Architecture(s) et application(s) Web

Télécom SudParis Olivier Berger (TSP)

26/08/2025

CSC4101 - Sessions, Contrôle d’accès

Table des matières

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.

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.

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 […] »
  Source : `https://en.wikipedia.org/wiki/Session_(computer_science)`

1.2 Paradoxe : applications sur protocole sans état

- L'expérience utilisateur Web suppose une **instance unique** d'exécution d'application, comme dans un programme « sur le bureau »
- Faciliter la contextualisation du HATEOS :
  est-ce que passer d'un état à un autre est une transition valide ?
- Pourtant, le protocole 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.).

1.3 Rappel : l'application n'arrête pas de s'arrêter

À chaque requête :

1. démarrage application
2. routage vers méthode Contrôleur
   (a) *chargement données depuis base*
   (b) traitements
   (c) *sauvegarde en base données à persister*
3. envoi Réponse
4. **mort**

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.

1.4 Application peut garder la mémoire

- Le programme Web peut stocker un état (par ex. en base de données) avant de s'arrêter
  tracer dans cet état qu'on a fini de répondre à un client donné
- Il peut le retrouver au démarrage (début de la requête suivante)
  filtrer les traces précédentes pour retrouver les données du même client

Simule une session d'exécution unique comprenant une séquence d'actions du même client
Le client doit pour cela se faire reconnaître du serveur

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

1.5 Exécution contextualisée pour un client

À chaque requête :

1. **Identification du client** (contexte de la requête)
2. démarrage application
3. routage vers méthode Contrôleur
   (a) **chargement (depuis base**) état précédent de **ce client**
   (b) traitements
   (c) **sauvegarde (en base**) nouvel état
4. envoi Réponse
5. **mort**

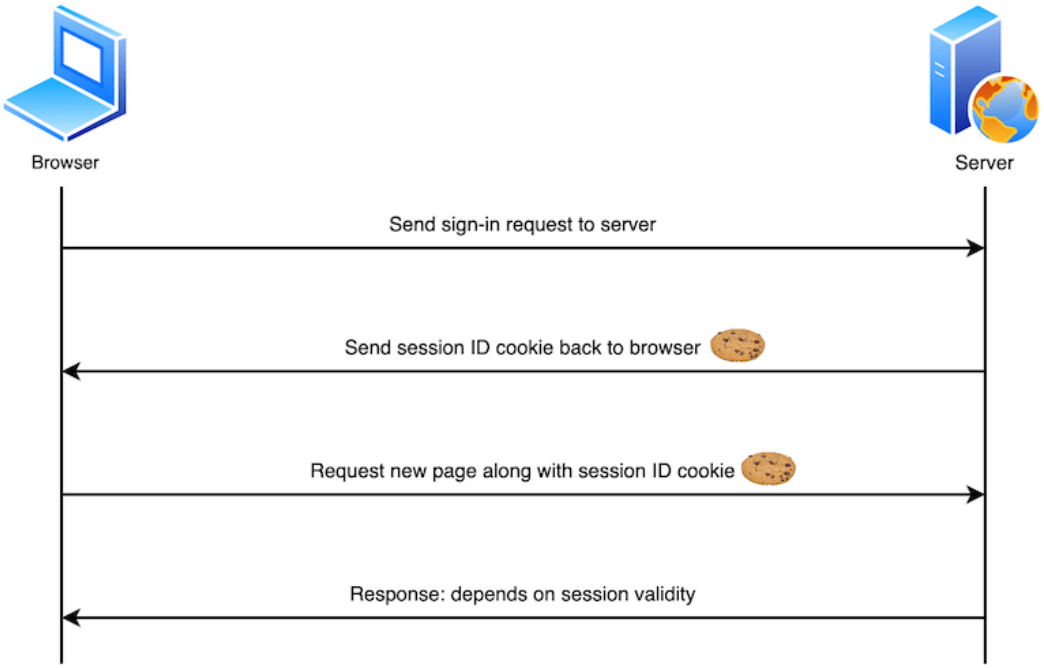
Rendre cela efficace !

1.6 Comment identifier le client HTTP

- Argument d'invocation dans URL : môche, car URL pas uniformes pour tous
- En-tête particulier de la requête ?
  - OUI : ***Cookie***
- …

Respecter l'interopérabilité : standardisé par RFC 6265

1.7 Session de navigation typique



1.8 *Cookies*

Analogie avec badge d'accès dans le monde réel

- Juste un « jeton » unique qui identifie un client HTTP
- 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.

1.9 Identification du client par *cookie*

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 ?

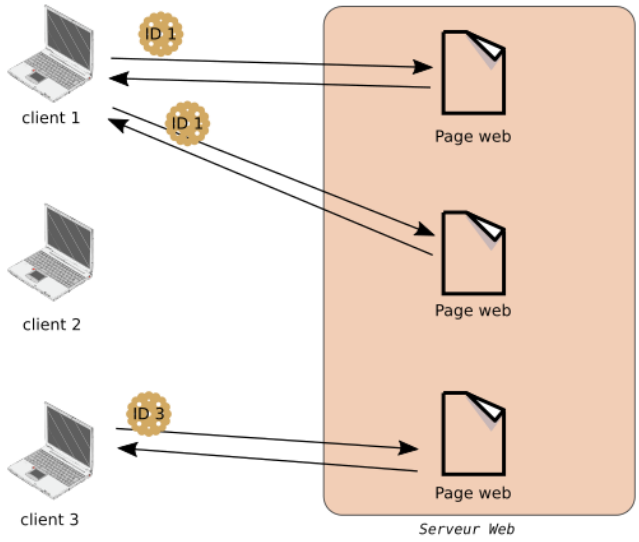


Figure 1 – Multiples clients et pages

1.10 Reconnaître le client HTTP

- Données du *cookie* stockées sur client, envoyées avec la requête au serveur (en-têtes)
- Serveur extrait données du cookie :
  - **identifiant** unique
  - …
- À partir de cet *identifiant*, le serveur peut trouver des données plus complètes stockées de son côté (base de données, stockage **session**, …)

1.11 Reconstituer côté serveur la continuité d'une session de na

À chaque nouvelle requête d'un client HTTP, la présentation du même *cookie* permet de relier :

- éléments de contexte *dans la requête* (en-têtes HTTP, arguments chemin,…)
- **mémoire de l'état précédent** sauvegardé côté serveur (notamment dans la session), à la **fin de la réponse à la précédente requête HTTP** 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.

1.12 Stocker de session côté serveur

Stockage de session côté serveur :

- Espace de stockage central **côté serveur** : référence du contexte d'exécution
- Accès en continu : rapide, solide, etc.
- Accès rapide : stocké dans la mémoire du serveur
- Durée de vie et unicité des sessions au choix du serveur

Accès rapide, y compris avec serveurs redondants !
Accessible pour la plate-forme applicative :

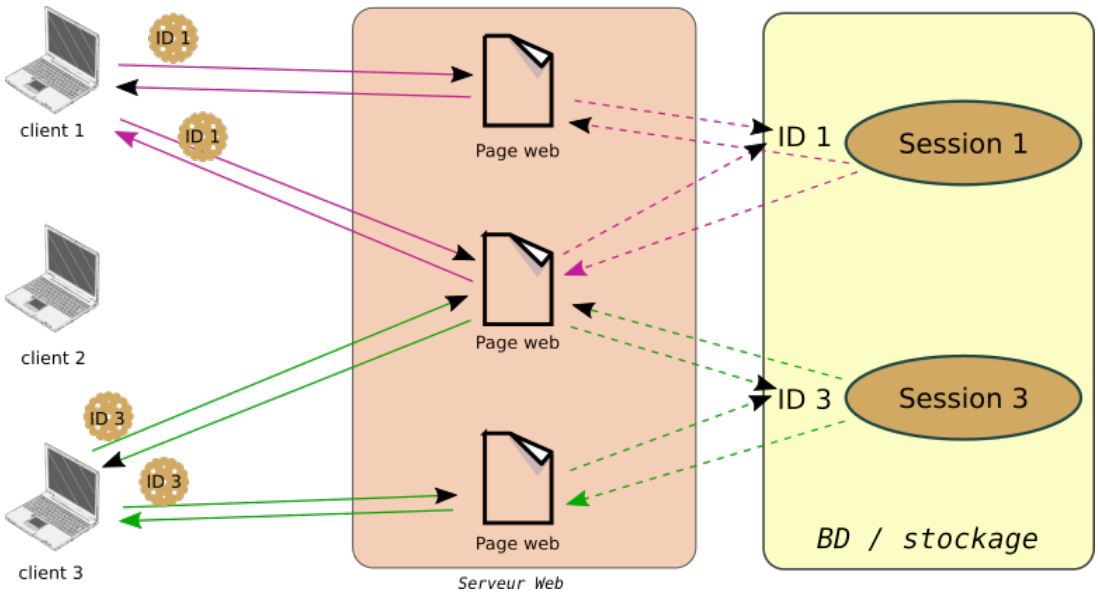


Figure 2 – Stockage session côté serveur

- Données plus ou moins volumineuses
- Disponibles rapidement pour les programmes (plus que SQL vers SGBD)
- **Objets du modèle** de l'application stockés dans la session (pas contrainte taille), *sérialisés*

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.

1.13 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/...`
- Sauf **navigation privée** dans navigateurs Web

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

1.14 *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).

2 Cookies !

Pourquoi tout ce tapage sur Cookies, RGPD, vie privée, etc.
- — traçage requête accès ressources (images, scripts JS, fonts, etc.)
- — régies publicitaires
- — profilage (pas que les cookies)

Plus de détails : « Les Cookies, qui sont-ils ? Que veulent-ils ? » (LQDN)
Mais la publicité… quels modèles économique du Web ?
Aller plus loin : cf. « économie de l'attention », « capitalisme de surveillance » (Stéphane Croizat - UTC)
Mon conseil : bloqueurs de pubs (uBlock Origin, Privacy Badger, …)

3 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.

3.1 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

3.2 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.

3.2.1 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).

3.2.2 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.

3.3 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.

3.4 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.

4 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.

4.1 Mécanismes

- Authentification HTTP
  - Authentification « Basic »
  - autres
- **Authentification applicative**

4.2 Basic Auth

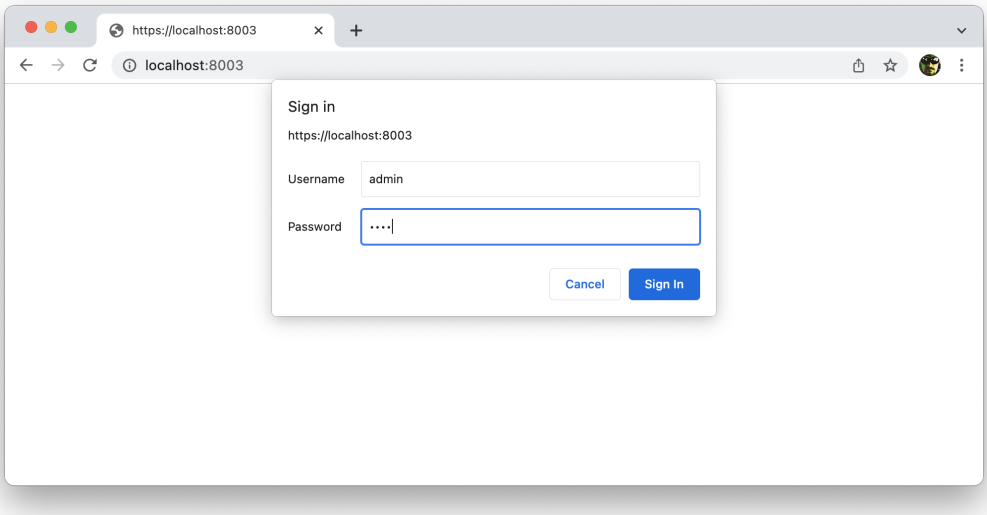


Figure 3 – Source : Why I'm Using HTTP Basic Auth in 2022 par Joel Dare

Inconvénient : pas de *logout*

4.3 Authentification applicative

4.3.1 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*
  - 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.

4.3.2 Formulaire d'authentification

- Formulaire « standard »
- Champs :
  - Login ou email
  - Mot-de-passe (saisie cachée)
- Requête POST
- Initialise / Récupère session applicative contenant identification ou directement les autorisations

Le formulaire est assez standard, mais recèle, en pratique des champs cachés que l'utilisateur ne voit pas.

4.3.3 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.

4.3.4 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

4.3.5 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… ?

4.4 Se protéger

En tant qu'hébergeur d'une application Web
- HTTPS everywhere
  - Cookie dans en-têtes, chiffrés => identifiant de session secret
- Déjouer les attaques par force brute

4.4.1 Captcha

Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA)
Vérifier qu'un humain est aux commandes

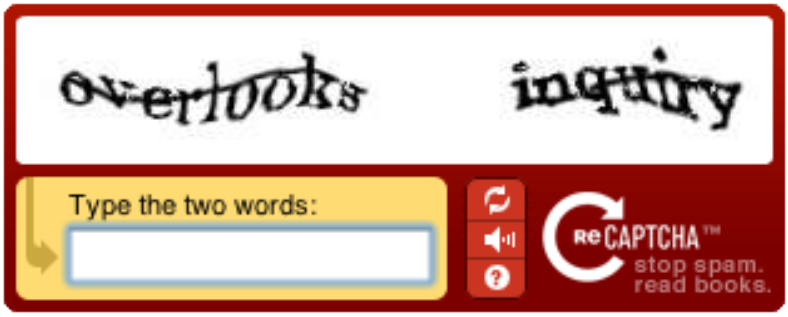


Figure 4 – Exemple reCAPTCHA

— Pas infaillible
— Problèmes accessibilité

4.4.2 « Merdification » du Web

Combien de fois par jour voyez-vous ça ?

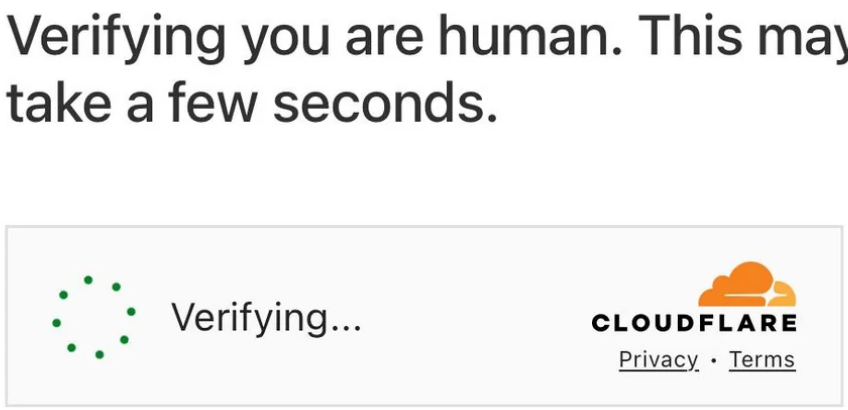


Figure 5 – Exemple Captcha « transparent » pour les humains

Ce genre de dialogues apparaît même dans des applis mobiles Android (le bon coin, par ex.) !
Le terme « merdification » vient de l'anglais « enshittification » inventé par Cory Doctorow (cf `https://next.ink/brief_article/l-entshittification-ou-merdification-mot-de-lannee-du-dictionnaire-macquarie/`)

4.4.3 Se protéger du *scraping* des *bots* des IA

Les *bots* des opérateurs d'IA génératives moissonnent (*scraping*) violemment, et les sites s'écroulent

Captation sauvage de la connaissance qui pénalise en particulier le monde du « libre » / bénévole
Aucun respect pour le fichier `robots.txt` (`https://robots-txt.com/`)

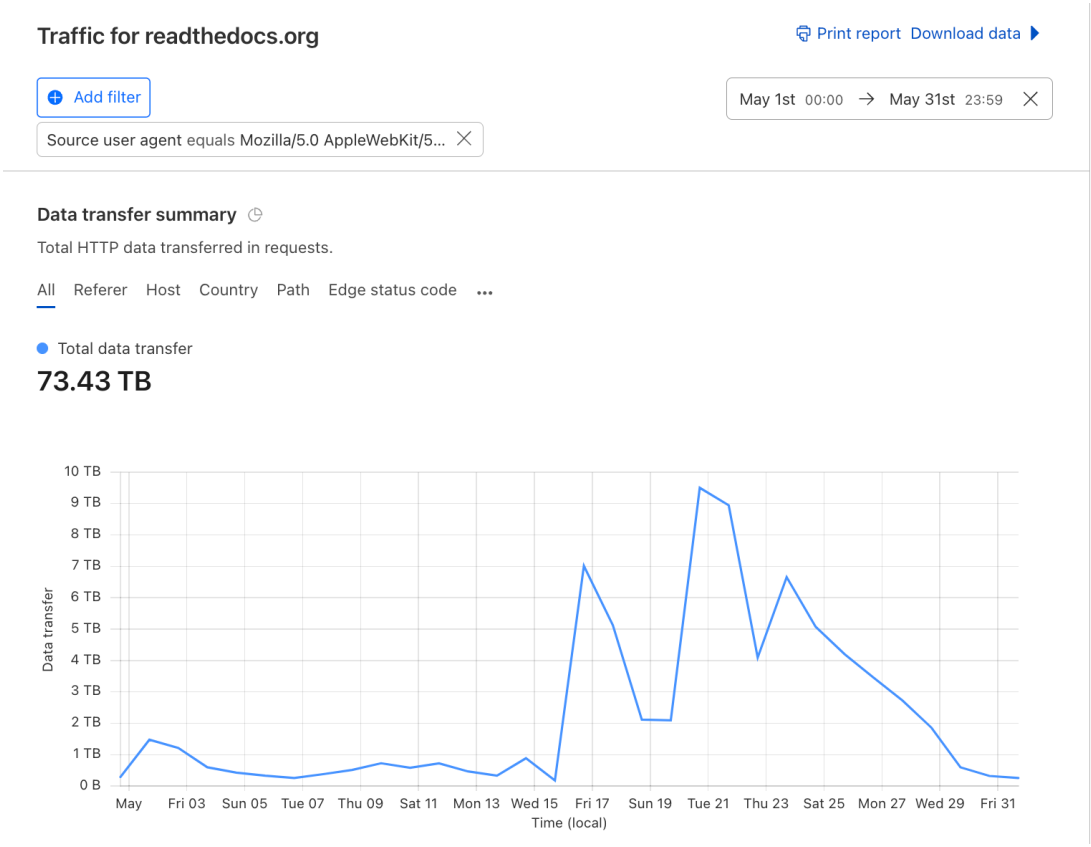


Figure 6 – statistiques bande-passante ReadTheDocs

4.4.4 Protection

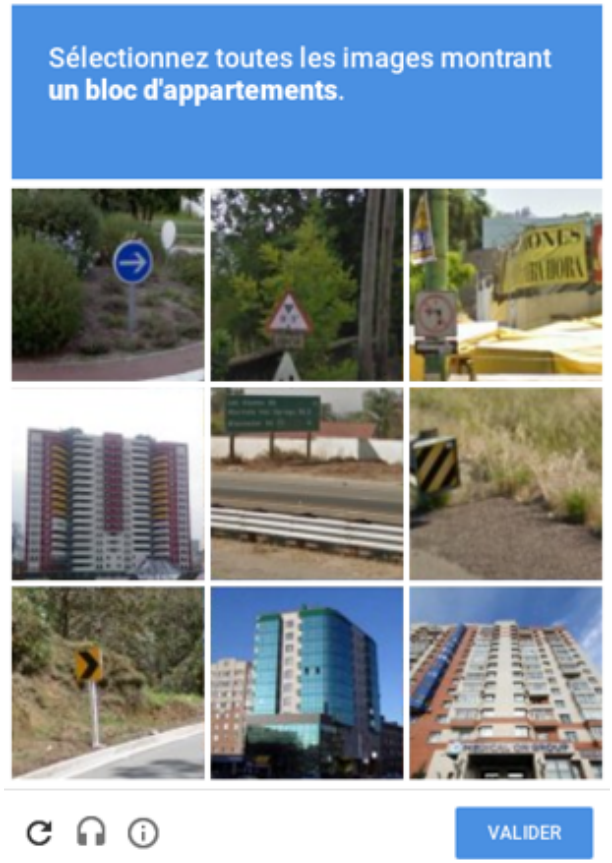
— Même style que pour attaques DDoS avant IA
— Pièges à bots : *proof of work* (Anubis)
— Pas évident (et pas gratuit)

Protection efficace, mais risque monopôle, et surveillance à grande échelle !
Généralisation tracking des utilisateurs légitimes ?
Impact environnemental !!!

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.

4.4.5 Travail dissimulé pour l'IA

En passant :



- Travail gratuit dissimulé (petites mains du clic des IA)
- Entraîner des systèmes d'armes autonomes ?

> In terms of cost, we estimate that – during over 13 years of its deployment – 819 million hours of human time has been spent on reCAPTCHA, which corresponds to at least $6.1 billion USD in wages.
>
> Traffic resulting from reCAPTCHA consumed 134 Petabytes of bandwidth, which translates into about 7.5 million kWhs of energy, corresponding to 7.5 million pounds of CO2.
>
> In addition, Google has potentially profited $888 billion USD from cookies and $8.75-32.3 billion USD per each sale of their total labeled data set.

Source : Dazed & Confused : A Large-Scale Real-World User Study of reCAPTCHAv2 Andrew Searles, Renascence Tarafder Prapty, Gene Tsudik

De plus, pour l'exemple de l'illustration « CAPTCHA et digital labor », 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*)

4.4.6 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

Authentification plus forte.
Attention : certains mécanismes s’avèrent moins fiable que prévu (SMS)
Attention aux exigences de sécurité réglementaires.

5 Rôles et permissions

Cette section présente le principal modèle de définition de permissions utilisé pour le contrôle d'accès, à base de rôles.

5.1 *Role-Based Access Control* (RBAC)

Contrôle d'accès à base de rôles
- Utilisateur
- **Rôle**
- Permissions

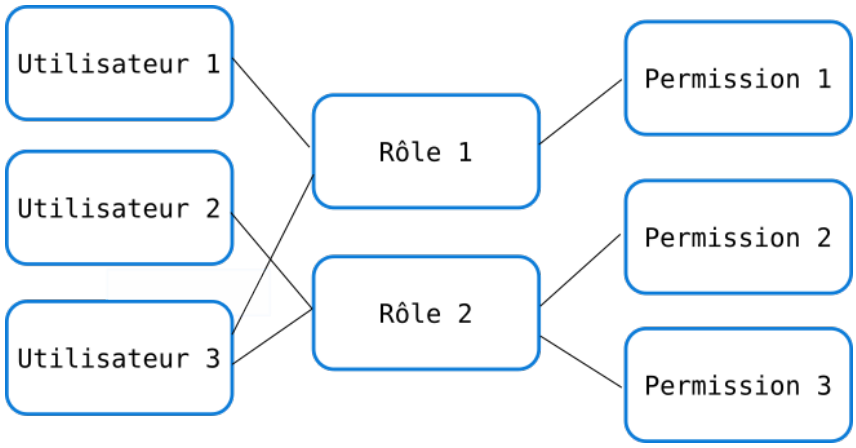


Figure 7 – Exemple d'affectation de rôles

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.

5.2 Permissions

- Modèle applicatif de permissions
- Vérifier les permissions à chaque traitement de requête
  - Routage
  - Dans les traitements fonctionnels

Module « *Firewall* » de Symfony

5.3 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 »…

6 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.

6.1 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.

6.2 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.

6.3 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)

6.4 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]
```

6.5 *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).

6.6 Utilisation dans les contrôleurs

— Contrôle d'accès sur les routes :

```
#[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 :

```
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.

6.7 Profil de l'utilisateur

- Accès aux propriétés de l'utilisateur :

```php
$this->getUser()

// ...

$email = $this->getUser()->getEmail();
$post->setAuthorEmail($email);
```

6.8 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.

6.9 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 apparaître 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.

6.9.1 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

7 Postface

7.1 Crédits illustrations et vidéos

— Illustration copie écran Basic Auth HTTP via Joel Dare
— Illustration session navigation typique via MDN : https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Cookies
— https://knowyourmeme.com/memes/darth-vaders-i-find-your-lack-of-faith-disturbing
— 3 spidermen : fabriquée avec « Spider Man Triple Meme Generator » d'imgflip - voir aussi https://knowyourmeme.com/memes/spider-man-pointing-at-spider-man
— Illustration « Statistiques bande-passante ReadTheDocs » : https://about.readthedocs.com/blog/2024/07/ai-crawlers-abuse/

Copyright