Serveur Web, middleware, MVC et framework Symfony

CSC4101 - séance 2

Télécom SudParis

19/09/2018

Table des matières

Préface

Rappel séance précédente

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

Objectifs de cette séance

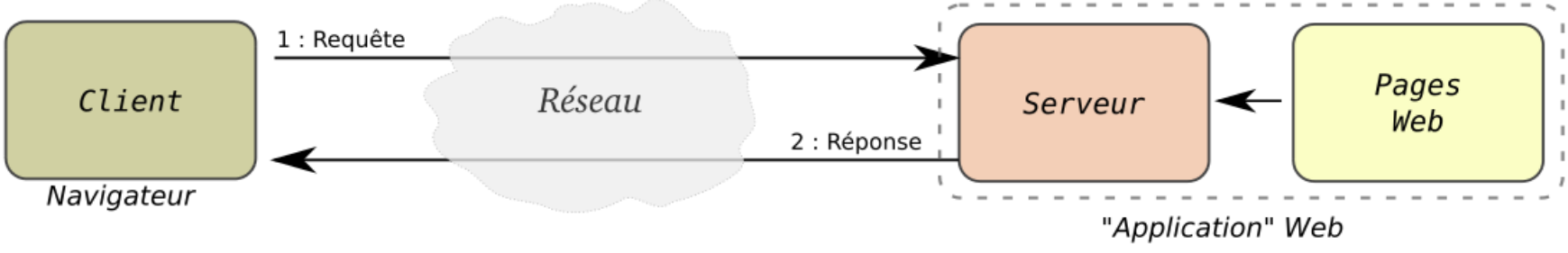
- Comprendre l'architecture de composants logiciels utilisés pour faire fonctionner des applications sur un serveur Web
- **Comment s'y prendre pour développer**
- Prise en main de l'environnement Symfony
- Programmer une application en PHP dans Symfony

Architecture

Principes de conception des applications

Serveur de pages Web statiques

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

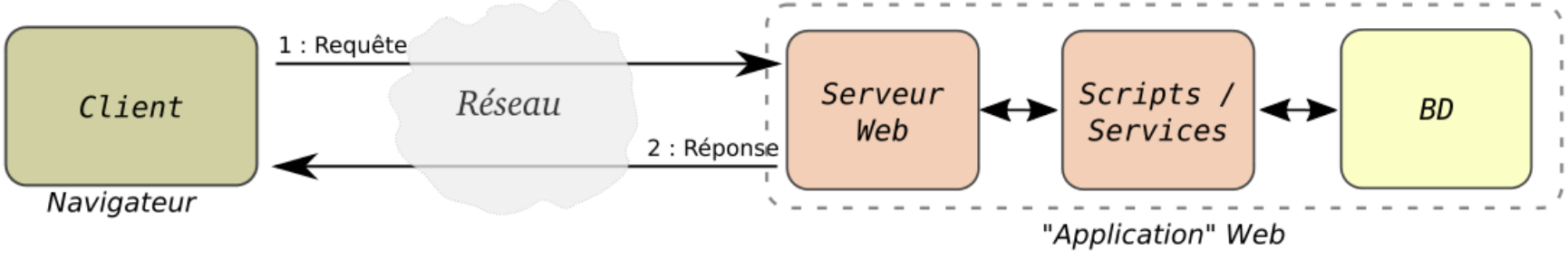


Caractéristiques d'un serveur de pages statiques

- Avantages
  - Efficace (tient la charge)
- 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.)

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.



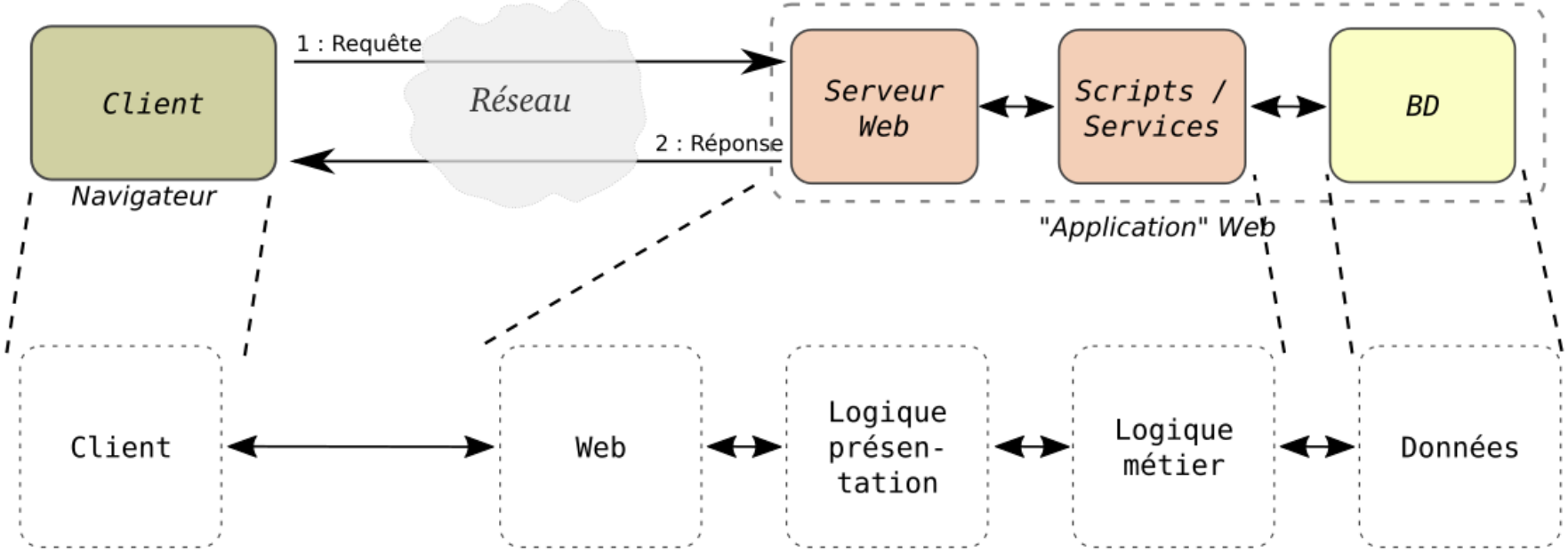
Caractéristiques d'un serveur pour applications BD + Web

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

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

Architecture multi-couches

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



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

Fonctionnement des serveurs Web

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.)

Mais aussi

- Vérifier la sécurité
- Gérer les performances
- …

Exemples

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

Programmes / Applications

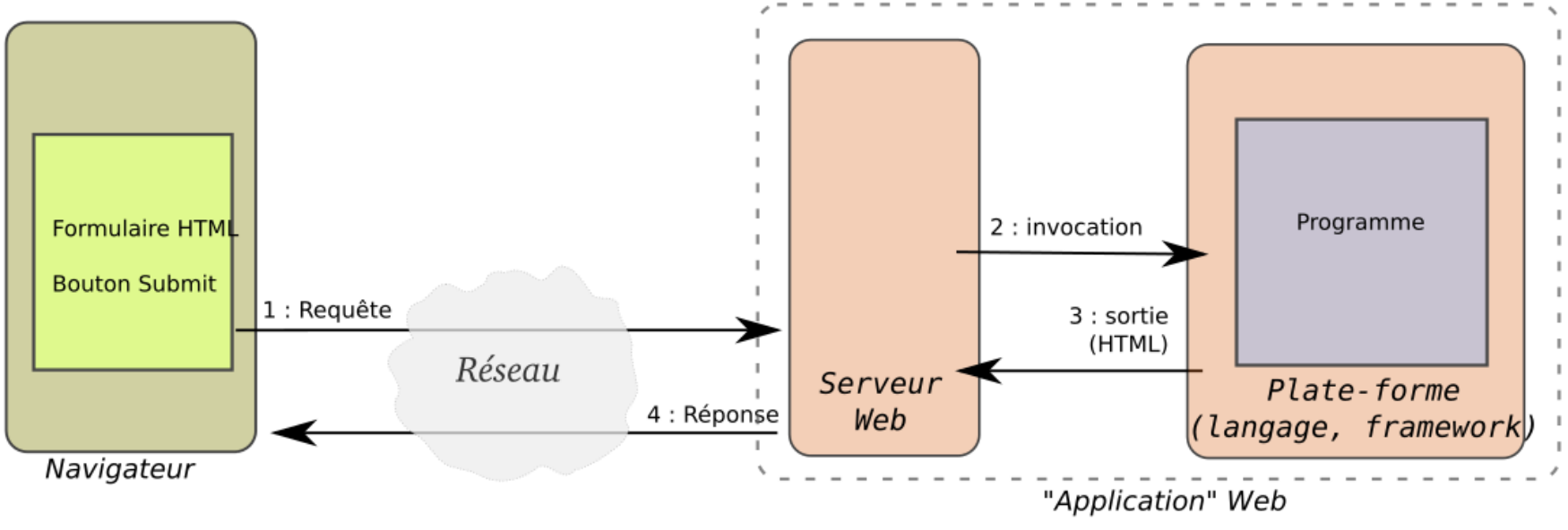
- Exécution :
  - **Directe** sur le système d'exploitation

  ou

  - Dans le contexte d'un **serveur d'applications**
    - Un module du serveur Web
    - Un serveur d'application dédié

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



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

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

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

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.

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)

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)

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 ?

Le programme en lui-même

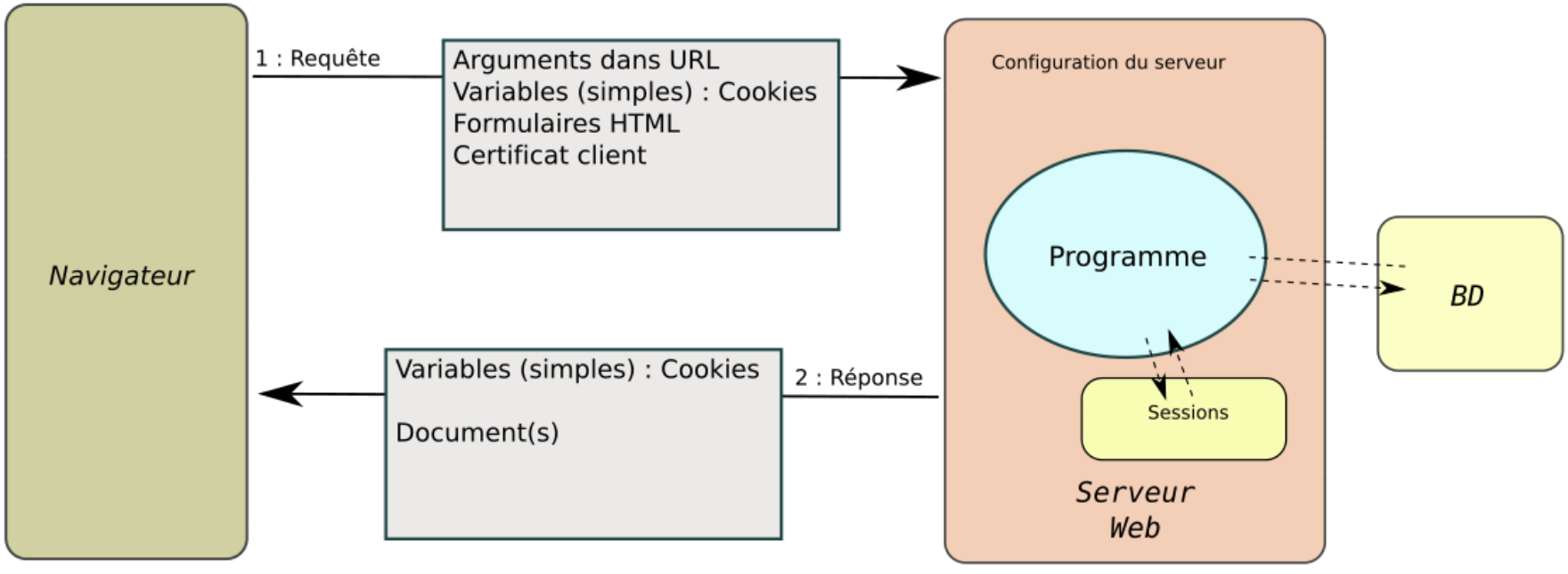
Étapes

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.

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"

Données échangées



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

TP A : Application simple en PHP

Mise en place environnement TP

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

Sessions applicatives

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

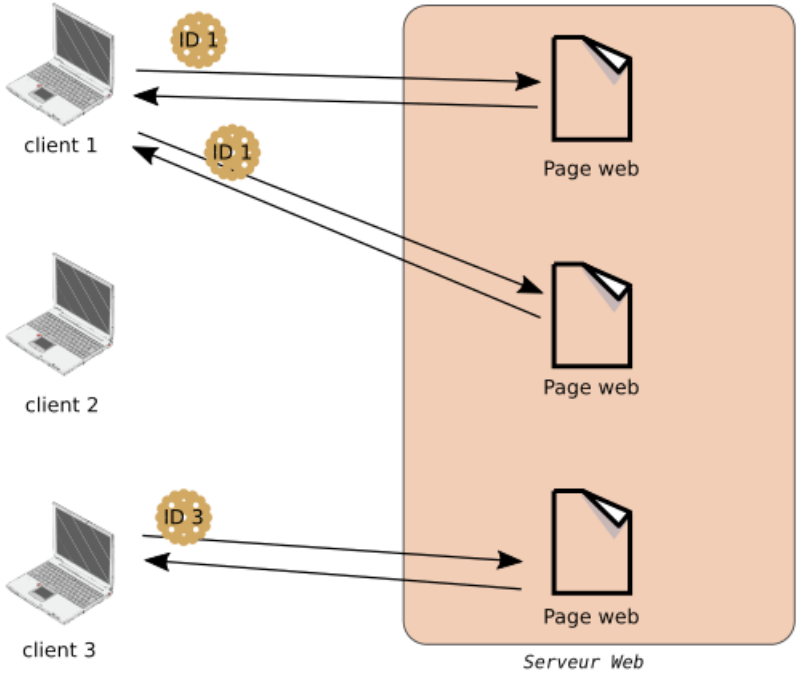
Application peut garder la mémoire

- 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

Le client HTTP peut s'identifier

- Argument d'invocation dans URL
- **Cookie**
- …

Identification du clients par cookie



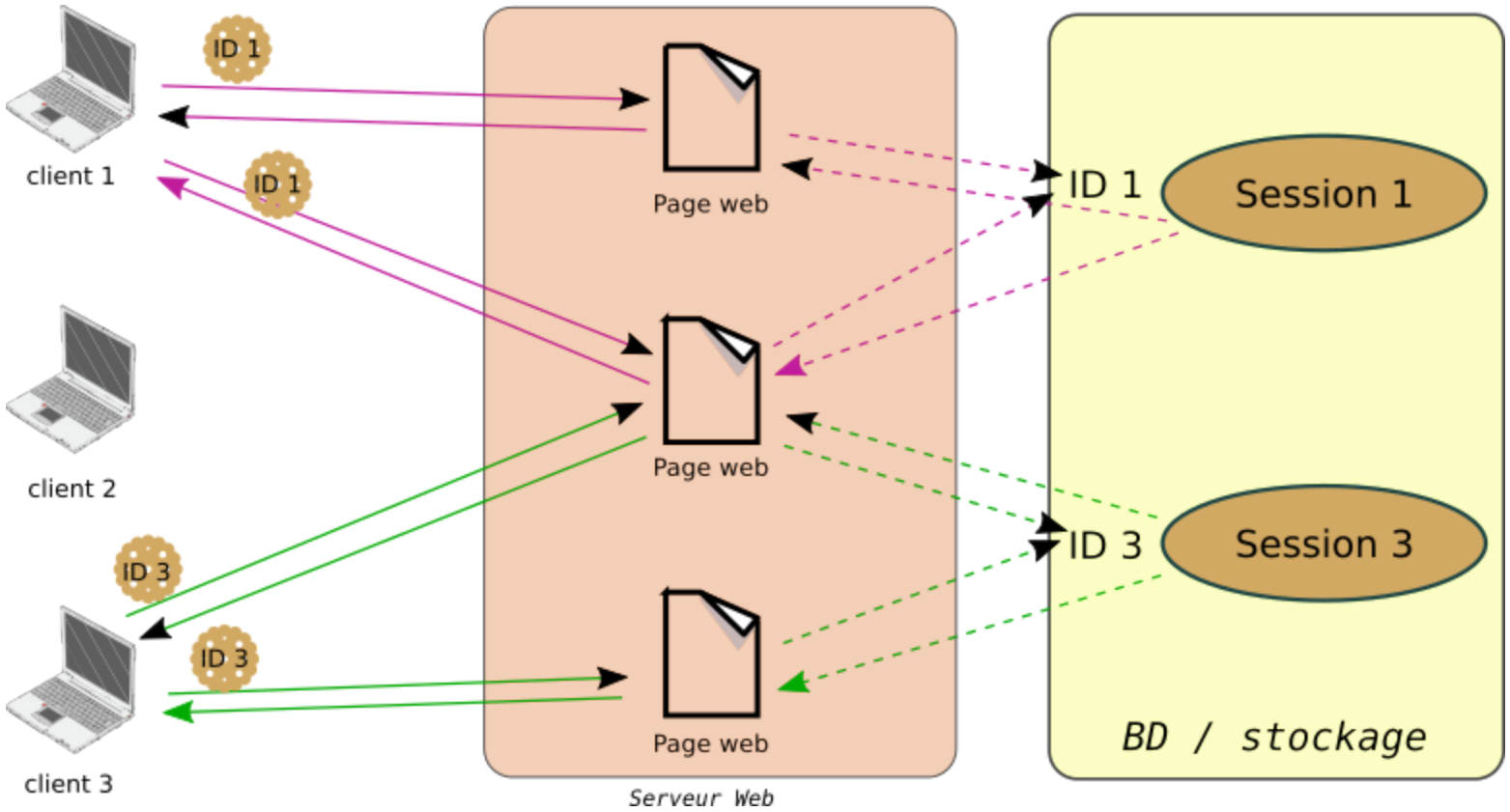
- Identifie le client
- Commun à un ensemble d'URLs

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

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


client 1
client 2
client 3
ID 1
ID 1
ID 3
ID 3
Page web
Page web
Page web
Serveur Web
ID 1
Session 1
ID 3
Session 3
BD / stockage

Détails cookie

- Le serveur crée les cookies et les intégre dans la réponse HT
- Il utilise le champ "header" particulier "`Set-Cookie`" (sur u seule ligne)

```
Set-Cookie: <nom>=<valeur>;expires=<Date>;domain=<NomDeDomaine>; path=<Pa
```

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 U de début `http://www.unsite.com/jeux/...`

Développer des applis Web

Intéressons-nous à l'humain :

- L'utilisateur
- **Le programmeur**

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

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)

Patron MVC

Patron de conception ?

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

MVC

Patron architectural "MVC" :

M

Modèle

V

Vue

C

Contrôleur

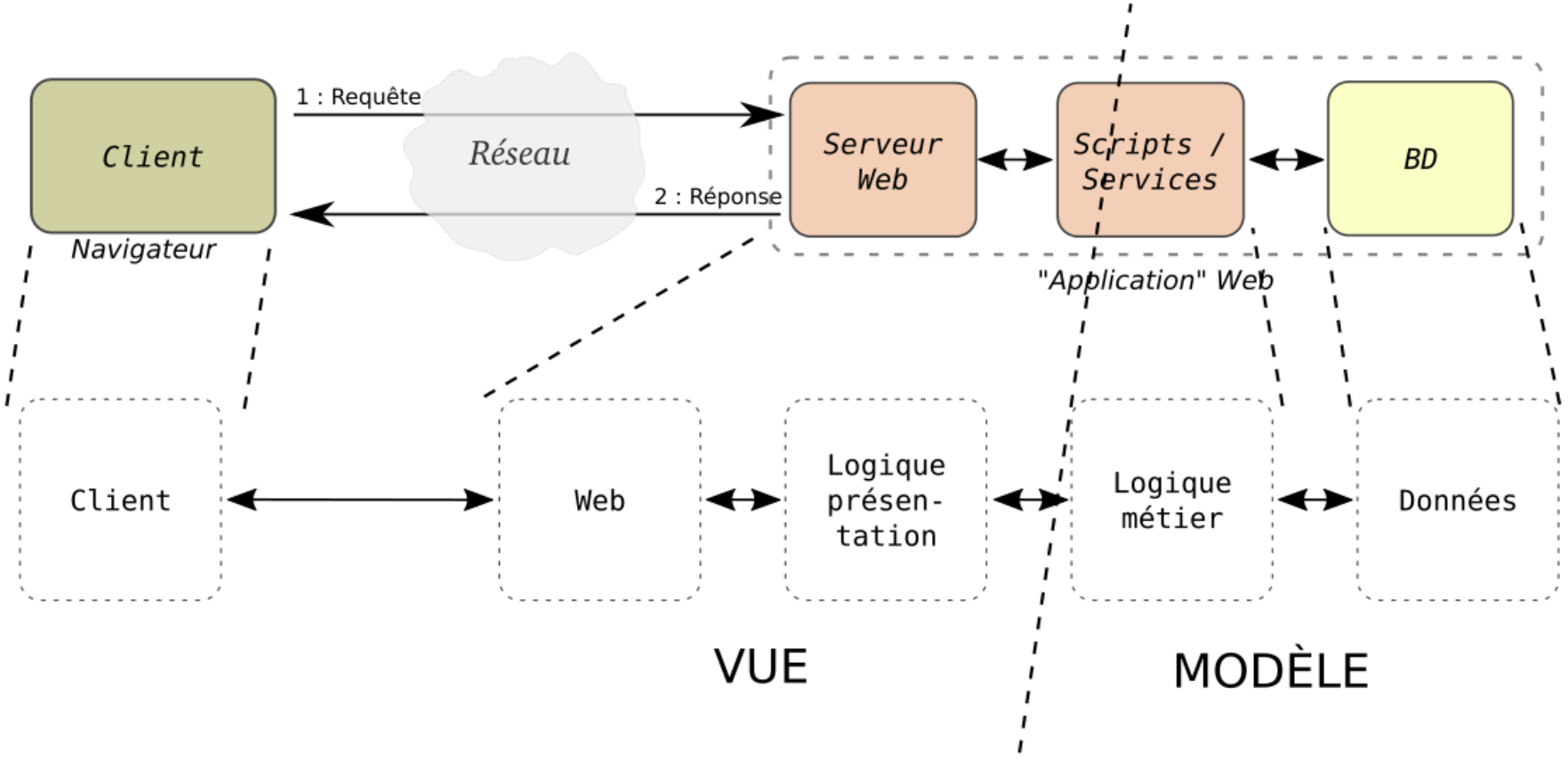
Modèle - Vue

Modulariser le code pour faciliter la maintenance.

Distinguons d'abord :

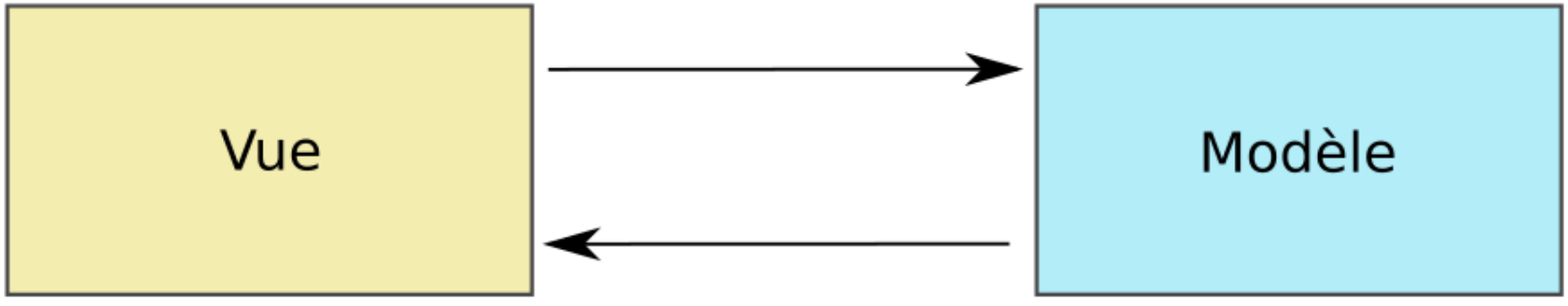
- Le **modèle** : les données applicatives, les objets, etc.
  - Côté serveur (PHP, SQL, …)
- La (les) **vue** (s) : une façon de présenter ces données dans une application (Web)
  - Côté serveur et client (PHP, HTML, CSS, JS)
  - Plusieurs représentations des mêmes données dans une même application

Frontière entre Modèle et Vue, côté serveur



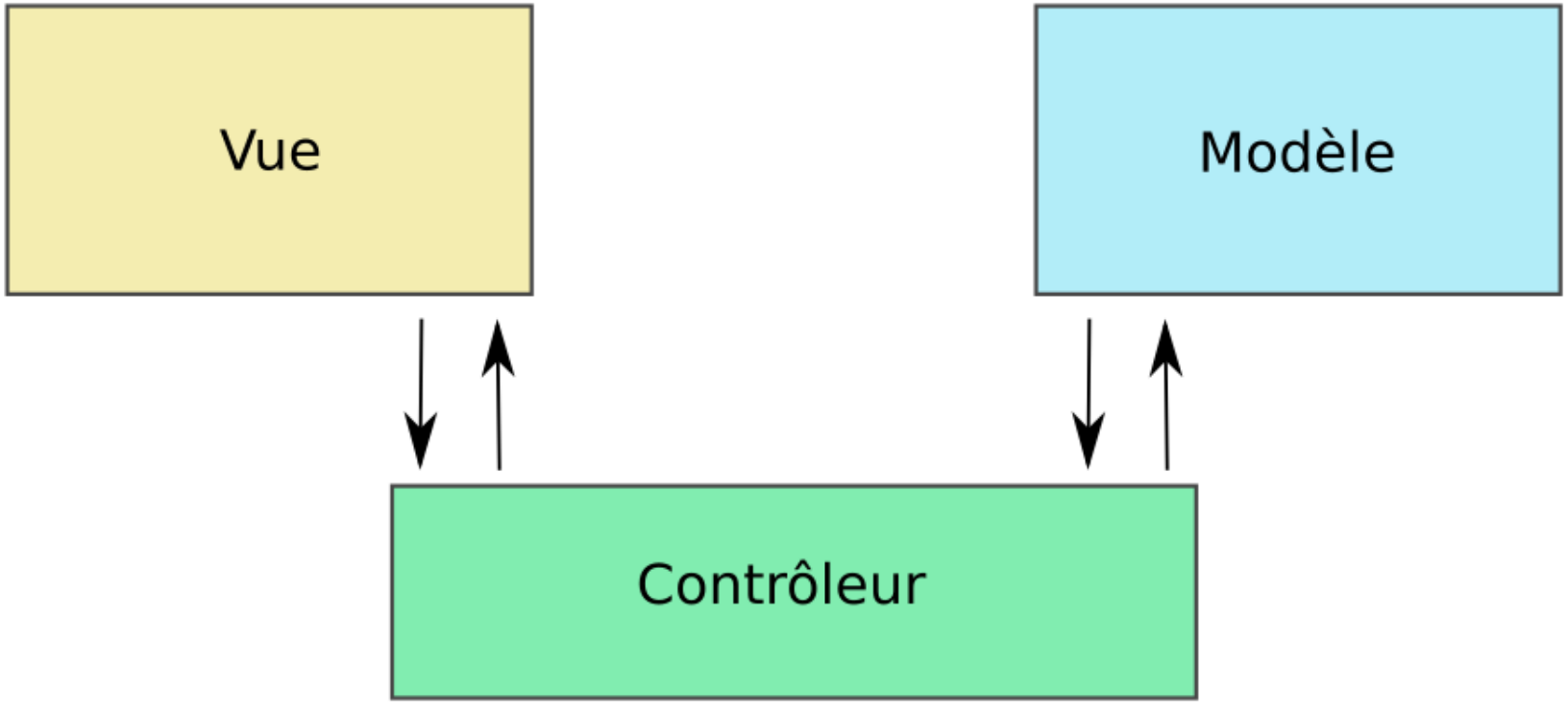
Modèle - Vue

Première étape : séparer le code



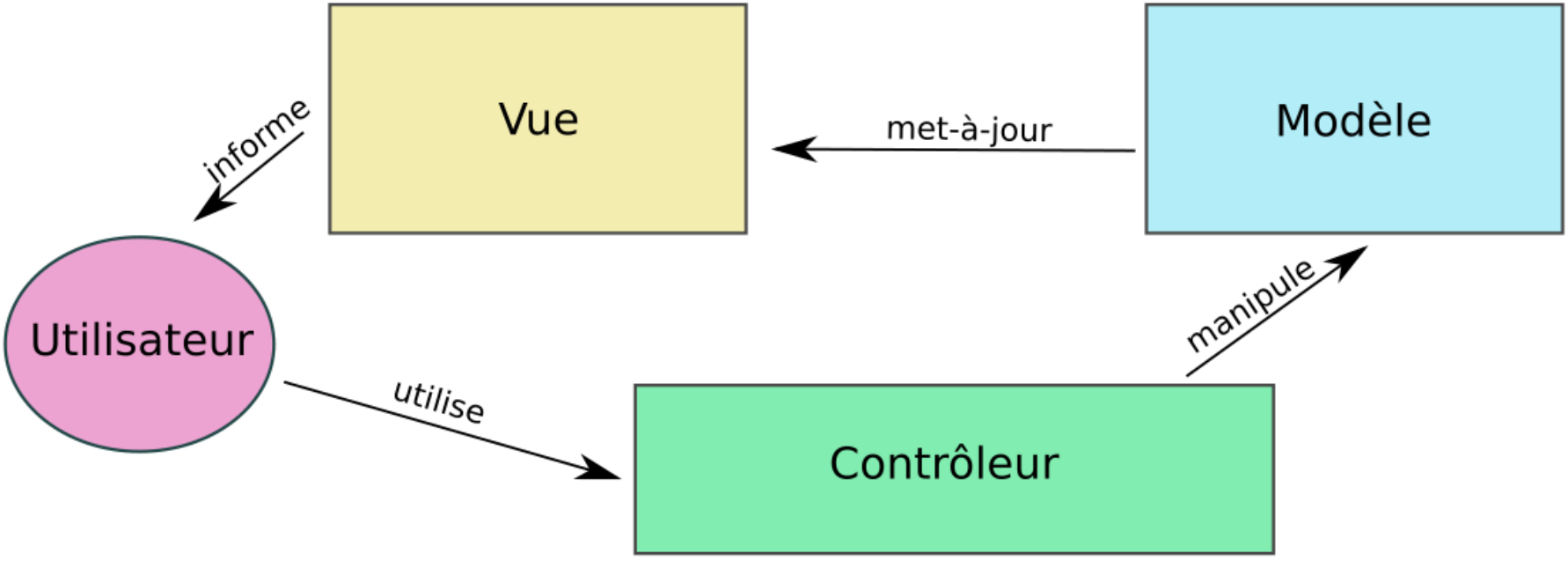
Modèle - Vue - Contrôleur

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

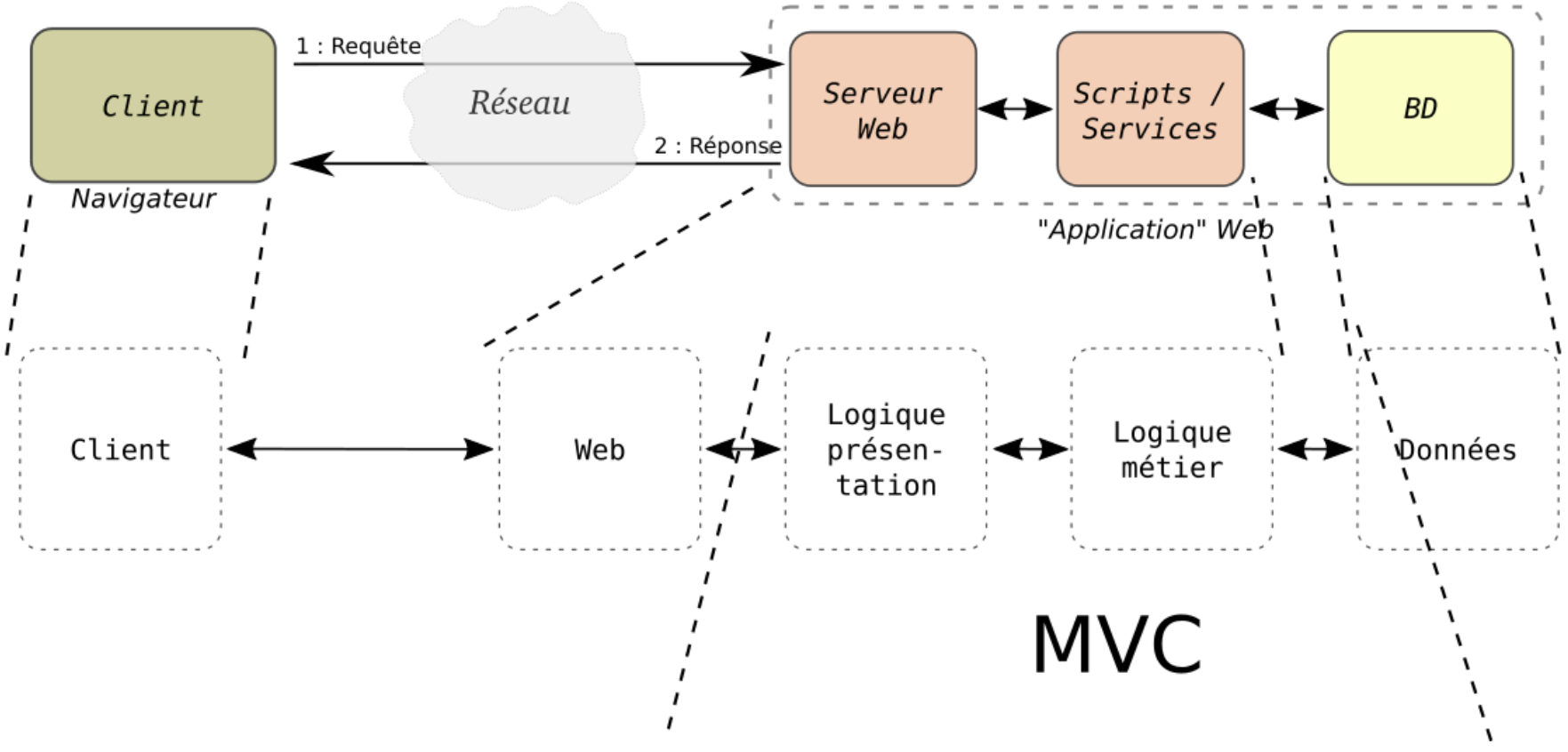


Rétroaction avec l'utilisateur

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



MVC Web


Client
1 : Requête
Réseau
2 : Réponse
Serveur Web
Scripts / Services
BD
Navigateur
"Application" Web
Client
Web
Logique présen-tation
Logique métier
Données
MVC

Framework Symfony

Symfony

https://symfony.com/

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

Documentation Symfony

RTFM

1. La documentation Symfony
2. StackExchange et autres forums

ATTENTION aux versions des logiciels et documentations

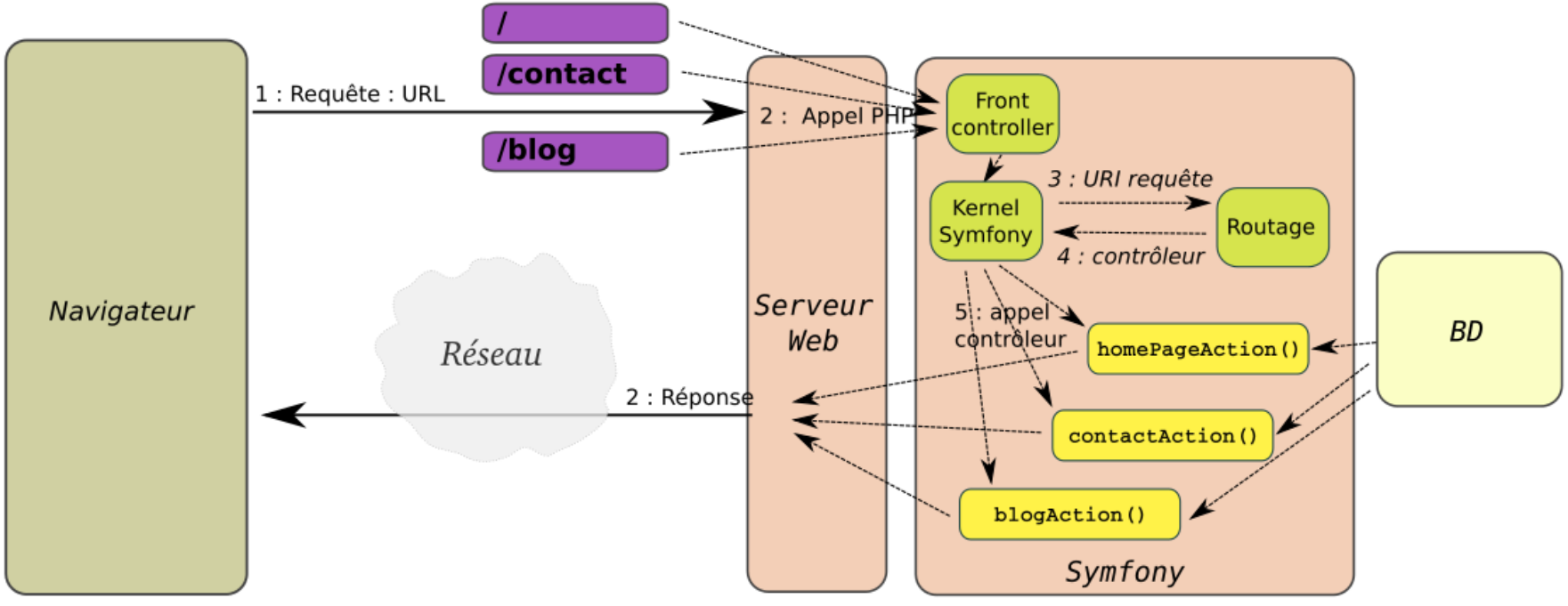
MVC dans Symfony

Cf. documentation du *Symfony Book* :

Symfony and HTTP Fundamentals

(ou traduit en français ?)

Routage d'une requête



Technologie PHP

PHP

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

Langage

"*PHP: Hypertext Preprocessor*"

- Syntaxe style C / Java
- Objets

Hello world

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

PHP *inline*

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

La documentation

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

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/

Tester / déployer

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)

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.)

TP B : Application simple en Symfony

Postface

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