

TP 2 - WebServices

Fournisseur

Contexte

Une API de WebServices très rudimentaire vous est fournie sous forme d'un environnement Dockerisé. A vous de l'exécuter sur votre machine, afin de vous servir de point de départ. Pour ce faire, lisez attentivement ce qui suit.

Cette API utilise un framework nommé **Slim**. Il s'agit d'un framework simple mais efficace et suffisant pour écrire des WebServices.

L'objectif de ce TP n'est pas de vous apprendre à utiliser **Slim** mais uniquement à coder des WS. Vous serez guidés pour la petite partie technique de **Slim**. Si vous souhaitez en apprendre plus sur ce framework, vous pourrez visiter ultérieurement le site www.slimframework.com.

Supprimez un éventuel volume Docker qui aurait été laissé par un-e autre étudiant-e :

```
| docker volume rm ws_tp2
```

Créez votre volume Docker de travail :

```
| docker volume create ws_tp2
```

Dans un terminal, récupérez l'image **r401-slimws** (sans tag, ce sera donc la **latest**) :

```
| docker image pull r401-slimws
```

Dans le terminal, démarrez l'environnement Docker (à taper sur une seule ligne) :

```
| docker container run --rm -p 8080:80 -v ws_tp2:/work r401-slimws
```

Une fois que l'environnement Docker est lancé, ouvrez votre navigateur sur l'URL **http://localhost:8080**. Vous devez obtenir un message qui ne laisse aucun doute sur le fait que le serveur Web qui va délivrer votre API fonctionne bien.

Comme vous en avez désormais l'habitude, pour accéder à votre volume Docker depuis votre machine hôte, vous devez utiliser la technique de **VSC + plugin Remote Development**.

Vérifiez maintenant que vous avez un fichier **api.php** qui est apparu dans votre volume **ws_tp2** (qui est monté sur **/work** dans le conteneur). Pour ce TP, c'est uniquement ce

script **api.php** que vous allez modifier. Vous pouvez donc l'ouvrir dans VSC et le garder ouvert pour y travailler en direct.

Si vous supprimez ce script et que vous redémarrez votre conteneur Docker, il réapparaîtra dans sa forme initiale.


Ce script étant dans un volume, il **NE suivra PAS** automatiquement votre profil. N'oubliez donc pas d'en faire une copie sur votre machine hôte en fin de séance !

Postman - Complément

Dans le TP précédent, vous avez appris à utiliser **Postman** avec une API sur Internet (**restcountries.com**).

Dans ce TP vous allez aussi utiliser Postman mais sur une API hébergée localement sur votre poste de travail, c'est-à-dire en **localhost**.

A l'IUT, pour accéder à Internet, vous passez par un proxy. Postman sait aussi utiliser le proxy, ce qu'il a d'ailleurs fait dans le TP1, mais pour accéder à **localhost**, vous devez désactiver son utilisation du proxy. Voici comment procéder :

- Rendez-vous dans **Settings** dans l'icône  en haut-droite de la fenêtre de Postman
- Allez ensuite dans **Proxy** et décochez la case ☐ **Use system proxy**

Vous n'oublierez pas de recocher cette case quand vous aurez besoin d'accéder à une API sur Internet, à l'avenir.

Exercice 1 - GET

Observez le contenu du script **api.php** et essayez de comprendre ce que fait ce code exactement.

Il contient 2 API :

- **GET /articles**
- **GET /articles/{id}**

Pour cet exercice, vous allez simplement observer le contenu de chaque fonction et vous allez tester dans un premier temps avec votre navigateur (puisque les 2 API sont des **GET**) puis avec **Postman** dans un second temps (consultez et inspirez-vous du TP 1 sur **WebServices - Consommateur**).

Exercice 2

Vous allez adapter et compléter le code de **api.php** ainsi :

Etape 1

Le stockage des données (**articles**) doit utiliser un fichier **JSON** au lieu d'un tableau en dur comme c'est actuellement le cas dans **api.php**.

Créez un fichier **data.json** (au même endroit que **api.php**) et placez-y ces données. Observez bien comment ce fichier est le reflet du tableau **\$articles** :

```
{
    "1": {
        "nom": "Livre"
    },
    "2": {
        "nom": "Crayon"
    }
}
```

Mettez en commentaire le tableau **\$articles** actuel et chargez le fichier **data.json** de la façon suivante :

```
$articles = json_decode(file_get_contents(__DIR__ . '/data.json'),
true);
```

L'appel à **file_get_contents()** permet de lire le fichier **data.json** et retourne l'intégralité du fichier sous forme d'une chaîne de caractères.

Le **__DIR__** est une variable globale de **PHP** qui contient le chemin complet où se trouve le script qui en fait usage. Ici ce sera donc **/work** (puisque l'on est dans le conteneur, voir l'option **-v** passée au **docker container run** précédemment).

Le **true** permet d'indiquer à **json_decode()** de renvoyer un tableau et non pas des objets (voir la documentation sur **php.net**) lors du décodage de la chaîne **JSON** lue dans **data.json**.

En plaçant ce code à la place de l'ancien tableau en dur, vous lirez vos données depuis un fichier, à chaque exécution du script.

Vérifiez qu'un **GET articles** fonctionne toujours correctement dans **Postman**.

Etape 2

Ajoutez ces attributs supplémentaires (et des valeurs pour chacun d'eux) à vos articles dans **data.json** :

- **qte** : une quantité disponible (valeur entière ≥ 0)
- **prix** : un prix (valeur réelle > 0)

Testez.

Exercice 3

Dans cet exercice, vous allez compléter le code de **api.php** pour traiter un **POST**.

Comme vous le savez, un **POST** sert à créer des ressources.

Le script **api.php** est exécuté à chaque requête d'API. Cela signifie qu'une fois la requête traitée, le script s'arrête et qu'il n'y a aucune conservation de données en mémoire entre 2 requêtes !

Ainsi, le tableau des articles qui est chargé depuis le fichier **data.json** en début de script doit donc être sauvegardé dans **data.json** en fin d'exécution du script si on veut conserver toute modification telle qu'un ajout d'article. C'est ce que vous devez faire dans cet exercice.

Etape 1

Créez une fonction **save()** qui sauvegarde le tableau d'articles dans **data.json**. Vous devez opérer de façon réciproque à ce qui a été fait pour le chargement (voir la documentation sur **php.net**) :

- **json_encode()** : transforme un tableau PHP en une chaîne au format JSON
- **file_put_contents()** : écrit une chaîne de caractères dans un fichier

Vous utiliserez cette fonction **save()** à chaque fois que votre API aura modifié les données du tableau d'articles.

Etape 2

Vous allez maintenant coder le **POST** qui permet de créer un nouvel article (n'oubliez pas d'appeler **save()** pour sauver votre tableau d'article à la fin)

Voici de l'aide :

- En vous inspirant du **GET**, vous devez ajouter une fonction :

```
$app->post('/articles', function ($req, $resp, $args) {...});
```

- Exemple de récupération des données **POST**ées :

```
$params = $req->getParsedBody();
```

Pour voir ce que vous recevez dans **\$params**, faites ceci pour le moment :

```
print_r($params);
```

- Un **POST** ne renvoie pas de données mais répond par un code. Vous devez placer ceci au lieu du **return** utilisé dans les **GET** :

```
| return $resp->withStatus(200);
```

Notez que le code **200** signifie que tout s'est bien déroulé. Il existe des codes d'erreurs aussi. Consultez ce site pour une liste détaillée :

https://fr.wikipedia.org/wiki/Liste_des_codes_HTTP

S'agissant de la création d'une ressource, cherchez un code plus approprié que **200**.

Testez votre **POST** à l'aide de **Postman** en créant une requête adaptée. Cherchez dans **Postman** ce qui peut correspondre à ça :

- Le verbe doit être **POST**
- Les paramètres doivent être passés dans le **Body** au format **x-www-form-urlencoded**
- Les paramètres sont passés sous la forme : **clé/valeur**

Vérifiez que votre fichier **data.json** est bien rempli avec les nouvelles données **POSTées**.

Exercice 4

Poursuivez l'implémentation de votre API **articles** par l'ajout des actions suivantes :

- **PUT** et **PATCH** : pour mettre à jour un article existant
- **DELETE** : pour supprimer un article existant

Vous aurez, cette fois-ci, à gérer les erreurs comme "Article inconnu". Ceci se fait au moyen de codes **HTTP** adaptés (voir **Exercice 3**)