Architecture(s) et application(s) Web

Télécom SudParis Olivier Berger (TSP)

22/08/2025

CSC4101 - Protocole HTTP, serveur Web et invocation programmes

Table des matières

Objectifs de cette séquence

Cette séquence de cours présente les mécanismes d’exécution de programmes derrière le serveur HTTP, permettant de faire fonctionner des applications Web dynamiques.

1 Fonctionnement des serveurs Web

L'objectif de cette section est de présenter ce qui se produit du côté des serveurs HTTP, et en particulier la façon dont des programmes sont appelés en réponse aux requêtes HTTP.

1.1 Rappel architecture « classique »

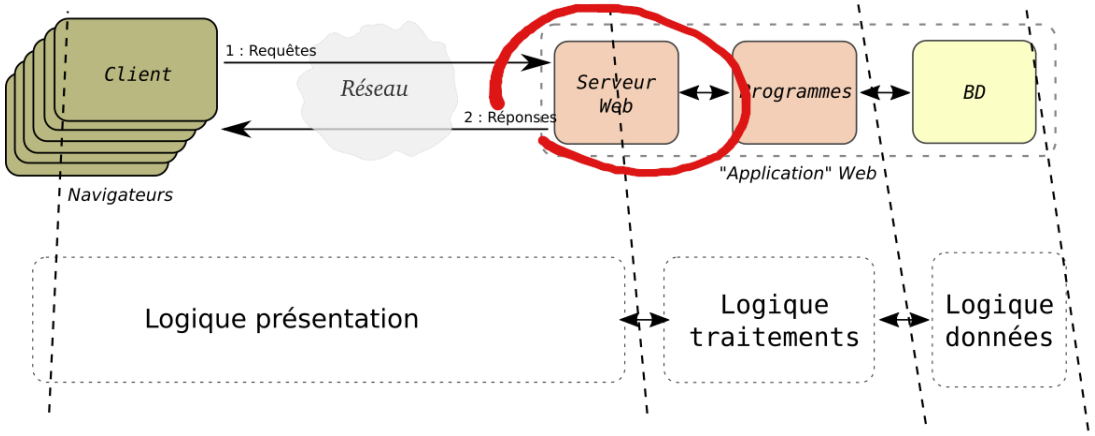


Figure 1 – Architecture 3 couches (focus sur le serveur Web)

1.2 Rôle du serveur HTTP

1. Comprendre les requêtes HTTP des clients
   — URL (`http(s)://example.com/hello`)
   — Verbe/méthode/action (GET, POST, …)
   — 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.

1.2.1 Mais aussi

— Vérifier la sécurité
— Gérer les performances
— …

1.2.2 Exemples

— Apache
— Nginx

— PaaS (*Platform as a Service*) prêt à l'emploi sur un *Cloud*, par ex. `platform.sh`

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.
On n'abordera pas les détails de configuration des serveurs Web comme Apache ou Nginx, faute de temps.

1.3 Programmes / Applications

1.3.1 Exécution d'appli Web à la demande

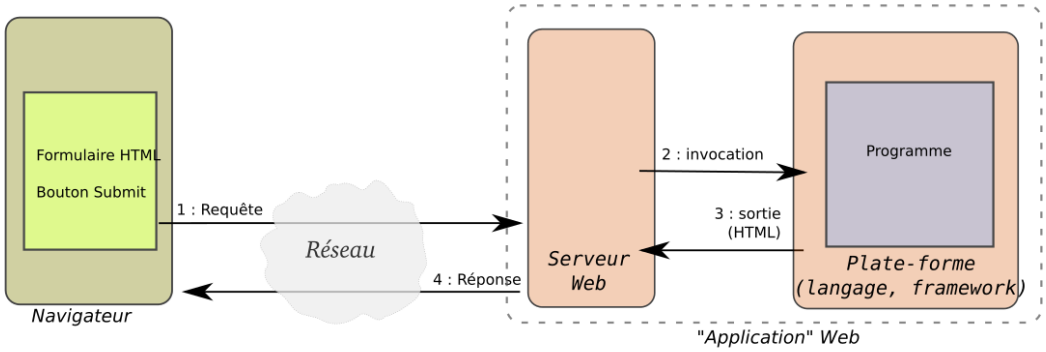
Serveur Web allumé en permanence. Application lancées à la demande.

1. Serveur Web attend des requêtes
2. Quand une requête HTTP particulière survient, déclencher **votre programme**
3. *recommencer*

Si plusieurs requêtes de plusieurs clients différents au même moment, même programme exécuté plusieurs fois en parallèle

1.3.2 Traitement typique d'une requête

— Le serveur Web invoque un **programme**
  — Programme s'exécute sur « *plate-forme* » : langage, etc.
  — Programme fournit sortie au format HTML (en général) : transmise « telle-quelle »

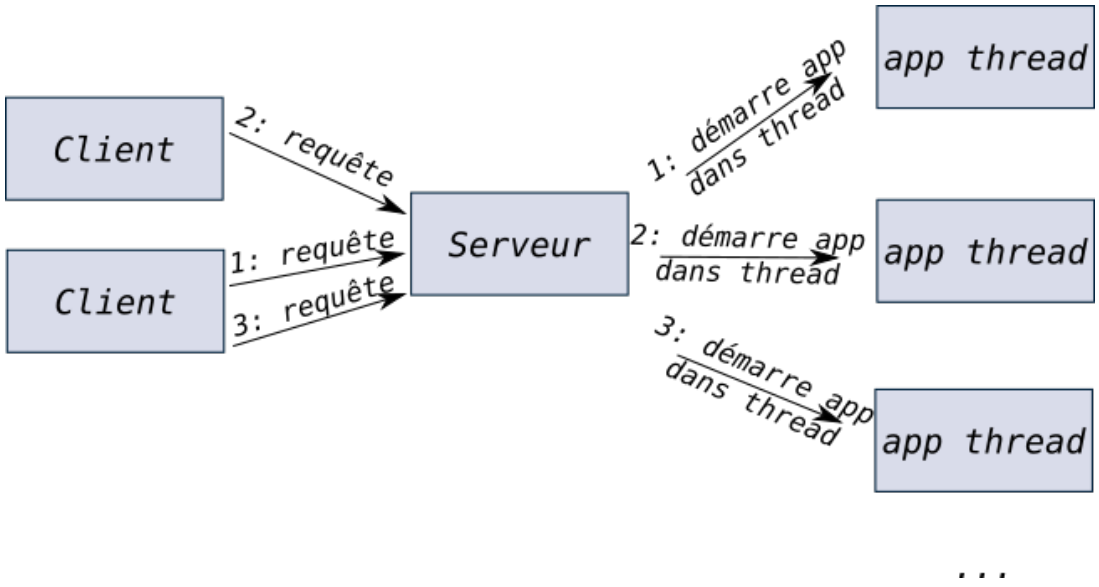


La plate-forme a pour rôle de faciliter la communication avec le programme : passage des données de la requête HTTP en entrées du programme, et transmission des sorties du programme vers une réponse HTTP.
Là où des programmes classiques en ligne de commande utilisent arguments, variables d'environnement et entrée et sortie standard, un programme Web utilise des en-têtes, des chemins de ressources, produit des codes de retour, du HTML, etc.

1.4 Multi-tâches

Gère plusieurs clients se connectant simultanément :
— au même serveur Web
— à la même application
… des centaines par secondes, OK ?

1.4.1 Concurrence d'exécution sur le serveur



Remarque : par défaut, les applications se terminent à la fin de la réponse à une requête. La requête 3 redémarre une requête. L'exécution pour la requête 1 du même client est oubliée.

1.5 Performances ?

Gère plusieurs clients se connectant simultanément :
- au même serveur Web
- à la même application

… des centaines par secondes, OK ?
Simplifions :
- 1 seul serveur Web, il gère OK (ressources statiques)
- démarrage centaines contextes application applications en parallèle… file d'attente utile ;-)

1.6 Technologies d'invocation d'un programme

- Différentes façons d'appeler un programme
- Exécution :
  - **Directe** sur le système d'exploitation : CGI (*Common Gateway Interface*)

  ou
  - Dans le contexte d'un **serveur d'applications** :
    - un module au sein du serveur Web
    - ou un **serveur d'application séparé** (aujourd'hui)

La différence entre ces deux modes d'exécution réside essentiellement dans la différence de performances : communication entre deux programmes distincts, au niveau système (chargement d'un exécutable, pipes, etc.), ou via un mécanisme intégré : programme (interpréteur) et bibliothèques déjà chargés, gestion de multi-processus/threads intégrée, etc.
Différentes techniques ont existé, mais on peut se concentrer sur celle utilisée classiquement dans le cas de PHP, qui hérite des concepts proches de l'exécution d'un programme par le système d'exploitation via les CGI, pour des raisons historiques.

1.6.1 Pour les archéologues : 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.

1.6.2 « Au sein » du serveur HTTP

- Le serveur HTTP « intègre » des fonctionnalités pour développer des applications (via des « *plugins* ») :
  - 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.

1.6.3 Serveur d'applications séparé

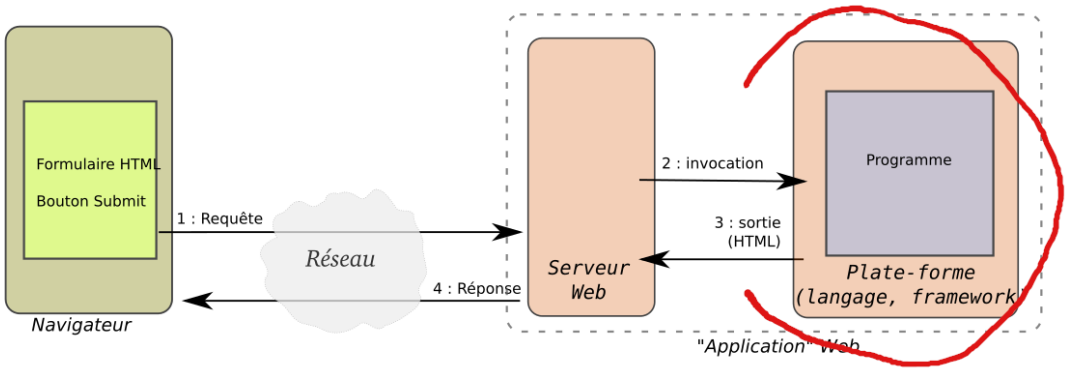


Figure 2 – Serveur d'application

- 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
— Rend indépendantes partie statique et partie dynamique : **s'adapter à la charge** plus ou moins dynamiquement.

1.6.4

Au final, peu importe : **nos programmes sont démarrés sur la plate-forme d'exécution le temps de traiter une requête, puis s'arrêtent**.

1.7 Exemple : plate-forme applications Web en PHP

Serveur d'exécution PHP dédié : PHP-FPM (*FastCGI Process Manager*)
— Embarque l'interpréteur du langage PHP
— Facile à tester : très populaire pour déploiement chez des hébergeurs (ex. nginx + php-fpm)
— Conserve des mécanismes très proches de la façon d'écrire les programmes en scripts CGI et de s'interfacer avec le serveur Web (FastCGI)
— Large choix de bibliothèques. Par exemple Symfony pour programmer les applications Web en objet

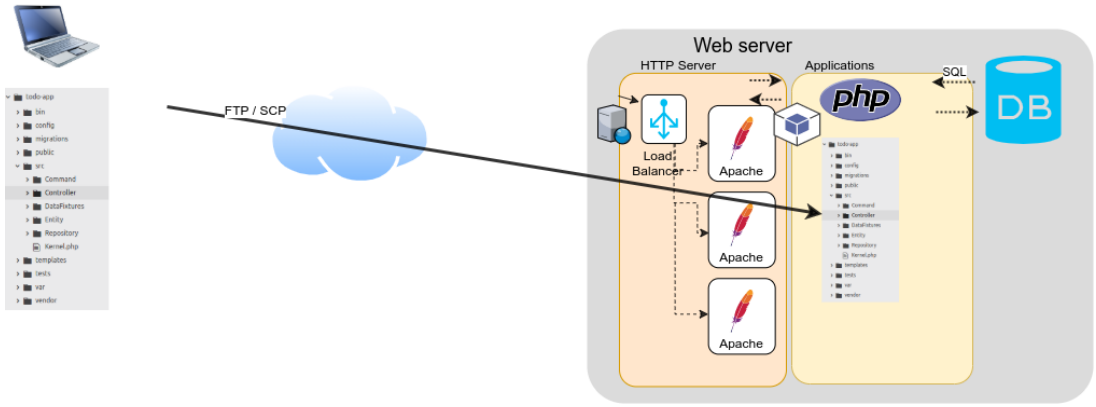
PHP est le langage qu'on va utiliser dans ce cours.
Aujourd'hui, dans le monde PHP, avec `PHP-FPM`, qui devient très répandu, on se rapproche un peu plus de ce modèle : serveur Web + serveur d'exécution PHP bien distincts (Apache + PHP-FPM ou NginX + PHP-FPM).
PHP-FPM implémente l'API FastGCI pour la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

2 Déployer

L'objectif de cette section est de présenter brièvement 2 solutions de déploiement des applications, couramment utilisées.

2.1 Serveur classique et recopie du code

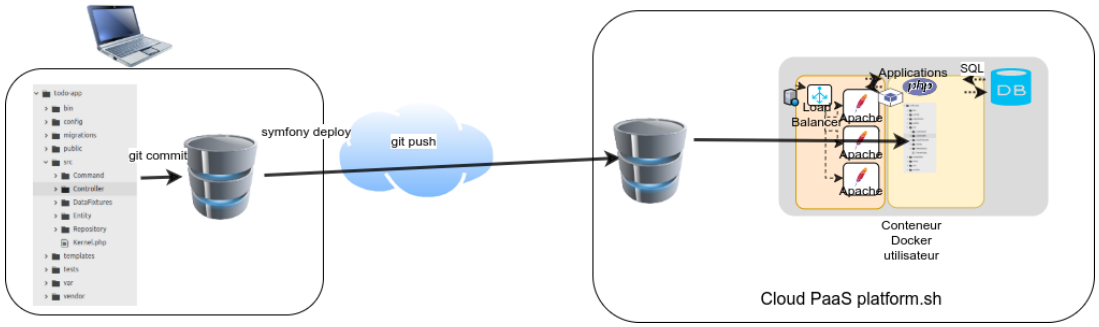
Exemple : avec FTP sur serveur Apache + PHP + SGBD



Il suffit de recopier le répertoire des fichiers source d'une application (ici, un projet PHP + Symfony), dans le répertoire ad-hoc d'un serveur Web.
La recopie des fichiers peut être faite avec `ftp`, `sftp` ou tout autre outil équivalent.
Cette solution traditionnelle a été employée depuis plusieurs décennies, mais elle a l'inconvénient de laisser la place à de multiples erreurs.
On risque notamment de permettre des modifications côté serveur en production, sans les répercuter dans le code source, a l'insu des développeurs.

2.2 Automatisation sur PaaS avec Git

Git push vers un hébergement *Cloud* PAAS (*Platform as a Service*)
Exemple : avec commande `symfony deploy` sur `platform.sh`



Le principe est ici de sauvegarder les modifications du code dans Git, puis de déployer une version automatiquement à partir de ce référentiel Git.
On automatise ainsi la sauvegarde d'une version bien identifiée, sans possibilité de modification côté serveur sans être repassé par la phase de contrôle qualité au niveau développement.
Ce mécanisme, utilisé dans l'approche *DevOps* permet d'automatiser et d'améliorer la qualité.
Les besoins côté serveur (version du langage, type de SGBD, dimensionnement, etc.) peuvent même être décrits dans des fichiers de configuration dans le code source.
Cette solution, illustrée ici avec la plate-forme *cloud* de `platform.sh` et Symfony fonctionne de façon similaire à nombre d'autres plateformes PaaS (*Platform as a Service*) sur le *Cloud*.
La plate-forme de développement et celle de déploiement sont gérées 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 Invocation du code applicatif

L'objectif de cette section est de détailler la façon dont le code des applications Web va être appelé, une fois qu'un serveur HTTP déclenche l'exécution d'un programme en réponse à une requête.

3.1 Concevoir les programmes

Objecif : **programmation événementielle** :

1. définir des **chemins d'URL**, et autres éléments de contexte qui déclencheront l'exécution du code
2. écrire les différents morceaux de code qui doivent s'exécuter quand les **requêtes HTTP** arrivent sur chacun de ces chemins

Exemple :
- `GET` sur `/todo/list` appelle `TodoController::index()`
- `GET` sur `/todo/show/{i}` appelle `TodoController::show($todo)`

Voyons en détails comment on y arrive

3.2 Contexte d'invocation

- Données transmises au serveur Web **dans la requête HTTP** :

  URL Arguments (*URL-encoded*)

  En-têtes *Cookies*

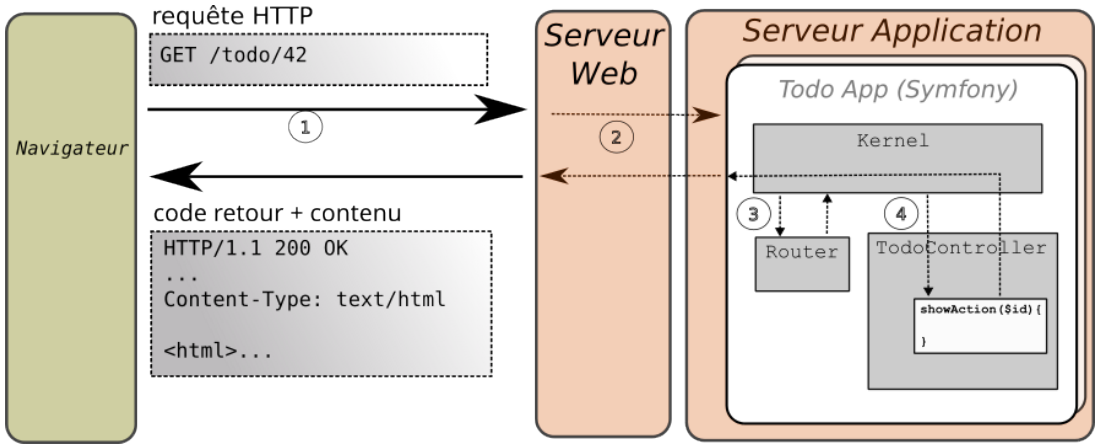
  Données contenu (*payload*) de la requête (ex. données de formulaires pour un `POST`)
- Le choix du **programme** à invoquer est déterminé par une combinaison d'éléments (a minima via le chemin dans l'URL)
  - La configuration du serveur détermine les règles à appliquer

3.3 Invocation des programmes

- Le serveur Web extrait le contexte des données reçues dans la requête (peut les filtrer)
- Le serveur d'application lance un « processus » pour le programme demandé
- Il transmet les données de contexte au programme
  - arguments d'un processus (CGI / FastCGI)
  - variables d'environnement (CGI / FastCGI)
- La plate-forme d'appli Web du cadriciel/langage :
  - **appelle une fonction/méthode** d'une API interne
  - passe en paramètres les variables extraites du contexte de la requête
  - …

Quels que soient le langage ou 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 (`getenv`) au temps des CGI).

3.3.1 Propagation invocation



3.3.2 Résultats d'exécution

En POSIX, exécution d'un processus :
- Statut d'une exécution de processus :
  - Code de retour système
    - `0` (succès)
    - ou `!= 0` (erreurs)
- Résultats :
  - Sortie standard du programme (`stdin`)
  - Erreur standard (`stderr`) ?

Idem en *CGI*, *FastCGI*, etc.

3.3.3 Conversion pour HTTP en contexte Web

Construction de la Réponse HTTP
- Code de retour -> Code de statut
  Exemple :

Succès	`0`	⇒	200
Erreur	`!= 0`	⇒	5xx

- Sorties
  - Standard (`stdout`) telle-quelle dans réponse HTTP : **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.
Pour mémoire, on a vu précédemment le format attendu pour les réponses HTTP, que la sortie standard doit donc respecter.
Attention donc 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 par le client HTTP. C'est particulièrement sensible si on affiche des traces de mise au point avec des *print* sur la sortie standard, qui risquent d'être ignorés par le client HTTP, s'ils apparaissent dans les en-têtes.
Le PHP « basique » pose un certain nombre de problèmes de ce type pour les programmeurs, qui seront résolus avec un *framework* comme Symfony.

3.3.4 Serveur HTTP intégré à PHP

Permet de tester comportement serveur d'application PHP simplement en local

— Démarrage :

```
php -S localhost:8000 -t public/
```

— Exécution script PHP demandé dans la requête, si présent dans `public/`, (ou par défaut, par `public/index.php`)

Exemples :

URL	Programme	Args
`http://localhost:8000/test.php`	`public/test.php`	
`http://localhost:8000/test.php?hello`	`public/test.php`	`hello`
`http://localhost:8000/`	`public/index.php`	
`http://localhost:8000/hello`	`public/index.php`	`hello`
`http://localhost:8000/index.php?hello`	`public/index.php`	`hello`

On utilisera la plate-forme PHP fortement en TPs et Hors-Présentiel, mais pas le serveur d'applications PHP-FPM.
On utilisera par contre l'environnement de développement et de mise au point fourni par le *framework* Symfony, qui permet un mode de fonctionnement un peu différent pour tester des programmes PHP.
Ce ne sera pas un serveur Web comme Apache ou NGinX qui invoquera des programmes PHP, mais on utilisera l'interpréteur PHP pour qu'il fasse fonctionner un serveur HTTP basique en amont des programmes, comme présenté ici.
Cela renverse un peu la logique : c'est l'interpréteur PHP qui démarre un serveur Web, et non l'inverse.
Par contre, en environnement de production, une fois l'application développée et déployée, Symfony s'exécuterait derrière PHP-FPM, de façon standard.
Le programme à invoquer est cherché dans le répertoire racine du serveur Web (l'argument de l'option `-t`, donc ici `public/`).
Si un programme PHP portant le même nom que le chemin d'accès à la ressource demandée existe, il est invoqué, comme pour `test.php` dans l'exemple.
Sinon, par défaut on invoque `index.php` (s'il existe… sinon, erreur). On lui passe alors éventuellement en argument le chemin d'accès à la ressource (`hello`).
Nous reverrons par la suite ce qui se produit dans ces programmes PHP.

3.4 Interface HTTP événementielle, en Objet

3.4.1 API Web

Principes d'une *Application Programming Interface* (API) **objet** d'une application Web, invoquée par le serveur d'application, dans un paradigme de **programmation évenementielle**.

3.4.2 Appel de méthode

— **Objets** : instances de classes (encapsulent un état)
— **Méthodes** : **récepteurs de messages**

Exemple : envoi du message « execute » à une commande

```
class Application {
    public run() {
        //...
        $command = new ListTodosCommand();
        $res = $command->execute();
```

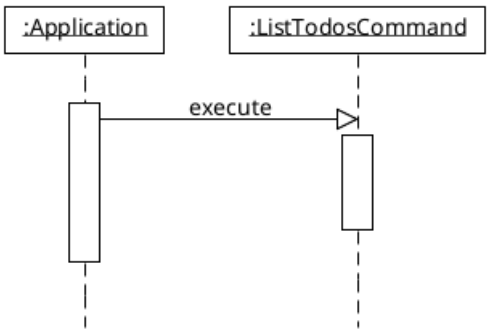


Figure 3 – « Invocation d'une méthode, en diagramme de séquence UML »

3.4.3 Modèle de programmation événementielle

Identique dans beaucoup de systèmes de programmation d'interfaces (GUI, HTTP, …) :
- objets gestionnaires d'interface
- méthodes activées par des **événements**

Réception d'une requête HTTP => appel de méthode

3.4.4 Contrôleurs HTTP

Appel de **méthodes** dans une classe conçue pour gérer des **actions** (HTTP) sur des **ressources** (Web)
- Classe contrôleur (*Controller*)
- « écoute » des événements : requêtes HTTP sur des ressources particulières propres à cette classe
- gère les actions spécifiques à effectuer (et délègue au reste du code)

Patron de conception *Model*, *Vue*, *Controller* (MVC)

3.4.5 Programmer les méthodes d'un contrôleur

- Le programmeur n'écrit pas le point d'entrée `index.php`
- Le *framework* fournit un composant offrant des fonctions de **routage** : décodage du contexte HTTP pour invoquer la bonne méthode de la bonne classe
- Le programmeur n'a plus qu'à écrire des méthodes dans une classe PHP : **programmation événementielle** dans un contexte **objet**

Pour le développeur PHP, on concoit l'application dans un style de programmation événementielle et orientée objet de façon similaire, pour une application Web/HTTP, à ce qu'on pourrait faire pour une application type GUI sur le bureau, ou sur mobile.
On utilise des mécanismes objet, comme les exceptions, par exemple, pour gérer les erreurs. Le framework les convertit en codes de retour HTTP de façon transparente pour le développeur.
Avec Symfony, on pense objet, et on bénéficie des apports du génie logiciel avec des bonnes pratiques de qualité logicielle, indépendamment du fait qu'on programme en PHP pour le Web.

3.4.6 URLs « lisibles »

Aiguillage des requêtes :
- URLs sans extension `.php`
- Tout est géré par `index.php`, qui passe la main au **routeur**

URL	Programme	Args
`http://localhost:8000/test.php`	404	
`http://localhost:8000/`	`public/index.php`	NULL
`http://localhost:8000/hello`	`public/index.php`	`hello`
`http://localhost:8000/blah/plop`	`public/index.php`	`blah/plop`

3.4.7 Rôle du *kernel* Symfony

Faire la conversion entre le mode d'invocation classique de PHP, hérité des CGI d'origine, et le mode objet moderne.
Le programmeur Symfony n'a plus qu'à s'occuper :
- déclarer le routage
- coder les classes contrôleurs

3.5 Routeurs et contrôleurs Symfony

3.5.1 Routage

- Le programmeur associe :
  - schéma de chemin d'URL
  - méthode d'une classe PHP « Contrôleur », à invoquer au traitement d'une requête
- Le composant « Routeur » réalise les appels vers les bonnes méthodes

3.5.2 Exemple `HomeController`

- Code : `src/Controller/HomeController.php`
- Gère :
  `http://localhost:8000/`
- Classe PHP `HomeController`
  - Méthode `indexAction()`

Route :

Chemin	Nom	Args
/	`home`	

3.5.3 Exemple `TodoController`

- Code : `src/Controller/TodoController.php`
- Gère :
  - `http://localhost:8000/todo/`
  - `http://localhost:8000/todo/42`
- Méthodes de la `TodoController` :
  - `listAction()`
  - `showAction(id)`
- Routes :

Chemin	Nom	Args
/todo/	`todo_list`	
/todo/{id}	`todo_show`	`id`

3.5.4 Méthodes gestionnaires de requêtes

Définition des routes avec annotations `Route` (attributs PHP 8)

```
class TodoController extends Controller {
    /**
     * Finds and displays a todo entity.
     */
    #[Route('/todo/{id}', name: 'todo_show',
            requirements: ['id' => '\d+'], methods: ['GET']) ]
```

```
    public function showAction(Todo $todo): Response {
        return ...;
    }
}
```

— *path* (arguments optionnels : `{id}`)
— *name* (obligatoire)
— *requirements* (optionnels. Ici `id` est un entier)
— *methods* (optionnel)

3.6 Exécution éphémère

Traitement d'un événement (réception d'une requête HTTP)
Temps de vie de l'application : appel d'une méthode de contrôleur

1. Initialisation cadriciel + routage
2. Exécution méthode du contrôleur
   (a) *Chargement données depuis base (et/ou session)*
   (b) *Traitements*
   (c) *Sauvegarde en base (et/ou session)*
   (d) *Retourne une réponse (inclut données/document …)*
3. Envoi Réponse HTTP
4. **Mort**

Répété à chaque requête

Pour ce qui nous concerne, le contexte d'exécution en mémoire est perdu après la fin de l'exécution du code de la méthode du contrôleur.
C'est comme si l'application s'arrêtait. Tout ce qui n'est pas explicitement sauvegardé est perdu.
On verra l'utilisation de la session ultérieurement dans le cours.
On notera qu'en réalité, l'application n'est pas complètement arrêtée : le code reste en mémoire, pour traiter la prochaine requête, afin de garder des performances acceptables. Mais les variables sont à coup sûr réinitialisées.

Take away

— serveur HTTP
— serveur application
— invocation des programmes qui génèrent du HTML
— programmation événementielle
— routage + controleurs Symfony
— méthodes des contrôleurs décorées avec `Route`

Copyright