Projet Quoicoutionnaire

Introduction

Ce rapport en forme de *storytelling* vise à raconter le déroulement pas à pas de cette SAE Instanciation de Services Réseaux. Il est bon de noter ici que cette SAE est pratiquement achevée à l'écriture de ce rapport version 2, 2e semaine de SAE, lors de laquelle nous avons réarrangé les binômes. Cette avance s'explique aussi par le fait qu'un des deux binômes actuels, Breval, a déjà fait 1 an de BUT R&T et connaît toute la pile sur le bout des orteils (ou presque).

Quoicouquoi ?

Quoicoubeh, quoicoubaka, quoicouflop... Ces mots en quoicou- font depuis 2023 partie du vocabulaire comique et loufoque de plusieurs jeunes générations. Vous pouvez retrouver plus d'informations sur Orthodidacte.com. Ce préfixe est donc combiné à d'autres mots existants dans le but d'amuser la gallerie, mais pour l'instant il semblerait que seul le mot « quoicoubeh » apparaisse sur un dictionnaire. Ainsi, nous avons décidé de créer notre propre dictionnaire exclusivement orienté vers les mots en quoicou-, comme solution rapide, facile et efficace pour un gag utile à la SAE.

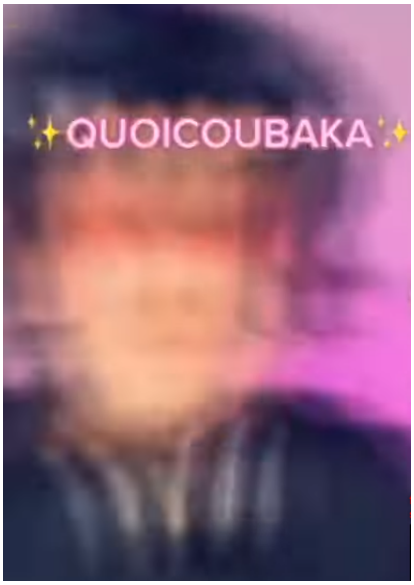

Figure 1: capture d'écran de la vidéo d'un jeune homme se servant du préfixe pour faire rire. [source]

Installation de LAMP

Pour l'installation de la pile, étant donné que l'état des machines sous CentOS ne nous permettait pas l'accès à LAMP sensé être pré-installé, nous avons choisi un bon vieil ordi portable sous Ubuntu. Une seule commande via Aptitude nous permet d'installer toute la pile :

```
sudo apt install php php-mysql php-json mysql-client mysql-server apache2
```

Apache2 et MySQL sont fournis sous forme de services qui démarrent dès l'installation terminée, contrairement à PHP qui est juste un interpréteur comme Python. Comme je les utilise sur un ordinateur que j'utilise régulièrement pour autre chose, je désactive leur lancement automatique au démarrage avec SystemCTL :

```
sudo systemctl disable mysql apache2
```

Comme je ne sais plus si SystemCTL arrête les services en même temps de les désactiver, je vérifie leur statut avec :

```
sudo systemctl status mysql apache2
```

Ouf, ils tournent.

MySQL nécessitait un serveur (le service en question) ainsi qu'un client pour lui parler en SQL. Après installation, il est conseillé de sécuriser ses bases avec divers MDP en utilisant **mysql_secure_installation**. Cependant, je compte n'utiliser qu'une base moindre et en local, donc je me contente de créer un accès pour les API de l'interpréteur PHP ainsi que la base en question.

Configuration du serveur MySQL

Pour avoir une base contenant les mots commençant par quoicou-, il me suffit d'une table à une colonne. Commençons par la base, je me connecte en local et en root :

```
sudo mysql
create database quoicou;
use quoicou;
```

Une fois dans la bonne base, je crée la table en question :

```
create table mots(label varchar(128) primary key);
```

Une chaine de 128 caractères devrait suffir pour contenir un seul mot. La clé primaire nous assurera qu'il n'y aura pas de doublon.
Pour ajouter des mots à cette table, je demande à ChatGPT qui a beaucoup d'humour. L'IA est capable de me générer plusieurs requêtes pour ajouter directement les données à la table :

```
INSERT INTO mots (label) VALUES ('quoicouillon'), ('quoicouac'), …;
```

Remarquez l'absurdité des deux exemples inclus. Quel chef d'oeuvre de technologie. Il nous reste à créer des accès pour plus tard :

```
create user php identified by 'K&a=K!pKv~5kf8qa';
grant select, insert on quoicou.mots to php;
```

Il n'y a pas beaucoup d'intérêt à glisser un MDP trop complexe ici, juste par réflexe. Passons au PHP.

Conception du site

Apache2, qui tourne en arrière-plan tel un rat enfermé dans un tonneau, est à l'écoute depuis tout ce temps. Il agit dès qu'on demande un HTML GET vers l'adresse de la machine, ce qu'on peut déclencher en demandant http://localhost depuis un navigateur. Par défaut, il affiche une page de démo Ubuntu correspondant au fichier index.html dans son répertoire root, par défaut chez nous **/var/www/html**. On peut alors tester le fonctionnement de l'interpréteur PHP avec un fichier contenant un petit script qui génère une page de test :

```
<?php phpinfo(); ?>
```

Un fichier PHP est un fichier HTML avec une extension PHP qui lui confère le pouvoir d'être transformé par l'interpréteur PHP. Tout ce qui est entre **<?php** et **?>** est interprété. C'est la fonction phpinfo qui génère toute la page.

Comme il nous est demandé, nous ajoutons dans le répertoire S203 un fichier index.html pointant (à l'aide de liens) vers le fichier de test PHP ainsi qu'un autre site pour tester les requêtes avec la base de données. Ces requêtes sont possibles grâce à l'API MySQL pour PHP que nous avons ajouté à la pile en installant le module **php-mysql**. Le module JSON était conseillé, je ne sais pas trop s'il était nécessaire.

Le site principal, d'où émanera les requêtes, est notre fameux dictionnaire. Nommé quoicou.php, il permet d'afficher les mots de la base à l'aide de 2 selects ; le premier nous renseigne sur les lettres par lesquelles commencent nos mots (après quoicou, soit l'index 8) et le nombre de mots par lettre. Grâce à ce nombre, pour chaque lettre, nous ajoutons une section avec en titre h2 la lettre et listons les mots correspondant. Chaque echo en PHP ajoute un élément au DOM et la page paraît comme du HTML statique côté client, comme on peut le voir quand on cherche la source de la page par un clic doit. Si jamais une erreur est rencontrée lors de la connexion ou des requêtes, tout plante et cette erreur est affichée grâce à :

```
error_reporting(E_ALL);
ini_set('display_errors', 'On');
```

A la fin du script, la connexion est fermée.

Comme nous utilisons sur cette page juste des select, nous allons tester les insert sur une autre page, et utiliser cette page pour ajouter des mots à la base. Cette seconde page est appelée submit.php et consiste en un formulaire avec une entrée de texte et un bouton pour confirmer l'entrée. Ce bouton nous mène vers une autre page, submit.result.php, qui peut traiter les infos envoyées par la méthode HTTP POST. La méthode GET peut aussi être utilisée mais la présence de variables dans la barre de recherche nous semble un élément parasite inutile. Les informations transmises sont disponibles dans le dictionnaire **$_POST** ; chaque nom d'input du formulaire est une clé.

Je ne l'ai pas précisé précédemment mais le MDP de l'utilisateur php utilisé pour les requêtes SQL est donné en clair dans les fichiers PHP, ce qui est déconseillé pour un serveur public (encore une fois on fait ce qu'on veut on est en local). Il pourrait être possible pour contrer cette contrainte d'avoir les identifiants en variables d'environement ; ainsi ces identifiants ne pourront être dérobés qu'en cas de hacking du serveur. Reprenons sur la page de résultat.

Comme cette page est susceptible de contenir des erreurs (le mot existant déjà dans la base ou ne commençant pas par quoicou-), nous avons structuré le script de cette page dans une fonction pour que celle-ci s'arrête à la moindre erreur mais en laissant le reste de la page s'afficher. Nous utilisons un boolean pour savoir si tout s'est bien passé (faux au départ puis vrai à la fin) et dans le cas échéant une chaîne de caractères nous indiquera quel est le problème. Les deux erreurs anticipées (mot existant ou mal commencé) sont

explicitées avec un test et un `try catch`, puisque le code d'erreur 1062 correspondra toujours à un mot déjà existant.

La suite de la page est juste du HTML conditionnel, parsemée de clauses PHP qui modifient le `div` en fonction du résultat.

Module complémentaire

Tout est terminé et nous relisons tranquilement le sujet jusqu'au moment où nous nous apercevons qu'il est nécessaire d'utiliser un module complémentaire dans la page. Comme tout module complémentaire installable avec Aptitude commence par php-, nous effectuons une recherche sur packages.ubuntu.com et sans rien trouver de particulier, nous allons sur GitHub. Le module en question ne doit pas être trop utile sinon nous devrons modifier une grande partie du site.

Nous trouvons alors FakerPHP, une librairie qui sert à générer des fausses données comme des adresses mail, des nombres aléatoires, des noms... Nous pensons alors à simuler un système d'utilisateurs en générant un nom d'utilisateur aléatoire et en faisant semblant qu'il est connecté.

On commence par importer Faker dans notre projet grâce à composer, installable avec Aptitude :

```
composer require fakerphp/faker
```

Composer télécharge la librairie dans un dossier vendor et garde une trace des dépendances avec un JSON. Ensuite on importe la librairie dans notre code (quoicou.php) :

```
require_once '../vendor/autoload.php';
```

Le tour est joué, il ne reste plus qu'à rajouter dans la DOM un nom d'utilisateur fabriqué :

```
echo Faker\Factory::create("FR_fr")->userName();
```

On crée un générateur avec une locale en français puis on lui demande un nom d'utilisateur.

Coloriage

Il ne reste que la dernière touche artistique : le CSS. Ayant grandit avec Comic Sans, comme ma génération et d'autres proches, j'ai appris à le détester jusqu'à... L'apprécier. Puisque ce site est un gag, autant le pousser au bout et honorer Comic Sans MS, la police des pauvres (de son utilisation souvent à défaut de mieux ou de considération). Même ChatGPT le déconseille sans qu'on lui demande son avis. Quelques couleurs grossières et non assorties, un look de blocs mal alignés pour les conteneurs de mots (display : inline-table), du rouge pour les erreurs et du vert pour les succès, enfin un banner coloré et on a un site mature.

Le tout nous a pris environ 6h et nous a permis à un des deux binômes de revoir ses bases web (et apprendre l'existence miraculeuse des `try` / `catch` en PHP) et à l'autre d'apprendre la mise en place d'un serveur avec LAMP.