Architecture(s) et application(s) Web

Télécom SudParis

2024-2025

CSC4101 - Polycopié

Table des matières

1 Contenu du présent document

Le présent document contient l'ensemble des polycopiés des différentes séquences du cours CSC 4101 « Architecture(s) et application(s) Web » de Télécom SudParis, dans son édition 2024-2025.
Ce polycopié n'est pas un support de cours autonome. **Sa lecture ne se substitue pas à la présence en cours magistral.**
La référence pour l'apprentissage est le discours prononcé par le professeur en cours magistral, qui est illustré sous forme de transparents vidéo-projetés.
Le présent polycopié reprend ces supports de projection, en y ajoutant des notes du professeur rédigées, qui reprennent ou complètent ce qui a été dit lors de l'affichage des transparents.

1.1 Auteurs

Par ordre chronologique inverse de contributions aux versions successives du cours : Olivier Berger (INF), Christian Bac (RST), Chantal Taconet (INF), Dominique Bouillet (INF).

1.2 Diffusion restreinte

1.3 Structure et contenu du document

Ce polycopié rassemble, en un seul document, l'assemblage de chacun des polycopiés des 10 séquences de cours, sous formes de sections successives.
Chaque section reprend le contenu abordé en cours, tel que présenté dans les transparents, complété par des explications littérales ou des informations complémentaires.

Ce contenu additionnel du polycopié sera placé dans des blocs sur fond grisé, tel que le présent paragraphe.

1.3.1 Forme de la rédaction

Ce polycopié est composé automatiquement à partir des sources des transparents. [1]
Certaines formulations en mode « télégraphique » seront donc parfois insuffisantes en cas de lecture indépendante du discours prononcé.

1.4 Mises à jour

Le polycopié imprimé diffusé en début de module est probablement dans un état de rédaction antérieur à celui présenté en cours.
On pourra se référer utilement aux versions PDF des polycopiés de chacune des séquences, téléchargeables sur le site du cours, pour des versions plus à jour.

1. Le système de génération des polycopiés et des slides depuis la même source est publié sur : `https://olberger.gitlab.io/org-teaching/README.html`

2 Séq. 1 (1/3) : Introduction du module

Objectifs de cette séquence

Cette première séquence de cours présente les obejectifs et l'organisation proposés pour le cours CSC4101.

2.1 Objectifs du cours CSC4101

Cette section présente la place de ce cours dans l'ensemble des enseignements d'informatique de TSP, les objectifs d'apprentissages visés, et les modalités d'organisation du dispositif pédagogique.

2.1.1 Enseignement en Informatique (CSC)

Architecture(s) et application(s) Web
Comment construire une application « classique », sur la plate-forme universelle, le Web, en utilisant une approche maîtrisée.

Ce cours fait suite aux enseignements de 1ère année à TSP

1ère année :
- CSC 3101 Algo. programmation (objet)
  - **algorithmique**
  - **objet** (en java)
- CSC 3102 Intro. systèmes d'exploitation
  - **shell**
- CSC 3601 Modélisation, BD et SI
  - **données**, **persistence**, **associations**
- PRO 3600 Projet dév. informatique
  - **interfaces graphiques** (Web ?)
  - Git ?

On considérera acquises les notions et savoir-faire introduits dans ces modules de première année.

2ème année :
- **CSC 4101 : VOUS ÊTES ICI !**
  En cas de doute : `http://perdu.com/`
- CSC4102, …

Objectifs d'apprentissage
- À l'issue de ce module, les étudiant(e)s […] seront capables de **développer une application Web** de type site d'*e-commerce* (une dizaine de pages), sur la base d'un cahier des charges fourni, en utilisant un *framework* PHP professionnel (Symfony).
  L'application sera réalisée […] en s'inspirant de versions successives d'une application exemple fournie. Elle devra permettre la saisie de données, et aura un comportement personnalisé en fonction du profil d'un utilisateur.
  - ce développement sera effectué par la **mise en œuvre des bibliothèques et outils du *framework* objet**, afin d'exploiter des fonctions de sécurité, de présentation dans des pages HTML, pour s'approcher d'une application réaliste, de qualité professionnelle.

  - ils/elles **utiliseront les outils de mise au point du *framework* et du navigateur** ;
- Les étudiant(e)s sauront **expliquer les rôles respectifs des traitements faits sur le client et le serveur** HTTP.
  - Ils/elles sauront **positionner les composants d'une application Web, dans une architecture en couches** (*multi-tiers*).
  - Ils/elles pourront **expliquer comment fonctionnent les sessions applicatives dans un protocole où le serveur est sans-état**.
- Les étudiant(e)s ont la liberté de personnaliser l'apparence des pages du site, ce qui permet **d'appréhender les principes généraux d'ergonomie des interfaces Web** (expérience utilisateur, accessibilité).

PHP, mais pourquoi ???!!! *Ô misère*
On va y revenir un peu plus loin

2.2 Organisation de CSC4101

2.2.1 10 séquences

1. Intro, PHP, accès aux données
2. Concepts, archi. appli Web 3 couches
3. Protocole HTTP, serveur Web, invocation programmes
4. Génération de pages, gabarits (*templates*), HTML
5. Expérience utilisateur Web, CSS
6. Contrôleurs, interactions CRUD, formulaires
7. Sessions, contrôle d'accès
8. Interface dynamique côté navigateur (JavaScript)
9. Sécurité, gestion des erreurs
10. Évolution des architectures applicatives

Ceci correspond aux grandes lignes du séquençage du cours. Nous y reviendrons plus tard dans cette séquence, plus en détail.

2.2.2 Format de chaque séquence de cours

- Travail encadré :
  - Cours intégré (*aka* BE ?) ~ 1h
  - TP ~ 2h
- Travail autonomie hors-présentiel :
  - apprentissage
  - projet d'application (au fil des séquences)

TP présentiel
- Grandes salles C00x - D0x (autant que possible)
- BYOD (*Bring Your Own Device*)

2.2.3 Projet : réaliser une appli Web

Début semaine 3
- pour membres d'une communauté :
  - gèrer un « inventaire » d'objets personnels,
  - publier des « galeries » publiques

- vous choisissez un **thème personnel** (*vinyls, cartes à collectionner, mangas, outils de bricolage, guitares, jeux …*)
- modèle de données même structure,
- fonctionnalités identiques
- *look* particulier

Évaluation : par les pairs (2 évals à mi-parcours et fin projet)

2.2.4 Travail étudiants

- Synchrone encadré :
  - CI (`s/CI/BE/`)
  - TP
- Autonomie :
  - fin des TP
  - auto-apprentissage
  - **projet** individuel
  - évaluation des projets de vos pairs

Attention : lisser l'effort du projet au long des 8 semaines du projet (10 semaines calandaires) !!!

2.2.5 Application « fil rouge » en TPs

Application Web de **gestion de tâches** : `ToDo`

- fonctionnalités très simples
- PHP + Symfony
- support de compréhension en TPs

Ensuite : application dans votre projet (guidée, mais plus autonome)

Cette application sera étudiée ensemble en TPs, afin de comprendre les mécanismes de mise en œuvre, et de servir de source d'inspiration pour le projet.

2.2.6 Évaluation

- Contrôle continu (évaluations projet)
- Contrôle final (sur table sans ordi)

2.2.7 Équipe pédagogique

Coordinateurs

- Olivier BERGER (B304.01 + Télétravail)
- Michel Simatic (idem)

Contact (2 en même temps) :

- `olivier.berger@telecom-sudparis.eu`
- `michel.simatic@telecom-sudparis.eu`

Cours Intégrés

- Olivier BERGER

Encadrants de TP

- Chourouk Belheouane
- Olivier Berger
- Éric Lallet
- Badran Raddaoui
- Mohamed Sellami
- Michel Simatic

2.2.8 Comment écrire à son prof

1. Utilisez l'adresse mail `@telecom-sudparis.eu`
2. Donnez un objet clair à votre email
3. Restez simple
4. Utilisez un français correct (ou l'anglais)
5. Soyez agréable
6. Remerciez-les

Just sayin'

Source : Comment envoyer un email à votre professeur ?

2.2.9 Ressources pédagogiques

— Moodle.ip-paris.fr et site Web du cours (supports de TP et Travail Autonome)
— Poly(s) PDF (inclus slides)
— **Documentation de référence** (en anglais la plupart du temps)

… et le Web ;-)
Importance de la documentation de référence.
Les polycopiés sont disponible en un seul document (papier / PDF), ou séparément, pour chaque séquence en PDF.
Une version au format texte OpenDocument (`.odt`) est également générée pour permettre aux lecteurs de disposer d'une version modifiable, leur permettant d'ajuster finement les paramètres de mise en forme (polices, interlignes, justification, etc.). Celle-ci pourra, nous l'espérons, permettre une adaptation particulière pour des besoins spécifiques (troubles dyslexiques, handicap, …), là où une version PDF standardisée ne conviendrait pas à tous.

2.2.10 Outillage

— Système BYOD (*Bring Your Own Device*)
  — **GNU/Linux : Ubuntu 24.04 LTS**
    — *upgrade* des Ubuntu 22.04 à faire
  — Mac OS/X, Windows : sans garantie !
— Ligne commande : shell bash, PHP 8, SQLite, …
— *IDE* avec support de PHP, Composer, Twig
  — **Eclipse** (+ PHP Dev. Tools)
— Git ?
— Navigateur
  — **Firefox**
  — Chrome
  — ~~Safari~~

Attention : ce module nécessite beaucoup d'activités pratiques en TP ou TA, et l'équipe enseignante n'a pas la possibilité d'effectuer un support sur les machines personnelles en environnements multiples. Nous avons configuré et testé les outils dans un environnement bien défini, mais supporter d'autres configurations avec toutes leurs subtilités n'est pas soutenable pour un module en promo entière.
Nous recommendons fortement l'utilisation d'un système GNU/Linux dans une version équivalente à la configuration standard GNU/Linux proposée par la DISI sur les machines des salles de TP (Ubuntu 22.04).
Même si les technologies PHP étudiées fonctionnent en principe sur d'autres systèmes d'exploitation, nous ne pourrons pas en faire le support.
Il conviendra de vous organiser pour les activités hors-présentielles pour pouvoir utiliser un système GNU/Linux.
Pour celles/ceux travaillant sur des machines personnelles, nous recommendons d'installer un système GNU/Linux dès que possible (en *dual boot* ou machines virtuelles).
De même pour l'environnement de programmation : nous nous appuierons sur les fonctionnalités de l'environnement de développement Eclipse, qui sera équipé des modules *PHP Development Tools* (PDT) et Symfony, afin d'offrir des fonctions de support au développement en PHP, avec Composer, Twig, CSS, YAML, etc.
Vous serez libres d'utiliser un autre environnement, du moment qu'il offre des fonctionnalités équivalentes, mais nous ne pourrons vous assister au niveau de l'équipe enseignante, si tout ne fonctionne pas comme prévu.

3 Séq. 1 (2/3) : Langage PHP

Objectifs de cette séquence

L'objectif de cette séquence du cours est de présenter le premier grand ensemble de technologies de mises en œuvre qui vont nous servir dans les séquences pratiques du cours, autour du langage PHP.

3.1 Généralités sur le langage PHP

En introduction de cette séquence, voici, pour votre information quelques éléments relatifs aux choix des technologies du monde PHP qu'on emploiera dans ce module.

3.1.1 Survol rapide

- Pas un cours pour enseigner le langage : apprentissage en autonomie
- Quelques éléments pour planter le décor

Les bases du langage seront étudiées en travail autonome, sur des ressources externes.

3.1.2 Faire tourner des programmes sur le serveur Web

- Besoin : Programmer des applications (couche traitements, qui s'exécute côté serveur)
- Choix d'un **langage** ?
- Choix d'une **technologie associée** ?

Dans ce module, notre but est de pouvoir faire tourner une application sur un serveur Web.
De nombreux langages et environnement de développement Web sont disponibles.
Pour une découverte de la programmation des applications Web par la pratique, nous avons fait le choix du **langage PHP** et de l'**environnement Symfony**, pour leur qualités dans un contexte pédagogique.
Une grande partie des aspects techniques étudiés se retrouveront dans d'autres langages ou *frameworks*.

3.1.3 Langage choisi : PHP

- Langage nouveau (pour vous)
- « Proche » de Java (ou C)
- Interprété (comme *Bash* ou *Python*)

Mais aussi des outils…

3.1.4 Mais pourquoi tant de haine ?

Cf. « Why developers hate PHP » par Mehdi Zedi

- Vous aimez apprendre des choses à la mode ?
  vous allez voir des choses modernes, si si (objet)
- PHP est un vieux langage à la mauvaise réputation
- Vous allez devoir bosser (de plus en plus) sur des vieux trucs : importance de la *maintenance* vs innovation (ou innovation dans la maintenance) : *ODD* ?
- Les outils qu'on vous présente (*Symfony*) sont à l'**état de l'art** (objet, génie logiciel)

— Pédagogie : **complexité maîtrisée** (application des concepts en 2A, chaque chose en son temps)

PHP est un vieux langage, et de nombreux tutoriaux ou manuels expliquent comment le prendre en main.
PHP est parfois dénigré car il supporte encore des façons de programmer qui sont déconseillées depuis longtemps.
Ce n'est pas une raison pour jeter PHP à la poubelle. Il y a de bonnes pratiques pour utiliser PHP de façon moderne et produire des programmes de bonne qualité.
Nous présentons rapidement quelques-uns de ces éléments, dans ce cours, que nous reverrons plus en détail dans les phases pratiques.

3.1.5 Symfony : PHP moderne

PHP « comme il faut » S'appuyer sur un cadriciel (*framework)* moderne comme Symfony

— orienté objet
— fonctionnel
— injection de dépendances, conteneurs
— *templates*
— PHPDoc
— tests
— …

`http://www.phptherightway.com/`

La plupart des cours ou tutoriaux qu'on trouve sur PHP sur le Net datent un peu… pourtant il existe d'excellents documents actuels, comme le site ci-dessus « *PHP the right way* » de Josh Lockhart *et al.*

3.1.6 Le cadriciel moderne : Symfony 6

— *Framework* de référence
— Assemblage de beaucoup de bibliothèques
— Modèle de composants objet évolué
— Documentation
— Communauté
— **Environnement de mise au point**

`https://symfony.com/`

On utilisera Symfony 6, la version stable actuelle, dans le cours.
Attention aux documents ou tutoriaux portant sur des versions antérieures, qui foisonnent sur le Web, et ne s'appliquent pas forcément à cette version récente de Symfony.

Une *licorne* française - SensioLabs `https://sensiolabs.com/`

« *SensioLabs, une des plus belles réussites françaises du web* » –

"HistoiresdeFrance, chapitre 21 (2016)"

In #HistoiresdeFrance, chapitre 21 (*post X du Gouvernement Français*, décembre 2016 - cf. sauvegarde sur/ Internet Archive)

3.1.7 Autres frameworks :

PHP Laravel, CodeIgniter, CakePHP, Zend, …
Ruby Ruby on Rails, Sinatra, …
Node.js Meteor, Express.js, …
Python Django, Flask, …
Java Spring, Struts,

3.2 Langage PHP

L'apprentissage des bases du langage sera fait en autonomie dans la prochaine séquence hors-présentielle.
Nous allons présenter ici quelques aspects avancés du langage et de son environnement, non couverts par les tutoriels de base, et qui serviront dans la suite du cours dès la prochaine séquence de travaux pratiques.
Il n'est pas utile de comprendre tous les détails dès maintenant, mais de pouvoir s'y référer ultérieurement.

3.2.1 Langage

— Syntaxe style C / Java
— Objets
— Interprété
— Héritage contexte Web, CGI (« *PHP : Hypertext Preprocessor* »)
— Depuis 1994 (PHP 8 depuis fin 2020)
— Versions pour ce cours : >= 8.2

PHP est l'un des langages les plus populaires sur le Web. Cf. Usage statistics and market share of PHP for websites pour quelques éléments chiffrés.
Les versions de PHP que vous allez utiliser devraient typiquement être PHP 8, comme sur machines DISI salle de TP
Pour plus de détails sur les *elePHPants* (mascote du langage), voir Comment et pourquoi la mascotte PHP est-elle venue à la naissance ? L'histoire secrète d'ElePHPant ! et A Field Guide to Elephpants - Detailing the attributes, habitats, and variations of the Elephpas hypertextus.

Hello world

```
<html>
  <head>
    <title>Test PHP</title>
  </head>
```

```
  <body>
    <?php echo '<p>Bonjour le monde</p>'; ?>
  </body>
</html>
```

ou bien :

```
<?php

  $html = "<html><head><title>Test PHP</title></head><body>";
  $html .= "<p>Bonjour le monde</p>";
  $html .= "</body></html>";
  print($html);
```

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

PHP *inline*

- Mélanger la présentation (HTML) et le code (PHP)… **c'est mal** : maintenable ?
- On verra comment faire autrement dans une prochaine séquence.

Cf. les gabarits (*templates*) en séquence 3.

3.2.2 La documentation

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

La documentation de référence est à préférer, en cas de doute, notamment pour les fonctions de la bibliothèque standard.
Cependant, on peut douter de sa qualité pédagogique pour l'apprentissage.
On verra que la documentation de Symfony est par contre d'excellente qualité en général.

3.2.3 Syntaxe

Vous allez commencer à l'apprendre en hors-présentiel, cette semaine
Vous devriez comprendre ce que je vais montrer

3.3 Syntaxe objet

Ce cours ne contient pas de manuel d'introduction à la programmation en PHP.
Nous présentons ici quelques éléments particulièrement importants du modèle objet, et des fonctionnalités avancées utiles dans un projet Symfony.

3.3.1 Appel de méthode

```
$io = new SymfonyStyle($input, $output);

$io->title('list of todos:');
```

— Syntaxe java : `instance.methode()`
— Syntaxe PHP : `$instance->methode()`

3.3.2 Constructeur, propriétés internes

```php
class MyCommand
{
        private $todoRepository;

        public function __construct($repository)
        {
                $this->todoRepository = $repository;
        }

        protected function execute()
        {
                $todos = $this->todoRepository->findAll();
        }
```

— Syntaxe java : `this.propriete`
— Syntaxe PHP : `$this->propriete`

3.3.3 Valeur de retour

```php
protected function execute(SymfonyStyle $io): int
{
        $todos = $this->todoRepository->findAll();

        if( empty($todos) ) {
                return Command::FAILURE;
        }
        else {
                $io->title('list of todos:');
                $io->listing($todos);
        }

        return Command::SUCCESS;
}
```

— `Command::SUCCESS` et `Command::FAILURE` sont des propriétés statiques de la classe `Command` (des constantes)
— `empty()` : tableau vide. Mais en fait « *A variable is considered empty if it does not exist or if its value equals false.* » (Cf. empty — Determine whether a variable is empty)

3.3.4 Surcharge / héritage

```php
use Symfony\Component\Console\Command\Command;

class ListTodosCommand extends Command
{
        public function __construct($repository)
        {
                $this->todoRepository = $repository;

                parent::__construct();
        }
```

— Hérite d'une classe existante de la bibliothèque Symfony
— Surcharge du comportement par défaut dans le constructeur, donc appel au constructeur de la classe parente (ici `Command`)

3.3.5 Espace de noms

```php
namespace App\Command;

use Symfony\Component\Console\Command\Command;
use App\Entity\Todo;
use Symfony\Component\Console\Input\InputOption;
use Symfony\Component\Console\Output\OutputInterface;

class MyCommand extends Command
{
        // ...
        public function __construct(ManagerRegistry $manager)
        {
                $this->todoRepository = $manager->getRepository(Todo::class);

                parent::__construct();
        }

        protected function execute(InputInterface $input, OutputInterface $output): int
        {
                // ...
                return Command::SUCCESS;
        }
```

— `Todo::class` : l'objet classe lui-même (appel de méthodes statiques, etc.). Pas une instance de cette classe.

3.3.6 *Docblocks* PHPDoc

— Commentaires améliorés
— méta-informations

```php
/**
* Classe "Circuit" du Modèle
*/
class Circuit
{
...
    /**
* Set description
*
* @param string $description
*
* @return Circuit
*/
    public function setDescription($description)
    {
        ...
```

Les *Docblocks* PHPDoc étendent le langage PHP en fournissant le moyen d'ajouter des méta-informations dans des « commentaires améliorés » associés au code PHP.
Ils sont particulièrement utiles pour embarquer la documentation des entités appelées, pour s'y référer quand on programme les entités appelantes : la documentation est accessible dans l'éditeur/IDE, qui peut proposer des fonctions d'auto-complétion et de détection des erreurs dans le code.
C'est particulièrement utile quand on est dans un langage interprété (pas de compilateur pour éviter certaines erreurs), et quand on n'a pas le temps de lire les manuels des bibliothèques !
Par exemple, dans l'exemple ci-dessus, l'éditeur proposera directement la saisie d'une chaîne de caractères quand le programmeur saisit une invocation de la méthode `setDescription()`.
Ce mécanisme et sa syntaxe sont identiques à d'autres langages de programmation, avec par exemple JavaDoc en Java.
Cf. documentation de PHPDocumentor : `https://docs.phpdoc.org/` pour plus de détails.

3.3.7 Attributs (> PHP 8.x)

```php
use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;

#[AsCommand(
    name: 'app:list-todos',
    description: 'List tasks',
)]
class ListTodosCommand extends Command
{
```

Les attributs ajoutent des méta-données dans le code source PHP, et permettent une programmation réflexive (introspection).
On « décore » le code avec des propriétés utiles, par exemple des contraintes sur la validité, ou le stockage des données.
On verra que le composant Doctrine, ou le routeur Symfony utilisent ces attributs pour « décorer » le code, facilitant la maintenance en ne dispersant pas à différents endroits les propriétés de l'application.
Il est préférable d'utiliser un éditeur récent compatible (comme *Eclipse PDT* en version supérieure à 2023-06), pour faciliter la reconnaissance de ces attributs et leurs ajouts dans le code.

3.4 Installation des outils et bibliothèques

Nous installerons/utiliserons la variante CLI (*Command Line Interface*)

- **Interpréteur ligne de commande** (`php-cli`)

  ```
  $ php helloworld.php
  ```

- ~~Interpréteur invoqué par le serveur HTTP (`php`)~~
- Bibliothèques :
  - `php-sqlite3`
  - `php-intl`
  - `php-xml`
  - …

La distribution PHP peut être installée en deux variantes selon le contexte d'utilisation.
Attention à bien spécifier la bonne variante à l'installation.
Dans le cours, nous utiliserons principalement la première, pour le développeur PHP : pour la ligne de commande (installer le paquetage `php-cli`).
La seconde est plutôt utilisée pour la mise en production sur des serveurs Web.

3.4.1 Bibliothèques complémentaires

Logiciels faisant partie de l'écosystème de bibliothèques, composants, *frameworks* PHP.
Sources :

- **Composer** : utiliser des bibliothèques libres
- ~~PEAR (*PHP Extension and Application Repository*)~~

Les bibliothèques distribuées via PEAR sont typiquement déjà installées (précompilées) via des paquetages de distributions GNU/Linux. Elles ne sous intéresseront pas dans le contexte de ce cours.
On va par-contre utiliser *Composer*, dont voici quelques caractéristiques.

3.4.2 Composer

Gestionnaire de paquetages PHP.

`https://getcomposer.org/`
Il fournit :

- gestionnaire de dépendances entre bibliothèques (et entre versions),
- téléchargement des paquetages des bibliothèques depuis *packagist*
- *autoloader* qui facilite les déclarations et les chargements associés au démarrage des programmes,

Composer est le gestionnaire de paquetages PHP, comme il en existe pour de nombreux langages de programmation.
On va utiliser Composer intensément quand on développera avec le *framework* Symfony. Les bibliothèques PHP utiles à un projet sont téléchargées grâce à Composer et installées dans le répertoire du projet.

Packagist (optionnel) `https://packagist.org/`
Référentiel en ligne de paquetages (*packages*) pour Composer :

- contributif
- différentes versions de chaque paquetage

Nous mentionnons l'existence de *Packagist* uniquement pour votre culture. On s'en servira via Composer, de façon transparente.
Packagist constitue un dépôt de paquetages contributif, où des développeurs PHP peuvent publier leurs bibliothèques (libres).
N'importe qui peut y publier ses contributions.

Packages registered	374 425
Versions available	4 008 934
Packages installed	84 730 075 331

(since 2012-04-13)
(source : `https://packagist.org/statistics`, au 28/06/2023) Une fois qu'un paquetage est référencé sur packagist, il est téléchargeable par d'autres développeurs PHP grâce à Composer.
Pour publier un paquetage, il suffit par exemple de rendre public un référentiel Git, ce qui facilite la contribution de paquetages PHP, en logiciel libre, par exemple depuis GitHub.

Descripteur `composer.json` Exemple de description d'un *projet local* :

```
{
    "type": "project",
    "require": {
        "php": ">=8.1",
        "symfony/console": "^6.4",
        "symfony/flex": "^1.0",
        "symfony/framework-bundle": "^6.4"
    },
    "require-dev": {
        "symfony/dotenv": "^6.4"
    },
    "config": {
        "preferred-install": {
            "*": "dist"
        },
        "sort-packages": true
    },
    "autoload": {
        "psr-4": {
            "App\\": "src/"
        }
    }
}
```

Chaque projet contiendra donc un fichier `composer.json` dont le contenu est renseigné par les développeurs pour y définir les dépendances particulières à télécharger.
Cet exemple (fictif) définit l'environnement nécessaire au développement d'une application Symfony.
On trouve les règles suivantes :

- dépendance sur PHP 8
- dépendance des bibliothèques nécessaires à l'exécution de l'application déployée : `symfony/console`, `symfony/flex` et `symfony/framework-bundle` dans leurs versions respectives (pour Symfony 6.3)
- dépendance sur la bibliothèque `symfony/dotenv` pour l'environnement de développement
- configuration de l'auto-loader pour associer l'espace de noms `App` au contenu du sous-répertoire `src/` de l'application (les codes source PHP écrites par le développeur s'y trouveront).
- etc.

En général, on s'inspire d'une documentation, ou on utilise un générateur, pour ne pas avoir à comprendre tous ces détails.
Voyons maintenant comment on l'utilise.

Installation des bibliothèques

1. Le développeur lance :

```
$ composer install
```

2. Analyse des règles contenues dans `composer.json`

3. Résultat est téléchargé dans `vendor/`

```
autoload.php
autoload_runtime.php
bin/
composer/
doctrine/
easycorp/
friendsofphp/
laminas/
masterminds/
monolog/
nikic/
psr/
symfony/
twig/
```

```
Loading composer repositories with package information
Updating dependencies (including require-dev)
Package operations: 20 installs, 0 updates, 0 removals
  - Installing symfony/flex (v1.0.89): Downloading (100%)
Enable the "cURL" PHP extension for faster downloads

Prefetching 19 packages
  - Downloading (100%)

  - Installing symfony/polyfill-mbstring (v1.9.0): Loading from cache
  - Installing symfony/console (v4.1.3): Loading from cache
  - Installing symfony/routing (v4.1.3): Loading from cache
  - Installing symfony/polyfill-ctype (v1.9.0): Loading from cache
  - Installing symfony/http-foundation (v4.1.3): Loading from cache
  - Installing symfony/event-dispatcher (v4.1.3): Loading from cache
  - Installing psr/log (1.0.2): Loading from cache
  - Installing symfony/debug (v4.1.3): Loading from cache
  - Installing symfony/http-kernel (v4.1.3): Loading from cache
  - Installing symfony/finder (v4.1.3): Loading from cache
  - Installing symfony/filesystem (v4.1.3): Loading from cache
  - Installing psr/container (1.0.0): Loading from cache
  - Installing symfony/dependency-injection (v4.1.3): Loading from cache
  - Installing symfony/config (v4.1.3): Loading from cache
  - Installing psr/simple-cache (1.0.1): Loading from cache
  - Installing psr/cache (1.0.1): Loading from cache
  - Installing symfony/cache (v4.1.3): Loading from cache
  - Installing symfony/framework-bundle (v4.1.3): Loading from cache
  - Installing symfony/dotenv (v4.1.3): Loading from cache
Writing lock file
Generating autoload files
```

Composer construit le graphe transitif des dépendance en partant des 4 dépendances `symfony/console`, `symfony/flex`, `symfony/framework-bundle` et `symfony/dotenv` qui ont été mentionnées explicitement dans `composer.json`, et télécharge les 20 paquetages qui en résultent (dépendances de base du noyau du *framework* Symfony).
Le code des différentes bibliothèques téléchargées est alors extrait dans des sous-répertoires du répertoire `vendor/` à la racine du projet.
L'extension Flex pour Composer (`symfony/flex`) est alors mise à contribution pour générer certains fichiers utiles au projet (cf. `https://github.com/symfony/flex`), si nécessaire (renseigner des valeurs par défaut, etc.).
Composer comporte bien d'autres fonctions, dont certaines seront utilisées plus tard, comme la création d'un squelette d'application, avec `composer create-project`.
Maintenant que tout le code des bibliothèque est présent, voyons comment il est chargé à l'exécution.

3.4.3 Chargement des bibliothèques via l'*autoloader (optionnel)*

Voici le fonctionnement du chargement via l'*auto-loader* lié à *Composer*, pour votre culture.
En pratique tout cela à transparent.
Dans l'exemple ci-dessous, le programme PHP `index.php` utilise l'*auto-loader* pour charger les bibliothèques nécessaires à son exécution.
Voici une bibliothèque dont le code est installé dans `vendor/symfony/http-foundation/Request.php` :

```php
namespace Symfony\Component\HttpFoundation;

...

    class Request {
```

et qui est chargée « automatiquement » via ce programme `index.php` :

```php
<?php

require __DIR__.'/vendor/autoload.php';

...

$request = Symfony\Component\HttpFoundation\Request::createFromGlobals();
```

Ainsi lorsqu'il utilise une classe comme `Symfony\Component\HttpFoundation\Request`, celle-ci est trouvée par l'interpréteur, car présente dans l'espace de noms `Symfony\Component\HttpFoundation` que l'*auto-loader* aura trouvé dans la déclaration d'espace de noms présente dans le fichier source `vendor/symfony/http-foundation/Request.php`.

Les programmes utilisent des identifiants d'espaces de noms PHP et non le chemin en dur des fichiers sources, ce qui rend le code maintenable en permettant la restructuration de l'arbre des fichiers sources des bibliothèques.

Cf. `https://www.phptherightway.com/#namespaces`.

3.5 Mettre au point et tester le code

- Langage interprété
- Trouver les bugs avant exécution ?
- Tester (systématiquement)
- IDE (*Integrated Development Environment*), pour détecter les erreurs quand on tape le code

Les tests sont particulièrement importants dans un langage interprété comme PHP.
Avec un langage compilé (comme Java) un grand nombre de problèmes et de bugs sont identifiés lors de la phase de compilation et doivent être résolus assez tôt.
Avec un langage interprété, c'est uniquement quand le programme fonctionne que ceux-ci seront détectés. Mieux vaudrait que ça soit en phase de mise au point, quand un développeur compétent est disponible, plutôt qu'une fois en production.
Il est aussi intéressant d'utiliser un éditeur ou un environnement de développement intégré offrant un support du langage, pour identifier certaines erreurs au plus tôt.

3.5.1 Mise au point : affichage

Afficher des traces d'exécution

- `print()` est votre ami ?
- mais on peut faire mieux !

La programmation avec PHP est souvent très itérative et nécessite de maîtriser les bonnes pratiques pour la mise au point.
Premier outil pour la mise au point : afficher des traces.
Astuces « débug » sur sortie standard :

- `echo` / `print( )` : basique, pourvu qu'il y ait une sérialisation en chaîne de caractères
- `print_r( )` : affichage formaté

```php
echo '<pre>';
print_r($data);
echo '</pre>';
```

- `var_dump( )` : très détaillé… MAIS **attention aux récursions** !

Attention, les affichages ne sont pas toujours visibles : capture de la sortie standard, formatage des réponses HTTP : en-têtes / corps de réponse…
Une astuce pour capturer le formattage de `print_r` dans une variable :

```php
$affichage = print_r($data, 1);
```

3.5.2 `dump()` dans Symfony

Utiliser `dump()`

```php
dump($myarray);
```

```
array:5 [▼
  "a simple string" => "in an array of 5 elements"
  "a float" => 1.0
  "an integer" => 1
  "a boolean" => true
  "an empty array" => []
]
```

Dans les *frameworks* comme Symfony on verra des outils permettant de mettre au point plus confortablement, sans ces inconvénients, comme l'utilisation de la fonction `dump( )`.
Symfony nous apportera des fonctionnalités d'environnement de développement ou tests.

3.5.3 Environnement de développement local dans Symfony

Ainsi, on utilisera au quotidien, pour la mise au point un serveur Web local.
Il permet de tester le code en direct, en faisant un rechargement automatique des fichiers modifiés.
Il s'accompagera de l'utilisation d'une base de données locale (*SQLite*) permettant de tester complètement le fonctionnement de l'application sur l'ordinateur du développeur.

3.5.4 Tests PHPUnit

- environnement de tests : `PHPUnit`
- systématiser les tests :
  - conformité
  - non-régression
- Tester avant de déployer en production

Ne sera pas utilisé dans le cours (faute de temps)

Avoir une démarche de tests bien définie est indispensable Langage **interprété**
La meilleure technique pour assurer que le code est testé, est de l'accompagner d'une suite de tests automatisée, qui rend explicites les tests du programme.
Nous n'aborderons pas le domaine des tests dans le présent cours faute de temps, même si les *frameworks* comme Symfony apportent un support pour les tests assez élaboré.
En PHP, c'est l'environnement **PHPUnit** qui est utilisé. Il fonctionne de façon très similaire à JUnit de Java, par exemple.
Le sujet des tests sera abordé en détail dans le module optionnel CSC 4102 « Introduction au Génie Logiciel Orienté Objets ».

Take away

— PHP moderne
— Syntaxe objet
— Outils du développeur : *Composer*, `dump()`

4 Séq. 1 (3/3) : Accès aux données avec l'ORM Doctrine

Objectifs de cette séquence

Cette séquence de cours magistral abordera l'un des outils qui est utilisé dès les premiers TP, et qu'on utilisera ensuite tout au long du module, l'ORM *Doctrine*, qui gèrera la couche d'accès aux données dans nos développements en PHP avec Symfony.

4.1 L'ORM Doctrine

Cette section présente succinctement les fonctionnalités de l'ORM Doctrine, pour détailler le fonctionnement des mécanismes qu'on utilise dans le cours, et approfondir des éléments plus avancés.

4.1.1 Pourquoi une base de données ?

Exécution typique d'une application Web

1. L'application se lance (arrivée requête HTTP)
   — Charge des données en mémoire
2. Elle répond aux requêtes (envoi réponse HTTP)
   — Impact sur données
3. L'application s'arrête
   — Enregistre les données mises à jour

Mais aussi interaction entre utilisateurs sur données partagées
Programmer :
— Modèle de données **en mémoire** (à chaud) :
programmation objet
— Modèle de données en base de données (à froid, persistant) :
base de données relationnelle (SGBD, SQL)

4.1.2 Rôle d'un *ORM*

ORM *Object Relational Mapper*
— Principe ORM
  — Manipuler les données de l'application via des **classes / objets**
  — Implémentation du **Modèle de données** applicatif (en mémoire)
  — Persistence dans un SGBD
  — Conversion d'un modèle **objet** en un modèle **relationnel**

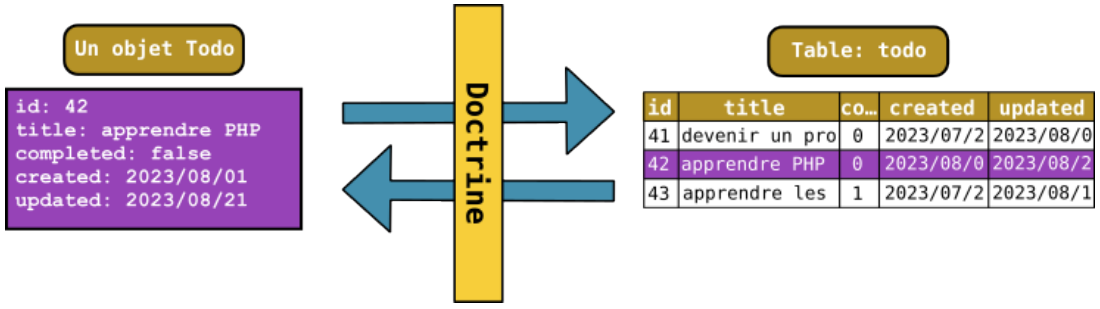


Figure 1 – Conversion objet - relationnel

Object Relational Mapping peut se traduire par *Mapping* (en bon franglais - sic) ou conversion objet-relationnel.
En principe, vous maîtrisez le modèle relationel, qui est un pré-requis de ce module. De même pour le modèle objet.
Par contre, la conversion objet-relationnel a été étudiée, mais probablement pas approfondie, d'où la nécessité de l'approfondir ici.

Doctrine, l'*ORM* standard en PHP

`https://www.doctrine-project.org/projects/orm.html`

- composant standard applis PHP
- bien intégré avec Symfony :
  - gestion modèle de données
  - intégration avec formulaires saisie données
  - assistant génération de code dans Symfony
  - …

Doctrine est le composant d'ORM standard qui est intégré dans Symfony. Il gère l'accès et la persistance des données de notre application en base de données.
Il apporte de nombreuses fonctionnalités, en particulier :

- la génération d'une base de données relationnelle, à partir du Modèle objet défini par le programmeur dans les classes PHP. En général, on n'a pas besoin d'écrire quoi que ce soit en langage SQL, ou de s'occuper des détails de tel ou tel SGBD, quand on développe une application simple avec Symfony. Doctrine s'occupe « de tout » pour nous.
- l'articulation automatique avec les formulaires de saisie de données de Symfony, pour le support du typage des données (vérification de contraintes de saisie, sécurité, etc.)

Doctrine est un des composants de Symfony (`http://symfony.com/doc/current/doctrine.html`), mais il est aussi utilisable dans des applications PHP, en dehors de Symfony (`http://www.doctrine-project.org/projects/orm.html`).
La plupart des programmes PHP écrits avec Symfony qui seront étudiés dans ce cours utiliseront Doctrine. Il est donc utile de mieux savoir comment fonctionne Doctrine.

4.1.3 Concevoir les classes du modèle

Conception **orientée objet**

- Classes (PHP)
- Propriétés *mono-valuées* des classes :
  types de bases + *références* à des instances d'autres classes
- Propriétés *multi-valués* :
  Collections d'objets (ou de références d'objets)

La conception orientée objets n'est pas un domaine trivial. On peut mettre à profit les connaissances en modélisation UML, par exemple.
Dans le module on essaiera de traiter uniquement des structures de données relativement simples.
On étudiera de façon plus avancée la Conception Orientée Objet dans le module CSC4102 « *Introduction au Génie Logiciel Orienté Objet* ».
Examinons le modèle orienté objet d'une application simple, pour rappeler des éléments de vocabulaire.

Exemple modèle de données objet Todo Modéliser des **tâches** : **classe** `Todo`

Propriétés :

- `title` : chaîne
- `completed` : booléen
- `created`, `updated` : dates

```php
class Todo
{
        private string $title;
        private bool $completed;
        private DateTime $created;
        private DateTime $updated;
```

Modéliser des **projets** : **classe** `Project`

Propriétés :

- `title` : chaîne
- `description` : chaîne

```php
class Project
{
        private string $title;
        private string $description;
```

Association 1-n entre `Project` et `Todo`

- les tâches d'un projet (`Project::todos`)

```php
class Project
{
        private string $title;
        private description $completed;

        private array $todos = array();
```

- le projet d'une tâche (`Todo::project`)

```php
class Todo
{
        private string $title;
        private bool $completed;
        private DateTime $created;
        private DateTime $updated;

        private Project $project;
```

En PHP, les types de base des propriétés mono-valuées sont disponible « de base », comme les chaînes, les entiers, ou les booléens. Laissons de côté les dates qui sont plus complexes (vu les soucis d'internationalisation, notamment).
Par contre, comme pour la programmation en Java, il est nécessaire de comprendre comment sont gérées les **associations** entre instances de différentes classes, pour les implémenter via des propriétés.
En PHP objet, la notion de **référence** existe naturellement, de façon assez similaire à ce qu'on trouve dans d'autres langages comme Java.
Une instance de `Todo` aura donc une propriété `project` (par convention `Todo::project`) qui pourra référencer une instance de `Project`, ou valoir `null` si elle n'est pas définie.
Pour ce qui concerne les **collections** (qui permettent de gérer des propriétés multi-valuées), la standardisation en PHP a été progressive, et on va s'appuyer, dans ce cours, sur les structures de données objet fournies par Doctrine, comme des « tableaux » / listes comme ArrayCollection.

Raffiner Quelle est la « force » de l'association ?
- association : tâches sans projet possibles ?
- composition : pas de tâche sans projet ?

Pourra être approfondi dans CSC4102 « *Introduction au Génie Logiciel Orienté Objet* »

4.1.4 Vous êtes en terrain connu

Vous maîtrisez déjà :
- Conception du **modèle des données en Objet** (*comme découvert en CSC3101*)
- Programmation **objet en PHP** (*assez proche de Java, en fait, même si pas compilé*) :
  - références
  - collections

À l'exécution, sous le capot : génère des requêtes SQL dans SGBD relationnel (*appris en CSC3601*)… mais pas besoin de les programmer

Le principe d'un **ORM** *Object Relational Mapper* est de permettre de concevoir une application dans le paradigme objet, indépendamment de la technologie de bases de données sous-jacente.
La plupart des concepts que vous connaissez, comme les **références** vers des objets et les **associations** se retrouvent dans le modèle objet de PHP.
Les structures de données disponibles en PHP pour gérer des collections d'objets sont légèrement différentes de celles d'autres langages (p. ex. Java), mais reposent en général sur des tableaux (équivalents à des listes) ou des tableaux associatifs (dictionnaires). On verra que Doctrine apporte des classes particulières pour les structures de données, mais leur manipulation repose en grande partie sur la syntaxe de manipulation des tableaux déjà vue dans l'apprentissage de la syntaxe PHP en autonomie.
L'ORM réalise les opérations nécessaires pour assurer la conversion du modèle **objet** (références, associations, collections) en un modèle **relationnel**, pour permettre un stockage persistant, et donc l'intégration avec d'autres applications (via le langage SQL).

Utiliser l'ORM
- Ne pas écrire les requêtes SQL de chargement / modification
- **Programmer en objet** avec Doctrine
- L'ORM (*Object Relational Mapper*) détecte les données nouvelles ou modifiées et génère le SQL sous le capot

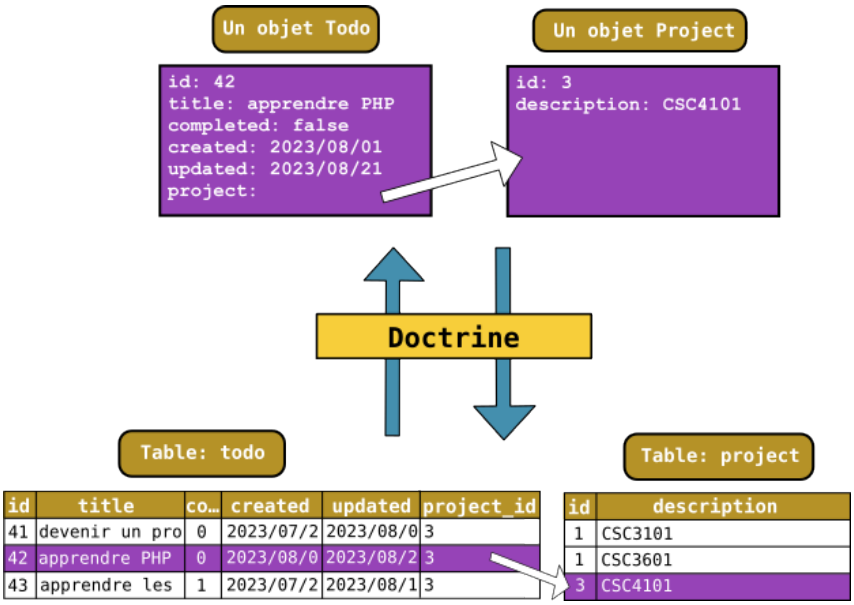
Le composant d'ORM permet de manipuler les données de l'application via des **objets** (instances de classes PHP).
Le programmeur réalise donc l'implémentation du **Modèle** applicatif grâce à des structures de données objet instanciées en mémoire, quand l'application s'exécute.
Sous le capot, l'ORM attribue des identifiants qui sont nécessaires au fonctionnement du modèle physique des données (dans le SGBD relationnel), alors que les objets PHP sont manipulés uniquement par leur référence (adresse mémoire).

Oublier SQL ?

— Pas toujours si simple
— Réviser un peu CSC3601 « *Modélisation, bases de données et systèmes d'information* » ?

Comprendre pour *debugger* si tout ne marche pas comme prévu automagiquement.

Le développeur Symfony n'est pas obligé d'apprendre SQL, s'il utilise Doctrine.
Mais **Doctrine n'est pas magique pour autant**. Il faut notamment bien comprendre la syntaxe des attributs, et mieux vaut bien connaître **les bases du modèle relationnel** pour ne pas faire n'importe quoi.



id	title	co…	created	updated	project_id
41	devenir un pro	0	2023/07/2	2023/08/0	3
42	apprendre PHP	0	2023/08/0	2023/08/2	3
43	apprendre les	1	2023/07/2	2023/08/1	3

id	description
1	CSC3101
1	CSC3601
3	CSC4101

Figure 2 – Traduction des références en clés étrangères

Les propriétés `Project::todos` et `Todo::project` `OneToMany/ManyToOne` entre un projet et ses tâch une collection, et respectivement une référence Mais si on s'intéresse au stockage en base de d stocker de propriétés multi-valuées dans le mo de stocker la collection des instances de la prop lonne de la table `project`. On pourra seulement à son projet via la clé étrangère `project_id` de tifiant `id` de `project`. Le chargement par jointur collection, à la demande.
Si vous ne maîtrisez pas cet aspect du modèle du mal à comprendre certains soucis de mise a cation des données, et la persistence de ce gen

Exemple : références traduites en clés étrangères

4.2 Coder les classes

Cette section présente succinctement la façon dont on peut coder en PHP les classes du modèle de données de l'application, avec l'aide de Doctrine

4.2.1 Codage des entités du modèle de données

Observons comment a été mis en œuvre le modèle des données de l'application « fil-rouge » *Todo*.

Emplacement du code PHP Classe de l'entité `Todo` :
- classe PHP `Todo`
- fichier source : `src/Entity/Todo.php`
- espace de nommage : module `App\Entity\Todo`

Les classes du modèle de données sont à coder dans un répertoire particulier du projet, et avec un espace de nommage particulier, par convention.
Notons qu'il n'est pas obligatoire de coder « à la main », mais qu'on utilisera aussi beaucoup des générateurs de code, pour aller plus vite et éviter les erreurs.

Classe PHP « naïve »

```php
class Todo
{
    private string $title;
    private bool $completed;

    public function getTitle() {
      // ...
    public function setTitle($title) {
      // ...
// ...
```

Pas de typage des propriétés, en « PHP basique », mais possible en PHP moderne (et objet).

Il n'y a pas de typage strict des propiétés, en PHP, comme dans d'autres langages interprétés. Mais c'est fortement recommandé dans un style de programmation moderne, y compris en objet.
Une fois cette classe disponible dans l'application, on peut gérer des instances de la classe en mémoire, par exemple en créant des objets, en les ajoutant dans un tableau, en parcourant ce tableau, etc.
Chaque objet est identifié par sa référence en mémoire.
Dans ce qui précède, le modèle objet de PHP ressemble fort à ce que vous connaissez déjà en Java.

Classe PHP documentée *Docblocks* PHP « standard » (sans ORM) :

```php
class Todo {
        /**
* @var string task title
*/
        private string $title;

        /**
* @var bool Is the task completed/finished.
*/
        private bool $completed;
```

```
}
```

On voit que le programmeur a ajouté des annotations sous forme de « doc blocks » dans les « commentaires », qui ne sont pas utilisés par PHP en première instance, mais sont destinés à des outils comme PhpDocumentor.
Cela permet d'assister d'autres programmeurs qui utilise cette classe, s'ils utilisent des outils comme un éditeur de texte moderne ou un IDE comprenant ces annotations *PHPDoc*.
Le format de ces annotations *DocBlocks* de PHPDoc n'est pas un simple commentaire PHP : il doit commencer par « `/**` », au lieu d'un simple « `/*` » (cf. What does a DocBlock look like ?)
On retrouve la même syntaxe générale que pour Java avec JavaDoc.

4.2.2 Introduction de la persistance avec Doctrine

Sur la base de ce code objet standard PHP, on ajoute des méta-données qui introduisent des contraintes sur les classes et leurs propriétés, que Doctrine sait exploiter pour réaliser ses mécanismes d'ORM.

Attributs PHP (8 +) Ajout de méta-données pour Doctrine

```php
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
class Todo
{
        #[ORM\Id, ORM\GeneratedValue, ORM\Column]
        private int $id;

        #[ORM\Column(length: 255, nullable: true)]
        private string $title;

        #[ORM\ManyToOne(targetEntity: Project::class,
                        inversedBy: 'todos')]
        private Project $project;
        //...
}
```

```php
#[ORM\Entity]
class Project
{
    #[ORM\Id, ORM\GeneratedValue]
    private int $id;

    #[ORM\Column(type: "text", nullable: true)]
    private string description;

    #[ORM\OneToMany(targetEntity: Todo::Class, mappedBy: 'project')]
    private $todos;
    //...
}
```

Les attributs définissent des règles de typage des valeurs des propriétés, ainsi que la définition de propriétés multi-valuées pour gérer les associations entre instances des classes du modèle comme étant des « tableaux » (listes d'éléments itérables).
Elles permettent aussi de définir des contraintes : propriété obligatoire, cardinalités des associations, compositions, etc.
Cf. Documentation Doctrine pour l'ensemble des fonctionnalités sur le *mapping* des propriétés des objets.
Notons en particulier, ici, la description d'une relation `OneToMany` permettant de gérer une association.
Une fois ces différentes méta-données ajoutées au code, Doctrine peut fonctionner.
Tant qu'on travaille en mémoire, sur des structures de données PHP, ces annotations entrent marginalement en ligne de compte.
Mais quand on souhaite charger ou sauvegarder dans la base de données les objets présents en mémoire, Doctrine fait le lien entre les objets PHP en mémoire et une base relationnelle, via la génération de requêtes SQL.

4.2.3 Générateur de code `make:entity`

Asssistant générateur de code : `make:entity`
Abus fortement recommandé !

4.3 Utiliser les classes du modèle de données

Voyons maintenant la façon dont on peut utiliser ces classes du modèle de données, dans un projet Symfony.
On va examiner comment fonctionnent les mécanismes de Doctrine pour déclencher le chargement des données depuis la base de données.
On verra plus tard, bien plus en détails, les fonctions permettant de sauvegarder les données de l'application.

4.3.1 Programmer le chargement en mémoire

Intéressons-nous d'abord au chargement des **Collections** d'instances, qui correspond au `SELECT ... FROM ...`, où on doit allouer une nouvelle instance d'une classe, pour chaque occurrence du résultat de la requête.

Le *repository* d'instances On gère le chargement des données grâce à un « générateur d'objets » (le *Repository*), associé à une classe Doctrine.

```php
use Doctrine\ORM\Mapping as ORM;

use App\Repository\TodoRepository;

#[ORM\Entity(repositoryClass: TodoRepository::class)]
class Todo
{
        #[ORM\Id, ORM\GeneratedValue, ORM\Column]
        private int $id;
```

Listing 1 : Extrait de `src/Entity/Todo.php`

Le *repository* est codé dans une classe, ici `TodoRepository` (dont le code est présent dans `src/Repository/TodoRepository.php`).
Cette classe utilitaire est associée à notre entité du modèle de données `Todo` via l'attribut `ORM\Entity`.
C'est un générateur d'instances d'une classe. C'est lui, dans Doctrine, qui sait générer des instances, ou des collections d'instances de nos classes PHP.
C'est par son intermédiaire qu'on va charger les données depuis la base relationnelle.

Chargement d'une collection d'instances (`findAll()`) Utilisation pour charger toutes les instances de `Todo` :

```
use App\Entity\Todo;
//...
        protected function execute()
        {
                // récupère une liste toutes les instances de Todo
                $todos = $this->todoRepository->findAll();

                foreach($todos as $todo)
                {
                        //...
                }

        }
```

Listing 2 : Extrait de `ListTodosCommand.php`

La méthode `findAll()` des *repositories* Doctrine permet ainsi de **charger, dans une collection en mémoire**, toutes les instances d'une classe, qui correspondent à toutes les données correspondantes en base de données.
Avec un ORM comme Doctrine, on raisonne en objet. Plus besoin de faire un `SELECT * from todo;` en SQL et d'allouer manuellement des instances de `Todo` en mémoire, et d'y recopier le résultat de chaque ligne du résultat du SELECT. C'est le job d'un *repository*, et on s'appuie sur ses méthodes pour faire cela.

Accès au *repository* de Doctrine

```
use App\Entity\Todo;
use App\Repository\TodoRepository;
//...
class ListTodosCommand extends Command  {
    /**
     *  @var TodoRepository data access repository
     */
    private $todoRepository;

    public function __construct(ManagerRegistry $doctrineManager) {
        $this->todoRepository = $doctrineManager
                              ->getRepository(Todo::class);
        parent::__construct();
    }

    protected function execute(): int {
        // fetches all instances of class Todo from the DB
        $todos = $this->todoRepository->findAll();
        // ...
```

Ce code est un peu complexe, car il met en œuvre des patrons de conception avancés (Symfony est un *framework* moderne, complexe, si on essaye d'en comprendre tous les détails dès le début). Rassurez-vous : on essaye de ne pas réinventer la roue, et on utilise des générateurs de code, ou bien on s'inspire de la documentation.
On pourrait écrire le début de ce code de façon moins condensée, pour mettre en évidence les différentes composantes de Symfony et Doctrine mises en œuvre.
Essayons quand même de comprendre. Accrochez-vous :

1. `ListTodosCommand` est la classe d'une commande que nous avons codé, dans `src/Command/ListTodosCommand.php`, qui hérite d'une classe Symfony, `Command` ;
2. nous surchargeons le constructeur de `Command` pour recevoir en argument, une instance du `ManagerRegistry` de Doctrine (inutile d'approfondir ce mécanisme de la tuyauterie Symfony pour l'instant) :
   — cet utilitaire de Doctrine nous permet de récupérer le *repository* d'une classe de notre modèle de données : `->getRepository(Todo::class)`.
   — on garde la référence de ce repository dans une propriété de notre classe : `ListTodosCommand::todoRepository` (de type `TodoRepository`), pour pouvoir le récupérer lorsque la méthode `execute()` sera appelée
3. une fois que la commande est appelée, cette propriété nous permet d'appeler la méthode `findAll()` du repository de `Todo`.

Ce mécanisme d'injection de dépendances, permet à une classe d'accéder à des services de Symfony depuis son constructeur, et on le retrouvera à différents endroits.
C'est un peu complexe, mais ça fonctionne. Même si on ne comprend pas tous les détails de l'usine à gaz qu'est un cadriciel moderne comme Symfony, on peut faire confiance à la documentation pour nous dire comment faire. Et surtout, le code devient relativement compact à écrire.

4.3.2 Chargement depuis la base de données (`find...()`)

Chargement d'**un seul** objet depuis la base de données.
3 variantes :
— par identifiant
— par critères de sélection
— par un valeur d'une propriété (raccourci)
Encore grâce à la même classe *repository*.

Chargement via l'identifiant : `find($id)`

```
$todo = $this->todoRepository->find($id);
```

génère, via Doctrine :

```
SELECT * FROM ... WHERE ID=[$id]
```

Attention : les identifiants sont uniques, mais un détail d'implémentation (valeur générée par le SGBD).

Cette fois on accède directement à une instance par son identifiant `find($id)`.
Mais le plus souvent, cet identifiant interne à la base de données reste caché dans les mécanismes internes de l'application.

Chargement par sélection sur des propriétés (`findOneBy()`) Chargement d'une instance de `Todo`, étant donnés un *titre* et un *état terminé* (extrait de `ShowTodoCommand.php`) :

```
$todo = $this->todoRepository->findOneBy(
    ['title' => $title,
     'completed' => $completed]);
```

génère une requête SQL style :

```
SELECT ... FROM todo
  WHERE title = [$title]
  AND completed = [$completed]
  LIMIT 1
```

On utilise la méthode `findOneBy` du *repository*, en lui passant en argument un tableau contenant les différents critères de sélection/restriction de la requête à faire dans la base de données.
Vous verrez alors, dans le log, des requêtes du type :

```
[2018-08-07 10:14:47] doctrine.DEBUG: SELECT t0.tid AS tid_1, t0.title AS title_2, t0.completed AS c
```

Doc Doctrine : Documentation Doctrine ORM / Working with Objects / By Simple Conditions.
La documentation indiquée dans le lien ci-dessus permet de retrouver les différents critères qu'on peut passer aux méthodes de chargement d'instances, ou de collections d'instances.
Globalement, tous les critères de sélections qu'on utilise classiquement dans le modèle relationnel sont utilisables, typiquement pour filtrer sur des valeurs de propriétés particulières.

Sélection via *findBy...* suivi du nom de propriété Méthodes `findBy*` nommées d'après les propriétés de la classe (*magic finders*) :
Exemple :

```
$todos = $this->todoRepository->findByCompleted(false);
```

donne :

```
..
  WHERE completed = 0;
```

Avec l'appel d'une méthode « magique » du type `findByPropriété(valeur)`, cette méthode charge « automagiquement » les instances de la classe ayant la propriété « `Propriété` » (issue du nom de la méthode) à la valeur « `valeur` »
Doctrine a généré la clause `WHERE` SQL correspondant à la valeur de la propriété `completed`, pour le `findByCompleted()`.
Notons que ces « *magic finders*« sont assez peu documentés. Mais on trouve une indication dans la documentation de l'API Doctrine sous l'intitulé sibylin »*Adds support for magic method calls.*« .
Une méthode `_call()` des *repository* permet d'intercepter l'appel aux méthodes et de détecter que la méthode `findByCompleted(false)` correspond en fait à un appel à `findBy(array('completed' => false))`.

Méthodes spécifiques de votre modèle applicatif Les classes *repository* sont générées avec des fonctionnalités basiques.
Si besoin, on complète le *repository* pour ajouter des requêtes spécifiques à notre contexte applicatif :

```
class TodoRepository extends ServiceEntityRepository
{
        //...
        public function findLatest($page)
        {
                //...
        }
```

Si besoin, le programmeur dispose aussi d'un langage de requêtage proche de SQL, dans Doctrine. Cf. `https://symfony.com/doc/current/doctrine.html#querying-with-the-query-builder`
Mais la plupart du temps, pour des applications simples, on n'en aura pas besoin, et on exprimera les algorithmes dans une syntaxe objet, avec ces méthodes des *repositories* Doctrine.

4.3.3 Création de la base de tests

Voyons comment fonctionnent les outils de Doctrine qui nous permettent de créer une base de données de tests.

Attention : en *dév* ou en *prod* Outils à utiliser dans env. de développement !

On n'étudie pas ici ce qui serait utilisé lors de la mise en production d'une application, car pour l'instant, notre besoin est de tester l'application en environnement de développement.

Base existante, ou base générée Qui gère la base de données ?
- brancher Symfony sur une base existante
- ou générer une base de données à partir du modèle de données de notre application Symfony

Dans les deux cas Doctrine fait le *job*
Dans notre contexte, on est dans le second cas.

À chaque évolution de notre modèle de données, suite à une modification du code PHP, on re-teste en re-générant une nouvelle base de données à partir du nouveau code.

Génération d'une BD à partir du code
- Recherche des attributs Doctrine `ORM` dans code des classes PHP
- Génération requêtes SQL de création du schéma (*SQL /modeling language*)
  - Spécificités SGBD cible (SQLite, PostgreSQL, MariaDB, …)

```
CREATE TABLE todo (
       id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
       title VARCHAR(255) NOT NULL,
       completed BOOLEAN NOT NULL);
```

Listing 3 : Schéma généré pour SQLite (symfony console doctrine:schema:create -vvv) :

Doctrine sait examiner le code des classes PHP, pour y découvrir les attributs qui le concernent (mécanisme d'instrospection PHP).
L'ORM en déduit les opérations à mettre en œuvre pour gérer la persistance des données, quand le programmeur en aura besoin.
Doctrine peut ainsi générer, en fonction du type de SGBD utilisé dans le projet, un schéma de données relationnel.
Différents types de bases de données (y compris NoSQL, clé-valeurs) sont supportés avec Doctrine, même si nous utiliserons uniquement le modèle relationnel et SQLite, dans le cours.

Commandes de re-génération de la base de données

1. Configurer quelle base de données cible (**SQLite**, MySQL, PostGreSQL, …) : variable dans le fichier `.env`
2. Exécuter :

```
symfony console doctrine:database:create
symfony console doctrine:schema:create
```

La base de données (fichier SQLite) est créée dans le répertoire courant, avec des tables vides.

C'est la configuration de la base de données cible (variable `DATABASE_URL` dans le fichier `.env`) qui indique comment générer le modèle concret, avec une syntaxe SQL peut varier d'un SGBDR à l'autre.

Structure du schéma Voir le schéma de données généré dans la base :

```
$ bin/console doctrine:schema:create --dump-sql
```

Exemple :

```
CREATE TABLE project (
  id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  title VARCHAR(255) NOT NULL,
  description CLOB DEFAULT NULL);

CREATE TABLE todo (
  tid INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
  project_id INTEGER DEFAULT NULL,
  title CLOB DEFAULT NULL,
  completed BOOLEAN NOT NULL,
  -- ...
  CONSTRAINT FK_CD826255166D1F9C FOREIGN KEY (project_id)
    REFERENCES project (id) NOT DEFERRABLE INITIALLY IMMEDIATE);

...
```

Listing 4 : Contrainte d'intégrité référentielle pour clé étrangère `project_id`, pour matérialiser l'association 1-N, avec SQLite

C'est la configuration de la base de données cible qui permet de générer le modèle concret pour la sérialisation des données dans un SGBDR.
La syntaxe SQL des contraintes d'intégrité référentielle change en effet, selon le SGBD.

Relations m-n `ManyToMany` Exemple : étiquettes (`Tag`) sur des tâches (`Todo`)

```
CREATE TABLE todo (
       id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
...);
CREATE TABLE tag (
       id INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL,
...);

-- Génération d'une table de relation
CREATE TABLE todo_tag (
       todo_id INTEGER NOT NULL,
       tag_id INTEGER NOT NULL,
       PRIMARY KEY(todo_id, tag_id),
       CONSTRAINT FK_D767A0BAEA1EBC33
                  FOREIGN KEY (todo_id)
                  REFERENCES todo (id),
       CONSTRAINT FK_D767A0BABAD26311
                  FOREIGN KEY (tag_id)
                  REFERENCES tag (id) );
```

Pour les relations m-n (configurées par l'annotation `ManyToMany`), Doctrine génère une table de relation dédiée, qui groupe les clés étrangères des différents relations concernées.
Par exemple, ici, pour une association entre tâches et étiquettes (classe `Tag` du modèle de données), on obtient une table dont les seules colonnes sont les clés étrangères des deux tables `todo` et `tag`, qui constituent une clé primaire composite.

4.3.4 Programmer la modifications des données

Une application Symfony effectue en général des modifications des données, qu'il faut donc sauvegarder, avant l'arrêt de l'exécution.
Voyons comment on programme cela dans notre code PHP.

Objet + ORM Sauvegarder les ajouts / suppressions / modifications
Deux étapes :

1. la création, modification ou suppression d'instances, **en mémoire**
2. **la synchronisation** entre le nouvel état obtenu et le contenu de la BD

C'est le rôle de l'*entity manager* (em) de l'ORM Doctrine d'effectuer cette synchronisation.

L'ORM sait charger les données, et gérer les modifications également.
On supporte alors l'ensemble des opérations CRUD sur les données du modèle de l'application :
- *Create* (création)
- *Request* / *Retrieve* (chargement)
- *Update* (modification)
- *Delete* (suppression)

On effectue des manipulations en codant en PHP objet, encore une fois.

Création et sauvegarde d'une nouvelle instance

```php
public function createAction(ManagerRegistry $doctrine)
{
        $project = new Project();
        $project->setTitle('CSC4101');
        $project->setDescription("Architectures et applications Web");

        // entity manager
        $entityManager= $doctrine->getManager();

        // indique à Doctrine que vous aimeriez éventuellement
        // sauvegarder ce Projet (mais pas encore de requête)
        $entityManager->persist($project);

        // exécute effectivement la sauvegarde (ex. la requête INSERT)
        $entityManager->flush();

        // L'identifiant est enfin renseigné (généré par le SGBD)
        return new Response("Sauvegardé le projet d'id ".
                                            $project->getId());
}
```

`persist()` permet de « tagger » des instances qui ont été modifiées : nouvelles, modifiées ou à supprimer.
Les instances ont été modifiées en mémoire seulement, pour l'instant. Elles sont en attente de sauvegarde.
Ce n'est qu'à l'exécution du `flush()` que Doctrine effectuera réellement les requêtes de suppression effectives (en SQL), en parcourant toutes les données ainsi « taggées » et en vérifiant ce qui doit être fait : INSERT, UPDATE, ou DELETE.

setter pour collection *...ToMany*

```php
class Project
{
        #[ORM\OneToMany(targetEntity: Todo::Class, mappedBy: 'project')]
        private $todos;

        public function addTodo(Todo $todo) {
                if (!$this->todos->contains($todo)) {
                        $this->todos->add($todo);
                        $todo->setProject($this);
                }
                //...
        }

        public function removeTodo(Todo $todo) {
                if ($this->todos->contains($todo)) {
                        $this->todos->removeElement($todo);
                        // set the owning side to null
                        // (unless already changed)
                        if ($todo->getProject() === $this) {
                                $todo->setProject(null);
                        }
                }
```

Le code ci-dessus est celui généré par `make:entity`.
Pour une collection d'instances liées comme `Project::todo`, il y a deux opérations de modifications qu'on effectue classiquement : ajout ou suppression d'un élément dans la collection.

Qui sauvegarder à l'ajout, pour les entités liées ? `OneToMany` entre `Project` et `Todo`

```php
class Project
{
        public function addTodo(Todo $todo): self
        {
                if (!$this->todos->contains($todo)) {
                        $this->todos->add($todo);
                        $todo->setProject($this);
                }
                return $this;
        }
```

Attention : qui est modifié (pour le `persist()` ultérieur) ?

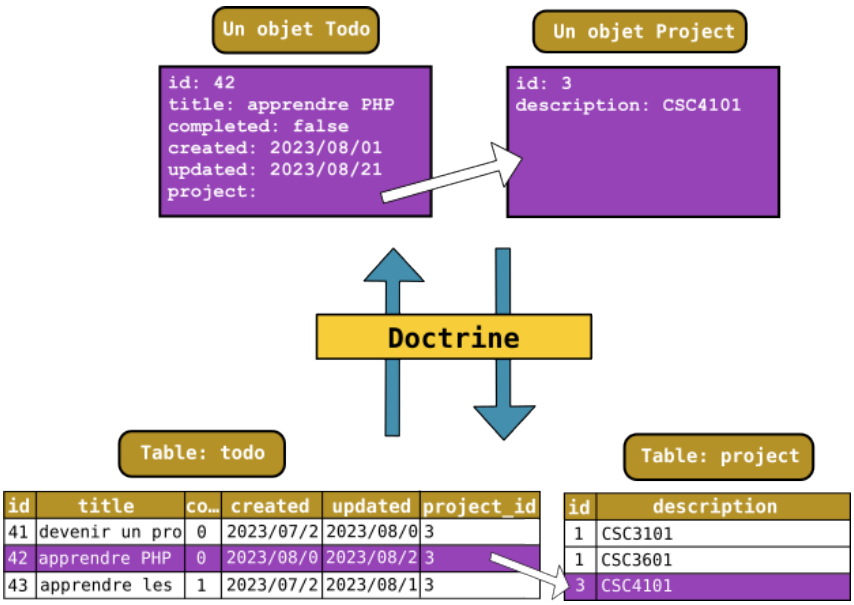


Figure 3 – Rappel du mapping d'une référence 1-N

Supposons qu'on a ajouté une nouvelle `Todo` à un `Project`.
Le projet déjà présent a-t-il besoin d'être modifié, dans la base de données ?
La collection des *Todos* du *Project, une relation /OneToMany,* est gérée en mémoire via `Project::todos`.
À l'ajout en mémoire, cette collection est bien modifiée.
Mais par rapport à la base de données relationnelle, il s'agit d'une propriété multi-valuée calculée, qui n'est pas stockée en base de données (cf. diagramme ci-dessus).
L'instance de projet présente en base n'est effectivement pas modifié : dans le schéma relationnel, ce sont les Todos contiennent une référence à leur projet.
Le `persist()` devra donc opérer sur la Todo.
Il sera donc inutile de faire un `persist()` sur l'instance du `Project`.

Suppression pour les entités liées Code généré par l'assistant `make:entity` :

```php
class Project
{
        public function removeTodo(Todo $todo): self
        {
                if ($this->todos->contains($todo)) {
                        $this->todos->removeElement($todo);
                        // set the owning side to null
                        // (unless already changed)
                        if ($todo->getProject() === $this) {
                                $todo->setProject(null);
                        }
                }
                return $this;
        }
```

Persist ?
Possibilité gestion automatique des associations

Comme dans le cas de l'ajout, pour la suppression, on peut se demander également quelles instances modifiées sont effectivement concernées par le marquage dy `persist()`.
Heureusement, il existe un moyen d'automatiser les sauvegardes.

Propagation du `persist()` en cascade Propagation en cascade :

```
#[OneToMany(... cascade: ['persist', 'remove'] ...)]
```

```
$toto = new Todo();
$project->addTodo($todo)

$entityManager->persist($project);
```

Listing 5 : Exemple de code

Sauvegarde de ses `todos` modifiés

Si on définit une option `cascade` pour l'attribut `OneToMany` de Doctrine, on peut alors gérer les sauvegardes en cascade, par transitivité, en parcourant les collections automatiquement. De même en cas de suppression.

Suppression des instances orphelines orphanRemoval :

```
#[OneToMany(... cascade: ['persist'], orphanRemoval: true)]
```

```
$todo = ...
$todo->setProject(null);

$entityManager->persist($todo);
```

Listing 6 : Exemple de code

Suppression en base
En cas de doute : vérifier les requêtes générées (dans l'outil Doctrine de la barre d'outils Symfony, ou dans les *logs*)

De plus, si on définit l'option `orphanRemoval` pour l'attribut `OneToMany` on obtient une suppression automatique des instances d'entités faibles n'ayant plus d'association vers l'entité forte.
De façon générale, on testera le code du modèle de données avec précaution pour éviter d'oublier des attributs Doctrine, ce qui risquerait d'entrainer des bugs.

4.4 Gérer des données de tests

Examinons l'utilitaire des *DataFixtures* Doctrine qui permet de tester le code du modèle de données et de tester les fonctions de l'application sur des jeux de données de tests.

4.4.1 Initialiser la base avec données de tests

Coder une classe utilitaire *DataFixtures* pour Doctrine
Exemple :

- Chargement dans la base de données :
- Définition des données dans un générateur :

```
private function getProjectsData()
{
        // project = [title, description];
        yield ['CSC4101', "Architectures et applications Web"];
        yield ['CSC4102', "Introduction au Génie Logiciel Orienté Objet"];
}
```

```php
private function loadProjects(ObjectManager $manager)
{
        foreach ($this->getProjectsData() as [$title, $description]) {
                $project = new Project();
                $project->setTitle($title);
                $project->setDescription($description);
                $manager->persist($project);
        }

        $manager->flush();
}
```

Listing 7 : `src/DataFixtures/ProjectFixtures.php`

L'instruction PHP `yield` permet de définir un tableau de n-uplets, qui seront renvoyés successivement en valeur de retour lors de l'appel à une méthode génératrice.

4.4.2 Lancer le chargement depuis la ligne de commande

À refaire à chaque recréation de la base de données dans l'environnement de développement :

```
$ symfony console doctrine:fixtures:load
Careful, database will be purged. Do you want to continue y/N ?y
  > purging database
  > loading App\DataFixtures\ProjectFixtures
```

Les fixtures ne servent que pour initialiser des données de test, pour un administrateur de l'application ou un développeur.

Take Away

— ORM : Entités : classes, objets -> schéma relationnel
— ORM : Chargement des objets en mémoire
— Gestion des données liées dans les associations
— Programmer les modifications synchronisées dans la BD
— Outils de génération de base de données relationnelle
— Outil de données de tests *Fixtures*

Postface

Crédits illustrations et vidéos

— illustrations mapping Doctrine empruntées à la documentation Symfony

5 Séq. 2 (1/3) : Concepts fondamentaux, architecture d’application Web 3 couches

Objectifs de cette séquence

Cette séquence de cours magistral présente les grands éléments du contexte du cours :

1. le *World Wide Web*
2. le dialogue entre clients et serveurs Web
3. les éléments principaux de l'architecture d'application à 3 couches

5.1 Le World Wide Web

On va balayer rapidement les concepts informatiques principaux qui caractérisent le Web.
Ils seront revus et détaillés dans les prochaines séquences.

5.1.1 *Web* = Toile

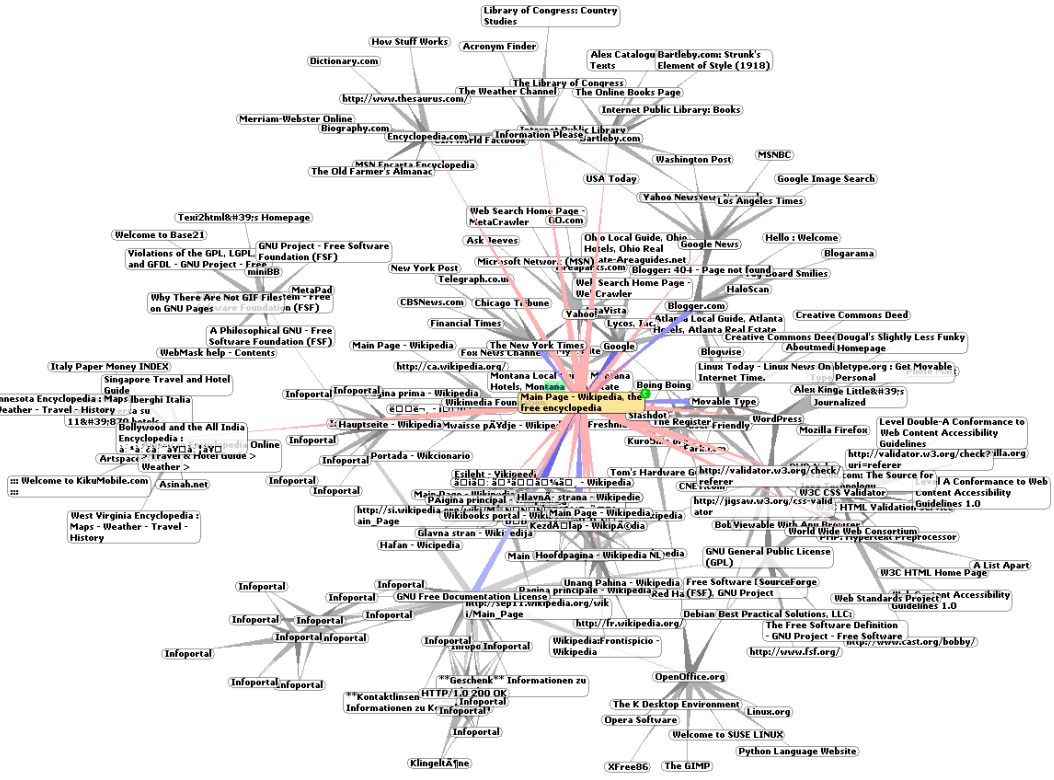
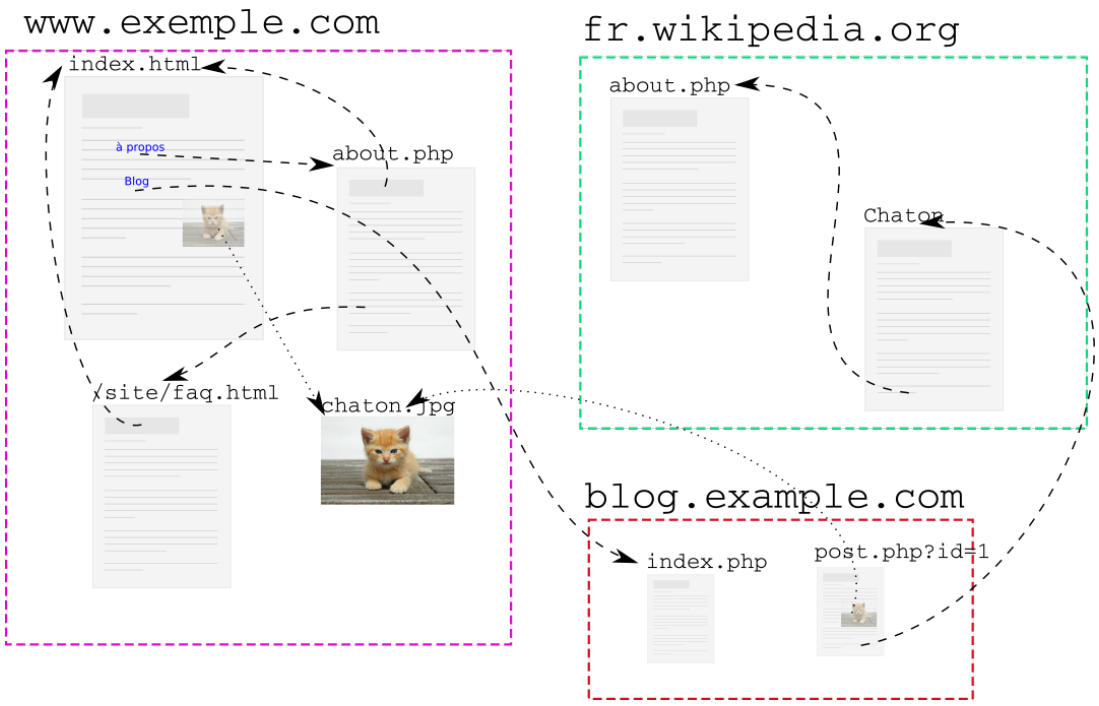


Figure 4 – « WorldWideWeb Around Wikipedia – Wikipedia as part of the world wide web »

Source : Chris 73 / Wikimedia Commons

Le *Web* signifie littéralement une toile (d'araignée), en anglais.
Chaque lien pointant d'un site à un autre dessigne une toile mondiale (*world wide web* : `www`).
Ce diagramme illustre le graphe de sites accessibles depus une page de Wikipedia.

5.1.2 Graphe de ressources liées décentralisé



Le Web est par essence **décentralisé**.
Les liens sont entre différents documents, du même site ou de sites différents.
Certains liens sont navigables, d'autres correspondent à l'inclusion de ressources externes (images).
Note : les liens sont uni-directionnels, du point de vue des serveurs : les documents ne « savent » pas qui leur pointe dessus. Aucune coordination : pas de permission à demander avant de tisser des liens, mais pas de garantie de disponibilité des ressources distantes.

Ressources

- Documents (statiques)
- Consultation dynamique :
  - ressources issues d'applications (dynamiques)
  - représentation de ces ressources dans les navigateurs Web
- Éléments « virtuels » décrivants des « faits » (Web sémantique/des données)

Pour être précis, on parle de **ressources**, sur le Web. Cette notion est très générique. Elle englobe des documents ou des pages de site, sans exclure des modèles plus avancés d'un Web des données par exemple.
Pour une discussion du terme, d'un point de vue historique, voir par exemple A Short History of « Resource » in web architecture rédigé en 2009 par Tim Berners-Lee.

Ressources liées

- Identification des ressources
- Localisation des ressources
- Agréger des ressources et leurs donner des propriétés pour créer de la connaissance

L'intelligence du Web réside justement dans les liens établis entre les ressources.
Cela permet de présenter des documents riches, mais aussi de naviguer vers d'autres sites.

Décentralisé

— Lier entre-elles des ressources localisées physiquement n'importe où sur Internet
— Confiance, provenance ?

C'est le travail des outils clients, comme le navigateur Web, de construire de l'intelligence en rendant accessible du contenu distribué, mis en ligne de façon pas nécessairement coordonnée.
Rien ne garantit alors que le contenu est fiable, pérenne dans le temps, ou qu'on puisse y faire confiance.
Contrairement au cas d'applications développées dans des environnements contraints et monolithiques, les applications déployées sur le Web doivent ainsi prendre en compte ces contraintes (on y reviendra).

5.1.3 Fonctionnement simplifié de l'accès aux pages d'une application Web

Cette section décrit les grands principes architecturaux qui sous-tendent le fait d'utiliser le Web et HTTP pour faire fonctionner des applications.

Structure générale

1. Client + Serveur

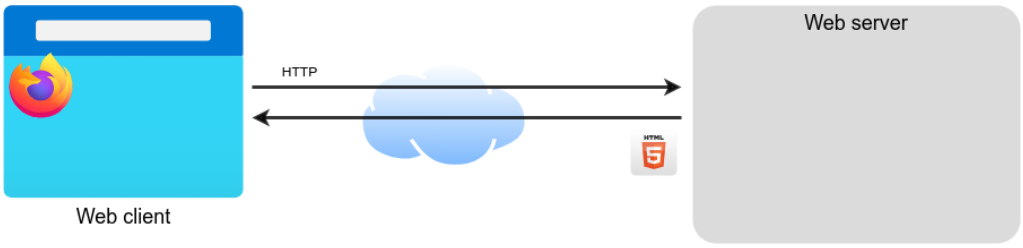
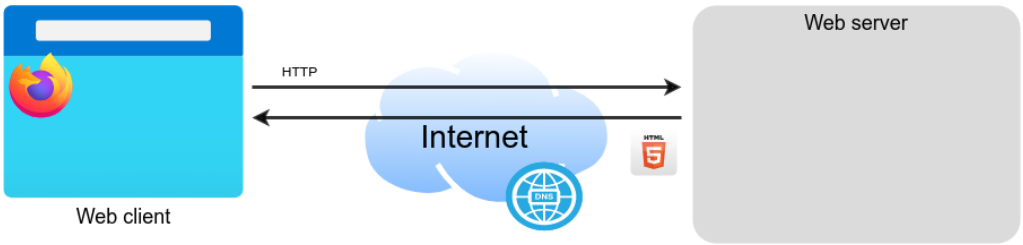


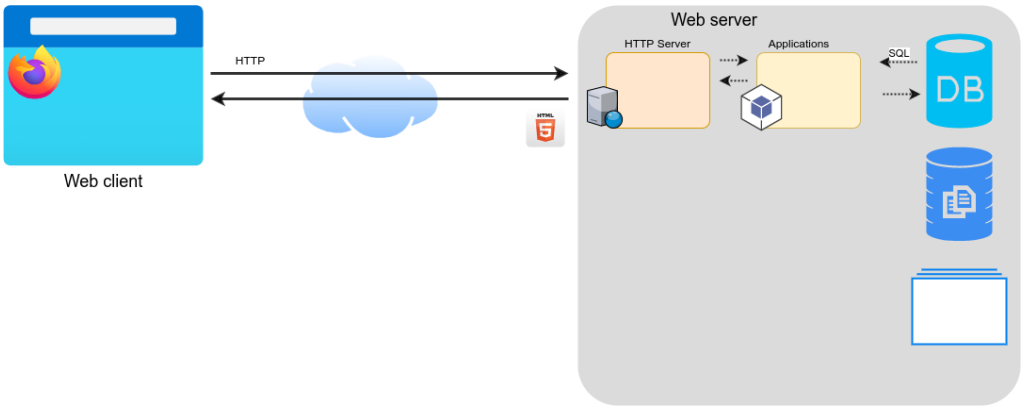
Illustration du navigateur (*Web browser*) Firefox, comme client Web, qui communique à travers l'Internet avec un serveur Web.
Il transmet des requêtes grâce au protocole HTTP, et récupère des documents HTML dans le contenu des réponses HTTP.
Regardons maintenant ce dont on a besoin au niveau d'Internet.

2. Internet sous-jacent



Le navigateur s'appuie notamment sur le système des noms de domaines (DNS) pour effectuer les requêtes auprès du serveur.
Ainsi, n'importe quel possesseur d'un nom de domaine peut mettre en ligne un serveur Web, qui sera immédiatement disponible pour que des clients s'y connectent, sans autre forme d'autorisation.
Aucune autorité centrale du Web n'entre en jeu pour délivrer une quelconque autorisation.
Attention cependant aux contraintes légagles en vigueur (LCEN, en France, par exemple).
Examinons maintenant ce qu'on trouve du côté du serveur

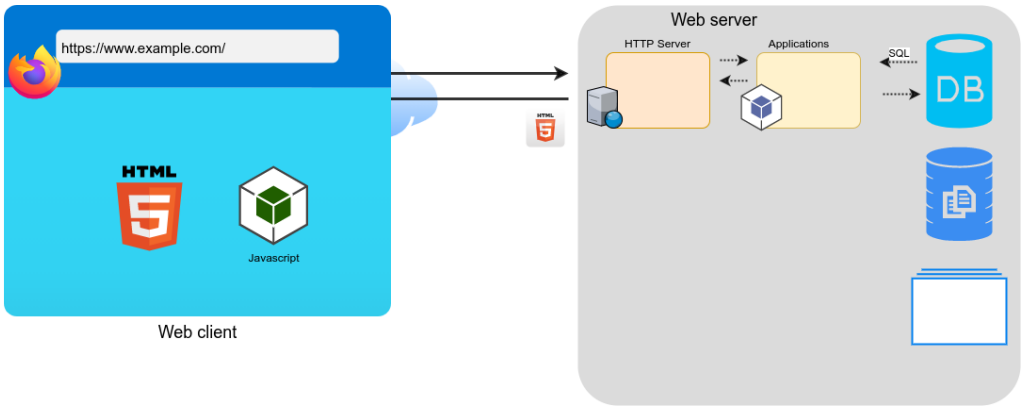
3. Côté serveur



Dans un premier temps, on distingue 3 ensembles d'applications mises en œuvre côté serveur :
- le serveur HTTP « frontal » qui écoute les requêtes transmises par les clients. Il en gère en direct un certain nombre (contenu « statique »)
- les applications, à qui le serveur HTTP passe la main, quand il s'agit de contenu dynamique, qui nécessite l'exécution d'une application pour obtenir la réponse attendue.
- différents composants additionnels, utilisés par les applications. En général, on peut trouver :
  - une base de données (*DB*), souvent relationnelle, dans laquelle l'application peut effectuer des requêtes avec le langage SQL
  - du stockage de fichiers, pour permettre la consultation de documents transmis par l'application, mais aussi pour gérer des stockages internes au serveur Web (sessions, etc.)
  - tout autre élément d'interfaçage avec le Système d'Informations (SI) d'entreprise

Revenons maintenant sur ce qu'on trouve dans le navigateur.

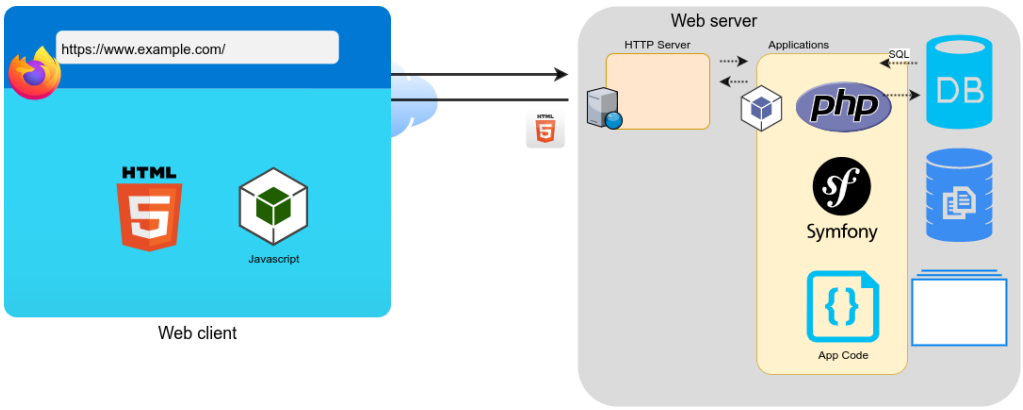
4. Dans le navigateur



Dans le navigateur, on distingue pour l'instant deux grands composants :
- le moteur de rendu HTML, qui va afficher les documents HTML transmis par le serveur sous forme consultable par l'utilisateur.
- une « machine virtuelle » Javascript capable de faire tourner des programmes dans le navigateur, pour disposer de plus d'intelligence du côté client.

Enfin, examinons le cœur de l'intelligence applicative, côté serveur

5. Code des applications



Finalement, concluons ce panorama rapide en regardant ce qui peut constituer une application qui fonctionne derrière le serveur HTTP frontal, côté serveur :

- le code spécifique de l'application est installé, et est exécuté lorsque certaines portions du site Web seront consultées, sur demande du frontal HTTP. Ce code est typiquement, dans ce cours, du code PHP, interprété
- on doit donc disposer d'un interpréteur du langage PHP
- mais il faut aussi disposer de toutes les bibliothèques (en général via un *framework*) nécessaires à l'exécution du code, comme par exemple avec Symfony, dans ce cours.

Le code s'appuiera sur le *framework* pour générer des pages HTML, en réponse aux requêtes, pour les rendre au frontal HTTP, afin que celui-ci les renvoie vers le client.

5.1.4 Clients et serveurs HTTP

Le cœur de l'architecture technique du Web est le protocole HTTP, qui repose sur un modèle client-serveur.

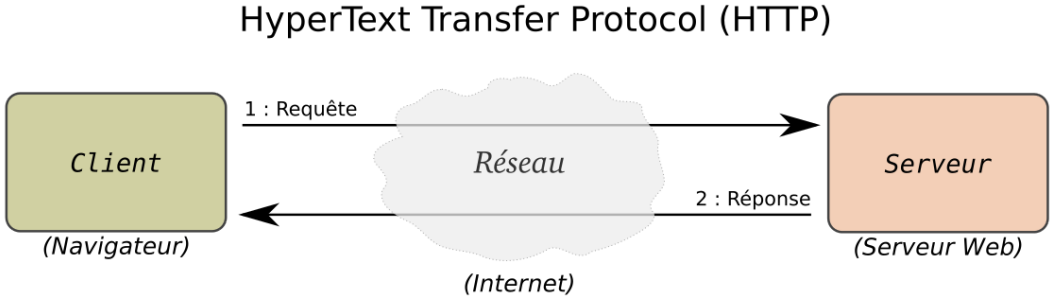
Architecture Client-Serveur

Protocole très simple :

1. requête
2. réponse

Cette architecture est très classique et n'est pas née avec le Web.
Ici, on doit plus précisément parler d'un serveur et de multiples clients qui s'y connectent. Sur le Web, et dans HTTP, les serveurs sont conçus pour communiquer systématiquement avec de multiples clients.
On rappelle qu'il est très fréquent qu'un même client parle avec différents serveurs pour des tâches très courantes, comme la consultation de pages Web dans un navigateur, qui nécessite de charger des contenus sur différents serveurs HTTP indépendants.

Client et serveur Web



En fait : 1 client - n serveurs (modèle distribué)

Le protocole de communication entre clients et serveurs est HTTP (*HyperText Transfer Protocol*).
Le client peut être tout type de programme (navigateur, robot, etc.).
Précision terminologique : dans les spécifications on parle de :
- Client appelé *user agent* dans les spécs
- Serveur appelé *origin server* dans les spécs

La notion d'*origin server* (serveur d'origine), permet de distinguer le serveur d'un éventuel *proxy* (serveurs mandataire) présent entre ce serveur et le client.
Les détails d'HTTP concernant les *proxies* ne seront pas abordés dans ce cours.
On étudiera le protocole HTTP plus en détails dans la prochaine séquence de cours magistral.

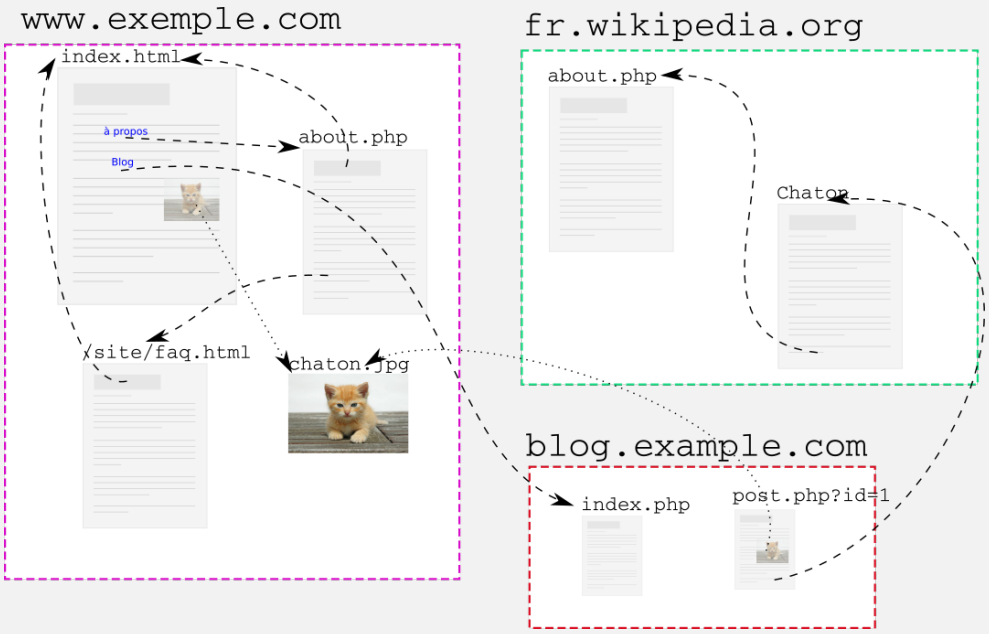
Dialogue entre client et serveur
- Communication en 3 étapes simples :
  1. Le **client** (navigateur) fait une **requête** d'accès à une **ressource** auprès d'un serveur Web selon le protocole **HTTP**
  2. Le **serveur** vérifie la demande, les autorisations et transmet éventuellement l'information demandée
  3. Le client interprète la **réponse** reçue (et l'affiche)
- On recommence pour des ressource complémentaires (images, …).

Concepts de base d'HTTP
- **Interaction** avec des ressources hypertexte :
  - Quoi : **Action** : verbe (CRUD)
  - *Qui* : **Ressources** identifiées par des **URI** (données d'une application, …)
  - *Où* : **URL** pour **localiser** les représentations informatiques de ces ressources (documents hypertextes, images, etc.) sur des serveurs Internet
  - *Comment* : Requêtes protocole de communication **HTTP**

L'acronyme CRUD correspond aux opérations classiques qu'on peut effectuer sur des données : *Create*, *Retrieve*, *Update*, *Delete*.
On parle en fait plutôt d'URI que d'URL dans les spécifications, même si pour l'instant cette subtilité a peu d'importance.

Construction de la toile (*Web*)
- **Graphe** de ressources hypertexte/hypermedia **décentralisé**
- Les ressources référencent d'autres ressources (attribut `href` dans HTML)
- Les liens sont unidirectionnels
- Aucune garantie de disponibilité, cohérence
- Parcourir la toile :
  1. trouver les liens,
  2. accéder aux ressources liées

3. enrichir le modèle applicatif
4. *recommencer* …



Hyper-reference, *hyper-link*
Historiquement, **HTML** comme langage de mise en forme pour la visualisation des ressources de type « document hypertexte »
Le navigateur permet de suivre les liens dans le graphe : le graphe n'existe pas (à part dans un cache) tant qu'on n'a pas essayer de naviguer dessus.
Seul point « stable » : le système DNS, comme pour tout Internet.

- Composants de ce graphe :
  - 3 sites/serveurs Web : `www.example.com`, `blog.example.com`, `fr.wikipedia.org`
    - (peut-être sur 2 serveurs (2 adresses IP) : `example.com` et `fr.wikipedia.org` ?)
  - Documents/ressources :
    - `http://www.example.com/index.html` (HTML)
    - `http://www.example.com/site/faq.html` (HTML)
    - `http://www.example.com/about.php` (HTML)
    - `http://www.example.com/chaton.jpg` (image)
    - `http://blog.example.com/index.php` (HTML)
    - `http://blog.example.com/post.php?id=1` (HTML)
    - `http://fr.wikipedia.org/about.php` (HTML)
    - `http://fr.wikipedia.org/Chaton` (HTML)
  - Liens :
    - Inclusions images
      - Même site
      - Autre site
    - Hyper-liens navigables
      - Document du même site
      - Document sur un autre site
      - Fragment à l'intérieur d'un document

Localisation des ressources

- Objectif : nommer, localiser et accéder à l'information
- Solution : **URL (*Uniform Resource Locator*)** : identification universelle de ressource composée de 3 parties :
  1. le **protocole** (*comment*)
  2. l'identification du serveur Web, le **nom DNS** (*où*)

3. l'**emplacement de la ressource** sur le serveur (*quoi*)

Plus tard, on raffine avec le concept d'URI : cf. RFC 3986 - Uniform Resource Identifier (URI) : Generic Syntax et standards associées.

5.1.5 Deploiement sur le Web

Allons-y, lançons-nous : pouvons-nous déployer nos sites Web ?

Interopérabilité ?

- Qui peut créer des clients Web ?
- Qui peut opérer des serveurs ?

Historiquement, le Web est conçu comme un bien commun, où tout le monde peut opérer des clients et serveurs, pour autant qu'on a accès à Internet, et qu'on respecte les standards élaborés par la communauté.
C'est encore vrai dans une certaine mesure, mais pour offrir des garanties de disponibilité, de sécurité, de performances, ça devient une affaire de professionnels, et certains prestataires deviennent incontournables.
Et pour que les différents clients et serveurs puissent se comprendre il faut assurer le respect des standards. Malheureusement, ce n'est pas toujours le cas, et les opérateurs ayant une part dominante sur le marché ne sont pas toujours des bons élèves.

Déploiement DIY

- Raspberry Pi
- Fibre
- Debian GNU/Linux
- Apache / Nginx
- …

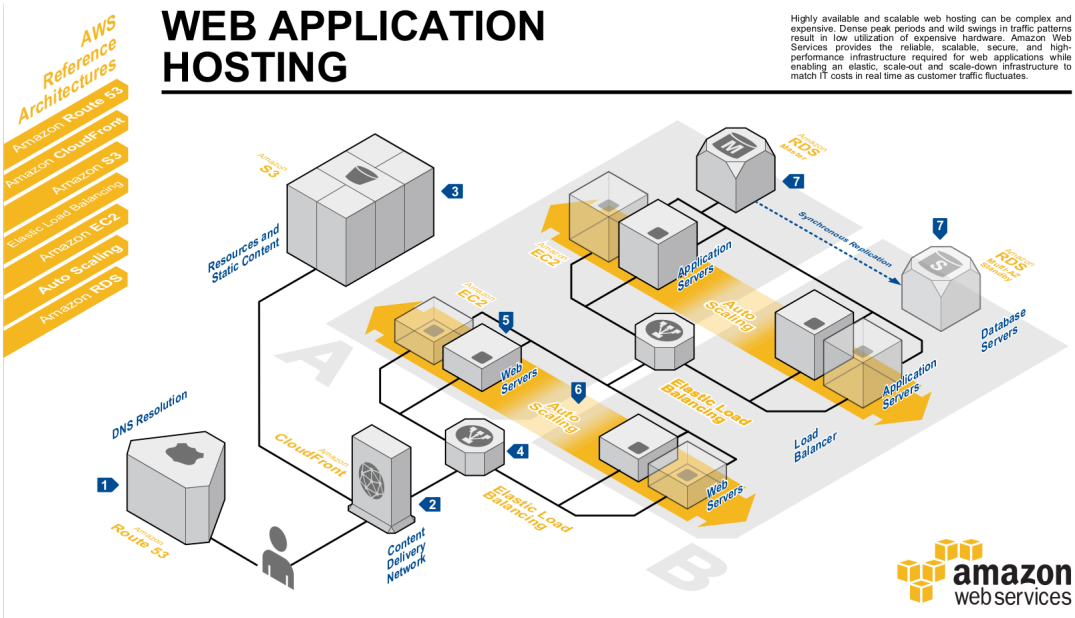


Figure 5 – Diagramme d'architecture d'*Amazon Web Services*

Déploiement Cloud

À titre d'information, l'hébergement et la conception des applications Web passe aujourd'hui par des plate-formes dédiées, sur le Cloud, comme celle d'Amazon illustrée ci-dessus.
Cependant, nous n'étudierons pas les propriétés de telles plate-formes dans le cadre de ce cours d'introduction. Nous observerons des plate-formes plutôt plus traditionnelles comme l'hébergement PHP sur un serveur Web « classique ».

5.2 Architecture 3 couches

Cette section présente les principes d'architecture des applications Web classiques, et notamment l'architecture dite 3 tiers.
Les applications Web sont ubiquitaires aujourd'hui, mais on n'a pas forcément toujours des idées très claires sur ce qui permet de faire fonctionner ces applications.
Cette séquence va nous permettre d'examiner les mécanismes et protocoles nécessaires au donctionnement d'une application Web.

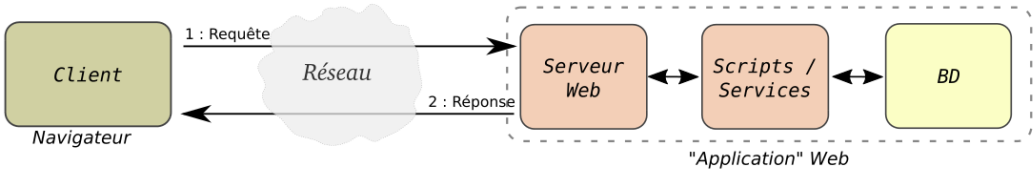
5.2.1 Architecture ?

- Structure générale d'un système informatique
  - matériel
  - **logiciel**
  - humain / responsabilités
- Conception
  - Normes ?
  - Standards ?
  - Bonnes pratiques ?

L'intitulé du cours mentionne ce mot « architecture ». Il est donc important de se pencher sur sa signification.
En informatique, comme dans d'autres domaines (bâtiment, …) on peut s'intéresser à des principes d'architecture qui permettent d'apprendre à construire des applications, de la « bonne façon ».
Nous n'étudierons pas en détail toutes les acceptions du terme, mais on verra par exemple, dans la suite du cours, un ensemble de bonnes pratiques consistant à s'appuyer sur un *framework* pour bénéficier d'un ensemble de bonnes pratiques, plutôt que d'avoir à réinventer la roue.

5.2.2 À l'origine, applications BD + Web

Les applications produisent des pages Web **dynamiques**, à partir de données issues de bases de données.



On décore le SGBD avec des pages qui génèrent des requêtes.

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

Succès historique des applications BD + Web Des **applications** sont réalisables facilement, dès que des programmes génèrent des pages HTML en fonction des requêtes HTTP du client

Pour « M / Mme Michu », quand même plus convivial…

… que SQL !

Avantages

- Base de données gère l'intégrité des données (transactions, accès multiples, intégrité référentielle, …)
- Tient la charge (tant qu'on arrive à dupliquer l'exécution des scripts)
- Large gamme de choix du langage de programmation

On ne raffine pas trop le modèle de couches avec l'adaptateur de données qui est parfois introduit, pour ne pas trop complexifier
Les programmes sont appelés *scripts* sur le diagramme, 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, comme CGI.
Un souci 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.

5.2.3 Architecture 3 *tiers*

Architecture *logicielle* en couches (*tiers*) :

- couche de **présentation** ;
- couche de **traitement** ;
- couche d'**accès aux données**.

Simplification de l'approche générale d'une architecture à plusieurs niveaux (*multi tier*)

L'objectif de cette décomposition en couches est de mieux comprendre la logique des interactions entre différents composants logiciels mis en œuvre pour faire fonctionner une application. Il s'agit ici d'une architecture logicielle/système, indépendant des fonctionnalités fournies par telle ou telle application (on parlerait d'architecture fonctionnelle).
On parle aussi d'**architecture à trois niveaux** ou **architecture à trois couches**.
Le mot *tiers* est une traduction approximative de l'anglais *tier* signifiant étage ou niveau.

Variante ligne de commande Application Symfony `Todo` (cf. TP), en ligne de commande :

présentation affichage sortie standard

traitement `App\Command\ListTodosCommand`

accès aux données Doctrine + SQLite (SQL)

Une application démarre pour une exécution pour un utilisateur unique, dans le contexte d'un shell, en ligne de commande dans un terminal.
L'application est un processus de l'interpréteur PHP qui démarre le programme `bin/console` du *framework* Symfony.
L'application est « monolithique », fonctionnant dans la mémoire d'un seul système d'exploitation, en local, en un seul processus.
Les 3 couches sont des couches logiques aidant à comprendre l'architecture fonctionnelle de l'application.

Variante « classique » sur le Web Classique : éléments logiciels principalement côté serveur (y compris présentation)

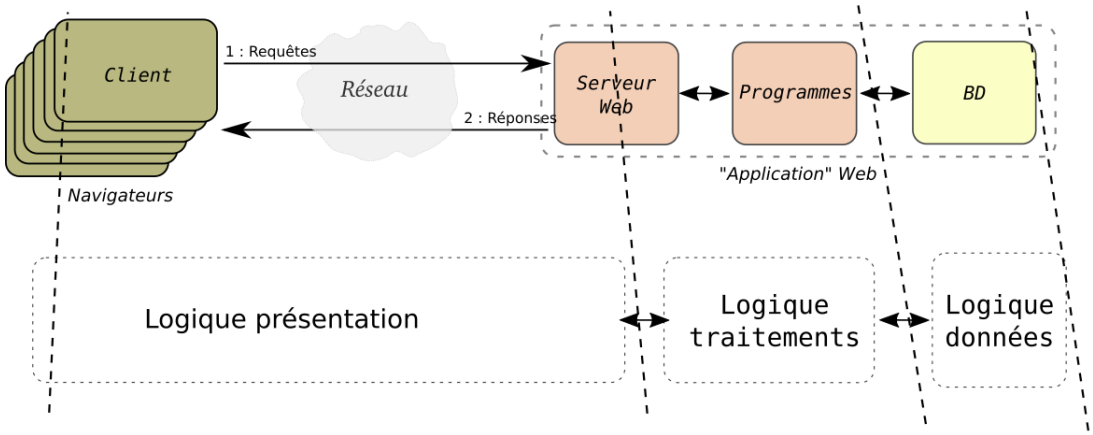


Figure 6 – Architecture 3 couches

L'application fonctionne sur un serveur accessible depuis le réseau via le protocole HTTP.
Un navigateur Web permet de s'y connecter et de consulter des pages Web présentant le résultat de l'exécution d'un programme fonctionnant dans la mémoire du serveur.
Attention, ici, plusieurs clients Web peuvent accéder simultanément à la même application fonctionnant sur le serveur Web.
La qualification de « classique » correspond à un positionnement des éléments logiciels principalement côté serveur (y compris pour une bonne part de la couche de présentation)

3 couches des applications Web

- **Présentation** : *pages HTML* :
  - *serveur* les construit (classique)
  - *client* (navigateur) les charge et affiche
- **Traitements** : programmes :
  - serveur Web (dialogue HTTP, invocation des applications)
  - serveur d'applications (exécute des programmes PHP, par ex.)
- **Accès aux données** : SGBD

Les multiples clients Web (de chacun des utilisateurs) communiquent via HTTP avec un serveur Web unique.
Le navigateur Web qui fait le rendu des pages HTML participe à l'application (présentation).
Les clients Web ne sont pas seulement les navigateurs des utilisateurs, mais peuvent aussi être des programmes se connectant à des APIs via HTTP.
Dans la variante classique, les traitements se font selon les programmes écrits par les développeurs (par exemple en PHP) fonctionnent sur / derrière ce serveur Web. Dans d'autres architectures plus récentes, les traitements peuvent s'exécuter de façon très importante sur le client.
Ils peuvent s'exécuter pour effectuer des traitements grâce aux services fournis par des serveurs d'applications sous-jacents (journeaux, envoi de mails, etc.).
Dans cette déclinaison en 3 couches, chaque couche est plus ou moins indépendante des autres. On peut par exemple imaginer que d'autres clients que les clients Web peuvent accéder aux mêmes serveurs d'applications pour interagir avec les programmes (via des protocoles autres que HTTP).
De même, les données stockées dans la base de données peuvent être manipulées par d'autres applications que ces applications Web, ou directement par des utilisateurs via les interfaces du SGBD (cf. CSC3601).

5.3 Développer aujourd'hui une appli « classique » ?

Dans ce cours, on concevra l'application de façon traditionnelle, en 3 couches.
On utilisera le cadriciel (*framework*) **Symfony** pour son approche **orienté objet**.
Dans le cours, on va se concentrer principalement sur l'apprentissage de la conception des couches :

1. présentation (*frontend*)
2. traitements et accès aux données (*backend*)

L'accès au données est considéré comme déjà acquis (SGBDR, SQL).

5.3.1 Architectures modernes, pédagogie…

On pourrait apprendre d'autres architectures plus modernes laissant une plus grande place aux composants sur le client (Javascript, serverless, etc.). Apprentissages complexes vue le temps imparti.
Objectif pédagogie ! Faciliter les apprentissages, donc architecture plus classique facilite la tâche.

5.4 Web comme plate-forme

Les technos du Web ne concernent plus seulement les « sites » vus dans un navigateur sur un PC
Importance du côté client :

- **Navigateur**
  - Moteur de rendu HTML

    - Machine virtuelle Javascript
- **Applications natives** programmées HTML5 + CSS + **Javascript** sur mobile
- Client HTTP + (micro) services REST

Native apps vs *Web apps*

L'architecture d'applications présentée dans le cours est très classique et repose sur un modèle un peu ancien, donc éprouvé.
De nouvelles architectures d'applications basées sur les technologies du Web ont vu le jour, mais qui reprennent des concepts similaire. Nous n'aurons pas le temps de les étudier en détails, mais elles reposent sur des variantes des concepts de base que nous aurons étudiés.
Ce que vous aurez appris vous servira (si vous continuez dans le développement d'applications), même avec d'autres architectures apparues plus récemment.

Take away

- Structure de la toile
    - Ressources
    - Ressources liées
    - Décentralisé
- Protocole client-serveur, HTTP, URL
- Applications générant des pages Web
- Modèle de couches (3 et +)
    - présentation
    - traitement
    - accès aux données

Aller plus loin

Le lecteur intéressé pourra consulter le document Web Architecture from 50,000 feet rédigé par Tim Berners Lee il y a 20 ans. Il date un peu pour certains aspects, mais l'essentiel est toujours applicable, étonamment.

Postface

Crédits illustrations et vidéos

- Illustration plate-forme Web Amazon AWS : `http://media.amazonwebservices.com/architecturecenter/AWS_ac_ra_web_01.pdf`
- Diagramme « Perdu sur le Web » : `#Geekscottes` par *Nojhan* `https://botsin.space/@geekscottes/101748180337263915`
- Illustration « chaton » : memegenerator.net
- « WorldWideWeb Around Wikipedia – Wikipedia as part of the world wide web » Chris 73 / Wikimedia Commons GFDL 1.3 or CC BY-SA 3.0

Figures interactives On utilise un export HTML d'une illustration réalisés avec `https://www.diagrams.net/` (moteur OpenSource de `draw.io`).

Annexes

Structure des URLs

— URL type :

— Composantes :

1. protocole : en majorité `http` ou `https`
2. adresse / nom du serveur : `www.monsite.fr`
3. chemin d'accès à la ressource / document : `/projet/doc.html` (ne garantit pas que le résultat est un document HTML)

URLs plus détaillées Exemple d'URL plus complexe :

1. protocole : `http` ou `https` (mais aussi `ftp`, `file`, `mailto`, `gopher`, `news`,…)
2. autorité : nom du serveur : `www.monsite.fr` + port : `8000`
3. chemin d'accès à la ressource : `/projet/doc`
4. requête : deux paramètres : `id` et `f`
5. fragment : `num42` à l'intérieur du document

Il existe aussi un sur-ensemble des URLs, les URI (*Uniform Resource Identifier*). On se préoccupe plus d'identifier une ressource que de savoir comment y accéder. En pratique, cette subtilité ne change pas grand chose dans le cadre de ce cours.
Pour plus de détails, voir la spécification générale des URIs (RFC 3986).

URLs absolues ou relatives

— Les URL permettent d'établir des **liens** entre documents
  — sur le même serveur
  — entre différents serveurs
— Liens sur le même serveur :
  — chemin absolu / chemin relatif
  — analogie avec chemin de fichier Unix : `.`, `..`, `/`
  — lien intra-document : fragments/ancres : `doc.html#conclusion`
  — convention : lien dans l'espace d'un utilisateur : `~taconet/menu.html`

Exemples

Document source	Ressource liée	Emplacement réel
`http://w.e.c/index.html`	`about.php`	`http://w.e.c/about.php`
`http://w.e.c/about.php`	`/`	`http://w.e.c/` (contenu de `index.html`)
`http://w.e.c/site/faq.html`	`../index.html`	`http://w.e.c/index.html`
`http://w.e.c/index.html`	`./chaton.jpg`	`http://w.e.c/chaton.jpg`
`http://w.e.c/index.html`	`http://b.e.c/`	`http://b.e.c/` (contenu de `index.php`)
`http://w.e.c/about.php`	`/site/faq.html`	`http://w.e.c/site/faq.html`
`http://b.e.c/post.php?id=1`	`http://w.e.c/chaton.jpg`	`http://w.e.c/chaton.jpg`
`http://b.e.c/post.php?id=1`	`http://f.w.o/Chaton`	`http://f.w.o/Chaton`
`http://f.w.o/Chaton`	`/about.php`	`http://f.w.o/about.php`
`http://f.w.o/Chaton`	`/about.php#terms`	`http://f.w.o/about.php` (`<div id=« terms »>`)

6 Séq. 2 (2/3) : Histoire de la toile

Objectifs de cette séquence

Cette séquence de cours abordera les éléments suivants :

1. Une présentation rapide de l'histoire du Web
2. L'évocation de quelques enjeux socio-technologiques relatifs au Web

6.1 Avènement du Web

Cette section présente l'historique du Web, sans entrer dans les détails techniques.

6.1.1 Historique

> 30 ans

Avant le Web

— Minitel (1980-2012)

Figure 7 – Minitel 2

— Systèmes *hypertextes* locaux
— FTP, Usenet, Gopher

Pour voir des minitels, cf. expo du « Musée de l'INT »
Gopher voit le jour à peu près au même moment que WWW, et fournit aussi un système hypertexte.
Il semble avoir subi la concurrence du Web, du fait d'une tentative de gestion de la propriété intellectuelle associée par l'université d'origine, mais aussi plus probablement du fait de l'ajout du support des images dans WWW, alors que Gopher était essentiellement textuel.

Hypertexte

— *Memex* de Vannevar Bush, dans *As We May Think* (*Atlantic Monthly*, 1945)
— *Xanadu* de Ted Nelson : `https://fr.wikipedia.org/wiki/Projet_Xanadu` (1965-…)
— *HyperCard* de Bill Atkinson (Apple, 1987)
— *World Wide Web* de Tim Berners-Lee (CERN, 1989)

Plus de détails dans `https://fr.wikipedia.org/wiki/Hypertexte`

Naissance du World Wide Web

— Le Web est né au CERN en 1989-1990.
— Tim Berners-Lee a défini une architecture pour accéder à des documents liés entre eux et situés sur des serveurs reliés par Internet (*Web* = toile d'araignée)

Figure 8 – Tim Berners-Lee

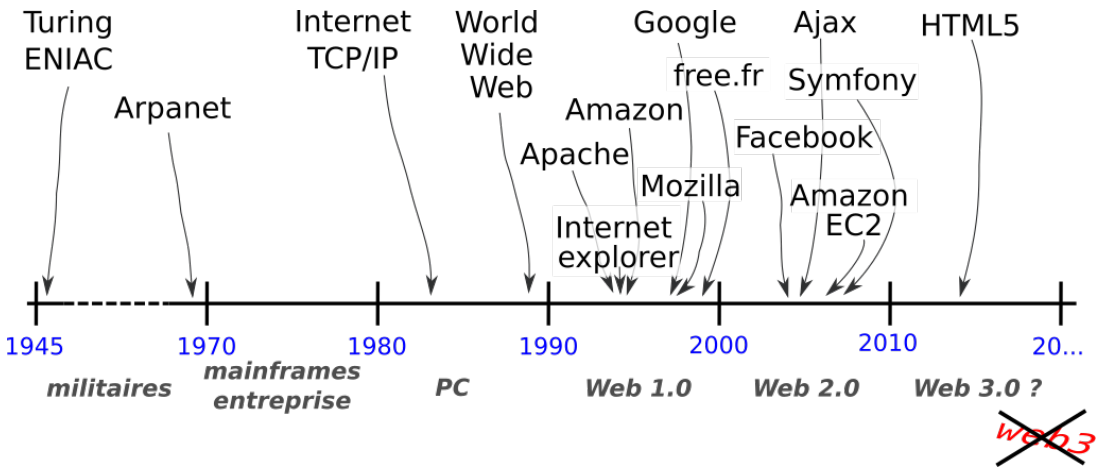
— le W3C (*World Wide Web Consortium*) mis en place rapidement (1994) pour définir des standards (ouverts).

Objectif : répondre à leur besoin d'échanges de documents (rapports, croquis, photos…) entre des équipes internationales.
Note : l'IMT est membre du W3C.
Important : ouverture des protocoles, standards ouverts.

Naissance du World Wide Web Vidéo Naissance du World Wide Web (extraite de l'expo Web du *Computer History Museum*) `https://www.youtube.com/watch?v=_mNOXDbXr9c`

Premier site Il existe toujours, pour les curieux
`http://info.cern.ch/hypertext/WWW/FAQ/Bootstrap.html`

6.1.2 *Timeline*



— Turing, ENIAC : 1946
— Arpanet : 1969
— TCP/IP : 1983
— WWW : 1989
— Apache : 1995
— Amazon : 1995
— Internet explorer : 1995
— Mozilla : 1998
— Google : 1998
— free.fr : 1999
— Facebook : 2004
— Ajax : 2005
— Amazon EC2 : 2006
— Symfony : 2007
— HTML5 : 2014

Voir aussi `http://webdirections.org/history/#0`

6.2 Grandes étapes de l'évolution du Web

6.2.1 1. Naissance du Web

Web 1.0 (début des années 1990)

— Accès à des documents structurés via des liens hypertextes
— Protocoles et langages simples
— Technologies de base HTML, HTTP, MIME, formats GIF…

6.2.2 2. Ouverture, homogénéisation et programmation

(fin des années 1990)

— Interactions avec les applications et programmation Web
— Langages plus riches, manipulation d'objets, développement des styles
— Evolution des technologies : XML, CSS, DOM, Server Pages, JavaScript …
— Standardisation difficile (guerre des navigateurs)

6.2.3 3. Evolution des usages et de l'interface utilisateur

Web 2.0 (depuis 2005)

— Partage d'informations, édition collaborative, sites communautaires
— Réseaux sociaux, mondes virtuels
— Technologie AJAX, HTML 5
— Intégration de flux RSS, de vidéos, de podcasts
— Personnalisation des accès
— *User-Generated* Content (UGC)

6.2.4 *Web 3.0* vs *web3* : le bin's

— **Web 3.0** : « Web Sémantique » *aka* « Web des données »
plaît plus à (certains) « vieux cons des Internets »™ (votre serviteur compris)
— **web3** : un machin à base de chaînes de blocs, et de spéculation sur la rareté artificielle de ressources ~~numériques~~ digitales… sérieux ??? (**hors sujet dans ce cours**)

Cf. `https://fr.wikipedia.org/wiki/Web3`

6.3 Enjeux

6.3.1 Ouverture

- Ouverture de l'Internet
- Standards ouverts
  - Guerre des navigateurs
  - Poids de Google / Chrome
- Logiciel libre vs *SaaS*
- Décentralisation

6.3.2 Bien commun

- Wikipedia
- OpenStreetMap
- Données ouvertes (*open data*, *open gov*, …)
- *Wayback machine* de l'*Internet Archive*

6.3.3 Menaces, problèmes

- Asymétrie réseaux
- Centralisation
- Surveillance
- *Digital labor*
- Revenus (gratuité, publicité)
- Censure
- Algorithmes (recommendation)
- Accès (haut débit)
- Neutralité du net
- …

La suite sur : `https://www.eff.org/work`

Ce cours est essentiellement technique et ne vise pas à aborder toutes ces thématiques.
On doit seulement observer que la technique n'est pas neutre, et que ces enjeux sont de plus en plus cruciaux dans une société où le numérique occupe une part de plus en plus forte.
Les ingénieurs doivent se préoccuper des conséquences de leurs choix technologiques, autant que possible…

Aller plus loin

Le lecteur intéressé pourra consulter le document Web Architecture from 50,000 feet rédigé par Tim Berners Lee il y a 20 ans. Il date un peu pour certains aspects, mais l'essentiel est toujours applicable, étonamment.

Postface

Crédits illustrations et vidéos

- Vidéo « Birth of the World Wide Web » © Computer History Museum (used by courtesy of Computer History Museum).
- Diagramme « Perdu sur le Web » : `#Geekscottes` par *Nojhan* `https://botsin.space/@geekscottes/101748180337263915`

— Illustration « chaton » : memegenerator.net
— « WorldWideWeb Around Wikipedia – Wikipedia as part of the world wide web » Chris 73 / Wikimedia Commons GFDL 1.3 or CC BY-SA 3.0

7 Séq. 3 : Protocole HTTP

Objectifs de cette séquence

Cette séquence de cours présente le **protocole *HyperText Transfer Protocol*** (HTTP). On étudie les principaux éléments du protocole HTTP permettant aux clients Web et aux serveurs d'interagir sur le Web.

7.1 Protocole HTTP

L'objectif de cette section est de comprendre le fonctionnement du protocole de communication entre clients et serveurs Web, HTTP (*HyperText Transfer Protocol*).
On a déja abordé les grandes lignes du fonctionnement client-serveur du protocole dans la séquence de cours précédente, mais on va approfondir ici la structure des échanges et le rôle des différents éléments.

7.1.1 Principes requêtes et réponses

Architecture Client-Serveur HTTP (*HyperText Transfer Protocol*)

— Multiples clients simultanés
— Serveur sans « connaissance » particulière des clients
— Réseau internet TCP/IP

Le client construit la requête

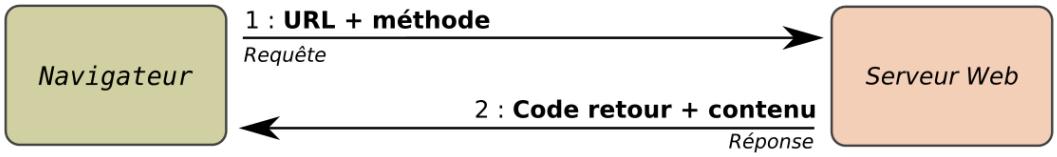
1. Le client (navigateur) interprète l'**URL**
2. demande au service de nom **une adresse IP** pour le nom du site (`www.monsite.fr`)
3. établit une **connexion TCP** avec cette adresse, sur le numéro de port : 80 pour HTTP (ou 443 pour HTTPS, …)
4. formule sa **requête** d'action (par ex. **méthode** `GET`) sur une **ressource** (`/projet/doc.html`), en écrivant un message :
   ex. : « `GET /projet/doc.html` »

Et, depuis HTTP 1.1 il indique au serveur qu'il souhaite parler au site en question avec l'en-tête `Host`, pour le cas où plusieurs sites Web seraient gérés par le même serveur qui écoute sur cette adresse IP.
Cf. Rappel : Services sur Internet (TCP/IP) en annexes pour plus de détails sur les aspects liés au réseau.

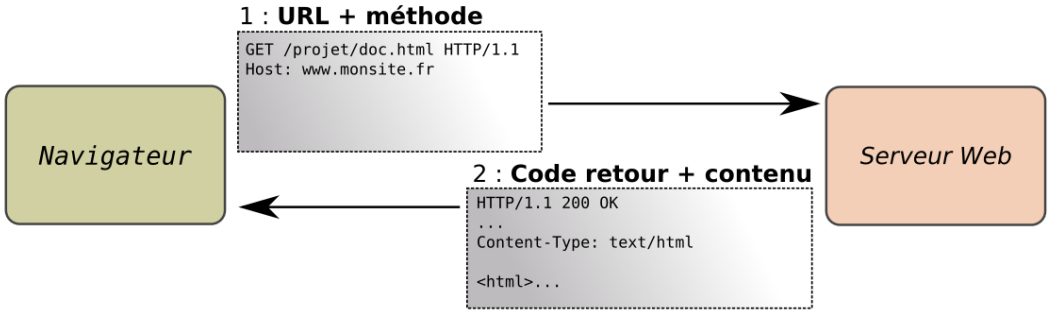
Le serveur traîte la requête

1. Analyse la requête (`GET /projet/doc.html`)
2. Vérification validité, autorisations d'accès…
3. Envoi d'une **réponse** au client avec :
   — un **code de statut**
   — le **contenu** (document ou résultat exécution)
   ou :
   — message d'erreur, ou une demande authentification, ou adresse de redirection
4. Fermeture de la connexion

Requête - Réponse



Contenu des messages



On verra plus loin le détail de ce qui circule dans les messages de requêtes et de réponses.

Production du contenu sur le serveur

- **chemin d'accès** à la ressource reçu : identifie la **ressource** demandée (« `/projet/doc.html` ») ;
- En fonction du **contexte** de la requête, pour la même URL, le serveur peut renvoyer différentes **représentations** de cette ressource, différents contenus/formats ;
- La façon dont le serveur fabrique le contenu renvoyé lui appartient : **le client ne peut rien supposer**, juste suivre les liens hypertextes.
- Arborescence de **chemins d'accès** aux ressources accessibles (dans les URL) **différente** d'une éventuelle **arborescence de stockage** effectif à l'intérieur du serveur.
  - Peut-être envoi d'un document stocké sur un système de fichier ?
  - Peut-être une application générant des documents dynamiquement ?
  - …

Choix du serveur arbitraire en fonction du contexte de la requête.

Le contexte pris en compte par le serveur dépend uniquement de la méthode HTTP, de l'URL accédée et des en-têtes présents dans la requête du client, mais pas des requêtes précédentes de ce même client (protocole sans état)
Le serveur Web va faire un certain nombre de choix qui dépendent de la façon dont il est configuré, avec des règles à appliquer relativement élaborées.
On ne couvre pas ces aspects de configuration dans le présent cours.

Transmettre des données au serveur Comment le client peut-il transmettre des données au serveur ?
Contexte explicite pour les applications dynamiques
L'URL peut contenir des informations, des variables ayant des valeurs.
Les **arguments** d'une méthode `GET` : clés - valeurs
Exemple :

```
http://www.monsite.fr/projet/service.php?id=3&nb=42
```

nom	valeur
id	3
nb	42

Le corps de la requête peut aussi servir transmettre des **données** : ex. méthode `POST` (voir ci-après)

Le séparateur `?` est un mot-clé particulier dans le format des URL. Le caractère `&` permet de séparer les couples clé-valeur des différents arguments transmis dans l'URL.
Le contenu de la requête POST permet de transmettre des données plus riches et plus volumineuses.

Protocole sans état (*stateless*)

- Chaque requête est effectuée de façon indépendante, et le serveur ne garde **pas la trace des requêtes précédentes** effectuées par le même client.
- De base, pas de gestion de session dans HTTP (mais des solutions existent, vues plus tard).
- Le contexte peut être transporté explicitement dans des arguments du `GET`… mais URL pas constantes (et passablement longues)

On reverra ce mécanisme de session ultérieurement dans le cours.

7.1.2 Démo

Installation *Apache* en local

7.1.3 Spécifications de base HTTP

Les spécifications de HTTP sont riches et très détaillées, et compréhensibles, et ont évolué pendant les 25 ans d'existence du protocole.
On en donne ici une présentation résumée et partielle, basée sur HTTP/1.1.

Format des messages HTTP

- Très peu de types de requêtes / méthodes (moins d'une dizaine)
- Tout en texte `ASCII` 7 bits (facile à débugger)
- Syntaxe très simple, avec 4 zones
  1. *ligne 1* : **requête** (ou **statut réponse**)
  2. *plusieurs lignes* : **en-têtes** (optionnels)
  3. *ligne vide* : **séparateur** (obligatoire)
  4. *reste du message* : **corps/contenu** (qui peut être vide)

Les en-têtes sont au format RFC 822, déjà popularisé pour l'email. Pour transporter autre-chose que de l'ASCII (binaire, accents, etc.) il faudra encoder / décoder le contenu.

Structure d'une requête client

- Canevas :

```
Méthode Document Version_HTTP
En-têtes
                                    (Ligne vide)
Contenu du Message          optionnel
  ...(saisie d'un formulaire, document à publier)...
```

La ligne vide est importante (fin des en-têtes)
- Exemple :

```
1  GET /hello.txt HTTP/1.1
2  User-Agent: curl/7.16.3 libcurl/7.16.3 OpenSSL/0.9.7l zlib/1.2.3
3  Host: www.example.com
4  Accept-Language: en, mi
5
6
```

Les lignes depuis la deuxième jusqu'au séparateur sont les en-têtes. Ici, les lignes 2 à 4, donc 3 en-têtes `User-Agent`, `Host` et `Accept-Language`.

Structure d'une réponse serveur

- Canevas :

```
Version_HTTP Code Signification
En-têtes
                                    (Ligne vide)
Contenu du Message
  ...(document HTML, image, ...)...
```

- Exemple :

```
1  HTTP/1.1 200 OK
2  Date: Mon, 27 Jul 2009 12:28:53 GMT
3  Server: Apache
4  Content-Length: 72
5  Content-Type: text/plain
6
7  Salut tout le monde. Mon contenu contient un retour charriot à la fin.
8
```

Le contenu du message (ce qui suit la ligne vide) est particulièrement important, mais devra être interprété par le client qui le reçoit en fonction des en-têtes. Comme il s'agit en général d'autre chose qu'un texte simple en ASCII, les en-têtes relatifs aux formats et méthodes d'encodage sont particulièrement importants.
Le corps/contenu du message, c'est à dire les données, est appelé *payload* en anglais.

7.1.4 Méthodes HTTP

Le client parle au serveur à travers une des méthodes correspondant à un ensemble d'actions qu'il souhaite effectuer sur les ressources du serveur.

Actions sur les ressources (CRUD)

- Le client demande dans la requête à effectuer des **actions/opérations** sur les ressources
- Il « invoque » différents types de **méthodes**

	Opération	**Méthode HTTP**	SQL
C	*Create* (création)	**PUT** / **POST**	INSERT
R	*Read* (accès)	**GET**	SELECT
U	*Update* (m-à-j)	**PUT** / **PATCH**	UPDATE
D	*Delete* (suppr.)	**DELETE**	DELETE

cf. RFC 9110 HTTP Semantics

Exemple de requête GET
— Demande du document `premier.html` par le navigateur (firefox)
Regardez les en-têtes dans les outils pour le développeur de vos navigateurs

Exemple de réponse à un GET
— Envoi du document HTML par le serveur Web Apache
La première ligne contient le code de statut :
`200 OK`

Méthode GET Récupérer la **représentation** d'une ressource
— Seule méthode à l'origine
⇒ usage très simple : `GET doc.html`
— Méthode la plus utilisée qui permet de :
— Récupérer le contenu d'un document
— Activer exécution de programme sur serveur
— Envoyer des données : `...programme?nom=paul`
— Avec GET, contenu du message dans la requête toujours vide

Autres méthodes
— **HEAD** : comme GET, mais récupérer seulement l'en-tête
— **POST** : envoi de données pour traitement côté serveur (ex : contenu d'un formulaire HTML)
— **PUT** : envoi du contenu d'un document, pour enregistrement sur le serveur
— **DELETE** : suppression d'un document
— **CONNECT** : établir un tunnel vers le serveur hébergeant une ressource (serveur proxy)
— **OPTIONS** : demande des options gérées par le serveur pour l'accès à une ressource
— **TRACE** : effectue un test d'accès complet le long du chemin d'accès à une ressource (ping / echo)

Pour une liste plus complète de toutes les méthodes des requêtes HTTP, consultez `http://webconcepts.info/concepts/http-method/`

Keep Calm En pratique, au-delà de cette séquence, vous utiliserez et gérerez principalement :
— `GET`
— `POST`
L'affaire se corsera pour ceux qui s'intéresseront aux services Web fonctionnant avec des API sur HTTP (REST)

Codes de réponse du serveur

De 100 à 199 Messages d'information

De 200 à 299 Succès. *Exemple* : `200 Okay`

De 300 à 399 Redirections

De 400 à 499 Erreurs du client. *Ex.* : `404 Not Found`

De 500 à 599 Erreurs du serveur. *Ex.* : `501 Internal Server Error`

En anglais *Status codes*.
Code numérique, complété par un message littéral
Aussi, 418 *I'm a teapot* …
Cf. `http://webconcepts.info/concepts/http-status-code/` pour une liste exhaustive

7.1.5 En-têtes sans mal de tête

Il existe de très nombreux en-têtes, de requêtes ou de réponses, qui complètent le contexte des demandes ou réponses.
Nous n'en présentons que certains, qui sont importants pour la compréhension générale du protocole et la mise au point des applications.

Rôle des en-têtes (*headers*)
- Ajoute du contexte (propriétés du client, autorisations …)
- Détaille les caractéristiques des flux (encodage, taille …)
- Optimisations (cache, …)
- Gestion d'une session au-dessus d'un protocole sans état (*cookies*, …)
- Éléments propres à l'application (extensible)

Il y a beaucoup d'en-têtes, mais seulement certains d'entre-eux sont réellement importants dans le cadre de ce cours (en gras)

En-têtes généraux Aussi bien dans requêtes ou réponses

Les caches et proxy `Cache-Control`, `Pragma`, `Via`

La connexion `Connection`

La date `Date`

Le codage `MIME-Version`, `Transfer-Encoding`

Les en-têtes généraux peuvent être présents aussi-bien dans les requêtes que dans les réponses HTTP.

En-têtes de requêtes
- Préférences du client : types, caractères, langage…
  - **`Accept`**
  - `Accept-Encoding`, `Accept-Charset`, `Accept-Language`, …
- Identification du client/serveur : site, e-mail, source…
  - **`Host`**
  - `From`
  - `Referer`
  - `User-Agent`

Ces en-têtes sont définis par le client qui les ajoute à partir de la 2e ligne du message de requête

`Accept` définit les priorités sur le format des données que le client aimerait recevoir de la part du serveur

`Host` précise au serveur sur quel site exactement les chemins d'URL sont spécifiés (quand plusieurs sites sont hébergés sur la même adresse IP)

En-têtes de requêtes (suite)
- Contrôle d'accès et sessions (login, cookies…)
  - **`Authorization`**
  - **`Cookie`**
- Conditions pour le serveur (sur la date…)
  - `If-Modified-Since`
  - `If-Match`
  - `If-Range`
  - `If-Unmodified-Since`

- Autres (pour proxy)
  - `Max-Forwards`

`Authorization` permet de transmettre au serveur des *lettres de créance* pour se faire connaître (cf. séquence ultérieure)

`Cookie` permet de transmettre (plus ou moins automatiquement) des données au serveur à chaque visite (requête GET). On verra que ça permet notamment de gérer des sessions dans les applications.

En-têtes de réponses

- Informations sur le contenu : type, taille, URL, encodage…
  - **`Content-Type`**
  - `Content-Length`
  - `Content-Encoding`
  - `Content-Location`
- Informations sur la date de modification ou la durée
  - `Last-Modified`
  - `Expires`

Ces en-têtes précisent notamment des informations (méta-données) relatives au contenu qui va être transmis par le serveur dans la suite de la réponse.

`Content-Type` précise le format de fichier qui sera présent dans les données

En-têtes de réponses (suite)

- Identification du serveur : nom, formats…
  - `Server`
  - `Accept-Range`
  - **`Location`**
- Compléments pour le client : durée, âge…
  - `Age`
  - `Retry-After`
  - `Warning`
- Autres :
  - `Allow : méthodes autorisées`
  - `Etag : entité de balisage`

`Location` est notamment utilisé en cas de redirection (code de statut 300 et +) pour indiquer la cible de la redirection

En-têtes de réponses (suite)

- Demande d'authentification
  - **`WWW-Authenticate`**
- Envoi de cookies
  - **`Set-Cookie`**

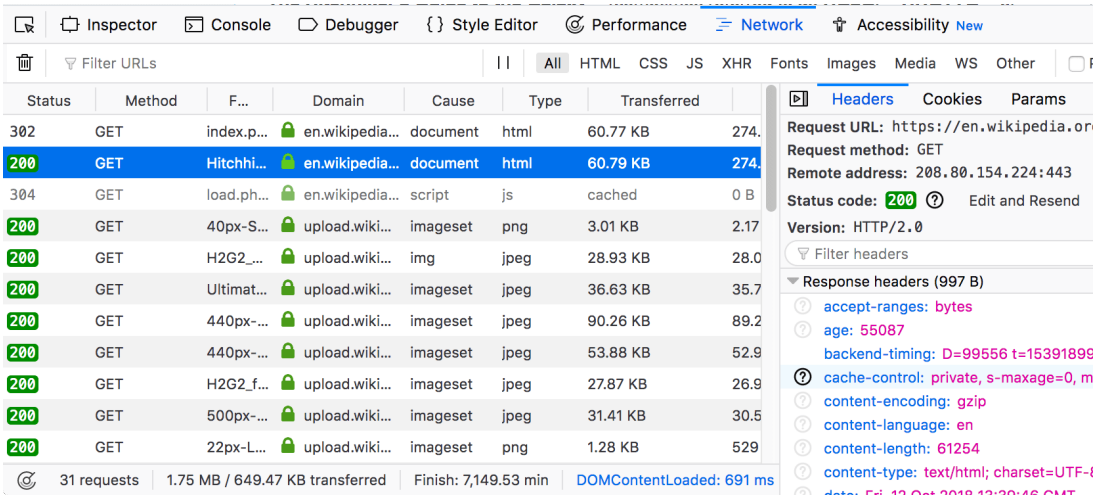
`WWW-Authenticate` complémentaire au code de retour indiquant que le contenu demandé est protégé (`401 Unauthorized`) et indique au client qu'il doit s'authentifier (sera vu dans une séquence ultérieure)

`Set-Cookie` permet au serveur de transmettre au client des données à conserver dans les cookies associés à cette URL (cf. plus loin).

7.2 Outils HTTP du développeur Web

L'objectif de cette section est de présenter certains outils qui vont nous servir dans la mise au point des programmes.

7.2.1 Moniteur réseau dans navigateur



Sera vu en partie TP

Cf. démonstration du « moniteur réseau » dans les outils du développeur Web intégrés aux navigateurs.

— Firefox
— Chrome

Mais aussi Poster pour Firefox et beaucoup d'autres outils

7.2.2 CURL en ligne de commande

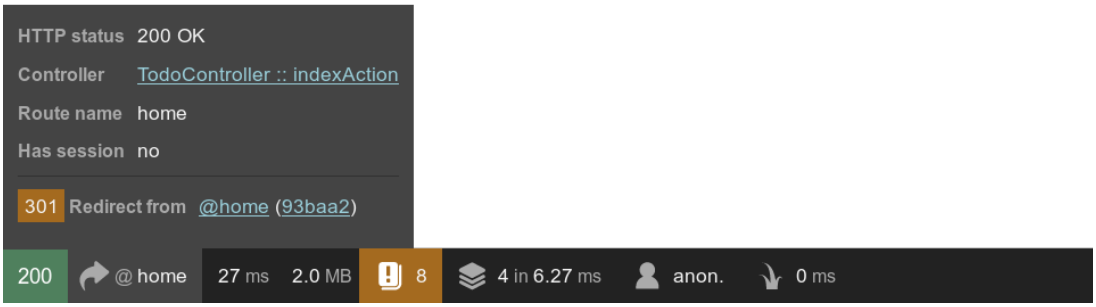
`https://curl.haxx.se/`

Client en ligne de commande

```
man curl
```

On verra dans la partie TP comment s'en servir, par exemple avec `curl -v 2>&1 | less`
Aussi `wget`, ou `get` autres outils populaire en ligne de commande.

7.2.3 Barre d'outils Symfony



Symfony fournit un ensemble d'outils « cachés » dans une barre d'outils en bas de pages Web, qui permet d'obtenir de multiples informations sur ce qui s'est passé pendant l'exécution du rendu de la page Web : base de données, gabarits, routage, etc.
C'est une aide précieuse pour les développeurs, et nous nous appuierons dessus dans les TP.

Take away

- client HTTP
- serveur HTTP
- format des requêtes / réponses dans HTTP
  - requêtes GET, POST, …
  - *status codes* : 200, 40x, …

Annexes

Rappel : Services sur Internet (TCP/IP)

- Un service sur Internet est caractérisé par :
  - une **application TCP/IP**
  - un **protocole**
  - un **numéro de port**
- Fonctionnement en mode Client/Serveur au dessus des couches réseau (IP, TCP ou UDP, …)
- Ports qui nous intéressent :
  - 80 par défaut (HTTP)
  - 443 par défaut (HTTPS)
  - 8000, par exemple si serveur configuré pour écouter sur ce port

Le Web n'est qu'une application parmi d'autres : ne pas confondre Internet et le Web
Services nombreux et variés.
Dans ce cours, on suppose une connaissance de base de TCP/IP et des réseaux.

Rappel : adresses IP

- Les URLs peuvent contenir l'adresse IP (v4 ou v6) du serveur. Ex. : `http://192.168.0.1/index.html`
  - Le serveur doit être lancé et écouter sur cette même IP
  - Attention aux problèmes de routage des paquets (firewall, NAT, …)
- Dans les TPs on utilisera souvent `localhost` : `127.0.0.1` (en IPv4), car dans l'environnement de développement Symfony, client et serveur sont sur la même machine.

En TP on se connectera à un serveur Web tournant en local, démarré depuis l'environnement de test de Symfony, donc tout se fait sur `http://localhost:8000/`
Pour tester « comme en vrai », ça peut devenir un peu plus compliqué au niveau réseau, selon la configuration de l'OS. Mais ceci dépasse le cadre du présent cours.
Le serveur peut connaître l'adresse apparente d'un client, mais ne peut pas toujours en faire quelque chose (proxy, NAT, etc.). Dans HTTP, le serveur n'en tient en général pas compte pour déterminer son comportement.

Évolution des spécifications de HTTP

Spécifications (version de juin 2014) :

- RFC 7230 *HTTP/1.1 : Message Syntax and Routing*
- RFC 7231 *HTTP/1.1 : Semantics and Content*
- RFC 7232 *HTTP/1.1 : Conditional Requests*
- RFC 7233 *HTTP/1.1 : Range Requests*
- RFC 7234 *HTTP/1.1 : Caching*
- RFC 7235 *HTTP/1.1 : Authentication*

Les RFC (*Request For Comments)* peuvent avoir valeur de standard de l'Internet (cf. `https://fr.wikipedia.org/wiki/Request_for_comments`).
Les spécifications d'HTTP ont été réécrites récemment, même si HTTP 1.1 est très vieux.

Versions HTTP historiques

- Version d'origine, notée HTTP 0.9 (1991)
  - Une seule méthode : `GET`
  - Pas d'en-têtes
  - Chaque requête = une connexion TCP
- HTTP version 1.0 (1996)
  - Introduction des en-têtes ==> échange de « méta » informations
  - Nouvelles possibilités :
    - utilisation de caches
    - authentification
    - connexion persistante
  - Ajout de méthodes : HEAD, POST…

HTTP version 1.1 (1997 -)

- Mode Connexions persistantes par défaut
- Exemple : page d'accueil avec 5 images
  - HTTP 0.9 ==> 6 connexions/déconnexions TCP
  - HTTP 1.1 ==> 1 SEULE connexion TCP
- Possibilité de transférer des séries d'octets

— Introduction du « *multihoming* » càd possibilité de gérer plusieurs noms de sites par un serveur

Bien que la version 1.1 soit assez ancienne, c'est aujourd'hui encore la version principalement utilisée sur le Web, donc celle avec laquelle vous allez travailler dans ce cours.

HTTP version 2 (2015)

— Tout nouveau
— Performances (binaire, compression, 1 seule connexion,…)

Spécs : Hypertext Transfer Protocol Version 2 (HTTP/2) RFC 7540

Pas forcément encore très diffusé sur le Web.
Pas utilisé dans le cadre de ce cours.

HTTP avancé

Notions avancées. Pas trop grave si tout ça n'est pas clair dès le départ.

Sécurité

— HTTP est en clair !
— => **Utiliser HTTPS**
— Mais attention : avoir confiance dans le DNS, ou se fier aux certificats

On verra les aspects de sécurité plus en détail en séquence ultérieure.

Contrôle d'accès Sera vue en séquence ultérieure.

Redirections Exemple : `http://www.example.com/` -> `http://www.example.org/`

```
curl http://www.example.com/
```

```
GET / HTTP/1.1
Host: www.example.com
User-Agent: curl/7.50.1
Accept: */*


HTTP/1.1 301 Moved Permanently
Location: http://www.example.org/
Content-Type: text/html
Content-Length: 174

<html>
<head>
<title>Moved</title>
</head>
<body>
<h1>Moved</h1>
<p>This page has moved to <a href="http://www.example.org/">http://www.example.org/</a>.</p>
</body>
</html>
```

Le navigateur Web n'affichera pas ce contenu HTML : par défaut, il suivra le lien et recommencera un `GET` sur l'URL reçue dans l'en-tête `Location`
Utiliser `curl -L` permet de suivre les redirects.
Par défaut les navigateurs effectuent les redirects, et les outils du développeur masquent parfois celà, sauf si on active une option permettant de garder l'historique des requêtes précédentes.

Proxy et cache

- Serveur *Proxy* : serveur HTTP intermédiaire
  - Diminuer le trafic.
- Cache : stockage d'informations (statiques)
  - Le navigateur dispose d'un cache à deux niveaux : en mémoire pour l'historique et sur disque entre les sessions.
- Exemple : 50 utilisateurs d'un site accèdent à une même page :
  - directement, elle sera transférée 50 fois
  - via un proxy, un seul transfert entre l'origine et le proxy
- Un proxy offre aussi d'autres services (autres protocoles, contrôles…)

Dans le cadre du cours, on n'explicite pas tous ces aspects. L'augmentation générale de la bande passante et des applications dynamiques tend à rendre les proxies moins utiles, en général.

Ressources / Documents

- HTTP agit sur des ressources, qui ne sont pas forcément des documents stockés physiquement sur le serveur.
- Pas uniquement du HTML (XML, JSON, RSS, HTML, PDF, JPG, CSS, JS, etc.).
- Types MIME, encodage.
- Négociation de contenu entre le client et le serveur
- Web Sémantique

Content-Negociation

- Le résultat de la requête sur une même URL dépend des en-têtes de la requête.

```
Accept: application/json, application/xml;q=0.9, text/html;q=0.8,
      text/*;q=0.7, */*;q=0.5
```

Priorité	Description
`q=1.0`	`application/json`
`q=0.9`	`application/xml`
`q=0.8`	`text/html`
`q=0.7`	`text/*` (ie. tout type de texte)
`q=0.5`	`*/*` (ie. n'importe quoi)

8 Séq. 4 : Serveur Web, invocation programmes

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.

8.1 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.

8.1.1 Role 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.

Mais aussi
— Vérifier la sécurité
— Gérer les performances
— …

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.

8.1.2 Programmes / Applications

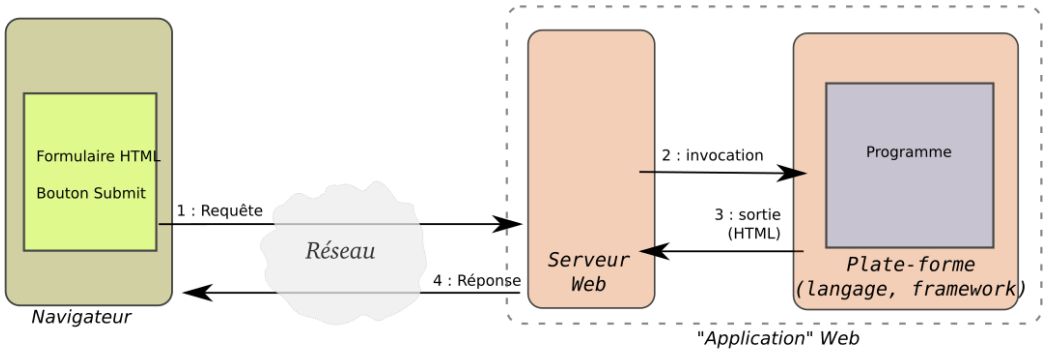
Modèle d'exécution

1. Attendre 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

Traitement typique d'une requête

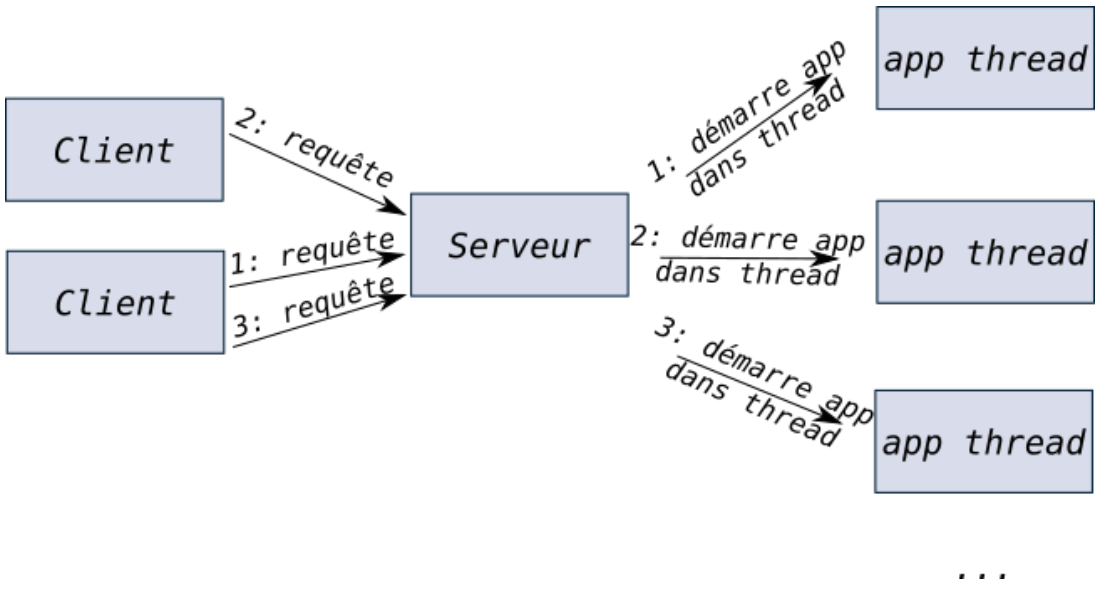
— Le serveur Web invoque un programme
— Programme s'exécute sur « plate-forme » : langage, etc.
— Programme fournit sortie au format HTML (en général) : transmise « telle-quelle »



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.

8.1.3 Multi-tâches

Concurrence d'exécution sur le serveur



8.1.4 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
    — Un serveur d'application séparé

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 biliothèques 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.

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.

« 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.

Serveur d'applications séparé

- 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.

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.

8.1.5 Exemple : programmes Web en 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)

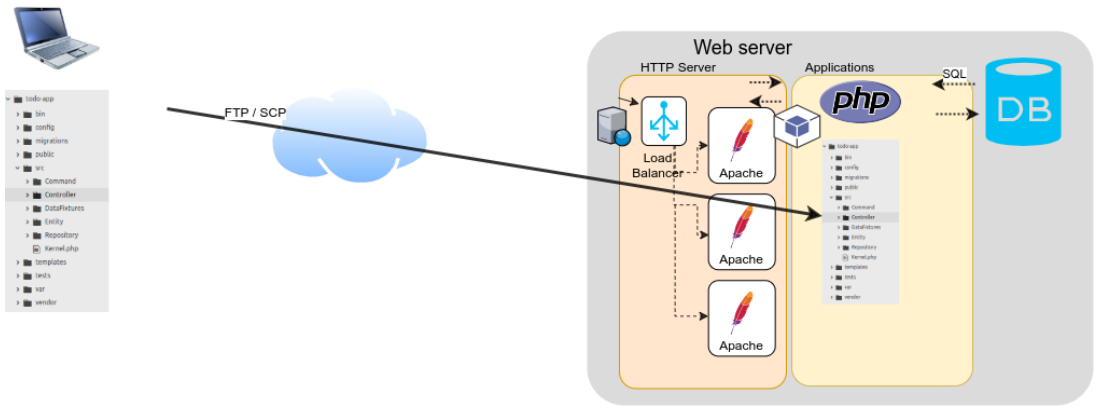
C'est le langage qu'on va utiliser dans ce cours.
On utilisera la plate-forme PHP fortement en TPs et Hors-Présentiel, mais pas le serveur d'applications PHP-FPM.
On utilisera par contre l'environnement de développement et de mise au point fourni par le *framework* Symfony, qui permet un mode de fonctionnement un peu différent pour tester des programmes PHP. Ce ne sera pas un serveur Web qui invoquera des programmes PHP, mais on utilisera l'interpréteur PHP pour qu'il fasse fonctionner un serveur HTTP basique en amont des programmes. 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.

8.2 Déployer

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

8.2.1 Serveur classique et recopie du code

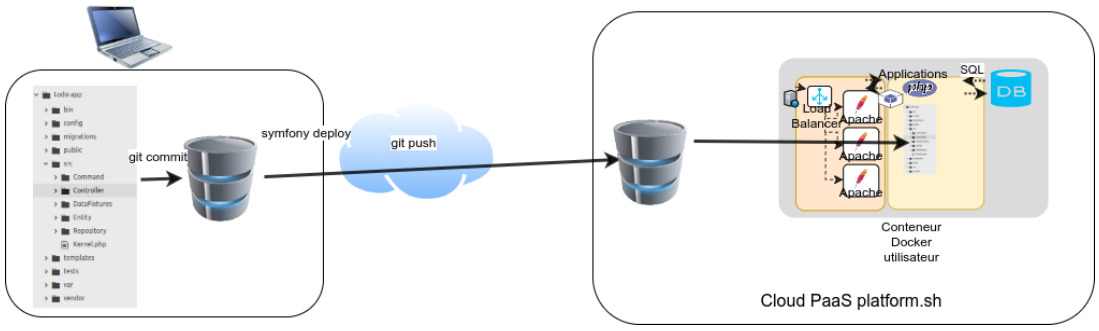
Exemple : avec FTP sur serveur Apache + PHP + SGBD



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.

8.2.2 Automatisation sur PaaS avec Git

Git push vers un hébergement *Cloud Platform as a Service*
Exemple : avec commande `symfony` CLI 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.

8.3 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.

8.3.1 Concevoir les programmes

Modèle de **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 arrivent sur chacun de ces chemins

8.3.2 Contexte d'invocation

— Données transmises au serveur Web
via 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

8.3.3 Invocation des programmes

— 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)

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).

Résultats d'exécution
— Statut d'un programme :
  — Code de retour système (exécution d'un processus POSIX) :
    — `0` ou `!= 0`
— Résultats :
  — Sortie standard du programme
  — Erreur standard ?

Conversion pour HTTP Construction de la Réponse HTTP
— Code de retour -> Code de statut

`0` ⇒ 200
`!= 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.

Serveur HTTP intégré à PHP
— 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`

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.

8.3.4 Interface HTTP moderne, en Objet

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

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

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

Contrôleurs HTTP Appel de **méthodes** dans une classe conçue pour gérer des **actions** (HTTP) sur des **ressources** (Web)
— Classe contrôleur (*Controller*)

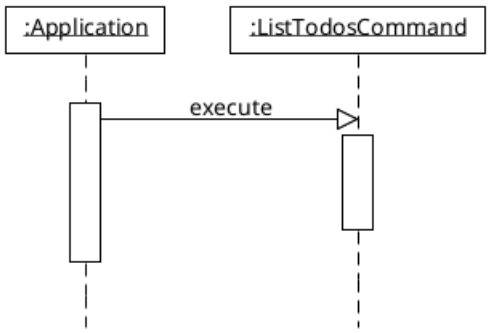


Figure 9 – « Invocation d'une méthode, en diagramme de séquence UML »

— « 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)

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.

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`

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

8.3.5 Routeurs et contrôleurs Symfony

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

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

Chemin	Nom	Args
/	home	

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

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

Méthodes gestionnaires de requêtes Méta-données en *attributs* PHP 8

```php
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 ...;
    }
    //...
}
```

8.3.6 Exécution éphémère

Temps de vie du code :

1. Requête entrante : Routage vers méthode Contrôleur
   (a) Chargement données depuis base (et/ou session)
   (b) Traitements
   (c) Sauvegarde en base (et/ou session)
2. Envoi Réponse
3. **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
— invocation des programmes qui génèrent du HTML
— programmation événementielle
— routage + controleurs Symfony

9 Séq. 5 : Génération de pages

Objectifs de cette séquence

L'objectif de cette troisième séquence du cours sera de présenter le standard ***HyperText Markup Language*** (HTML) et la façon dont il peut être produit pour réaliser des **interfaces** pour les applications Web.

On explicitera la façon dont les interactions entre un client HTTP et le serveur Web ont permis de passer d'une consultation de documents Hypertextes sur la Toile à la mise en œuvre d'applications interactives.

On présentera le composant **Twig** utilisé dans l'environnement Symfony pour **générer les pages** HTML côté serveur à partir d'un système de **gabarits** (*templates*).

9.1 Ressources, Pages, Vues

Cette section aborde trois notions liées, pour la couche présentation : pages, ressources et vues.

9.1.1 Récupération de représentation de ressources par HTTP

- CRUD :
  - *Create*
  - **Read** / Retrieve
  - *Update*
  - *Delete*
- on s'intéresse dans un premier temps à la **consultation** d'une (représentation de) ressource : méthode `GET` (+ URL)
- on verra la modification ultérieurement.

9.1.2 Représentation de ressource

- **Ressource** Web : objet d'une requête
  Exemple : page, document vidéo, formulaire, …
- **Représentation**
  - **document** (statique)
  - ou **vue** d'une application (dynamique)

Le format est transmis dans la réponse, via l'en-tête `content-type` On va se concentrer sur le format le plus courant pour la représentation des ressources, HTML.
Une même ressource peut avoir différentes représentations

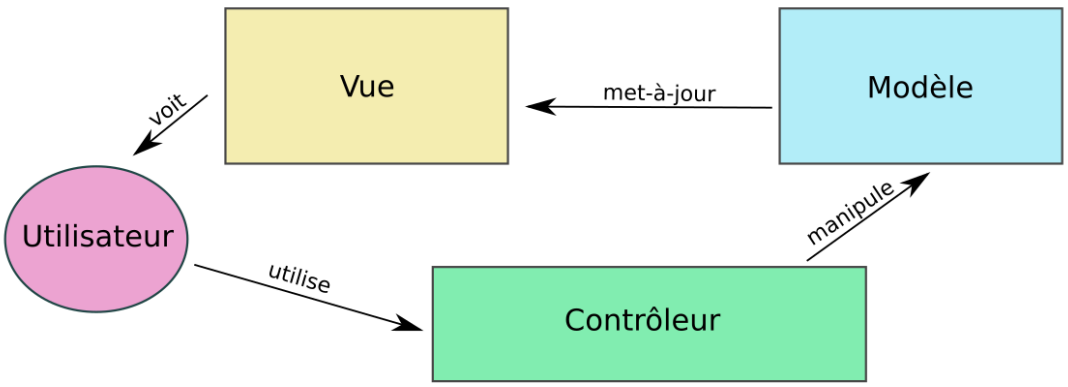
Questions philosophiques

- Ressource == Représentation ?
- Représentation == Visualisation réalisée par le client HTTP ?

Discussion philosophique, qui est parfois importante à prendre en compte…
Une ressource a beau être identifiée de façon unique par son URI, elle peut avoir de multiples représentations.
En déréférençant une URI, on peut demander à un serveur Web « faisant autorité » de donner sa représentation, canonique… mais sur le Web, les choses sont parfois plus floues que celà, notamment quand on parle du Web sémantique (aussi appelé Web des données). Exemple : page Wikipedia qui parle d'une ressource, comme par exemple un personnage historique. Un personnage historique n'a pas d'URL propre, ce qui n'empêche pas de le représenter, et de considérer qu'une représentation effectuée par Wikipedia peut faire autorité.
On peut aussi se demander si un même document, par exemple une représentation sous forme de document HTML donne toujours la même représentation sur tous les écrans… en principe oui, pour les informations importantes, la structure du contenu. Si le développeur a bien fait son travail.
Cf. l'allégorie de la caverne…

9.1.3 Vues d'une application

La **vue** représente un **état** d'une application interactive, pour l'utilisateur
Patron de conception **M V C** (Modèle, **Vues**, Contrôleur)



Modèle déjà vu avec Doctrine.

Ce diagramme représente une version simplifiée d'une boucle de rétroaction sur l'utilisation d'une application.
On parle souvent du patron de conception MVC, de *frameworks* MVC.
C'est un modèle d'architecture d'application interactive classique, qui se retrouve pas uniquement dans le monde Web.

9.1.4 Contenu d'une page Web

Page Web = *représentation* d'une **vue** de l'application (affichable dans une navigateur)
Visualisation d'une page contenant différentes ressources
- **document HTML principal**
- ressources additionnelles : images, *feuilles de style*, *scripts*, …

Hypertexte : proviennent pas toutes du même serveur HTTP (requêtes supplémentaires)

À noter qu'une partie de ces éléments peuvent être mis en cache dans le navigateur.

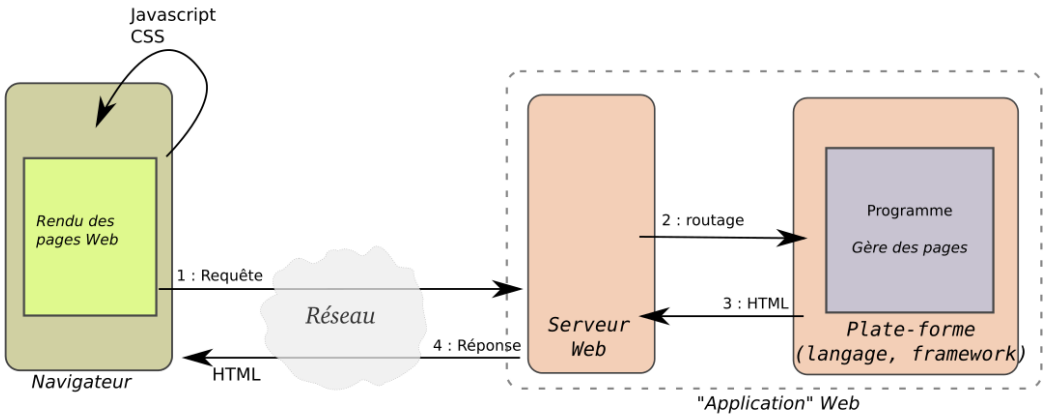
9.1.5 Recette (classique) de fabrication d'une page

- Générer une **représentation** d'une ressource gérée par l'application

  - **HTML** (construit côté serveur)
  - transmise dans la réponse HTTP
- Avoir prévu :
  - ressources statiques complémentaires : **CSS**, Images, **JavaScript**, …
  - outils d'adaptation dynamique (exécution CSS et JavaScript côté client)

On se concentre maintenant sur la génération du contenu HTML qui va servir d'interface à nos programmes, mais d'autres formats de représention sont possibles et utiles (JSON, XML, etc.).
Les CSS et images sont en général statiques : définis une fois pour toute par les développeurs, et mis en ligne tels-quels.
La visualisation finale dépend de l'interprétation du HTML, du CSS et de l'exécution du JavaScript qui modifie le contenu de la page.

Construction de la page à visualiser



Fabriquer la page

soit côté serveur générer du HTML qui transite vers le navigateur dans la réponse à **une** requête HTTP

soit côté client un programme **Javascript** s'exécute dans le contexte d'une page chargée par le navigateur, et construit du DOM **en mémoire** pour enrichir cette page (et déclenche **d'autres requêtes** HTTP)

Dans ce cours, principalement 1ère option (application Web « classique »)

La construction dynamique du contenu des pages sera vue en fin de cours.

9.1.6 Langages standards pour pages Web

Examinons maintenant plus eb détails les techniques dont nous disposons pour mettre en œuvre les pages de l'application, avec les langages standard compatibles avec tous les navigateurs

Séparation structure / mise en forme

- **HTML** : structure de la page, texte, méta-données
- **CSS** : mise en forme

On ne va pas rentrer dans les détails des langages HTML et CSS : pas un cours de Web designer.
CSS sera vu plus spécifiquement dans la prochaine séquence de cours.

9.2 HTML

HyperText Markup Language

9.2.1 Structure

- Arbre sections, sous-section (plan document textuel), visible :
  - `<body>`
    - `<h1>...</h1>`
      - `<h2>...</h2>`
      - `<p>...</p>`
    - `<h1>...</h1>`
  - `</body>`

  mise en forme « par défaut » par navigateur
- **Structure additionnelle** : sections / zones (pas forcément visible, par défaut, mais modifiables par le **CSS**) :
  - `<div>...</div>`
  - `<span>...</span>`

La structure « pas forcément visible » prend tout son sens lorsqu'on ajoute du CSS.
Tester la fonction d'affichage sans styles du navigateur.

9.2.2 Balises sémantiques

- `<article>`
- `<aside>`
- `<details>`
- `<figcaption>`
- `<figure>`
- `<footer>`
- `<header>`
- `<main>`
- `<mark>`
- `<nav>`
- `<section>`
- `<summary>`
- `<time>`

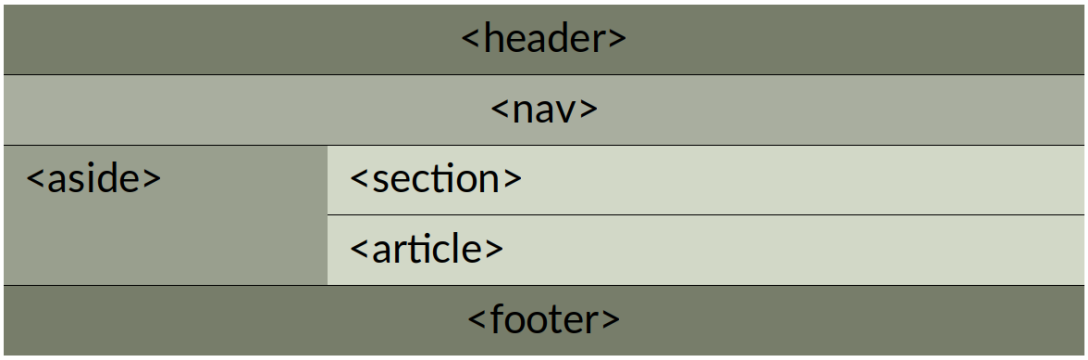


Figure 10 – « Mise en forme typique des balises sémantiques »

Le navigateur ne sait pas forcément quoi faire de ces sections, donc il faut obligatoirement un CSS associé.
Par contre, pour l'accessibilité, par exemple, on peut sauter directement à la lecture des sections en ignorant les en-têtes et autres.

9.2.3 Quelques balises

Listes à puces :

```
<ul>
  <li> ... </li>
  <li> ... </li>
</ul>
```

Lien vers une autre ressource :

```
<a href="/todo/42">Coder en HTML</a>
```

9.2.4 Formulaires

Soumission de données pour les applications : génère des requêtes HTTP vers des ressources.
-> séances suivantes

Sera vu principalement dans 2 séance.

9.2.5 Code source

Pomper le code source dans le navigateur
`Ctrl U` !

On peut suivre des cours sur HTML, mais ce qui a fait son succès, c'est que le code source est visible, depuis le navigateur, via un raccourci clavier, le menu « Voir le source de la page », les outils du développeur comme l'inspecteur, etc.

9.2.6 Produire du HTML avec PHP

— « affichage » d'HTML sur la **sortie standard** du programme

```
echo "<html>...</html>";
```

ou

```
print("<html>...</html>");
```

— Mieux : **Bibliothèque de génération HTML**
  — Gabarits (*Templates*)

Cf. la section précédente pour les mécanismes d'appel des programmes en PHP par le serveur Web.
La sortie standard est transmise dans la réponse à la requête HTTP, d'où la possibilité d'utiliser `echo` ou `print()`.
Dans d'autres langages de programmation, le principe est équivalent.

9.3 Gabarits (*templates*)

9.3.1 Génération du HTML grâce aux gabarits

— Page construite à partir :
  — structure de la page : **gabarit** (statique)
  — **données** générées par le programme (dynamiques)
— Génération HTML par moteur de *templates*

Vieux principe de conception (architecture / couture / …)
Le principe d'un gabarit, modèle ou patron est de prévoir la structure générale, les proportions des différents éléments.
On décompose en vues plus simples spécifiques à certains éléments pour rendre la compréhension plus simple et permettre de faire évoluer des sous-éléments indépendamment.

9.3.2 Moteur de gabarits Twig

`https://twig.symfony.com/`
— Moteur de *templates* pour PHP, utilisé dans Symfony
— Utilisable seul
— Syntaxe inspiré de *Jinja* (populaire en Python, avec Django)

Syntaxe inspirée de celle du moteur de templates populaire Jinja, du monde Python : syntaxe à la Python…

9.3.3 Exemple : liens d'un menu

Résultat souhaité (liste à puces) :

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Ma page</title>
    </head>
    <body>
      <ul id="navigation">
        <li><a href="products/">Produits</a></li>
        <li><a href="company/">Société</a></li>
        <li><a href="shop/">Boutique</a></li>
      </ul>
      <h1>Ma page</h1>
      Salut le monde
    </body>
  </html>
```

On veut un menu sous forme de balises `<li><a href="...">...</a></li>` identiques. La seule différence est le lien cible (contenu attribut `href`) et son libellé (texte dans la balise `<a>`, appelé *caption*).

Gabarit de base d'un élément de menu Motif à répéter :

```
<li>
  <a href="{{ href }}">
    {{ caption }}
  </a>
</li>
```

Données à injecter

```
array(
          array('href' => 'products/', 'caption' => 'Produits'),
          array('href' => 'company/', 'caption' => 'Société'),
          array('href' => 'shop/', 'caption' => 'Boutique')
     );
```

On alimente le moteur de template avec le template précédent, en lui fournissant la bonne structure de données, par exemple, ici, un tableau associatif en PHP.
Twig fonctionne aussi avec d'autres types de données : JSON, etc.

9.3.4 Gabarit de page complet

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ma page</title>
  </head>
  <body>
    <ul id="navigation">

      {% for item in navigation %}
        <li><a href="{{ item.href }}">{{ item.caption }}</a></li>
      {% endfor %}

    </ul>
    <h1>Ma page</h1>

    {{ a_variable }}

  </body>
</html>
```

Listing 8 : Template `home.html.twig`

Boucle `for` qui itère pour construire une barre de navigation contenant des liens hypertextes, à partir d'une liste de couples (URL, titre du lien), et ajoute la valeur d'une variable dans le corps de la page.

Données complètes

```
array(
        'navigation' => array(
                array('href' => 'products/', 'caption' => 'Produits'),
                array('href' => 'company/', 'caption' => 'Société'),
                array('href' => 'shop/', 'caption' => 'Boutique')
        ),
        'a_variable' => "Salut le monde"
);
```

Résultat HTML généré

```
<!DOCTYPE html>
  <html>
    <head>
      <title>Ma page</title>
    </head>
    <body>
      <ul id="navigation">
        <li><a href="products/">Produits</a></li>
        <li><a href="company/">Société</a></li>
        <li><a href="shop/">Boutique</a></li>
      </ul>
      <h1>Ma page</h1>
```

```
      Salut le monde
    </body>
  </html>
```

Une fois la boucle et les substitutions effectuées, le moteur de template a généré cette page HTML.
Avantage : si on décide de renommer les URLs des rubriques du site, pas besoin de retoucher le HTML. A priori, un seul endroit impacté dans le code.

9.3.5 Syntaxe Twig

Variable `{{ var }}`

Instruction `{% ... %}` : `for i in`, `if`, `block` …

Commentaire `{# ... #}` Différent de `<!-- ... -->` (commentaire HTML)

9.3.6 Programmer en Twig

— Programmation simple
  — Mélange HTML et instructions Twig
  — Algorithmique basique : boucles, conditionnelles
  — syntaxe à la *Python*
— *Pattern matching* (filtrage par motif)
— Structuration en motifs factorisables (va en général de pair avec les CSS)

Tests sur `http://twigfiddle.com/`

Exemple code d'appel à Twig

```php
<?php
require_once("../lib/Twig/Autoloader.php" );
Twig_Autoloader::register();

$loader = new Twig_Loader_Filesystem('../templates/');

$twig = new Twig_Environment($loader);

$data = array('navigation' => array(
      array('href' => 'products/', 'caption' => 'Produits'),
      array('href' => 'company/', 'caption' => 'Société'),
      array('href' => 'shop/', 'caption' => 'Boutique')
    ),
              'a_variable' => "Salut le monde");

echo $twig->render('home.html.twig', $data);
```

On verra que Twig est intégré dans Symfony, permettant de faire à peu près le même genre de choses.
Ce genre de bouts de code PHP très simple à écrire peut être utilisé pour faire un prototype fonctionnel et tester les Vues de l'application auprès des utilisateurs, sur la base des données de test (statiques, comme la variable `$data` dans l'exemple ci-dessus).

9.3.7 Conception des gabarits

Surcharge
— Définir une structure de page standard contenant des **blocs**
— Compléter le contenu dans chaque page spécifique en surchargeant des blocs

Structure de base `base.html.twig`

```
<!DOCTYPE html>
<html>
    <head>
        {% block head %}
            <link rel="stylesheet" href="style.css" />
            <title>{% block title %}{% endblock %}</title>
        {% endblock %}
    </head>
    <body>
        <div id="content">
            {% block content %}{% endblock %}
        </div>
        <div id="footer">
            {% block footer %}
                &copy; Copyright 2016 by ...</a>.
            {% endblock %}
        </div>
    </body>
</html>
```

Cette structure principale permet de se définir et faire évoluer le design général des pages de l'application : inclusion de CSS, modification de la présentation, etc.
Chaque page sera générée à partir de ce template de base, en spécialisant le bloc `content` (initialement vide).
Avantage : un seul fichier impacté en cas de changement du look du site.
Possibilité de spécialiser les CSS de chaque sous-rubrique du site, par exemple, en surchargeant le contenu par défaut du bloc `head`.

Spécialisation pour une page `accueil.html.twig`

```
{% extends "base.html.twig" %}

{% block title %}Accueil{% endblock %}

{% block content %}
    <h1>Accueil</h1>
    <p class="important">
        Bienvenue sur ma page géniale
    </p>
{% endblock %}
```

Exemple de template pour une page du site, qui surcharge (`extends`) le bloc `content`.
Similaire à la surcharge dans les classes filles en programmation orientée objet : spécialisation.

Mockup exécutables

1. Conception des maquettes (*mockups*)
2. Codage des *templates* et CSS
3. Validation des interfaces sur un prototype
4. Implémentation du reste de l'application

Travail en parallèle de différentes équipes
Si on connaît HTML, on peut faire du Twig facilement.

9.4 Outillage Symfony

On donne maintenant quelques détails sur des outils utiles à la construction des vues, dans l'environnement Symfony.

9.4.1 Coder le MVC

— PHP + Doctrine : Modèle des données
— HTML (+ CSS) + **Twig** : Vues
— PHP : Contrôleurs (routeur, etc.)
Twig s'interface avec le modèle Doctrine => moins de code à écrire en PHP ø/
Inconvénient : pas mal de syntaxes différentes

On a déjà vu comment le modèle de données est géré avec Doctrine.
La génération des vues est maintenant dans notre panoplie, grâce à Twig.
Il nous manquera donc les contrôleurs, pour compléter le tour des technologies utiles pour mettre en œuvre MVC. On étudiera les contrôleurs Symfony dans les séquences ultérieures.

9.4.2 Mise au point

— *logs*
— *dump()*
— *Data fixtures*
— *ligne de commande*
— **barre d'outils Symfony**

On a déjà vu certains outils de mise au point dans Symfony.
La partie vues n'est pas en reste.

9.4.3 dump()

En PHP :

```php
$var = [
    'a simple string' => "in an array of 5 elements",
    'a float' => 1.0,
    'an integer' => 1,
    'a boolean' => true,
    'an empty array' => [],
];
dump($var);
```

```
array:5 [▼
  "a simple string" => "in an array of 5 elements"
  "a float" => 1.0
  "an integer" => 1
  "a boolean" => true
  "an empty array" => []
]
```

En Twig :

```
{% dump var %}
```

Attention :

```
{% dump todos %}

{% for todo in todos %}
```

```
    {# the contents of this variable are displayed on the web page #}
    {{ dump(todo) }}

    <a href="/todo/{{ todo.id }}">{{ todo.title }}</a>
{% endfor %}
```

Un appel à la fonction `dump()` dans le code PHP affiche des informations sur les données.
C'est la même chose dans Twig, avec une syntaxe légèrement différente : c'est une instruction à traîter, donc dump est entouré par `{%` et `%}` : `{% dump todos %}`.
Les données *dumpées* ne sont pas affichées en plein milieu de la page, mais masquée et visibles dans une section dédiée de la barre d'outils Symfony.
Attention, toutefois à l'utilisation des doubles accolades (comme dans `{{ dump(todo) }}`) qui signifie, dans Twig, l'affichage d'une valeur dans la page, et va donc afficher les données *dumpées* directement dans la page. Ça fonctionne, mais ce n'est pas idéal, notamment en cas de redirection qui va supprimer le contenu affiché.
L'utilisation de `dump` dans les gabarits pollue un peu le code, mais offre l'avantage de ne fonctionner qu'en environnement de développement. `dump` n'est plus exécuté une fois l'application déployée en production. Le code peut contenir des instructions pour la mise au point et n'a pas forcément à être nettoyé avant la livraison en production.

9.4.4 Barre d'outils

— Requêtes
— Réponses
— **Gabarits**
— Doctrine
— Performances
— …

Cette fois encore, comme pour HTTP, la barre d'outils affichée en bas de pages fournit des outils spécifiques pour les gabarits, pour comprendre la compilation des blocs, notamment.

Take away

— HTML
— Modèle, Vues, Contrôleurs
— Gabarits pour mettre en œuvre les Pages (production des pages HTML avec Twig)
— Surcharge des blocs

Postface

Crédits illustrations et vidéos

— logo Twig © 2010-2017 SensioLabs

10 Séq. 6 : Expérience utilisateur Web

Objectifs de cette séquence

Cette quatrième séquence de cours porte sur différents aspects de l'**expérience utilisateur** dans l'interaction avec les applications Web.
On présentera les enjeux généraux de l'**ergonomie** (autrement appelée « expérience utilisateur ») des interfaces utilisateur ainsi que les problématiques d'**accessibilité**.
On détaillera les fonctionnalités principales, dans cette optique, pour la construction et l'**habillage des pages HTML**, que sont les standards ***Document Object Model*** (DOM) et ***Cascading Style Sheets*** (CSS).
On verra enfin comment les contrôleurs de l'application Symfony permettent de gérer les transitions hypertexte au sein des pages de l'interface d'une application Web.

10.1 Interface utilisateur

Cette section présente les principes généraux qui guident la conception d'interface utilisateur, en général (y compris en dehors du domaine des applications Web)

10.1.1 Interface homme-machine (IHM)

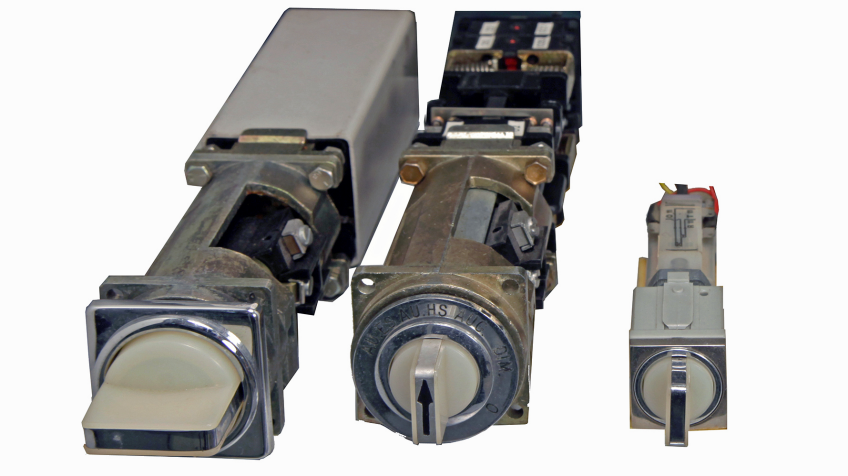


Figure 11 – Boutons Tournés Poussés Lumineux

Cette illustration présente des boutons « Tournés Poussés Lumineux ». Leur fonction est d'éviter la commande d'organes sensibles : typiquement commande d'organes dans des postes de transformation électrique ou centrales nucléaires. Le but est de forcer l'opérateur à réfléchir à deux fois :

1. tourner : choisir la fonction
2. pousser : activer la fonction
3. allumage lampe : acquittement de la fonction correctement exécutée

Ce genre de widget est difficile à informatiser.
Pour la petite histoire on travaillait sur ce genre de fonctionnalité pour EDF dans les années 90, pour les postes d'exploitation informatisés : GUI sous X/Motif au lieu de tableau synoptiques physiques…

10.1.2 Interface utilisateur

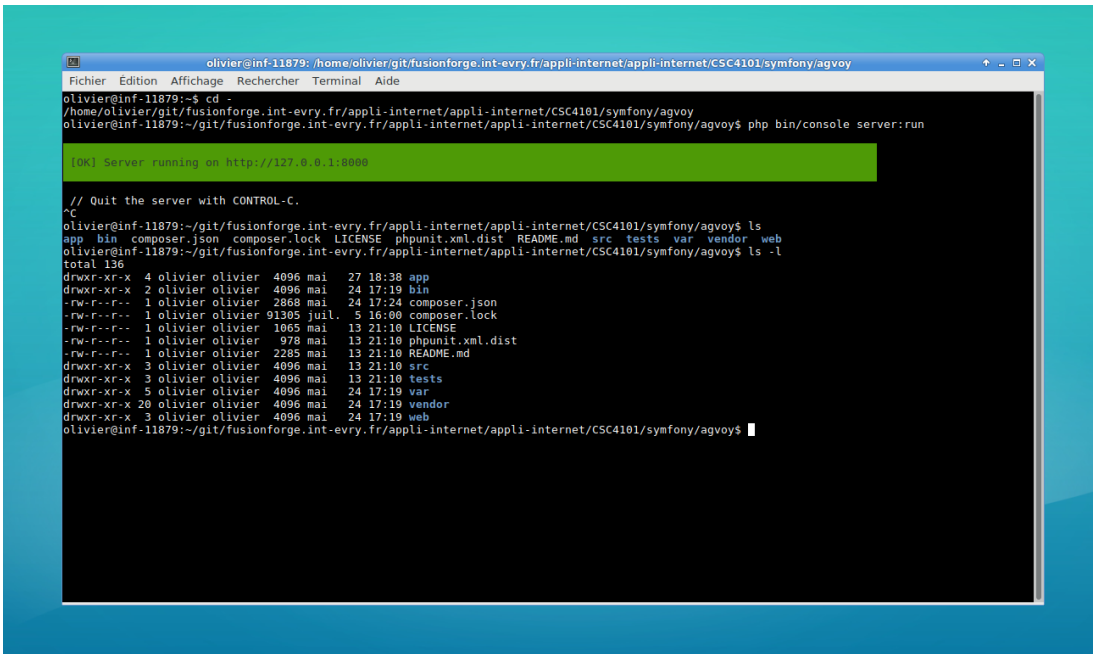


Figure 12 – Ligne de commande bash

Vous connaissez bien cette interface utilisateur, en ligne de commande.

10.1.3 Propriétés de la ligne de commande

Command-Line Interface (CLI) :
— Codifiée
— Stricte
— Statique

Cf. CSC3102
Améliorations possibles avec libcurses, etc.
Pratique : tunnel via SSH : peu de bande passante, *screen* pour rendre la session rémanente, Emacs, convient bien aux interfaces braille, etc.

10.1.4 Interface utilisateur

Un peu avant le Macintosh d'Apple, le Xerox Alto invente l'interface graphique avec souris.
Cf. Xerox PARC et la naissance de l'informatique contemporaine par Sacha Krakowiak
Anecdote perso : j'ai utilisé une telle interface sur Geos sur Commodore 64 pour mon premier rapport rendu en *WYSIWYG* (*What You See Is What You Get*) au collège dans les années 80.

10.1.5 Propriétés des interfaces graphiques

Graphical User Interface (GUI) :
— Métaphore (bureau)
— Exploratoire

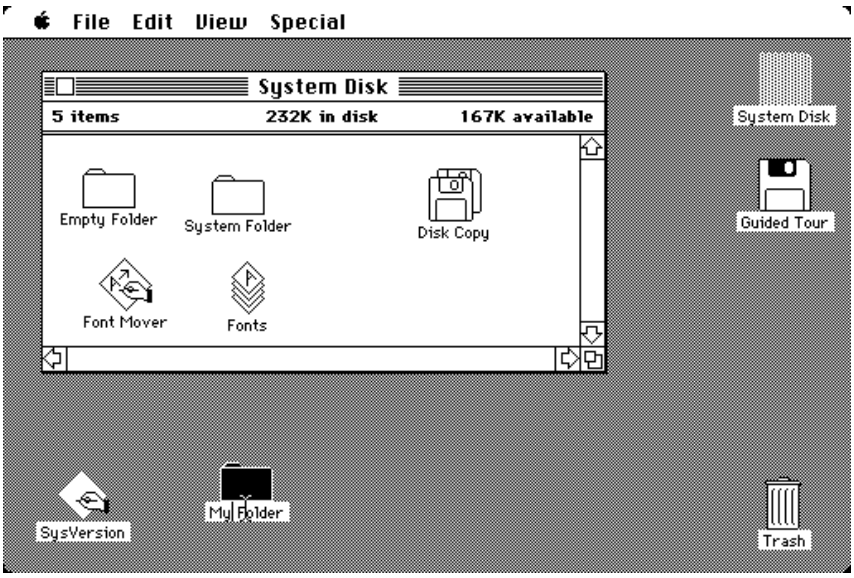


Figure 13 – Desktop du Macintosh d'Apple

10.1.6 Interfaces de 0 à 77 ans

Vidéo `https://www.youtube.com/watch?v=gc9NpYrbZgQ` par `UserExperiencesWorks`

10.1.7 Propriétés des interfaces « naturelles »

Natural User Interface (NUI) :
— Directe (passage de novice à expert facilement)
— Naturelle / Intuitive

10.1.8 Importance des ordiphones (mobiles)

Plate-forme préférentielle (majorité des utilisateurs)
Utiliser technos Web pour applis mobiles :-)

10.2 Interfaces Web

Voyons les caractéristiques propres des interfaces des applications Web.

10.2.1 Qualité des interfaces Web

On va passer en revue quelques grandes propriétés intéressantes du point de vue de la qualité des interfaces des applications Web.

Ergonomie
— **Expérience utilisateur** (*User eXperience : UX*)
— Utilisabilité :
  — apprenabilité (novices)
— « Fléau » de l'abandon de panier
— Utilisateurs « PIP » : *Pressés, Ignorants et Paresseux*

Le but ultime est que l'« expérience » de navigation sur l'application soit bonne pour les utilisateurs… sachant qu'ils n'ont pas tous les mêmes compétences, ni les mêmes capacités.
C'est un art difficile à atteindre, en pratique.

Portabilité Le Web est la plate-forme universelle.
— Standardisation = portabilité (merci HTML5)
— Applications mobiles :
    — Développé en HTML
    — « Compilé » en *toolkit* natif (ex. Apache Cordova)
— **Attention** : prédominence de Chrome de Google… *Best viewed in Chrome*

Sur le contexte de la standardisation, et l'émergence actuelle de Chrome, les « *Browser Wars* » : `https://fr.wikipedia.org/wiki/%C3%89volution_de_l'usage_des_navigateurs_web`

10.2.2 Accessibilité

Vidéo : `https://www.youtube.com/watch?v=DePdWynmd_Y`
Source : article Comment les aveugles utilisent-ils internet ? de *l'Obs* Voir aussi Usage plage braille et synthèse vocale par personne non voyante de `https://design.numerique.gouv.fr/`.

Cette vidéo illustre comment les aveugles ou mal-voyants utilisent le Web, avec plage braille et logiciel de synthèse vocale.
La problématique de l'accessibilité dépasse de loin la situation des personnes en situation de handicap « reconnue ». Elle concerne également tous les utilisateurs à partir du moment où leur utilisation des applications Web n'est pas idéale, car ils sont en mode « dégradé » (petit écran sur *smartphone*, un seul doigt de libre, réseau lent, environnement bruyant, etc.)

Obligations
— Règles pour l'accessibilité des contenus Web (WCAG) 2.1
    — Différents niveaux de priorité / niveaux de conformité à la norme
    — Principes : Perceptible, Utilisable, Compréhensible, Robuste
    — Règles, critères de succès
    — Techniques suffisantes et techniques recommandées
— Référentiel Général d'Amélioration de l'Accessibilité (RGAA) version 4.1.2 (avril 2023)

Certains éléments du standard *Web Content Accessibility Guidelines* (WCAG) deviennent obligatoires, par exemple pour les sites Web des administrations publiques, d'où le RGAA.
Référence sur le sujet : `http://www.accessiweb.org/` de `http://www.braillenet.org/`

Principes WCAG 2.0

Principe 1 : perceptible
— 1.1 Proposer des **équivalents textuels** à tout contenu non textuel qui pourra alors être présenté sous d'autres formes selon les besoins de l'utilisateur : grands caractères, braille, synthèse vocale, symboles ou langage simplifié.
— 1.2 Proposer des versions de remplacement aux média temporels.
— 1.3 Créer un contenu qui puisse être présenté de différentes manières sans perte d'information ni de structure (par exemple avec une mise en page simplifiée).

— 1.4 Faciliter la perception visuelle et auditive du contenu par l'utilisateur, notamment en séparant le premier plan de l'arrière-plan.

Principe 2 : utilisable
— 2.1 Rendre toutes les fonctionnalités **accessibles au clavier**.
— 2.2 Laisser à l'utilisateur suffisamment de temps pour lire et utiliser le contenu.
— 2.3 Ne pas concevoir de contenu susceptible de **provoquer des crises**.
— 2.4 Fournir à l'utilisateur des **éléments d'orientation** pour naviguer, trouver le contenu et se situer dans le site.

Principe 3 : compréhensible
— 3.1 Rendre le contenu textuel lisible et compréhensible.
— 3.2 Faire en sorte que les pages apparaissent et fonctionnent **de manière prévisible**.
— 3.3 Aider l'utilisateur à éviter et à corriger les erreurs de saisie.

Principe 4 : robuste
— 4.1 Optimiser la compatibilité avec les agents utilisateurs actuels et futurs, y compris avec les technologies d'assistance.
etc.

La suite sur DesignGouv

> Concevons des services publics numériques accessibles, inclusifs et humains.

`https://design.numerique.gouv.fr/`

Évaluation qualité
— Critères Opquast (*Open Quality Standards*) : `https://checklists.opquast.com/fr/qualiteweb/`

Cf. bibliographie en annexes

10.3 Habillage des pages Web

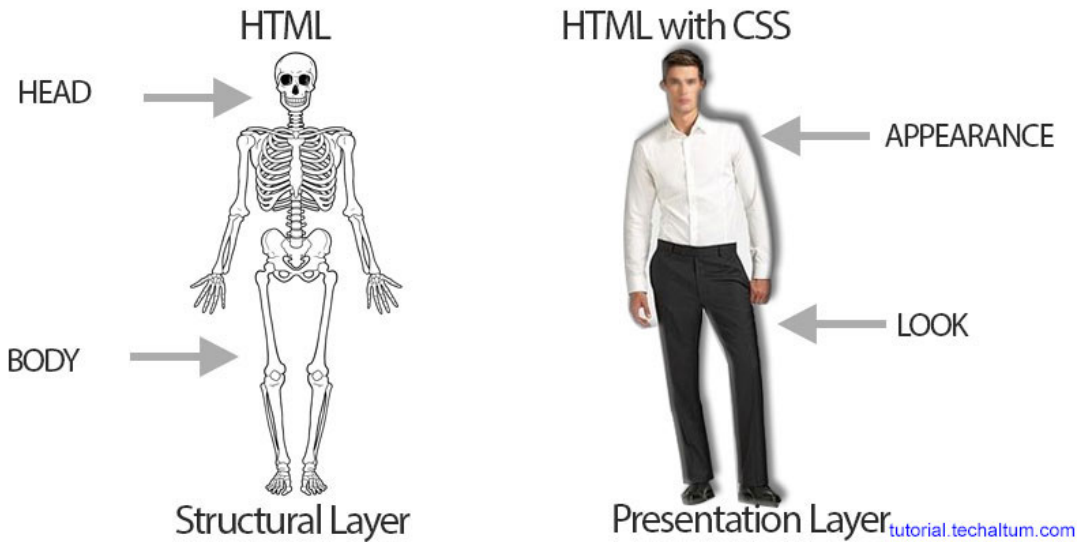
Voyons maintenant ce qui concerne la mise en forme, l'habillage des pages Web de l'application.

10.3.1 Découplage structure / présentation

(*suite*)

Page Web et document HTML Conversion de **documents** en **une présentation** dans le navigateur.

HTML + CSS



Chaque langage a son rôle :
- HTML (5) : la structure du document, essentiellement sémantique
- CSS : les règles de présentation

Structure des sources des pages Une « page Web » :
— un document HTML (maître)
— plus des images, boutons, menus, etc. (éventuellement externes)
Un document (HTML) :
— un arbre de rubriques,
— sous-rubriques,
— etc.

Arbre de balises HTML (DOM : *Document Object Model*)

On visualise des ressources (URL), dont le serveur produit une représentation en HTML, pour qu'elle soit visualisée par un navigateur.
La structure d'un document HTML est essentiellement arborescente (langage à balises ouvrantes/fermantes).
En général on associe une sémantique à cette arborescence : décomposition en concepts, sous-concepts, etc.
Un standard existe pour décrire et parcourir cet arbre : DOM (*Document Object Model*).
Mais l'utilisateur humain ne consulte pas un arbre de balises.

Structure d'une page affichée Une page affichée en 2D (ou imprimée) :
— des boîtes qui contiennent d'autres boîtes
— boîtes qui contiennent texte ou images, etc.
— texte qui contient des caractères
— caractère qui contiennent des pixels
— …

C'est plus ou moins le modèle d'une page d'un document papier.
Là, la page est purement syntaxique (on perd la structure sémantique du document).
On trouve ce genre de structure dans PostScript par exemple.

Arbre DOM et rendu des pages
— Le navigateur (moteur de rendu) convertit :

  - Arbre DOM : *Document Object Model*

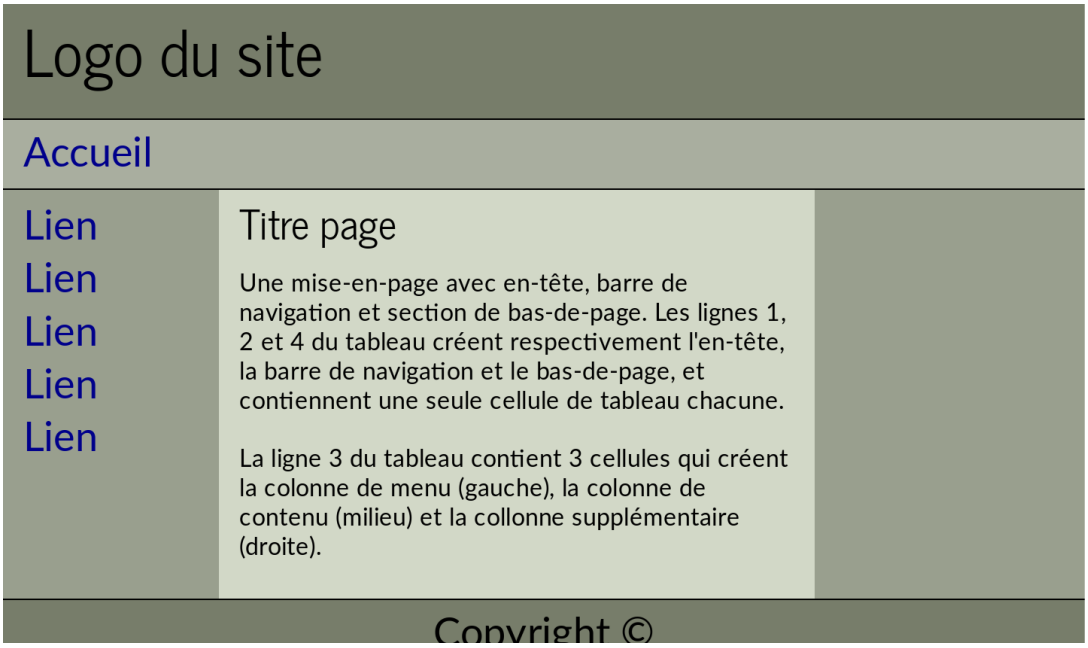
en :

  - Arbre d'éléments à afficher

- **Règles de conversion**
  - Prédéfinies (navigateur)
  - Programmables (**CSS**)

Le travail du navigateur est la conversion de l'arbre de balises en une page à deux dimensions équivalente au texte d'un document papier.
Le moteur de rendu du navigateur est un engin loin d'être trivial.
Le programmeur n'a pas besoin de spécifier la façon dont ce rendu est effectué, mais il peut le faire en détaillant les attributs de mise en forme des éléments graphiques (CSS).

Exemple de page avec un tableau



On décide d'afficher à l'utilisateur une page Web structurée globalement comme un tableau.
Toutes les cases de ce tableau n'ont pas la même taille, la même couleur, etc.
Comment peut-on construire un document HTML pour représenter cette page.

Structure HTML correspondante ?

```
<table>
  <tr> <!-- SECTION EN-TETE  -->
    <td colspan="3">
      <h3>Logo du site</h3>
    </td>
  </tr>
  <tr>  <!--  SECTION BARRE NAVIGATION == -->
    <td colspan="3">
      <a href="#">Accueil</a>
    </td>
  </tr>
  <tr>
    <td width="20%"> <!--  COLONNE GAUCHE (MENU) == -->
      <a href="#">Lien</a><br>
      <a href="#">Lien</a><br>
    </td>
    <td width="55%"> <!--  COLONNE MILIEU (CONTENU) == -->
      <h4>Titre page</h4>
      Une mise-en-page avec en-tête, barre de navigation et section de
```

```
      ...
    </td>
    <td width="25%"></td>
  </tr>
  <tr>  <!--  SECTION PIED-DE-PAGE == -->
    <td colspan="3">Copyright ©</td>
  </tr>
</table>
```

Globalement, cette approche fonctionne… mais c'est très verbeux, et un peu de la bidouille.
Les balises HTML des différentes cellules se ressemblent toutes. Difficile de s'y repérer.
Dur aussi de faire évoluer la mise en page (par exemple agrandir ou diminuer la taille des colonnes, ou ajouter des sous-cases.

Faire mieux !
— Structurer la page en 6 **sections**, par exemple, avec des balises `<div>`
— Positionner le contenu de chaque section avec des règles CSS.
— Permet de faire évoluer la mise en page, par exemple sous forme linéaire et non plus tabulaire, etc.

On peut donc donner une structuration sémantique aux sections :
— en-tête
— menu
— cœur du texte,
— …
Le moteur de rendu doit alors être programmé de façon précise pour positionner les sections :
— menu : à gauche
— cœur du texte : 60 % de la largeur
— etc.
C'est le rôle de CSS de faire se positionnement, sans polluer le texte du source HTML avec toutes ces directives.
C'est beaucoup plus évolutif, et adaptable (rendu sur petit écran, etc.)

10.3.2 Programmer en HTML et CSS

Structure archétypale d'une page Source : `http://webstyleguide.com/wsg3/6-page-structure/3-site-design.html` (versions suivante : `https://webstyleguide.com/6-page-structure.html`

Certains éléments se répètent sur tout le site
Seul le contenu de la partie centrale est réellement spécifique à chaque page
Nécessite de factoriser le code, les éléments de construction des pages, etc.
)

Exemples : *CSS Zen Garden* `http://www.csszengarden.com/tr/francais/`

Même page HTML (voir sans feuille de style dans navigateur et consulter source HTML), mais différentes feuilles de style.
C'est tout l'intérêt du CSS : changer le look d'un site / application sans avoir à retoucher le code HTML ou les programmes, mais uniquement le CSS.

Design Outils de *Mockup* :
— Papier + crayon + gomme
— Outils (penpot)
— HTML + CSS (+ *templates*)

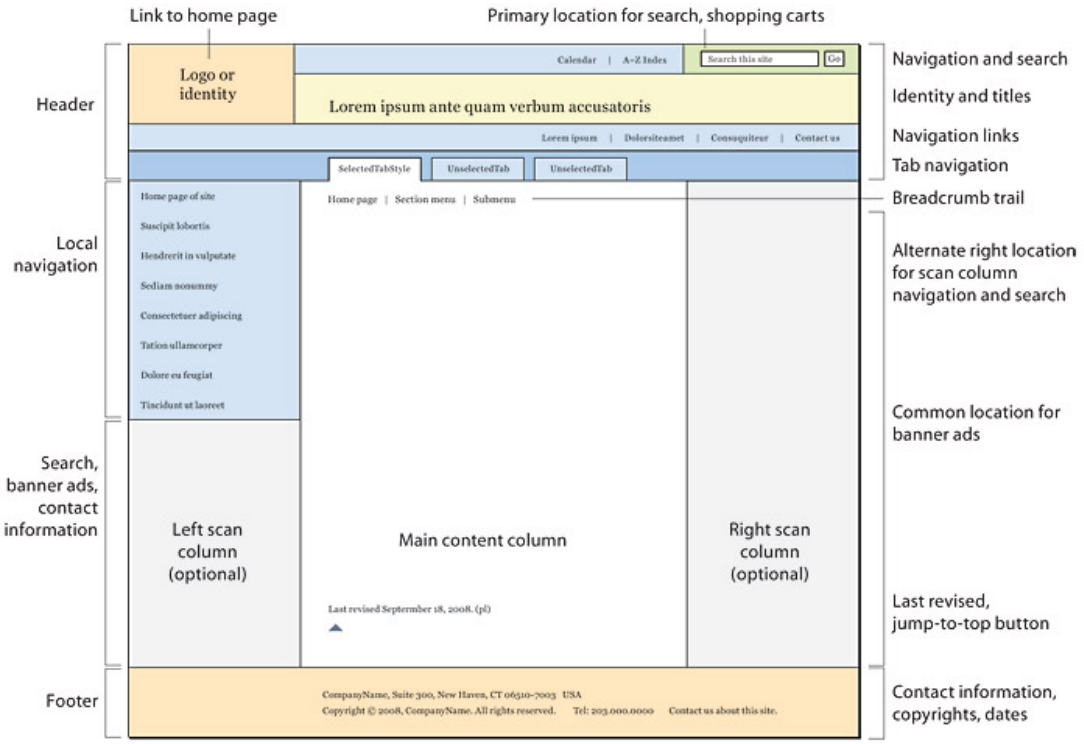


Figure 14 – Structure classique de mise en forme d'une page Web

En français : maquettage.
Métier à part entière : *Web design*, avec des agences spécialisées

Guides de style
— Chercher « *web interface style guide* » sur votre moteur de recherche préféré
— Faire comme sur les applis mobiles :
  — Android : *Material design* : `https://developer.android.com/design/index.html`
  — Apple : *Designing for iOS* : `https://developer.apple.com/design/human-interface-guidelines/designing-for-ios`

Des goûts et des couleurs ?
Attention à respecter un certain nombre de règles pour ne pas dérouter les utilisateurs. Mieux vaut s'inspirer de ce qu'ils connaissent.
Dans cette séquence, nous n'allons pas approfondir les différents principes d'ergonomie et les variantes, mais vous pourrez approfondir en travail personnel.

Inspiration / prêt à l'emploi
— `https://www.awwwards.com/` : récompenses pour le *Web design*
— `http://www.webdesign-inspiration.com/` : sélection de *designs*
Marché :
— par ex. : `https://themeforest.net/` : thèmes prêts à l'emploi
— *plein d'autres*

10.3.3 Caractéristiques de HTML

HTML 5
— Tout (ou presque) est possible en terme d'interface d'applications.
— Y compris l'accès à des zones de graphisme libre et l'intégration des périphériques du téléphone/ordinateur

Exemples :
- ~~http://trumpdonald.org/~~ `https://funkypotato.com/gamez/trump-donald/index.html` (HTML5 « natif » ?)
- `http://funhtml5games.com/?play=angrybirds` (portage de GWT -> HTML5)
- `https://princejs.com/`

Les bons artistes copient, les grands artistes volent Avantage d'HTML : le code source HTML dispo.
Vive le copier-coller :-)

Citation attribuée à Pablo Picasso
C'est ce qui a assuré le succès de HTML.

Contenu adaptatif
— *Responsive design* : prend en compte la taille de l'écran automatiquement
— p. ex. : avec Bootstrap (voir plus loin)

Principe : masquer ou faire varier les propriétés des éléments de la page pour visualiser plus ou moins de détails.

Interactions dynamiques
— Javascript
— Interactions asynchrones

Javascript va pouvoir faire évoluer la représentation du document obtenue dans la page Web du navigateur, sans avoir à refaire une requête auprès du serveur.
Exemple : naviguer dans différents « onglets » : les onglets sont tous obtenus lors de la même requête HTTP et contenus dans différentes sous-sections du même document HTML. Mais un seul onglet est visible dans une zone de la page à un instant donné.
À l'inverse, il est possible de ne charger qu'un document maître partiel, qui est affiché initialement, et peupler ensuite le contenu des sous-rubriques de ce document en tâche de fond ou à la demande, en complétant avec le résultat d'autres requêtes.
C'est le principe d'AJAX (cf. séquence ultérieure).

Interactions avec ordiphone mobile
— Application allant plus loin que l'affichage et la saisie de texte : accès à toutes les fonctions du terminal mobile depuis le navigateur Web
Roadmap of Web Applications on Mobile du W3C (September 2020)

Applications à base de technologies Web, mais utilisant les dispositifs autres qu'écran, clavier, souris et haut-parleur :
- micro
- webcam
- gyroscopes
- GPS
- multi-touch
- etc.

10.4 Principes de CSS

Cascading Style Sheets (Feuilles de style de Cascade)

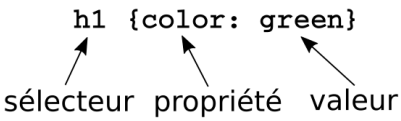
Concevoir l'habillage des pages du site / application

10.4.1 Cascade

Modèle « objet » particulier
— Langage déclaratif
— Combinaison de plusieurs feuilles avec priorités
— « Héritage » de propriétés des parents
— Factorisation de motifs :
DRY – *don't repeat yourself*

10.4.2 Langage à base de règles

— **Si** motif trouvé (sélecteur)
— **alors** valeur donnée à attribut/propriété de mise en forme



10.4.3 Exemple

```css
h1 {
  font-size: 60px;
  text-align: center;
}

p,
li {
  font-size: 16px;
  line-height: 2;
  letter-spacing: 1px;
}
```

10.4.4 Niveaux croissants de proximité

Emplacement du code CSS :

1. pas de style : affichage navigateur par défaut
2. fichier CSS externe :

```html
<link rel="stylesheet" href="external.css"/>
```

3. bloc style interne :

```html
<style type="text/css">
.underline { text-decoration: underline; }
</style>
```

4. dans élément :

```html
<div class="column" style="float:left; width: 50%">
```

Les illustrations de ces sections sont tirées de l'article ~~http://cssway.thebigerns.com/special/master-item-styles/~~ « Freeway Pro 6, Master Item styles and the Cascade Order » de Ernie Simpson
Ici, on entend par proximité, celle entre une définition d'une règle de rendu CSS, et un élément concerné, de l'arbre HTML.
Plus la spécification d'une règle CSS est proche, plus elle annule et remplace des règles plus éloignées qui s'appliqueraient à ce même élément.

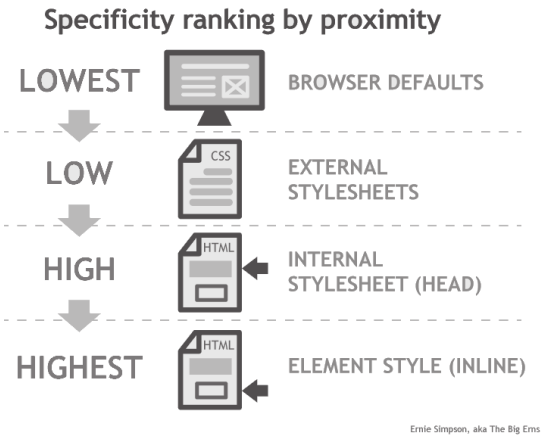


Figure 15 – Specificity ranking by proximity

10.4.5 Spécificité selon la proximité

Quelles règles s'appliquent, si plusieurs définies ?

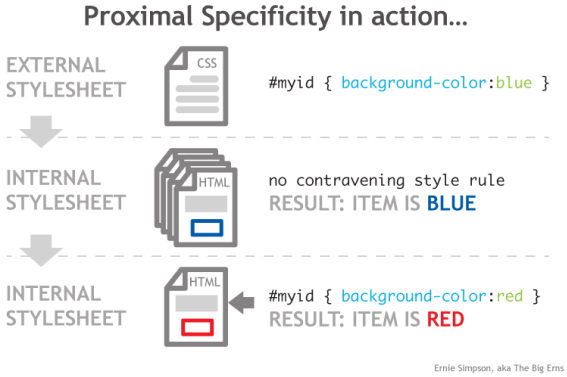


Figure 16 – Proximal specificity in action…

Dans cet exemple, une page spécialise, localement, un comportement défini pour l'ensemble du site.
On peut définir la couleur de fond bleu pour l'élement d'identifiant `myid`, via une feuille de style `external.css` contenant :

```
#myid { background-color: blue; }
```

Elle sera incluse dans le source HTML des pages du site :

```
<head>
  <link href="external.css" rel="stylesheet" type="text/css">
</head>
```

Dans toutes les pages qui contiennent cette feuille de style, un élément d'identifiant `myid` sera sur fond bleu.
Mais via une règle placée dans une page particulière, on peut surcharger cette règle, pour utiliser plutôt du rouge :

```
<head>
 <style type="text/css">
   #myid { background-color: red; }
 </style>
```

Encore plus près, on pourrait faire :

```
<p style="background-color: fuchsia;">Aren't style sheets wonderful?</p>
```

10.4.6 Spécificité du sélecteur

Ordre croissant de spécificité du sélecteur :

1. par balise : `div {style:valeur}`
2. par classe : `.myclass {style:valeur}`
3. par identifiant : `#myid {style:valeur}`

Figure 17 – Specificity ranking by selector type

Un sélecteur générique, comme le nom d'une balise (`<div>`) couvre beaucoup de contenu, et est donc moins spécifique qu'une classe (`.myclass`), ou bien qu'un identifiant unique (`#myid`).
Tout le jeu consistera à définir des règles générales, par exemple au niveau des classes, et à autoriser parfois des spécificités, pour certains identifiants.

10.4.7 Outils du développeur

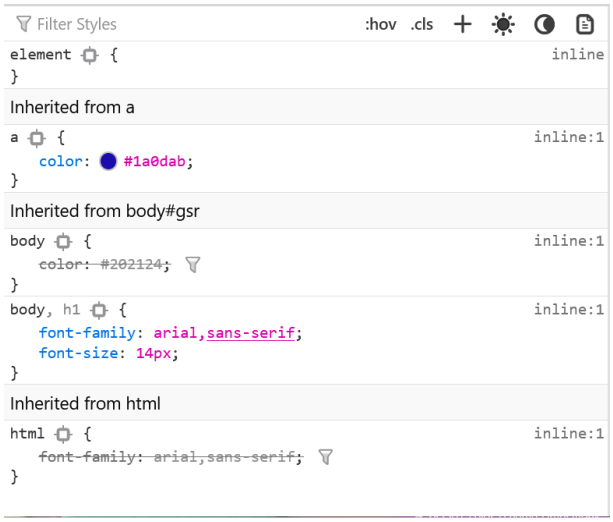


Figure 18 – Outil du développeur : règles CSS, dans Firefox

Source : Examiner et modifier le CSS sur MDN

10.4.8 Conception de CSS

— Pas toujours simple
— Abstraction
— *Frameworks* CSS (et Javascript)
    — ex. : Bootstrap

Couplage blocs *templates* Twig et CSS

Ne pas partir de rien et réutiliser des *frameworks*.
Les détails ne seront pas présentés en cours : travail hors présentiel.

10.5 Conception blocs Twig et CSS - Bootstrap

Examinons finalement des éléments méthodologiques et des outils qu'on va mettre en œuvre dans les projets.

10.5.1 Conception des gabarits et CSS

On peut définir des règles de structuration des pages HTML, au niveau des gabarits, en parallèle de la définition des règles de mise en forme.
Voici une méthodologie assez simple pour ce faire.

Concevoir les gabarits des pages Conception parallèle :
— des blocs des gabarits HTML
— des sections `<div>` qu'ils contiennent (identifiants ou classes)
— des mises en forme

Même convention de nommage

Modèle de page souhaité

```
<html>
  <head>
  </head>
  <body>
    NAVIGATION

    MENU (COLONNE DE GAUCHE)

    CONTENU PRINCIPAL

    BAS DE PAGE
  </body>
</html>
```

Structure sémantique

```
<body>
  <div id="navigation">
    (NAVIGATION)
  </div>
  <div id="menu">
    (MENU RUBRIQUES)
  </div>
  <div id="main">
    (CONTENU PRINCIPAL)
  </div>
  <div id="footer">
    (BAS DE PAGE)
  </div>
</body>
```

Pour chaque grande zone de la page qu'on souhaite mettre en forme, on va structurer le contenu HTML à l'intérieur d'une balise `<div>` ayant une classe particulière.
On va définir en parallèle les règles de mise en forme CSS de ces différentes zones, l'identifiant faisant le lien. On pourrait aussi utiliser des classes.

```
{% block body %}
<body>
   <div id="navigation">
   {% block navigation %}
   {% endblock %} {# navigation #}
   </div>
   <div id="menu">
   {% block menu %}
   {% endblock %} {# menu #}
   </div>
   <div id="main">
   {% block main %}
        <h1>{{ Message }}</h1>
   {% endblock %} {# main #}
   </div>
   <div id="footer">
   {% block footer %}
   {% endblock %} {# footer #}
   </div>
</body>
{% endblock %} {# body #}
```

Figure 19 – Structure de blocs Twig du gabarit de base

Gabarit Twig

Dans les gabarits, on va définir des blocks surchargeables, correspondants au contenu des divisions, et portant les mêmes noms que les classes précédamment définies.
On a ainsi un nom de zone/block/classe unique pour faire le lien entre les différents éléments.

Standardisation / Spécialisation Spécialisation dans une sous-section ou une page de certains éléments de structure ou de présentation
— héritage des gabarits : redéfinition du contenu de blocs

```
{% block main %}
...
{% endblock %}
```

— cascade des feuilles de style CSS : redéfinition de la mise en forme

```
#main { background-color: lightblue; }
```

10.5.2 CSS avec Bootstrap

Bootstrap est un *framework* permettant de mettre en place un jeu de feuilles de styles CSS, en s'appuyant sur de mécanismes réactifs/adaptatifs.

Utilisation d'un *framework* de présentation CSS
— Standardisation du look
— Adaptatif
— Grille pour le positionnement graphique
— Intégration avec Twig / Symfony

Bootstrap

Framework CSS (+ JS)
`http://getbootstrap.com/`

Système de grille Mise en page basée sur une grille de 12 colonnes

Figure 20 – Grille de mise en page dans Bootstrap

Exemples
— exemples, dans la documentation Bootstrap
— exemples sur *Start Bootstrap*

Thèmes
— Thèmes sur *Start Bootstrap*
— Feuille de syle additionnelle

Take away

— Structure != présentation
— Rôle de CSS
— Principe de surcharge des feuilles (cascade)
— Accessibilité
— Bootstrap

Annexes

Sur la qualité des interfaces Web, consulter le livre « *Qualité Web - La référence des professionnels* » - 2e édition - Eyrolles : `http://qualite-web-lelivre.com/` (disponible à la médiathèque)
Sur le sujet de l'accessibilité, vous pouvez également consulter la vidéo Voir autrement : l'ordinateur accessible

Postface

Crédits illustrations et vidéos

— Illustration Desktop Macintosh source Wikipedia
— Vidéo `https://www.youtube.com/watch?v=gc9NpYrbZgQ` par `UserExperiencesWorks`
— vidéo : Comment les aveugles utilisent-ils internet ? (Le nouvel obs)
— Image « HTML + CSS » empruntée au « CSS Tutorial » de Avinash Malhotra : `http://tutorial.techaltum.com/css.html`
— Diagrammes « *Specificity ranking by proximity* », « *Proximal specificity in action…* » et « *Specificity ranking by selector type* » empruntés à par Ernie Simpson, *aka The Big Erns* : ~~`http://cssway.thebigerns.com/special/master-item-styles/`~~ (site disparu)
— Diagramme de grille de mise en page de Bootstrap empruntée à Bootstrap de Twitter : un kit CSS et plus ! par Maurice Chavelli
— Illustration « Enjoy Websurfing », supposémment de Bruce Sterling

11 Séq. 7 : Contrôleurs, interactions CRUD, formulaires

Objectifs de cette séquence

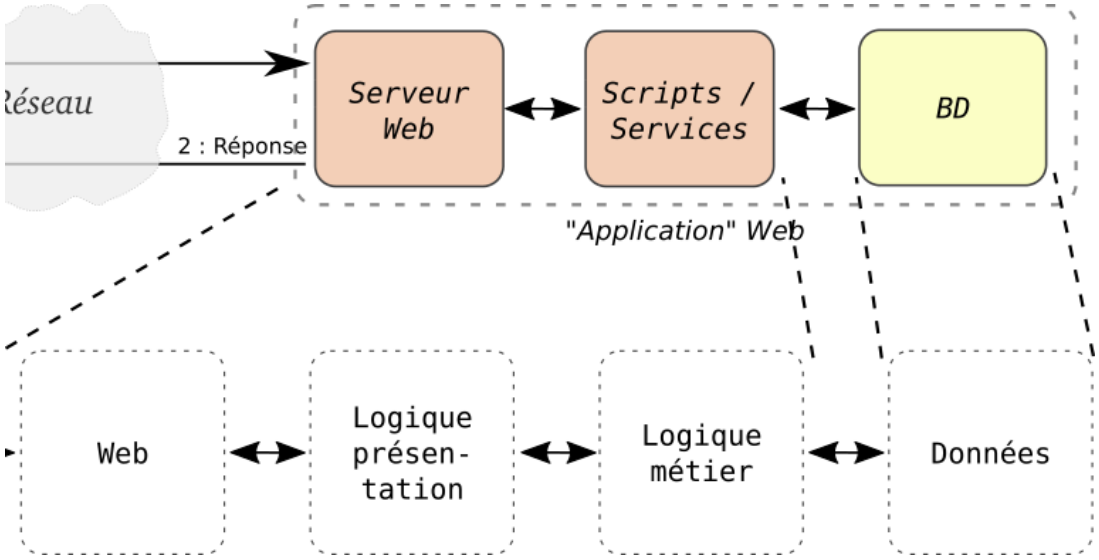
Cette séquence aborde les mécanismes permettant de rendre **interactive** une **application basée sur HTTP**.
On explique la façon dont les concepteurs de HTTP on énoncé les principes ReST qui permettent facilement de mettre en œuvre des interactions simples pour des API de type CRUD (*Create Retrieve Update Delete*).
On explique également comment les applications Web on intégré des mécanismes permettant d'offrir une expérience utilisateur riche, via les formulaires HTML pour la soumission de données.

11.1 Rôle des contrôleurs

11.1.1 Rappel architecture et MVC

Cette section revient sur le patron de conception MVC, déja présenté à la séquence précédente.

Rappel : Architecture multi-couches



Vous connaissez déjà ce diagramme, où l'on décrit l'architecture de l'application en distinguant un ensemble de couches.
Nous arrivons maintenant au cœur de la logique de présentation, par rapport à la dimension interactive d'une application Web.

Rappel : Patron MVC

Nous avons vu le modèle, et les vues.
Reste à compléter la description du rôle du contrôleur, ce qu'on voit maintenant dans le contexte d'une application Web.

Modèle et Vue

Concrètement, les **Vues** sont les représentations des **états** de l'application.
À chaque instant, un utilisateur de l'application voit une vue, ce qui lui permet d'agir sur un état particulier de l'application (donc des données du modèle).

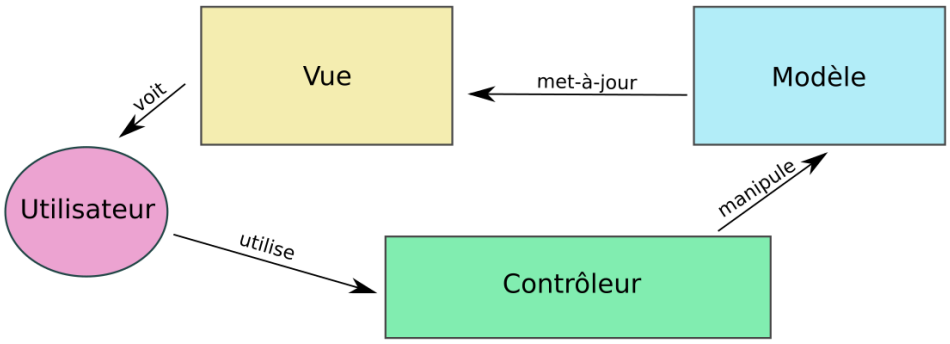


Figure 21 – Patron de conception MVC

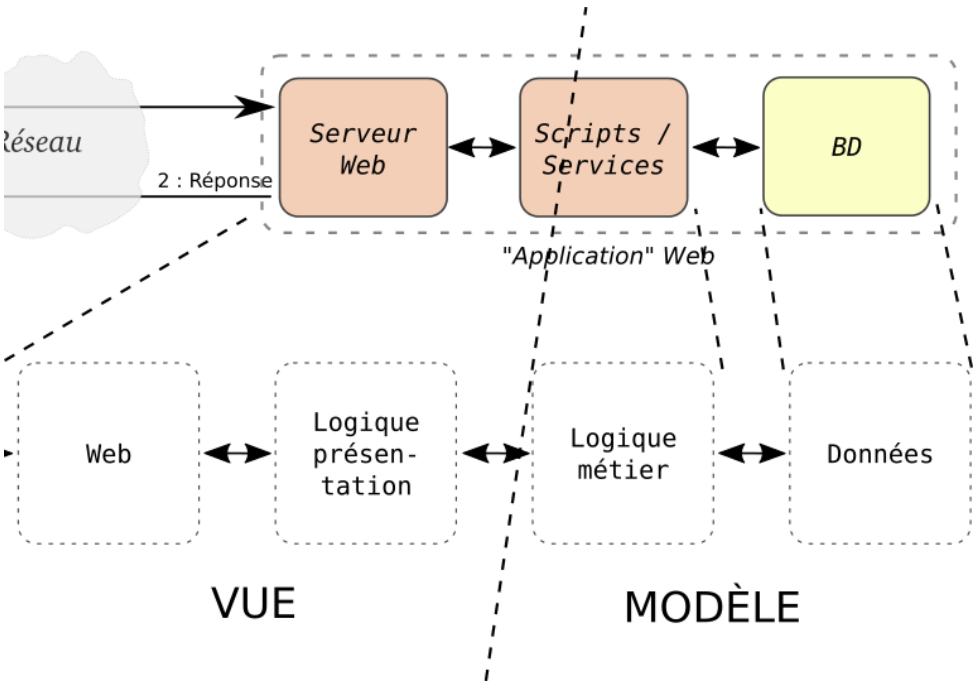


Figure 22 – Architecture : Modèle + Vue

Pilote des interactions : le Contrôleur
— **Contrôleur** : coordonne les transitions entre vues, et les entrées de l'utilisateur, en fonction de l'état du modèle
— Vues (pages)
— Modèle
Objectif : programmer les vues mais aussi le contrôleur, au-dessus des mécanismes de HTML et HTTP, pour offrir une bonne expérience utilisateur.

Le contrôleur est l'élément nouveau de notre analyse, celui qui doit gérer les événements issues des requêtes HTTP et dispatcher le travail vers les modules de notre application. C'est le point d'entrée spécifique qui contrôle le fonctionnement interactif de l'application.

11.1.2 Modèle de conception : États - Transitions

Nous présentons ici le modèle conceptuel HATEOAS issu de l'approche ReST qui sous-tend toute l'interaction dans les applications Web classiques.

Ensemble de vues / écrans / pages
— Naviguer dans une arborescence de pages / **vues** / écrans / dialogues
— Trouver de l'information au sein d'une page
— *Widgets* pour interaction : composants d'interface graphique (boutons, listes, etc.)

Un état de l'application Une page / vue, représente l'état courant

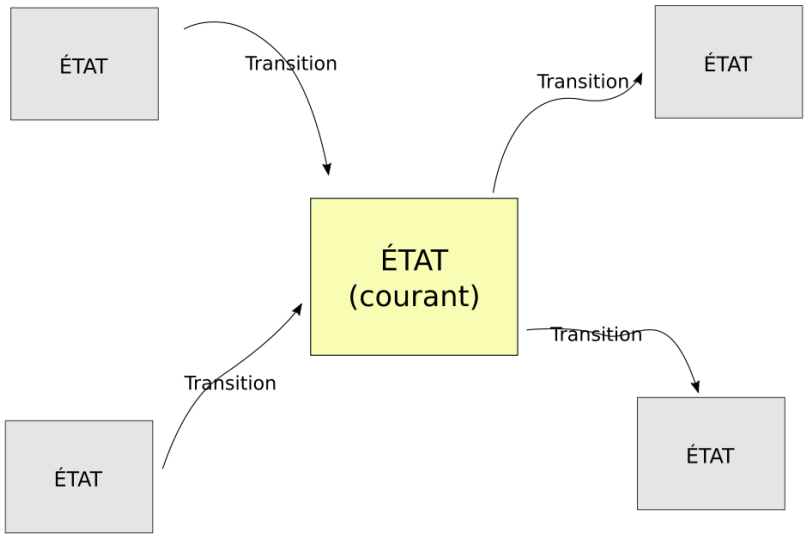


Figure 23 – Transitions sur une page

***Hypermedia As The Engine Of Application State* (HATEOAS)**
— Une application = diagramme d'**états fini**
— Transitions = hyperliens sur lesquels on clique (et autres *widgets*)

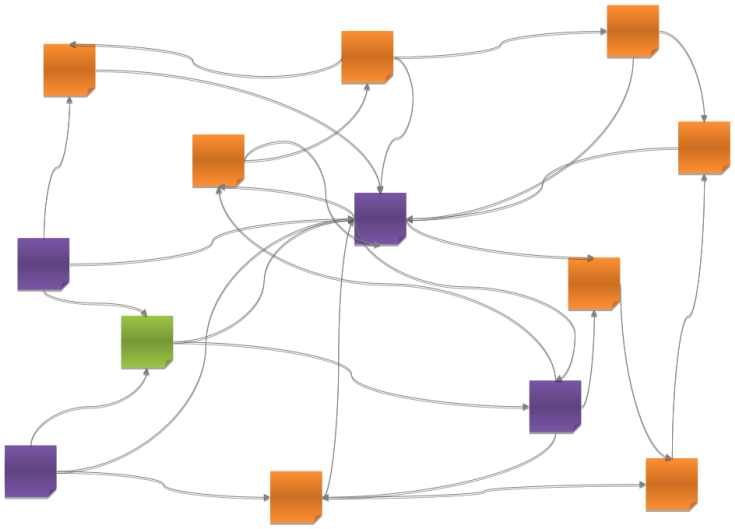


Figure 24 – Diagramme états fini

Un diagramme d'états fini définit différents états possible, mais à tout instant, un seul état actif pour une session de consultation de l'application : la *page courante* affichée dans le navigateur.
HATEOAS est l'un des principes du REST (*REpresentational State Transfer*), le fondement de l'utilisation de HTTP pour faire fonctionner des applications.
Prononcer *hay-dee-ous*

Exemple Application Web Point départ :

Figure 25 – Premier *Tweet*

(`https://twitter.com/jack/status/20`)

Diagramme états Web app Twitter 1 transition en entrée / 32 transitions en sortie

Source : API Design : Honing in on HATEOAS par Brian Mulloy, APIGEE
La transition en entrée, est l'accès à la ressource `https://twitter.com/jack/status/20` dans le navigateur. Les transitions de sortie sont les différentes pages de l'application accessibles depuis celle-là.

Interactions avec utilisateur
— Affichage de pages (lecture sur des ressources)
— Invocation de méthodes sur des ressources dédiées (y compris modification : CRUD)
— Invocation d'APIs pour interactions fines (lecture / écriture)

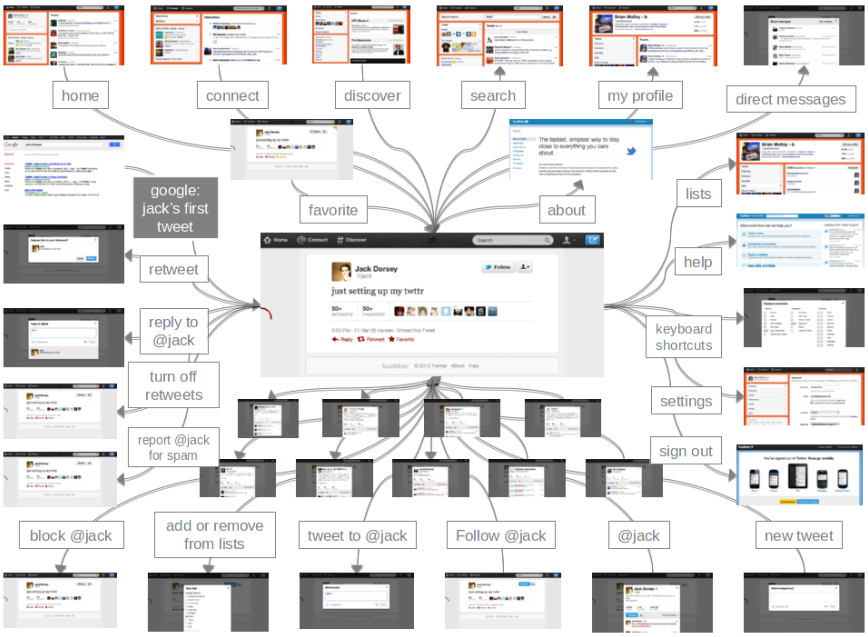


Figure 26 – Diagramme états Web app *Twitter*

C'est les choix de nommage des chemins d'accès aux ressources qui détermine la cohérence de l'application par rapport au schéma d'état souhaité.
Aucune contrainte particulière liée à HTTP.
Les *frameworks* de développement peuvent aider à mettre en œuvre des bonnes pratiques (REST).
Le client est universel (le navigateur), stupide, c'est à dire sans connaissance du modèle de transitions d'états de l'application, et n'a pas besoin d'être programmé par rapport à chaque application qui va être utilisée, pour savoir à quel endroit la contacter. Il se contente de suivre la trace des liens que lui propose le serveur, via les cibles `href` des balises `<a>`, par exemple (*follow your nose*).

***Framework* Web MVC**
— Modèle
— Vue
— **Contrôleur**
Le contrôleur gère les transitions *HATEOAS*

On peut maintenant mieux comprendre le rôle du **Contrôleur**, qui est le « cerveau » des interactions Web avec l'application, celui qui va piloter les changements d'état dans le diagramme d'états fini de notre application (HATEOAS vu plus haut), en réponse aux requêtes HTTP transmises par les clients/utilisateurs.

11.1.3 Vers une plate-forme Web universelle

Les technologies Web deviennent progressivement une plate-forme universelle pour déployer des applications, de façon accessible, portable, interactive.
Le déploiement est facilité : rien à installer sur les clients. Comme l'accès à Internet se généralise, on déploie à un seul endroit : côté serveur, et les clients existent déjà : navigateurs Web.
L'apparition de Javascript et d'HTML5 notamment, rendent obsolètes d'autres technologies comme Flash ou les applets Java par exemple.

« Services Web »
— Utiliser HTTP pour faire interagir différents **programmes**, sans navigateur Web, sans humain dans la boucle :
  — programme client
  — programme serveur
— Services Web, **APIs** (*Application Programming Interfaces*)
— Problème : HTTP est vraiment de très bas niveau, donc il faut ajouter pas mal de standardisation par-dessus, ou que les concepteurs des programmes des deux côtés se concertent.

HTTP peut permettre de mettre en relation des programmes clients et serveur, notamment pour faire fonctionner des interfaces programmatiques (API) sur Internet.
On peut utiliser la sémantique assez limitée de HTTP, mais qui est souvent un peu limitée pour des applications sophistiquées.
Dans tous les cas, il faut bien comprendre HTTP pour être capable de tester / debugger. Sinon on ne sait pas déterminer si les problèmes viennent du navigateur ou de l'application côté serveur (ou des deux).

Applications pour l'utilisateur humain
— Utiliser les technologies du Web pour faire fonctionner des applications affichées dans un navigateur.
— Mettre en œuvre le principe HATEOS : expérience utilisateur
— Mécanismes de **saisie de données** riches (widgets affichés par le navigateur), envoyées au serveur
— Mécanisme de traitement des données reçues, côté serveur Web

Deuxième type d'applications, des applications « pour les humains » interagissant avec un client navigateur Web.
Les pages doivent alors intégrer des mécanismes permettant d'interagir avec le serveur Web qui gère la production des pages, pour gérer des transitions dans le graphe HATEOS, et permettre d'obtenir une ergonomie équivalent à celle des applications GUI du « desktop » qui existaient précédemment.
On ajoute à HTML et HTTP ce qui est nécessaire pour supporter ces dynamiques d'interactions plus riche qu'auparavant pour une simple consultation de pages et de liens : formulaires, etc.

11.1.4 Sémantique des interactions avec un serveur HTTP

Les concepteurs d'HTTP ont intégré dans le protocole des fonctions permettant d'interagir avec les ressources d'un serveur accessibles sur le Web, avec une sémantique d'opérations couvrant aussi bien les modifications que la consultation.

Principes REST
— *REpresentational State Transfer* : passer de documents hypertextes à des applications Web
— dans la Thèse de Roy Fielding en 2000 :
— **HTTP 1.1** pour des applications (APIs)
  — Verbes/méthodes pour une sémantique plus riche d'opérations : `GET`, `POST` **+** `PUT` **ou** `PATCH`**,** `DELETE`**,** `OPTIONS`
  — Codes de retour HTTP
— Pas de session ?
— ***The six constraints of the architectural style of REST*** (voir dessous)

Les méthodes GET ne sont plus les seules disponibles. Si on souhaite interagir avec les ressources d'un serveur pour les modifier, on utilise les méthodes ad-hoc. GET ne sert plus qu'à la consultation, dans ce cas (en principe).
Remarquez que *ReST* date de 2000, mais ce n'est qu'environ 10 ans plus tard que ces concepts ont été revisités pour devenir les mots-clés du *hype* informatique. Le temps pour les travaux de recherche de diffuser dans l'industrie ?

6 contraintes architecture REST

1. Client-serveur
2. Serveur sans mémoire état du client (*stateless*)
3. Cache (client)
4. Système à plusieurs couches
5. Code sur le client *on-demand* (optionnel)
6. Contrainte d'interface uniforme
   (a) **Identifiants de ressources**,
   (b) Représentations des ressources,
   (c) **Messages auto-descriptifs**,
   (d) **HATEOAS** : *Hypermedia As The Engine Of Application State*

Toutes ces contraintes ne sont pas complètement évidentes de prime abord, et une littérature fournie existe sur le sujet.
Nous aurons abordé une bonne partie de ces contraintes au fil des séquences.

11.2 Soumission de données à une application

Cette section présente la façon dont les mécanismes de HTTP, que nous avons déjà croisés, vont nous servir à mettre en œuvre la soumission de données dans les applications Web relativement simples.
On explicite aussi pourquoi c'est le serveur qui va construire l'interface de l'application qui saura générer les bonnes requêtes de soumissions de données, et qu'il peut donc ainsi la déployer vers les utilisateurs.

11.2.1 *Read* + *Write* !

Transmission de données au serveur HTTP (par le client).
Pour **quoi** faire ?
CRUD :
— **création** (***C*** *reate*)
— consultation (***R*** *ead*)
— **modification** (***U*** *pdate*)
— **suppression** (***D*** *elete*)

On suppose qu'un serveur Web ressemble un peu à un gestionnaire de bases de données, ou un serveur de ressources, dont le point d'entrée HTTP autorise les clients à les modifier.
On a déjà vu la consultation précédemment, qui emploie les requêtes GET pour accéder à des représentations des ressources stockées sur le serveur.
On peut maintenant définir d'autres méthodes pour la **modification**.
On élargit la **sémantique** des opérations possibles sur HTTP.
Voyons comment les données peuvent être transmises via HTTP.

11.2.2 Méthodes HTTP pour envoi de données à l'application

— Différentes solutions :
  — `GET`
    — arguments dans URL
      `?arg1=val1&arg2=val2...`
    — en-têtes
  — `POST`
    — arguments dans URL
    — en-têtes
    — **données de contenu de requête**

Historiquement, GET a pu être utilisée pour transmettre des données, mais assez vite, POST apparaît dans les applications Web, pour expliciter qu'il s'agit d'une méthode dont le rôle est de transmettre des données au serveur HTTP. D'autres méthodes suivront.

11.2.3 Exemple : ajout d'une tâche

On voudrait une page dans l'application pour ajouter une tâche, du genre :

Imaginons qu'on a une application tournant sur un ordinateur de l'utilisateur, qui lui affiche une interface GUI, et qui déclenche une interaction avec le serveur HTTP, en réponse aux actions de l'utilisateur.

11.2.4 Exemple : ce que le programme comprend

Le programme Symfony derrière le serveur HTTP s'attend à la transmission de données via :
— requête `POST` sur ressource `/todo/new`
— variables dans les données du corps :

1. titre de la tâche (texte, obligatoire) : `todo[title]`
2. statut de la tâche (booléen, optionnel) : `todo[completed]`

Par exemple, avec un en-tête `Content-Type: application/x-www-form-urlencoded` :

```
todo[completed]: 1
todo[title]: Apprendre+le+CSS
```

Les développeurs d'une application côté serveur ont imaginé une façon pour les clients de transmettre des données via HTTP.
Le client HTTP doit encoder le contenu des valeurs des variables pour que ça transite sur du texte simple via HTTP 1.1. D'où une spécification dans les entêtes de la requête POST du `Content-Type` utilisé, et l'encodage correspondant (ici, remplacement des espaces par des `+`).
La sémantique de l'opération via la méthode POST sur un chemin `/todo/new` : **création** d'une nouvelle ressource dans la collection `todo`.
Le serveur s'attend à ce que les noms des variables soient `todo[completed]` et `todo[title]`… mais ça aurait tout aussi bien pu être `todo_completed` et `todo_title`, par exemple… comment savoir ?
Vous le savez si je vous le dis, mais rien n'oblige à utiliser ce nommage. C'est un choix du développeur de l'application, qui dépend aussi des technologies utilisées.
Pourquoi doit-on utiliser POST et pas GET ? C'est juste une bonne pratique REST pour une interface CRUD via HTTP.
Il y a beaucoup de façon de faire possibles via HTTP pour faire la même chose : il faut donc qu'on sache comment fonctionne cette application du côté serveur, pour pouvoir interagir avec elle.
Rien n'oblige les développeurs à fournir la documentation de l'application !

11.2.5 Utiliser un « générateur de requêtes » ?

L'utilisateur humain ne construit pas des requêtes HTTP (en général) : il faut une interface dans un programme.
Comment mettre cette interface à disposition de l'utilisateur : **déploiement de la partie cliente** de l'application ?

Les utilisateurs avancés peuvent utiliser cURL… mais en général les utilisateurs « normaux » préfèrent un GUI, et ne pas gérer les détails d'HTTP.
Comment **déployer** auprès des utilisateurs un programme qui sait comment générer la requête POST, et affiche un GUI compatible avec tous les systèmes d'exploitation ?
Des programmes existent, comme Poster, qui est un équivalent de cURL, qui offre une interface dans le navigateur pour la génération de requêtes CRUD conformes à REST… mais ce n'est pas exactement l'ergonomie attendue par l'utilisateur moyen.
Et encore faut-il **lire la documentaton** pour comprendre la **sémantique** et la syntaxe des interactions attendue par le serveur !
Comment faire fonctionner ce type d'applications client-serveur de façon interopérable, à l'échelle du Web ?

11.2.6 Installer un interface utilisateur, pour chaque application ?

— Client HTTP à déployer pour chaque application ?
  Exemple : applications mobiles natives
  Problème de déploiement !
— Client HTTP **universel** (donc « stupide ») ?
  Exemple : Navigateur FireFox
  Comment rendre ce navigateur capable de parler **comme il faut** au serveur ?

Le navigateur est nécessairement stupide, du point de vue de la sémantique fonctionnelle des applications, si on veut qu'il soit universel. Il se contente de suivre des liens, et de réagir à ce que le serveur HTTP lui propose.
Le serveur doit donc préparer une interface « prête à l'emploi » pour ce client Web stupide. C'est à cela que vont servir les formulaires HTML !

11.2.7 Ajout au navigateur de mécanismes d'interface génériques

Fonctionnalités **génériques** dans navigateur Web, de génération d'**interfaces de soumission de données** :

1. Récupère l'interface graphique propre à chaque application (HTML), **transmise par le serveur**
2. Affiche l'interface à l'utilisateur
3. Génère les **bonnes requêtes** qui satisferont les besoins du serveur

C'est le programme sur le serveur qui construit une interface (HTML) spécifique à son application, pour que le client (stupide) puisse l'afficher aux humains.
Dans le monde Web, on n'a pas besoin de reprogrammer un client spécifique pour chaque application et de le re-déployer.
Auparavant, des protocoles de communication client-serveur existaient, mais il fallait développer et déployer des clients spécifiques.
C'est ce qui explique le succès de la plate-forme Web pour le déploiement des applications.

11.2.8 Envoi de l'interface par le serveur

1. Le serveur peut envoyer aux clients une page HTML contenant l'interface particulière qui sait générer la bonne requête.
2. La page HTML contient :
   — un **formulaire** qui est affichable (incluant tous ses widgets, pour chaque variable)
   — un **« réflexe »** qui générera une requête bien formée (`POST` ?) pour transmettre les données selon les conventions du serveur

Le développeur implémente en parallèle :
— les gestionnaires de soumission des données traitant la requête POST
— la page HTML contenant le formulaire qui génère la requête
En théorie la partie envoyée au client, et celle tournant sur le serveur sont cohérente, si le programmeur a bien travaillé.
Les clients et le serveur peuvent se comprendre, et une évolution du fonctionnement de soumission des données est redéployée de façon instantannée.
Ça passe à l'échelle du Web, et ça marche.

11.3 Gestion des formulaires HTML

Cette section présente le fonctionnement des formulaires Web dans HTML, et les propriétés des requêtes HTTP associées.
On détaille ensuite la façon de les gérer en PHP et avec Symfony

11.3.1 Séquence d'interaction

Examinons la séquence d'interactions qui permet de faire fonctionner des interfaces de formulaires HTML avec le dialogue HTTP entre navigateur Web et serveur.

Dialogue entre serveur et client

1. Client HTTP « stupide » demande au serveur à **récupérer une interface HTML** pour avoir accès à une fonctionnalité de l'application
2. Le serveur lui prépare une « jolie » page HTML contenant tous les widgets ad-hoc, et qui est **prête à générer l'invocation de la méthode REST** ad-hoc
3. Le client construit une page Web utilisable par un humain : formulaires, boutons, etc. Il fait une confiance « aveugle » au serveur, concernant l'utilisabilité

1. L'humain saisit les données requises et valide leur transmission, sans imaginer la structure et le format réel des données qui devront transiter vers le serveur via HTTP
2. Le **client transmet « stupidement » ces données** au serveur avec la méthode HTTP bien formée (en principe)
3. Le serveur réagit à cette soumission de données et confirme au client si c'est bon pour lui

L'intelligence est du côté du serveur.
La construction de la page HTML contenant les formulaires doit être conçue de façon étroite avec le gestionnaire des requêtes REST… sinon bug.
C'est l'enjeu du développement des interfaces qui nous attend maintenant : cohérence des étapes 2 et 5.

Séquence HTML + HTTP

1. Requête (méthode `GET`) du navigateur sur URL
2. Réponse avec un document HTML contenant `<form>...</form>`
3. Affichage du formulaire dans le navigateur
4. Saisie de données dans les champs du formulaire
5. Clic sur le bouton de soumission : nouvelle requête d'envoi des données sur l'URL cible du formulaire (souvent méthode `POST`)
6. Traitement du formulaire par le serveur : code de réponse HTTP
7. …

Selon si la soumission est réussie ou non, en parallèle de du code de réponse, à l'étape 6, le serveur renvoit le formulaire (en cas d'erreur sur les données saisies), ou redirige vers une autre page (s'il est satisfait).

Traitement des données transmises

1. Le serveur reçoit les données
2. Il décide s'il les accepte
3. En cas de refus, le client devra :
   — ré-afficher les données pour demander de corriger
   — ou afficher une autre page (**redirection**)
4. En cas d'acceptation, le client devra :
   — afficher une autre page (**redirection**)

Redirection

— Évite de rejouer la requête de transmission du formulaire en cas de rechargement de la page par l'utilisateur dans son navigateur
— Le serveur envoie :
   — code de réponse `30x`
   — en-tête `Location` + URL de destination

On peut utiliser le code de réponse HTTP 303 pour la redirection (*See other*).
Attention, rend difficile le debug : les messages d'erreur du traitement de la requête reçue initialement ont tendance à disparaître de l'affichage du navigateur.
On verra que la barre d'outils de mise au point de Symfony ajoute des outils intéressant pour ce cas.
En attendant une partie des informations utiles peuvent être vues dans l'inspecteur réseau du navigateur : contenu des requêtes et réponses, suivi des redirections, codes de retour, etc.

11.3.2 Exemple HTML + PHP

Exemple sur l'application « fil rouge » de gestion de tâches.

Examinons comment on peut mettre en œuvre la création de nouvelle tâche avec du PHP « de base » et du HTML.
Ces exemples sont illustratifs et correspondent à une façon de programmée obsolète, maintenant qu'on a un *framework* comme Symfony, mais ils illustrent certains mécanismes et surtout des points d'attention pour les développeurs Web.

1-2 : Formulaire de saisie HTML

1. Requête : `GET /todo/new`
2. Réponse : page avec formulaire HTML « `new` »

```
<!DOCTYPE html>
<html>
  <body>

    <form method="post" action="/todo/new">
      Ce qu'il faut faire :<br>
      <input name="todo[title]" required="required" type="text">
      <br>
      Déjà terminé :<br>
      <input name="todo[completed]" type="checkbox">
      <br><br>
      <input type="submit" value="Ajouter">
    </form>

  </body>
</html>
```

Le serveur transmet le formulaire de création d'une nouvelle tâche, quand on accède à la ressource `/todo/new` (GET).
Le navigateur peut afficher les deux champs de saisie et un boton de validation correspondants.
L'attribut `method` de la balise `<form>` définit explicitement la méthode à utiliser pour la soumission des données du formulaire : `POST` sur la cible `/todo/new` (la même ressource que pour l'affichage du formulaire).
On peut générer ce même genre de formulaire HTML via un programme PHP (via un gabarit Twig, par exemple).

3-4 : Gestion des contraintes de saisie du formulaire Le formulaire est affiché et l'utilisateur peut saisir des données dans les champs de saisie (ou cocher les cases, pour la boîte à cocher (*checkbox*).
Le navigateur peut effectuer certains contrôles de saisie, par exemple sur les champs en saisie obligatoire (attribut `required` du champ `<input>`).

5 : Requête transmise
— Méthode `POST` :

```
<form method="post" action="/todo/new">
  ...
```

— Requête `HTTP` soumise :

```
POST /todo/new HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 33

todo[completed]=on&todo[title]=Apprendre+le+CSS
```

6 : Exemple de code de gestion en PHP PHP naïf :

```
<html>
  <head>
    <title>Résultats du Formulaire</title>
  </head>
  <body>
   <p>Bonjour <?php
     echo $_POST[todo][title] ." ". $_POST[todo][completed];
     ?>.</p>
  </body>
</html>
```

Listing 9 : Version basique de `gestionnaire.php`

On voit ici une gestion des soumissions du `POST` très naïve, mélangeant PHP et HTML, **fortement déconseillée en production**.
Le code utilise le tableau global `$_POST` qui contient les données transmises dans le code de la requête `POST`, dans `$_POST[todo]`.
On a vu que ce n'est pas la meilleure façon de s'y prendre car on mélange présentation et traîtements.
Problème : si on renomme les noms des champs `input`, deux fichiers sont impactés, le template et le gestionnaire du POST.
En outre, pas de gestion d'erreur, par exemple pour vérifier que les champs sont bien remplis si c'est une valeur obligatoire.
On va voir qu'on peut mieux faire.

Version « deux en un »

```
<html>
  <head>
    <title>Résultats du Formulaire</title>
  </head>
  <body>
   <p>Bonjour <?php
       echo $_POST[todo][title] ." ". $_POST[todo][completed];
     ?>.</p>

      <form method="post">
        Ce qu il faut faire :<br>
        <input name="todo[title]" required="required" type="text"><br>
        Déjà terminé :<br>
        <input name="todo[completed]" type="checkbox">
        <br><br>
        <input type="submit" value="Ajouter">
      </form>
```

```
  </body>
</html>
```

Cette version intègre l'affichage du formulaire vide et la gestion des soumissions dans un seul et même script PHP, ce qui résout une partie des problèmes précédents.
Le même script est appelé sur les GET ou les POST sur `/todo/new`. La cible de la requête n'a plus besoin d'être mentionnée explicitement : plus de `action` dans le `<form>` : c'est la même ressource qui recevra la requête POST.
Le code est plus maintenable. Par exemple, en cas de renommage des identifiants des champs `input`, on ne modifie qu'un seul fichier.
Mais toujours pas de gestion d'erreur. Cela commencerait à alourdir le script si on l'ajoutait, et toujours pas de séparation présentation / gestion : mélange HTML et PHP.

Sécurité Attention : injection de code malveillant !
— Que se passe-t-il si `$_POST[todo]` contient du code HTML, qui déclenche, par exemple, l'exécution d'instructions Javascript, et qu'on fait `echo $_POST[...]` ?

On verra plus tard comment se protéger de ce genre de problèmes.

11.4 Programmation avec Symfony

Étudions maintenant les composants Symfony permettant de mettre en œuvre les contrôleurs et la gestion des formulaires de soumission de données.

11.4.1 MVC Symfony complet

Modèle des données : Doctrine
— Classes PHP
— Attributs Doctrine : typage des propriétés
— Synchronisation dans la base de données relationnelle (générée)
— …

Le modèle est géré en interne dans l'application. Il obéit aux contraintes de cohérence, à l'articulation entre les différentes fonctionnalités de l'application.
Il masque la complexité, et n'est pas accessible directement dans les interfaces de l'application.
Des éléments d'interface seront construits automatiquement en fonction du typage des attributs dans le modèle objets (formulaires, contrôles de saisie, etc.).

Vues : Twig Une représentation de la ressource demandée dans la requête du client.
— arguments de `render()` :

```php
$this->render ( 'todo.html.twig', [
          'id' => $id,
          'todo' => $todo
      ] );
```

— injectés dans le gabarit Twig

Le programme choisit de générer une vue, et le moteur de gabarits en génère une représentation, par exemple, ici, sous forme de page HTML.
Le code du gabarit n'a accès qu'aux données de la vue qui lui sont transmises.
Peu importe le modèle interne de l'application : plusieurs entités reliées par une association, une composition.
La vue est la construite à partir de ce modèle interne, pour obtenir un modèle conforme aux besoins fonctionnels de l'application, potentiellement différent : combinaisons, filtres, tris, masquage, etc.
C'est plus sûr : peu importe toutes les données du modèle que le programme PHP a pu manipuler : Twig n'a accès qu'à un sous-ensemble ou une copie de certaines données, qu'on lui transmet explicitement.
On peut facilement concevoir un code qui peut construire différentes représentations de la même vue (*content negociation*).
Mais on peut aussi donner différentes vues de la même ressource à différents utilisateurs (droits d'accès).

Routage vers les méthodes du contrôleur

```php
#[Route('/todo/show/{id}', name: 'todo_show', methods: ['GET'])]
public function todoShow($id)
{
    // ...
}
```

Routage des requêtes venant de la route (URL) `/todo/show/ID` vers une méthode de la classe Controller qui traîte la requête (GET).
On donne un nom à cette route, qu'on appelle `todo_show`, de façon à pouvoir la référencer ultérieurement, par exemple pour renvoyer vers cette page de l'application.

Cohérence des routes avec `path()`
— Utilisation des routes pour les transitions dans l'application (nommées)
— ~~chemins « en dur »~~.

```twig
<a href=" {{ path('todo_show', { 'id' : todo.id }) }}">
  détails
</a>
```

On peut renommer facilement les chemins, à un seul endroit, dans la déclaration de la route, sans avoir à retoucher tous les gabarits.
Le composant de routage du contrôleur est le garant d'une application qui fonctionne, sans liens pointant dans le vide.

11.4.2 Formulaires avec Symfony

Les contrôleurs permettent de gérer des interactions fines comme les mécanisme d'affichage d'un formulaire HTML et de soumission de ses données

Affichage du formulaire Généré avec un gabarit Twig ?
Différentes façons de faire plus ou moins sophistiquées.

On donne des versions simples à comprendre dans les exemples ci-après. On verra par la suite des versions plus ou moins complexes.
La documentation de référence est dans `http://symfony.com/doc/current/forms.html`

Gabarit d'affichage (GET) Injection d'un formulaire dans un template Twig :

```php
#[Route('/todo/new', name: 'todo_new_get', methods: ['GET'])]
public function newGet(Request $request): Response
{
    $form = $this->createNewTodoForm();

    return $this->render('todo/new.html.twig', [
        'formulaire' => $form->createView(),
    ]);
}
```

Listing 10 : méthode d'affichage d'un formulaire (`TodoController.php`)

on verra `createNewTodoForm()` un peu plus tard

La méthode `createNewTodoForm( )` génère un objet formulaire Symfony. La « vue » qui le représente (sous forme HTML) est injectée dans le template Twig ci-dessous.

Gabarit Twig de la page `new` Code HTML minimal d'un formulaire (liste pas tous champs de saisie des propriétés d'une `Todo`)

```twig
{% extends 'base.html.twig' %}

{% block title %}New Todo{% endblock %}

{% block body %}
  <h1>Ajout d'une nouvelle tâche</h1>

  <form action="#" method="post">
    {{ form_widget(formulaire) }}

    <input type="submit" name="Ajouter" />
  </form>

{% endblock %} {# body #}
```

Listing 11 : `templates/todo/new.html.twig`

`form_widget()` va générer les champs du formulaire HTML automagiquement, en fonction des champs qu'on veut faire gérer par le gestionnaire de formulaire (voir ci-dessous).

Construction du formulaire Formulaire câblé sur une entité du modèle de données : énumère quelles propriétés doivent être traitées dans formulaire

```
function createNewTodoForm()
{
    $todo = new Todo();
    $formbuilder =$this->createFormBuilder($todo);
    $form = $formbuilder
            ->add('title')
            ->add('completed', CheckboxType::class,
                      array('required' => false))
            ->getForm();
    return $form;
}
```

Listing 12 : Génération d'un formulaire avec le *Form Builder* Symfony (`TodoController.php`)

On utilise un constructeur de formulaire Symfony, qui, dans cet exemple, sait gérer l'accès aux données d'une classe PHP du modèle (`Todo`), via des méthodes *getters* ou *setters* de cette classe.
Les options de customisation du formulaire peuvent être définies, comme, par exemple ici, une requête de saisie obligatoire désactivée pour le champ *completed*.
Ici, l'écriture sur les appels à `add( )` qui sont chaînés, tire parti d'une astuce sur les valeurs renvoyée par la méthode `add()` (l'objet *form builder* lui-même) pour qu'on puisse chaîner ainsi les appels.

11.4.3 Gestion de la requête de soumission

Le contrôleur Symfony gère la soumission des données prévue dans le `form` HTML généré par le gabarit

Méthode de soumission des données du formulaire Même chemin URL (`/todo/new`), mais `POST`

```
#[Route('/todo/new', name: 'todo_new_post', methods: ['POST'])]
public function newPost(Request $request, ManagerRegistry $doctrine)
{
    $todo = new Todo();
    $form = $this->createNewTodoForm($todo);
    $form->handleRequest($request);
    if ($form->isValid()) {
        $em = $doctrine->getManager();
        $em->persist($todo);
        $em->flush();

        return $this->redirectToRoute('todo_index');
    }

    // ...
```

Listing 13 : Réception des données du formulaire (`TodoController.php`)

En entrée de la soumission, la requête POST vers `/todo/new` est cette fois aiguillée vers la route `todo_new_post` qui invoque `newPost( )`.
Notez que les deux routes GET et POST sur la même ressource `/todo/new` invoquent deux méthodes différentes du contrôleur (resp. `newGet()` et `newPost()`) puisque l'attribut `methods` des annotations `@Route` est différent dans chaque méthode.
Le formulaire va aller chercher les données qui l'intéressent dans la requête (`$form->handleRequest($request)`), en fonction de la façon dont il a été construit par `createNewTodoForm()`.
Comme le formulaire a été « câblé » vers une instance de la classe `Todo` (dans `$this->createNewTodoForm($todo)`), l'appel à `$form->isValid()` renseigne le contenu de cette instance avec les données saisies (si les contraintes de saisie sont respectées : présence des attributs obligatoires, etc.)
On peut alors utiliser les instructions Doctrine, pour sauver cette nouvelle instance en base de données (l'ajouter, puisqu'elle ne correspond pas à un identifiant connu en interne).

Algorithme fondamental Algorithme de gestion des soumissions

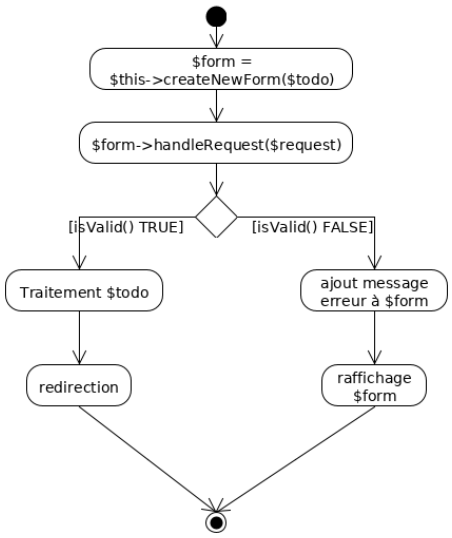


Figure 27 – « Diagramme de décision »

Ce diagramme illustre l'algorithme du code de la méthode ci-dessus. Il est conforme à ce qui a été présenté plus haut (cf. 11.3.1).
En cas d'erreur, on raffiche le même formulaire.
En cas de succès, on effectue une redirection pour renvoyer vers une autre page.

11.4.4 Mise au point dans la barre d'outils Symfony

Historique des requêtes, dans le *profiler*, pour visualiser les détails de la requête POST, avant la redirection :

1. GET (affichage d'un formulaire)
2. **POST** (soumission des données du formulaire)
3. GET (redirection suite au traitement des données soumises)

11.4.5 Générateur Contrôleur CRUD

Encore un générateur :

```
symfony console make:crud Todo
```

Crée pour nous tous ces éléments de formulaires :

```
created: src/Controller/TodoController.php
created: src/Form/TodoType.php
created: templates/todo/_delete_form.html.twig
created: templates/todo/_form.html.twig
created: templates/todo/index.html.twig
created: templates/todo/show.html.twig
created: templates/todo/new.html.twig
created: templates/todo/edit.html.twig
```

```
TodoType
```

11.4.6 Sécurité

— Éviter l'injection de code malveillant
— Éviter de rejouer les requêtes POST
    — utilisation d'un *cookie* CSRF (vu plus tard)
    — vérifie que le POST correspond bien au formulaire généré précédamment via le GET

Take away

— principe HATEOS pour gérer les transitions entre pages
— Formulaires HTML
— Algorithme de gestion de la soumission des données dans les contrôleurs Symfony

12 Séq. 8 : Gestion contextuelle des formulaires

Objectifs de cette séquence

Cette section revient sur les fonctionnalités des formulaires de Symfony, pour détailler le fonctionnement des mécanismes qu'on a utilisé jusqu'ici dans le cours, et approfondir des éléments plus avancés.

12.1 Formulaires améliorés

Cette section revient sur le fonctionnement des formulaires et certains aspects de leur intégration avec le modèle des données Doctrine, pour gérer plus de contraintes et rendre le code plus spécifique, sans perdre en expressivité.

12.1.1 Vers des formulaires gérés avec une seule méthode

— **Affichage d'un formulaire** HTML : méthode d'un contrôleur (insertion d'un formulaire dans un template)
— Gestion de la **soumission du formulaire** : méthode du même contrôleur (par exemple la même méthode)

Passer à une seule méthode :
— Rendre le code plus compact
— Lisibilité ?

Examinons tout d'abord le code à deux méthodes qu'on a déjà vu :

Rappel : Variante à 2 méthodes Méthode gérant le GET : affichage du formulaire HTML

```php
#[Route('/todo/new', name: 'todo_new_get', methods: ['GET'])]
public function newFormGet(): Response
{
    $todo = new Todo();
    // init valeurs par défaut
    $form = $this->createForm(TodoType::class, $todo);

    return $this->render('todo/new.html.twig', [
        'todo' => $todo,
        'form' => $form->createView(),
    ]);
}
```

Méthode gérant le POST : traitement des données soumises

```php
#[Route('/todo/new', name: 'todo_new_post', methods: ['POST'])]
public function newFormPost(Request $request, TodoRepository $todoRepository): Response
{
    $todo = new Todo();
    $form = $this->createForm(TodoType::class, $todo);
    $form->handleRequest($request);

    if($form->isSubmitted() && $form->isValid()) {
        $todoRepository->add($todo, true);

        return $this->redirectToRoute('todo_index');
    }
    # else {
    return $this->render('todo/new.html.twig', [
        'todo' => $todo,
        'form' => $form->createView(),
    ]);
    # }
}
```

Les deux méthodes foctionnent de concert, l'une gérant la requête GET initiale, et l'autre la soumission des données dans la requête POST.
La gestion de la continuité entre ces deux méthodes s'effectue grâce à la classe gestionnairede formulaire `TodoType`.

12.1.2 Formulaire moderne

Fusionne en une seule méthode gestion du GET et du POST

Cf. Forms dans la documentation Symfony.

Route et méthode unique

```php
#[Route('/todo/new', name: 'todo_new', methods: ['GET', 'POST'])]
public function new(Request $request, TodoRepository $todoRepository): Response
{
    $todo = new Todo();
    $form = $this->createForm(TodoType::class, $todo);
    $form->handleRequest($request);
    if ($form->isSubmitted() && $form->isValid()) {
        $todoRepository->add($todo, true);

        return $this->redirectToRoute('todo_index');
    }

    return $this->render('todo/new.html.twig', [
        'todo' => $todo,
        'form' => $form->createView(),
    ]);
}
```

La méthode unique gère à la fois les requêtes GET et les requêtes POST.
En cas de requête GET, `isSubmitted()` sera faux.
Moins de code à écrire, mais moins simple à comprendre si on n'a pas étudié la décomposition en deux méthodes.

12.1.3 Génération des contrôleurs et formulaires

Symfony peut aussi générer des classes contenant le code des **contrôleurs** et les **formulaires**

```
symfony console make:crud
```

12.1.4 Exemple : nouvelle tâche

```php
#[Route('/new', name: 'todo_new', methods: ['GET', 'POST'])]
public function new(Request $request, TodoRepository $todoRepository): Response
{
    $todo = new Todo();
    $form = $this->createForm(TodoType::class, $todo);
    $form->handleRequest($request);

    // Gestion du POST
    if ($form->isSubmitted() && $form->isValid()) {
        $todoRepository->add($todo, true);

        return $this->redirectToRoute('todo_index', [], Response::HTTP_SEE_OTHER);
    }

    // Gestion du GET ou problème validation
    return $this->renderForm('todo/new.html.twig', [
        'todo' => $todo,
        'form' => $form,
    ]);
}
```

— `TodoType` est vue plus loin

Gabarit `todo/new.html.twig` :

```twig
{% extends 'base.html.twig' %}

{% block title %}New Todo{% endblock %}

{% block main %}
    <h1>Create new Todo</h1>

    {{ include('todo/_form.html.twig') }}

    <a href="{{ path('todo_list') }}">back to list</a>
{% endblock %}
```

Gabarit `templates/todo/_form.html.twig` :

```twig
{{ form_start(form) }}
    {{ form_widget(form) }}
    <button class="btn">{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

(aussi inclus par `todo/edit.html.twig`)

Ici, on a reproduit le code tel qu'il est généré par `make:crud`.
Ci-dessus, on ne voit rien qui spécifie la liste des « *widgets* » qu'on veut voir apparaître dans le formulaire de création.
Symfony peut en générer un automatiquement, par défaut, ou on peut en spécifier un explicitement au niveau d'une classe `TodoType` qui surcharge la méthode standard de construction d'un formulaire, qui sera invoquée par l'appel à `createForm()`.

12.1.5 Classe `TodoType` gestionnaire de formulaire

L'examen des propriétés des attributs de `Todo` permet de générer des champs de saisie de formulaires HTML ad-hoc.
Y compris pour le champ `project` qui permettra de sélectionner un projet existant dans une liste défilante.
On voit comment on a spécifié ici que l'attribut `completed` sera affiché sous forme de liste à choix multiples, plutôt qu'une case à cocher, par exemple.

```php
class TodoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title', TextType::class,
                  ['required' => false])
            ->add('completed', ChoiceType::class,
                  ['choices' => [
                      'Maybe' => null,
                      'Yes' => true,
                      'No' => false ]
                  ])
            ->add('project');
    }
    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([
            'data_class' => Todo::class,
        ]);
    }
}
```

Listing 14 : Extrait de `src/Form/TodoType.php`

12.2 < / code formulaire moderne >

12.2.1 Chargement auto des instances

Mécanisme *MapEntity* des contrôleurs
— Paramètre `{id}` de l'attribut `Route` : extrait du chemin de la requête HTTP
— Argument de la méthode : `$todo`, instance de `Todo`
— Chargement auto grâce au *Repository* de `Todo` (appel à `TodoRepository::find()` *automagique*)

```php
class TodoController extends Controller
{
    #[Route('/todo/{id}', name: 'todo_show'),
      requirements: ['id' => '\d+'], methods: ['GET'])]
    public function showAction(Todo $todo): Response
    {
        return $this->render('todo/show.html.twig', array(
            'todo' => $todo,
        ));
    }
```

Dans la plupart des méthodes des contrôleurs CRUD, on va agir sur une entité de la base de données, identifiée par son identifiant unique transporté dans les paramètres fournis par le client dans l'URL.
On peut utiliser ce mécanisme *MapEntity* pour effectuer le chargement automatique des données, et s'éviter l'écriture des appels à `find()`.
En cas de chargement infructueux, la gestion de l'erreur 404 est automatique.

12.3 Aller au-delà des CRUDs basiques

Dans une application Web, on ne se contente pas d'encapsuler de simples interactions CRUD sur des entités seules, isolées.
Dans l'expérience utilisateur, on navigue au sein d'un ensemble de données liées.

12.3.1 Ajout entités liées

Examinons un contrôleur permettant de gérer un formulaire d'ajout d'une tâche directement dans le contexte d'un projet.
Un tel contrôleur sera légèrement plus complexe que les contrôleurs par défaut générés par `make:crud`.

Version basique Version `/todo/new` créée via `make:crud`
— Formulaire `TodoType` :

```
class TodoType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            // ...
            ->add('project');
```

la référence au projet de la tâche (`Todo::project`) est gérée comme une propriété « ordinaire » de la tâche dans le formulaire
Résultat : liste de choix parmi les projets

Figure 28 – « Sélection d'une entité liée dans un popup »

Pas ce qu'on souhaite, si une tâche ne peut pas changer de projet

Le formulaire par défaut ajoute un sélecteur de projet dans le formulaire de création d'une tâche.
Ce n'est pas idéal, puisque nous connaissons déjà le projet : inutile de le redemander à l'utilisateur.

Formulaire d'ajout, depuis l'affichage d'un projet
— Dans page affichage d'un projet (`project_show`, sur `/project/{id}`)
— Sous l'affichage de la liste des tâches d'un projet : bouton/lien « *ajouter nouvelle tâche* »
— Chemin du formulaire d'ajout : `/project/{id}/addtodo`
— `id` de projet (connu) !

On voudrait disposer d'un bouton directement sous la liste des tâches d'un projet permettant d'en ajouter une nouvelle, pointant vers un formulaire ayant déjà la connaissance du projet, sans avoir à le sélectionner, comme précédamment.

Code final

```
#[Route('/project/{id}/addtodo', name: 'todo_add',
        methods: ['GET', 'POST'])]
public function addTodo(Request $request, Project $project,
                        TodoRepository $todoRepository): Response
{
        $todo = new Todo();
        $todo->setProject($project);
```

```
        $form = $this->createForm(TodoType::class, $todo);
        $form->handleRequest($request);
        if ($form->isSubmitted() && $form->isValid()) {

            $todoRepository->add($todo, true);
            // ...
```

Contexte :
— chargement *automagique* du projet d'après l'`{id}` de la route
— tâche créée dans son projet :
`$todo->setProject($project)`

Dans cet exemple, l'*Entity Value Resolver* associe l'identifiant présent dans la route (`{id}`) au chargement de l'instance de `Project` attendue en argument. Ensuite, la seule chose qui change par rapport au code de `new()` vu précédamment, est l'initialisation de l'association entre la nouvelle tâche et son projet, réalisée via `$todo->setProject($project)`.
Il ne reste plus qu'à modifier le comportement du formulaire, pour qu'il n'affiche plus le champ de saisie du projet quand ce n'est pas nécessaire.

Supprimer le champ `project` du formulaire `TodoType` Éviter la modification du champ `project`.
Comportement optionnel (garde fonctionnement initial formulaire nouvelle tâche isolée).

```
$form = $this->createForm(TodoType::class, $todo,
        ['display_project' => false]);
```

Listing 15 : Passage d'une option dans `TodoController.php` :

```
public function buildForm(FormBuilderInterface $builder, array $options): void
    {
        $builder
            ->add('title')
            ->add('created', DateType::class, [
                  'widget' => 'single_text'])
            ->add('project', null, [
                  'disabled' => $options['display_project'] ]);
    }
```

Listing 16 : Génération du contenu optionnel du formulaire (`TodoType.php`) :

On spécialise le formulaire en lui ajoutant un comportement différent, au cas où une option est passée, qui vise à désactiver l'affichage du champ de saisie du projet.
Ici, quand l'option `display_project` est définie à *faux* à la création du formulaire pour cet usage particulier, le champ est rendu inactif.
Il reste cependant utilisable si l'option n'est pas précisée, dans le cas où le formulaire est utilisé pour la création de tâche en dehors du contexte d'un projet.

12.4 Téléversement d'images

Examinons enfin l'utilisation d'un *bundle* additionnel pour permettre d'ajouter simplement à notre application un mécanisme de télé-versement (*upload*) d'images dans le site de l'application.

12.4.1 Outil pour upload d'image dans formulaire Symfony

Exemple : utilisation de `VichUploaderBundle` pour mettre des photos dans les *Pastes*
— Formulaire upload de photo
— Stockage dans public/
— Affichage dans gabarit

Nous reprenons quelques éléments sur la mise en oeuvre de ce gestionnaire, mais la référence est sa documentation.

12.4.2 Entité *Paste*

```
#[ORM\Entity(repositoryClass: PasteRepository::class)]
#[Vich\Uploadable]
class Paste
{
    //...
    #[ORM\Column(nullable: true)]
    private ?string $imageName = null;

    #[Vich\UploadableField(mapping: 'pastes',
                           fileNameProperty: 'imageName')]
    private $imageFile;

    //           .../...
```

— `imageName` stocké en base (Doctrine)
— `imageFile` objet « fichier téléversé » Symfony

```
        /**
    * @param File|\Symfony\Component\HttpFoundation\File\UploadedFile $imageFile
    */
        public function setImageFile(?File $imageFile = null): void
        {
                $this->imageFile = $imageFile;

                //...
        }

}
```

L'attribut `imageName` stockera dans la base de donnée le nom de l'image qui a été téléversée. Il n'est pas destiné à être modifié par l'utilisateur (on le désactivera ci-après).
L'attribut `imageFile` sert à gérer la soumission dans le formulaire qui sera envoyé par le navigateur. Il n'a pas son pendant dans la base de données. Il sert uniquement à la soumission dans le formulaire HTML et l'envoi d'un fichier encodé dans le contenu de la requête POST.

12.4.3 Formulaire *Paste*

```
class PasteType extends AbstractType
{
        public function buildForm(FormBuilderInterface $builder,
                                  array $options)
        {
                $builder
                        ->add('content')
                        ->add('created')
                        ->add('content_type')
                        ->add('imageName', TextType::class,
                              ['disabled' => true])
                        ->add('imageFile', VichImageType::class,
                              [
                                'required' => false,
```

```
                            'delete_label' => 'Delete image ?'
                        ])
            ;
    }
```

L'attribut `delete_label` qu'on ajoute dans les paramètres du champ `imageFile` du formulaire correspond à la correction d'un bug dans le code, qui oublie l'intitulé de cet élément.

Take Away

— Gestion optimale intégrée entre formulaires et Doctrine
— Ne pas se limiter à `make:crud`
— Gestion des données liées dans les associations
— *Bundle* pour téléversement d'images simple

13 Séq. 8 : Sessions - Contrôle d’accès

Objectifs de cette séquence

Dans une première partie, on va étudier la façon dont les **sessions** permettent des interactions évoluées, au-dessus des requêtes et réponses HTTP successives, afin que l'utilisateur fasse l'expérience d'une réelle **session applicative**, et bien que le serveur HTTP soit naturellement **sans état**.
Dans un deuxième temps, on va présenter le principe des mécanismes de contrôle d'accès qui seront mis en œuvre pour reconnaître les utilisateurs de l'application et leur donner la permission d'utiliser ou non les fonctionnalités de celle-ci.

13.1 Sessions applicatives

Les serveurs HTTP sont basés sur les principes d'architecture REST, donc sans état (*stateless*).
Pourtant les applications Web fonctionnant sur le serveur doivent connaître et reconnaître les utilisateurs et leurs clients HTTP pour offir une expérience utilisateur satisfaisante.
Cette section présente le mécanisme des sessions qui permet aux applications Web de palier au caractère sans état du serveur HTTP.

13.1.1 Session (en informatique)

— « **session** n. f. (*télécommunications*, *informatique*) : Période de temps continue qui s'écoule entre la connexion et la déconnexion à un réseau ou à un système, ou encore l'ouverture et la fermeture d'un logiciel d'application. »
Source : Grand dictionnaire terminologique - Office Québécois de la langue française
— « In computer science and networking in particular, a session is a time-delimited two-way link, a practical (relatively high) layer in the TCP/IP protocol enabling interactive expression and information exchange between two or more communication devices or ends – be they computers, automated systems, or live active users (see login session). […] »
Source : `https://en.wikipedia.org/wiki/Session_(computer_science)`

13.1.2 Paradoxe : applications sur protocole sans état

— L'expérience utilisateur suppose une **instance unique** d'exécution d'application, comme dans un programme « sur le bureau »
— Faciliter la contextualisation du HATEOS : quelles transitions sont valides à partir de l'état courant ?
— 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.).

13.1.3 L'application n'arrête pas de s'arrêter

À chaque requête :

1. démarrage application
2. routage vers méthode Contrôleur
   (a) chargement depuis base **et/ou session**
   (b) traitements

(c) sauvegarde en base **et/ou session**

3. envoi Réponse
4. **mort**

Sans la session, continuité entre deux requêtes ?

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.

13.1.4 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

13.1.5 Le client HTTP peut s'identifier

— Argument d'invocation dans URL
— en-tête particulier ?
— ***Cookie***
— …

13.1.6 Identification du client par *cookie*

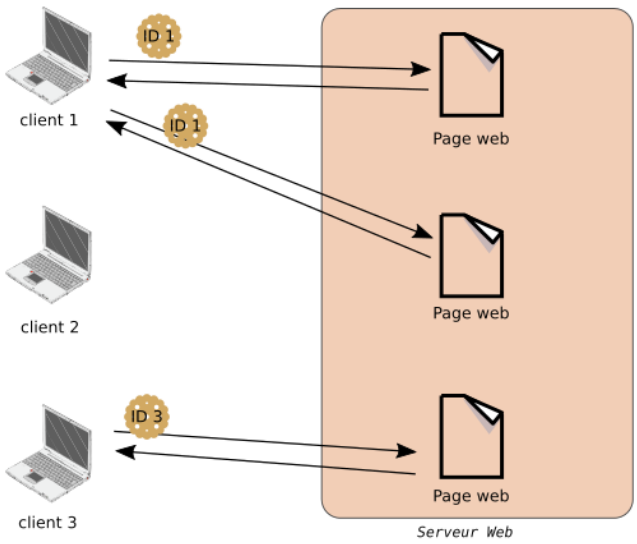


Figure 29 – Multiples clients et pages

— 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 ?

13.1.7 *Cookies*

— Juste un « jeton » unique sur un certain périmètre (serveur, chemin d'URL, application, …)
— Format choisi par le serveur
— Taille limitée :
    — peut contenir des données de l'application
    — pas un état de l'application complet
— Données en clair dans le *stockage de cookies du navigateur*
— Durée de vie potentiellement grande

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.

13.1.8 Reconnaître le client HTTP

— Données du *cookie* stockées sur client, envoyées avec la requête au serveur
— Le serveur peut trouver des données plus complètes stockées de son côté (une **session**), sur présentation d'un identifiant (présent dans le *cookie*).
— À chaque requête, le *cookie* permet relier :
    — nouvelle requête d'un client HTTP (en-têtes HTTP : **valeur d'un *cookie* existant**)
    — **mémoire de l'état précédent** sauvegardé côté serveur à la **fin de la réponse HTTP précédente** du même client

La session correspond au client HTTP, qui va mémoriser le contexte d'utilisation de ce client par l'utilisateurice.
Si une même utilisatrice utilise différents clients HTTP (par ex. un mobile d'un côté, et un ordinateur de bureau de l'autre), il peut y avoir plusieurs sessions en cours, avec des données pas nécessairement cohérentes dans chacune de ces sessions.

13.1.9 Stocker une session

— Session : espace de stockage unique **côté serveur**
— **Objets du modèle** de l'application stockés dans la session (pas contrainte taille), *sérialisés*
— Session retrouvée via un **identifiant**
    — Identifiant fourni par un *cookie* (local au client)
— Durée de vie et unicité des sessions au choix du serveur

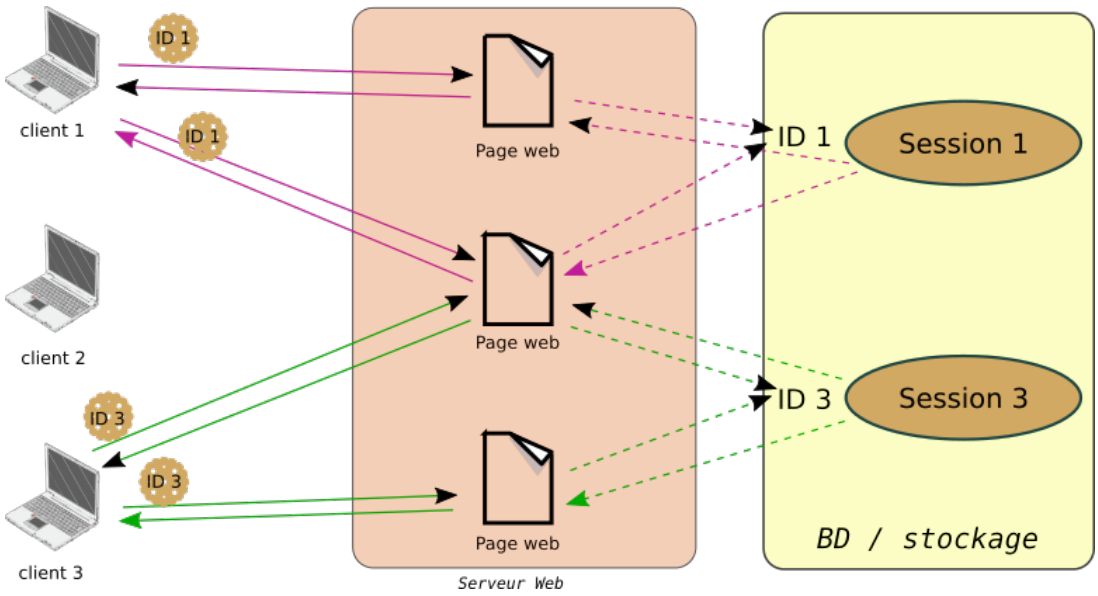


Figure 30 – Stockage session côté serveur

La « même » page Web accédée par deux clients présentant des ID de cookies différents, ne sera peut-être pas le même document, s'il est généré dynamiquement en fonction des informations présentes dans la session.
Là où le *cookie* est stocké côté client HTTP, la session correspondante est stockée côté serveur.
La session est potentiellement grosse, peut-être stockée en base de données, ou dans des fichiers… Mais pas dans la même base de données que le Modèle de l'application.
En pratique, le plus souvent, cette session est stockée 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 avec des exécutions successives sur des serveurs différents).
La plate-forme du langage de programmation rend l'utilisation de la session très facile pour le programmeur, en masquant la complexité sous-jacente. Cf. `https://symfony.com/doc/current/session.html` pour Symfony.

13.1.10 Détails *cookie*

— Créé lors de la première requête d'un client (n'ayant pas fourni ce *cookie*)
— Le serveur délivre les *cookies* en les **intégrant dans en-têtes de réponse** HTTP
  — utilise l'en-tête particulier « `Set-Cookie` » (sur une ligne)
    `Set-Cookie: <nom>=<valeur>;expires=<Date>;domain=<NomDeDomaine>; path=<Path>`
— Le client stocke le *cookie* reçu dans la réponse, pour pouvoir le renvoyer aux prochaines requêtes vers ce serveur
— 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

<HTML>...
```

— Par la suite, ce cookie sera renvoyé par le client au serveur, dans 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.

13.1.11 *Wrap-up* sessions

— Requêtes HTTP déclenchées lors demandes de **transition d'un état à l'autre** de l'application
— L'exécution (PHP) résultante s'effectue sur serveur HTTP **sans mémoire** des interactions précédentes entre client et serveur (*stateless*)
— L'utilisateur a une **impression de continuité** : une seule session d'utilisation de l'application, où requêtes successives ont des **relations de causalité**
— Différentes solutions techniques, dont les *cookies*

Une autre solution consiste par exemple à matérialiser l'historique des dialogues requête-réponse précédents entre le même client et le mêm serveur dans l'URL (arguments).

13.2 Contrôle des accès

Cette section présente le principe des mécanismes de contrôle d'accès qui seront mis en œuvre pour reconnaître les utilisateurs de l'application et leur donner la permission d'utiliser ou non les fonctionnalités de celle-ci.

13.2.1 Protéger les données / fonctions

— Confidentialité : application accessible sur Internet, même si processus / données privés
— Privilèges : qui fait quoi
    — Spécifications fonctionnelles (profils utilisateurs)
    — Contrôle par l'application (HATEOS)
— Contrôle d'accès : reconnaître les utilisateurs, et mettre en place les restrictions (sans nuire à l'utilisabilité, mobilité, etc.)

Autres aspects sécurité vus dans une séance ultérieure

13.2.2 Contrôle des accès

— Protéger l'accès aux fonctionnalités de l'application
— Qui est autorisé à faire quoi

Dans un monde ouvert (Internet, Web, standards)

Dans la vie d'une entreprise, on peut déployer des applications sans nécessairement les déployer sur le Web, sur Internet, donc ouvertes à tous les vents…
Mais on rend alors l'accès délicat : intranet/extranet, nécessité d'un VPN, compatibilité avec terminaux mobiles, utilisateurs nomades, etc.
Déployer sur le Web garde des avantages.

Sécurité par obscurcissement ?
— Ne pas protéger spécifiquement,
— et ne pas documenter / expliquer / rendre visible ?

Ce n'est pas parce que le code de l'application est caché sur le serveur que les méchants ne trouveront pas des failles !
#Fail

Il faut partir d'un principe de mise en place de contrôles effectifs, d'autant plus si le code source de certains éléments de l'application est facilement récupérable (HTML, JS).
Attention aussi à protéger l'accès aux « couches basses » : middleware, base de données, code source, fichiers de configuration.
Le Cloud n'aide pas, de ce point de vue (repositories de code publics, stockage Cloud non-protégé).

Contrôle effectif
— Au niveau de la configuration du serveur (ne pas permettre aux clients de découvrir les failles en regardant le source)
— **Dans les fonctionnalités du logiciel** : configuration dans le code du projet Symfony (module « *firewall* »)
— Mesures complémentaires (audit, etc.)

C'est donc le travail du programmeur de vérifier, à chaque étape, notamment du routage des requêtes, que le client est bien autorisé à accéder aux fonctionnalités de l'application, qu'il ait suivi un lien proposé par l'application, ou qu'il ait fait une tentative malveillante.

13.2.3 Modèle contrôle des accès

Identification l'utilisateur fournit à un service un moyen de le reconnaître : **identité**

Authentification le service **vérifie** cette identité

Autorisation le service donne à l'utilisateur certaines **permissions**

Les trois éléments ci-dessus sont fondamentaux et se retrouvent dans de nombreux contextes, pas uniquement pour les applications déployées sur le Web.

1) Identification
— Identifiants :
  — email
  — identifiant d'utilisateur (*login*)
  — certificat client X509 (TLS)
  — Jeton dans un en-tête HTTP
  — …
— L'identification doit être confirmée par l'authentification

Importance des standards : compatibilité avec tous les clients Web.
Les certificats clients X509, par exemple, ont posé des problèmes du fait d'une ergonomie discutable dans la mise en œuvre dans les navigateurs… et de la méconaissance des utilisateurs des mécanismes associés (révocation, non-dissémination, etc.).

2) Authentification
— Vérifie client agit bien pour le **propriétaire légitime de l'identifiant** présenté
— Protocole de vérification :
  — mot-de-passe
  — signature valide d'une donnée quelconque (*challenge*) avec la clé privée associée au certificat client
  — délégation à service externe (Shibboleth/CAS, OAuth 2, …)
  — 2FA (*two-factor authentication*)
  — nouveaux standards : U2F, Fido, TOTP, …
  — …

Importance des mécanismes de double facteur d'authentification (2FA).
Problème des mots-de-passe : évolution vers un monde sans mots-de-passe sur le Web : diffusion de nouveaux standards.
Attention à déléguer à un service tiers : est-il fiable ?

3) Autorisations
— Permissions applicatives associées à l'identifiant
— Pour une certaine période
— Génération d'un jeton temporaire (session authentifiée)
    — *Cookie*

Alors que l'authentification peut être déléguée, la gestion des autorisations est du ressort du développeur de l'application.

13.2.4 Dans protocole HTTP

— Identification / authentification de « bas niveau » **dans le protocole** HTTP (cf. ~~RFC2617~~ RFC 7617 : The 'Basic' HTTP Authentication Scheme)
— Rappel : HTTP est sans état
    — Le client HTTP doit se réauthentifier à chaque requête
— Permet de transporter l'authentification dans les en-têtes
— Alternative : **authentification applicative** + session applicative

Le protocole HTTP supporte certaines fonctionnalités relatives à l'authentification, mais que l'ergonomie des navigateurs rend assez difficile à utiliser en pratique. Par exemple, absence de fonction permettant de se déconnecter, nécessitant de quitter le navigateur.
C'est pourquoi on en vient aujourd'hui à l'utilisation de l'authentification applicative dans la très grande majorité des applications Web.

13.3 Authentification Web

Cette section présente un mécanisme d'authentification de base, l'authentification applicative, via un formulaire dédié construit par l'application, dans une de ses pages Web.

13.3.1 Mécanismes

— Authentification HTTP
    — Authentification « Basic »
    — autres
— **Authentification applicative**

13.3.2 Basic Auth

Inconvénient : pas de *logout*

13.3.3 Authentification applicative

Gestion de l'identification et de l'authentification par l'application
— L'authentification est une des fonctionnalités de l'application, via la session
— **Formulaire** d'authentification
    — *Login*

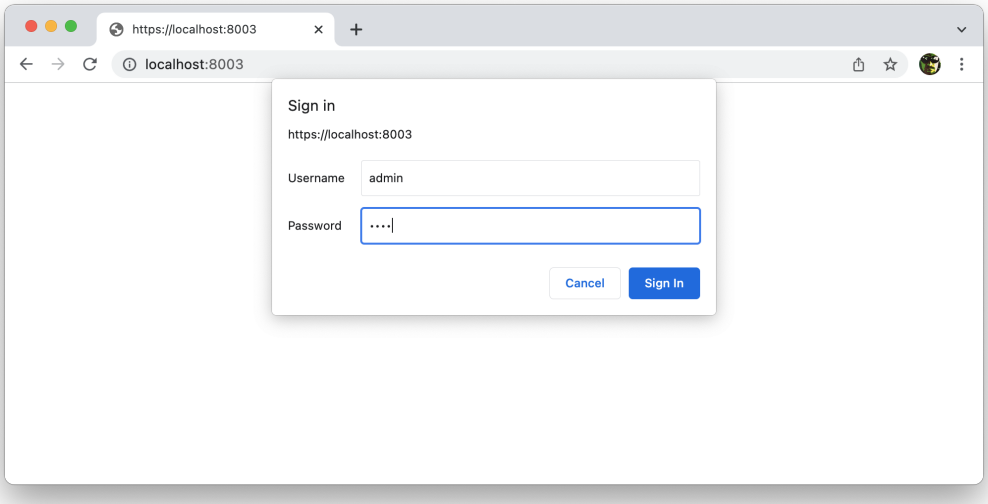


Figure 31 – Source : Why I'm Using HTTP Basic Auth in 2022 par Joel Dare

    - Mot-de-passe
- « base de données » de comptes

C'est un module particulièrement sensible : ne pas improviser son développement.
Atention aux contraintes juridiques en plus de techniques.

Formulaire d'authentification
- Formulaire « standard »
- Champs :
    - Login ou email
    - Mot-de-passe (saisie cachée)

Le formulaire est assez standard, mais recèle, en pratique des champs cachés que l'utilisateur ne voit pas.

Vérification de l'authentification
- Comparer avec profil d'utilisateur connu (en base de données)
- Générer une **session** pour reconnaître l'utilisateur par la suite
- Attention : attaques « force brute »
    - Invalider un compte/profil, ou faire une gestion de surcharge qui désactive les tentatives (*throttling*, *blacklist* réseau, etc.)

Pour contrer les attaques par force brute, différentes stratégies sont possibles, qu'on ne détaille pas plus dans ce cours.
On va voir un peu plus loin l'utilisation d'un mécanismes de ce type, les CAPTCHA.

Dans Symfony
- Composant `Security`
- Flexible : gestion souple et extensible de l'authentification
- Gère par exemple les utilisateurs dans la base de données via classe `User` + Doctrine
- Assistants générateurs de code pour les dialogues

Procédures ?
— Gestion des mots-de-passe (qualité aléa, longueur, stockage approprié, etc.)
— Récupération de compte si oubli mot-de-passe
    — Canal sécurisé ou envoi jeton de réinitialisation sur email (implique gestion emails)
— Confirmations d'authentification pour sections critiques de l'application
— Garder des traces (audit, obligations légales)
— Conformité RGPD (données personnelles dans les profils)

Complexe, donc tentative de déléguer à un tiers… mais ce tiers est-il fiable ?
Augmentation des risques pour les utilisateurs.
Si on est piraté, seuls nos clients sont victimes. Mais est-ce que ça vaut le coup pour les pirates ?
Il vaut peut-être mieux qu'ils essayent de pirater FaceBook, et ce jour-là gare à nous (tous) qui avons intégré une authentification via FaceBook… ?

Captcha *Completely Automated Public Turing test to tell Computers and Humans Apart*
Vérifier qu'un humain est aux commandes

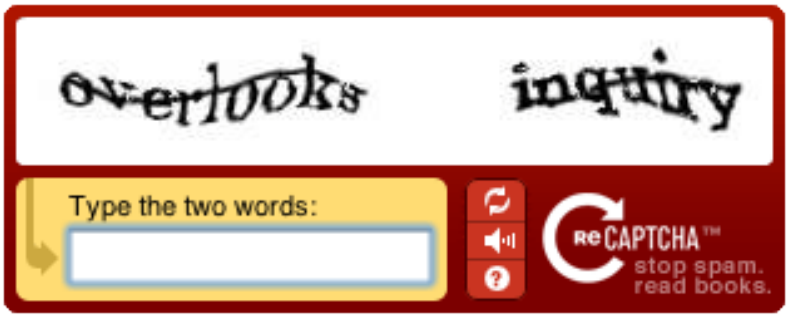


Figure 32 – Exemple reCAPTCHA

— Pas infaillible
— Problèmes accessibilité
— Travail dissimulé
— Surveillance

Tout comme pour les CDN (*Content Delivery Network*) ces Captcha sont opérés par des tiers, et peuvent donc leur servir à tracer les utilisateurs des applications, ce qui est potentiellement problématique à l'ère post-Snowden, et au regard du RGPD.
De plus, pour l'exemple de l'illustration « CAPTCHA et digital labo », les Captcha de ce type peuvent permettre à leurs opérateurs (ici Google) d'entrainer des mécanismes d'IA grâce au travail « bénévole » (contraint) des utilisateurs qui essayent de résoudre le puzzle… en espérant que ça ne serve pas *in-fine* à des applications militaires, par exemple (véhicules ou systèmes d'armes autonomes) !
Pour une ressource récente sur le sujet, voir CAPTCHA : les machines « prouvent » plus rapidement qu'elles sont des humains (*NextImpact août 2023*)

Authentification à double facteur 2FA (*Two factor authentication*)
— + robuste :
    1. élémént connu
    2. élément possédé
— Exemples :
    — carte bancaire (possession) + code PIN (connu)
    — login + mdp (connu) + SMS reçu (possession mobile)
    — login + mdp (connu) + badge de sécurité générant un code unique (possession)
    — login + mdp (connu) + code TOTP récupéré dans appli sur ordiphone

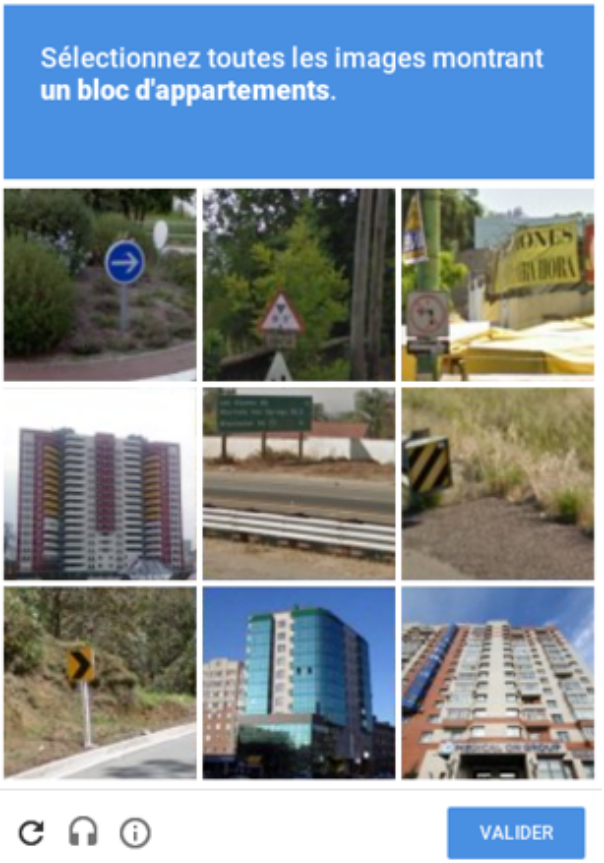


Figure 33 – CAPTCHA et digital labor

Authentification plus forte.
Attention : certains mécanismes s'avèrent moins fiable que prévu (SMS)
Attention aux exigences de sécurité réglementaires.

13.4 Rôles et permissions

Cette section présente le principal modèle de définition de permissions utilisé pour le contrôle d'accès, à base de rôles.

13.4.1 *Role-Based Access Control* (RBAC)

Contrôle d'accès à base de rôles
— Utilisateur
— **Rôle**
— Permissions

Au-lieu d'attribuer des permissions à un utilisateur, on les attribue à un rôle, qu'on délègue à un utilisateur : les permission sont gérées en fonction de la structure de l'organisation, indépendamment des embauches, départs ou changement de responsabilité des individus.
Ce modèle n'est pas spécifique aux applications Web, mais est présent dans de nombreux contextes applicatifs ou système.

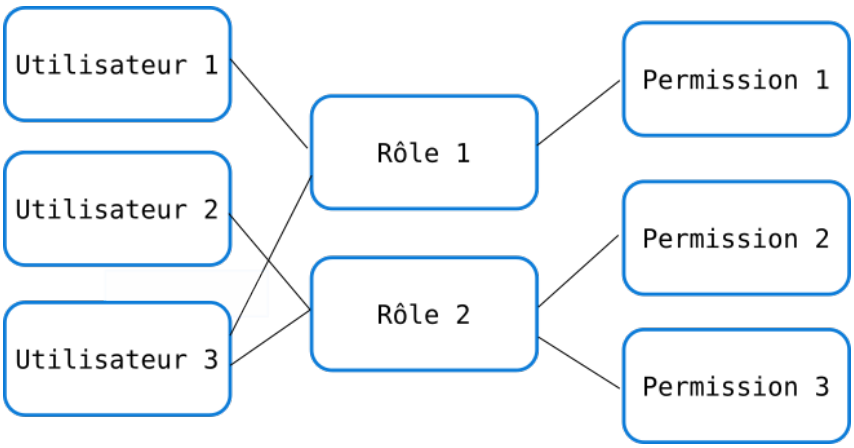


Figure 34 – Exemple d'affectation de rôles

13.4.2 Permissions

— Modèle applicatif de permissions
— Vérifier les permissions à chaque traitement de requête
  — Routage
  — Dans les traitements fonctionnels

Module « *Firewall* » de Symfony

13.4.3 Réponses Web

— Code 200 + Page mentionnant problème de permissions
— **Code 403** (et peut-être un message dans la page) ?

Idéalement, les applications doivent renvoyer un code de statut 403, en cas d'interdiction d'accès, mais certains programmeurs oublient cela, et renvoient un message dans une page « classique » chargée en réponse 200 « OK »…

13.5 Mise en œuvre avec Symfony

Cette section présente la façon dont on peut mettre en œuvre les mécanismes d'authentification et de contrôle d'accès dans Symfony.

13.5.1 Flexibilité

— Symfony permet de gérer plein de modalités d'authentification
— Choix : s'appuyer sur la base de données, et des contrôleurs d'authentification générés par les assistants

Symfony peut s'adapter à de nombreux contextes de déploiement, et permet de s'interfacer avec différentes sources pour la gestion de l'identification et l'authentification des utilisateurs.
On fait le choix de présenter ici le système le plus classique qui pourra être utilisé pour le projet, qui s'appuie sur la base de données.

13.5.2 Gestion des utilisateurs avec Doctrine

— Classe `User` du Modèle (et *mapping* Doctrine en base)
— Définition de règles dans le *firewall* Symfony

— Rôles
— Ajouter des formulaires (+ *templates*) :
    — Login + password
    — Logout
    — (Inscription, rappel du mot-de-passe, …)

On utilise les assistants générateurs de code pour mettre en place une classe utilisateur et un contrôleur et ses formulaires nécessaire à l'authentification.

13.5.3 Classe `User`

```
symfony console make:user
```

```php
namespace App\Entity;

use App\Repository\UserRepository;
use Doctrine\ORM\Mapping as ORM;
use Symfony\Component\Security\Core\User\PasswordAuthenticatedUserInterface;
use Symfony\Component\Security\Core\User\UserInterface;

#[ORM\Entity(repositoryClass: UserRepository::class)]
class User implements UserInterface, PasswordAuthenticatedUserInterface
{
    // ...

    #[ORM\Column(length: 180, unique: true)]
    private ?string $email = null;
```

(appelée `Member` dans le projet)

13.5.4 Hiérarchie de rôles

— Arbitraire, selon les besoins de l'application
— Exemple :

1. `ROLE_SUPER_ADMIN`
2. `ROLE_ADMIN`
3. `ROLE_CLIENT`
4. `ROLE_USER`

```
# security.yml

role_hierarchy:
    ROLE_CLIENT:      ROLE_USER
    ROLE_ADMIN:       ROLE_USER
    ROLE_SUPER_ADMIN: [ROLE_USER, ROLE_ADMIN]
```

13.5.5 *Firewall*

Contrôle l'accès aux URLs en fonction des rôles :

```
# app/config/security.yml
security:
    # ...

    firewalls:
        # ...
        default:
            # ...

    access_control:
        # require ROLE_ADMIN for /admin*
        - { path: ^/admin, roles: ROLE_ADMIN }
```

Cette première façon de contrôler les accès, au niveau du « firewall » applicatif, agit très en amont.
Il s'agit de bloquer les requêtes par rapport à des motifs de chemins des routes, définis globalement, via des fichiers de configuration : peu de souplesse pour des cas particuliers.
Expressions rationnelles : `"^/admin"` signifie tout chemin de route qui commence par `/admin`.
D'autres possibilités existent (programmées).

13.5.6 Utilisation dans les contrôleurs

— Contrôle d'accès sur les routes :

```php
#[Route('/comment/{postId}/new', name: 'comment_new', methods: ['GET', 'POST'])]
#[IsGranted('IS_AUTHENTICATED_FULLY')]
function addComment(Post $post): Response {
    //...
```

`IS_AUTHENTICATED_FULLY` : un utilisateur qui vient vraiment de se reconnecter
— Contrôle d'autorisation dans le code des méthodes :

```php
public function adminDashboard(): Response {
    $this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'Access denied!');
```

« Entrée interdite, à moins que… »

Ces façons de faire sont plus fines, et permettent un filtrage :
- — au cas par cas, route par route
- — ou encore plus fine dans une algorithme, en fonction d'éléments de contexte très spécifiques

On voit ici des exemples de critères comme :
`is_granted('IS_AUTHENTICATED_FULLY')` qui correspond à tout utilisateur authentifié (quelque soit son rôle), ou `denyAccessUnlessGranted('ROLE_ADMIN'...` qui vérifie bien qu'un utilisateur dispose d'un rôle précis.

13.5.7 Profil de l'utilisateur

— Accès aux propriétés de l'utilisateur :

```php
$this->getUser()

// ...

$email = $this->getUser()->getEmail();
$post->setAuthorEmail($email);
```

13.5.8 Personnalisation apparence

Gabarits Twig

```twig
{% if is_granted('ROLE_ADMIN') %}
  <a href="...">Delete</a>
{% endif %}
```

Une fois que le filtrage des accès possibles est bien en place, et vérifié activement dans le code, comme exposé ci-avant, on peut finir le travail en spécialisant l'affichage dans les pages.
Ici, on supprime par exemple les liens pointant vers des routes qui ne seraient pas accessibles à un utilisateur qui ne disposerait pas du rôle adéquat.

13.5.9 Gestion fine

— Dans code d'une méthode de contrôleur :

```php
$this->denyAccessUnlessGranted('ROLE_ADMIN', null, 'Access denied!');
```

— équivalent à :

```php
if (! $this->get('security.authorization_checker')->isGranted('ROLE_ADMIN')) {
    throw $this->createAccessDeniedException('Access denied!');
}
```

Déclenche une **exception** :
— erreur 403
— ou redirection vers login

On voit ici appraître un schéma de programmation classique consistant en fait à déclencher la levée d'une exception qui correspond à une permission manquante.
Le comportement de l'application Web dépend alors d'un choix de configuration du comportement face à une telle exception : levée d'une erreur simple (403), ou bien redirection vers une page demandant l'authentification. La deuxième solution est mise en oeuvre par défaut dans Symfony.

Exceptions et codes retour

```php
try {
    // faire quelque chose qui appelle : throw
} catch (Exception $e) {
    echo 'Exception reçue : ',  $e->getMessage(), "\n";
}
```

Permet d'intercepter de façon standard les exceptions :
— `AccessDeniedException` (403)
— `NotFoundHttpException` (404)

La syntaxe des exceptions en PHP est assez similaire à cette de Java déjà supposée connue.

Take away

— Sessions
  — Cookies
  — Session Symfony
— Contrôle d'accès
  — Principes
    — Identification
    — Authentification
    — Autorisations
  — Rôles (RBAC)
  — Contrôle dans Symfony

13.6 Postface

13.6.1 Crédits illustrations et vidéos

— Illustration copie écran Basic Auth HTTP via Joel Dare

14 Séq. 9 : Interface dynamique côté navigateur

Objectifs de cette séquence

Cette séance introduit les mécanismes permettant la mise en œuvre de certaines fonctionnalités des applications Web du **côté du client**, dans le navigateur Web.
On présente les technologies JavaScript comme JQuery, qui permettent par exemple de rendre l'interface des applications plus dynamique et d'améliorer l'expérience des utilisateurs.
Jusqu'ici, l'interface des applications Web est basée sur le contenu de pages HTML produites sur le serveur, et visualisée par le navigateur.
On introduit ici les mécanismes permettant de faire tourner du côté du client Web, dans le navigateur, du code applicatif qui servira à **modifier le contenu de l'arbre DOM** de ces pages.
L'utilisateur peut donc percevoir une **interface modifiée dynamiquement**, de façon indépendante du serveur d'origine, en fonction de ses **interactions avec le navigateur** (mouvements souris, etc.).
Ce code exécuté sur le navigateur peut inclure le chargement de ressources additionnelles depuis la Toile, permettant ainsi d'intéragir avec de multiples serveurs ou de réaliser des agrégations de ressources, comme cela a été popularisé par l'émergence du Web 3.0 avec l'approche AJAX.

14.1 Aller au-delà des formulaires basiques Symfony

Cette section vise à montrer les limites des formulaires « basiques » tels qu'ils sont générés par l'assistant de Symfony `make:crud`, et à voir comment obtenir une expérience utilisateur améliorée, notamment pour les champs des attributs multi-valués.

14.1.1 Edition des OneToMany

Améliorons l'UX des formulaires…
Rappel modèle données :
— `Project` et `Todo` reliées par un *OneToMany*

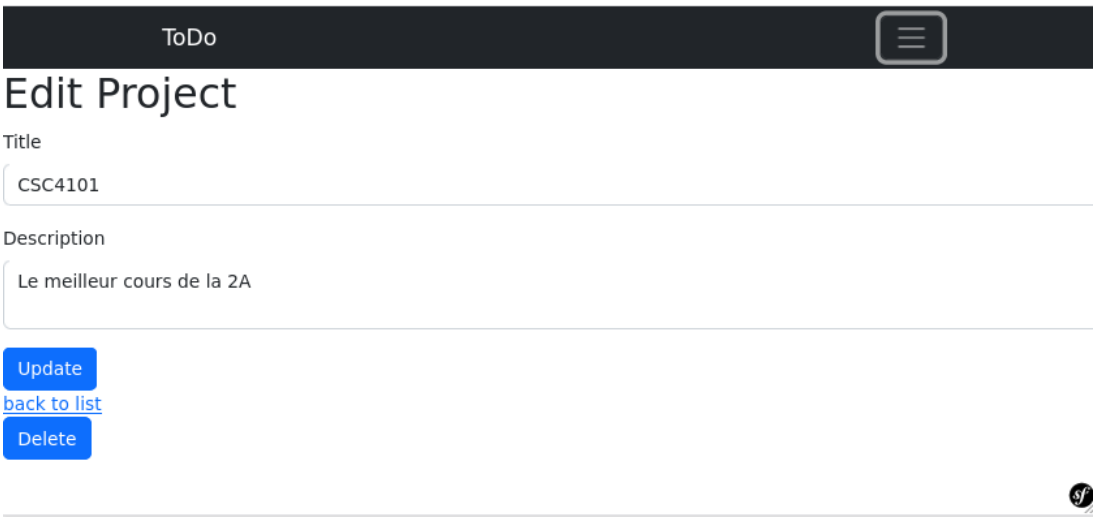


Figure 35 – version du formulaire d'édition généré par `make:crud`

Version de base

Ajout d'une association OneToMany Propriété *multi-valuée* `Project::todos`

```
->add('description')
```

Listing 17 : *form builder* pour `Project` dans `src/Form/ProjectType.php`

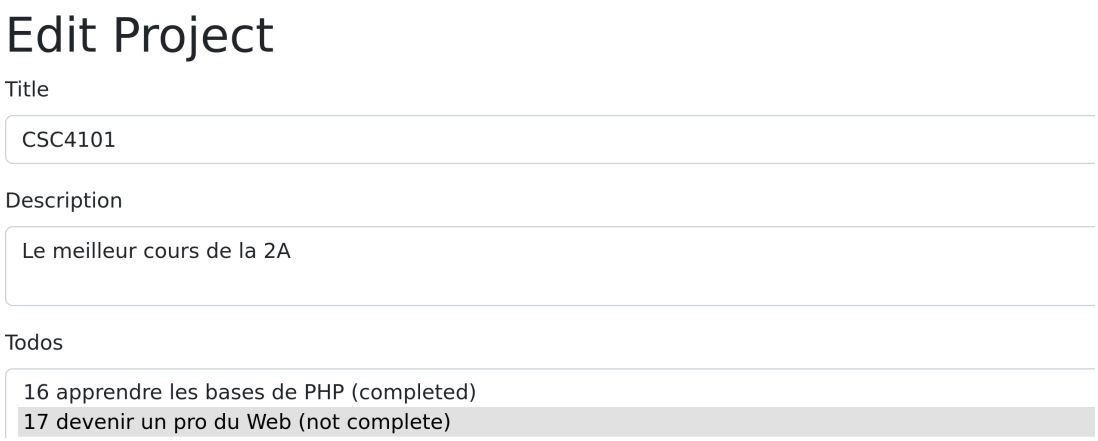


Figure 36 – version basique avec liste

Champ par défaut : liste à sélection multiple Problème : comment sélectionner plusieurs valeurs ? (*Crtl* + *clic*)

Alternative : utilisation de *checkboxes* Par défaut, génère un champs *ChoiceType* … OK, merci la doc…

> Si on ne précise pas le type du champ (second argument à `null`), comme il s'agit d'une propriété multi-valuée (côté *many* d'une OneToMany), Symfony génère un champ de formulaire de type `ChoiceType`.
> Cf. explication des options du type de champ ChoiceType : multiple et expanded.

Résultat : propriété multi-valuée via des *checkboxes*

Encore faut-il réparer la sauvegarde `'by_reference' => false` nécessaire pour que ça sauvegarde les modifications sur les tâches du projet … merci la doc…

14.1.2 Mise en forme du formulaire

Rappel : gabarit Twig généré par défaut par `make:crud`
Sous-gabarit pour le formulaire proprement dit

```php
<?php

//...

class ProjectType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title')
            ->add('description')
            ->add('todos', null, [
              'multiple' => true,
              'expanded' => true
          ])
            ;
```

Listing 18 : Version simplifiant la sélection multiple

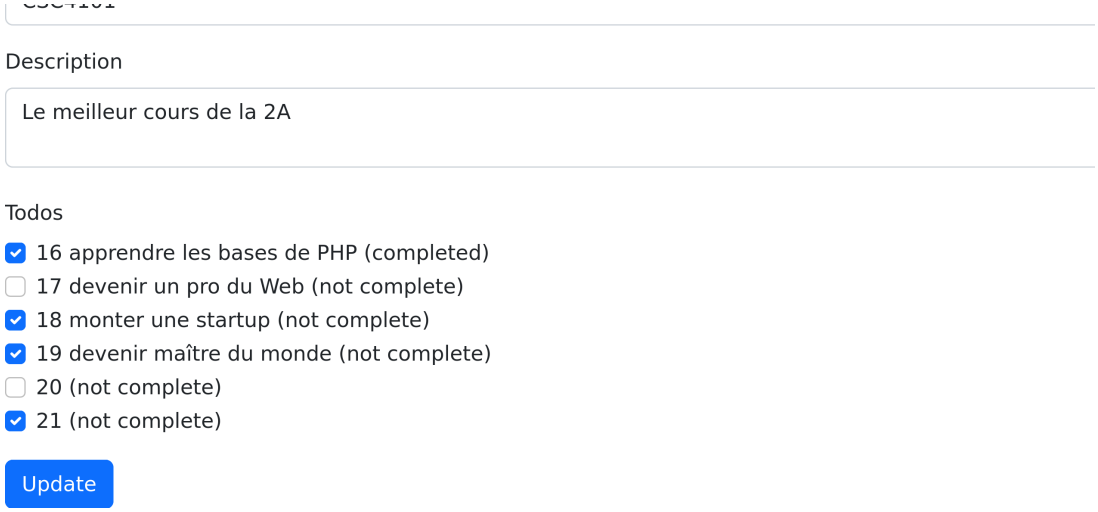


Figure 37 – sélection multiple via *checkboxes*

```php
<?php
//...
class ProjectType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('title')
            ->add('description')
            ->add('todos', null, [
              'multiple' => true,
              'expanded' => true,
              'by_reference' => false
          ])
            ;
```

Listing 19 : correction pour permettre la sauvegarde

```twig
{% extends 'base.html.twig' %}

{% block title %}Edit Project{% endblock %}

{% block body %}
    <h1>Edit Project</h1>

    {{ include('project/_form.html.twig', {'button_label': 'Update'}) }}

    <a href="{{ path('app_project_index') }}">back to list</a>

    {{ include('project/_delete_form.html.twig') }}
{% endblock %}
```

Listing 20 : gabarit `project/edit.html.twig`

```twig
{{ form_start(form) }}
    {{ form_widget(form) }}
    <button class="btn btn-primary">{{ button_label|default('Save') }}</button>
{{ form_end(form) }}
```

Listing 21 : gabarit `project/_form.html.twig`

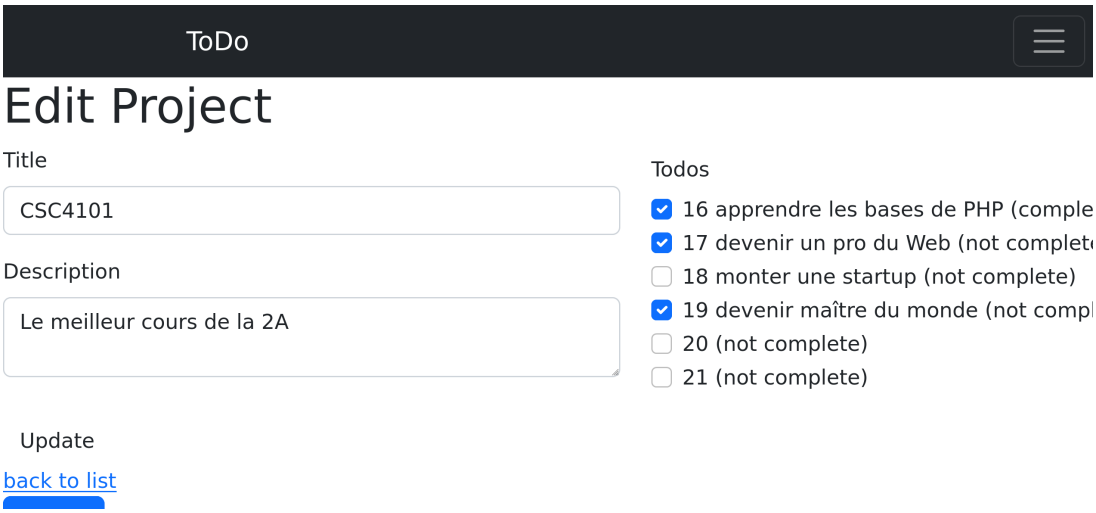


Figure 38 – séparation des champs en plusieurs colonnes

Comment faire un affichage en colonnes ? Éclater la structure des *widgets* du formulaire

```
{{ form_start(form) }}
    {{ form_errors(form) }}
    <div class="row">
        <div class="col">
            {{ form_row(form.title) }}
            {{ form_row(form.description) }}
        </div>
        <div class="col">

            {{ form_row(form.todos) }}
        </div>
    </div>
{{ form_end(form) }}
```

Listing 22 : gabarit `project/_form.html.twig` en colonnes

Cf. documentation Symfony How to Customize Form Rendering pour savoir comment remplacer `{{ form_widget(form) }}` par du code HTML plus spécifique.

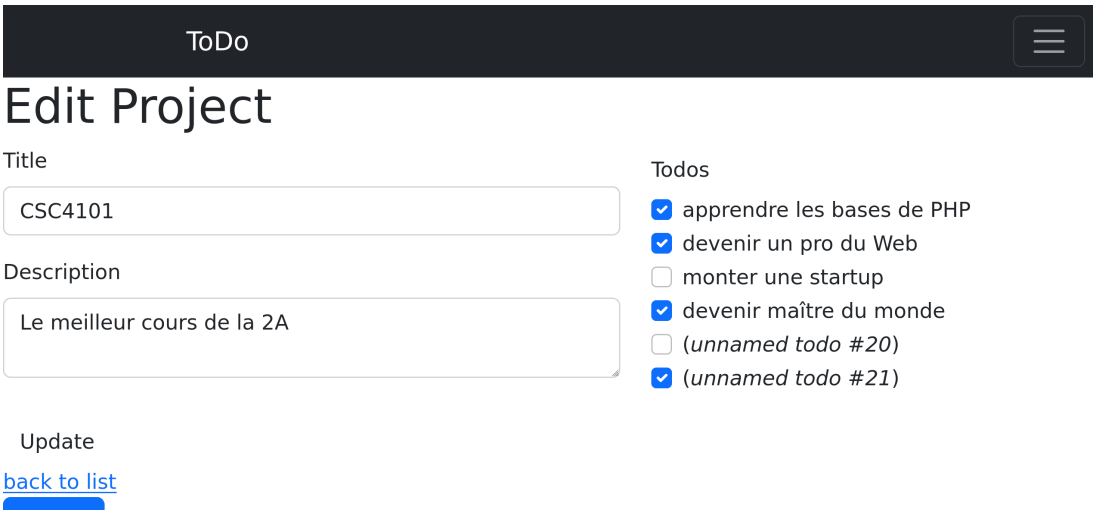


Figure 39 – affichage des tâches en HTML stylé

Améliorer l'affichage des *checkboxes* Surcharge du gabarit par défaut… c'est faisable, mais à quel coût…

Cf. documentation Symfony How to Work with Form Themes pour savoir comment surcharger des éléments de construction des fragments de gabarits, qui génèrent le code HTML pour les champs pour les collections.
Ça fonctionne, mais probablement risqué si Symfony évolue et qu'on obtient quelque chose de difficilement maintenable

Et pour le projet ? Faites-en sorte que ça fonctionne… mais on n'est pas là pour faire une vraie application, donc ne pas faire tout cela.
Des listes à sélection multiple peuvent suffire… même si *Ctrl+clic* ce n'est pas formidable.
Sinon, explosion de l'effort nécessaire.

```
{% extends 'base.html.twig' %}

{% form_theme form _self %}
{% block _project_todos_entry_widget %}
  {% set entity = form.parent.vars.choices[form.vars.value].data %}
  {% set checked = form.parent.children[form.vars.value].vars.checked %}
  <div class="form-check">
    <input type="checkbox" id="project_todos_{{ entity.id }}"
           name="project[todos][]" class="form-check-input"
           value="{{ entity.id }}"
       {% if checked %}
           checked="checked"
       {% endif %}
     />
    <label class="form-check-label" for="project_todos_{{ entity.id }}">
      {% if entity.title is empty %}(<i>unnamed todo #{{ entity.id }}</i>)
      {% else %}{{ entity.title }}
      {% endif %}
    </label>
  </div>
{% endblock %} {# _project_todos_entry_widget #}

{% block title %}Edit Project{% endblock %}

...
```

Listing 23 : gabarit `project/edit.html.twig` avec surcharge gabarit *checkboxes*

14.2 Interfaces dynamiques

Cette section présente le principe des interfaces Web dynamiques, où les pages visualisées par le navigateur Web sont modifiées directement sur le navigateur, via des programmes en JavaScript.

14.2.1 Vues générées côté serveur

Jusqu'à présent, nous avons travaillé sur un modèle d'application dont les vues reposent sur des pages HTML qui sont intégralement calculées côté serveur et où la grande majorité des interactions sont gérées par le Contrôleur dont le code est côté serveur.
Seul le CSS introduit des éléments de variabilité côté client, mais le HTML reste inchangé.

14.2.2 Vues générées côté client

« HTML dynamique »

On peut améliorer le modèle précédent pour ajouter des éléments de programme du côté client, qui permettent de faire évoluer la vue et gérer d'autres interactions, sans avoir besoin de refaire appel à une requête auprès du serveur.
Cela introduit moins de latence liée aux transferts réseau, ou de charge sur le serveur.
Principalement utilisé pour améliorer l'expérence utilisateur en mode GUI, traité au plus près des actions de l'utilisateur, sans recourir à HTTP.

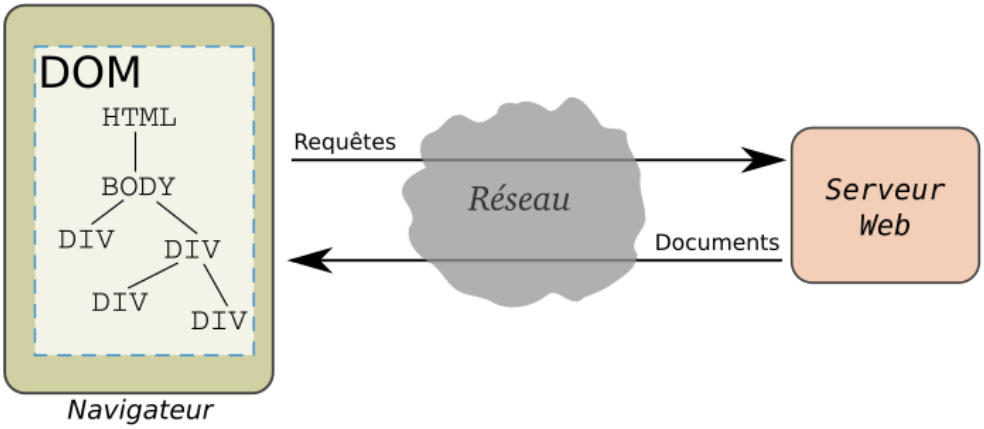


Figure 40 – Documents statiques

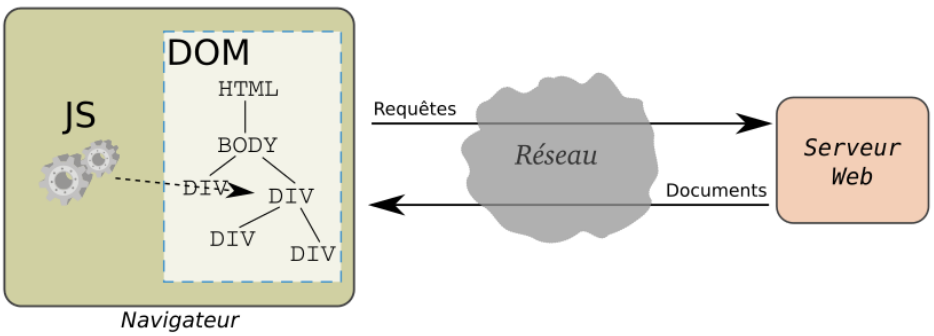


Figure 41 – Documents dynamiques avec Javascript

14.3 JavaScript

Cette section présente les caractéristiques du langage JavaScript

14.3.1 JavaScript pour applications Web

Abrégé : JS
— Langage de script orienté objet non typé
— Pilier dans la création d'applications Web
    — Principalement utilisé côté client : interprété par les navigateurs Web
    — Mais peut aussi être utilisé côté serveur (exemple *NodeJS*)

Aujourd'hui, JavaScript peut être utilisé pour programmer le *backend* côté serveur, par exemple avec NodeJS, en lieu et place de PHP.
Dans ce cours, et dans votre projet, vous utiliserez cependant Javascript uniquement pour l'exécution dans le navigateur.

14.3.2 Intérêt d'un *script* côté navigateur

— Améliorer les temps de réponse (exécution côté client, pas de latence d'envoi vers le serveur)
— Contrôler les données des formulaires avant envoi vers le serveur (éviter les envois erronés)
— Réaliser des pages animées
— Modifier le contenu, le style de la page courante en fonction des actions de l'utilisateur sur la page

14.3.3 Historique

— Développé en 1995 par Brendan Eich pour Netscape (nom d'origine *LiveScript*)
— Grand succès au début du Web (mais nombreuses incompatibilités entre navigateurs)
— Nombreuses étapes de standardisation nécessaires
    — ECMA (*European Computer Manufacturers Association*) a défini le standard ECMAScript
— L'arrivée d'**AJAX** avec le Web 2.0 replace JavaScript au premier plan (> 2005)

JavaScript devient même populaire hors du Web. Ex. programmes pour l'environnement de bureau *Gnome Shell*

14.3.4 Principes de base

— Un **noyau** (le cœur du langage)
    — des opérateurs, des structures, des objets prédéfinis (`Date`, `Array`…)
— Un ensemble d'**objets associés au navigateur**
    — fenêtre (*window*), document (*document*), formulaire (*form*), image (*image*)…
— **Programmation événementielle**
    — Capter les évènements : clics, déplacement de la souris, chargement de la page …
    — Associer des fonctions aux événements

On retrouve un style de programmation évenementielle courant dans d'autres contextes, dès qu'il y a des interfaces graphiques

14.3.5 Plate-forme

— L'interface **DOM** (*Document Object Model*) standardise les méthodes pour accéder par objets aux documents
— Nombreuses bibliothèques (*frameworks*) aident à programmer.
Ex : Dojo, Rialto, **Angular**, **JQuery**, Yui…
— « *bytecode* » pour programmer à un plus haut niveau, dans un langage compilé en JS : TypeScript, …
— **JSON** (*JavaScript Object Notation*) utilise la notation des objets JavaScript pour transmettre de l'information structurée

```
{
  "nationalTeam":"Norway",
  "achievements":[
      "12 Olympic Medals",
      "9 World Championships",
      "Winningest Winter Olympian",
      "Greatest Nordic Skier"
    ],
  "name":"Bjoern Daehlie",
  "age":41,
  "gender":"Male"
}
```

De nouveaux langages, comme TypeScript ou Elm, introduisent des paradigmes de programmation plus sécurisés et améliorant la qualité du code (typage strict, fonctionnel, etc.), tout en restant compatibles avec la plate-forme ECMAScript, car compilés en JavaScript, et permettant donc un déploiement sur tout navigateur.

14.3.6 Utilisation

Intégration dans pages HTML
— Code JS contenu ou référencé dans balises `<script>` et `</script>`
  — Soit directement :

```
<head>
    <script>
        code javascript (fonctions, variables...)
    </script>
</head>
```

  — Soit dans un fichier séparé (ou URL externe) :

```
<head>
    <script src="monprog.js"></script>
</head>
```

Appel des fonctions JavaScript
— à partir d'évènements dans les balises

```
<input type="button"
       onClick="alert('clic sur bouton');">
```

— à partir d'un lien

```
<a href="javascript:affiche(menu);">
```

— par d'autres fonctions

```
function hello() {
  alert("hello");
}
```

14.3.7 Gestion d'événements

— Chaque événement a un traitement par défaut
— On peut associer une fonction à un évènement, pour l'exécuter **avant** traitement par défaut
  — Si retourne faux (*false*), alors traitement par défaut pas exécuté
  — Si retourne vrai (*true*), alors traitement par défaut exécuté ensuite
— Association fonction à évènement utilise un mot-clef de type « *onEvent* » (`onClick`, `onLoad`, `onChange`, `onMouseOver`, `onMouseOut`, `onKeyDown`…)

14.3.8 Exemples de réflexes

— clic sur bouton :

```
<input type="button" name="surface" value="Surface"
       onClick="Test(all);">
```

— déplacement souris au-dessus :

```
<a href="menu.html" onMouseOver="affiche();">
```

— fin de chargement du document :

```
<body onLoad="init(1);">
```

Exemples évènements / balises

Evénements	Balises	Fonctions
onBlur	*	désélection
onChange	*idem*	modification
onFocus	*idem*	sélection
onSelect	password, text, textarea	sélection de valeur
onClick	a href, button, checkbox, radio, reset, submit	clic
onMouseover	a href, area	souris pointe
onLoad	body	chargement page
onSubmit	form	soumission

Cf. liste complète d'événements

Exemple de gestion d'événements

```
<html>
    <head>
        <script language="JavaScript">
            function maFonction() {
                alert('Vous avez cliqué sur le bouton GO');
            }
            function autreFonction() {
                alert('Cliqué sur le lien "Cliquer ici"');
            }
        </script>
    </head>
    <body>
      <ol>
        <li>Appel sur évènement :
            <input type="button" value="GO" onClick="maFonction();">
        </li>
        <li>Appel sur un lien :
            <a href="javascript:autreFonction();">Cliquer ici</a>
        </li>
      </ol>
      <a href="javascript:history.go(-1);">Retour</a>
    </body>
</html>
```

14.3.9 Objets JavaScript

— JavaScript a des types primitifs et des objets
— Types primitifs créés par affectation directe
— Utilisation d'objets
  — Instanciation par un constructeur
  — Syntaxe *à la java* :
    — Accès aux attributs : `monObjet.attributs`
    — Accès aux méthodes : `monObjet.méthode()`
  — Modèle objet beaucoup moins strict que Java

Exemple

```
// Création d'une variable
var etienne = {
    nom : "etienne";
    branche : "marketing";
}

// Création d'un objet par constructeur
function Employe (nom,branche) {
    this.nom = nom;
    this.branche = branche;
    this.fullDescription = function () {
        return this.nom + " " + this.branche;
    };
}

var marc=new Employe("marc","informatique");

marc.fullDescription(); // appel de méthodes
```

Objets prédéfinis 1/2 Les objets du noyau
— `Object` : création d'un objet
— `Function` : création d'une fonction
— `Array` : création d'un tableau
— `String` : création d'une chaîne de caractères
— `Number` : création d'un nombre
— `Date` : création d'une date
— `Math` : création d'un objet mathématique
— etc.

Objets prédéfinis 2/2 Les objets du navigateur :
— fenêtre,
— document,
— formulaires
— …

Hiérarchie d'objets présents en mémoire dans le navigateur
— Un navigateur (objet *navigator*),
  — Contient une fenêtre (*window*),
    — Qui contient son document HTML (*document*),
      — ses formulaires (*form[]*), ses images (*image[]*), ses liens (*link[]*)…
— Un formulaire contient
  — des boutons (*button[]*),
  — des listes déroulantes (*select[]*)
    — qui contiennent des options (*option[]*)
— etc.

Attributs et méthodes
— Les attributs d'un objet correspondent aux attributs des balises HTML (par exemple *height*, *width*, *name*, *src* … pour l'objet *image*)

— Les méthodes correspondent aux fonctionnalités du navigateur (par exemple *back*, *forward*, *go* pour l'historique (*history*))
— Certains attributs et méthodes sont communs à un ensemble d'éléments

Se rapporter aux références pour connaître les méthodes et attributs d'un élément, par exemple Référence W3School

14.3.10 Mise au point

Délicate avec les langages interprétés car :

— Les erreurs n'apparaissent qu'à l'éxécution
— Par défaut, les erreurs ne sont pas affichées

Outil : console JS

— **console de développement** du navigateur
  — Possibilités variables selon les navigateurs
  — Indication des erreurs de syntaxe
  — Inspection du contenu des variables
  — Débogueur, points d'arrêt, affichage des variables
— Affichage d'informations sur la console :
  — Affichage des erreurs de syntaxe éventuelles
  — Par programme avec la fonction *consoleLog(« message » + variable)*

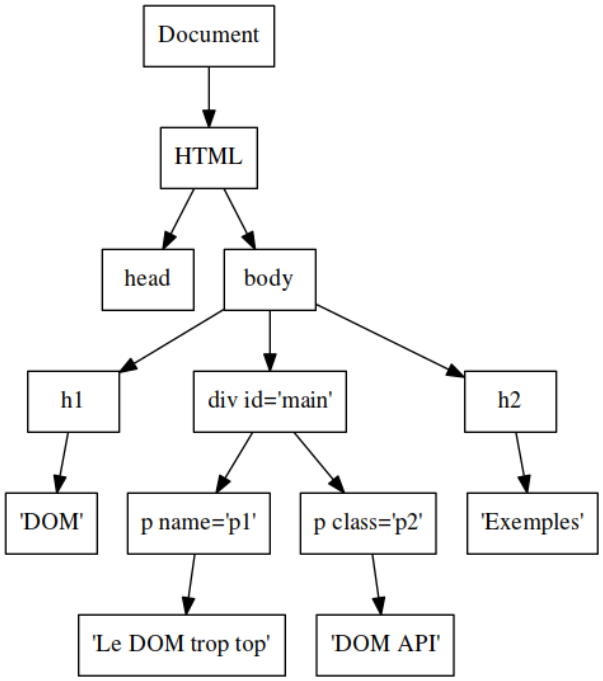
14.4 Manipulation du DOM

Cette section rappelle le principe du *Document Object Model* (DOM), déjà vu en lien avec CSS.
Cette fois, on s'intéresse surtout à l'interface de programmation standardisée de manipulation du DOM.
Elle permet aux programmes de manipuler le contenu des pages Web en JavaScript, de façon portable quelque soit le navigateur.

14.4.1 DOM

Document Object Model (Modèle Objet du Document)

— Modèle de la structure des documents HTML
— Interface de programmation standardisée par le W3C (API DOM)
— Implémentée par les navigateurs avec plus ou moins d'extensions
— Références :
  — spécifications du W3C,
  — le DOM Gecko (Mozilla)



Exemple d'arbre DOM

```
<html>
  <head>
    ...
  </head>
  <body>
    <h1>DOM</h1>
    <div id='main'>
      <p name="p1">Le DOM trop top</p>
      <p class="p2">DOM API</p>
    </div>
    <h2>Exemples</h2>
  </body>
</html>
```

14.4.2 Manipulation avec JavaScript

— Le DOM offre un ensemble de méthodes de parcours de l'arbre pour accéder et/ou modifier son contenu
  — ajouter des éléments
  — supprimer des éléments
  — ajouter et/ou modifier des attributs
  — etc.

14.4.3 Exemple d'utilisation du DOM

— Le DOM comporte de nombreuses méthodes pour accéder à un élément, de différentes façons :
  — par son *identifiant* : `<div id="intro">`
  — par sa *position* dans l'arbre : (fils, parent…)
  — par son *type* : ensemble des paragraphes, des niveaux `h2`, …

14.4.4 Utilité

— Modifier l'apparence
— Par exemple, changer dynamiquement le lien (attribut `class`) avec des définitions de feuilles de style : interfaces réactives
— Exemple : masquage, démasquage des slides dans reveal.js, utilisé pour les diapos projetées en CM

Pour l'utilisateur, on dirait que des actions graphiques sont effectuées à l'écran, mais en fait, le code JavaScript effectue des modifications sur l'arbre DOM, et le moteur de rendu du navigateur redessine la page, en appliquant le CSS, ce qui crée les différences d'affichage.
JavaScript permet également de faire des choses au niveau de tout ce que sait faire le navigateur, en vrai langage de programmation, indépendant d'une utilité pour le Web.
On peut ainsi aller très loin, pour réaliser des choses très pointues : émuler un système d'exploitation, virtualisé dans le navigateur, etc.

14.5 Framework JavaScript jQuery

Cette section présente le *framework* de développement JQuery qui facilite le développement en JavaScript.

14.5.1 Avantages d'un *FrameWork*

Faciliter la tâche des développeurs.
— Il existe un grand nombre de cadriciels (*frameworks*) JS
— Lequel choisir ? éviter d'en utiliser plusieurs dans une même aplication
— Critères de choix : les performances, les fonctionnalités, le poids, la commmunauté, etc.
— Il existe une page de comparaison sur Wikipédia : `https://en.wikipedia.org/wiki/Comparison_of_JavaScript-based_web_frameworks`

14.5.2 jQuery

`https://jquery.com/`
— Bibliothèque des plus populaires et des plus utilisées
— Utilisable avec tous les navigateurs pour un résultat identique (avant la standardisation)
Un peu ancien, et il y a sûrement mieux aujourd'hui… mais montré ici pour illustrer, pas comme recommendation ultime.

Caractéristiques
— Slogan : « *write less, do more* » : écrire moins pour faire plus
— Facilité d'apprentissage
— Simplification de la programmation JavaScript
— Encapsulation des actions courantes dans des méthodes, ce qui réduit le code à écrire
— Simplification de certaines tâches comme la manipulation du DOM ou la programmation AJAX

Fonctionnalités
— Accéder aux objets HTML/DOM
— Manipuler les styles CSS
— Gérer les évènements
— Réaliser des effets et des animations
— Programmer avec AJAX
— etc
jQuery dispose aussi d'extensions sous forme de *plugins*

Chargement de JQuery

— téléchargé depuis `jQuery.com` et inclus en local sur notre site

```
<head>
  <script src="jquery-3.6.0.min.js"></script>
</head>
```

— ou référencer sur serveur de contenu externe, p. ex. sur CDN (*Content Delivery Network*) jQuery

```
<head>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js">
  </script>
</head>
```

Attention : traçage des utilisateurs

Remarque : il existe 2 versions : compressée (.min) ou non.

Programmation « événementielle »

1. Le navigateur (lui-même), ou les interactions de l'utilisateur déclenchent des **événements**
2. les événement déclenchent l'exécution d'actions sur les objets du programme (dont le DOM).

Structure générale des programmes

1. **Sélectionner** des éléments du DOM HTML : générer des objets javascript qui les représentent
2. Installer des gestionnaires d'événements / **actions** sur ces éléments : ces gestionnaires sont des **fonctions**

Syntaxe de base

— Syntaxe de base jQuery :

```
$(selector).action(...)
```

— La construction `$()` permet d'implanter un traitement avec jQuery
— Le sélecteur (`selector`) permet de sélectionner un (ou plusieurs) élément(s) HTML
— L'action (`action`) est exécutée sur chaque élément sélectionné

Actions

— Actions immédiates : changer l'état d'un élément du DOM. Exemple :

```
$(this).hide() // cacher l'élément courant
```

— Mise en place de **réflexes** (*callbacks*) sur des événements (actions différées) :

```
$('#bouton1').click(function() {
    // actions à faire lors d'un clic sur le bouton
    //...
});
```

Point d'entrée du code

— Pour éviter que nos actions jQuery s'exécutent sur des objets non encore chargés, jQuery permet de ne démarrer le tout qu'une fois reçu l'évènement « *document ready* » :

```
$(document).ready(function(){

   // code jQuery...

});
```

Exemples sélecteurs JQuery

l'élément courant `$(this)`

le nom de la balise `$("p")`, pour tous éléments `<p>`

l'identifiant `$("#id1")` : l'élément ayant l'identifiant `id1`

la classe `$(".test")` tous éléments ayant attribut `class test`

Même syntaxe que les sélecteurs CSS

14.6 AJAX

Cette section présente le modèle « AJAX » qui permet de mettre en œuvre des applications JavaScript qui interagissent avec des services Web externes pour récupérer des données, de façon asynchrone.

14.6.1 AJAX

Asynchronous JavaScript And XML
— Ne plus lire l'acronyme littéralement (~~XML~~)
— AJAX signifie qu'on intègre différentes technologies
  — centrées sur Javascript
  — **requêtes HTTP asynchrones** vers serveurs
— Traitement de **données supplémentaires** récupérées sur serveur(s)
— Exemple : construction progressive d'un formulaire en fonction des réponses de l'utilisateur.

Terme introduit en 2005. Plus lié à la technologie XML aujourd'hui, mais plutôt à HTML et JSON, par exemple… mais le nom de la fonction n'a pas été changé

14.6.2 Classe `XMLHttpRequest` (XHR) de JavaScript

— Élément clef de la technologie AJAX
— Créé en 1999 par Microsoft comme objet ActiveX
— Intégré comme objet JavaScript par Mozilla.
— Création : `new XMLHttpRequest();`
— Ensuite, utilisation comme tout objet via ses
  — propriétés
  — et/ou ses méthodes
— Alternative avec jQuery (voir plus loin)

Encore une fois, pas spécifique à XML : on peut charger ce qu'on veut sur le serveur cible.

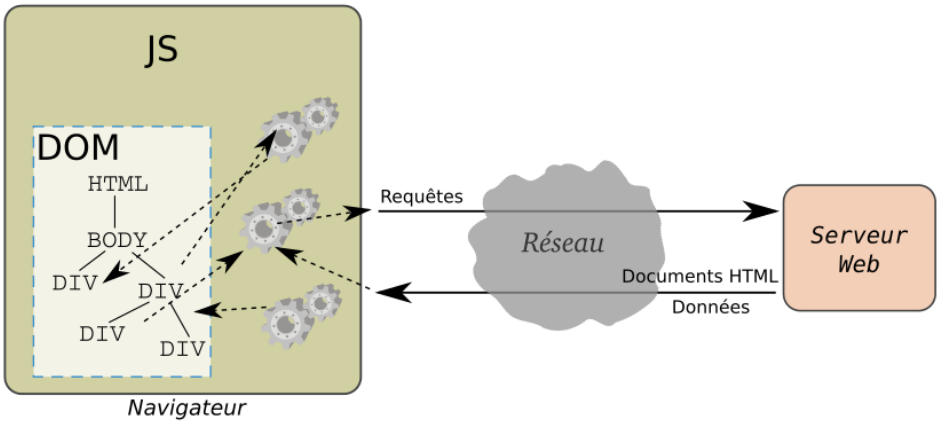
14.6.3 Principe de fonctionnement

— Un évènement (clic, saisie de texte…) provoque l'exécution d'une fonction JavaScript
— Cette fonction

1. instancie un objet `XMLHttpRequest`
2. transmet sa requête HTTP à un serveur (`GET` d'un document ou `POST` de données, ou …)
3. récupère la réponse lorsque celle-ci est disponible (asynchrone)
4. met à jour la partie concernée de la page courante (un champ de saisie, une division…)

Naturellement, la fonction JavaScript peut
- appliquer différents traitements à la réponse obtenue
- modifier des attributs de styles de type CSS pour certains objets du document HTML.
- etc

14.6.4 Sous-requêtes HTTP



Les possibilité sont très nombreuses : à partir d'une page principale HTML, renvoyée une fois pour toutes par le serveur, et accompagnée d'un programme Javascript, on va pouvoir faire tourner une application complexe, qui se comporte comme un client HTTP vis-à-vis de services Web divers, et agrégeant les données renvoyées par différents serveurs HTTP, les traitant et les visualisant.
Cette architecture est de plus en plus populaire.
Elle déporte la gestion des Vues et des Contrôleur de l'application très fortement du côté du client, en comparaison de l'architecture traditionnelle qu'on voit dans le modèle étudié en cours jusqu'ici.

14.6.5 Utilisation d'AJAX avec jQuery

- L'utilisation de jQuery permet de faciliter l'écriture de programmes AJAX avec la fonction :
  `$.ajax()`
- En paramètres la description de la requête (objet au format JSON) :
  - `url` : définit l'URL de la requête
  - `type` : précise la méthode (`GET` ou `POST`)
  - `data` : précise les données à envoyer si `POST`
  - `success` : définit traitement à effectuer si requête réussit (fonction)
  - `error` : définit traitement à effectuer si requête échoue (fonction)

Exemple

```
//...
$.ajax( {
    url: "test.php",
    error: function(resultat, statut, erreur) {
        // ...
    }
    success: function(resultat, statut) {
        // ...
    }
} );
```

14.6.6 Exemple d'application

Construction de formulaires dynamiques avec Symfony

Take away

— JavaScript (programmation événementielle)
— manipulation du DOM
— JQuery
    — Sélecteurs similaires à ceux de CSS
— AJAX : requêtes asynchrones

Postface

Crédits illustrations et vidéos

— Logo *JQuery* (source : `https://brand.jquery.org/logos/`)

15 Séq. 9 : Gestion de la sécurité, des erreurs

Objectifs de cette séquence

Cette section présente les enjeux généraux de sécurité qui s'imposent aux concepteurs d'applications Web, avec un catalogue rapide de quelques problèmes courants.

15.1 Sécurité

15.1.1 Objectifs

— Éviter que les données ne soient corrompues
— Garantir le bon fonctionnement du service
— Garantir que les permissions accordées aux utilisateurs soient respectées
Danger : toute entrée de donnée

15.1.2 Exemple d'attaques

Injection SQL
— **Objectif** : garantir que les requêtes SQL sont sans effet de bord
— Nettoyer les données servant à constituer ces requêtes SQL
Cf. `http://php.net/manual/fr/security.database.sql-injection.php`
— code PHP

```php
<?php

  $id = $_GET['id'];
  $sql = "SELECT username
FROM users
WHERE id = $id";
...
```

— requête envoyée à :
`http://localhost/?id=-1%20UNION%20SELECT%20password%20FROM%20users%20where%20id=1`
— argument de la requête GET :
`id : -1 UNION SELECT password FROM users where id=1`
Exploits of a Mom (xkcd)

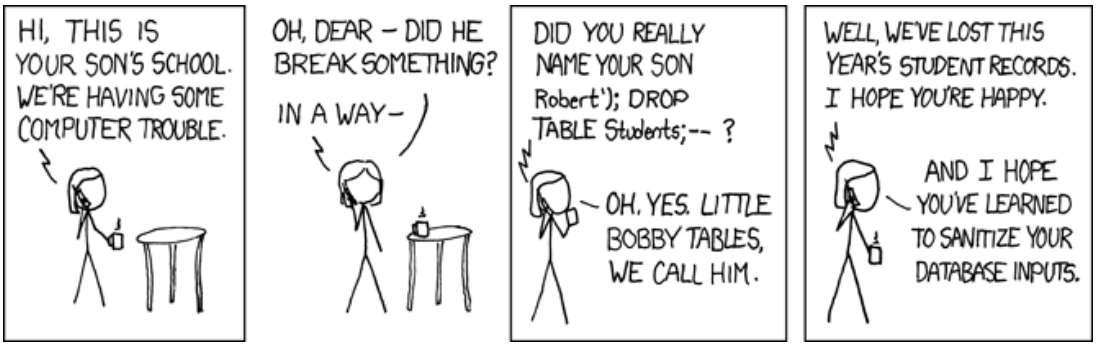
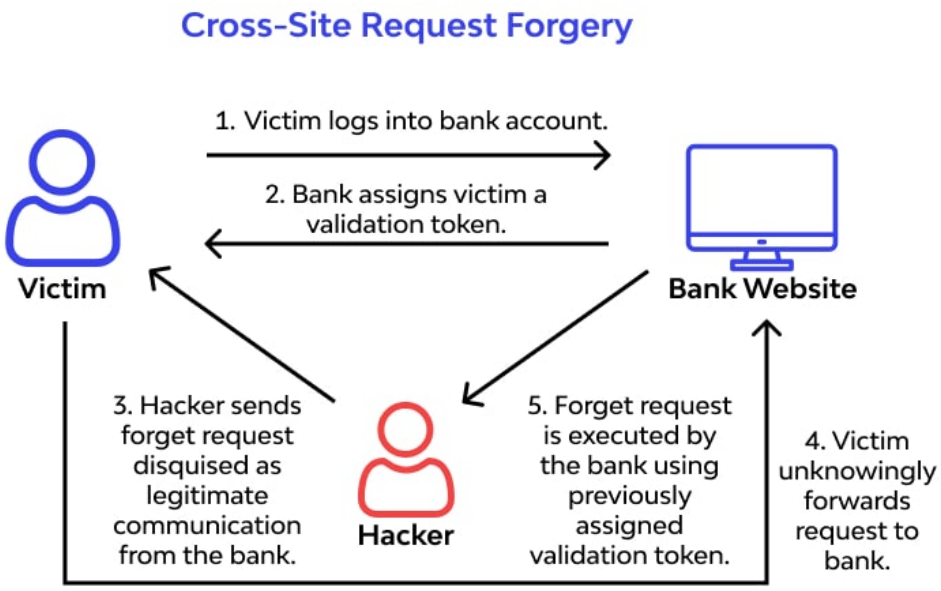


Figure 42 – *Exploits of a Mom (xkcd)*

Danger : le PHP codé comme montré au début de la séance 5 où on manipule directement les données reçues, type `_GET[]` ou `_POST[]`.
Si ces données sont injectées dans la construction de requêtes en base, on peut amener le serveur à intégrer du contenu malicieux dans la page, ou dans la base.
Ne surtout pas passer les données telles-quelles à la base. Doctrine nous évite celà.

***Cross Site Request Forgery* (CSRF)** Induire l'exécution de transition dans le graphe d'états de l'application en jouant des requêtes à l'insu de l'utilisateur.
`https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)`



Source : `https://www.wallarm.com/what/what-is-cross-site-request-forgery` Exemple de transaction sur site banque : virement de 100 euros à destination d'*olivier* :

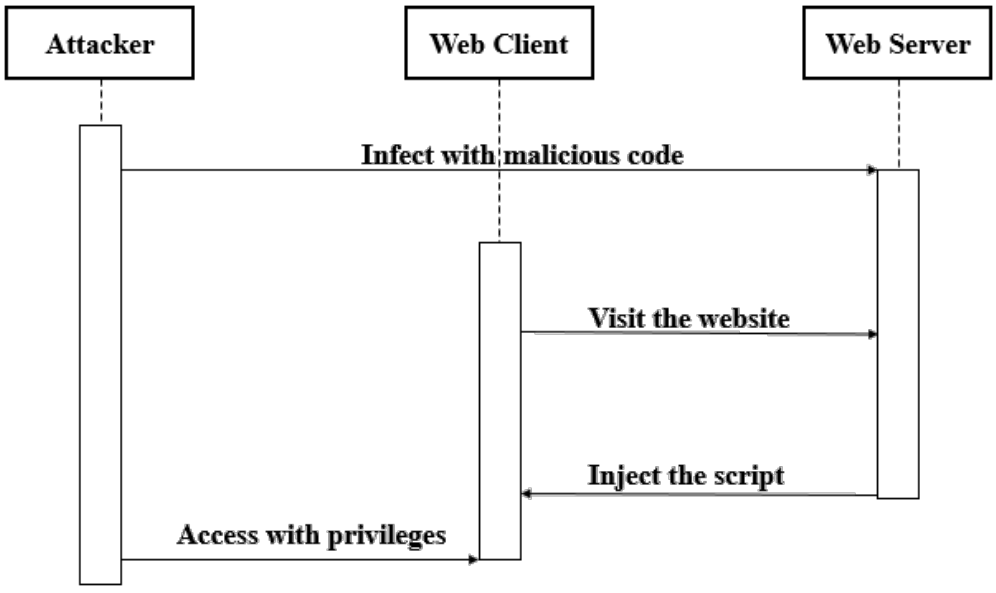
```
POST http://bank.com/transfer.do HTTP/1.1

acct=olivier&amount=100
```

Objectif : faire exécuter ce formulaire à l'insu de l'utilisateur :

```
<form action="http://bank.com/transfer.do" method="POST">
  <input type="hidden" name="acct" value="philippe"/>
  <input type="hidden" name="amount" value="100000"/>
  <input type="submit" value="View my pictures"/>
</form>
```

***Cross-site scripting* (XSS)**



Source : Michel Bakni, CC BY-SA 4.0, via Wikimedia Commons
— Donne accès aux données de l'application à un script tiers
— cookies
— jetons session
`https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`
— Exemple de page :

```php
<?php

  echo '<html>';
  // ...
  echo $_POST["name"];
```

— Requête POST : injection de `name` qui vaut
`name=%3Cscript%3Ealert%2842%29;%3C/script%3E`
— Résultat :

```
<script>alert(42);</script>
```

Tout dépend où se trouve le `echo` …
Solution possible :

```php
<?php

  echo '<html>';
  // ...
  echo htmlentities($_POST["name"]);
```

Résultat :

```
<html>
  ...
  &lt;script&gt;alert(42);&lt;/script&gt;
```

Normalement, c'est plus sûr…

Deuxième danger, les données sont injectées dans la construction du contenu HTML des pages visualisées par le navigateur, qui peut ainsi être amené, par exemple, à exécuter un script JS dans le contexte courant de l'utilisateur.
Ici, le système de gabarits va nous aider. On en construit pas le HTML à la main, et les données sont nettoyées.

15.1.3 Avantage du framework

Le *framework* permet de gérer automatiquement certaines précautions

Sanitization
— **Objectif** : ne pas transmettre au Modèle des données avec des caractères indésirés
— Supprime de manière automatique les caractères indésirés,
— Champs des formulaires associés à un type de donnée pour restreindre les plages de valeur

En plus, le développeur peut intégrer des règles de validation des données pour éviter d'avoir des données incohérentes par rapport aux règles de gestion de l'application
Cf. `https://symfony.com/doc/current/validation.html`

Doctrine On ne manipule pas SQL directement.

Gabarits On ne manipule pas HTML directement

Sécurité des formulaires Protection CSRF
Objectif : éviter d'arriver sur la soumission d'un formulaire directement, contrôler les transitions dans le graphe HATEOS

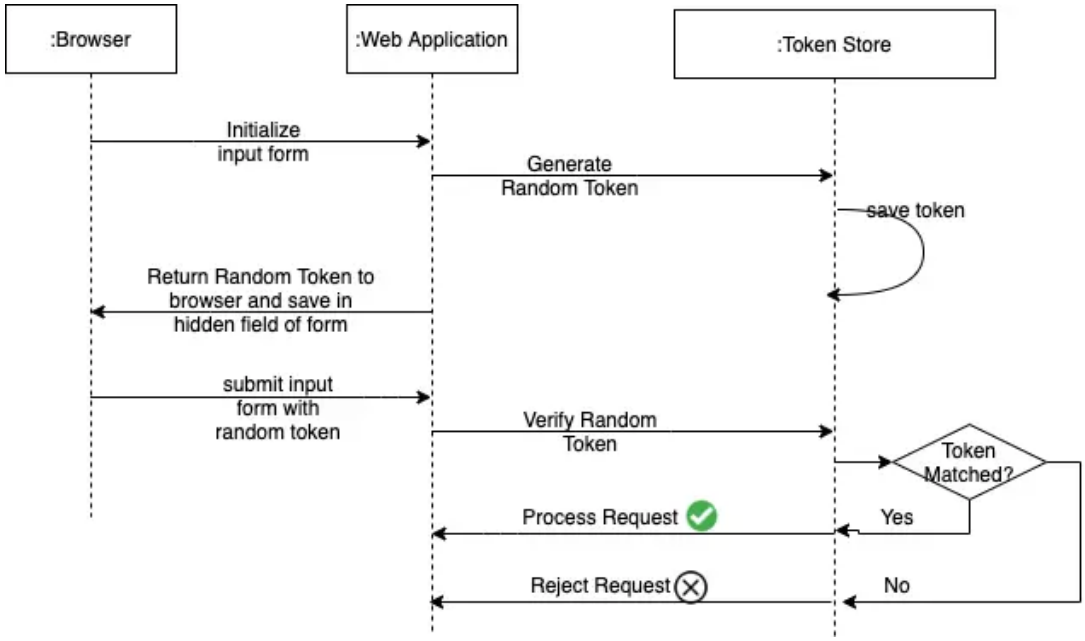


Figure 43 – Identifying Legitimate Requests with an CSRF Token

Source : `https://reflectoring.io/complete-guide-to-csrf/`
— Contrôle que la gestion de la soumission d'un formulaire suit immédiatement l'affichage du formulaire
— Paramètre caché généré de manière automatique (`_token` en Symfony)
— Associé à une mémorisation du paramètre côté serveur (dans la session)
Construction du formulaire :

1. stockage dans la session

```
Session Attributes

_csrf/post : "RfofubWU_auc33Mef-EdbvVlHOBh5J1561Z9rJ_FJ5I"
```

2. génération du formulaire envoyé au client

```
<form>
  ...
  <input type="hidden" id="post__token"
         name="post[_token]"
         value="RfofubWU_auc33Mef-EdbvVlHOBh5J1561Z9rJ_FJ5I" />
</form>
```

Réception de la requête POST :
— données du POST

```
[
  "title" => "Lorem Ipsum"
  "summary" => ""
  "content" => "Lorem Ipsum"
  "publishedAt" => "2017-11-26T20:07:34+01:00"
  "tags" => ""
  "_token" => "RfofubWU_auc33Mef-EdbvVlHOBh5J1561Z9rJ_FJ5I"
]
```

— code du contrôleur (dans `isValid()`) :

```
if ($this->isCsrfTokenValid('token_id', $submittedToken)) {
        // ...
}
```

Attention, Symfony intègre du CSRF dans certains formulaires, mais on doit le gérer manuellement lors de certains opérations avancées.
Par exemple, c'est le cas dans le code qui est généré dans les contrôleurs et gabarits obtenus avec `make:crud`. On y gère des formulaires particuliers pour la suppression des entités, via de fausses méthodes « DELETE » simulées par des POSTS. On doit y gérer manuellement l'ajout d'un jeton CSRF dans le gabarit (type `_delete_form.html.twig`) et le vérifier ensuite dans les méthodes `delete()` des contrôleurs.

15.1.4 Inconvénient du framework

Sécurité des dépendances ?
S'abonner à des listes de notification de vulnérabilités, pour être notifié des alertes (CVE).
— pour PHP : `https://wiki.php.net/cve`
— pour le reste : `https://github.com/FriendsOfPHP/security-advisories`

15.1.5 Sécurité des utilisateurs

— Protéger les données des usagers :
  — argent
  — infos personnelles (RGPD)
  — vie privée
  — réputation
— Lutter contre surveillance
  — chiffrement TLS (HTTPS)
  — accessibilité via TOR (domaine en `.onion`)

Exemple : le profil d'un usager ne doit pas être accessibles par d'autres.
RGPD : Réglement Général (Européen) sur la Protection des Données
Contraintes : garder des traces du consentement au traitement des données personnelles.

15.1.6 Contrôle d'accès

déjà vu

15.1.7 HTTPS *everywhere*

— BD : `https://howhttps.works/`
— Cf. LetsEncrypt, pour générer des certificats TLS.

15.1.8 Sécurité des applis Web

Approfondir :
— `https://phptherightway.com/#security`
— Guidelines Web Security Mozilla
— OWASP Web Security Testing Guide
— Recommandations pour la sécurisation des sites web ANSSI

Ressource vraiment intéressante : *Open Web Application Security Project* (OWASP)

15.2 Gestion des erreurs

Cette section aborde la gestion des erreurs, notamment à travers l'utilisation des exceptions.

15.2.1 Code de réponse

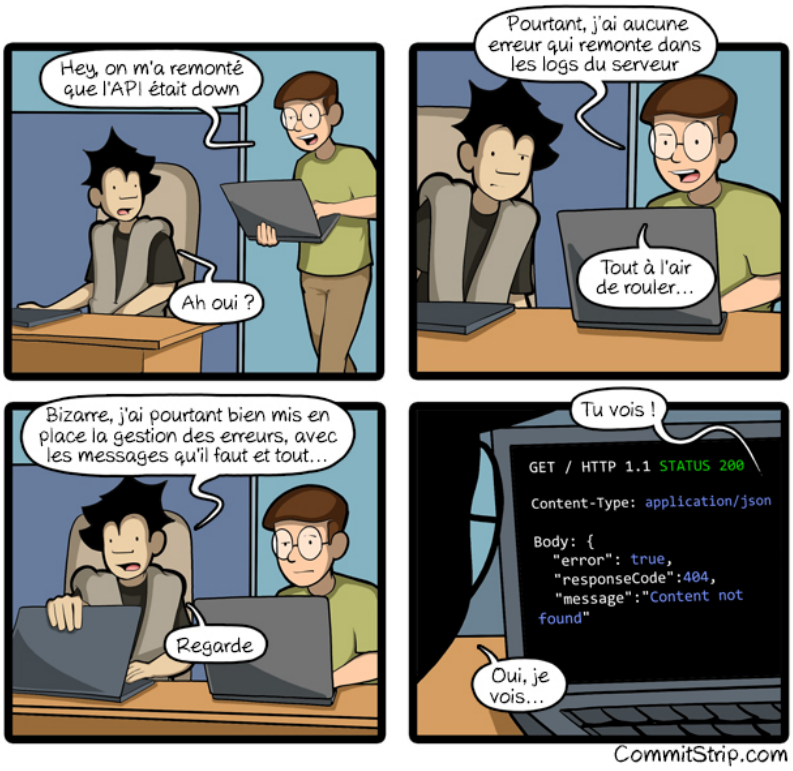


Figure 44 – *HTTP Headers FTW (CommitStrip)*

HTTP Headers FTW par CommitStrip

En cas de problème, ne pas renvoyer 200 dans la réponse HTTP, contrairement à ce qui est fait dans cet exemple humoristique.

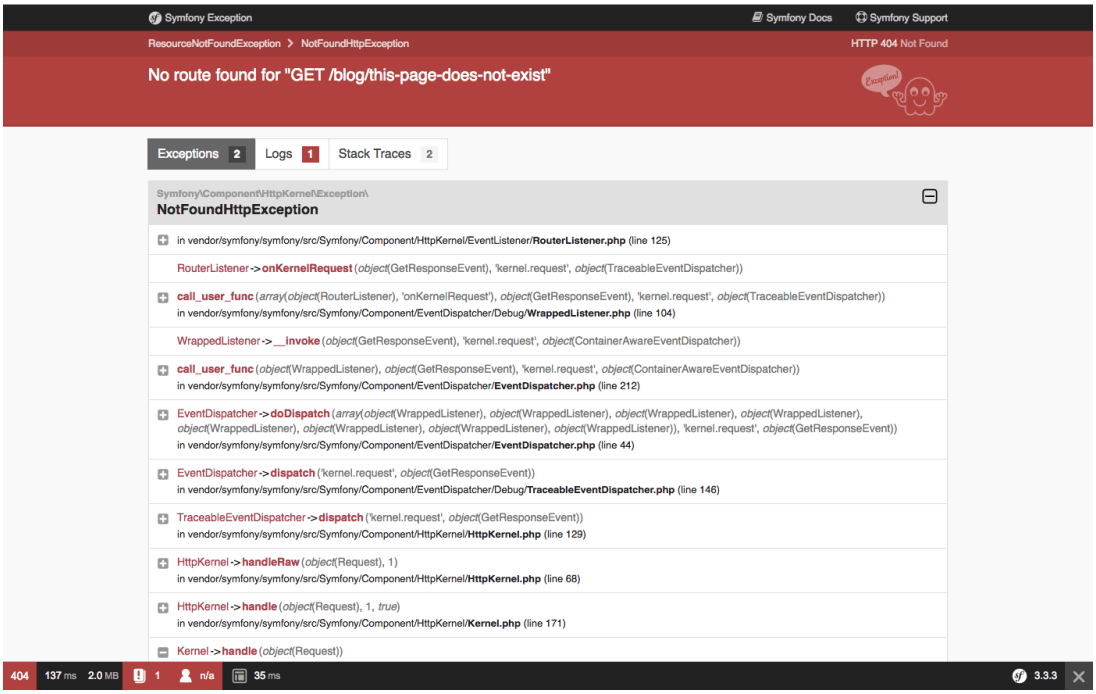
15.2.2 Exceptions

```php
throw new OutOfBoundsException("not good");
```

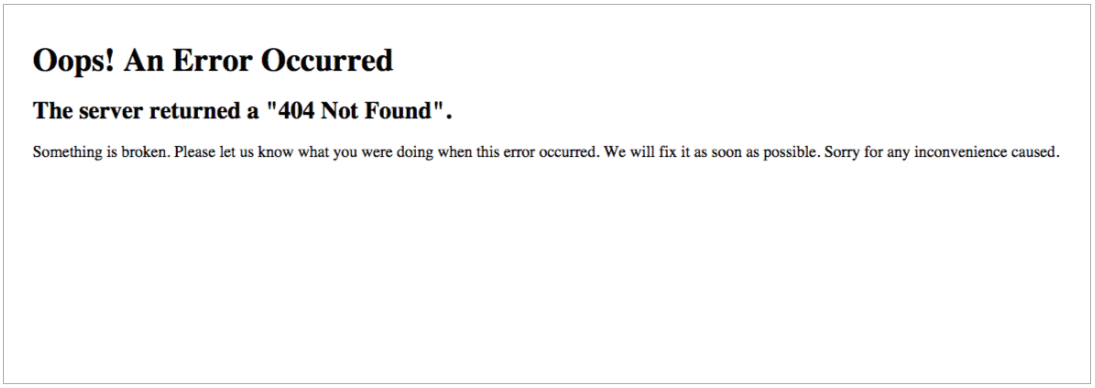
```php
try {
    ...
} catch (RequestException $e) {
    ...
}
```

Certains d'entre-vous approfondiront la programmation des exceptions dans le module CSC4102.

15.2.3 Environnement de dév



15.2.4 Environnement de prod ?



15.2.5 Pages d'affichage des erreurs dans les gabarits

Cf. How to Customize Error Pages.

15.2.6 Déployer

Checklists : `https://symfony.com/doc/current/deployment.html`

15.3 Bugs, Qualité

Cette section évoque l'enjeu de la qualité du code, qui permet de garantir que l'application aura le fonctionnement attendu, et qu'ainsi, on évitera des problèmes de corruptions des données du modèle, ou de fuite d'infos, par exemple.

15.3.1 Éviter les bugs

— Coder
— **Tester**
PHPUnit

15.3.2 Tests manuels ?

Vraiment ?

15.3.3 Tests unitaires

— Tests des méthodes des objets du modèle, en dehors du contexte Web.
— PHPUnit (très similaire à JUnit)

15.3.4 Tests « système »

Tester de bout en bout, systématiquement :

1. Fabriquer une « requête » semblable à celle entrant depuis un client HTTP : accès à une route avec ses arguments
2. Traiter
3. Vérifier la réponse :
    — code de retour HTTP
    — présence de données dans le contenu (HTML)

Cf. `https://symfony.com/doc/current/testing.html`

```php
public function testPageIsSuccessful($url)
{
        $client = self::createClient();
        $client->request('GET', $url);
        $this->assertTrue($client->getResponse()->isSuccessful());
}
```

Take Away

— sécurité de l'application :
    — données entrées ou fournies
— sécurité des utilisateurs
— qualité du code
— le framework fait les choses « bien », et c'est tant mieux !
— ne plus programmer en PHP « basique » comme autrefois

Postface

Crédits illustrations et vidéos

— Exploits of a Mom by xkcd - Creative Commons Attribution-NonCommercial 2.5 License.
— HTTP Headers FTW par CommitStrip - utilisation non-commerciale autorisée
— Cross-site scripting attack sequence diagram par Michel Bakni, CC BY-SA 4.0, via Wikimedia Commons
— WHat is cross-site request forgery - Wallarm
— Identifying Legitimate Requests with an CSRF Token - Reflectoring

16 Séq. 10 : Évolution des architectures applicatives

16.1 Architectures modernes

Cette section présente succinctement des architectures plus récentes que l'architecture « classique » détaillée jusqu'alors, comme par exemple les technologies de conception d'applications dans une page unique (*Single Page Application*).

16.1.1 Déploiement de l'intelligence vers le client

— Rendre le client (navigateur / mobile) de plus en plus actif, intelligent
— Standardisation des navigateurs (croisons les doigts)
— Écosystème JS plus fort
  — NodeJS
  — Angular, React, Vue.js, Ember, etc.

16.1.2 *Single Page Application*

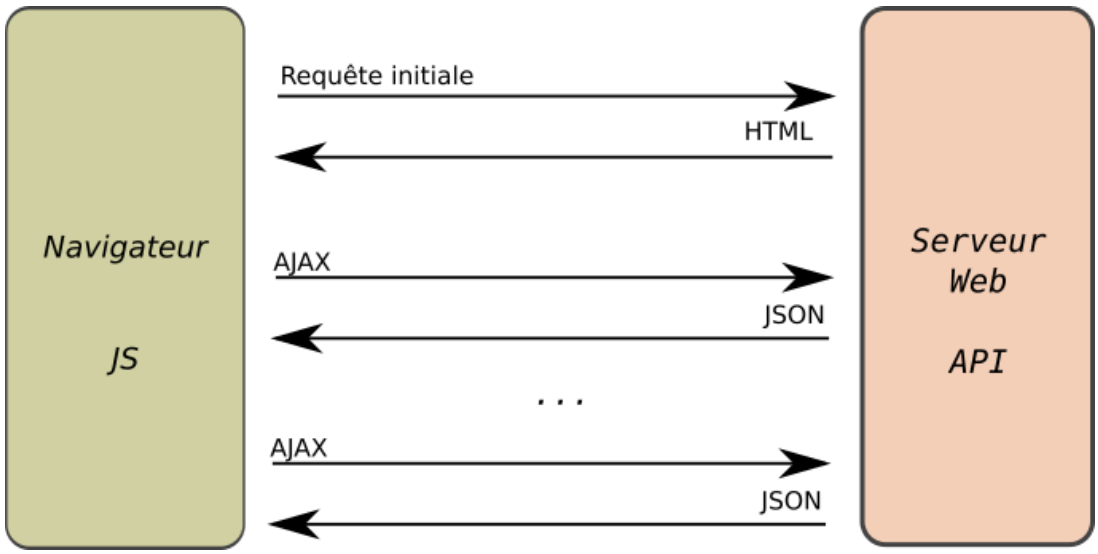


Figure 45 – Requêtes AJAX principalement

Le serveur n'a plus qu'à gérer une API Web (*Application Programming Interface*), sans aucune couche de présentation HTML.
Une seule « page Web » est chargée depuis un serveur, qui initialise toute une application en chargeant et lançant du Javascript, qui s'exécute ensuite complètement côté client.

16.1.3 MVC sur le client

Toute la logique des états de l'application, affichés en différentes pages à l'utilisateur, est entièrement opérée via le code côté client.

16.1.4 Progressive Web Apps

PWA
— *Fast* !
— Mobile : résoud les pertes de connexion
— Fonctionne grâce aux *Service Workers*
Cf. Applications web progressives sur MDN

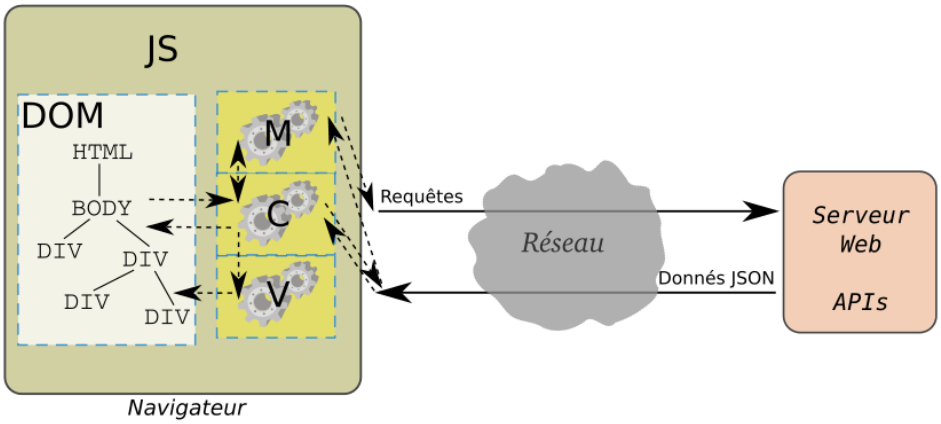


Figure 46 – MVC sur le client

Service Workers *A Service Worker is just a javascript file that runs in the background.*
— Script (JS) tourne en tâche de fond
  — Pas lié à une page Web
  — Gestion des API du navigateur (connectivité, notifications…)
  — *thread* séparée
  — partagé entre des tabs
— « *Proxy* » entre l'application et le Web
Cf. Service Worker API sur MDN

Principe de cache du *proxy*
— Interception requêtes réseau
— Cache appel à API

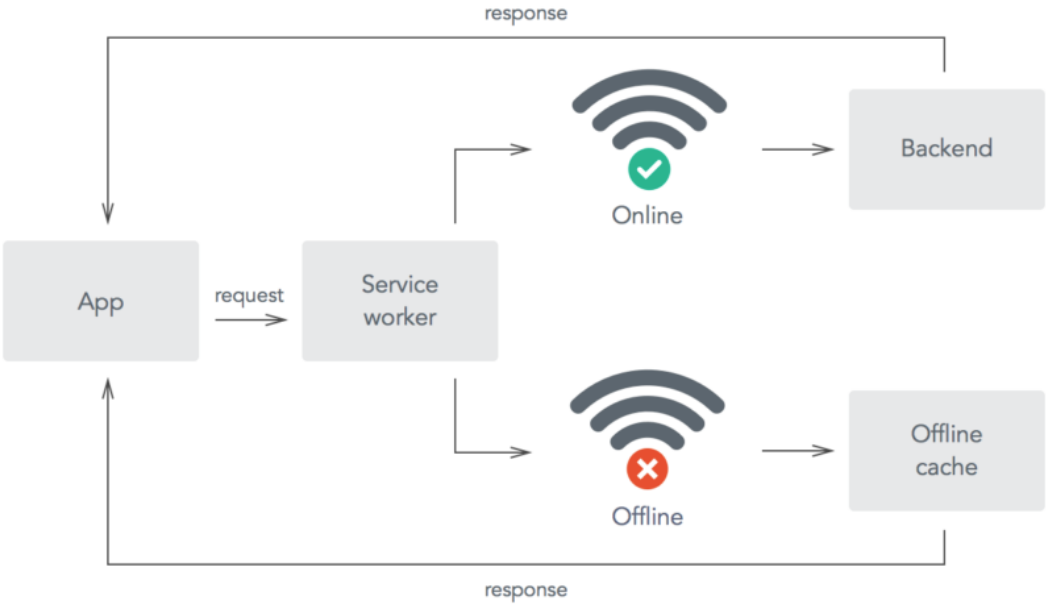


Figure 47 – Principe de cache du *Service Worker*

Cf. `https://developer.mozilla.org/fr/docs/Web/API/Service_Worker_API` pour plus de détails sur les *Service Workers*.

16.2 Nouveaux outils de développement

16.2.1 Nouveaux langages

— ~~Javascript~~ (et tous ses frameworks)
— TypeScript : « *TypeScript is JavaScript with syntax for types* »
— CofeeScript : « *CoffeeScript is a little language that compiles into JavaScript* »
— Flutter « *Build apps for any screen* », développé avec Dart : « *Dart is a client-optimized language for fast apps on any platform* »
— …

16.2.2 Nouveaux paradigmes

— Programmation JS -> fonctionnelle, pour meilleure qualité
— Exemple elm, basé sur Haskell

16.2.3 WebAssembly

aka `wasm`
Langage bas niveau (« assembleur du Web ») plus efficace que JS
Plus sur : `https://madewithwebassembly.com/`

16.3 Décentralisation

16.3.1 Web3

Crypto, BlockChain…
Euh…

16.3.2 Fédiverse

— Mastodon : `https://joinmastodon.org/`
— PixelFed : `https://pixelfed.org/`
— PeerTube
— Mobilizon
— … (more at `https://fediverse.party/`)
protocole clé : `ActivityPub`

16.3.3 Solid

Le futur du Web selon Tim Berners Lee `https://solidproject.org/`
aka le vrai Web 3.0 (ou plus)

16.3.4 …

Et plein d'autres choses…
to be continued

Postface

Crédits illustrations et vidéos

— Diagramme *Service Workers* par David Novicki : "Work It" featuring Service Workers

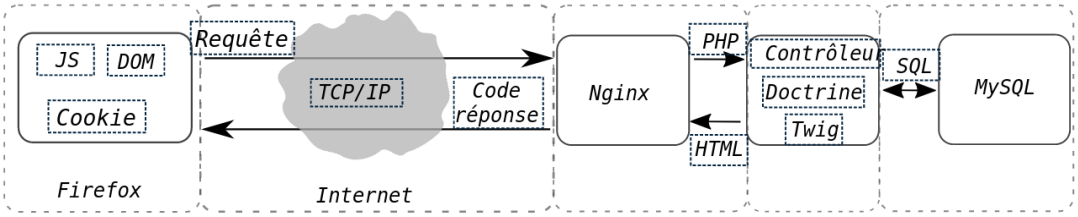
17 Conclusion

17.1 Architecture Appli Web (”classique”)

Cette section récapitule les grands principes de l’architecture « classique » qui a été étudiée pendant le cours, où l’essentiel de l’intelligence de l’application est déployé du côté du serveur Web.

17.1.1 Ce que nous avons vu

Paysage des technos



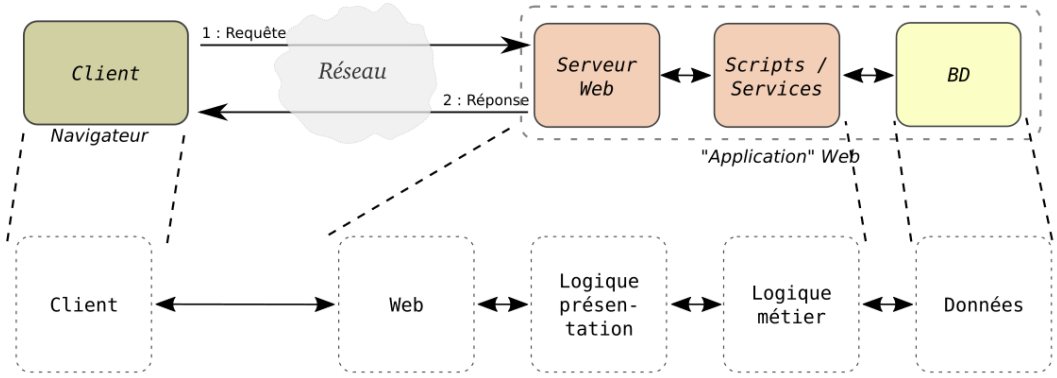


Figure 48 – Modèle en plusieurs couches (*tiers*)

Multi-couches

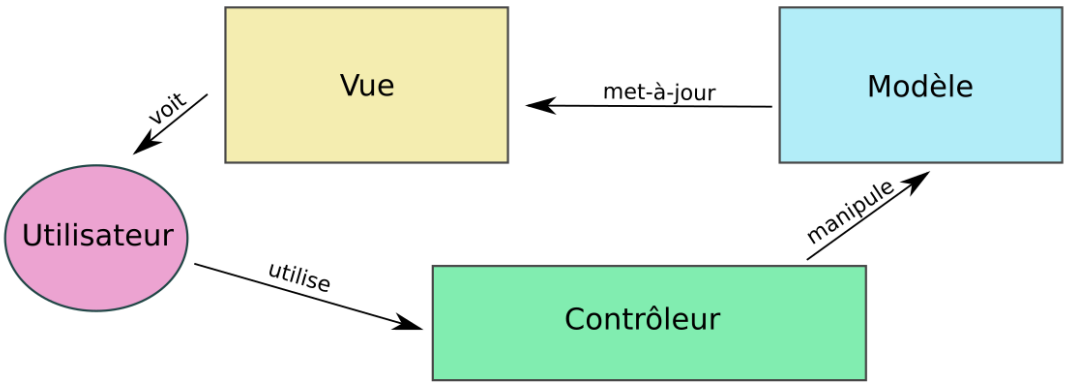


Figure 49 – Modèle - Vues - Contrôleur

MVC appliqué aux interfaces sur le Web

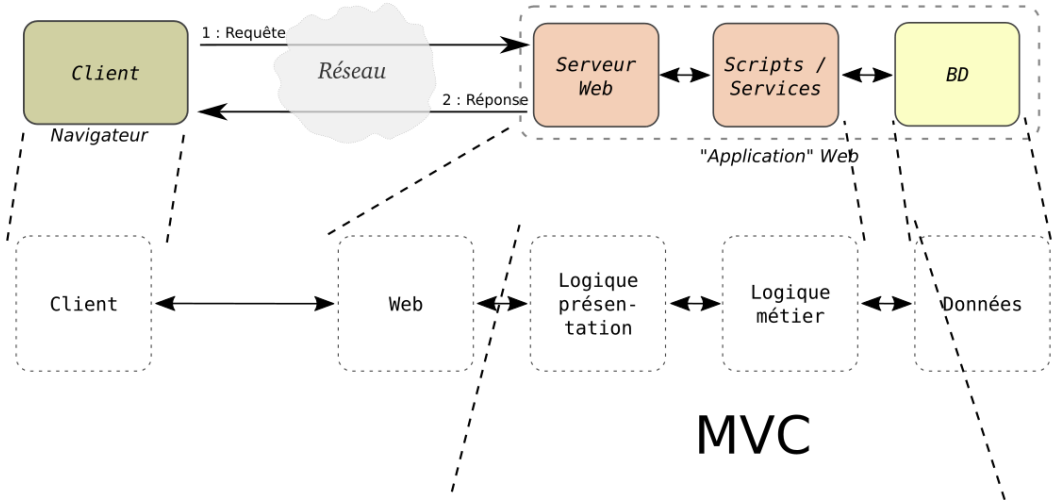


Figure 50 – « MVC dans les couches Web »

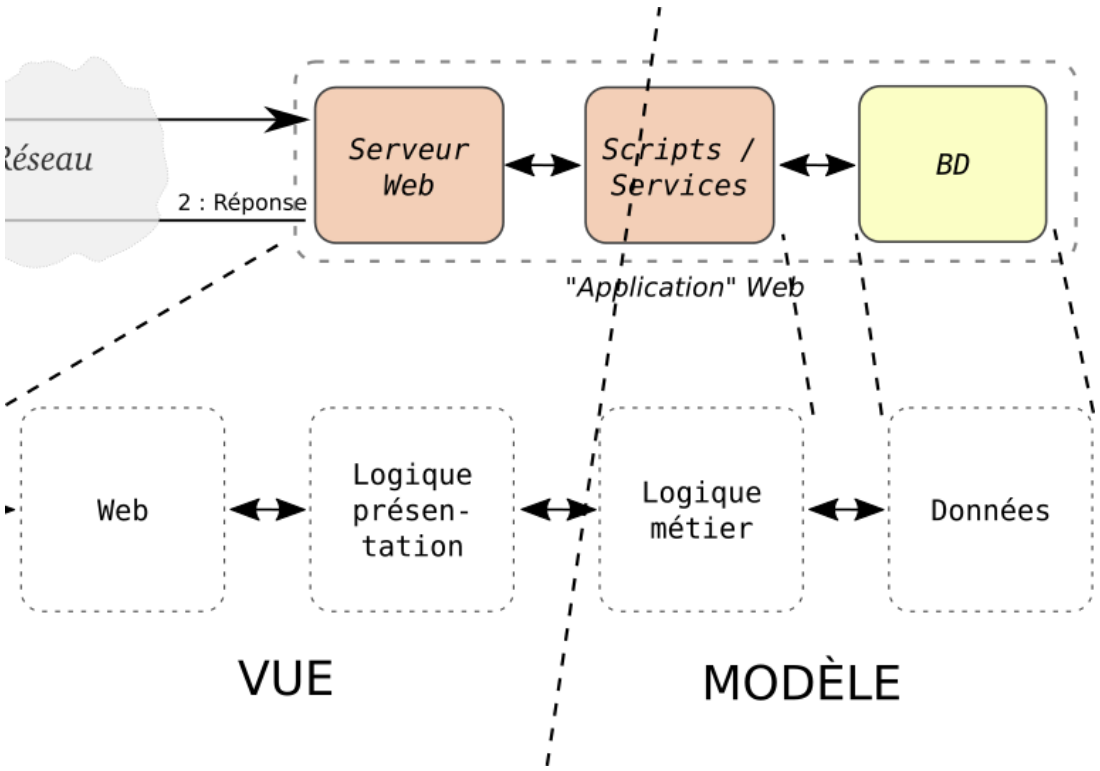


Figure 51 – « Vue et Modèle »

MVC principalement sur le serveur Contrôleur : Codé en PHP objet avec Symfony
Avantages :
— client (navigateur) stupide,
— formulaires câblés avec modèle de données Doctrine
— cohérence des interactions HATEOS gérées par le serveur
HATEOS : Hypermedia As The Engine Of application States

17.1.2 Client de moins en moins stupide

HTML statique

Formulaires standards

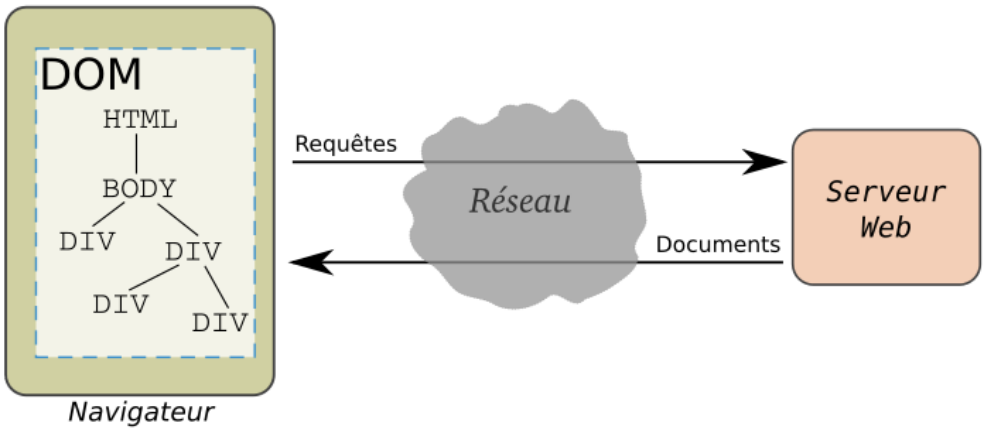


Figure 52 – « Documents statiques »

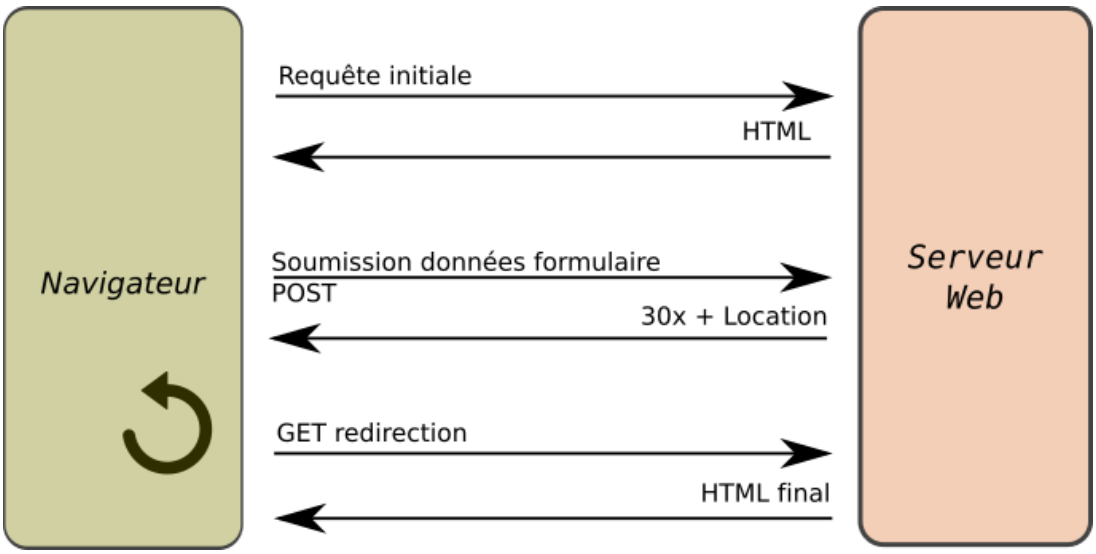


Figure 53 – « Comportement des formulaires standard »

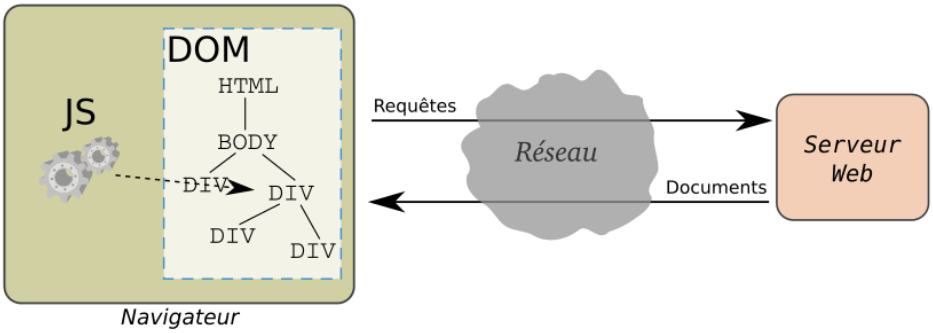


Figure 54 – « DHTML »

HTML dynamique

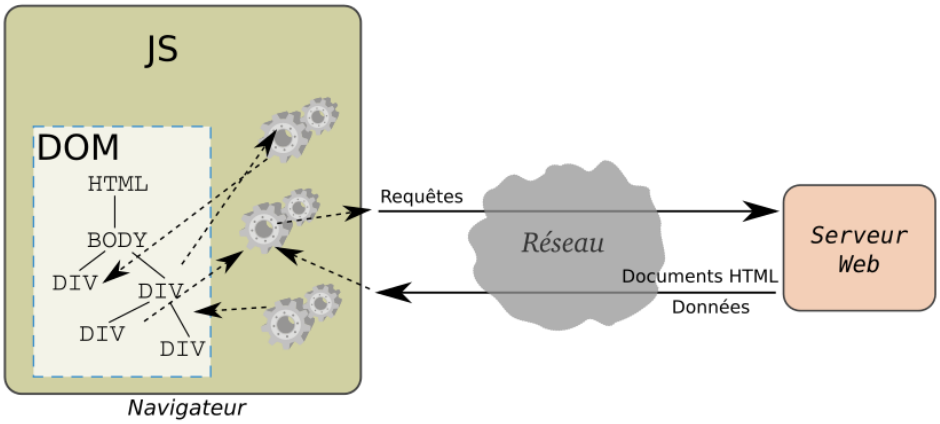


Figure 55 – « AJAX »

AJAX avec JQuery

17.1.3 Programmation côté serveur

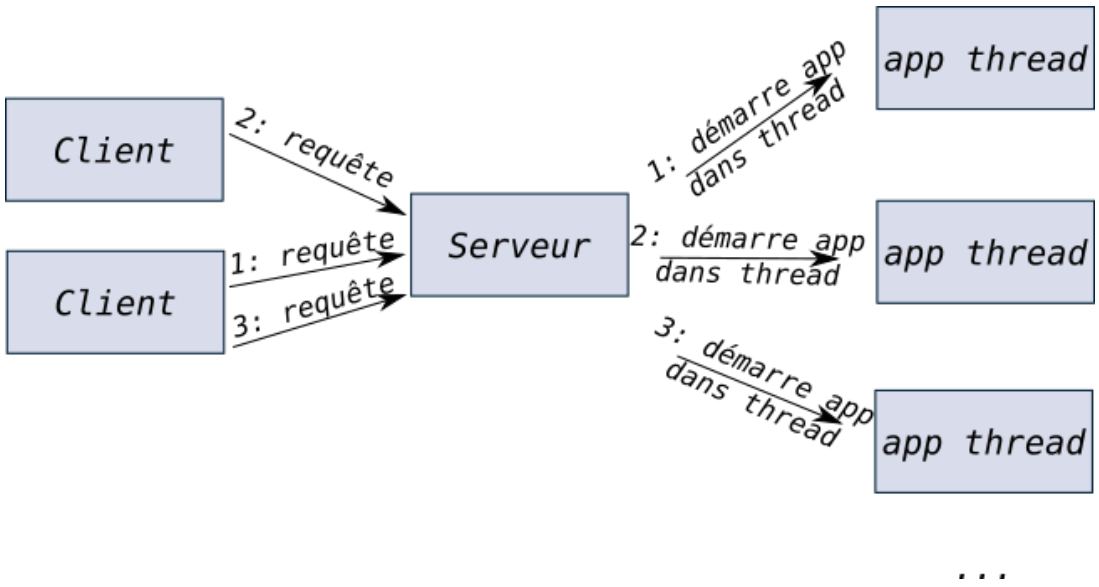
Programmation événementielle

— déclenchement programmes sur serveur : **requêtes HTTP**
  — méthodes des classes PHP des Contrôleurs Symfony, réflexes cablés sur des routes

D'ailleurs, aussi sur le client :

— déclenchements de programmes dans le navigateur en Javascript, quand **événements interface** utilisateur du navigateur (+ horloge, …)
  — fonctions JS, câblés sur sélecteurs (DOM)

Concurrence d'exécution sur le serveur



17.1.4 Exécution de l'application

Dev vs Prod

env de tests serveur de tests local `symfony server:start`

production code déployé derrière Nginx, PHP-FPM, Cloud PaaS, etc.

Durée de vie
— Serveur Web : tourne en permanence
— Application :
  — redémarrée à chaque requête entrante
  — terminée après chaque réponse
  — plusieurs fils d'exécution parallèles : requêtes HTTP de différents clients HTTP

Contexte Contexte (variables PHP) propre à chaque client HTTP
— contexte de la requête (arguments + en-têtes + données)
— **session** :
  — propre à un client / utilisateur
  — liée aux *cookie* …

Cohérence
— dans le temps entre deux requêtes du même clients (HATEOS) : **sessions**
— entre clients : **base de données** partagée, dans SGBDR

17.1.5 Composants Symfony

— Routeur + Contrôleur
— Doctrine (attributs `ORM`, *repository*)
— *MapEntity* (raccourci accès au chargement des objets depuis BD dans méthodes contrôleurs)
— Gabarits Twig
— `*Type` (*FormBuilder* + validation données)
— Exceptions
— `Security`
  — rôles RBAC, permissions
— Session
  — Cookies

Mais aussi
— *Data fixtures*
— Générateurs de code

— Barre d'outils de mise au point
— …

17.1.6 Notions clés architecture « classiques »

— Client **s** - serveur **s** avec HTTP
— Multi-couches :
  — Présentation HTML + CSS,
  — Traitements : PHP objet,
  — Modèle : ORM + SGBD
— Code essentiellement sur serveur (PHP + Symfony)
— Programmation événementielle (MVC)
— REST (HATEOS, …) + Sessions

17.2 Architecture système

Cette section aborde le déploiement d'une application Web, chose que l'on n'a pas le temps d'expérimenter dans les TP de ce cours.

17.2.1 Architecture Web générale

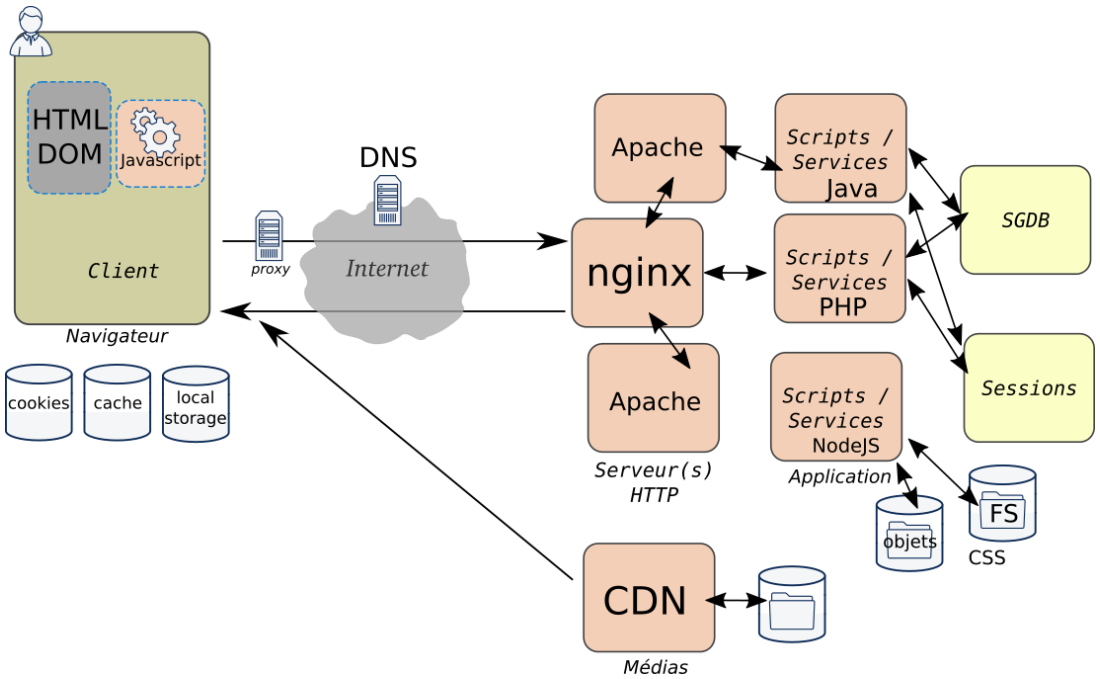


Figure 56 – « Composants utilisés »

17.2.2 Application Web + services additionnels

Source : Web Architecture 101 de Jonathan Fulton

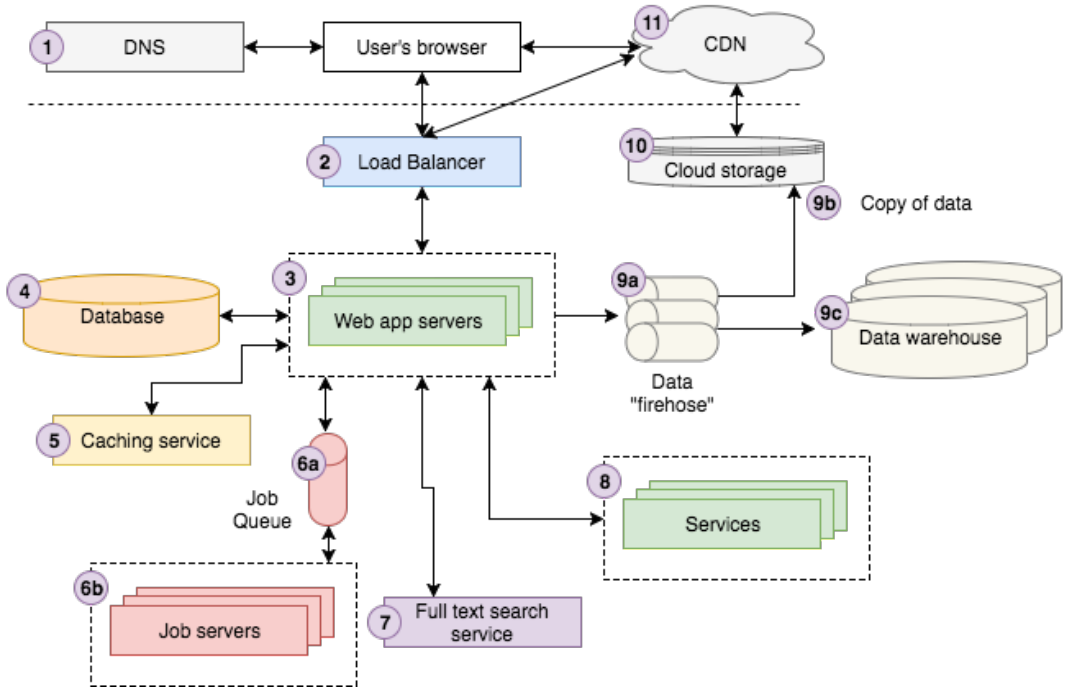
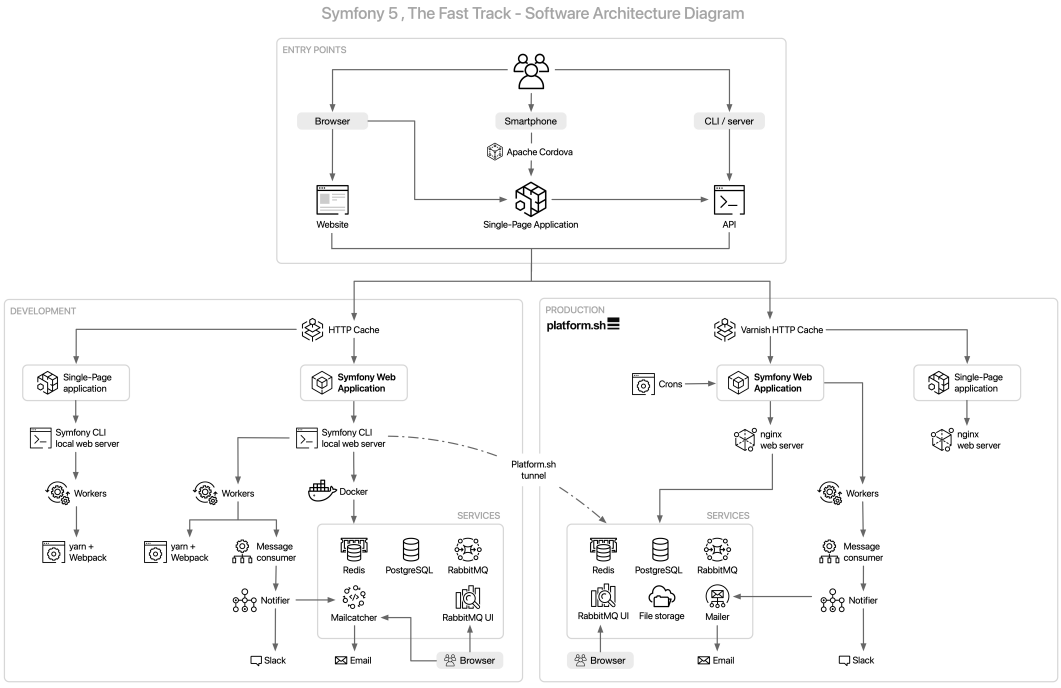


Figure 57 – « Composants utilisés »



Source : diagramme de l'infrastructure finale du livre *Symfony : The Fast Track*

17.3 Approfondir

— Dungeons and developers ?
— Symfony : The Fast Track « *The official Symfony book for newcomers and experienced developers* »

17.4 Conclusion de la conclusion

17.4.1 Développer des applications Web

— Web : OK
— Reste problèmes applicatifs, conception
— Productivité (ex. formulaires Symfony)
— Qualité logicielle
    — Tests automatisés (une fois codés)

La « suite » en CSC4102 !

17.5 Postface

17.5.1 Crédits illustrations et vidéos

— Diagramme Web Architecture 101 par Jonathan Fulton
— Diagramme « *Software Architecture* » Symfony : *Symfony : The Fast Track* par Fabien Potencier

18 Index

Index