Introduction à REST

Interrogation d'une API REST avec JQuery et fetch/then

Exercice 1. Rappels REST

Dans ce TD, on va introduire la notion d'architecture REST. Un projet vous sera proposé à réaliser cette année sur la base d'une architecture REST entre un serveur et un Client riche (ou RIA) réalisé sous forme d'une Application sur une seule page (SPA) JavaScript.

On rappelle les définitions :

- RIA = Rich Internet Application
- REST = Representational State Transform
- API = Application Programming Interface
- SPA = Single Page Application
- 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 2. 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)

2.1 Installer le module json-server :

```
yarn add json-server
```

2.2 Observez la liste des paquets node installés localement :

```
npm ls -l --depth=0
```

2.3 Faire de même avec les paquets node installés globalement :

```
npm ls -g --depth=0
```

2.4 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
    }
  ]
}
```

2.5 Lançons notre serveur (sur le port 3000) :

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

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

2.7 Quelles sont les urls des tâches individuelles ?

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

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

2.9 Installez *Postman* et utilisez le pour envoyer de nouveaux articles au serveur en utilisant la méthode POST. Essayez d'autres méthodes. Créez une collection de requêtes de test.

Exercice 3. 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.

3.1 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>
```

```
30   <footer>
31   <h4>Departement Informatique - IUT d'Orleans</h4>
32   </footer>
33    </body>
34  </html>
```

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

Exercice 4. Retour à JavaScript et Ajax

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

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

```
1  function refreshTaskList(){
2          $("#currenttask").empty();
3          $.ajax({
4              url: "http://localhost:3000/tasks",
5              type: "GET",
6              dataType : "json",
7              success: function(tasks) {
8                  console.log(JSON.stringify(tasks));
9                  $('#taches').empty();
10                 $('#taches').append($('<ul>'));
11                 for(var i=0;i<tasks.length;i++){
12                     console.log(tasks[i]);
13                     $('#taches ul')
14                     .append($('<li>')
15                     .append($('<a>')
16                         .text(tasks[i].title)
17                         ).on("click", tasks[i], details)
18                     );
19                         }
20                     },
21             error: function(req, status, err) {
22                         $("#taches").html("<b>Impossible de r
                               écupérer les taches à réaliser !</
                               b>");
23                         }
```

```
24                              });
25          }
```

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

```
1     function details(event){
2           $("#currenttask").empty();
3           formTask();
4           fillFormTask(event.data);
5           }
```

4.3 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          }
```

4.4 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      }
```

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

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

Exercice 5. 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()*

5.1 La fonction refreshTaskList devient ainsi

```
 1  function refreshTaskList(){
 2          $("#currenttask").empty();
 3          requete = "http://localhost:3000/tasks";
 4          fetch(requete)
 5          .then( response => {
 6                    if (response.ok) return response.json();
 7                    else throw new Error('Problème ajax: '+
                         response.status);
 8                  }
 9              )
10          .then(remplirTaches)
11          .catch(onerror);
12        }
```

avec les fonctions auxiliaires :

```
1  function remplirTaches(tasks) {
2       console.log(JSON.stringify(tasks));
3       $('#taches').empty();
4       $('#taches').append($('<ul>'));
5       for(var i=0;i<tasks.length;i++){
6           console.log(tasks[i]);
7           $('#taches ul')
8                 .append($('<li>')
9                 .append($('<a>')
10                .text(tasks[i].title)
11                    ).on("click", tasks[i], details)
12                );
13        }
14      }
15
16      function onerror(err) {
17          $("#taches").html("<b>Impossible de récupérer les
              taches à réaliser !</b>"+err);
18      }
```

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

```
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          fetch("http://localhost:3000/tasks/",{
9          headers: {
10           'Accept': 'application/json',
11           'Content-Type': 'application/json'
12         },
13         method: "POST",
14         body: JSON.stringify(task)
15             })
16         .then(res => {
17               console.log('Save Success') ;
18               $("#result").text(res['contenu'])})
19         .catch( res => { console.log(res) });
20         refreshTaskList();
21     }
```

Remarquez le traitement des erreurs.

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

5.4 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.

Exercice 6. Données plus complexes avec json-server

json-server peut vous permettre d'exposer en APi des données un peu plus complexes ayant des relations entre-elles comme par exemple :

```
{
  "articles": [
    {
      "id": 1,
      "contenu": "Bonjour tout le monde !",
      "auteur": "jrobert"
    },
    {
      "id": 2,
      "contenu": "Le JS c'est super.",
      "auteur": "jarsouze"
    },
    {
      "id": 3,
      "contenu": "Les API REST c'est quoi ?",
      "auteur": "groza"
    },
    {
      "id": 4,
      "contenu": "Comment on fait des Promesses en JS ?",
      "auteur": "sbouez"
    },
    {
      "contenu": "Comment ça marche fetch et then ?",
      "auteur": "jarsouze",
      "id": 5
    },
    {
      "contenu": "Rest pour tous",
```

```
30        "auteur": "sbouez",
31        "id": 6
32      },
33
34    ],
35    "commentaires": [
36      {
37        "id": 1,
38        "contenu": "Bonjour.",
39        "articleId": 1,
40        "auteur": "john"
41      },
42      {
43        "id": 2,
44        "contenu": "Quelle politesse.",
45        "articleId": 1,
46        "auteur": "jules"
47      },
48      {
49        "id": 3,
50        "contenu": "Oui, c'est super REST !",
51        "articleId": 2,
52        "auteur": "zorro"
53      },
54      {
55        "id": 4,
56        "contenu": "Super bien.",
57        "articleId": 4,
58        "auteur": "jimmy"
59      },
60      {
61        "id": 5,
62        "contenu": "Je me le demande..",
63        "articleId": 5,
64        "auteur": "johnny"
65      }
66    ]
67  }
```

Cette base de données est une version simplifiée de ce que pourrait contenir un blog. Il y a deux tables :

— une table "articles" contenant pour chaque article un id, un contenu et un auteur.

— une table "commentaires" contenant pour chaque commentaire, un id, un contenu, un auteur, et l'id de l'article auquel ce commentaire se réfère.

Vous pouvez accéder par exemple à l'article numéro 1 sous format JSON à la route :

```
http://localhost:3000/articles/1
```

De la même manière, vous pouvez lister tous les articles à la route :

```
http://localhost:3000/articles
```

De même pour les commentaires :

```
http://localhost:3000/commentaires
http://localhost:3000/commentaires/1
```

Exercice 7. Client OnePage

Proposez maintenant un client *One Page* avec un layout de votre choix qui permet d'effectuer toutes les actions de base sur cette API d'articles en utilisant exclusivement fetch/then.