TP NodeJS step-by-step

TOUS LES FICHIERS JAVASCRIPT (.js*) DOIVENT COMMENCER PAR** `'use strict';`

1. Créer l'arborescence suivante

```
- tp
|       \- public
|       |
|       +- app
|       |       \- controllers
|       |       +- models
|       |       +- routes
|       |       +- utils
|       |       |       \- utils.js
|       +- presentation_content
|       +- uploads
|       +- app.js
|       +- config.json
```

2. Initialiser le projet avec npm. Le main de l'application va s'appeler *app.js*.

```
npm init
```

3. Installer express comme dépendance et l'ajouter au fichier package.json

```
npm install express --save
```

4. Modifier le fichier *package.json* pour rendre privée l'application et ajouter un script de démarrage

```
// package.json
[...]
"scripts": {
        "test": "node testContentModel.js",
        "start": "node app.js"
},
"private": true,
[...]
```

5. Créer le fichier *app.js* et ajouter un `console.log('It Works !')`. Démarrer votre application avec npm.

```
npm start
```

Point de validation 1

6. Initialiser express

```
// app.js
var express = require("express");
[...]
var app = express();
```

7. Créer le fichier *config.json* à la racine du projet, et l'alimenter avec le port d'écoute du serveur.

```
// config.json
{
        "port": 1337
}
```

8. Initialiser votre serveur web en utilisant express et la bibliothèque http de NodeJS. Récupérer le port d'écoute depuis le fichier de configuration *config.json*.

```
// app.js
var http = require("http");
var CONFIG = require("./config.json");
[...]
// init server
var server = http.createServer(app);
server.listen(CONFIG.port);
```

   - Pour que la configuration soit accessible par tous les modules pour la suite, déclarer une variable `CONFIG` dans `process.env` et injecter la configuration en JSON "stringifier" comme ceci

```
// app.js
var CONFIG = require("./config.json");
process.env.CONFIG = JSON.stringify(CONFIG);
```

   - Ainsi, dans les autres modules, l'accès à la configuration sera faite comme cela

```
var CONFIG = JSON.parse(process.env.CONFIG);
```

9. Faire en sorte que la route *"/"* réponde *"It works"*.

   - La "meilleure" façon de faire une route est de créer un router (`express.Router()`) dans un nouveau fichier (*default.route.js*) dans le répertoire *route*. Ce fichier se compose comme ceci:

```
// default.route.js
var express = require("express");
var router = express.Router();
module.exports = router;

// Routing using
/*
router.route(__PATH__)
        .get()
        .post()
        .put()
        .delete()
        .all()
        .[...]
*/
```

   - Dans *app.js*, on importe la nouvelle route et on l'utilise avec `app.use(myRoute)` . On peut également passer comme premier argument le chemin d'accès de la route (`app.use([URI], myRoute)`). Dans ce cas, les chemins indiqués dans le routeur sont alors relatifs.

```
// app.js
var defaultRoute = require("./app/routes/default.route.js");
[...]
app.use(defaultRoute);
```

   - Pour des routes simples, on peut aussi les gérer directement dans *app.js*.

```
// #2
app.get("/", function(request, response) {
        response.send("It works !");
});

// #3
app.use(function(request, response, cb) {
        response.send("It works !");
        cb();
});
```

10. Récupérer les projets Angular et les déposer dans *public/*.

```
- tp
|       \- public
|       |       \- admin
|       |       |       \- [...]
|       |       +- watch
|       |       |       \- [...]
```

11. Créer les routes statiques pour les pages *admin* et *watch* directement dans *app.js*. Utiliser la méthode `express.static` .

```
// app.js
var path = require("path");
[...]
app.use("/admin", express.static(path.join(__dirname, "public/admin")));
app.use("/watch", express.static(path.join(__dirname, "public/watch")));
```

Point de validation 2

12. Mettre à jour le fichier *config.json* en ajoutant les paramètres suivants:
    - ***contentDirectory***: chemin d'accès absolu vers le répertoire *uploads*. Ce répertoire contiendra les fichiers de données et de métadonnées associées à ces fichiers. Les métadonnées seront stockées en JSON.
    - ***presentationDirectory***: chemin d'accès absolu vers le répertoire *presentation_content*. Ce répertoire contiendra les métadonnées de présentation au format JSON.
13. Créer les services *"/loadPres"* et *"/savePres"*. Ces 2 services seront créés directement dans app.js (Cf #9.3).

    13.1. Le service "*/loadPres*". Ce service doit envoyer la liste de toutes les présentations présentes dans le répertoire *CONFIG.presentationDirectory*. Pour ce service, on lit le contenu de tous les fichiers `*.json` de présentation contenus dans *CONFIG.presentationDirectory*, on parse le contenu des fichiers pour extraire les données et on retourne un objet JSON au format "clé-valeur". La clé est l'ID de la présentation et la valeur est l'objet retourné par le parseur JSON.

```
{
        "pres1.id": [Object_Pres1],
        "pres2.id": [Object_Pres2],
        "pres3.id": [Object_Pres3]
        ...
}
```

    13.2. Le service "*/savePres*". Pour ce service, on récupère des données au format JSON et on les enregistre dans le répertoire *CONFIG.presentationDirectory* dans un fichier qui doit s'appeler *[pres.id].pres.json*. L'ID est a récuperer dans les données reçues.

Point de validation 3

14. Créer le modele de donnée pour le contenu des slides.
    - Créer le fichier *content.model.js* dans *app/models/*.
    - Ce fichier va contenir la "classe" ***ContentModel*** avec la définition suivante:
        - ***attributs***
            - **type**: public - ['img', 'img_url', 'video', 'web']
            - **id**: public - UUID
            - **title**: public
            - **src**: public - l'URL qui permet d'acceder au contenu
            - **fileName**: public - le nom du fichier stocké dans *[CONFIG.contentDirectory]* dans le cas d'un contenu de type 'img'. Il correspond a l'id du contenu + l'extension qui sera récupérée à partir du fichier original (png, jpeg...).
            - **data**: privé - accessible par getData() et setData()
        - ***méthodes*: /!\ Toutes ces méthodes doivent être statiques**

        - **create(content, callback)**: Prend un objet *contentModel* en paramètre, stocke le contenu de *[content.data]* dans le fichier *[content.fileName]* (dans le cas d'un contenu de type 'img') et stocke les meta-données dans un fichier *[contentModel.id].meta.json* dans le répertoire *[CONFIG.contentDirectory]*.
        - **read(id, callback)**: Prend un id en paramètre et retourne l'objet *contentModel* lu depuis le fichier *[content.id].meta.json*
        - **update(content, callback)**: Prend l'id d'un *ContentModel* en paramètre et met à jour le fichier de metadata (*[content.id].meta.json*) et le fichier *[content.fileName]* si *[content.data]* est renseigné (non nul avec une taille > 0).
        - **delete(id, callback)**: supprime les fichiers data (*[content.src]*) et metadata (*[content.id].meta.json*)
    - ***constructeur***: Le constructeur prend en paramètre un objet *ContentModel* et alimente l'objet en cours avec les données du paramètre.
- Lancer les tests unitaires via la commande `npm test`. La commande doit se terminer par un `=== FIN TESTS ===`.

Point de validation 4

15. Créer le router pour exposer les web services REST d'accès au contenu (*content.router.js*). De manière générale, les routeurs ne comportent pas de métier, ils se contentent d'appeler le controleur avec les bons paramètres. Ajouter ce router à *app.js* (comme pour le *default.route.js*).

> ### Tuto
>
> Pour avoir des WS RESTful, on utilise les verbes HTTP (GET, POST, PUT, DELETE) pour déterminer quel action doit être effectuée et les URI doivent permettre d'identifier directement sur quel ressource on doit effectuer l'action. Par exemple, une adresse possible pour accéder à un annuaire est: `http://MyService/users/1`. L'URI est donc de la forme `Protocol://ServiceName/ResourceType/ResourceID`.
>
> Le routeur peut s'articuler ainsi pour une ressource `users` :
>
> ```
> // user.route.js
> "use strict";
>
> var express = require("express");
> var router = express.Router();
> module.exports = router;
>
> var userController = require('./../controllers/user.controllers');
>
>  router.route('/users')
>    .get(userController.list)
>    .post(userController.token, userController.create);
>
>  router.route('/users/:userId')
>    .get(userController.read)
>    .put(userController.update)
>    .delete(userController.delete);
>
>  router.param('userId', function(req, res, next, id) {
>        req.userId = id;
>        next();
>  });
> ```

Dans notre cas, le routeur doit fonctionner ainsi:

- "/contents" + GET => retourne la liste des métadonnées de contenu de slides disponibles sur le serveur (*[content.id].meta.json*)
- "/contents" + POST => crée un nouveau contenu à partir du formulaire d'ajout de contenu ('file', 'title', 'type', 'src')
- "/contents/[content.id]" + GET => retourne le contenu avec l'ID correspondant.

Pour pouvoir faire un upload de fichiers sur le serveur (dans le cas d'un POST avec un contenu de type 'img'), on ajoute le module ***multer*** au projet.

```
// content.route.js
var multer = require("multer");
var express = require("express");
var router = express.Router();
module.exports = router;

var multerMiddleware = multer({ "dest": "/tmp/" });

router.post("/contents", multerMiddleware.single("file"), function(request, response) {
        console.log(request.file.path); // The full path to the uploaded file
        console.log(request.file.originalname); // Name of the file on the user's computer
        console.log(request.file.mimetype); // Mime type of the file
});
```

Point de validation 5

16. Créer le controleur (*content.controller.js*) pour faire le lien entre le routeur et le modèle. Le controleur va donc avoir les fonctions suivantes:

    - **list**: liste les fichiers de contenu du répertoire *[CONFIG.contentDirectory]* et retourne le résultat sous la forme un objet JSON au format "clé-valeur". La clé est l'ID du contenu (*ContentModel.id*) et la valeur est l'objet *ContentModel* au format JSON.

```
{
        _ContentModel[1].id_ : _ContentModel[1]_,
        _ContentModel[2].id_ : _ContentModel[2]_,
        ...
}
```

    - **create**: récupère les paramètres de requète pour créer un objet *ContentModel* et le stocker via la méthode statique du modèle.
    - **read**: Lit le contenu dont l'id est passé en paramètre et:
        - soit retourne l'url d'accés aux données (*ContentModel.src*) dans le cas où les données sont hébergés sur le serveur (c'est-à-dire dans le cas où le type de contenu est 'img')
        - soit effectue une redirection dans le cas où les données ne sont pas stockées sur le serveur.
        - soit retiourne les metadatas (le *ContentModel* au format JSON) si on passe en paramètre `json=true` dans l'URL.

Point de validation 6

17. Créer le serveur de websocket et gérer les évènements.

    - Installer la bibliothèque `socket.io` via npm (et l'ajouter au *package.json*). Cette librairie permet de créer des websockets avec NodeJS.
    - Créer un nouveau controleur (*io.controller.js*). Les évènements des websockets seront gérés dans ce controleur. Il expose une fonction *listen(httpServer)* et prend en paramètre une instance de serveur HTTP de NodeJS.

```
// app.js
var IOController = require("./app/controllers/io.controller.js");
[...]
IOController.listen(server);
```

    - Emettre l'évenement "*connection*" sur la nouvelle socket quand une nouvelle connexion est ouverte sur le serveur de websocket
    - Ecouter l'évènement "*data_comm*" et enregistrer la socket dans une map, avec en clé l'id du client (qui est fourni dans le message).
    - Ecouter l'évenement "*slidEvent*". Le message que nous fourni cet évènemment est un objet JSON qui contient la commande de la présentation et l'id de la présentation

```
{
        "CMD": [START | PAUSE | END | BEGIN | PREV | NEXT ],
```

```
        "PRES_ID": [pres.id] // Seulement pour la commande START
}
```

Pour les commandes START, END, BEGIN, PREV et NEXT, on récupère et on envoie les métadonnées du contenu de la slide que l'on doit diffuser à toutes les sockets connectées (penser à passer par *ContentModel* pour lire les métadonnées). En plus des données présentes dans le fichier de métadonnée de la slide, on ajoute un attribut "src" qui contient l'url pour accéder directement aux données.

```
// io.controller.js
[...]
ContentModel.read(..., function (err, content) {
        [...]
        content.src = "/contents/" + content.id;
        [...]
})
[...]
```

Point de validation 7

18. Gérer les évènements côté clients en utilisant un controleur dédié.
    - Récupérer la bibliothèque *socket.io* côté client, en insérant la balise HTML suivante:

```
<script type="text/javascript" src="/socket.io/socket.io.js"></script>
```

    - Initialiser la connexion au serveur de websocket dans le controleur et récupérer la socket sur laquelle on doit gérer les évènements.

```
var socket = io.connect();
```

    - Ecouter l'évènement *connection*. Lorsqu'il est détecté, émettre l'évènement *data_com* avec comme message l'id de la socket. Cet étape doit être faite sur les pages */admin* et */watch*.
    - Côté */admin*, émettre un évènement *slidEvent* avec la commande associée (START, END, PAUSE, NEXT...) en fonction des actions sur les boutons de commande de la présentation. Pour la commande START, ne pas oublier d'ajouter le *PRES_ID* dans le message JSON.
    - Côté */watch*, écouter l'évènement *currentSlidEvent* et mettre à jour l'affichage en utilisant les données reçus.

Point de validation 8

19. Quelques idées pour aller plus loin...
    - Créer un webservice pour créer les UUID systématiquement côté serveur et supprimer la fonction *generateUUID()* côté client.
    - Créer un modèle de données pour les présentations et passer les webservices en RESTful.
    - Permettre la diffusion de plusieurs présentation en même temps. Côté admin, en listant les présentations disponibles et en permettant de sélectionner celle que l'on veut diffuser. Côté serveur, en créant des URL de */watch* différentes en fonction des présentations diffusées (par exemple, */watch/[PRES_ID]*). On pourra utiliser les "rooms" de socket.io pour compartimenter les présentations et pouvoir faire des broadcasts par "room".