GUIDE
CRÉER ET EXPLOITER UNE API REST

Contenu

À travers un exemple simple en PHP, vous allez voir les bases de la création d'une API REST et de son exploitation.

Pour rappel il est possible de créer une API REST dans n'importe quel langage et l'exploiter dans n'importe quel langage.

1. Créer une API REST

Cette API va plutôt servir de structure de base pour créer d'autres API en PHP. Le but est de voir le principe.

Commencez par lancer WampServer (ou équivalent).

Dans le dossier www (ou équivalent), créez un dossier 'rest_test' dans lequel vous allez mettre 2 dossiers : 'api' et 'client'. Cela permettra de mémoriser l'api (dans 'api') et le programme qui interroge l'api (dans 'client').

1A. Coder l'API

Dans le dossier 'api', créez le fichier 'monapi.php'.

Dans ce fichier, commencez par insérer l'entête qui permet de préciser que le retour sera en JSON :

```
header('Content-Type: application/json');
```

Pour respecter les bonnes pratiques, le but est d'afficher (donc de retourner) au final une réponse au format JSON, constituée d'un tableau associatif contenant :

- un code de retour (200 si ok...) ;
- un message de retour (normalement, en rapport avec le code) ;
- un résultat éventuel.

Autant créer une fonction qui reçoit ces 3 valeurs en paramètre, crée le tableau associatif correspondant et affiche l'information encodée en JSON :

```
function reponse($code, $message, $result=""){
    $retour = array(
        'code' => $code,
        'message' => $message,
        'result' => $result
    );
    echo json_encode($retour, JSON_UNESCAPED_UNICODE);
}
```

Le paramètre JSON_UNESCAPED_UNICODE permet de gérer l'encodage (et donc de garder les caractères accentués).

Suivant l'API, elle peut avoir besoin d'une ou de plusieurs informations en entrée. Par exemple, une API qui gère des produits peut attendre des informations pour récupérer tous les produits ou un produit précis, ajouter un produit, le modifier, voire le supprimer.

On pourrait se contenter de tout gérer en GET ou en POST. Mais encore une fois, pour se rapprocher des bonnes pratiques de l'écriture d'une API REST, l'idée va être de tester quel verbe a été utilisé pour appeler l'API.

En rappel, on a principalement 4 verbes et leur utilisation conseillée :

- GET : pour récupérer des données ;
- POST : pour ajouter des données ;
- PUT : pour modifier des données ;
- DELETE : pour supprimer des données.

Du coup, par une succession de tests, il est possible de savoir quel verbe a été utilisé et ajouter le code désiré (à placer sous la fonction) :

```
if($_SERVER['REQUEST_METHOD'] === 'GET'){
    // traitements de récupération

}else if($_SERVER['REQUEST_METHOD'] === 'POST'){
    // traitements d'ajout

}else if($_SERVER['REQUEST_METHOD'] === 'PUT'){
    // traitements de modification

}else if($_SERVER['REQUEST_METHOD'] === 'DELETE'){
    // traitements de suppression

}
```

Dans le cas du verbe `'POST'`, les éventuels paramètres arrivent en mode `'POST'`. Dans tous les autres cas, ils arrivent en mode 'GET'.

On va partir du principe que cette API attend deux paramètres : un nom de table et un id. On pourrait vérifier si on reçoit `$_GET["table"]` ou `$_POST["table"]`, même chose pour "id". Sachant que `$_REQUEST` permet de récupérer les deux types de paramètres, écrivez juste au-dessus de ces tests (et sous la fonction), le code qui permet de récupérer le paramètre, s'il existe :

```
$table = "";
if(isset($_REQUEST['table'])) {
    $table = $_REQUEST[‘table’];
}
$id = "";
if(isset($_REQUEST['id'])) {
    $id = $_REQUEST['id'];
}
```

Important

Que ce soit avec `$_GET`, `$_POST` ou `$_REQUEST`, on n'est pas dans une bonne pratique de récupération sécurisée des informations. Ici, le but est de simplifier le code pour se concentrer sur l'API, mais en réalité, pour éviter les injections SQL et nettoyer le contenu des variables reçues, il faudrait plutôt écrire :

```
$table = filter_input(INPUT_GET, 'table', FILTER_SANITIZE_STRING) ??
         filter_input(INPUT_POST, 'table', FILTER_SANITIZE_STRING);
```

On obtiendrait, dans $table, soit null (si la variable n'existe pas), soit un contenu "propre" (par exemple, avec `FILTER_SANITIZE_STRING` en cas de présence de texte pouvant exécuter une commande, celui-ci est tout simplement supprimé, et les caractères spéciaux sont échappés). Il existe d'autres filtres. Vous les trouverez tous ici :

https://www.php.net/manual/fr/filter.filters.sanitize.php

En plus, filter_input évite le isset() qui permet de vérifier si la variable existe, et cela apporte un niveau supplémentaire de sécurité. Cela ne doit pas vous empêcher de gérer ensuite correctement les requêtes SQL pour éviter les injections SQL.

Même chose pour `$id`.

Rien qu'avec ces 2 paramètres, on pourrait imaginer plusieurs possibilités suivant le verbe HTTP utilisé :

- GET, s'il n'y a que le nom de la table, on retourne tous les tuples : `"select * from $table" ;`
- GET, s'il y a les 2 paramètres, on retourne un tuple : `"select * from $table where id=$id" ;`
- DELETE, s'il n'y a que le nom de la table, on vide la table : `"delete from $table" ;`
- DELETE, s'il y a les 2 paramètres, on supprime un tuple : `"delete from $table where id=$id".`

Pour les ajouts et modifications, on peut imaginer qu'un paramètre est au format JSON à décomposer pour récupérer toutes les valeurs à ajouter ou modifier.

C'est juste pour expliquer le principe : dans cet exemple, on ne va pas directement coder les accès à une BDD.

Important

Même avertissement que précédemment.
Ces requêtes sont là pour que vous compreniez la logique d'utilisation des paramètres, mais bien sûr, écrites sous cette forme, ne sont absolument pas sécurisées et donc ouvertes aux injections SQL.
Je vous renvoie vers la partie sécurité abordée dans l'application Symfony (B2.1 séquence 5, séance 4).

Donc, à partir de là, pour finaliser cette API de test, le but est juste, pour chacun des 4 cas, d'appeler la méthode réponse en faisant en sorte d'afficher le verbe utilisé pour voir s'il a été bien pris en compte, et le contenu des paramètres :

```
if($_SERVER['REQUEST_METHOD'] === 'GET'){
    // traitements de récupération
    reponse('200', 'OK', "GET table=$table id=$id");
}else if($_SERVER['REQUEST_METHOD'] === 'POST'){
    // traitements d'ajout
    reponse('200', 'OK', "POST table=$table id=$id");
}else if($_SERVER['REQUEST_METHOD'] === 'PUT'){
    // traitements de modification
    reponse('200', 'OK', "PUT table=$table id=$id");
}else if($_SERVER['REQUEST_METHOD'] === 'DELETE'){
    // traitements de suppression
    reponse('200', 'OK', "DELETE table=$table id=$id");
}
```

L'API de test est terminée.

1B. Tester l'API

Avant d'écrire le code d'un client qui exploite cette API, vous allez utiliser un outil qui permet de faire les tests. En effet, excepté pour GET, on ne peut pas faire de test directement par l'URL.

D'ailleurs vous pouvez faire le test de GET en tapant l'adresse suivante dans le navigateur :

```
http://localhost/rest_test/api/monapi.php?table=produit&id=3
```

Vous devriez obtenir ceci :

```
{"code":"200","message":"OK","result":"GET table=produit id=3"}
```

Cela prouve qu'il a bien reçu l'url en GET, et le contenu des paramètres a été récupéré.

Remarquez que vous obtenez bien le résultat au format JSON. D'ailleurs, suivant les navigateurs et la façon dont ils ont été paramétrés, vous avez peut-être un affichage plus "joli" :

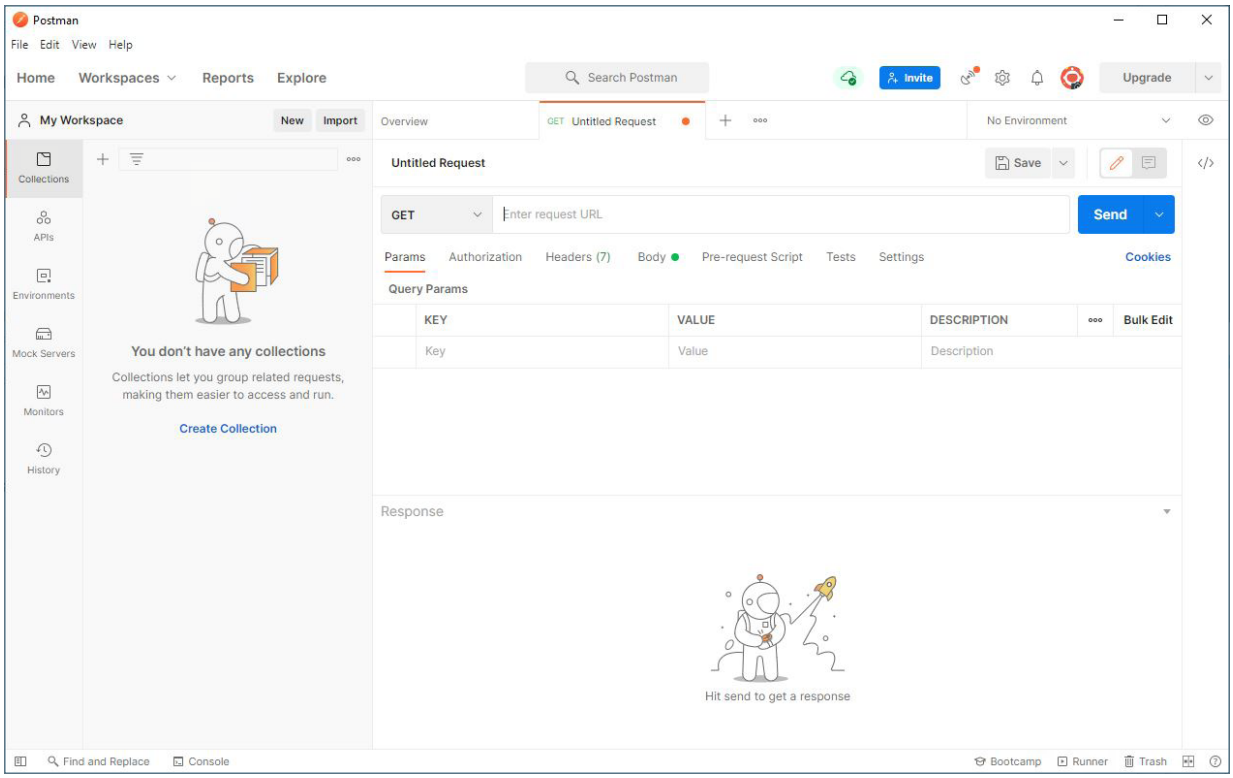
```
code      "200"
message   "OK"
result    "GET table=produit id=3"
```

Cela vient du fait que vous avez précisé le format JSON, en en-tête du fichier php.

Vous pouvez aussi tester avec un seul paramètre, voire aucun. En retour, vous aurez toujours le message, mais sans aucune information à droite du "=" si vous n'avez pas renseigné le paramètre correspondant.

Pour pouvoir tester les autres verbes sans écrire le code d'un client, le plus simple est d'utiliser le logiciel Postman que vous pouvez télécharger gratuitement. Allez sur le site officiel de Postman (https://www.postman.com/), connectez-vous (c'est gratuit) et une fois la console ouverte, prenez le lien "Download Desktop App" qui normalement est en bas à gauche.

Une fois le logiciel installé et lancé, vous devriez obtenir quelque chose qui ressemble à ceci :



Remarquez, dans la zone centrale :

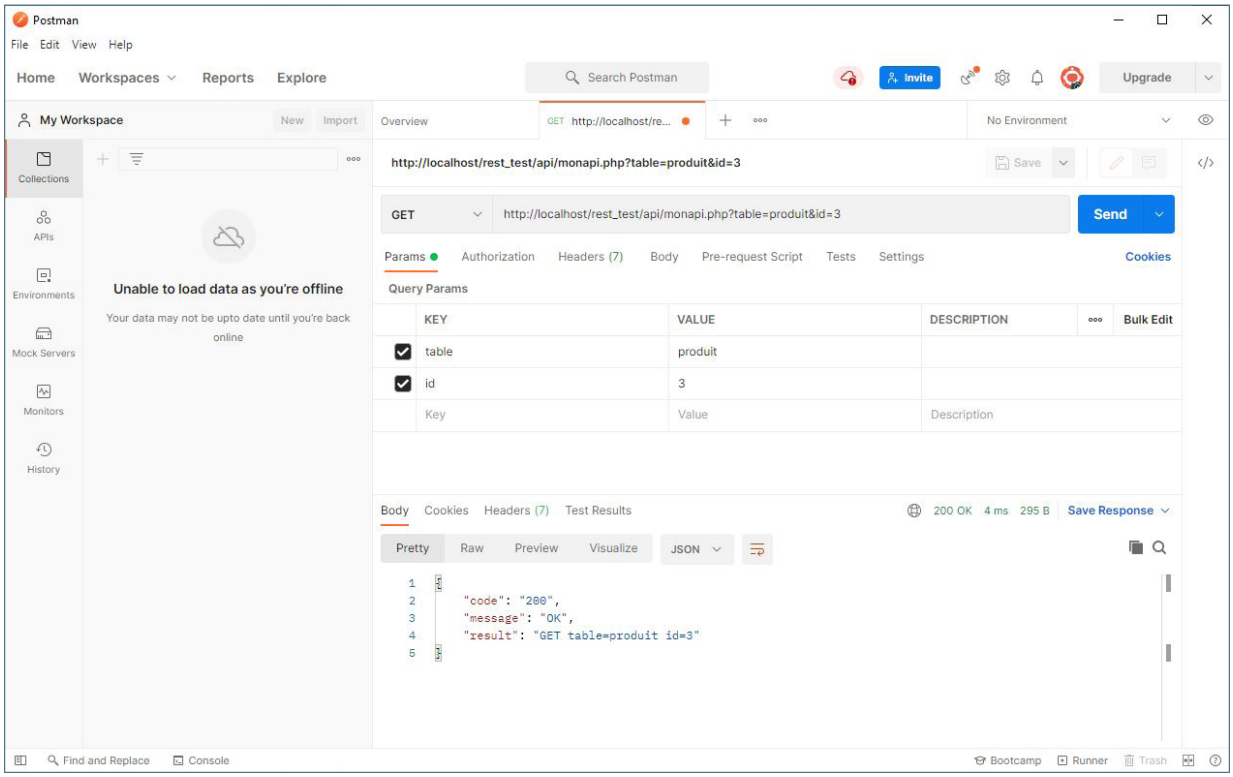
- le verbe GET avec une petite flèche qui vous permet de choisir d'autres verbes ;
- à droite de GET, une zone de saisie pour taper l'url de l'API ;
- à droite de la zone de saisie, le bouton Send pour tester ;
- en dessous, un menu contenant, entre autres, "Params" pour saisir des paramètres de type GET, "Body" pour saisir des paramètres de type POST ;
- en dessous, un tableau pour saisir les paramètres (KEY et VALUE) ;
- en dessous, l'espace "response" pour afficher le résultat obtenu.

Faites un premier test.

Gardez le verbe GET et, dans l'url, tapez :

```
localhost/rest_test/api/monapi.php
```

Puis, dans l'onglet "Params", ajoutez le paramètre 'table' avec la valeur que vous voulez, idem pour le paramètre 'id' avec une valeur numérique, en 2e ligne. Remarquez que les paramètres se sont ajoutés dans l'url. Cliquez sur Send. Vous devriez obtenir le même résultat que précédemment :



Maintenant vous allez pouvoir tester les autres verbes. Gardez les mêmes paramètres et testez POST, PUT puis DELETE. Vous devriez obtenir respectivement le message (partie "result") commençant par "POST", "PUT" ou "DELETE", suivi du contenu des paramètres.

En réalité, cela ne devrait pas marcher avec POST. Mais cela vient du fait que les paramètres s'ajoutent dans l'URL quand ils sont mis dans l'onglet "Params", et que du coup, à la réception, $_REQUEST le récupère.

Pour vous persuader qu'il n'y a que le verbe POST qui peut récupérer un paramètre envoyé en POST, supprimez les paramètres du tableau des "Params" (en cliquant sur la croix à droite sur la ligne, qui apparait quand vous mettez la souris sur la ligne) puis ajoutez ces paramètres dans l'onglet "Body". Testez à nouveau les 4 verbes : vous constaterez que le contenu des paramètres ne s'affiche qu'avec le verbe POST.

1C. URL Rewriting

Une des bonnes pratiques concernant l'API REST préconise la réécriture d'URL de façon à simplifier son écriture et la rendre aussi plus lisible.

Voyons directement un exemple.

Pour remplacer :

```
localhost/rest_test/api/monapi.php
```

On aimerait plutôt écrire l'URL sous cette forme :

```
http://localhost/rest_test/api/produit/3
```

Avec cette écriture, plus besoin de connaître le nom du fichier PHP à appeler dans le dossier "api" et plus besoin de connaître les noms des paramètres. Il faut juste savoir qu'on peut mettre 2 valeurs : un nom de table et un id. On peut ne mettre aussi qu'une seule valeur, voire aucune. C'est nettement plus simple.

Pour que ça marche, il faut créer un fichier ".htaccess" dans le dossier du projet. Ce type de fichier de configuration permet de faire des tas de choses et, entre autres, écrire des règles de réécriture d'URL.

Pour cela, dans le dossier "api", commencez par créer un fichier "htaccess.txt" et commencez par remplir cette première ligne :

```
RewriteEngine on
```

Cela permet d'activer la possibilité de réécriture d'URL, à condition que le serveur l'accepte. Sous Wamp, normalement cette option est activée par défaut. Vous pouvez le vérifier en faisant un clic gauche sur l'icône de Wamp > Apache > Modules Apache, et regardez si "Rewrite_module" est coché. Sur un serveur distant, il faudra contrôler que cette option est active.

Maintenant il est possible d'écrire les règles.

Première règle : si l'utilisateur ne tape que le chemin du dossier, il doit être redirigé vers "monapi.php" :

```
RewriteRule ^$ monapi.php
```

'RewriteRule' permet de définir une règle de réécriture.

Celle-ci se décompose en 2 parties :

- la première partie représente ce que saisit l'utilisateur (entre '^' qui représente le début et '$' la fin) ;
- la seconde partie (mise à la suite séparée par au moins un espace) représente la redirection ver le chemin réel.

Du coup cette première règle dit que si l'utilisateur ne tape rien, cela correspond à "monapi.php".

Seconde règle : si l'utilisateur tape le chemin, suivi d'un "/" et d'un nom de table, il doit être redirigé vers "monapi.php?table=" suivi du nom saisi (exemple : `http://localhost/rest_test/api/produit`).

```
RewriteRule ^([a-zA-Z]+)$ monapi.php?table=$1
```

Pour la première partie (ce que saisit l'utilisateur), on utilise une expression régulière.

`a-z` : représente tous les caractères de a à z.
`A-Z` : idem en majuscule.
`[ ]` : permet de regrouper une règle (si on n'en met pas plus, on attend un seul caractère alphabétique).
`+` : mis à la suite de [ ], permet de dire que le contenu peut se répéter 1 à plusieurs fois (donc plusieurs caractères alphabétiques).
`( )` : permet de définir une variable qui sera utilisée dans la seconde partie (la redirection). Dans la seconde partie, les variables sont notées $ suivi d'un numéro, dans l'ordre des parenthèses (car il peut y en avoir plusieurs).

Pour la seconde partie, on remarque que le fichier "monapi.php" est appelé, avec le paramètre "table" qui reçoit $1 en valeur ($1 correspond donc au contenu des parenthèses de la première partie, donc à ce que l'utilisateur a saisi).

Troisième règle : si l'utilisateur tape le chemin, suivi d'un premier "/", puis d'un nom de table, puis d'un second "/" et enfin d'un numéro, il doit être redirigé vers "monapi.php?table=" + nom de la table saisi + "&id=" + numéro saisi (exemple : `http://localhost/rest_test/api/produit/3`).

```
RewriteRule ^([a-zA-Z]+)/([0-9]+)$ monapi.php?table=$1&id=$2
```

Avec les explications précédentes, vous devriez comprendre le fonctionnement de cette nouvelle règle.

Enregistrez le fichier puis renommez-le en ".htaccess". N'oubliez pas le "." du début et enlevez bien l'extension ".txt" : pensez à configurer votre explorateur de fichiers pour voir les extensions des fichiers.

Si vous refaites un test avec les anciennes URL, cela ne devrait plus marcher. En revanche, si vous testez les nouvelles URL (vide ou avec un nom de table ou aussi avec un numéro, le tout séparé par des "/" comme dans les exemples ci-dessus), quel que soit le verbe HTTP, vous devriez obtenir les bons messages.

Voici un lien pour avoir les informations nécessaires sur l'url rewriting :

https://httpd.apache.org/docs/2.4/rewrite/intro.html

1D. Documenter l'API

Pour qu'une API puisse être utilisée, il faut créer une documentation. Généralement elle est présentée dans une page accessible par Internet.

La documentation doit contenir :

- l'adresse de l'API ;
- les éventuels endpoints (points d'entrée) pour accéder à différentes fonctionnalités qui sont expliquées : ils vont alors s'ajouter à l'adresse précédente sous la forme "/endpoint". Ces endpoints doivent permettre d'accéder par exemple à différents fichiers ou sous-dossiers ;
- les paramètres éventuels et leurs rôles ;
- des exemples de requêtes et de réponses.

2. Exploiter une API REST avec un client PHP

Dans le sous-dossier 'client', créez le fichier 'index.php'.

Dans ce fichier, vous allez mettre le code du client qui exploite l'API précédente. Le but va juste être de tester l'envoi d'une information en testant les 4 verbes.

Une fonction pour gérer les verbes HTTP

Avant tout, on a besoin d'une fonction qui est justement capable d'envoyer une information sur une url, avec des verbes HTTP différents. Cette fonction va recevoir 3 paramètres, la méthode d'envoi qui est le verbe, l'url et éventuellement l'information (data) à envoyer.

Voici le code de la fonction, suivi d'explications :

```php
/**
 * Method: POST, PUT, GET etc
 * data: array("param" => "value") ==> index.php?param=value
 */
function callAPI($method, $url, $data = false)
{
    $curl = curl_init();

    switch ($method)
    {
        case "POST":
            curl_setopt($curl, CURLOPT_POST, 1);
            if ($data)
                curl_setopt($curl, CURLOPT_POSTFIELDS, $data);
            break;
        case "PUT":
            curl_setopt($curl, CURLOPT_PUT, 1);
            if ($data)
                $url = sprintf("%s?%s", $url, http_build_query($data));
            break;
        case "DELETE":
            curl_setopt($curl, CURLOPT_CUSTOMREQUEST, "DELETE");
        default:
            if ($data)
                $url = sprintf("%s?%s", $url, http_build_query($data));
    }

    curl_setopt($curl, CURLOPT_URL, $url);
    curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);

    $result = curl_exec($curl);

    curl_close($curl);

    return $result;
}
```

La fonction `'curl_init()'` permet de créer un objet qui va se connecter à une url.

Suivant le contenu de `$method`, différentes actions sont faites.

Avant tout, excepté pour `GET` qui est le verbe par défaut, il faut préciser le verbe avec `'curl_setopt'`.

Ensuite, dans le cas particulier de `POST`, les données (`$data`) sont envoyées en `POST` (option `CURLOPT_POSTFIELDS`). Pour les autres verbes, l'envoi est ajouté dans l'url.

Remarquez le cas du `DELETE` qui n'a pas de break pour gérer aussi les paramètres dans l'url.

En effet, le default correspond aux autres cas, donc ici, au verbe `DEL`.

Ensuite, d'autres options sont renseignées.

Le plus important est l'exécution qui permet d'appeler l'url et de récupérer le résultat dans `$result`.

Il suffit ensuite de retourner ce résultat, après avoir proprement fermé le lien vers l'url.

2A. L'envoi de paramètres sans réécriture d'URL

Commençons par tester l'envoi classique de paramètres.

Après cette fonction, créez une variable qui permet d'enregistrer l'url de l'API :

```
$url = "http://localhost/rest_test/api/monapi.php";
```

Comme expliqué en commentaire de la fonction, le paramètre `$data` doit être sous la forme d'un tableau associatif. Créez un tableau associatif pour mémoriser le paramètre 'info' :

```
$infos = array(
    'table' => 'produit',
    'id' => 3
)
```

Il ne reste plus qu'à tester les 4 verbes en appelant 4 fois la fonction précédente :

```
echo(callAPI('GET', $url, $infos)."<br>");
echo(callAPI('POST', $url, $infos)."<br>");
echo(callAPI('PUT', $url, $infos)."<br>");
echo(callAPI('DELETE', $url, $infos));
```

Avant de lancer le test dans le navigateur, enlevez le fichier ".htaccess" du dossier "api" et copiez-le ailleurs.

Dans le navigateur, appelez votre client :

```
http://localhost/rest_test/client/index.php
```

Vous devriez obtenir :

```
{"code":"200","message":"OK","result":"GET coucou"}
{"code":"200","message":"OK","result":"POST coucou"}
{"code":"200","message":"OK","result":"PUT coucou"}
{"code":"200","message":"OK","result":"DELETE coucou"}
```

Évidemment, c'est juste un test. Vous pouvez maintenant imaginer le code de l'API mettant à jour une BDD suivant les verbes et informations reçues, et le code du client demandant de mettre à jour la BDD.

2B. La version avec la réécriture d'URL

Faites un copier/coller du fichier index.php (pour mémoriser la version précédente) et modifiez le contenu de index.php.

Cette fois, il n'y aura plus de paramètres à envoyer. Vous pouvez supprimer le tableau $infos et le troisième paramètre dans les 4 appels de la méthode callAPI. Modifiez l'URL qui doit contenir directement les paramètres :

```
$url = "http://localhost/rest_test/api/produit/3";
```

Pour tester, pensez à remettre le fichier ".htaccess" dans le dossier "api".

2C. L'utilisation de formulaires pour accéder à l'API

Pour rappel, les formulaires HTML ne permettent que 2 méthodes d'envoi : POST et GET.

La plupart des frameworks utilisent une astuce pour palier à l'absence des autres méthodes, en ajoutant un champ caché au formulaire POST, contenant le nom "_method" et le contenu "PUT ou "DELETE" :

```
<form method="POST" action="http://localhost/rest_test/api/monapi.php">
    <input name="_method" type="hidden" value="PUT">
    <input name="table" type="text" value=""><br>
    <input name="id" type="text" value=""><br>
    <button type="submit">Envoyer</button>
</form>
```

Cela n'empêchera pas le transfert en POST. Donc soit l'API gère ce paramètre (il faudrait que ce soit le cas dans l'exemple ci-dessus puisqu'on appelle directement l'API), soit il faut rediriger vers la même page ou une autre page du client qui ensuite envoie les informations en PUT vers l'API. Cette seconde solution est la plus logique, car, sauf dans le cas d'une API maison, a priori il n'y a pas de raison qu'elle gère le champ "_method".

3. Les outils pour générer le code

Sachant qu'il existe des normes et bonnes pratiques pour les API REST, du coup, des outils sont apparus pour générer automatiquement le code.

Ces outils existent aussi bien pour la création d'API que pour la création d'une application cliente exploitant une API.

4. Les autres variantes

Vous avez précédemment vu comment utiliser les verbes HTTP pour distinguer les types d'opérations.

Il est possible d'avoir une approche différente, en n'utilisant qu'un verbe (GET ou POST) et en distinguant les opérations par un paramètre supplémentaire ou un chemin amenant à des fichiers différents.

L'url en est plus parlante et le code allégé.

Par exemple, on pourrait avoir comme URL :

```
http://localhost/rest_test/api/delete/produit/2
```

Ou combiner l'URL rewriting et l'envoi de paramètres en POST.

Toutes les variantes sont envisageables, en s'éloignant des bonnes pratiques du REST mais parfois en gagnant en lisibilité et simplicité.

Pour que vous compreniez plus concrètement, cela donnerait au niveau de l'API, le contenu suivant :

```
<?php
header('Content-Type: application/json');

function reponse($code, $message, $result){
    $retour = array(
        'code' => $code,
        'message' => $message,
        'result' => $result
    );
    echo json_encode($retour);
}

$ordre = "";
if(isset($_REQUEST['ordre'])) {
    $ordre = $_REQUEST['ordre'];
}
$table = "";
if(isset($_REQUEST['table'])) {
    $table = $_REQUEST['table'];
}
$id = "";
if(isset($_REQUEST['id'])) {
    $id = $_REQUEST['id'];
}

reponse('200', 'OK', "ordre=$ordre table=$table id=$id");
```

On ne s'occupe plus du verbe HTTP reçu. En revanche il y a un paramètre en plus ('ordre') pour savoir ce qu'il faut faire.

De même, le client est nettement plus léger :

```
<?php

/**
 * appel d'une URL
 */
function callAPI($url)
{
    $curl = curl_init();

    curl_setopt($curl, CURLOPT_URL, $url);
    curl_setopt($curl, CURLOPT_RETURNTRANSFER, 1);

    $result = curl_exec($curl);

    curl_close($curl);

    return $result;
}
$url = "http://localhost/rest_test/api2/delete/produit/3";

echo(callAPI($url));
```

La fonction callAPI ne se préoccupe plus des différents verbes pour l'envoi.

L'url est plus parlante (avec 'recup').

On retrouve une syntaxe d'url vue dans Symfony.

CNED, BP 60200, 86980 Futuroscope Chasseneuil Cedex, France

87D22TDWB1G21