



**POLYTECHNIQUE  
MONTREAL**

UNIVERSITÉ  
D'INGÉNIERIE

# LOG2440 – Méthod. de dévelop. et conc. d'applic. Web

## Travail pratique 4

Chargés de laboratoire:

Charles De Lafontaine

Antoine Poellhuber

Justin Lachapelle

Hiver 2023

Département de génie informatique et génie logiciel

# 1 Objectifs

Le but de ce travail pratique est de vous familiariser avec le développement de serveurs Web statiques et dynamiques et la communication réseau à travers le protocole HTTP. Pour ce faire, vous allez utiliser l'environnement `NodeJS` ainsi que la librairie `Express` pour créer votre propre serveur dynamique. Vous allez vous familiariser avec le module `FS` (*File System*) afin de gérer des fichiers côté serveur. Vous allez également utiliser l'API `Fetch` pour construire les requêtes HTTP envoyées par votre site web.

Plus particulièrement, à partir de votre site web, vous aurez à faire des requêtes HTTP dans le but de communiquer avec un serveur. Ce dernier vous transmettra alors les informations souhaitées et vous fournira les données demandées.

Contrairement aux travaux pratiques précédants, un deuxième serveur web s'ajoutera au serveur de contenu statique (HTML, CSS, JS, images, etc.) *lite-server* et sera un serveur dynamique qui est responsable de répondre aux requêtes HTTP qui lui sont envoyées.

## 2 Introduction

Lors des trois derniers travaux pratiques, vous avez mis en place un site web à l'aide de fichiers HTML, CSS et JavaScript qui étaient tous récupérés grâce à un serveur HTTP statique. Votre site web était également capable de charger dynamiquement des playlists et des chansons et de les sauvegarder localement sur l'entrepôt local *LocalStorage*.

Toutes ses étapes se déroulaient uniquement du côté client. Ceci est peu efficace, car les informations ne sont pas partagées entre plusieurs utilisateurs ou plusieurs navigateurs.

Habituellement, le contenu du client tel que les fichiers HTML, CSS, JS, les images et ainsi de suite sont hébergés sur un serveur web. Lorsqu'un client accède à une URL, le navigateur fait une requête HTTP avec la méthode `GET` au serveur pour récupérer les fichiers correspondants. Le client peut également envoyer d'autres types de requêtes qui sont interprétées de manière différente par le serveur résultat en une action quelconque. Par exemple, une requête `POST` est souvent utilisée pour envoyer de l'information qui sera traitée et/ou sauvegardée par le serveur tandis qu'une requête `DELETE` se comporte comme une demande de suppression d'une ressource quelconque spécifiée dans l'URI ciblé par la requête.

## 3 Configuration du projet

### 3.1 Dépendances manquantes et librairie Nodemon

La configuration initiale fournie pour votre projet `server` manque une dépendance importante : la librairie `Express` dans la liste des dépendances du fichier `package.json`. Vous devez utiliser l'outil `npm` et ses commandes pour installer et ajouter `Express` à vos dépendances de projet avant de pouvoir lancer votre serveur. N'oubliez pas d'ajouter les fichiers `package.json` et `package-lock.json` modifiés à votre entrepôt `Git`.

Pour vous aider dans votre développement, vous utiliserez l'utilitaire `nodemon`. `Nodemon` permet d'exécuter du code JS, similairement à la commande `node`, mais surveille des changements dans le code source et relance le serveur automatiquement (*hot-reload*).

Pour lancer le serveur dynamique, vous pouvez utiliser la commande `npm start` dans votre terminal avec : `npm start`. Le serveur est accessible à l'adresse : `"http://localhost:5020"`.

**Note :** le nom `localhost` est un raccourci pour l'adresse IP `127.0.0.1`. Le lancement du serveur `lite-server` affichera les adresses auxquelles le serveur est accessible.

### 3.2 Tests automatisés

Plusieurs tests vous sont déjà fournis dans le répertoire `server/tests` et vous permettront de valider votre travail et les différentes requêtes au serveur. Certains tests vérifient le bon comportement des réponses HTTP de votre serveur en simulant des requêtes à l'aide de la librairie `Supertest`. D'autres se concentrent sur la logique de gestion des ressources sur le serveur. Dans les deux cas, la librairie `Jest` est utilisée.

Pour lancer les suites de tests, vous n'avez qu'à aller à la racine du répertoire `server` avec un terminal et de lancer la commande `npm test`. Les tests d'API lanceront votre serveur et feront plusieurs requêtes avant de le fermer à la fin de la suite. Une commande pour la couverture de code (`npm run coverage`) est également à votre disposition.

Certains tests pour le code de votre site web vous sont également fournis dans `site-web/tests`. Notez que dû à une limitation de la librairie `Jest`, la méthode `loadForEdit` n'est pas couverte, mais son code vous est fourni et vous pouvez assumer qu'il est fonctionnel.

Il est fortement recommandé de regarder les tests fournis et leur description avant d'écrire votre code. Vous pouvez ajouter vos propres tests si vous le considérez pertinent.

## 4 Travail à réaliser

Premièrement, votre serveur dynamique s'occupera de la gestion des playlists et chansons et le client devra alors faire des requêtes `GET` pour récupérer les informations ou effectuer une recherche par mot clé. La création d'une nouvelle playlist se fait avec une requête `POST` dont le corps contient la nouvelle playlist. La modification de l'état "aimé" d'une chanson aura également lieu sur le serveur dynamique suite à une requête `PATCH`.

Deuxièmement, une nouvelle fonctionnalité consiste à permettre à l'utilisateur du site de modifier ou supprimer une playlist existante. La playlist modifiée est transmise à travers une requête `PUT` et sauvegardée dans le fichier *playlists.json* dans *server/data* qui contient toutes les playlists déjà transmises. La suppression utilise une requête `DELETE`.

### 4.1 Éléments à réaliser

La première étape consiste à implémenter les fonctionnalités nécessaires pour rendre le client et le serveur fonctionnels. Nous vous recommandons d'implémenter le code côté client et côté serveur pour chaque fonctionnalité (Playlists, Chansons, Recherche) une à la fois. Notez que, par défaut, les routes HTTP sur votre serveur sont configurés avec la méthode générique `use`. Vous devez changer ceci pour utiliser les bons verbes HTTP.

### 4.2 Gestion des playlists sur le serveur

Contrairement aux TPs 2 et 3, vous devez désormais gérer l'état des playlists sur votre serveur et non seulement à travers le *LocalStorage*. Vous aurez donc à implémenter la gestion des playlists sauvegardées dans le fichier *playlists.json* dans le répertoire *server/data*.

Vous aurez à compléter certaines méthodes de la classe `PlaylistManager`, notamment `getPlaylistById`, `updatePlaylist` et `deletePlaylist`. Lisez bien les en-têtes de fonctions fournies dans le code pour mieux comprendre les entrées, sorties et fonctionnement nécessaires pour chaque méthode. Les méthodes `addPlaylist` et `savePlaylistThumbnail` vous sont fournies pour vous aider. Notez que l'id d'une playlist est aussi géré par le serveur.

La classe `PlaylistManager` est utilisée par le *Router* défini dans le fichier *playlists.js*. Ce routeur implémente la logique de gestion pour les différentes requêtes en lien avec la gestion des playlists. Le routeur est accessible à travers tout chemin relatif qui commence par `/api/playlists` tel que défini dans la configuration de l'application Express dans *server.js*.

Vous aurez à compléter les gestionnaires de requêtes définies dans les sous-sections suivantes. Notez que le chemin indiqué ici est relatif du chemin du routeur `/api/playlists`.

#### 4.2.1 GET /

Retourne toutes les playlists présentes dans le fichier `playlists.json` dans un objet JSON. La gestion de cette requête est déjà implémentée pour vous.

La réponse HTTP doit contenir un code de retour pertinent.

#### 4.2.2 GET /:id

Retourne la playlist spécifiée par l'attribut `id` dans le chemin dans un objet JSON.

La réponse HTTP doit contenir un code de retour pertinent dans le cas d'un attribut `id` valide et dans le cas d'un attribut invalide.

#### 4.2.3 POST /

Vous devez vérifier la présence d'un corps sur la requête et, si présent, sauvegarder le contenu du corps comme une nouvelle playlist dans le fichier `playlists.json`.

La réponse HTTP doit contenir un code de retour pertinent dans le cas d'un corps invalide, d'une sauvegarde réussie ou d'une erreur quelconque sur le serveur.

#### 4.2.4 PUT /:id

Vous devez vérifier la présence d'un corps sur la requête et, si présent, remplacer la playlist ayant le même `id` par le contenu de la requête et mettre à jour le fichier `playlists.json`.

La réponse HTTP doit contenir un code de retour pertinent dans le cas d'un corps invalide, d'une sauvegarde réussie ou d'une erreur quelconque sur le serveur.

#### 4.2.5 DELETE /:id

Supprime la playlist spécifiée par l'attribut `id` et mets à jour le fichier `playlists.json`.

Exemple : `DELETE /12` supprime la playlist ayant l'`id` "12" si présente dans le fichier.

La réponse HTTP doit contenir un code de retour pertinent dans le cas d'une suppression réussie ou non et dans le cas d'une erreur quelconque sur le serveur.

## 4.3 Gestion des chansons sur le serveur

Contrairement aux TPs 2 et 3, vous devez désormais gérer l'état des chansons sur votre serveur et non seulement à travers le *LocalStorage*. Vous aurez donc à implémenter la gestion des chansons sauvegardées dans le fichier `songs.json` dans le répertoire `server/data`.

Vous aurez à compléter certaines méthodes de la classe `SongsManager` : `getSongById` et `updateSongLike`. Lisez bien les en-têtes de fonctions fournies dans le code pour mieux comprendre les entrées, sorties et fonctionnement nécessaires pour chaque méthode. La méthode `getAllSongs` vous est fournie pour vous aider avec votre travail.

La classe `SongsManager` est utilisée par le *Router* défini dans le fichier `songs.js`. Ce routeur implémente la logique de gestion pour les différentes requêtes en lien avec la gestion des playlists. Le routeur est accessible à travers tout chemin relatif qui commence par `/api/songs`. Vous devez configurer l'ajout du routeur à votre application Express dans `server.js`.

Vous aurez à compléter les gestionnaires de requêtes définies dans les sous-sections suivantes. Notez que le chemin indiqué ici est relatif du chemin du routeur `/api/songs`.

### 4.3.1 GET /

Retourne toutes les chansons présentes dans le fichier `songs.json` dans un objet JSON. La gestion de cette requête est déjà implémentée pour vous.

La réponse HTTP doit contenir un code de retour pertinent.

### 4.3.2 GET /:id

Retourne la chanson spécifiée par l'attribut `id` dans le chemin dans un objet JSON.

La réponse HTTP doit contenir un code de retour pertinent dans le cas d'un attribut `id` valide et dans le cas d'un attribut invalide.

### 4.3.3 GET /player/:id

Retourne le fichier de chanson spécifiée par l'attribut `id` dans le chemin dans un objet JSON. La gestion de cette requête est déjà implémentée pour vous. Le code utilise la notion de *Stream* pour lire les fichiers sur disque de manière asynchrone sans bloquer le fil d'exécution.

La réponse HTTP doit contenir un code de retour pertinent dans le cas d'un attribut `id` valide et dans le cas d'un attribut invalide.

#### 4.3.4 PATCH / :id/like

Modifie l'état "aimé" de la chanson spécifiée par l'attribut `id`. Le serveur doit mettre à jour le fichier `songs.json`. La réponse HTTP contient le nouvel état "aimé" de la chanson dans un objet JSON utilisant le format suivant : `{liked : nouveauEtat}`.

La réponse HTTP doit contenir un code de retour pertinent dans le cas d'un attribut `id` valide et dans le cas d'un attribut invalide.

### 4.4 Gestion de la recherche par mot clé

Pour ce travail pratique, vous aurez à gérer la recherche par mot clé dans les playlists et chansons. Cette fonctionnalité est pareille à celle qui vous a été demandée au TP3, mais, cette fois-ci, la recherche sera effectué par le serveur dynamique qui retourne les éléments trouvés dans une réponse HTTP.

La classe `SearchBarManager` qui contient la logique de recherche est déjà implémentée pour vous. Vous n'aurez qu'à l'utiliser dans l'implémentation du Router dans `search_bar.js`. Le routeur est accessible à travers tout chemin relatif qui commence par `/api/search`. Vous devez configurer l'ajout du routeur à votre application Express dans `server.js`.

#### 4.4.1 GET / ?search\_query=X&exact=Y

Effectue une recherche à l'aide du mot clé défini par **X** dans la *query* de la requête et retourne toutes les playlists et/ou chansons ayant au moins un élément dans leur description qui correspond au mot clé de la recherche.

Cette requête possède également un paramètre de *query : exact*. Si ce paramètre a la valeur `true`, la recherche se fait en tentant compte des majuscules et minuscules dans `search_query`, sinon la recherche se fait en cherchant sans en tenir compte.

Exemple basé sur les playlists et chansons par défaut :

- GET `/api/search?search_query=Co&exact=false` retourne les playlists *Playlist de Rock*, *Discover Weekly* et la chanson *Bounce*.
- GET `/api/search?search_query=Co&exact=true` retourne que la chanson *Bounce*.

Référez-vous aux figures 1 et 2 dans l'Annexe pour le résultat attendu.

La réponse HTTP doit contenir un code de retour pertinent.

## 4.5 Gestion des données sur le site web

Puisque la gestion des données est maintenant faite par le serveur, le site web doit communiquer avec en utilisant le protocole HTTP pour obtenir cette information. Le site web doit pouvoir effectuer les mêmes opérations que les TPs 2 et 3, mais sans utiliser *LocalStorage*.

La classe `StorageManager` a été remplacée par la classe `HTTPManager` qui est responsable de la communication avec le serveur. Vous devez compléter plusieurs fonctions de cette classe afin d'implémenter la communication avec votre serveur.

Les fonctions à compléter contiennent le décorateur `TODO` dans leur en-tête de commentaires. Lisez bien les en-têtes de fonctions fournies dans le code pour mieux comprendre les entrées, sorties et fonctionnement nécessaires pour chaque méthode. Certaines méthodes vous sont déjà fournies, notamment la réinitialisation des recettes sur le serveur.

Pour vous aider à démarrer, un objet `HTTPInterface` vous est fourni dans le fichier `http_manager.js`. Cet objet implémente plusieurs fonctions utilitaires qui utilisent l'API `Fetch` pour envoyer des requêtes HTTP. Certaines méthodes qui envoient des paramètres contiennent l'en-tête *Content-Type : application/json* qui n'est pas toujours nécessaire. Vous devez enlever cet en-tête pour les requêtes qui n'en ont pas besoin.

### 4.5.1 Ma bibliothèque

`HTTPManager` est utilisée par le code de `Library` pour charger les playlists et les chansons à afficher. Vous devez implémenter la fonction `search` qui permet d'effectuer la recherche par mot clé à travers une requête HTTP. La fonction doit mettre à jour le contenu de la bibliothèque après avoir reçu la réponse du serveur.

### 4.5.2 Page de Playlist

`HTTPManager` est utilisée par le code de `PlayListManager` pour charger une playlist spécifique et ses chansons. Si votre implémentation de `HTTPManager` est correcte, cette page devrait fonctionner correctement.

Le bouton de crayon à droite redirige maintenant vers `create_playlist.html?id=X` avec `X` étant l'identifiant de la playlist. Lorsque la page est chargée de telle manière, elle permet de modifier ou supprimer la playlist ciblée, tel qu'expliqué à la section 4.5.3.



### 4.5.3 Créer une Playlist

Contrairement aux TPs précédents, le formulaire de création de playlist permet désormais d’aussi modifier ou supprimer une playlist existante. Si la page est accédée à travers le menu de navigation, le formulaire habituel de création d’une nouvelle playlist est affiché.

Si la page est accédée avec un `id` de playlist dans son URI, le formulaire est prérempli avec l’information de la playlist. Le bouton *Ajouter la playlist* est remplacé par *Modifier la playlist* et un bouton de suppression est visible. Référez-vous à la figure 3 de l’Annexe.

La fonction `loadForEdit` vous est fournie et si l’implémentation de `getPlaylistById` est correcte, le chargement devrait être fonctionnel.

Vous devez compléter la fonction `createPlaylist` qui doit envoyer la bonne requête HTTP en fonction de si vous voulez créer une nouvelle playlist ou en modifier une déjà existante. Notez que comme l’identifiant d’une nouvelle playlist est généré par le serveur, vous devez vous assurer que la modification d’une playlist ne modifie pas son attribut `id`.

Vous devez compléter la fonction `deletePlaylist` qui envoie une requête de suppression d’une playlist spécifique. Suite à la réponse du serveur, l’utilisateur doit être redirigé vers la page principale et la playlist supprimée ne doit pas faire partie des playlists affichées.

## Conseils pour la réalisation du travail pratique

---

1. Séparez bien la logique des requêtes de la logique de gestion des fichiers.
  2. Lisez bien les tests fournis pour vous aider avec les fonctionnalités demandées.
  3. Consultez les exemples de serveurs NodeJS/Express sur le [GitHub du cours](#).
  4. Exécutez les tests fournis souvent afin de valider votre code.
  5. Respectez la bonne sémantique pour les codes de retour.
  6. Assurez-vous de gérer les cas d'erreur pour les différentes requêtes HTTP.
  7. Respectez la convention de codage établie par *ESLint*. Utilisez la commande `npm run lint` pour valider cet aspect.
- 

## 5 Remise

Voici les consignes à suivre pour la remise de ce travail pratique :

1. Le nom de votre entrepôt Git doit avoir le nom suivant : **tp4\_matricule1\_matricule2** avec les matricules des 2 membres de l'équipe.
2. Vous devez remettre votre code (*push*) sur la branche **master** de votre dépôt git. (pénalité de 5% si non respecté)
3. Le travail pratique doit être remis avant **23h55, le 28 mars**.

**Aucun retard** ne sera accepté pour la remise. En cas de retard, la note sera de **0**.

Le navigateur web **Google Chrome** sera utilisé pour tester votre site web. Vos serveurs doivent être déployables avec la commande `npm start`.

## 6 Évaluation

Vous serez évalués sur le respect des exigences fonctionnelles de l'énoncé, ainsi que sur la qualité de votre code JS et sa structure. Le barème de correction est le suivant :

Exigences	Points
Implémentation du serveur web et site web	
Implémentation du <i>Routeur</i> <code>search_bar.js</code>	1
Implémentation du <i>Routeur</i> <code>playlists.js</code>	3
Implémentation du <i>Routeur</i> <code>songs.js</code>	2
Implémentation de <code>PlaylistsManager</code>	3
Implémentation de <code>SongsManager</code>	2
Implémentation de <code>HTTPManager</code> et son utilisation	5
Qualité du code	
Structure du code	2
Qualité et clarté du code	2
Total	20

L'évaluation se fera à partir de la page d'accueil du site, soit `index.html`. À partir de cette page, le correcteur devrait être capable de consulter toutes les autres pages de votre site web et interagir avec les différents éléments du site.

L'évaluations des tests se fera à partir des tests unitaires dans le projet `server`. Tous les tests fournis doivent obligatoirement passer. Tous les tests unitaires additionnels écrits par vous (si lieu) doivent obligatoirement passer.


Le code doit respecter les règles établies par *ESLint*. La commande `lint` ne doit pas indiquer des erreurs dans la console après son exécution.

Ce travail pratique a une pondération de **7%** sur la note du cours.

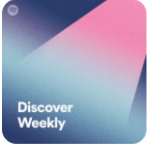
# Annexe

☐ Exact

**Mes Playlists**



Playlist de Rock  
Playlist avec une descr...



Discover Weekly  
Playlist avec beaucoup ...

**Mes Chansons**

Bounce

Electronic

Coma-Media

FIGURE 1 – Recherche de chaîne **Co** sans match exact

☒ Exact

**Mes Playlists**

**Mes Chansons**

Bounce

Electronic

Coma-Media

FIGURE 2 – Recherche de chaîne **Co** avec match exact

Ma Bibliothèque

+ Créer Playlist

À Propos

Informations générales

Nom:

Playlist de Rock

Description:

Playlist avec une descriptior

Image:

Choose File

assets/img/rock.jpg

Chansons

+

#1

Whip

#2

Bounce

—

#3

—

Modifier la playlist




FIGURE 3 – Vue pour la modification de playlist existante