Qu’est-ce que PHJS

PHjs est un framework nodeJS ayant pour ambition de pouvoir remplacer PHP dans le développement backend.

Ainsi, tout ce qui peut être fait en PHP peut aussi être fait avec PHjs, mais avec du javascript et en bénéficiant de l’asynchrone de ce dernier.

Cependant, comme c’est du nodeJS, il tourne nativement en single-thread, il faut donc utiliser le mode cluster de nodeJS avec un outil comme pm2 pour faire tourner PHjs en multi-threads, et ainsi avoir plus de performances.

Installation / Configuration

Tout d’abord, télechargez PHJS avec git :

git clone https://github.com/AutoCoderr/PHjs.git

Une fois fait, un dossier ‘PHjs’ contenant tout PHjs vient d’être créé.

Pour le lancer, nous allons nous baser sur le fichier ‘start-example.js’ présent dans ce dossier.

Copiez ce fichier dans le répertoire courant :

cp ./PHjs/start-example.js ./start.js

Puis ouvrez ce dernier

var PHjs = require("./PHjs");

var options = {};

var libs = {};

PHjs("/projects/PHjs/website/","http",8000,options,libs,"/projects/PHjs/access.log","/projects/PHjs/error.log","/projects/PHjs/config.txt");

Dans le fichier ci-dessus :

var PHjs = require(‘./PHjs’) ; => importe tout simplement le module PHjs

options => Cette variable, qui est vide dans cet exemple, sert dans le cas où l’on utilise https, à stocker toutes les options https comme la clé privé ou le certificats, pour cela, aller voir ici : < le futur lien >

libs => Cette variable contient toute les libs que l’on souhaite pouvoir utiliser depuis nos scripts PHjs, nous pouvons par exemple y mettre le module mysql ou encore mongodbClient

La dernière ligne :

C’est cette ligne qui s’occupe de lancer le serveur PHjs

"/projects/PHjs/website/" => le chemin vers le dossier contenant tout les fichiers de votre futur site.

"http" => le protocole utilisé, qui peut être ‘http’ ou ‘https’

8000 => le numéro de port utilisé, s’il n’est pas spécifié, il sera automatiquement à 80 si vous êtes en http et à 443 si vous êtes en https

"options" => déjà expliqué plus haut

"libs" => déjà expliqué plus haut

"/projects/PHjs/access.log" => le chemin vers le fichier de log qui enregistrera toutes les connexions, s’il n’existe pas, il sera automatiquement créé

"/projects/PHjs/error.log" => le chemin vers le fichier de log qui enregistrera toutes les erreurs, s’il n’existe pas, il sera automatiquement créé

"/projects/PHjs/config.txt" => le chemin vers le fichier contenant la config pour PHjs, si vous ne voulez pas de config particulière, vous pouvez mettre "nothing"

Maintenant, vous n’avez plus qu’à modifier ce fichier pour qu’il corresponde à vos paramètres et à le lancer avec

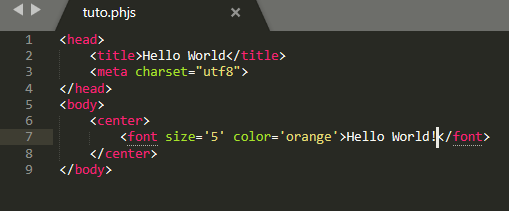
node start.js

Utilisation :

Comme dit plus haut, PHjs vise à remplacer PHP, mais avec du javascript.

Ainsi je vais montrer ce qu’il est possible de faire avec des exemples de script PHjs simple.

Tuto d’abord, un fichier basique, que l’on stocke dans le dossier spécifié dans start.js :



On peut noter que le fichier doit porter l’extension ‘.phjs’ pour pouvoir exécuter du PHjs (comme en PHP), mais qu’on peut aussi choisir de ne mettre que du code coté client (HTML), comme dans l’exemple ci-dessus.

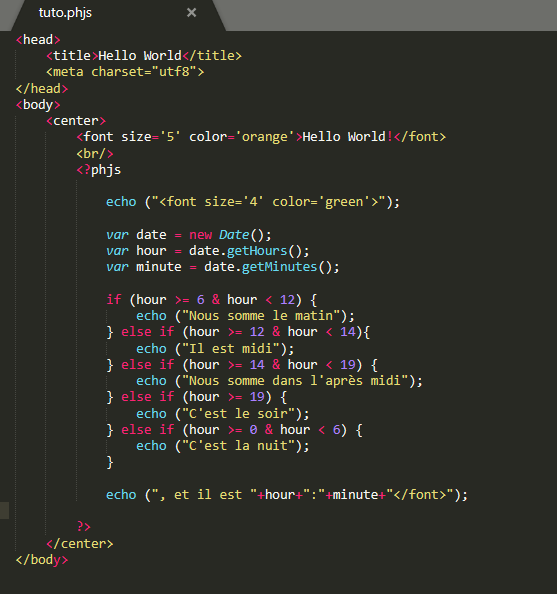
Nous allons ajouter un peu de code PHjs à note page.



Dans cet exemple, j’ai utilisé la fonction ‘echo’, qui, comme en PHP renvoie son contenu au client, à noter qu’il y a aussi sa variante ‘echo\_json’ qui envoie des informations au format json.

Pour le moment, nous n’avons fait que du code basique se contentant de faire de l’affichage, mais il est bien sûr possible d’exécuter tout type de code js coté serveur.

Voici un exemple de code plus complexe :



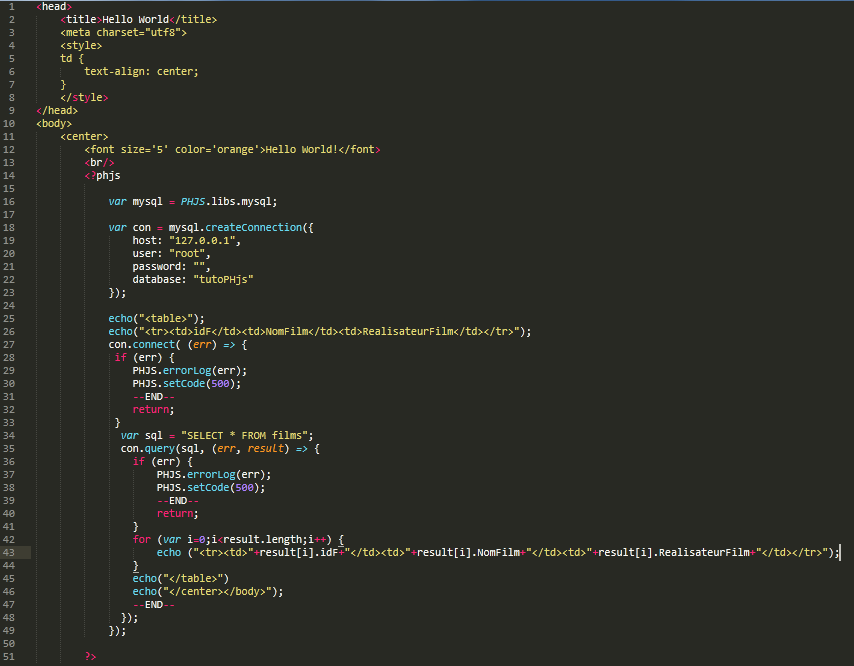
Le code PHjs ici est plus volumineux, vous pouvez aussi voir que j’ai changé la coloration syntaxique de sublime text pour voir le javascript de PHjs.

Le script ci-dessus affiche ‘Hello World !’ comme dans les exemples précédant, fais un saut à la ligne, dit que nous somme le matin, le midi, l’après midi, le soir, ou la nuit en fonction de l’heure de la journée, puis nous affiche l’heure. Pour cela on utilise du code javascript comme si on faisait un code javascript classique dans nodeJS ou dans le navigateur, mais intégrer à la page web et exécuté coté serveur.

Dans l’exemple suivant, j’irais chercher des données dans une base sql et je les afficherais dans un tableau en utilisant le module ‘mysql’ de node JS, et j’en profiterais pour expliquer l’utilisation de l’asynchrone dans PHjs.

Pour cet exemple, nous partons du principe que nous avons créé une base de donnée SQL nommé « tutoPHjs » et que nous y avons créé la table suivante.

films(idF,NomFilm,RealisateurFilm)



Avant d’exécuter ce code, j’ai rajouté ‘libs.mysql = require("mysql");’ dans start.js

Explication de l’asynchrone et du mot clé --END-- dans PHjs :

Déjà il est important de noter la présence d’un mot clé particulier dans ce code, il s’agit du mot clé : --END-- (lignes 31, 39 et 47)

En effet, ce mot clé indique à quel moment la page web doit être envoyée au client, quand ---END-- n’est pas indiqué (comme dans les exemples précédents), le moteur PHjs le rajoute automatiquement après la dernière ligne du script.

Dans notre cas, nous utilisons une fonction asynchrone, nous voulons donc que la page soit envoyée à l’utilisateur une fois la fonction asynchrone terminée (dans son callback), si nous laissons PHjs envoyer la page au client à la fin du script, à l’extérieur de la fonction asynchrone, le client recevra la page avant même que la fonction asynchrone n’ai finis de s’exécuter, et il n’aura pas le tableau contenant les données de la table sql.

Comme le principe d’une fonction asynchrone est de ne pas s’executer de manière synchrone avec le reste du code, le code mentionner après et à l’extérieur la fonction asynchrone s’exécute avant le code censé s’executer une fois la fonction asynchrone terminée (son callback).

Il est alors nécessaire de mettre le mot clé –END—dans le callback de la fonction asynchrone, après la génération de la table html, pour s’assurer que le client reçoive la page une fois ce tableau généré.

Maintenant, analysons ce script depuis le début,

Nous commençons par du code html classique (comme dans les exemples précedents), avec un petit code css que j’ai rajouté pour rendre le tableau html plus lisible (ligne 4 à 8)

Nous avons ensuite le hello world, un saut à la ligne, puis c’est là que commence notre code PHjs (ligne 14).

var mysql = PHJS.libs.mysql; => Ce code enregistre dans la variable ‘mysql’ la lib mysql qui a été spécifié dans start.js, en effet, toutes les libs enregistré dans la variable ‘libs’ et transmise à PHjs dans start.js peuvent être récupérés dans PHJS.libs depuis le script PHjs. J’expliquerais plus tard la variable ‘PHJS’

var conn = mysql. createConnection({… (ligne 18)=> nous créons une connexion à mysql comme quand on utilise mysql dans un programme node JS classique.

Ensuite dans avons des ‘echo’ pour afficher le début du tableau html ainsi que sa première ligne (ligne 25 et 26)

Puis enfin notre première fonction asynchrone : con.connect( (err) => { ..

Gestion des erreurs dans PHjs :

Déjà nous pouvons voir de la ligne 28 à 33 la gestion des erreurs pour ‘con.connect’.

En PHjs, quand une erreur fait planter notre script (comme l’utilisation d’une variable non définie), PHjs la détecte automatiquement et renvoie une erreur 500 au client tout en stockant l’erreur dans le error.log. Mais il est aussi possible de gérer manuellement les erreurs comme dans ce script aux ligne 28 à 33 et au lignes 36 à 41.

Déjà nous voyons que s’il y a une erreur,

le script effectue la fonction : PHJS.errorLog(err); (ligne 29)

Cette fonction écrit l’erreur stocké dans la variable ‘err’ dans le fichier error.log.

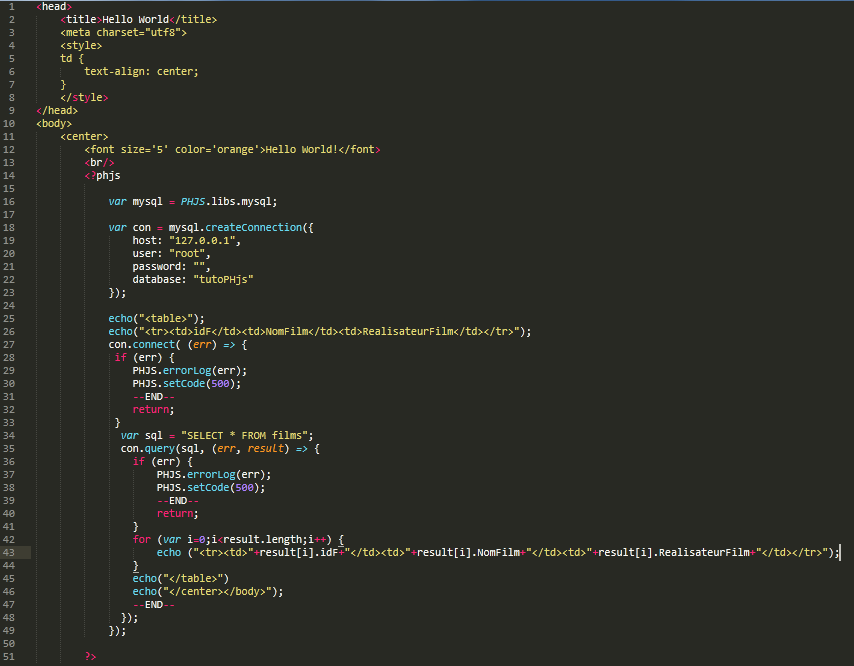
Ensuite nous avons : PHJS.setCode(500); (ligne 30)

Cette fonction indique que nous souhaitons renvoyer une erreur 500 à l’utilisateur

Par la suite, nous avons le mot clé --END--qui envoie la page avec l’erreur 500 à l’utilisateur.

Puis un ‘return;’ qui précise que nous souhaitons arrêter l’exécution du script maintenant, autrement, PHjs pourrais continuer d’exécuter le script même quand la page a déjà été envoyée au client.

Rappel du script :



Maintenant analysons le code qui s’exécute, s’il n’y a pas eu d’erreur à partir de la ligne 34.

Nous avons d’abord la déclaration de la variable ‘sql’ contenant la requête qui va récupérer le contenu de la table ‘films’, puis, juste après, la fonction asynchrone ‘con.query’ qui exécute cette requête SQL, pour le moment, rien ne diffère d’un programme nodeJS classique.

Ensuite, nous avons, comme précédemment, la gestion des erreurs pour cette fonction.

Puis, par la suite, une boucle for qui parcourt l’array ‘result’ (c’est cet array qui contient le contenu de note table sql), qui pour chaque ligne de la table sql, écrit une nouvelle ligne pour le tableau html avec ‘echo’.

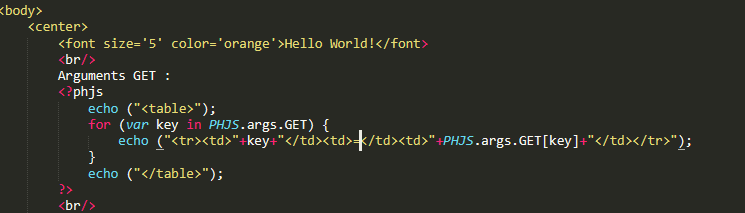
Une fois fait, on ferme les balise avec </table>, </center> et </body>, puis on renvoie l’information au client avec --END—

Les arguments GET et POST

En PHjs, les argument transmis en GET et en POST se trouvent dans : PHJS.args

GET => PHJS.args.GET.leNom

POST => PHJS.args.POST.leNom.type pour le type (text ou file) et PHJS.args.POST.leNom.content pour le contenu de l’argument

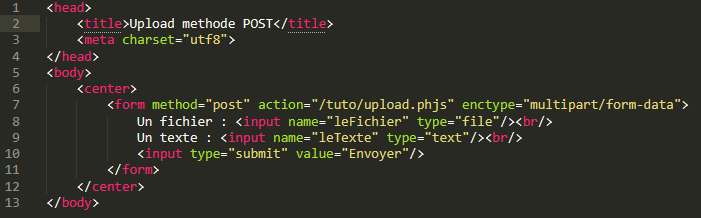


J’ai rajouté au début de notre code, une boucle parcourant PHJS.args.GET et qui génère un tableau html à partir de ça, ainsi, si on transmet des argument par la méthode GET, ils s’afficheront dans un tableau HTMl. Notez que PHJS.args.GET et PHJS.args.POST sont des objects javascript, comparable au $\_GET et au $\_POST de PHP.

Upload d’un fichier via la méthode POST

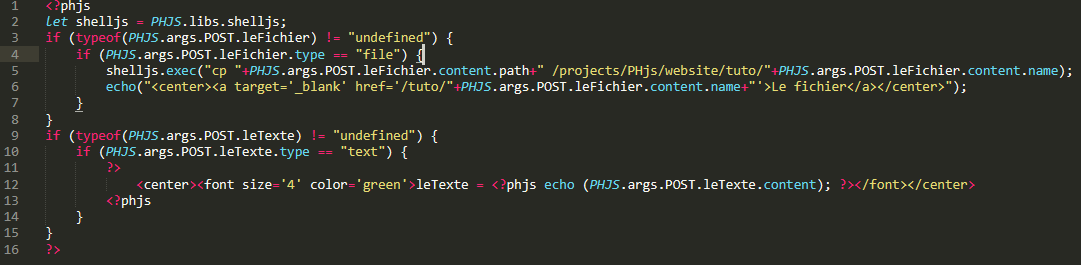
On va créer un dossier tuto dans lequel on créer tout d’abord un fichier formulaire.phjs

formulaire.phjs =>



Un formulaire HTML tout ce qu’il y a de plus classique, comme si l’on utilisait du PHP, à l’exception que l’on met dans le champs ‘action’ un fichier .phjs plutôt qu’un fichier .php

Ensuite, le fichier upload.phjs, qui récupérera les information POST :



Avant d’exécuter ce script j’ai rajouté la ligne libs.shelljs = require("shelljs"); dans start.js

On récupère à la ligne 2, shelljs stocké dans PHJS.libs.shelljs.

Comme on peut le voir, le champ ‘leFichier’ précisé dans formulaire.phjs se retrouve dans PHJS.args.POST.leFichier, de même pour le champ ‘leTexte’

Ligne 3 et 4 : on vérifie sur le champ ‘leFichier’ existe en voyant si son type est ‘undefined’ (avec typeof()) puis, si le champs existe, on vérifie si son type est ‘file’ (comme spécifié dans le formulaire).

Ligne 5 et 6 : si le champs ‘leFichier’ existe et que c’est bien un fichier, on utilise shelljs pour le déplacer de là où il a été enregistré (PHJS.args.POST.leFichier.content.path), vers le dossier tuto de notre serveur (‘/projects/PHjs/website/tuto/’ correspond bien évidement à l’arborescence de mon serveur, à adapter à votre cas et au chemin du serveur spécifier dans start.js), et on lui donne le nom qu’il avait sur le pc du client (PHJS.args.POST.leFichier.content.name).

Puis, on utilise un echo pour faire un lien vers le fichier venant d’être uploadé

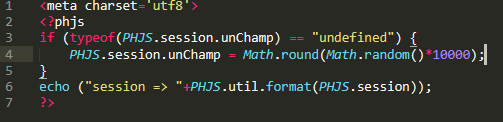
Dans les lignes 9 à 15, nous suivons le même principe que le champ ‘leFichier’ mais avec le champ ‘leTexte’, on vérifie s’il existe et si c’est du texte, puis nous affiche ce texte dans un code html.

Les sessions en PHjs

Dans PHjs, les variables de session sont stockées dans ‘PHJS.session’, c’est l’équivalent de $\_SESSION en PHP.

Contrairement à PHP, pas besoins de faire un ‘session\_start()’, on peut directement créer ou supprimer des champs dans ‘PHJS.session’, et tout supprimer avec ‘PHJS. session\_destroy()’.

Un petit exemple :



Il vérifie d’abord l’existence du champ ‘unChamp’ dans PHJS.session,

S’il n’existe pas, il stocke un nombre aléatoire entre 0 et 10000 dedans, puis affiche le contenu de PHJS.session.

Je vais rajouter quelques conditions :



J’ai rajouté un argument ‘action’ à envoyer avec la méthode GET. Si le champs ‘action’ est égal à ‘destroy’, on effectue un ‘PHJS.session\_destroy()’ pour tout supprimer de la session.

Si le champ action est égal à ‘create’ et que ‘unChamp’ n’existe pas dans PHJS.session, il créer ‘unChamp’ avec nombre aléatoire.

On peut bien évidement utiliser cette variable pour stocker par exemple, le nom, le prénom, ou encore le token de l’utilisateur connecté.

Un champ stocké dans ‘PHJS.session’ sera accessible depuis tout les autres script PHjs, mais uniquement pour l’utilisateur connecté.

PHJS.global :

PHJS.global se comporte comme PHJS.session, à l’exception près qu’il n’est pas lié à la session de l’utilisateur, mais qu’il est global à tout PHjs.

Un champ stocké dans PHJS.global sera accessible depuis tous les autres scripts PHjs, pour n’importe quel utilisateur.

Il dispose aussi de son équivalent : PHJS.global\_destroy() ; pour détruire tous les éléments de PHJS.global.

Les includes en PHjs :

Tout comme en PHP, il est possible de faire des include de d’autre script PHjs

Pour cela, il suffit de faire :

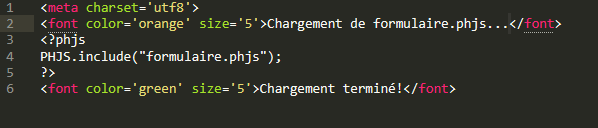
PHJS.include(‘/Le/Chemin/Vers/Le/Script.phjs’) ; (sans callback)

ou

PHJS.include(‘/Le/Chemin/Vers/Le/Script.phjs’, () => { //action à faire après l’include }) ; (avec callback)

Il y a deux manières de faire un include, avec ou sans callback en fonction de si vous voulez prendre en compte l’asynchrone ou non.

Un exemple (ce fichier se trouve dans le répertoire ‘tuto’ du serveur) :



Cette page nous affiche :

« Chargement de formulaire.phjs »

Affiche le contenu de formulaire.phjs

Puis « Chargement terminé »

Il est maintenant important de parler d’une nouvelle variable : PHJS.includeVars

Cette variable est semblable à PHJS.global et PHJS.session à la différence près qu’elle n’est ni globale, ni relié à la session. Un champ stocké dans PHJS.includeVars dans un script PHjs parent sera accessible par tous les scripts PHjs fils includés par le script parent.

Appeler une fonction, une classe ou une variable depuis un autre fichier :

Si on se contente d’appeler un autre script PHjs avec la fonction include, les variables, fonctions et classes créer dans ce dernier, ne seront pas accessible depuis le script parent, il y a donc deux solutions :

* Utiliser un script PHjs dans lequel on stocke les fonctions/classes/variables que l’on souhaite transmettre au parent dans ‘PHJS.includeVars’, ainsi tout le contenu de ‘PHJS.includeVars’ sera accessible par le script parent.
* Créer un fichier javascript « normal » en .js avec un module.exports pour exporter ce que l’on veut exporter, en faisant un « require » depuis le script PHjs.

Variables de PHJS dont nous n’avons parlés :

PHJS.cd : le répertoire, en chemin absolu du script PHjs en cours d’executions

PHJS.fileName : le nom du fichier du script PHjs en cours d’executions

PHJS.ipsrc : l’ip source de la personne connectée au serveur.

Fichier de configuration :

Le fichier de configuration est optionnel, et il suffit de mettre ‘nothing’ dans le start.js si on ne souhaite pas s’en servir.

Si on s’en sert, il permet de :

* Autoriser ou interdire certains fichiers ou dossiers.
* Rediriger certains fichiers ou dossiers vers d’autres liens.
* Dire à PHjs d’exécuter certains types de fichiers comme d’autres types de fichiers (par exemple si on souhaite exécuter les fichiers phjs comme des fichiers html normaux dans un dossier spécifique)
* Activer et configurer ‘FastUpload’, un outil d’upload en temps réel utilisant socket.io et intégré à PHjs.

Voici des exemples de syntaxes :

* Pour interdire des dossiers :

forbidden: /folder1, /folder2; (bloque l’accès à /folder1 et /folder2)

* Pour autoriser des dossiers :

allow: /folder1/folder3; (dans ce cas, tout le contenu de /folder1 sera bloqué sauf celui de /forder1/folder3)

* Pour rediriger des dossiers :

redirect: /folder4 to http://github.com/AutoCoderr;

* Pour appliquer des paramètres à des fichiers dans un dossier :

option(/folder1/folder3): file \*.mp4 is forbidden, file \*.phjs as text/html, file \*.png redirect <https://www.google.com/>, file \*.html openInstead /index.html; recursive yes

Dans cet exemple, nous appliquons des paramètres au dossier ‘/folder1/folder3’, de manière récursive (dans tous ses sous dossiers).

Tous les fichiers .mp4 sont interdits (avec la directive ‘is forbidden’).

Tous les fichiers .phjs sont executés comme des fichiers html normaux (avec la directive ‘as text/html’).

Tous les fichiers .png sont redirigés vers <https://www.google.com> (avec la directive ‘redirect’).

Et lorsqu’un utilisateur souhaite ouvrir un fichier .html, PHjs lui retourne le contenu de /index.html avec la directive ‘openInstead’.

Et ce, dans le dossier ‘/folder1/folder3’ ainsi que dans tous ses sous dossiers avec la directive ‘recursive yes’ après le ‘;’.

FastUpload (upload en temps réel):

‘FastUpload’ est un outil que j’ai intégré au serveur PHjs et qui permet d’uploader des fichiers en temps réel en utilisant socket.io

Pour l’utiliser, voici la syntaxe à mettre dans le fichier de configuration :

fastupload: id unId maxsize laTailleMaximaleEnOctets finish /Chemin/vers/finish.js forbidden fichiersInterdits allow fichiersAutorises;

id : l’id que vous mettez à ce fastupload, car il est possible de mettre plusieurs config de fastUpload avec chacune son id unique.

maxsize : la taille maximale d’un fichier uploadé exprimée en octets

finish : le chemin vers le fichier contenant la fonction à exécuter coté serveur une fois le fichier uploadé

forbidden : les noms de fichier interdits

allow : les noms de fichier autorisé

Un exemple :

fastupload: id film maxsize 2147483648 finish /projects/PHjs/fastupload/finish.js forbidden \* allow \*.mp4,\*.MP4,\*.avi,\*.AVI;

Nous avons créé une config avec l’id ‘film‘, un fichier ne peut pas dépasser 2147483648 octets (2 Go), le fichier /projects/PHjs/fastupload/finish.js s’execute une fois le fichier uploadé et tous les fichiers sont interdit, sauf les fichier .mp4, .MP4, .avi et .AVI .

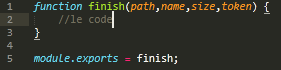
Vous pouvez aussi mettre plusieurs id avec le ‘&’ :

fastupload:

id film maxsize 2147483648 finish /projects/PHjs/fastupload/film/finish.js forbidden \* allow \*.mp4,\*.MP4,\*.avi,\*.AVI &

id image maxsize 52428800 finish /projects/PHjs/fastupload/image/finish.js forbidden \* allow \*.jpg,\*.JPG,\*.jpeg,\*.JPEG,\*.png,\*.PNG ;

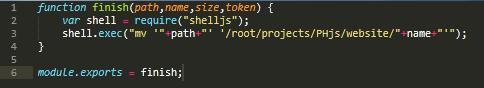
Ce que doit contenir finish.js :



La fonction finish prend en paramètre :

* path : le chemin vers le fichier venant d’être uploadé (dans les fichier temporaires)
* name : le nom du fichier uploadé
* size : sa taille
* token : il s’agit du token de l’utilisateur, s’il n’est pas spécifié coté client, il contient une chaine de caractères vide.

Un exemple de ce que peut faire la fonction finish :



Il copie tout simplement le fichier des fichiers temporaire à un dossier du site.

FastUpload coté client :

Pour l’utiliser coté client, il suffit d’importer :

* jquery
* socket.io
* fastupload.js

Les trois librairies sont déjà intégrées à PHjs, vous n’êtes donc pas obligé d’utiliser un CDN ou de les télécharger vous-même, sauf si vous voulez être sûre que jquery et socket.io soit à jour

Un exemple :



Premièrement, ont créer l’input ‘type file’ en lui mettant un id (‘leFichier’), qui sera spécifié à fastupload.

Ensuite, on créer une div, avec un id (‘progression’) qui sera lui aussi spécifié à fastupload, c’est là que s’affichera la barre de progression.

Dans le javascript, on créer une nouvelle instance de la classe ‘FastUpload()’ (ligne 14).

Puis en exécute la fichier ‘setUpload()’, en lui donnant comme arguments :

* L’adresse du serveur
* L’id de l’input du fichier
* L’id du div où s’affichera la barre de progression
* Le callback, qui s’exécutera une fois l’upload terminé
* L’id du fastupload coté serveur
* Et le token, présent, dans notre cas, dans ‘PHJS.session.token’

Sans oublier de mettre un bouton, dans le html, qui lance la fonction ‘upload()’ (ligne 6)