (ou rapporter des bugs) si ce n'est pas clair. Ces pointeurs sont des manuels fondamentaux de la culture *hacker* d'Internet.
Cf. Wayback machine si la doc la disparaît.

8.4 Routage d'une requête



Le "point d'entrée" du traitement des requêtes HTTP pour le programmeur d'application devient des **méthodes** d'une classe **Controleur** recevant des **arguments**, et utilisant des bibliothèques addititonnelles.
Beaucoup plus simple que ce qu'on devait faire en PHP classique.
Cohérent avec d'autres *frameworks* pour des langages objets, et avec des modèles de programmation événementielle dans d'autres contextes que les applications Web.
Orienté Objet FTW !

9 Technologie PHP

9.1 PHP

- Langage interprété
- Logiciel libre
- Versions : 5.x ou 7 ?
- Bibliothèques
  - Langage
  - Tierces

9.2 Langage

"*PHP : Hypertext Preprocessor*"

— Syntaxe style C / Java
— Objets

9.2.1 Hello world

```
<html>
 <head>
  <title>Test PHP</title>
 </head>
 <body>
 <?php echo '<p>Bonjour le monde</p>'; ?>
 </body>
</html>
```

PHP permet d'écrire des morceaux de programme à l'intérieur de pages HTML (*inline*).
Que peut-on en penser ?

9.2.2 PHP *inline*

— Mélanger la présentation et le code... **c'est mal** : maintenable ?
— On verra comment faire autrement dans la prochaine séance.

9.3 La documentation

— Le site de PHP : `http://php.net/`
— La documentation : `http://php.net/manual/fr/`

9.4 PEAR / Composer

Écosystème de bibliothèques, composants, *frameworks*
— PEAR : *PHP Extension and Application Repository* : `https://pear.php.net/`
— Composer : gestionnaire de dépendances : `https://getcomposer.org/`
  — Packagist : référentiel en ligne de packages `https://packagist.org/`

Composer : équivalent pour PHP de Maven pour Java Composer est utilisé par Symfony, mais on n'insiste pas trop dans ce cours : pas utilisé obligatoirement en TP

9.5 Tester / déployer

9.5.1 Tester

— Langage interprété
— Fonctionnalités environnemet de développement / tests de Symfony
  — Tests sur serveur Web local, en direct, en rechargement automatique des fichiers modifiés
  — Base de données locale (SQLite)

Les tests sont particulièrement importants dans un langage interprété.
Avec un langage compilé (comme Java) un grand nombre de problèmes et de bugs sont résolus lors de la phase de compilation.
Avec un langage interprété, c'est uniquement quand le programme fonctionne que ceux-ci seront détectés. Mieux vaudrait que ça soit en phase de développement plutôt qu'une fois en production.
Interessant aussi d'utiliser un éditeur ou un environnement de développement intégré offrant un support du langage. Dans ce cours, nous proposons l'utilisation d'Eclipse avec le bundle additionnel pour Symfony.

9.5.2 Déployer

— Déployer les fichiers de l'application
— Configurer (sécurité : accès direct aux mdp de connexion, etc.)
— Exécution dans *mod-php* dans Apache par exemple
— SGBDR (PostgreSQL)
— Test de performances, optimisations (*pre-fork* des threads, etc.)

> Dans ce cours on n'aborde pas plein de notions liées à la production, l'exploitation, notamment en terme de performances et d'optimisation, et d'impact sur l'architecture
> On n'aborde pas non-plus les solutions PaaS, qui offrent des avantages notamment si on utilise un *framework* supporté par la plate-forme.

10 Postface

10.1 Et maintenant

Installer **l'application de démo de Symfony** dans votre compte, pour être prêt à naviguer dedans (par exemple avec Eclipse) dans les prochaines séances.

`https://github.com/symfony/symfony-demo#installation`