# TP 15 : Client JS avec API PHP

## Objectifs du TP:

L'objectif de ce TP est de comprendre la structure d'une application Web avec une page dynamique en JS qui accède à une API coté serveur.

## 1. Accès à une API depuis le client

Une Interface de programmation (API pour "Application Programming Interface") est un ensemble de fonctionalités qu'un logiciel met à disposition d'un autre logiciel. Dans le cas du WEB, il s'agit d'un service coté serveur, qui est mis à disposition de l'interface coté client.

Typiquement, une API Web ne produit pas d'HTML, mais elle produit un autre format de données qui est exploité par le client. Cela peut être de l'XML, du JSON, ou tout autre format, y compris un format binaire, comme de l'audio, une image, une vidéo. Cela peut être aussi tout simplement du texte.

### 1.1 Création de l'API (coté serveur)

Une API au format REST utilise le protocole HTTP pour activer du code d'un serveur WEB. L'activation se fait de manière identique à d'habitude, c'est à dire avec une demande *GET* ou *POST* et des données dans la query string pour *GET*, ou dans le corps du message HTTP pour *POST*. Ce qui change est simplement le type de données produit par l'API dans sa sortie standard (ex: par un *print* en PHP). Par défaut, l'interpréteur PHP du serveur Apache, produit une entête de message avec comme type HTML.

```
Host: VotreSiteWeb
Date: Mon, 28 Nov 2024 09:56:58 GMT
Connection: close
X-Powered-By: PHP/8.1.11
Content-type: text/html; charset=UTF-8
```

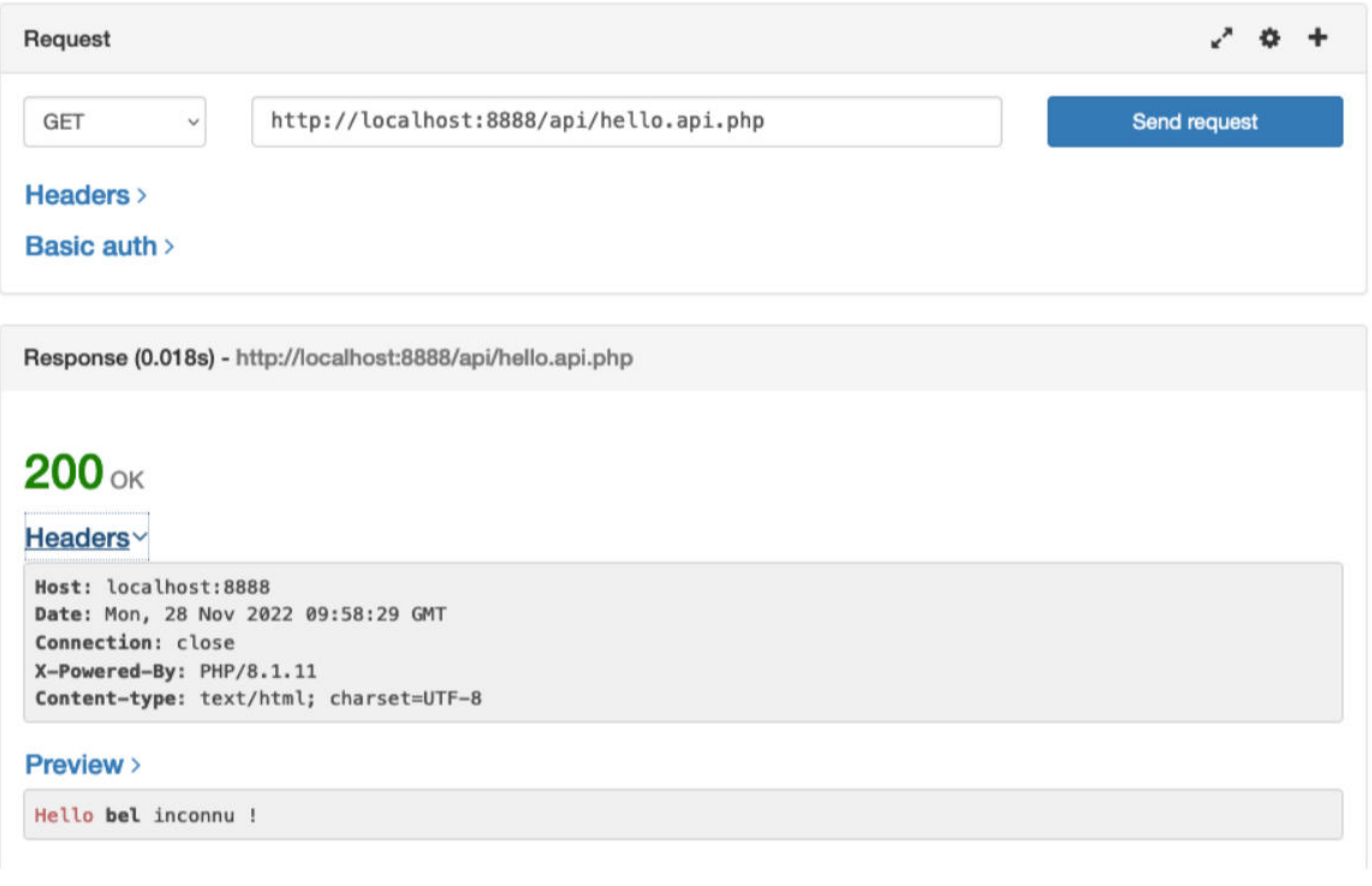
Il faut indiquer de manière explicite de type de donnée que l'API produit, avec la syntaxe des Types de média qui dérive du type MIME.

**A faire :**

- Télécharger et installer l'application incomplète dans le fichier compressé : data/api.zip
- Installez un outil de test de client REST dans votre navigateur (ex: Firefox) Pour Firefox, nous vous proposons le client RESTED. Pour l'installer, allez dans le menu (à droite) "Add-ons and themes" puis (à gauche) "Extensions", recherchez "RESTED" et l'installer.
- Complètez le fichier `api/hello.api.php` pour réaliser les fonctionalités suivantes :
  - Recupérer dans la query string (*$_GET*) la valeur du paramètre *nom* . Donner une valeur par défaut s'il n'est pas présent.
  - Produire en sortie standard un message de bienvenu. Par exemple :

```
print("Hello  $nom !");
```

- Activez l'outil RESTED sur votre URL. Vous obtenez par exemple :



```
http://localhost:8888/api/hello.api.php
```

Response (0.018s) - http://localhost:8888/api/hello.api.php

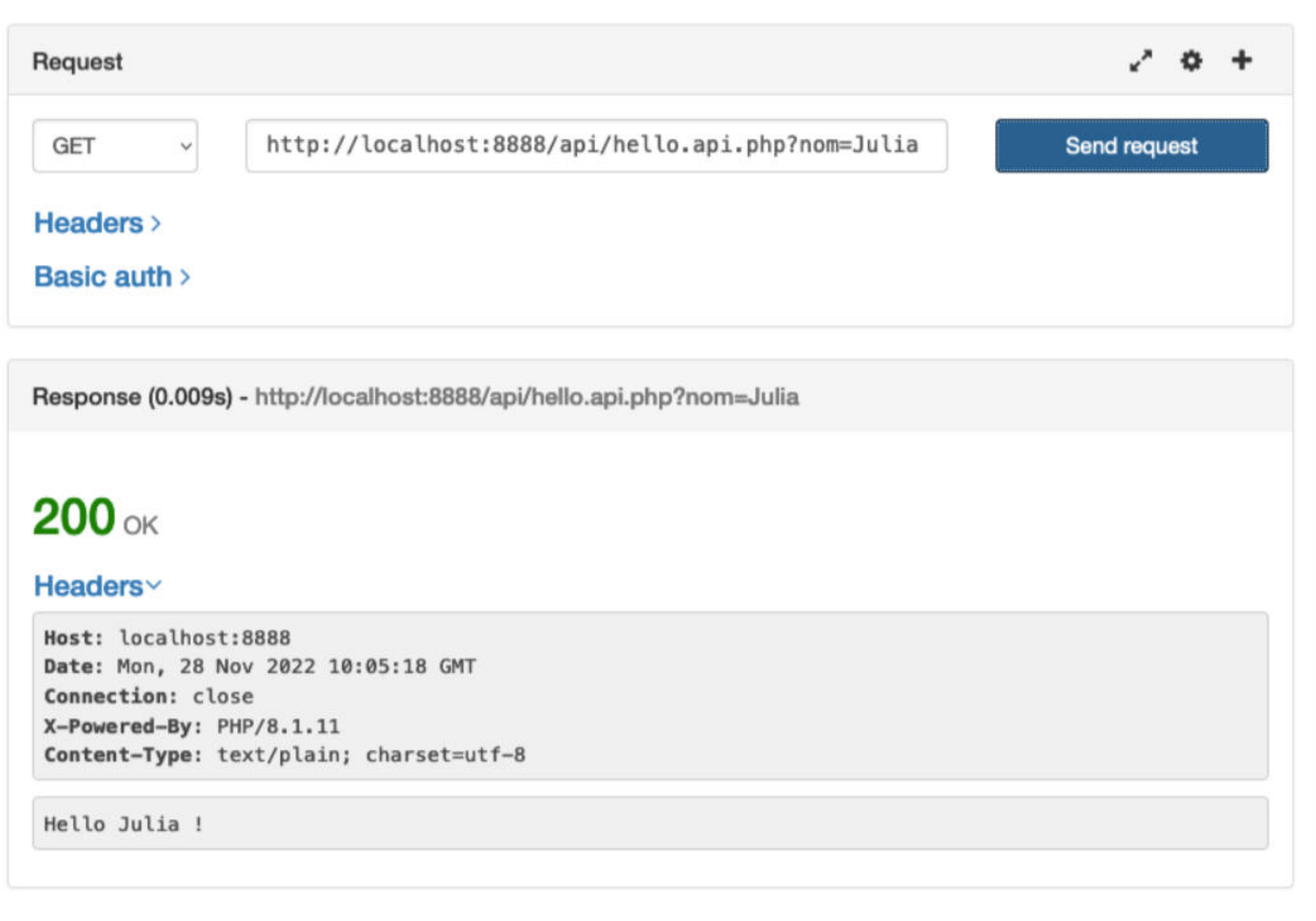
200 OK

```
Host: localhost:8888
Date: Mon, 28 Nov 2022 09:58:29 GMT
Connection: close
X-Powered-By: PHP/8.1.11
Content-type: text/html; charset=UTF-8
```

```
Hello bel inconnu !
```

On voit que le type du format de sortie n'est pas correct.

- Dans le fichier `hello.api.php` changez le header pour indiquer le format *text/plain* ainsi que l'usage de l'encodage utf-8 : *charset=utf-8*. Vous devriez maintenant voir s'afficher :



```
http://localhost:8888/api/hello.api.php?nom=Julia
```

Response (0.009s) - http://localhost:8888/api/hello.api.php?nom=Julia

200 OK

```
Host: localhost:8888
Date: Mon, 28 Nov 2022 10:05:18 GMT
Connection: close
X-Powered-By: PHP/8.1.11
Content-Type: text/plain; charset=utf-8
```

```
Hello Julia !
```

### 1.2 Utilisation de l'API (coté client) : le modèle

Notre API est prête à être utilisée coté client en Javascript. Il s'agit de décrire les éléments DOM de la vue qui sont actifs dans cette application, puis indiquer l'action à réaliser quand on clique sur le bouton. Nous allons décrire cela avec un modèle MVC.

L'échange avec l'API est du coté du modèle. C'est lui qui est chargé de demander le service de l'API pour réaliser un calcul. Dans non exemple, le calcul consute juste à prendre le nom en paramètre et retourner une salutation en mode texte.

```
////////////// Partie Modèle ////////////////////
import dao from "./dao.js";

const model = {
    saluer: function (nom,callback) {
        // Crée une query string
        const queryString = new URLSearchParams();
        // Ajoute le parametre nom
        queryString.append('nom', nom);
        // construit l'URL de l'API
        const URL = "public/api/hello.api.php?" + queryString;

        // Lance la demande
        fetch(URL).then(r => {
            if (r.ok && r.status == 200) {
                return r.text();
            } else {
                throw new Error('Erreur serveur', {cause: r});
            }
        }).then(text => callback(text));
    }
};

export default model;
```

Examinons le code du modèle : c'est une objet qui contient une seule méthode, la méthode ***saluer(nom,callback)***. Le premier paramètre ***nom*** est le nom de la personne à saluer. Comme le modèle doit interroger une API, il y a un temps d'attente : le temps que l'API réponde à la demande. Il n'est pas raisonnable de bloquer l'exécution de JS (c'est à dire figer l'interface) dans l'attente de la réponse du serveur. Pour cette raison, le paramètre ***callback*** contient une fonction qui sera appelée lorsque la réponse du serveur sera disponible. Notez-bien qu'il n'y a pas de parallélisme en JS : l'interpréteur exécute toujours au maximum qu'une seule instruction. Le paradigme de programation avec une fonction callback permet ainsi de réaliser de la programmation asynchrone avec un langage sans parallélisme.

Les trois premières instructions servent à produire une URL avec une partie query string correcte. Ensuite, la fonction fetch(URL) envoye l'URL sur le réseau et n'attend pas la réponse du serveur. En retour, ***fetch*** produit un objet très particulier : une promesse d'exécution, un objet de type promise. C'est cet objet promise qui se bloque sur l'attende de la réponse du serveur.

Quand la promesse se débloque, dans notre cas, quand elle a obtenue la réponse du serveur, alors elle déclanche sa méthode then qui doit avoir en paramètre une fonction callback. Dans le cas de fetch, un objet de type Response est passé à la callback (appelé *r* dans le code). Avec ce paramètre, on peut alors tester soit l'attribut ok qui vaut true si le code HTTP de la réponse est entre 200 et 299, ou bien tester directement le code de la réponse HTTP avec status. Dans le code on fait les deux. Voir la signification de ces codes sur Liste des codes HTTP(Wikipedia)

Une méthode ***then*** doit retourner une promesse ou rien du tout. La méthode text() de l'objet Response permet d'aller chercher le contenu du message sous la forme d'une simple chaine de caractère. Cette méthode ne produit pas directement le contenu du message car le contenu d'un message HTTP peut prendre du temps à arriver, mais produit une nouvelle promesse. C'est pour cette raison qu'on enchaine sur un autre appel à ***then*** pour finalement récupérer le texte du message. A ce moment là, on peut enfin déclancher la callback que l'on a passé en paramètre de *saluer()*.

### 1.3 Utilisation de l'API (coté client) : la vue

La vue est assez simple. Elle place un lien vers les 3 objets DOM qui nous intéressent : le bouton, l'entrée (input) et la balise de sortie (output). La méthode `read` retourne la valeur (value) de l'entrée du formulaire. La méthode `update` met à jour l'affichage en placant la valeur dans la balise de sortie (textContent). La méthode `addEventListener` place une fonction callback pour réagir au clic du bouton.

**A faire**

Dans le fichier `view.js` :

- Complétez le vue avec les constantes *input* et *output* qui doivent pointer sur un seul objet DOM de l'interface.
- Complétez la méthode *read()* qui retourne le texte de la balise *input* de la vue.
- Complétez la méthode *update(out)* qui place dans le texte de la balise *output* de la vue par textContent, la chaîne passée en paramétre.

### 1.4 Utilisation de l'API (coté client) : le contrôleur

Le controleur est chargé d'accrocher une callback de réaction au bouton de la vue (dernière ligne). Cette callback `onSaluer` doit lire le contenu du formulaire, et le passer au modèle. Comme le modèle a besoin d'une fonction callback pour réagir de manière non synchrone, il faut lui passer la fonction `onAnswser`. Cette fonction doit simplement mettre à jour la vue avec le texte passé en paramètre par le modèle.

**A faire**

Dans le fichier `controler.js` :

- Compléter la fonction callback `onSaluer`.
- Compléter la fonction callback `onAnswser`.

## 2. Le temps dans le monde

L'objectif de cet exemple est de montrer comment structurer une application Web avec une partie Javascript et une partie PHP. La partie PHP a un modèle très simple : une classe qui va donner l'heure dans un fuseau horaire donné.

**A faire :**

- Récupérez le code incomplet de ce projet dans : data/time.zip

### 2.1 Le modèle en PHP

Il s'agit de la classe `Time` dans le répertoire `model`. Cette classe représente la notion de temps. Elle possède 2 attributs :

- dateTime : l'objet PHP du type DateTime
- timeZone : une chaîne qui décrit une Time Zone. Par exemple, nous sommes dans la Time Zone *Europe/Paris* .

La classe PHP Time possède aussi deux méthodes de transformation :

- `dateToString` : transforme l'objet en une chaîne de caractères que l'on peu afficher dans l'interface HTML.
- `jsonSerialize` : sérialise cet objet en une chaine de caractère au format JSON. C'est utilisé par le code de l'API qui envoye cette information au client en JS. Le format JSON est pratique car il est reconnu par JS et re-transformé en objet JS qui est ensuite utilisable dans le code.

**A faire :**

- Complétez le constructeur. Il faut créer un nouvel objet DateTime puis utiliser sur lui la méthode setTimezone en passant en paramètre un objet de type DateTimeZone créé à partir de la time zone (une chaîne) passée en paramètre.
- La classe doit suivre l'interface JsonSerializable car nous allons passer cet objet au client, et nous décidons de le sérialiser dans le format JSON. C'est un format de données de type texte très répandu et dont les libraries de transformation sont disponibles en PHP et JS. Remarquez que dans la sérialisation, nous ne transformons pas l'attribut `dateTime`. A la place nous créons l'attribut `date` qui est une chaîne de caractères correspondant à la date et l'heure formatée.
- Testez le bon fonctionnement de cette classe (`time.test.php`).

### 2.2 L'API coté PHP

Une Application Programming Interface (API) est une interface logicielle qui expose des services (fonctions et prodécure) et permet à un autre programme d'utiliser ses services à travers une interface normalisée. L'interface que nous utilisons est le protocole HTTP. L'API est écrite en PHP et donne acces à travers le WEB au modèle coté PHP. Ce type d'API se nomme une API de type REST.

Le fonctionnement est le suivant. Il faut activer le code de l'API `time.api.php` en mode GET, à l'aide d'une URL qui possède une query string avec les deux noms suivants :

- action : c'est le type d'action que l'on demande à l'API. Dans notre cas, une seule action est possible : *read*. Dans le cas d'une API qui accède à une base de données, on peut implémenter toutes les actions CRUD. Dans notre application, cette action demande la lecture de la date coté serveur, avec le time zone indiqué en paramètre.
- timeZone : c'est le paramètre de l'action *read*.

Cette API doit produire une chaîne de caractère au format JSON. Notez qu'à la fin du script, nous avons produit une partie du header HTTP qui indique que l'on ne va pas produire du HTML, comme d'habitude mais du JSON.

```
header('Content-Type: application/json; charset=utf-8');
```

Egalement à la fin, nous produisont la chaîne JSON en transformant le tableau `$out` avec la fonction json_encode. La chaîne est produite directement dans la sortie standard, comme on le fait quand on produit du HTML.

La sortie JSON doit indiquer les erreurs suivantes :

- {"error": "incorrect action"} : si l'action est absente ou inconnue, par exemple si l'URL est *http://siteWeb/api/time.api.php*
- {"error": "timeZone missing for read"} : si l'action est correcte, mais si le nom timeZone n'est pas dans l'URL. Par exemple : *http://siteWeb/api/time.api.php?action=read*
- Récuperez toutes les autres erreurs possibles qui peuvent provoquer une exception et indiquez le message d'erreur dans la sortie en JSON.

S'il n'y a pas d'erreur, l'API doit fournir le résultat de la requête au format suivant :

```
{
"time":
    {
    "timeZone": "Europe/Paris",
    "date": "03/12/2024 15:14:05"
    }
}
```

Cela correspond à la requête : *http://siteWeb/api/time.api.php?action=read&timeZone=Europe/Paris*

**A faire :**

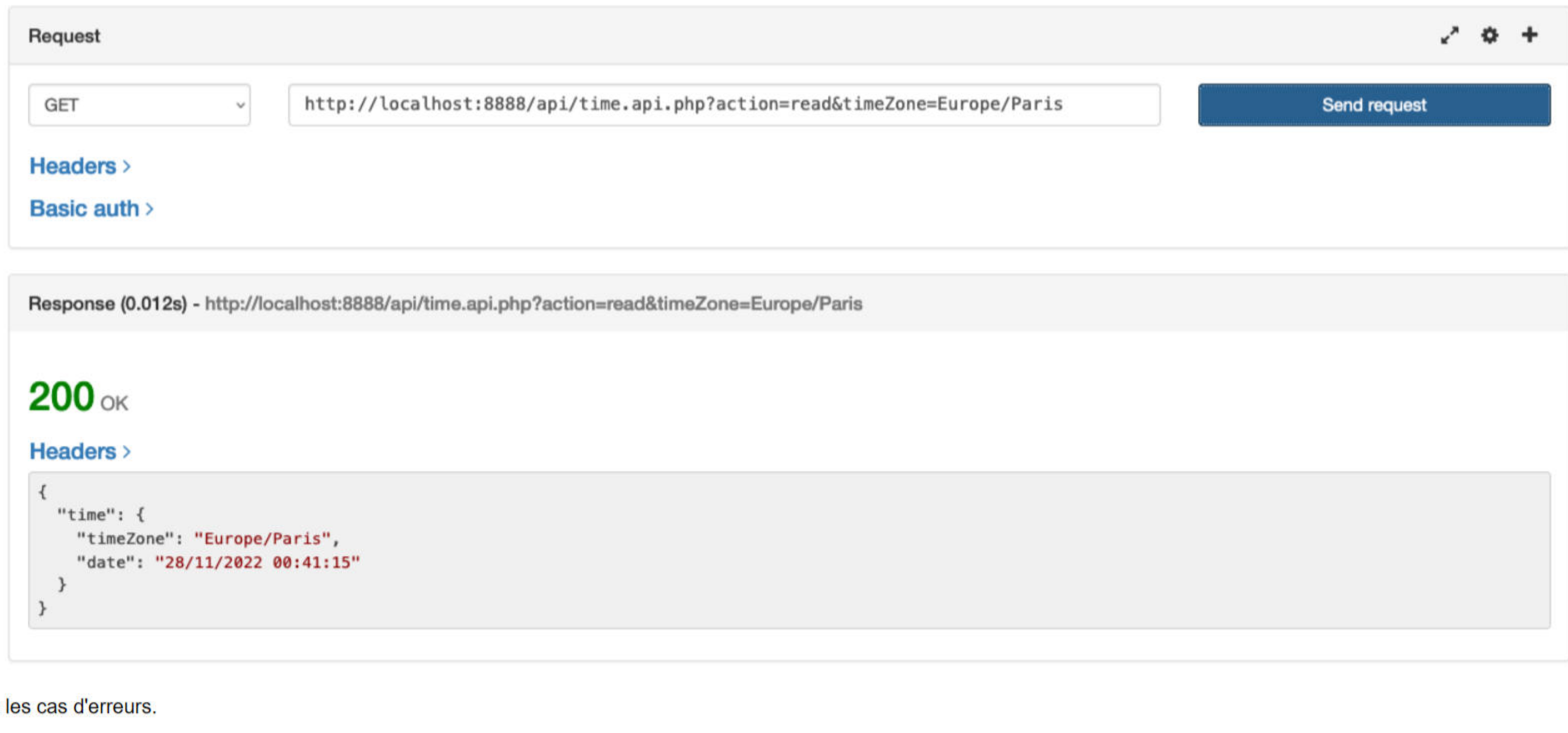
- Ecrire le code de l'API pour qu'elle produise les erreurs prévues, et qu'elle réponde à la requête : créer un objet *Time*, et le placer dans le tableau de sortie *out* à l'index *time*. C'est la fonction de sérialisation qui transformera cet objet en JSON tel que prévu par le modèle (fonction *jsonSerialize*).
- En cas d'erreur, changer le header pour qu'il produise un code d'erreur et un message au début de la réponse (*HTTP/1.1*)
- Testez le fonctionnement de cet API avec *api.test.php*.

### 2.3 L'api REST vue coté client

Le client REST sert à recevoir les requêtes de l'interface et à la traduire pour acceder aux données. Il n'affiche pas de HTML, mais produit une sortie dans un format convenu à l'avance dans le protocole de communication. Nous avons choisit le format JSON dans cet exercice. Dans cette partie, nous proposons d'utiliser une extension au navigateur Firefox pour tester séparément l'API.

**A faire :**

- Installez un outils de test de client REST. Nous vous proposons le client RESTED. Pour l'installer, aller dans le menu (à droite) "Add-ons and themes" puis (à gauche) "Extensions", recherchez "RESTED" et l'installer.
- Testez le fonctionnement normal.



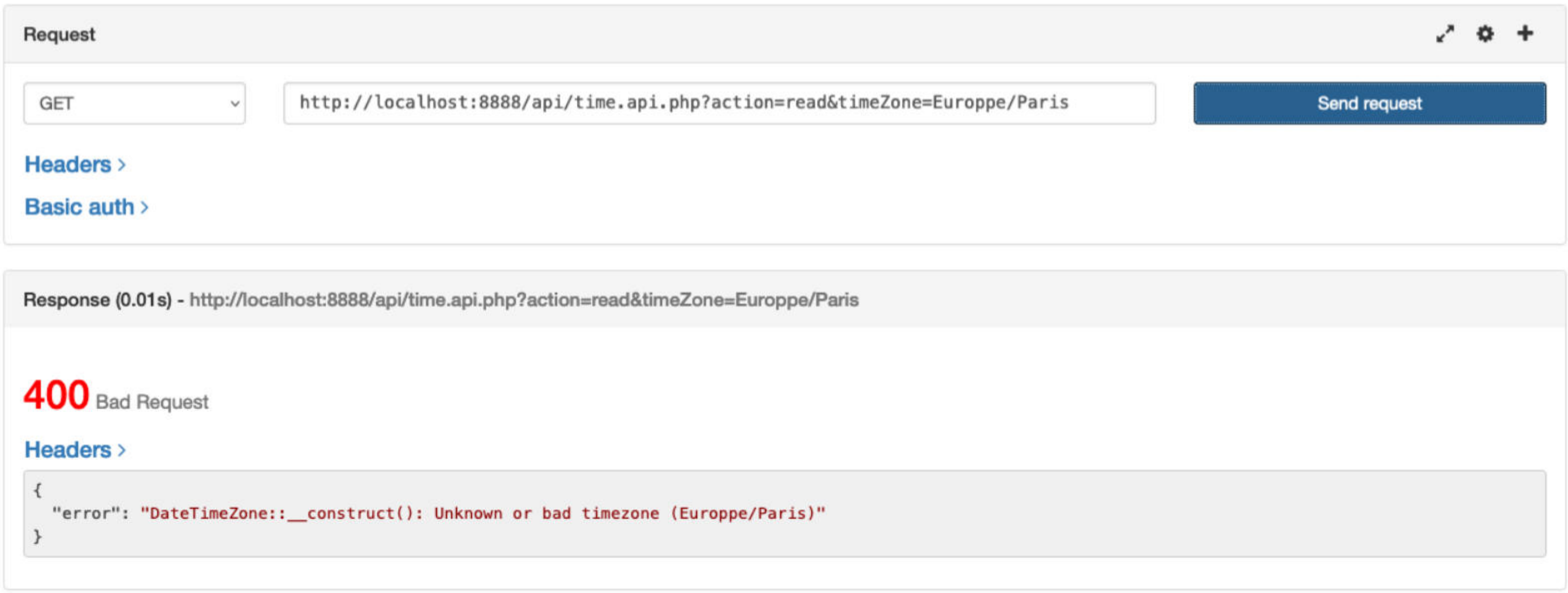
```
http://localhost:8888/api/time.api.php?action=read&timeZone=Europe/Paris
```

Response (0.012s) - http://localhost:8888/api/time.api.php?action=read&timeZone=Europe/Paris

200 OK

```
{
  "time": {
    "timeZone": "Europe/Paris",
    "date": "28/11/2022 00:41:15"
  }
}
```

- Testez les cas d'erreurs.



```
http://localhost:8888/api/time.api.php?action=read&timeZone=Europpe/Paris
```

Response (0.01s) - http://localhost:8888/api/time.api.php?action=read&timeZone=Europpe/Paris

400 Bad Request

```
{
  "error": "DateTimeZone::__construct(): Unknown or bad timezone (Europpe/Paris)"
}
```

```
http://localhost:8888/api/time.api.php?action=read
```

Response (0.008s) - http://localhost:8888/api/time.api.php?action=read

400 Bad Request

```
{
  "error": "timeZone missing for read"
}
```

### 2.4 Coté client en JS : le DAO et le modèle

Coté client, en JS, la communication avec l'API est matérialisée par un objet dao.

**A faire**

Examinez le code `dao.js`

- Il s'agit d'un objet unique avec une seule méthode.
- Le premier paramètre *params* est un objet JS qui liste les paramètres à passer dans l'URL. Par exemple :

```
const params = {
    'action': 'read',
    'timeZone': timeZone
    }
```

  La query string est alors construite avec un objet de type *URLSearchParams*.
- La communication se fait avec la fonction *fetch(URL)*. La particularité de cette communication c'est qu'elle est asynchrone. Cela veux dire qu'on ne sait pas quand le communication va se faire et quand on aura le résultat. La fonction *fetch(URL)* lance la requête URL sur le serveur puis se termine en produisant une promesse (objet de type *promise*).
- On donne à cet objet promesse (dans le code de *then(...)*) la fonction à déclancher quand le résultat du serveur sera arrivé. L'exécution de cette méthode *then(...)* produit une autre promesse par d'exécution de la méthode *json()* qui extrait la chaîne au format JSON de la réponse du serveur.
- Une fois le json décodé, alors la fonction dans la méthode *then(...)* lance la callback *onAnswer(obj)* donnée en paramètre. La promesse produite par la méthode *json()* produit directement un objet *obj* qui est passé à sa fonction dans son *then(...)*.

Coté client, il faut représenter à nouveau le modèle, avec les méthodes CRUD comme coté serveur en PHP. Nous n'avons implémenté que le *read*. Cette méthode est une méthode de classe (comme avec le PHP). Son rôle est de déclancher la demande de lecture (read) sur le dao, et de lui passer une fonction callback qui doit créer un objet Time avec l'objet envoyée par le dao, puis à son tour déclancher la callback *onAnswser* passée en paramètre.

**A faire :**

- Complétez la fonction *read* de la classe *Time*