Node TP-06

Création du projet

Créez un répertoire de projet « etudiantserver », rentrez dans le nouveau répertoire puis tapez la commande :

npm init

Elle va lancer la création du projet :

Validez le nom du package (nom du répertoire)

Validez la version (1.0.0)

Description : écrivez une description de la fonction du serveur

Entry point : Le fichier de lancement du server. Validez.

Test command : Script npm de test. Validez.

git repository : Validez.

keywords : Validez.

Author : renseignez fièrement votre nom, classiquement on met son mail. Validez.

Licence : par défaut ISC, ou pour info les autres types de licence. Validez.

Validez une dernier fois pour confirmer tous les choix.

Le projet est créé… Ou plutôt, le fichier package.json est créé.

On va préparer un peu notre projet !

Actuellement on est en ECMA 5, mais on souhaite profiter des fonctionnalités de ECMA 6.

Pour cela il va falloir trans-compiler notre code avant qu'il puisse être compilé par javascript.

On va créer un répertoire pour héberger notre application front et un pour notre server

Créez le répertoire app à la racine du projet, puis le répertoire server à la racine.

Dans le répertoire server, on va créer le fichier de lancement serveur.

Créez server.js.

Installer Les composants babel :

npm install --save-dev @babel/cli @babel/core @babel/preset-env

Créer un fichier .babelrc à la racine projet, et éditer le :

```
{
  "presets": [
    ["@babel/preset-env",
      {
        "targets": {
          "browsers": ["last 2 Chrome versions"]
        }
      }
    ]
  ]
}
```

Il contient le paramétrage de Babel.

Maintenant, on va créer les scripts pour trans-compiler notre projet avant exécution.

Ouvrez avec un éditeur package.json, dans l'objet « scripts » ajouter les lignes suivantes :

"build": "babel server/ -d dist && rm -rf dist/app && cp -r app/ dist/",

"start": "npm run build && node dist/server.js",

"inspect": "node --inspect dist/server.js",

Build : transcompile le code et le met à disposition dans le répertoire dist, puis il vide et copie l'application front pour la rendre disponible dans le répertoire dist.

Start : lance le script build puis démarre le serveur depuis le répertoire dist.

Inspect : lance le serveur depuis le répertoire dist avec la possibillité dans la console de dev chrome d'ouvrir une nouvelle console de débuging pour la partie Node. On peut aussi utiliser le commutateur --inspect-brk qui mettra un breakpoint au début du script node.

On en profite pour mettre à jour le main :

"main": "server/server.js",

Presque prêt …

Maintenant on va ajouter avec npm un module :

npm install express

Express est l'outil qui va nous permettre de gérer notre serveur Node.

Voir la documentation : express

C'est bon on est prêt.

Publier le projet Angular

1ere étape récupérer le projet Angular.

Ouvrez une console a la racine du projet etudiant :

lancer : ng build. (avant de faire un build, il est préférable de tester en local avec ng server le front et de lancer le build uniquement pour une version stable.)

Un répertoire dist va se créer à la racine du projet.

Copiez le répertoire dist/etudiants et collez-le dans le répertoire app du projet etudiantserver.

Ouvrez le fichier server.js dans le répertoire server.

```
import express from 'express';

const app = express();


app.use(express.static('./app/etudiants'))


app.get('/', function (req, res) {
  res.sendFile('index.html');
});
app.listen(8080);
```

En haut on fait l'import au format ecma6,

On crée un objet express pour gérer le serveur.

App.use on lui applique une route static pour qu'il retrouve notre projet front.

App.get on dit au serveur que s'il reçoit une requête get a la racine il retourne le fichier index.html à la racine de notre site front.

App.listen : on dit au server d'écouter les requêtes sur le port 8080.

C'est prêt !

Ouvrez une commande et lancez à la racine du projet server :

npm run start.

Ou npm start.

Le projet se transcompile, met à disposition le front et lance le server.

Ouvrez Chrome sur la page http://localhost:8080

Testez !

C'est sympa mais la donnée est toujours dans l'application front.

Créer des routes pour échanger avec le serveur.

On va basculer les données utilisateurs du front au server.

Dans le projet front :

On va rajouter un service pour gérer les connexions au server.

Dans le fichier app.module.ts, on va importer HttpClientModule.

import { HttpClientModule } from '@angular/common/http';

et l'ajouter dans les imports :

imports : […,

HttpClientModule,

…]

Créer un fichier http.service.ts dans le répertoire service et éditer le :

```
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs/Observable';
import { Injectable } from '@angular/core';

@Injectable({providedIn : 'root'})
export class HttpService {
  private serverUrl = 'http://localhost:8080/';
  constructor (private http: HttpClient) {  }

  public getUsers() Observable<any>{
    return this.http.get(this.serverUrl+'users') ;
  }
}
```

Ce service va nous permettre d'injecter le httpClient dans des components et de récupérer les données sur le serveur.

On va connecter la vue à la fonction getUsers().

Dans user-list component importez et injectez dans le constructeur HttpService.

Puis modifiez le ngOnInit :

```
    this.userSubscription = this.http.getUsers().subscribe(
      (users: User[]) => {
        this.users = users;
```

```
    }
  );
```

Testez !

La compilation va marcher mais au moment où vous accédez à la liste des utilisateurs, la liste restera vide et vous aurez une erreur dans la console chrome.

Avant de retourner coté serveur on va récupérer la liste des users dans users services.

Puis dans le projet on va ajouter un répertoire services dans server puis créer un fichier users.js :

On va importer express,
puis créer l'objet router avec express.Router().
On crée aussi la variable users de type array() avec notre user récupéré depuis le service user du front.

Et on crée la route :

```
router.get('/', function (req, res) {
  res.send(users);
})
```

Et on exporte le router :

```
export default router
```

La correction page suivante.

```
import express from 'express';

const router = express.Router();
let users = [{firstName:'Will',
  lastName: 'Alexander',
  email: 'will@will.com',
  diploma: 'License 3 Informatique',
  options: ['web application', 'baby-foot']
}];

router.get('/', function (req, res) {
  res.send(users);
})

export default router;
```

Il faut maintenant importer la route dans le main server.js

```
import router from './services/users';
```

Et on va dire au serveur d'envoyer toutes les routes de type /users vers le service user.

```
App.use('/users', router) ;
```

Faite un build du front, copier la dist dans le répertoire app du server.
Puis lancer npm run start dans le répertoire server.

Testez !

La liste est là mais maintenant c'est le server qui l'héberge. Par contre l'ajout de user ne fonctionne plus.

Créer une route pour ajouter un user

On recommence l'opération côté front en 1er.
On va éditer le new-user.component,

On injecte le service HTTP Service dans le component :

```
import { HttpService } from '../services/http.service';

constructor( …, private httpService: HttpService)
```

Dans la fonction onSubmitForm, on va supprimer l'appel au service user et le remplacer par un appel au service http.

```
this.httpService.createUser(newUser).subscribe((response)=>{
  if(response && response.firstName === 'ok') {
    alert('User crée');
  } else {
    alert('User Existe !');
  };
```

```
}, (e) => {
console.log('erreur',e);
}, ()=>{
  this.router.navigate(['/users']);
}
);
```

La fonction appelle createUser en envoyant un objet de type User et en s'abonnant à la réponse.

Dans la réponse, on teste si le service a renvoyé 'ok' à la place du firstName.

On peut supprimer l'import et l'injection du service user. On peut aussi supprimer le service user et son import dans app.module.

On édite ensuite le fichier http.service.ts :

On va importer le model User et HttpHeaders depuis @angular/common/http,

Et on va créer une variable pour modifier les headers :

```
private httpOptions = {
    headers: new HttpHeaders ({
        'Content-Type': 'application/json'
    })
}
```

En modifiant le content-type du header on prévient le server sur le type d'envoi pour lui permettre de récupérer les données.

Puis on crée la méthode pour appeler le serveur :

```
public createUser(user): Observable<User> {
    return this.http.post<User>(this.serverUrl+'users', user, this.httpOptions);
}
```

De la même manière que pour le get on s'abonne avec un Observable de type User.
On passe toujours l'url en 1er paramètre, puis l'objet qui sera dans le body de la requête et enfin l'objet qui continent les options.

Correction page suivante :

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs/Observable';
import { User } from '../models/user.model';

@Injectable({providedIn : 'root'})
export class HttpService {

  private serverUrl = 'http://localhost:8080/';

  constructor (private http: HttpClient) {

  }

  private httpOptions = {
    headers: new HttpHeaders ({
      'Content-Type': 'application/json'
    })
  }

  public getUsers(): Observable<any> {
    return this.http.get(this.serverUrl+'users')
  }

  public createUser(user): Observable<User> {
    return this.http.post<User>(this.serverUrl+'users', user, this.httpOptions);
  }
}
```

Testez le front ! On n'a ni la liste ni la possibilité d'ajouter les users mais on peut voir dans la console que le problème est au niveau des échanges avec le serveur.

On build le front, on vide le répertoire serveur app/ et on copie le front dedans.

Retour côté serveur :

1ere étape, il va falloir parser le fichier json que l'on va recevoir pour la demande d'ajout d'un user.

Mais express est là pour nous avec sa fonction express.json(). Dans server.js, on va rajouter la fonction à notre app avec :

```
app.use(express.json());
```

Attention à rajouter cette commande avant les autres app.use() ! Sinon, la fonctionnalité de parsing de json sera indisponible.

Puis on va éditer le service user :

On a prévu côté front d'observer un objet de type User. Et de procéder à l'ajout d'un user que si son firstName n'existe pas dans la liste.

```
router.post('/', function (req, res) {
  const userExist = users.find(user => user.firstName === req.body.firstName);

  If (userExist) {
    res.send({});
  } else {
    users.push(req.body);
    res.send({ firstName : ‘ok’ });
  }
}
```

D’abord, on cherche s’il y a déjà un user avec un firstName identique. S’il ne le trouve pas userExist prend la valeur undefined.

Puis en fonction du retour, on renvoie un objet vide ou on push le user dans le tableau users et on renvoi l’objet d’origine.

Testez !

Là c’est bon la fonction addUser remarche.

On a fini les échanges avec le front pour confier la gestion des users au serveur.
Mais pour l’instant les données sont hébergées directement dans le code.