EPFL CS212 : ImgStore – Système de fichiers orienté images — 11 : webserver 1 : API et `list`

E. Bugnion & J.-C. Chappelier EPFL

Rev. 2021.05.21 / 2

Table des matières

Introduction 1
Matériel fourni 2
Travail à faire 2
Installer et utiliser `libjson` 2
Modifier `do_list()` 3
Compiler et utiliser libmongoose 4
Développer le serveur web 4
Tests 6
Rendu 7

Introduction

L'utilitaire de ligne de commande `imgStoreMgr` est maintenant opérationnel (nous le compléterons encore d'un « *garbage collector* » en semaine 13). Nous pouvons alors, cette semaine, commencer le serveur web, lequel nous occupera encore la semaine prochaine.

L'objectif principal de cette semaine ci est de mettre en place une première version, de façon à offrir l'équivalent de la commande `list` aux clients de ce serveur. Lorsqu'il sera complet (semaine prochaine), le serveur web implémentera les mêmes fonctionnalités que l'utilitaire de commandes `imgStoreMgr`, excepté la commande `create` qui restera spécifique à la ligne de commande.

La différence essentielle avec l'utilitaire de commandes est que le serveur tournera en continu : il interagira via une connexion réseau (un « *socket* ») avec un programme client (lequel tournera à l'intérieur d'un navigateur). Nous vous fournissons le code du client, écrit en Javascript (comme l'essentiel des applications web actuelles).

Pour l'implémentation du serveur web, nous nous baserons sur la bibliothèque `libmongoose` que vous avez déjà pratiquée en semaine 2. Pour rappel, il s'agit

d'une bibliothèque très simple à utiliser qui implémente le protocole HTTP (par exemple, `apache` ou `ngnx` sont des serveurs web plus robustes et avec plus de fonctionnalités, mais également beaucoup plus difficiles à intégrer).

`libmongoose` s'occupe en particulier d'écouter sur un port choisi (8000 pour nous), d'accepter de nouvelles connections, et d'implémenter le protocole HTTP. Votre projet devra implémenter les commandes spécifiques liées a `imgStore` dans le cadre d'intégration sur le protocole HTTP.

Matériel fourni

Même si l'on passe à un nouvel exécutable, vous construirez également cette phase du projet sur la base des semaines précédentes. En plus, vous aurez à utiliser `libmongoose/` et `tests/data/index.html` qui ont déjà été fournis au départ du projet (mais nous venons de mettre à jour `index.html` ; faites un `git pull` pour le récupérer).

Vous aurez de plus à utiliser la bibliothèque `libjson` qui permet de parser et d'écrire des commandes en format « JSON ». Il s'agit du format standard communément utilisé par les applications Javascript, facile à lire (et beaucoup plus simple à traiter et interpréter que XML). `libjson` est une bibliothèque standard à installer dans votre système (comme mentionné en semaine 1).

Travail à faire

Installer et utiliser `libjson`

Installation Si ce n'est pas déjà fait (évoqué en semaine 1), commencez par installer la `libjson` sur votre machine de développement.

- Sur Linux :
  ```
  sudo apt install libjson-c-dev
  ```
  Pour vérifier que vous avez la bonne version, tapez `apt-cache show libjson-c-dev` et vérifier que la `Homepage` est bien https://github.com/json-c/json-c/wiki (il peut y avoir différentes variantes de cette bibliothèque).
- Pour les utilisateurs OS X, la bibliothèque est dans `brew` :

```
brew install json-c
```

Utilisation Pour utiliser la bibliothèque :

- l'interface est définie dans `<json-c/json.h>` — à regarder ; rajouter le `include` dans tous les fichiers `.c` qui ont besoin de cette fonctionnalité ;
- il est possible que vous deviez rajouter `-ljson-c` aux `LDLIBS` de votre `Makefile` (ou, sur Mac, `pkg-config --cflags json-c`, comme vous l'avez fait pour VIPS, par exemple).

La documentation de l'API se trouve ici : http://json-c.github.io/json-c/json-c-0.15/doc/html/

Les fonctions que vous aurez à utiliser sont :

- `json_object_new_array()` ; si le pointeur retourné est `NULL`, `do_list()` renverra simplement la chaîne vide ;
- `json_object_new_string()` ;
- `json_object_array_add()` ; si le code d'erreur retourné n'est pas 0, `do_list()` renverra simplement la chaîne vide ;
- `json_object_new_object()` ; si le pointeur retourné est `NULL`, `do_list()` renverra la chaîne vide ;
- `json_object_object_add()` ; si le code d'erreur retourné n'est pas 0, `do_list()` renverra la chaîne vide ;
- `json_object_to_json_string()` ;
- `json_object_put()` ; on pourra ignorer sa valeur de retour.

Modifier `do_list()`

Le premier objectif est d'intégrer le format JSON dans l'application `imgStoreMgr` ; cette étape est indépendante de l'intégration du serveur web et peut se faire en parallèle, p.ex. par votre binôme.

Nous allons changer le prototype de `do_list()` afin de 1. prendre comme paramètre additionnel un `enum` pour choisir entre les différents formats ; et 2. retourner le contenu comme une chaîne de caractères (plutôt que d'imprimer directement à l'intérieur de la fonction).

Concrètement, il faut faire les modifications suivantes dans `imgStore.h` (voir l'endroit marqué « `TODO WEEK 11` ») : * définir un `enum do_list_mode` comprenant les modes `STDOUT` et `JSON` ; * changer le prototype de `do_list()` pour qu'il retourne une chaîne de caractères et qu'il prenne un paramètre supplémentaire de type `enum do_list_mode`.

Modifier ensuite l'implémentation de `do_list()` (dans `imgst_list.c`) :

- si le mode est `STDOUT`, la fonction doit opérer comme avant et retourner `NULL` ;
- si le mode est `JSON`, la fonction doit utiliser la bibliothèque `libjson` (voir ci-dessus) pour construire un objet JSON avec la structure suivante :

  ```
  {
      "Images": [] # an array of the strings of the img_id fields from the metadata
  }
  ```

  (c'est donc un «*objet*» JSON qui contient une «*array*» de «*string*» qui sont les `img_id`) ; puis le convertir en chaîne de caractères (`json_object_to_json_string()`) pour le retourner.
  Attention à la durée de vie/portée des objets manipulés !.. En particulier, les chaînes contenues dans un «*objet JSON*» lui appartiennent et disparaissent avec lui.
- si le mode n'est pas connu (ni `STDOUT` ni `JSON`) la chaîne retournée sera le *message* d'erreur `"unimplemented do_list output mode"`.

Adaptez aussi, bien sûr, votre (vos ?) appel(s) à `do_list()`.

Pour tester, vous pouvez ponctuellement forcer votre `do_list_cmd()` à faire l'appel en mode `JSON` :

```
printf("%s\n", do_list(&myfile, JSON));
```

(Pour un test rapide, cela suffit ; mais ce n'est évidemment pas correct car il y a fuite de mémoire de la chaîne créée !)

Compiler et utiliser libmongoose

Comme en semaine 2, nous avons fourni `libmongoose` dans un répertoire qui contient son propre `Makefile` pour générer la bibliothèque `libmongoose.so`.

Revoir si nécessaire la semaine 2 pour sa génération et son utilisation.

Développer le serveur web

Normalement, tout est maintenant en place. Il s'agit alors d'écrire le serveur web qui combinera 1. la bibliothèque `libmongoose` en charge du protocole HTTP, 2. avec les fonctionnalités développés depuis plusieurs semaines pour le traitement du format `imgStore`.

Sur la base de votre travail effectué en semaine 2, vous aller écrire le code du serveur web dans le fichier `imgStore_server.c`.

Dans le `Makefile`, ajoutez une deuxième cible exécutable `imgStore_server` en plus de `imgStoreMgr` ; c.-à-d. que `make imgStore_server` doit compiler et faire l'édition de liens d'un programme exécutable (`imgStore_server`) qui est un serveur web.

Pour la structure de `imgStore_server.c`, commencer par reprendre votre travail effectué en semaine 2, et, *si nécessaire*, étudier quelques exemples fournis par libmongoose, en particulier :

- HTTP_server (exemple simple ; utilisation de `mg_http_serve_dir()`) ;
- RESTful_server (pour `mg_http_reply()`) ;
- video_stream (pour `mg_printf()`, `mg_send()` (pour les images) et `mg_http_match_uri()`) ;
- upload_file (pour `mg_http_upload()` la semaine prochaine).

Il ne s'agit pas de tout comprendre en détail, mais de s'en inspirer. Si nécessaire, vous pouvez compiler l'exemple correspondant en tapant simplement `make` dans une copie du répertoire d'exemple, et voir son effet en lançant l'exécutable puis en ouvrant un navigateur et vous connectant sur `http://localhost:8000`.

La documentation complète se trouve ici : https://cesanta.com/docs/ ; par exemple : `mg_http_listen()`.

Configuration de départ du serveur web Lorsque le serveur web démarre, il doit prendre un argument sur la ligne commande : le nom du fichier qui a les données en format `imgStore` ; par exemple :

```
./imgStore_server test02.imgst_dynamic
```

(en ayant si nécessaire fait une copie de sauvegarde de `test/data/test02.imgst_dynamic`).

En particulier, veillez à ce que :

- au démarrage :
  - il utilise la fonction `do_open()` pour ouvrir le `imgStore` ;
  - il lance proprement le server mongoose ;
  - puis imprime sur le terminal (*après* avoir démarré le server, et donc après le message de la libmongoose) le message : « `Starting imgStore server on http://localhost:8000` »,
  - puis imprime l'entête en utilisant la fonction `print_header()` (des semaines passées) ;
- à l'arrêt (« *shutdown* »), il utilise la fonction `do_close()`.

Pour la gestion de l'arrêt, le plus simple est de s'inspirer de l'exemple « HTTP_server ».

Interface et protocole Cette semaine, nous nous limitons à la fonctionnalité listant les images (`do_list()`).

Le protocole de communication est imposé (et utilisé par le programme client `index.html` fourni) :

- le serveur doit écouter le port 8000 ;
- le serveur doit répondre par une réponse HTTP valide, en format JSON à l'URI `/imgStore/list` (l'équivalent de `/api/f1` ou `/api/f2/*` dans l'exemple « RESTful_server ») ; écrivez pour cela :
  - une fonction `imgst_event_handler()` chargée de gérer les différents URI (pour le moment, nous n'avons que `/imgStore/list`) ;
  - une fonction `handle_list_call()` qui effectue la « requête » `do_list()` ; c'est, si vous voulez, l'équivalent pour notre server de `do_list_cmd()` pour l'interpréteur de commandes ; vous pouvez ici utiliser soit `mg_http_reply()`, soit `mg_printf()` (et `mg_send()` la semaine prochaine). (**Note :** il semble que pour que le 302 fonctionne correctement avec certains navigateurs, il faille ajouter `nc->is_draining = 1;` après le `mg_printf()`. Nous vous recommandons donc de le faire.)
- pour toutes les autres commandes, il est acceptable qu'il serve simplement du contenu statique en appelant la fonction mg_http_serve_dir(), à savoir en appelant :

```
mg_http_serve_dir(c, ev_data, &opts);
```

(comme dans l'exemple http-restful-server).

Le message HTTP que la commande `list` fournit doit avoir le format suivant :

```
HTTP/1.1 200 OK\r\n
Content-Type: application/json\r\n
Content-Length: XXX\r\n\r\n
YYY
```

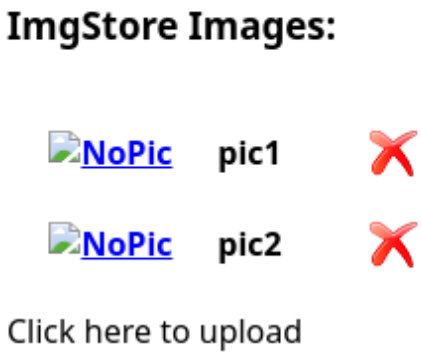
La valeur `XXX` correspondent à la longueur de la chaîne de caractères fournie (`YYY`) ; le contenu `YYY` correspond au contenu de `do_list()` retourné en format `JSON`. Utilisez la fonction `mg_printf()` (ou `mg_http_reply()`) de `libmongoose` pour écrire la réponse dans la connexion (et la semaine prochaine `mg_send()` pour envoyer des images comme les vidéos dans l'exemple « video_stream ») et faites attention à ne pas faire de fuite mémoire.

Tests

Pour tester la nouvelle version de `list`, vous pouvez, comme indiqué plus haut, simplement ponctuellement modifier son appel dans `imgStoreMgr` et voir si vous obtenez la bonne séquence JSON, par exemple :

```
{ "Images": [ "pic1", "pic2" ] }
```

Pour tester votre serveur web, lancez simplement votre `imgStore_server` depuis un répertoire où se trouve le `index.html` fourni (ou alors recopiez/linkez le depuis ici, mais ne l'ajoutez pas à votre `git`), puis ouvrez `http://localhost:8000/` dans un navigateur Web. Vous devriez obtenir quelque chose comme ça (dépend de l'image store avec lequelle vous lancez votre serveur ; ici la `test02.imgst_dynamic`) :



NOTE : pour que la bibliothèque `libmongoose` soit trouvée lors de l'exécution de `imgStore_server`, il faut indiquer au système le *répertoire* où la chercher en affectant la variable d'environnement :

— sous Linux : `LD_LIBRARY_PATH` ;
— sous Mac OS : `DYLD_FALLBACK_LIBRARY_PATH`.

Par exemple (sous Linux) :

— exemple 1 (depuis `done`) :

```
export LD_LIBRARY_PATH="${PWD}/libmongoose"
cd tests/data                    # go where index.html is
cp test02.imgst_dynamic test.db  # making a safe working copy
../../imgStore_server test.db
```

— exemple 2 (1 seule ligne, depuis `done/tests/data`) :

```
LD_LIBRARY_PATH=../../libmongoose ../.../imgStore_server test02.imgst_dynamic
```

(mais attention, dès la semaine prochaine, à ne pas corrompre la base d'images `test02.imgst_dynamic` !).

Note : pour quitter le server, tapez simplement `Ctrl-C` sur le terminal.

Rendu

Vu que nous n'avons qu'un travail partiel, vous n'avez pas à rendre tout de suite le travail de cette semaine. Celui-ci ne sera à rendre qu'à la fin (délai : le dimanche 06 juin 23:59) en même tant que tout le reste.

Ceci dit, nous vous conseillons comme toujours de marquer votre progression par des commits réguliers. Et, comme la charge de cette semaine est relativement importante, faites attention à ne pas prendre trop de retard.