

REST Sécurité- Travaux pratiques

L'énoncé est à adapter que vous soyez sur Linux, Windows ou Mac.

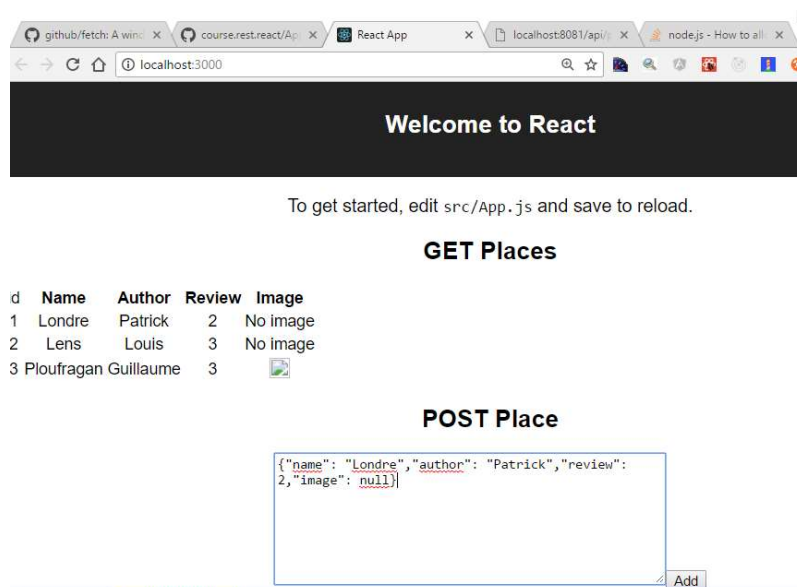
Le TP contient des questions auxquelles il faut répondre, elles sont en lien avec les manipulations.

Le but de ce TP :

Nous allons maintenant consommer l'API du TP précédent. Vous allez avoir les problématiques de « Cross Domain » et d'authentification. Il faut à minima avoir les routes GET et POST en état de fonctionnement. Il n'est pas nécessaire d'avoir réalisé les dernières parties du TP précédent (implémentation MongoDB par exemple). **Aujourd'hui ce que vous désirez est de contrôler le plus finement que possible tous les types d'accès à votre API.**

```
{
  name:"Place name",
  author:"Author",
  review: 0,
  image: { <= null si pas d'image
    url:"",
    title:""
  }
}
```

La data




localhost:3000

Welcome to React

To get started, edit `src/App.js` and save to reload.

GET Places

id	Name	Author	Review	Image
1	Londre	Patrick	2	No image
2	Lens	Louis	3	No image
3	Ploufragan	Guillaume	3	

POST Place

```
{ "name": "Londre", "author": "Patrick", "review": 2, "image": null }
```

Add

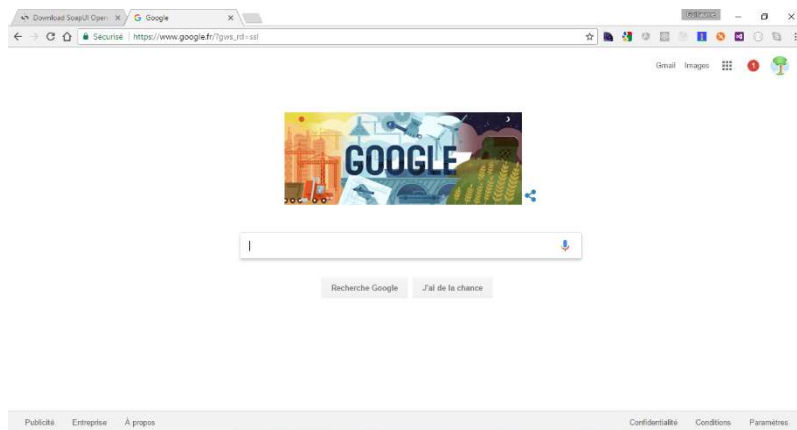
Les notions abordées :

- Protéger votre api des appels Cross Domain via le protocole HTTP CORS (client et serveur)
- Gérer finement les caches HTTP
- Protéger votre API avec une authentification l'utilisation de JWT Token

1. Installation

- Node.js dernière version LTS (Long Time Support)
- Client Git dernière version
- Visual Studio code dernière version (c'est un logiciel gratuit)
- Logiciel Postman

Vous pouvez utiliser un moteur de recherche :



2. Création d'une application client

Vous allez créer une application cliente moderne à l'aide du Framework « React ». Aucune connaissance n'est requise sur ce Framework pour le TP. Nous allons nous baser sur le starter kit créée par Facebook.

- Starter kit Facebook : <https://github.com/facebookincubator/create-react-app>
- Si vous voulez aller plus loin, la documentation officielle du Framework : <https://facebook.github.io/react/docs/hello-world.html>

Placez-vous à la racine du répertoire de vos TP ; par exemple « C:/TP/WebServices/ », on va créer un nouveau projet, donc un nouveau répertoire :

1. Exécuter les commandes ci-dessous (il vous faut une connexion sans proxy) :

```
npx create-react-app rest.client  
cd rest.client/  
npm start
```

Une fois le « start » effectué, la page se rafraichie automatiquement à chaque mise à jour de votre code. Cette fonctionnalité s'appelle du « live-reload ».

2. Il vous faut maintenant ajouter un peu de code afin d'appeler votre API de place. Dans le fichier situé dans « src/App.js » ajouter le code ci-dessous en gras.

```

import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.state = { places : [] } ;
  }
  componentDidMount() {
    const _self = this;
    fetch('http://localhost:8081/api/places', {
      method: 'GET',
      headers: {}
    }).then(function(response){
      if (response.status >= 200 && response.status < 300) {
        return response.json();
      } else {
        var error = new Error(response.statusText)
        error.response = response;
        throw error;
      }
    }).then(function(data){
      _self.setState({ places: data.places });
    }).catch(function(error){
      console.log(error);
    });
  }
  render() {
    const listItems = this.state.places.map((place) =>
      <tr key={place.id.toString()}>
        <td>{place.id}</td>
        <td>{place.name}</td>
        <td>{place.author}</td>
        <td>{place.review}</td>
        <td>{place.image? (<img src={place.image.url} />): 'No image' }</td>
      </tr>
    );

    const tableHead = (<tr>
      <td>id</td>
      <th>Name</th>
      <th>Author</th>
      <th>Review</th>
      <th>Image</th>
    </tr>);

    const placeItems = this.state.places.length <= 0 ?
      (<p>Chargement en cours</p>) :
    (<table><tbody>{tableHead}{listItems}</tbody></table>);
  }
}

```

```

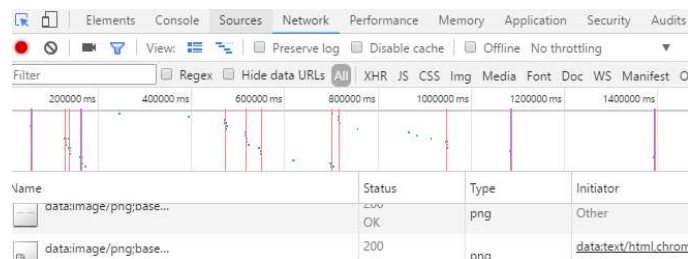
return (
  <div className="App">
    <div className="App-header">
      <img src={logo} className="App-logo" alt="logo" />
      <h2>Welcome to React</h2>
    </div>
    <p className="App-intro">
      To get started, edit <code>src/App.js</code> and save to reload.
    </p>
    <h2>GET Places</h2>
    {placeItems}
  </div>
);
}
}

export default App;

```

3. CORS et http GET

Vous appelez maintenant depuis l'url <http://localhost:3000> votre API située sur l'url <http://localhost:8081>. Vous pouvez observer dans le debugger chrome (ctrl+shift+i) les requêtes qui sont réalisées vers votre API.



Vous pouvez constater que votre requête http GET est réalisée sur un domaine différent de l' « origin ». La page reste bloquée sur le message « chargement en cours ».

1. Est-ce que le **serveur** a reçu et renvoie bien une réponse HTTP ?

2. Quel comportement **client** constatez-vous ? (Par exemple : Est-ce que ça marche ? à quel endroit/moment est-ce que ça bloque ?)

3. Effectuer la correction pour que votre browser web « client » puisse accepter la réponse provenant du serveur. Vous ne devez pas utiliser de middleware node.js express qui gère le « cors ». Ni utiliser « * » (qui entraîne une faille de sécurité). Utilisez la méthode « [response.setHeader](#) » côté serveur.
4. Quels HTTP Headers (ainsi que leurs valeurs) avez-vous dû ajouter afin que l'appel fonctionne correctement et que le tableau de place s'affiche ? Préciser si l'ajout que vous avez réalisé est côté serveur ou client.

5. Pourquoi les références vers les images via la balise qui sont aussi exposées par votre API, n'ont pas la problématique ci-dessus ?

6. Ajoutez un http header comme ci-dessous afin que votre client l'envoie à votre API.

```
fetch('http://localhost:8081/api/places', {
  method: 'GET',
  headers: {
    'my-header-custom': 'i love places'
  }
}).then(function(response){
  if (response.status >= 200 && response.status < 300) {
    return response.json();
  } else {
    var error = new Error(response.statusText)
    error.response = response;
    throw error;
  }
}).then(function(data){
  _self.setState({ places: data.places });
}).catch(function(error){
  console.log(error);
});
}
```

7. Quel comportement constatez-vous au niveau des requêtes réalisées par votre client ? (Par exemple : Est-ce que ça marche ? A quel endroit est-ce que ça bloque ?)

8. Réaliser le code serveur qui va permettre de refaire fonctionner votre client. Quelle méthode http avez-vous du ajouter du côté du serveur ? et quel header http (ainsi que sa valeur) avez-vous du ajouter ?

4. Un peu de cache

Il y a maintenant beaucoup de requêtes réalisées envers votre serveur, qui s'écroule au niveau des performances. Vous souhaitez diminuer le nombre de requêtes.

Vous estimez que vos requêtes de type http get peuvent être mise en cache pendant 15 secondes (par défaut le Framework express de node.js ajoute le http header **ETag** sur les réponses).

1. Quel http header (ainsi que sa valeur) faut-il ajouter pour mettre en cache pendant 15 secondes le http get ?

2. Quel est le code retour de votre API lorsque les 15 secondes sont passées et qu'aucune donnée serveur n'a été modifiée entre temps ? (Attention : il se peut que chrome masque le vrai code retour, utiliser « Firefox » qui affiche le bon)

3. Avez-vous déclaré les caches en mode « public » ou « privée » ? Justifier pourquoi ?

Vous estimez aussi maintenant que vos requêtes de type http options peuvent être mis en cache pendant 30 secondes.

4. Quel http header (ainsi que sa valeur) faut-il ajouter pour mettre en cache pendant 30 secondes les appels via la méthode HTTP OPTIONS ?

5. CORS et http POST

On va maintenant ajouter un « textarea » dans la page, ceci afin de pouvoir saisir les données qui vont permettre d'envoyer vers le serveur une place directement au format JSON. Il faut mettre à jour le fichier client « src/App.js ». Vous pouvez copier/coller le code depuis cette url git :

<https://github.com/guillaumechervet/course.rest.react/blob/master/src/App.js>

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

class App extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      places: [],
      value: ''
    };
  }

  componentDidMount() {
    const _self = this;
    fetch('http://localhost:8081/api/places', {
      method: 'GET',
      headers: {
        'my-header-custom': 'i love place'
      }
    }).then(function(response) {
      if (response.status >= 200 && response.status < 300) {
        return response.json();
      } else {
        var error = new Error(response.statusText);
        error.response = response;
        throw error;
      }
    }).then(function(data) {
      _self.setState({ places: data.places });
    });
  }
}
```

```

    }).catch(function(error){
        console.log(error);
    });
}
handleClick(event){
    event.preventDefault();
    console.log('The link was clicked. ');
    console.log(this.state.value);
    const _self = this;

    fetch('http://localhost:8081/api/places', {
        method: 'POST',
        body: this.state.value,
        headers: {
            'Content-Type': 'application/json'
        }
    }).then(function(response){
        if (response.status >= 200 && response.status < 300) {
            return response;
        } else {
            var error = new Error(response.statusText);
            error.response = response;
            throw error;
        }
    }).then(function(data){
        _self.setState({value: ''});
    }).catch(function(error){
        console.log(error);
    });
}
handleChange(event){
    this.setState({value: event.target.value});
}

render() {

    const listItems = this.state.places.map((place) =>
        <tr key={place.id.toString()}>
            <td>{place.id}</td>
            <td>{place.name}</td>
            <td>{place.author}</td>
            <td>{place.review}</td>
            <td>{place.image? (<img src={place.image.url} />): 'No image' }</td>
        </tr>
    );

    const tableHead = (<tr>
        <td>id</td>
        <th>Name</th>
        <th>Author</th>

```



```

    <th>Review</th>
    <th>Image</th>
  </tr>);
  const placeItems = this.state.places.length <= 0 ?
    (<p>Chargement en cours</p>) :

  (<table><tbody>{tableHead}{listItems}</tbody></table>);

  return (
    <div className="App">
      <div className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
        <h2>Welcome to React</h2>
      </div>
      <p className="App-intro">
        To get started, edit <code>src/App.js</code> and save to reload.
      </p>
      <h2>GET Places</h2>
      {placeItems}
      <h2>POST Place</h2>
      <textarea rows="8" cols="50" value={this.state.value}
        onChange={this.handleChange} />
      <input type="submit" value="Add" onClick={this.handleClick} />
    </div>
  );
}
}

export default App;

```

Un Google ira plus vite que d'écrire à la mains l'url vers le git. C'est bien sûr ce choix que vous avez fait 😊 car vous êtes malin.

Comme pour le http GET,

1. Qu'avez-vous du ajouter/modifier côté serveur afin que l'appel http POST fonctionne ? (Au niveau du http OPTIONS, au niveau du http POST)



6. Route de login et JWT

Vous désirez créer une route « **/api/users/login** » en http POST qui va vous permettre d'authentifier vos utilisateurs. Vous aurez 1 utilisateur en dur dans le code (l'idée de cette partie est de vous faire manipuler le standard jwt sans pour autant avoir la complexité de gérer un fournisseur d'identité/authentification), les informations de login :

```
{  
  password: 'password',  
  username : gaston,  
};
```

Pour éviter les problèmes de cross domain, vous appellerez cette route via POST MAN (pas via le client web). Le but est de récupérer le token JWT que vous pourrez copier puis coller en dur dans le code coté client afin d'envoyer le token JWT au serveur (**en mode Bearer**) lors de l'appel http POST.

Cette route va vous permettre de récupérer un token JWT, qui va vous permettre de protéger l'accès à votre route http POST qui permet l'ajout de place. Uniquement les utilisateurs authentifiés auront le droit de POSTer des nouvelles places. La route http GET reste libre d'accès à tous.


Vous pourrez stocker l'information « username » dans le token JWT (ce qui évite normalement les appels à votre base de données). Ainsi la propriété « name » du http POST n'est plus obligatoire. Si elle n'est pas présente, la propriété « username » présente dans le token JWT est utilisée.

Les 2 librairies node.js qui sont à utiliser :

- <https://github.com/auth0/node-jsonwebtoken>
- <https://github.com/auth0/express-jwt>

Vous pouvez fortement vous aider de cette vidéo :

- <https://jwt.io/introduction/>

1. Réaliser toutes les modifications de code et manipulations
 2. Quel code retour devez-vous retourner lorsque vous appelez votre route « ajout de place » alors que vous n'êtes pas authentifié ? (C'est-à-dire que vous n'envoyez pas de token JWT en mode bearer)
- 

3. Quel HEADER http le client web est-il bien d'utiliser pour envoyer le token JWT en mode bearer ?

4. Imaginer que vous envoyez les JWT Token via des cookies (et non en mode bearer). Quel sont les 2 choses à modifier afin que les cookies soient envoyés via un domaine différent (Cross-Domain) ?

7. Authentication Grant Type « implicit »

Il existe des implémentations toutes faites de serveurs OpenID Connect (plutôt que d'en redévelopper un dans votre coin, ce qui vous prendrait des années). Par exemple en dotnet core :

- <https://github.com/IdentityServer> (dotnet core)
- <https://www.keycloak.org> (java)

Il existe aussi des fournisseurs SaaS (Software As A service) de serveur OpenID Connect :

- <https://demo.identityserver.io/>
- <https://auth0.com/>

Ces outils sont pratiques pour réaliser des développements locaux sur votre machine.

8. Authentication Grant Type « Authorization Code »

Si vous êtes arrivé jusqu'à cette section, un grand bravo !

L'idée ici est de maintenant connecter votre serveur web node.js à Facebook. Votre API va devenir un partenaire de Facebook. Vos utilisateurs du site web qui consomment votre API pourront ainsi s'authentifier à votre API via leur login Facebook.

Pour cette partie, vous pouvez abandonner la partie cliente (site web). Faire les actions via « Postman » vous aidera à comprendre les mécanismes qui se déroulent en « background ».

Utiliser le module node.js « passport-facebook » : <https://github.com/jaredhanson/passport-facebook> (Il vous suffit de suivre la documentation).

Pour la création de l'application sur Facebook (<https://developers.facebook.com/apps>), vous pouvez vous inspirer cet article qui décrit bien les opérations à réaliser :

<https://docs.microsoft.com/en-us/aspnet/mvc/overview/security/create-an-aspnet-mvc-5-app-with-facebook-and-google-oauth2-and-openid-sign-on> (sur Google, rechercher : **asp.net oauth facebook**)

1. Déclarer/configurer l'application sur le portail Facebook développeur.
2. Modifier le code de votre api web.
3. Quel type d'autorisation (workflow) OAUTH s'agit-il ? (Le nom est suffisant)

4. Que vous manque-t-il **d'essentiel** afin que votre API soit très sécurisée ?