# REST API

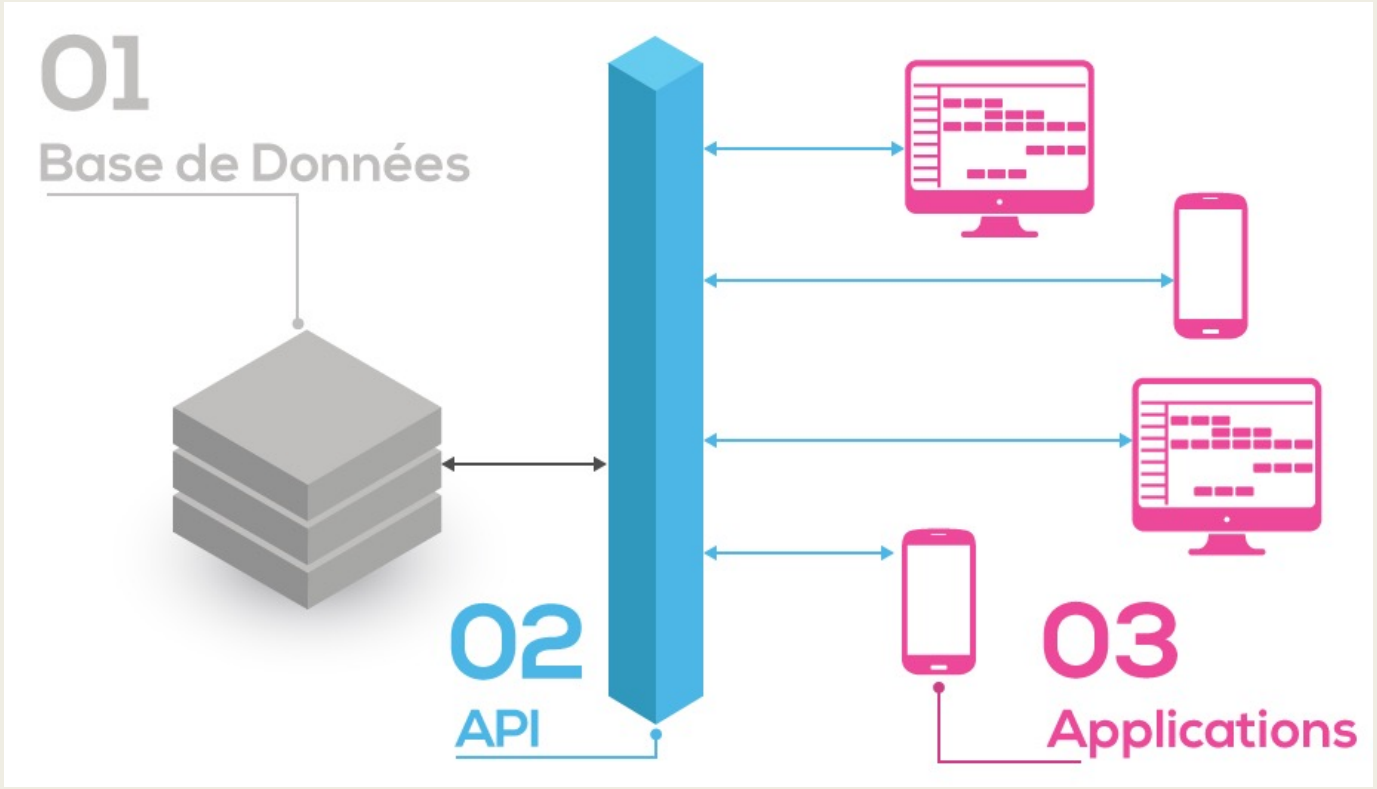
Plus sur cette architecture et sur les micro services

# UN PEU DE THÉORIE

Définissons déjà de quoi on va parler

# Déjà c’est quoi un API ?

- API : Application Programming Interface
- Le but d’une api est de servir d’intermédiaire entre deux logiciels qui veulent communiquer ensemble
- L’intérêt d’une API va être de standardiser les échanges de données qui pourront être réutilisées de plusieurs façon différentes à plusieurs endroits



# API REST ?!

- REST : Representational State Transfer
- Elles sont une alternative aux API SOAP (Simple Object Access Protocol)
- REST est en fait un ensemble de normes et un style d'architecture pour les API :
  - *Séparation Client / Server*
  - *Sans état*
  - *Cacheable*
  - *Interface uniforme*
  - *Système de couches*
  - *Architecture code-on-demand*

# Séparation Client / Serveur

- Jusque là, rien de très nouveau, c'est aussi le paradigme de ce que l'on faisait jusque là
- Le client est celui qui va utiliser l'API (une app, un navigateur…). Il va faire une requête au serveur et traiter sa réponse
- Le serveur est ce qui va être capable de traiter la demande. Il reçoit la requête du client, va chercher les informations nécessaires et renvoie une réponse

# Stateless / Sans état

- Le serveur ne doit pas sauvegarder les requêtes ou les réponses précédentes
- Chaque requête doit donc contenir TOUTES les données nécessaires au serveur pour formuler sa réponse
- L’avantage est que chaque réponse est déterminée et prévisible puisqu’elle ne tient pas compte des requêtes précédents

# Cacheable

- La réponse doit contenir l’information sur la capacité (ou non) du client à mettre en cache les données
- En architecture REST, le cache doit être mis en place dès que possible que ce soit du coté du client ou du serveur

# Interface uniforme

- Pour faire simple, une API REST doit avoir les mêmes « règles du jeu » du début à la fin
- Imaginons qu'un client ait récupéré des données client et qu'il sait qu'il existe aussi des données de commandes, il doit être capable de les récupérer en suivant me même pattern que celui suivi pour récupérer les données client
- Cela permet d'avoir une API simple à parcourir et à utiliser

# Système de couches

- Encore une fois, le but est de simplifier la vie de l'utilisateur
- Si vous décidez de mettre en place une architecture ou l'API est stockée sur un serveur A, les données sur un serveur B et l'authentification sur un serveur C, le client ne doit pas pouvoir distinguer si il est connecté directement à la fin de la chaine ou à un intermédiaire
- En gros : vous gérez vos serveurs comme vous le voulez tant que le client ne voit pas de différence

# Code-on-demand (option)

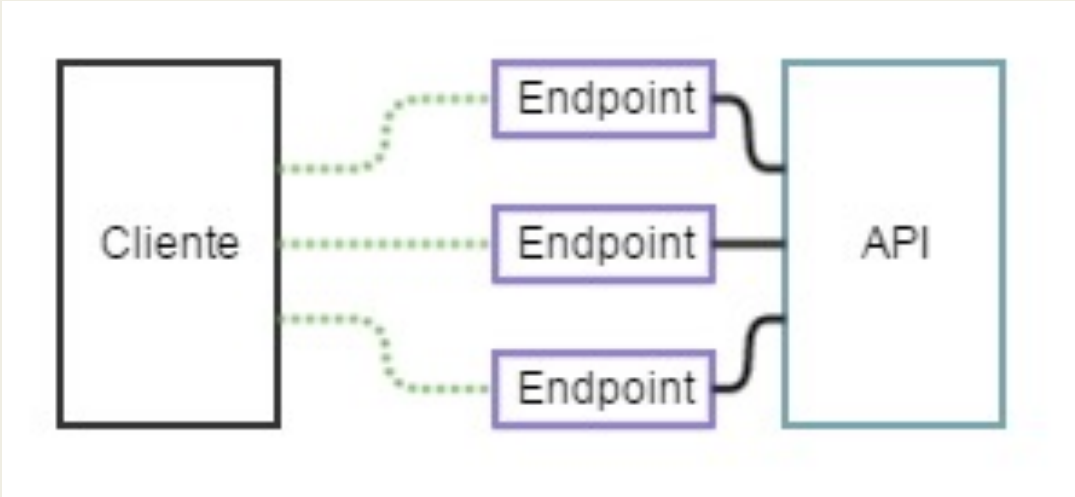
- Cette contrainte n’en est pas vraiment une puisque c’est une option
- La majorité du temps vous allez envoyer une représentation statique des données mais vous avez le droit d’envoyer un executable
- Pour le coup c’est plus une liberté qu’une contrainte

# Des ressources

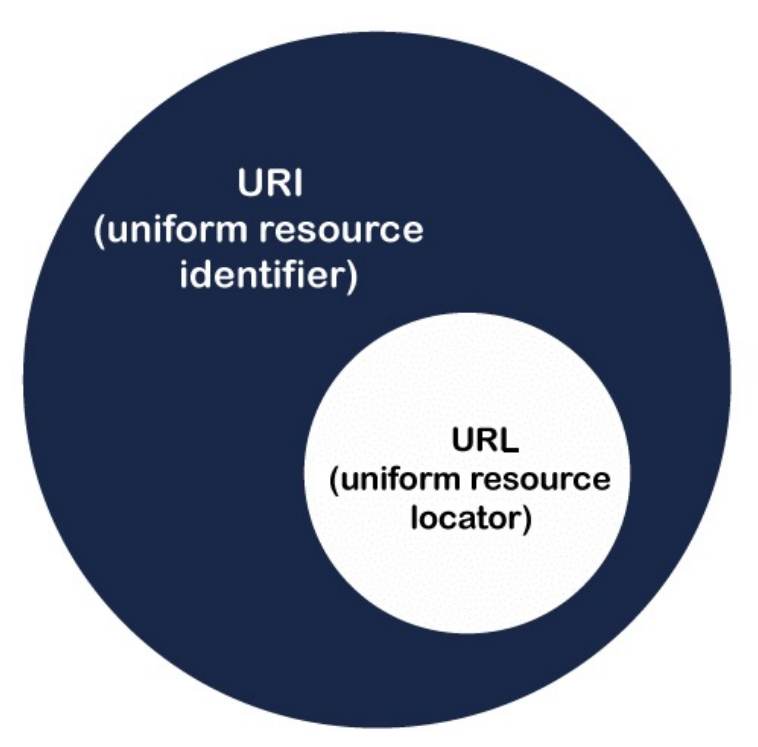
- Un API permet d'échanger des données, il va falloir maintenant trier tout ça
- On va trier en différentes ressources. Par exemple on pourrait avoir dans une boutique en ligne les ressources suivantes :
  - *Article*
  - *Client*
  - *Commande…*
- Chaque ressource va contenir des informations la concernant (nom, prénom, email… pour un client)
- Ces ressources vont ensuite être rangées dans des collections (le nom de la ressource au pluriel)
  - *Articles*
  - *Clients*
  - *Commandes..*

# Endpoint

- Pour accéder à ces ressources, on va devoir les désigner. C'est là que les endpoints entrent en jeu
- Un endpoint est une URL qui fait partie d'une API
- Chaque endpoint permettra d'accéder à une partie de l'API
  - *https://mon-api.com/articles/:id*
  - *https://mon-api.com/clients/:id*



# URI vs URL




- Tant que je suis là, j'en profite pour clarifier un point :
  - *URI : Uniform Resource Identifier*
  - *URL : Uniform Resource Locator*
- Les URL sont un sous ensemble des URI, ce qui veut dire que toutes les URL sont des URI (mais pas l'inverse)
- L'URL permet de localiser une ressource unique là ou l'URI est juste l'identifiant
- Par exemple :
  - *« https://mon-api.com/articles/180 » est une URL*
  - *« /articles/180 » est une URI*

# REQUÊTE ET RÉPONSE

Savoir formuler sa demande et comprendre ce qu’on reçoit

# Que contient la requête

- Une requête HTTP contient 5 éléments :
  - *Le verbe HTTP (GET, POST, PUT, PATCH, DELETE…)*
  - *L'URI*
  - *La version de HTTP utilisée (HTTP/1.1 bien souvent)*
  - *Les headers (l'host, les autorisations…)*
  - *Le body (le corps de la requête, seulement utilisé avec POST, PUT ou PATCH)*

```
GET / HTTP/1.1
Host: developer.mozilla.org
Accept-Language: fr
```

# Précisions sur le verbe HTTP

- GET : C'est la requête « standard », elle permet de récupérer des données. Les paramètres additionnels sont passés en clair dans l'URL
- POST : La requête faite pour… poster des données. Les paramètres sont cette fois passés dans le body. Dans le cas d'un formulaire HTML, par défaut, la forme est « x-www-form-urlencoded »
- PUT / PATCH : Ce sont les deux requêtes faites pour modifier des données, la différence principale est qu'on devra passer l'intégralité des données dans un PUT (même celles qu'on ne modifie pas) là ou dans un PATCH on ne passera que les données modifiées (par défaut, les autres resteront inchangées)
- DELETE : Pour supprimer des données (j'avais pas tellement besoin de l'expliquer)

# Que contient la réponse

- Le format est assez similaire et contient 4 éléments :
  - *La version HTTP (encore la même)*
  - *Le code de réponse HTTP :*
    - 100+ : Information
    - 200+ : Succès
    - 300+ : Redirection
    - 400+ : Erreur coté client
    - 500+ : Erreur coté serveur
  - *Les headers (type de données renvoyées, possibilité de mettre en cache…)*
  - *Le body : le corps de la réponse (à ne pas confondre avec le <body> HTML)*

```
HTTP/1.1 200 OK
Date: Sat, 09 Oct 2010 14:28:02 GMT
Server: Apache
Last-Modified: Tue, 01 Dec 2009 20:18:22 GMT
ETag: "51142bc1-7449-479b075b2891b"
Accept-Ranges: bytes
Content-Length: 29769
Content-Type: text/html

<!DOCTYPE html... (here comes the 29769 bytes of the requested web page)
```

# Réfléchissons

- Avant de nous jeter corps et âme dans le code, il va falloir se poser la question de ce que nous voulons comme fonctionnalités pour notre API
- Quels endpoints ? Pour quoi faire ? Aura-t-on des options de recherche ? De la pagination ?… On va un peu faire de l'UX en fait
- Il va aussi falloir penser à documenter votre API. N'oubliez pas que les API sont des outils faits pas développeurs pour des développeurs, si vous n'aimez pas galérer sur les outils des autres, ne faites pas galérer les autres sur vos outils !

# CODONS

Parce-que elle ne va pas se faire toute seule non plus…

# Notre exemple

- Je vais ici vous montrer un exemple très simple basé sur notre petit CRUD de blog
- Je ne vais faire qu'un endpoint : posts
- Dans ce endpoint je pourrai :
  - *Afficher tous les posts*
  - *Afficher les X derniers posts*
  - *Afficher un post en particulier*
  - *Updater un post*
  - *Updater une partie d'un post*
  - *Supprimer un post*
- Mon API renverra les données au format JSON

# Nos endpoints

- Nous aurons donc à notre disposition les endpoints suivants :
  - *GET /api/posts(?number={number})*
  - *GET /api/posts/:id*
  - *POST /api/posts*
  - *PUT /api/posts/:id*
  - *PATCH /api/posts/:id*
  - *DELETE /api/posts/:id*
- Je vais utiliser un système d'autorisation « Basic » c'est-à-dire avec un login et un password pour accéder aux parties privées
- Je ne vais pas installer de cache coté serveur mais nous pourrions le faire avec REDIS par exemple

# Mettre en place la route

- La première chose à faire va être de modifier mon .htaccess pour me permettre de récupérer la route correctement
- Le flag [QSA] permet de dire que si des variables sont passées en fin d'URL, elles doivent être prises en compte. C'est indispensable pour passer des paramètres additionnels (recherche, tri...)

```
RewriteEngine On

RewriteCond %{REQUEST_FILENAME} !-f
# RewriteCond %{REQUEST_FILENAME} !-d

# REST API
RewriteRule ^api/([a-z0-9-]+)(/?)([0-9]*)$ index.php?p=$1&id=$3 [QSA,NC,L]

# FRONTEND
RewriteRule ^([a-z0-9-]+)(/?)([0-9]*)$ index.php?p=$1&id=$3 [NC,L]
```

# Mettre en place la route

- Il ne me reste plus qu'a déclarer ma route dans mon fichier routes.xml pour que mon routeur puisse la prendre en compte correctement
- Comme mon .htaccess fait le travail, je récupère bien mon paramètre « p » qui spécifie la route (path) et qui pointe vers le bon controller

```
    <!-- REST API -->
    <route p="posts" controller="PostController" action="postsApi" params="id,number"/>
</routes>
```

```php
namespace Vendor\Core;


class HTTPRequest
{
    public function cookieData($key)
    {
        return isset($_COOKIE[$key]) ? $_COOKIE[$key] : null;
    }

    public function cookieExists($key)
    {
        return isset($_COOKIE[$key]);
    }

    public function method()
    {
        return $_SERVER['REQUEST_METHOD'];
    }

    public function requestURI()
    {
        return $_SERVER['REQUEST_URI'];
    }
}
```

# HTTPRequest

- Comme nous l'avons vu, pour avoir une interface uniforme, la même URL aura des comportements différents en fonction du type de requête
- Je vais donc devoir récupérer la « méthode de requête », en PHP, elle est inscrite dans $_SERVER['REQUEST_METHOD']
- Comme je risque de m'en servir souvent, je vais faire une classe « HTTPRequest » avec toutes les méthodes nécessaires et je l'ajouterai au constructeur de mon « BaseController » pour l'avoir à disposition

```php
class HTTPResponse
{
    public function addHeader($header)
    {
        header($header);
    }

    public function redirect($location, int $code = 0, bool $replace = true)
    {
        header('Location: ' . $location, $replace, $code);
        exit;
    }

    public function unauthorized(array $messages): void
    {
        $this->addHeader('WWW-Authenticate: Basic realm="This area needs authentication"');
        $this->addHeader('HTTP/1.0 401 Unauthorized');
        exit(json_encode($messages, JSON_PRETTY_PRINT));
    }

    // Changement par rapport à la fonction setcookie() : le dernier argument est par défaut à true
    public function setCookie($name, $value = '', $expire = 0, $path = null, $domain = null, $secure
    {
        setcookie($name, $value, $expire, $path, $domain, $secure, $httpOnly);
    }

    public function setCacheHeader(int $seconds = 0): void
    {
        $timestamp = time() + $seconds;
        $date = new \DateTime();
        $date->setTimestamp($timestamp);

        $this->addHeader('Cache-Control: public, max-age=' . $seconds);
        $this->addHeader('Expires: ' . $date->format('D, j M Y H:i:s') . ' GMT');
    }
}
```

# HTTPResponse

- Et comme nous sommes dans l'idée de nous faciliter la vie, pourquoi ne pas aussi faire une classe HTTPResponse avec toutes les méthodes utiles pour faire notre réponse

# Quelques pré-requis

- Avant d'analyser la requête et de savoir comment répondre, je vais appeler quelques objets dont je risque d'avoir besoin tout au long de mon controller
- Je vais devoir vérifier l'autorisation et donc potentiellement récupérer un utilisateur. Une autorisation basique va voir ses paramètres passés dans deux variables PHP :
  - *$_SERVER['PHP_AUTH_USER']*
  - *$_SERVER['PHP_AUTH_PW']*
- Je vais aussi devoir récupérer un objet de post
- Un dernier truc, PUT et PATCH ne passent pas de données dans $_POST, je vais donc devoir parser le body et récupérer ces données à la main (et les mettre dans une variable $_PUT pour que ce soit plus clair)

```php
$postManager = new PostManager();
$userManager = new UserManager();

$user = $userManager->checkCredentials($_SERVER['PHP_AUTH_USER'], $_SERVER['PHP_AUTH_PW']);
$postId = !empty($this->params['id']) ? $this->params['id'] : false;
$post = $postManager->postExists($postId) ? $postManager->getPostById($postId) : false;

parse_str(file_get_contents('php://input'), $_PUT);
```

# GET /api/posts

- On va maintenant s'occuper de ce qui se passe quand on fait une requête GET sur ce endpoint
- 3 scénarios sont possibles :
  - *J'interroge juste /api/posts sans rien passer de plus*
  - *J'interroge /api/posts?number={number}*
  - *J'interroge /api/posts/:id pour accéder à un post en particulier*
- Il faut de suite se poser la question suivante :
  - *Que se passe t'il si je passe un paramètre inattendu ?*

# GET /api/posts

- Je vais donc commencer par ouvrir un « if » et tester la méthode HTTP
- Ensuite je regarde si un paramètre « id » est passé (pour savoir si je suis dans le cas /api/posts ou dans le cas /api/posts/:id)
- Si aucun id n'est passé, je renvoie l'index des derniers posts avec comme limite le nombre de posts passé via number
- Je fais d'ailleurs attention à ne pas pouvoir recevoir n'importe quoi de la part de ce paramètre

```php
// GET
if ($this->HTTPRequest->method() === 'GET') :
    switch ($postId) {
        case false:
            $this->HTTPResponse->setCacheHeader(300);
            isset($this->params['number']) ? $number = abs(intval($this->params['number'])) : $number = null;
            return $this->renderJSON($postManager->getPosts($number, true));

        case true:
            $post = $postManager->getPostById($postId, true);
            if (empty($post)) {
                return new ErrorController('noRouteJSON');
            }
            $this->HTTPResponse->setCacheHeader(300);
            return $this->renderJSON($post);
    }
endif;
```

# Quelques précisions

- Vous remarquerez que ma méthode « getPosts » prend un second paramètre (ici true). Cette modification me permet de choisir si je récupère un objet Post ou un array avec les données du post
- En effet, la fonction json_encode() ne comprend pas bien comment encoder un objet, il faut donc lui passer un array

```php
/**
 * @param $content
 */
public function renderJSON($content)
{
    $this->HTTPResponse->addHeader('Content-Type: application/json');
    echo json_encode($content, JSON_PRETTY_PRINT);
}
```

# POST /api/posts

- Dans les règles de notre site, il faut être connecté pour pouvoir poster un article, il n'y a pas de raison que ce soit différent dans notre API
- Je vérifie donc que mon utilisateur est connecté et que des variables $_POST sont bien passées
- Je fais aussi attention à ne pas autoriser la route POST /api/posts/:id qui n'aurait aucun sens

```php
// POST
if ($this->HTTPRequest->method() === 'POST' && !$postId) :

    if ($user && !empty($_POST['title']) && !empty($_POST['content'])) {
        $newPost = new Post(array(
            'title' => $_POST['title'],
            'content' => $_POST['content'],
            'authorId' => $user->getId()
        ));
        $success = $postManager->addPost($newPost, true);

        if ($success) {
            $this->HTTPResponse->setCacheHeader(300);
            return $this->renderJSON($success);
        }
    }
endif;
```

```php
// PATCH (renvoyer que l'élément à modifier)
if ($this->HTTPRequest->method() === 'PATCH' && $postId && $post) :

    if ($user && (!empty($_PUT['title']) || !empty($_PUT['content'])) && $user->havePostRights($post)) {
        $postTitle = empty($_PUT['title']) ? $post->getTitle() : $_PUT['title'];
        $postContent = empty($_PUT['content']) ? $post->getContent() : $_PUT['content'];

        $post->setTitle($postTitle);
        $post->setContent($postContent);
        $success = $postManager->updatePost($post, true);

        if ($success) {
            $this->HTTPResponse->setCacheHeader(300);
            return $this->renderJSON($success);
        }
    }
endif;
```

# PUT et PATCH

- Ces deux la sont assez semblables et il n'est pas obligatoire de créer les deux, je l'ai fait pour l'exercice mais c'est un peu overkill
- Pour modifier un article il faut soit en être l'auteur, soit être admin
- Je vérifie aussi que le l'article que l'on cherche à modifier existe et que des informations ont bien été passées
- Rappelez-vous, $_PUT est une variable crée de toute pièce en parsant les informations du body de ma requête

# DELETE /api/posts/:id

- On termine par quelque chose de plus simple
- Il suffit de vérifier là aussi que l'utilisateur à les droits sur l'article et que l'article existe bien

```php
// DELETE
if ($this->HTTPRequest->method() === 'DELETE' && $postId && $post && $user && $user->havePostRights($post)) :
    $success = $postManager->deletePost($postId);

    if ($success) {
        return $this->renderJSON([
            "status" => 1,
            "message" => 'Post deleted'
        ]);
    }
endif;
```

# En cas d’erreur

- Il pourrait être utile de faire une gestion d’erreurs plus détaillée mais il faut, à minima, donner un message d’erreur avec quelques informations sur comment la corriger

```
// If something goes wrong :
$this->HTTPResponse->unauthorized([
    'Authentication' => "Basic",
    "Needed arguments" => ['title', 'content']
]);
```

# Exercice

- Pour aller plus loin je vous propose de faire un endpoint sur notre API pour récupérer les commentaires des posts
- Le endpoint devra être de la forme /api/posts/:id/comments/:id
- Mais aussi supporter les appels de la forme /api/comments/:id quand c’est pertinent

# CORRECTION

On a des choses à dire ici

# .htaccess

- Encore une fois, il va devoir se modifier légèrement pour accepter les URI de notre API… la REGEX commence à être longue
- Subtilité cependant, je fais en sorte que $_GET[‘id’] soit l’id du commentaire et non celui du post, c’est ce qui donne la résilience pour exploiter aussi les routes du type /api/comments/:id

src > .htaccess

```
1   RewriteEngine On
2
3   RewriteCond %{REQUEST_FILENAME} !-f
4   # RewriteCond %{REQUEST_FILENAME} !-d
5
6   # REST API
7   RewriteRule ^api/posts/([0-9]+)/([a-z0-9-]+)(/?)([0-9]*)$ index.php?p=$2&id=$4&mainId=$1 [QSA,NC,L]
8   RewriteRule ^api/([a-z0-9-]+)(/?)([0-9]*)(/?)$ index.php?p=$1&id=$3 [QSA,NC,L]
9
10  # FRONTEND
11  RewriteRule ^([a-z0-9-]+)(/?)([0-9]*)$ index.php?p=$1&id=$3 [NC,L]
```

# Les routes

- Ici, rien de vraiment méchant, il suffit de rajouter la ligne et les paramètres correspondants

```
    <!-- REST API -->
    <route p="posts" controller="PostController" action="postsApi" params="id,number"/>
    <route p="comments" controller="CommentController" action="commentsApi" params="id,mainId"/>
</routes>
```

# Les pré-requis

- On va pouvoir reprendre à peu près la même base que pour les posts mais en rajoutant deux ou trois choses

```php
$postManager = new PostManager();
$userManager = new UserManager();
$commentManager = new CommentManager();

$user = $userManager->checkCredentials($_SERVER['PHP_AUTH_USER'], $_SERVER['PHP_AUTH_PW']);
$postId = !empty($this->params['mainId']) ? $this->params['mainId'] : false;
$post = $postManager->postExists($postId) ? $postManager->getPostById($postId) : false;
$commentId = !empty($this->params['id']) ? $this->params['id'] : false;
$comment = $commentManager->commentExists($commentId) ? $commentManager->getCommentById($commentId) : false;

parse_str(file_get_contents('php://input'), $_PUT);
```

# GET

- Cette fois ci, il va être plus long puisque il y a 4 situations différentes :
  - *Je veux la liste de tous les commentaires via la route /api/comments*
  - *Je veux les commentaires d'un post : /api/posts/{postId}/comments*
  - *Je veux un commentaire en particulier, j'ai deux routes possibles :*
    - /api/comments/:id
    - /api/posts/{randomId}/comments/:id
- Dans le dernier cas, l'id du post n'est même pas vérifié ou pris en compte mais ca aurait été dommage de ne pas avoir cette route qui semble logique

# GET

Du coup, ça donne un truc comme ça

```php
// GET
if ($this->HTTPRequest->method() === 'GET') :
    if ($post) {
        switch ($commentId) {
            case false:
                $this->HTTPResponse->setCacheHeader(300);
                return $this->renderJSON($commentManager->getCommentsByPostId($postId,

            case true:
                $comment = $commentManager->getCommentById($commentId, true);
                if (empty($comment)) {
                    return new ErrorController('noRouteJSON');
                }
                $this->HTTPResponse->setCacheHeader(300);
                return $this->renderJSON($comment);
        }
    }
    switch ($commentId) {
        case false:
            $this->HTTPResponse->setCacheHeader(300);
            return $this->renderJSON($commentManager->getAllComments(true));

        case true:
            $comment = $commentManager->getCommentById($commentId, true);
            if (empty($comment)) {
                return new ErrorController('noRouteJSON');
            }
            $this->HTTPResponse->setCacheHeader(300);
            return $this->renderJSON($comment);
    }
endif;
```

# POST

- Lui ne sera pas accessible via /api/comments puisque un commentaire à forcément un post qui lui est rattaché
- Je vais donc vérifier que j'ai bien un post valide et que l'utilisateur est connecté

```php
// POST
if ($this->HTTPRequest->method() === 'POST' && !$commentId && $post) :

    if ($user && !empty($_POST['content'])) {
        $newComment = new Comment(array(
            'postId' => $postId,
            'content' => $_POST['content'],
            'authorId' => $user->getId()
        ));
        $success = $commentManager->addComment($newComment, true);

        if ($success) {
            $this->HTTPResponse->setCacheHeader(300);
            return $this->renderJSON($success);
        }
    }
endif;
```

# PUT / PATCH

- Autant j'avais fait les deux pour les posts car ils avaient deux comportement différents mais comme un commentaire n'a qu'un seul champs modifiable, je suis obligé de renvoyer l'entité complète à chaque fois.
- PUT et PATCH deviennent donc identiques
- Je ne vérifie pas l'existence du post puisque j'ai déjà un id de commentaire et donc, un post attaché
- Je peux y accéder depuis les deux endpoints

```
// Comme je n'ai qu'un élément dans le commentaire, PUT et PATCH deviennent identiques
if (($this->HTTPRequest->method() === 'PUT' || $this->HTTPRequest->method() === 'PATCH') && $comment) :

    if ($user && !empty($_PUT['content']) && $user->haveCommentRights($comment)) {
        $comment->setContent($_PUT['content']);
        $success = $commentManager->updateComment($comment, true);

        if ($success) {
            $this->HTTPResponse->setCacheHeader(300);
            return $this->renderJSON($success);
        }
    }
endif;
```

# DELETE

- Lui aussi est accessible depuis les deux endpoints
- Je vérifie juste l'intégrité de l'id du commentaire et que l'user à bien les droits

```php
// DELETE
if ($this->HTTPRequest->method() === 'DELETE' && $comment && $user && $user->haveCommentRights($comment)) :
    $success = $commentManager->deleteCommentById($commentId);

    if ($success) {
        return $this->renderJSON([
            "status" => 1,
            "message" => 'Post deleted'
        ]);
    }
endif;
```