1. Créer l'arborescence

```
 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 = gestionnaire packages → permet d'installer packages

```
npm init
```

3. Installer express comme dépendance et l'ajouter au fichier package.json
Express = framework permettant d'aller plus vite dans la création d'application avec Node.js
Package.json → gestion de dépendances

```
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" :  "echo \"Error: no test specified\" && exit 1",  // Affiche une
erreur par défaut
    "start" :  "node app.js"  // npm start lance app.js
},
"private" :   true ,  // Rendre l'application privée
[…]
```

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

```
npm start // lance app.js
```

6. Initialiser express

```
// app.js
var  express  =  require("express");
[...]
var  app  =  express(); // Pour que l'on puisse définir les routes après
```

require(« framework ») → on utilise le framework en question
→ retourne un objet

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
}
```

On utilisera le port 1337 pour les entrées/sorties.

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"); // on utilise http qui est un protocole de
communication client-serveur
var  CONFIG  =  require("./config.json");
[...]
// init server
var  server  =  http.createServer(app);  // on crée le serveur en lui passant
app (=express())
server.listen(CONFIG.port);  // On mets le serveur sur écoute du port de
configuration 1337
```

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);  // pour pouvoir la parser plus
tard
```

process.env = objet qui contient l'environnement de l'utilisateur (path, mdp…)

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

```
var  CONFIG  =  JSON.parse(process.env.CONFIG); // on parse pour accéder à
l'information
```

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"); // utilise le framework express()
var  router  =  express.Router();  // créer un router
module.exports  =  router; // objet retourné par le résultat d'un require →
router sera retourné après un require
// 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");  // on définit
route par défaut
[...]
app.use(defaultRoute);  // imposer/utiliser route par défaut
```

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

```
// #2
app.get("/",  function (request, response) {  // on utilise la méthode get
    response.send("It works !");  // On retourne la réponse
});
// #3
app.use( function (request, response, cb) { // on invoque un middleware → on
ajoute la fonction qui gère les réponses en utilisant les paramètres suivants :
request, response, cb
    response.send("It works !"); // On retourne la réponse
    cb();  // appel de la fonction de callback
});
```

.use = methode pour configurer le middleware utilisé par les routes de l'objet Express HTTP server
Middleware = logiciel qui permet à plusieurs applications d'échanger

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"); // On va utiliser l'objet path, qui gère les
chemins des fichiers qui nous intéressent.
[...]
app.use("/admin", express. static (path.join(__dirname, "public/admin")));
app.use("/watch", express. static (path.join(__dirname, "public/watch")));
```

__dirname : nom du répertoire
express.static : rajoute une route statique (route configurée inchangeable)
Quand on se logue, soit on est admin soit on est watch (utilisateur).

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 des slides. Les métadonnées seront stockées en JSON.
Métadonnées → données stockées dans les slides (description, source...)
- **presentationDirectory**: chemin d'accès absolu vers le répertoire presentation_content. Ce répertoire contiendra les métadonnées de présentation au format JSON.
Métadonnées → données stockées dans les présentations (tableau de slides, id, titre…)

```
{
      "port": 1337,
      "contentDirectory": "./uploads",
      "presentationDirectory": "./presentation_content"
}
```

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 "clévaleur". La **clé est l'ID** de la présentation **et la valeur est l'objet retourné** par le parseur JSON.

```
// 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.

14. **Créer le modele de donnée pour les slides.**
Créer le fichier **slid.model.js** dans app/models/. Ce fichier va contenir l**a "classe" SlidModel** avec la définition suivante:

- **attributs**
  - **type**: public
  - **id**: public
  - **title**: public
  - **fileName**: public le nom du fichier stocké dans [CONFIG.contentDirectory]. Il correspond a l'id de la slide + 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 (utilisables partout + ne peut pas être redéfinie)

- **create(slid, callback)**: Prend un objet slidModel en paramètre, stocke le contenu de [slid.data] dans le fichier [slid.fileName] et stocke les metadonnées dans un fichier [slidModel.id].meta.json dans le répertoire [CONFIG.contentDirectory].
- **read(id, callback)**: Prend un id en paramètre et retourne l'objet slidModel lu depuis le fichier [slid.id].meta.json
- **update(slid, callback)**: Prend l'id d'un SlidModel en paramètre et met à jour le fichier de metadata ([slid.id].meta.json) et le fichier [slid.fileName] si [slid.data] est renseigné (non nul avec une taille > 0).
- **delete(id, callback)**: supprime les fichiers data ([slid.src]) et metadata ([slid.id].meta.json)

- **constructeur**: Le constructeur prend en paramètre un objet SlidModel et alimente l'objet en cours avec les données du paramètre.

15. **Créer le router** pour exposer les web services d'accès aux slides (slid.router.js). Ces web services doivent être RESTful. 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) :

```
var slidRoute = require("./app/routes/slid.route.js");
app.use(slidRoute);
```

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.
REST = style d'architecture pour les systèmes hypermédia istribués

AVANTAGES
L'application est plus **simple à entretenir**, car elle **désolidarise la partie client de la partie serveur**. La nature hypermédia de l'application permet d'accéder aux états suivants de l'application par inspection de la ressource courante.

•L'**absence de gestion d'état du client sur le serveur** conduit à une **plus grande indépendance entre le client et le serveur**. Elle permet également de ne pas avoir à maintenir une connexion permanente entre le client et le serveur. Le serveur peut ainsi répondre à d'autres requêtes venant d'autres clients sans saturer l'ensemble de ses ports de communication. Cela devient essentiel dans un système distribué.

•L'absence de gestion d'état du client sur le serveur permet également une **répartition des requêtes sur plusieurs serveurs** : une session client n'est pas à maintenir sur un serveur en particulier (via une *sticky session* d'un *loadbalancer*), ou à propager sur tous les serveurs (avec des problématiques de fraîcheur de session). Cela permet aussi une **meilleure évolutivité** et **tolérance aux pannes** (un serveur peut être ajouté facilement pour augmenter la capacité de traitement, ou pour en remplacer un autre).

•Dans un contexte Web :
•l'**utilisation du protocole HTTP** permet de tirer parti de son enveloppe et ses en-têtes ;
•l'**utilisation d'URI** comme représentant d'une ressource permet d'avoir un système universel d'identification des éléments de l'application ;

•la **nature auto-descriptive du message** permet la **mise en place de serveurs cache** : les informations nécessaires sur la fraîcheur, la péremption de la ressource sont contenues dans le message lui-même.

INCONVENIENTS

Le principal inconvénient de REST est la **nécessité pour le client de conserver localement toutes les données** nécessaires au bon déroulement d'une requête, ce qui induit une **consommation en bande passante réseau plus grande**. Notamment dans l'univers des appareils mobiles, chaque aller-retour de connexion est consommateur d'électricité. La latence de la connexion rend également l'interaction moins fluide.

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:

```
// user.route.js
"use strict"; // Le code javascript ne sera pas exécuté de la même manière en
fonction des web browsers
var  express  =  require("express");
var  router  =  express.Router(); // gère les routes
module.exports  =  router;
var  user  =  require('./../controllers/user.controllers'); // On récupère la
classe controllers

router.route('/users') // pour ajouter possibilités aux valeurs renvoyées
  .get(user.list)  // retourne liste des slides
  .post(user.token,user.create); // Crée la slide

router.route('/users/:userId')
  .get(user.read) // lit l'objet slidModel
  .put(user.update) // Modifie slidModel
  . delete (user. Delete ); // Supprime les fichiers data et metadata

router.param('userId',  function (req, res, next, id) {
    req.userId  =  id;
    next(); // On passe à une autre slide
});
```

Dans notre cas, le routeur doit fonctionner ainsi:

"/slids" + GET => retourne la liste des slides
"/slids" + POST => crée la slide dont les informations sont en paramètres de requête
"/slids/[slidId]" + GET => retourne la slide avec l'ID correspondant.

Les autres verbes (DELETE, PUT) sont facultatifs pour le fonctionnement de l'application.
Pour **faciliter l'upload de fichiers sur le serveur**, on utilise le module **multer**.

```
// slid.route.js
var  multer  =  require("multer"); // faciliter l'upload de fichiers sur serveur
var  SlidController  =  require("./../controllers/slid.controller.js");
var  express  =  require("express");
var  router  =  express.Router();
module.exports  =  router;
var  multerMiddleware  =  multer({ "dest" :  "/tmp/" }); // On indique le chemin
```

```
de dépôt de fichiers uploadés

// Creation de la slide
router.post("/slids", 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
});
```

REQUETE COTE CLIENT !!

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

- **list**: liste toutes les slides du répertoires [CONFIG.contentDirectory] et retourne le résultat sous la forme un objet JSON au format “clévaleur”. La clé est l’ID de la slide et la valeur est l’objet SlidModel au format JSON.

```
SlidController.list = function (request, response, callback) {
      SlidModel.list(response, function (err, data) {
            if (err) {
                  response.send(err);
                  return;
            }
            response.send(data);
      });
}
```

- **create**: récupère les paramètres de requête pour créer un objet SlidModel et le stocker via la méthode statique du modèle.

```
SlidController.create = function (request, response) {
      var content = "";
      request.on("data", function (data) {
            content += data.toString();
      });
      request.on("end", function () {
            var json_string = content;
            SlidModel.create(new SlidModel(json_string));
      });
}
```

.on → lie un évènement à un objet

- **read**: Lit le slide dont l’id est passé en paramètre et retourne:
  - soit la slide (le contenu du fichier de données)
  - soit le SlidModel au format JSON si on passe en paramètre json=true

```
SlidController.read = function (id, callback, json) {
      var myPath = path.join(CONFIG.contentDirectory, id + ".meta.json");
      fs.stat(myPath, function (err, data) {
            if (err) {
                  callback("no file in " + myPath);
                  return;
            }
            fs.readFile(myPath, "utf-8", function (err, data) {
                  if (err) {
                        callback(err);
                  } else {
                        if (json == true) {
                              callback(null, JSON.parse(data.toString()));
```

```
                        } else {
                              callback(null, new
SlidModel(JSON.parse(data.toString())));
                        }
                  }
            });
      });
}
```

fs = objet contenant plusieurs outils

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

```
//io.controller.js²
"use strict";
var io = require('socket.io');
var SlidModel = require("../models/slid.model.js");

var mapSocket = {};
var controller = function () {};

controller.listen = function (server) {
      var ioServer = io.listen(server);
      ioServer.set('log level', 1);

      ioServer.on('connection', function (socket) {
            /**
             * Log de connexion et de déconnexion des utilisateurs
             */
            socket.emit('connection', 'I\'m ready');
            console.log('a user connected');
            socket.on('slidEvent', function (data) {
                  console.log("slidEvent received " + data);
                  var obj = JSON.parse(data);
                  var id = obj.PRES_ID;
                  SlidModel.read(id, function (err, data) {
                        if (err) {
                              socket.emit('connection', JSON.stringify(err));
                        } else {
                              socket.emit('connection', JSON.stringify(data));
                        }
                  });
            });

            socket.on('data_comm', function (err, data,arg0,arg1,arg2) {
                  var json = "";
                  console.log(arg0);
                  console.log(arg1);
                  console.log(arg2);

                  try{
                        json = JSON.parse(data);
                  }catch(e){}
                  if(json == "")
                  {return;}
                  mapSocket[data.id] = "";
            });
      })
};
module.exports = controller;
```

17.1/ Installer la bibliothèque socket.io via npm (et l'ajouter au package.json). Cette librairie permet de créer des websockets avec NodeJS.

17.2/ Créer un nouveau controleur (io.controller.js). Les événements 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);
```

17.3/ Émettre l'événement "connection" sur la nouvelle socket quand une nouvelle connexion est ouverte sur le serveur de websocket

17.4/ Écouter l'événement "data_comm" et enregistrer la socket dans une map, avec en clé l'id du client (qui est fourni dans le message).

17.5/ Écouter l'événement "slidEvent". Le message que nous fourni cet événement 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
}
```

17.6/ Pour les commandes START, END, BEGIN, PREV et NEXT, on récupère et on envoie les métadonnées de la slide que l'on doit diffuser à toutes les sockets connectées (penser à passer par SlidModel 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 d'accès aux données de la slide.

```
// io.controller.js
[...]
SlidModel.read(..., function (err, slid) {
    [...]
    slid.src = "/slid/" + slid.id;
    [...]
})
[…]
```

18. Gérer les événements côté clients en utilisant un controleur dédié.

18.1/ Récupérer la bibliothèque socket.io côté client, en insérant la balise HTML suivante:

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

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

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

18.3/ Écouter l'événement connection. Lorsqu'il est détecté, émettre l'événement data_com avec comme message l'id de la socket. Cet étape doit être faite sur les pages /admin et /watch.

18.4/ Côté /admin, émettre un événement 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.

18.5/ Côté /watch, écouter l’événement currentSlidEvent et mettre à jour l’affichage en utilisant les données reçus.