Introduction à REST

Ajax et API REST avec JQuery et fetch/then

Exercice 1. Météo Ajax

Réalisons tout d'abord un petit service Météo utilisant Ajax. La partie serveur en PHP installe un Web Service en JSON :

```php
<?php
  header("Content-Type: text/plain;charset='utf-8'");
   if (isset($_REQUEST['cp']) && !empty($_REQUEST['cp'])){
     $cp = $_REQUEST['cp'];
     setLocale(LC_TIME,"fr_FR");
     date_default_timezone_set("Europe/Paris");
     $today = strftime('%A %d %B %y',time());
     $hour = date('H:i:s');
     $meteo = array("cp"=>$cp,
                    "jour"=>$today,
                    "heure"=> $hour,
                    "meteo"=>"Il va faire très beau !" );
     if ($cp=='45000') echo json_encode($meteo,JSON_PRETTY_PRINT);
     elseif ($cp=='13000') echo json_encode($meteo,JSON_PRETTY_PRINT
        );
     elseif ($cp=='06000') echo json_encode($meteo,JSON_PRETTY_PRINT
        );
     else {
       $meteo['meteo'] = "Inconnu";
       echo json_encode($meteo,JSON_PRETTY_PRINT);
   }
  }
```

C'est à dire qu'on attend la transmission d'un code postal sur l'URL associé à la variable cp et on fabrique json avec la date et la météo du jour.

Il nous reste maintenant à coder le client JS qui contactera ce service PHP grâce à la méthode `ajax()` de jQuery. Placez le code JS suivant dans un document html :

```javascript
$(function() {
    // Définition de l'évènement sur le changement de valeur dans le
       champ de saisie pour le code postal
       $("#code_postal").keyup(function(event) {
             // Lecture du code postal
             const cp = $(this).val();
             // On teste que le code postal est complet
             if (cp.length == 5) {
```

```
8               // Exécution de la requete Ajax appelant notre sevice mét
                   éo
9               $.ajax({
10                // Définition de l'URL à appeler
11                url: "http://localhost/~roza/meteo2/meteoAjaxJSON.php",
12                // Méthode HTTP
13                type: "GET",
14                // Définition des paramètres transmis sur la chaine de
                     requete (ici cp) sous la forme param_X=val_X
15                data : "cp=" + cp,
16                // Définition du type de données en retour(ici, type
                     JSON)
17                dataType : "json",
18                // Définition du traitement à effectuer en cas de
                     succes
19                success: function(msg) {
20                // Mise à jour du bloc pour la météo
21                // Récupération et affichage des données
22                   console.log(msg);
23                   if (Object.entries(msg).length === 0 \tabularnewline
24                   && msg.constructor === Object){
25                                $("#meteo").html("<b> CP inconnu !</b>");
26                         }
27                   else{
28                         const jour = msg['jour'];
29                         const heure = msg['heure'];
30                         const meteo = msg['meteo'];
31                         // Formatage du résultat avant de l'afficher
32                         $("#meteo").html("<b>Le " + jour + " à " +
                              heure + "</b> : " + meteo + " (" + cp + ")")
                              ;
33                       }
34                },
35                // Définition du traitement à effectuer en cas d'échec
36                error: function(req, status, err) {
37                         $("#meteo").html(`<b>Impossible de contacter
                              le service météorologique ${status} !</b>`)
                              ;
38                       }
39                    });
40                  }
41              });
42          });
```

Remplacez maintenant l'appel ajax en jQuery par un appel ajax effectué avec les méthodes `fetch/then`.

Exercice 2. Listes déroulantes chainées

Il arrive qu'on doive présenter plusieurs listes déroulantes en cascade dont la 2ème dépend de la première par exemple. Ici encore, c'est Ajax qui nous donnera la bonne solution. Commençons par entrer ces nouvelles tables dans notre base de données en exécutant le script *dbfilms.sql* fourni. Puis préparons un fichier php qui sera appelé par le script qui présentera les listes déroulantes. Ce script attend une année passée par la méthode POST et renvoie un fragment HTML correspondant au ¡select¿ avec ses différentes options de films de cette année là.

```php
<?php
// ajax2.php
include("connexion.php");
$annee=$_POST['annee'];
if (!empty($annee)){
  $connexion = connect_bd();
  $sql="SELECT * FROM FILM WHERE annee=:annee ORDER BY titre";
    $stmt=$connexion->prepare($sql);
    $stmt->bindParam(':annee', $annee, PDO::PARAM_INT);
    $stmt->execute();
  echo "Film : <select name='film'>";
  while ($row= $stmt->fetch(PDO::FETCH_OBJ))
    echo "<option value='".$row->id_film."'>".$row->titre."</option>
       ";
  echo "</select>";
  }
```

Puis nous allons écrire le script ajaxjQuery.php qui va présenter les 2 listes déroulantes en cascade.

```php
<!doctype html>
<html>
  <head>
    <meta charset="utf-8"/>
  </head>
  <body>
    <form action="#">
    <?php
    include("connexion.php");
    $connexion = connect_bd();
    if (empty($connexion)){
      echo "Problème de connexion";
    }
    else{
      $films=Array();
      $sql="SELECT * FROM FILM ORDER BY annee";
      echo 'Année : <select name="annee" size="1" onChange="ajax()">
         ';
      foreach ($connexion->query($sql) as $row)
        echo "<option value='".$row["annee"]."'>".$row["annee"]."</
           option>";
      echo "</select>";
    }
```

```
22 ?>
23 <div id="affiche"></div>
24 </form>
25 </body>
26 </html>
```

La fonction JS appelée *ajax()* peut être écrite de diverses manières. Voici une façon de l'écrire avec jQUery :

```
1  <script src="js/jquery.min.js"></script>
2  <script>
3  function ajax()
4  {
5    $.ajax({
6            url: "http://localhost/~login/ajax/ajax2.php",
7            type: "POST",
8            dataType: "text",
9            data:"annee=" + $('select[name=annee]').val(),
10           success: function(reponse) { affichage(reponse); },
11           error: function(req, status, err) {
12                   console.log('Problème ajax: '+ err)
13                   }
14           });
15 }
16 function getAnnee(){
17     return document.forms[0].annee.value;
18 }
19 function affichage(reponse){
20     console.log(reponse);
21     $('#affiche').html(reponse);
22 }
23 </script>
```

Puis une autre en utilisant directement l'API fetch/then, notez bien les headers que nous sommes obligés d'envoyer avec la méthode POST pour transmettre le paramètre annee :

```
1  function ajax(){
2      const url = new URL("http://localhost/~login/ajax/ajax2.php")
3      fetch(url,  {
4        method: 'POST', headers: {
5          "Content-type": "application/x-www-form-urlencoded;charset=
               UTF-8" },
6        body: 'annee=' + getAnnee()  })
7       .then( response => {
8              if (response.ok) return response.text();
9              else throw new Error('Problème ajax: '+response.status)
10         })
11      .then(affichage)
12      .catch(err => console.log(err));
13 }
14 function affichage(reponse){
```

```
15      document.getElementById("affiche").innerHTML = reponse;
16  }
```

on peut aussi faire une version avec XHR. Voir le corrigé si celà vous intéresse.

Exercice 3. REST

On va étudier à présent la notion d'architecture REST. Un projet vous sera proposé à réaliser cette année en PHP, intégrant une architecture REST entre un serveur et un Client riche (ou RIA) codé en JavaScript.

On rappelle les définitions :

- RIA = Rich Internet Application
- REST = Representational State Transform
- API = Application Programming Interface
- Logique métier déportée vers le client
- Tâche principale du serveur : Offrir des services de récupération et de stockage de données

Les technologies concurrentes à REST sont XML-RPC et SOAP (Microsoft) REST est une façon moderne de concevoir ce genre de service et possède les avantages suivants :

- Bonne montée en charge du serveur
- Simplicité des serveurs (retour aux sources du protocole HTTP)
- Equilibrage de charge
- le serveur offre une API
- les services sont représentés par des URL's donc simplicité et bonne gestion du cache
- Possibilité de décomposer des services complexes en de multiples services plus simples qui communiquent entre eux

Les principes de REST ont été théorisés par Roy Fielding dans sa thèse : `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm` :

1. Séparation claire entre Client et Serveur
2. Le client contient la logique métier, le serveur est sans Etat
3. Les réponses du serveur peuvent ou non être mises en cache
4. L'interface doit être simple, bien définie, standardisée
5. Le système peut avoir plusieurs couches comme des proxys, systèmes de cache, etc
6. Eventuellement, les clients peuvent télecharger du code du serveur qui s'exécutera dans le contexte du client

Pour mémoire, une API REST peut offrir les méthodes suivantes :

Méthodes HTTP et REST

Méthode	URI	Rôle	HTTP
GET	/articles/7	Récupérer un élement	200
GET	/articles	Récupérer une collection d'élements	200
POST	/articles	Poster une collection d'élements	201
DELETE	/articles/3	pour effacer un élement	200
PUT	/articles/5	pour modifier un élement	200

Mais on peut aussi avoir des erreurs :

- 400 Bad Request : requête mal formée
- 404 Not Found : la resource demandée n'existe pas
- 401 Unauthorized : Authentification necessaire pour accéder à la resource.
- 405 Method Not Allowed : Cette méthode est interdite pour cette resource.
- 409 Conflict : Par exemple un PUT qui crée une ressource 2 fois
- 500 Internal Server Error : Toutes les autres erreurs du serveur.

Par ailleurs, le serveur REST ne maintient pas d'état, les requêtes sont indépendantes les unes des autres. C'est un retour aux fondamentaux du protocole HTTP qui n'est pas doté de beaucoup de capacités de mémorisation ...

Exercice 4. mise en place rapide d'une API REST avec json-server

Côté serveur, on peut utiliser différentes technologies pour implémenter une architecture REST : PHP avec des Frameworks ou un simple CMS comme WordPress, Python avec Flask ou Django ou Node. Cette dernière technologie permet une mise en place très rapide pour des données simples comme nous allons le voir avec le module *json-server*. On va simplement installer localement ce paquet node et puis l'utiliser pour lancer un serveur sur une base décrite dans un document JSON. (NoSQL)

4.1 Installer le module json-server :

```
yarn add json-server
```

4.2 Observez la liste des paquets node installés localement :

```
yarn list --depth=0
```

4.3 Faire de même avec les paquets node installés globalement :

```
yarn list -g --depth=0
```

4.4 Partons d'une base simple de tâches à réaliser :

```
{
  "tasks": [
    {
      "id": 1
      "title": "Courses",
      "description": "Salade, Oignons, Pommes, Clementines",
      "done": true
    },
    {
      "id": 2,
      "title": "Apprendre REST",
      "description": "Apprendre mon cours et comprendre les exemples"
          ,
      "done": false
    },
    {
      "id": 3,
      "title": "Apprendre Ajax",
      "description": "Revoir les exemples et ecrire un client REST JS
           avec Ajax",
      "done": false
    }
  ]
}
```

4.5 Lançons notre serveur (sur le port 3000) :

```
./node_modules/json-server/lib/cli/bin.js ./tasks.json
```

4.6 Visitez la page http ://localhost :3000/tasks pour vérifier que tout va bien

4.7 Quelles sont les urls des tâches individuelles ?

4.8 Essayez ces routes dans le terminal en utilisant curl, en commençant par :

```
curl -H "Content-Type: application/json" -X GET http://localhost
   :3000/tasks/
```

Testez aussi les méthodes PUT, POST et DELETE. (cf fichier curl-i.txt fourni sur Celene)

Attention, il peut être nécessaire de définir la variable no_proxy :

```
export no_proxy="localhost, 127.0.0.1"
```

4.9 Installez éventuellement *Postman* dans votre HOME et utilisez la pour envoyer de nouveaux articles au serveur en utilisant les méthodes GET et POST. Essayez d'autres méthodes (PUT et DELETE). Postman peut permettre d'exporter toutes ces requêtes en JSON pour les rejouer automatiquement en nodeJS.

4.10 On peut également utiliser d'autres outils pour tester une API comme `Guzzle` en PHP ou le module `requests` en Python

Exercice 5. Client OnePage en JS

On va maitenant écire un client de type *One Page* en JS pour cette API en utilisant des requêtes ajax en JQuery et à l'aide de fetch/then.

5.1 On part de la page Web suivante :

```
<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8"/>
    <title>Vos tâches</title>
    <link rel="stylesheet" href="css/flex.css"/>
    <script src="js/jquery.min.js"></script>
    <script src="js/todo.js"></script>
  </head>
  <body>
 <header>
 <h1>Choses à faire</h1>
 </header>
 <div id='main'>
    <nav id="nav1">
      <h2>Todo</h2>
        <input id="button" type="button" value="Recuperer les taches" />
        <div id="taches">
        </div>
    </nav>
    <article>
      <h2>Editeur de Tâches</h2>
      <section id="tools">
        <img id="add" src="img/new.png" alt="Nouvelle chose à faire" />
        <img id="del" src="img/delete.png" alt="Enlever cette tache" />
      </section>
      <section id="currenttask"> </section>
    </article>
 </div>
 <footer>
 <h4>Departement Informatique - IUT d'Orleans</h4>
 </footer>
  </body>
</html>
```

Le fichier flex.css et le début de todo.js vous serons fournis.

Exercice 6. Retour à JavaScript et Ajax

Nous revenons maintenant au côté client. Nous supposons avoir le service REST de base avec un GET pour lensemble des ressources et un GET par ressource individuelle implémentés. Ecrivons à laide de JQuery un client simple qui permet dafficher la liste des ressources dans une liste html avec des liens vers les urls des tâches. Complétons le fichier todo.js.

6.1 Tout d'abord pour avoir un affichage de toutes les tâches

```
function refreshTaskList(){
        $("#currenttask").empty();
        $.ajax({
            url: "http://localhost:3000/tasks",
            type: "GET",
            dataType : "json",
            success: function(tasks) {
                console.log(JSON.stringify(tasks));
                $('#taches').empty();
                $('#taches').append($('<ul>'));
                for(var i=0;i<tasks.length;i++){
                    console.log(tasks[i]);
                    $('#taches ul')
                    .append($('<li>')
                    .append($('<a>')
                        .text(tasks[i].title)
                        ).on("click", tasks[i], details)
                    );
                        }
                    },
            error: function(req, status, err) {
                        $("#taches").html("<b>Impossible de récupérer
                            les taches à réaliser !</b>");
                        }
                        });
        }
```

6.2 La fonction *details* permettra l'affichage du détail d'une tâche lorsqu'elle sera sélectionnée dans la liste :

```
  function details(event){
        $("#currenttask").empty();
        formTask();
        fillFormTask(event.data);
        }
```

Une tâche sera représentée en JS de la façon suivante :

```
// Objet Task en JS
    class Task{
        constructor(title, description, done, uri){
            this.title = title;
            this.description = description;
            this.done = done;
```

```
7               this.uri = uri;
8               console.log(this.uri);
9           }
10      }
```

6.3 La fonction *formTask* permettra de présenter un formulaire (vierge ou non) détaillant une tâche :

```
1   function formTask(isnew){
2           $("#currenttask").empty();
3           $("#currenttask")
4               .append($('<span>Titre<input type="text" id="titre"><br
                    ></span>'))
5               .append($('<span>Description<input type="text" id="descr
                    "><br></span>'))
6               .append($('<span>Done<input type="checkbox" id="done"><br
                    ></span>'))
7               .append($('<span><input type="hidden" id="turi"><br></
                    span>'))
8               .append(isnew?$('<span><input type="button" value="Save
                    Task"><br></span>').on("click", saveNewTask)
9                           :$('<span><input type="button" value="Modify
                                Task"><br></span>').on("click",
                               saveModifiedTask)
10                  );
11          }
```

6.4 La fonction *saveNewTask* permettra d'envoyer une nouvelle tâche vers votre serveur :

```
1   function saveNewTask(){
2           var task = new Task(
3               $("#currenttask #titre").val(),
4               $("#currenttask #descr").val(),
5               $("#currenttask #done").is(':checked')
6               );
7           console.log(JSON.stringify(task));
8           $.ajax({
9               url: "http://localhost:3000/tasks",
10              type: 'POST',
11              contentType: 'application/json',
12              data: JSON.stringify(task),
13              dataType: 'json',
14              success: function (msg) {
15                  alert('Save Success');
16              },
17              error: function (err){
18                  alert('Save Error');
19              }
20              });
21          refreshTaskList();
22      }
```

6.5 Proposez ensuite une méthode *saveModifiedTask* permettant de modifier une tâche en utilisant la méthode HTTP PUT.

6.6 Puis une méthode *delTask* permettant sa suppression via la méthode HTTP DELETE.

Exercice 7. Client JS avec Promesses fetch/then/catch

On reprend ensuite l'exercice sans utiliser la méthode ajax de JQuery mais en utilisant les Promesses accessibles via *fetch()/then()*

7.1 La fonction refreshTaskList devient ainsi

```
function refreshTaskList(){
        $("#currenttask").empty();
        requete = "http://localhost:3000/tasks";
        fetch(requete)
        .then( response => {
                  if (response.ok) return response.json();
                  else throw new Error('Problème ajax: '+response.
                     status);
                }
            )
        .then(remplirTaches)
        .catch(onerror);
      }
```

avec les fonctions auxiliaires :

```
 function remplirTaches(tasks) {
      console.log(JSON.stringify(tasks));
      $('#taches').empty();
      $('#taches').append($('<ul>'));
      for(var i=0;i<tasks.length;i++){
          console.log(tasks[i]);
          $('#taches ul')
                .append($('<li>')
                .append($('<a>')
                .text(tasks[i].title)
                    ).on("click", tasks[i], details)
                );
        }
      }

      function onerror(err) {
          $("#taches").html("<b>Impossible de récupérer les taches à
             réaliser !</b>"+err);
      }
```

7.2 La fonction saveNewTask implémente un POST de la manière suivante :

```
function saveNewTask(){
        var task = new Task(
            $("#currenttask #titre").val(),
            $("#currenttask #descr").val(),
            $("#currenttask #done").is(':checked')
            );
        console.log(JSON.stringify(task));
        fetch("http://localhost:3000/tasks/",{
        headers: {
          'Accept': 'application/json',
          'Content-Type': 'application/json'
        },
        method: "POST",
        body: JSON.stringify(task)
            })
        .then(res => {
              console.log('Save Success') ;
              $("#result").text(res['contenu'])})
        .catch( res => { console.log(res) });
        refreshTaskList();
    }
```

Remarquez le traitement des erreurs. L'appel à la fonction `refreshTaskList()` est-il bien placé ?

7.3 Implémentez de même l'appel aux méthodes PUT et DELETE de votre API

7.4 Pour en savoir plus sur fetch/then, consulter `https://developer.mozilla.org/fr/docs/Web/API/Fetch_API/Using_Fetch`. Un *polyfill* est disponible pour assurer de la compatibilité avec les anciens navigateurs : `https://github.com/github/fetch`. Si vous utilisez jQuery, pas besoin de polyfill.