
Controle des lumières KNX

last modified by Forestier Robin

on 2022/01/17 09:26

- [1.0 - But](#)
- [2.0 - Liens](#)
- [3.0 - Composants](#)
- [4.0 - Schéma Bloc](#)
- [5.0 - Shield Opto](#)
 - [5.1 - Shéma Shield Opto](#)
 - [5.2 - PCB Shield Opto](#)
- [6.0 - GUI](#)
 - [6.1 - Utilisateurs](#)
 - [6.2 - Pages](#)
 - [6.2.1 - Login](#)
 - [6.2.2 - page1](#)
 - [6.2.3 page 2](#)
 - [6.2.4 - Settings](#)
 - [6.2.5 - téléphone](#)
- [7.0 - Programmation](#)
 - [7.1 - Code Python](#)
 - [7.1.1 - Config](#)
 - [7.2 - HTML](#)
 - [7.3 - CSS](#)
- [8.0 - Instalation du raspberry](#)
 - [8.1 - Instalation](#)
 - [8.1.1 - Image](#)
 - [8.1.2 - Git clone](#)
 - [8.1.3 - NGINX](#)
 - [8.1.4 - Configuartion du server](#)
 - [8.1.5 - Configuration du Proxy NGINX](#)
 - [8.1.6 - SSL certificat](#)
 - [8.1.6.1 - Configuartion du Certificat SSL](#)
 - [8.1.7 - Sauvgarde de l'image](#)
- [9.0 - Câblage](#)
- [10.0 - Mécanique](#)

1.0 - But

Le but de ce project est de réaliser une interface graphique pour le controle des lumières de l'atelier.
L'écran est un Raspberry Pi 7" Touchscreen Display controlé par un raspberry Pi 3B+.

Une interface à l'aide d'optocoupleur est utilisée et directement contrôlé par les GPIO du raspberry.

Pour le controle des lumière, nous utilisons deux module, un US/U 4.2 de ABB et un SA/S 4.16.2.2.

2.0 - Liens

[Gitlab](#)

[ABB - US/U 4.2](#)

[ABB - SA/S 4.16.2.2](#)

[Explication KNX](#)

3.0 - Composants

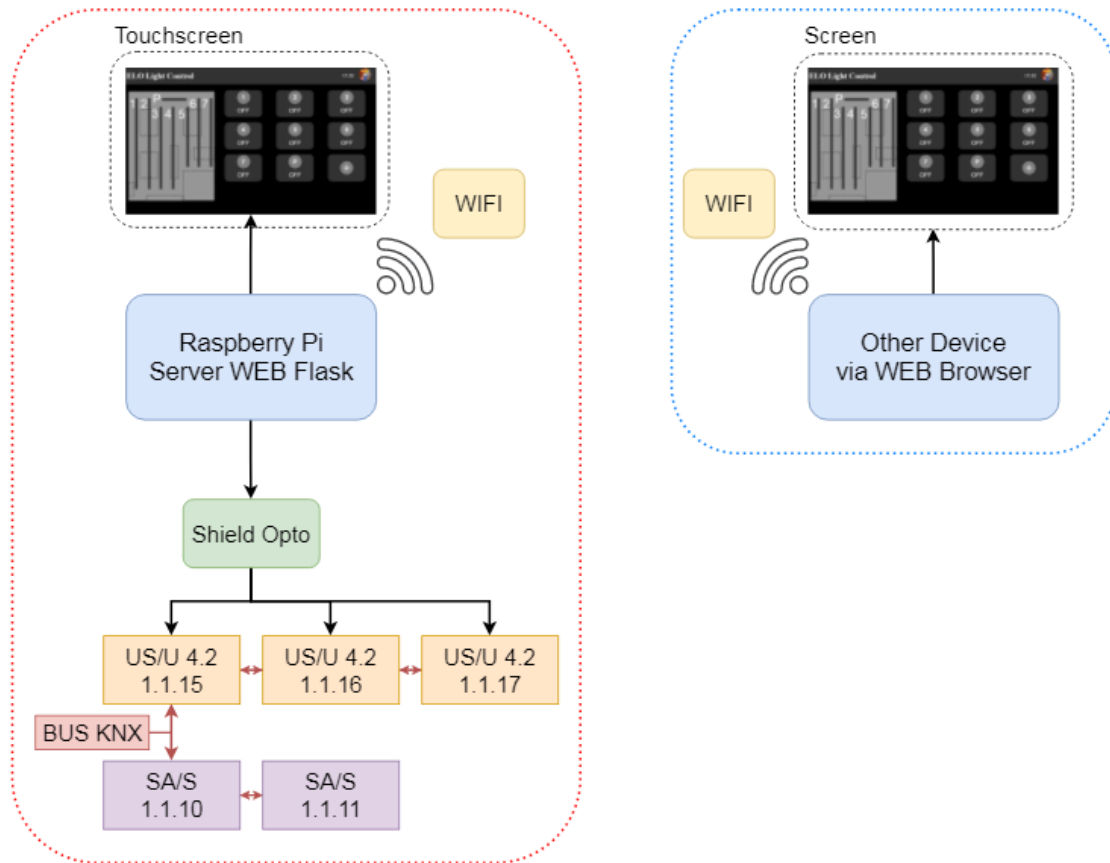
Pour le project :

Nom	Nombre	Prix unité	Prix total
Raspberry Pi 3B+	1	40.90	40.90
Raspberry Pi 7" Touchscreen	1	74.90	74.90
US/U 4.2	3	83.80	251.40
SA/S 4.16.2.2	2	169.60	339.20
Fibox ARCA 403015	1	96.90	96.90
Total			803.30

Pour la carte Shield Opto :

Reference	Value	Prix unité	Prix total
K2-K13	ASSR-1218	1.75	21
x1-x3	WAGO 218-505	3.25	9.75
x4	WAGO 218-502	1.55	1.55
Total			32.30

4.0 - Schéma Bloc

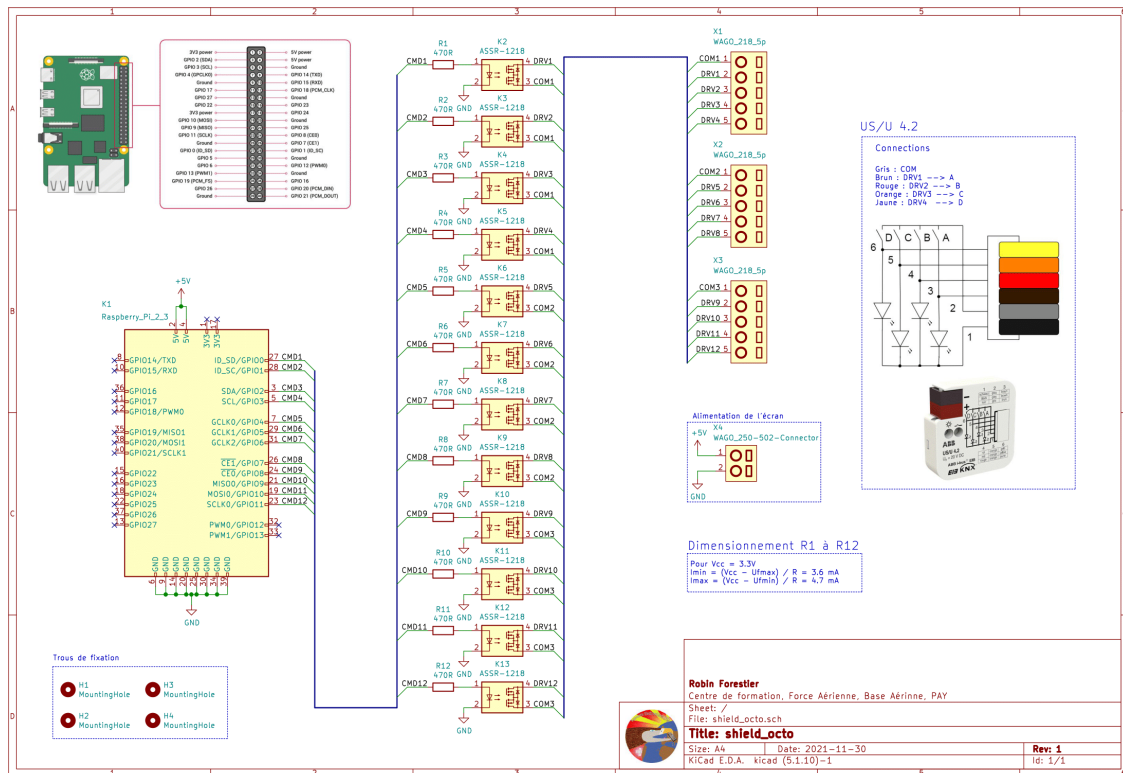


Comme on peut le voir sur ce schéma block, on utilise plusieurs système de communication.
 Les Gpio du Raspberry control directement le shiel octo puis les US/U 4.2.
 Les SU/U 4.2 communique avec les SA/S en KNX (www.knx.ch).
 Et les autre appareil communique en TCP via le wifi.

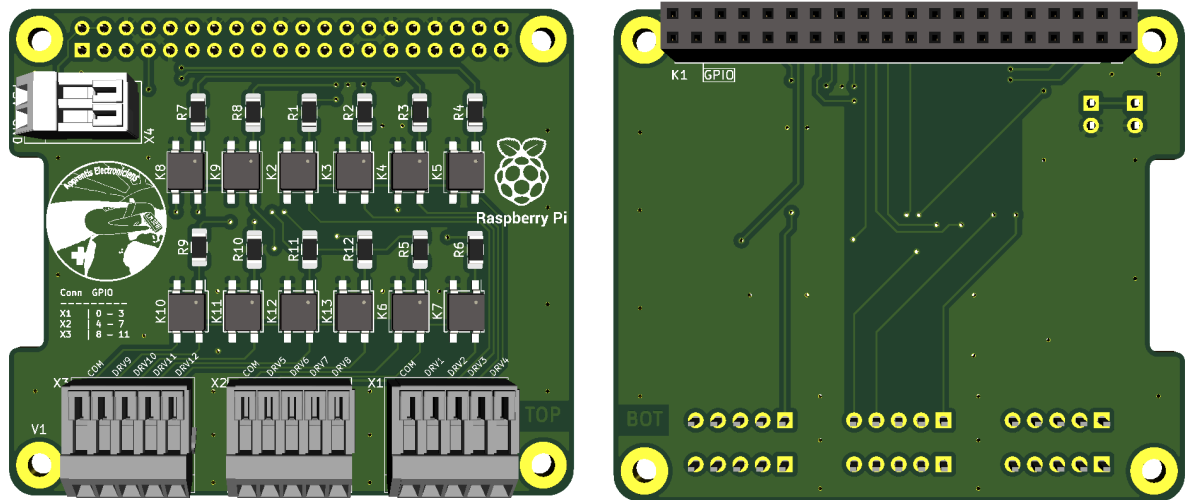
5.0 - Shield Opto

Pour contrôler les modul US/U 4.2 de chez ABB, nous utilisons des Optocoupleur mosfet contrôler par les GPIO du raspberry. J'ai donc réaliser le shield opto qui vient directement se brancher sur le raspberry pour faciliter le câblage et minimiser l'espace.

5.1 - Shéma Shield Opto



5.2 - PCB Shield Opto



6.0 - GUI

Pour la partie graphique, j'ai tout d'abord essayé avec la librairie Python TKinter.

<https://docs.python.org/fr/3/library/tkinter.html>

Les version de test se trouve sur sur le gitlab sous 5_Programmation.

Mais pour simplifier le tout et pouvoir dans le futur ajouter une partie réseaux à ce project, j'ai donc choisi d'héberger un site web. En utilisant Flask sur Python, je peux donc facilement héberger le site sur mon raspberry pi et me connecter dessus simplement en connaissant l'adresse IP.

6.1 - Utilisateurs

Pour assurer la sécurité du site, il existe 3 utilisateur différent.

1: elo

L'utilisateur elo peut, s'il connaît le mot de passe, allumer ou éteindre les lumières sans pouvoir accéder aux paramètres.

2: admin

L'utilisateur admin a tous les accès.

3: local

L'utilisateur local n'a pas besoin de se connecter pour accéder au site, son authentification se fait à l'aide de son IP. Il ne peut pas accéder aux réglages de l'application.

6.2 - Pages

Le site est constitué de 4 pages distinctes.

6.2.1 - Login

La page de login permet de se connecter via un périphérique autre que l'écran tactile.

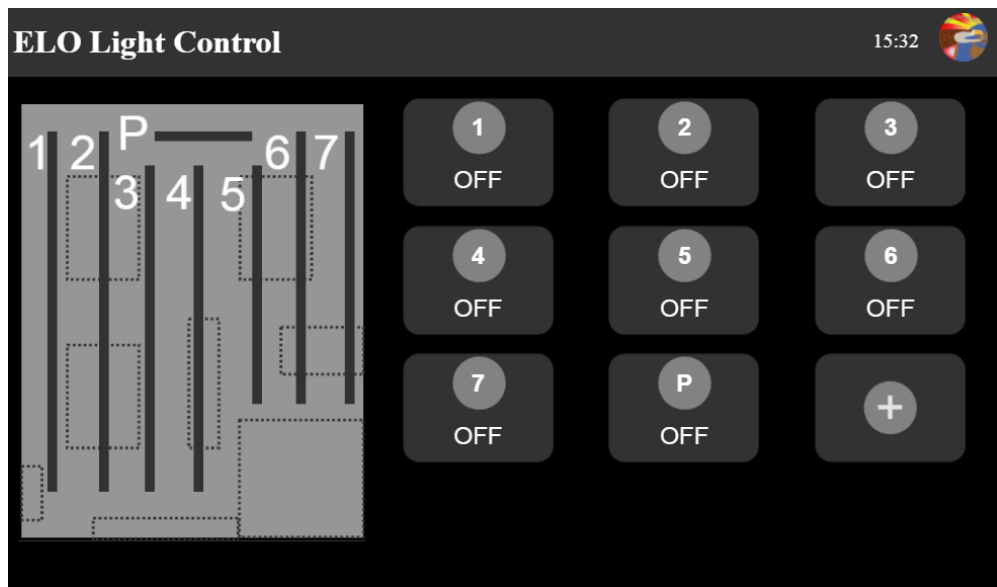


6.2.2 - page1

La page 1 contient à gauche un plan de l'atelier qui se modifie selon les lumières allumées.

A droite 9 boutons permettent d'allumer séparément chaque rail de lumières. Le bouton en bas à droite permet de se rendre sur la page 2.

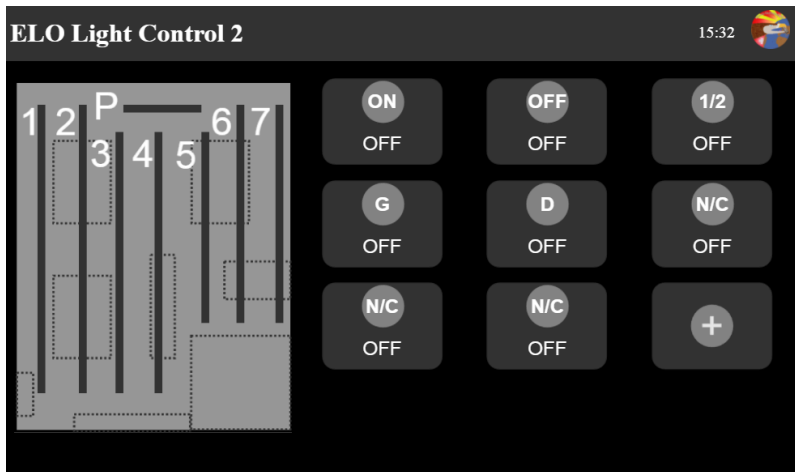
Le logo ELO en haut à droite permet d'accéder aux réglage (admin only).



6.2.3 page 2

La page 2 est une copie de la page 1 avec comme modification l'action des boutons de droites. On y retrouve:

- All ON | tout allumer
- ALL OFF | tout éteindre
- 1/2 | allumer 1 sur 2
- D | allumer la droite de l'atelier
- G | allumer la gauche de l'atelier
- N/C | non connecter (réserver pour usage future)

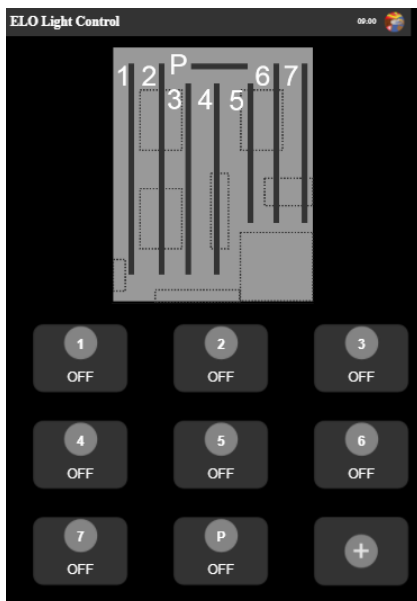


6.2.4 - Settings

La page settings accessible uniquement par le compte admin permettra de régler les paramètres importants.

6.2.5 - téléphone

J'ai aussi réalisé une mise en page pour téléphone.



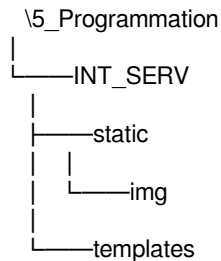
7.0 - Programmation

Pour la programmation de cette interface j'ai utilisé python. Plus précisément flask.

<https://flask.palletsprojects.com>

Flask est un micro framework open-source de développement web en Python. (Wikipedia)

Avec flask je peux facilement héberger un server web. Il suffit de créer les fichiers: static & templates.



Le programme python se trouve dans le dossier principal (INT_SERV).

Les fichiers HTML se trouvent dans templates.

Les fichiers CSS se trouvent dans static.

Les images se trouvent dans static\img.

On retrouve aussi le fichier config.csv dans le dossier principal.

7.1 - Code Python

Pour ce code j'ai besoin de ces bibliothèques.

```
main.py

from flask import Flask, g, render_template, request, session, url_for, redirect
import time
import datetime
import threading
import csv
import os


import RPi.GPIO as GPIO
```

Quand on vient se connecter avec le raspberry sur le site web on ne veut pas qu'il aie à se logger. Pour le logger automatiquement, j'ai créé une liste des IP autorisées.

```
main.py

authorize_ip = ["localhost", "127.0.0.1", "172.16.32.199"]
```

Pour pouvoir créer une session pour chaque utilisateur se connectant sur le site, j'ai créé une class User.




```
class User:
    def __init__(self, id, username, password):
        self.id = id
        self.username = username
        self.password = password

    def __repr__(self):
        return f'<User: {self.username}>'
```

Il y a 3 sortes d'utilisateur.

- 1 - le raspberry pi en local, **username = local**.
- 2 - un utilisateur, **username = elo**.
- 3 - un admin, **username = admin**.



```
users = []
users.append(User(id=1, username='elo', password='elo'))
users.append(User(id=2, username='admin', password='admin'))
```

Une fonction **before_request()** et automatiquement appelée quand un utilisateur arrive sur le site. Cette fonction viens verifier si l'utilisateur et déjà enregistrer ou pas pour lui afficher la page login ou non. C'est aussi là qu'est créer l'utilisateur local.

```
main.py

@app.before_request
def before_request():
    g.user = None

    ip = request.environ.get('HTTP_X_REAL_IP', request.remote_addr)

    for i in authorize_ip:
        if ip == i:
            if 'user_id' in session:
                user = [x for x in users if x.id == session['user_id']][0]
                g.user = user
            else :
                users.append(User(id=3, username='local', password='local'))
                user = [x for x in users if x.id == 3][0]
                session['user_id'] = user.id
                user = [x for x in users if x.id == session['user_id']][0]
                g.user = user
                return redirect(url_for('page1'))

    if 'user_id' in session:
        user = [x for x in users if x.id == session['user_id']][0]
        g.user = user
```

Chaque page web a sa propre fonction pour pouvoir interagir avec les request.
Dans l'ordre, login, page1, page2 et settings.

```
main.py

@app.route("/", methods=['POST', 'GET'])
def login():

@app.route("/page1", methods = ['POST', 'GET'])
def page1():

@app.route("/page2", methods = ['POST', 'GET'])
def page2():

@app.route("/settings", methods = ['POST', 'GET'])
def settings(setting=None):
```

Pour que l'utilisateur puisse voir quelque chose, il faut lui retourner la page.
On va donc return le fichier HTML correspondant à l'aide de la fonction *render_template()*.
On ajoute aussi toutes les variable qui seront affichées sur la page come l'heur (time).

```
main.py

return render_template('page1.html',
                       button=buttonSts_p1,
                       color=color,
                       time=current_time,
                       warning=warning)
```

Une dernière fonction est la fonction **activate_job()**.
Cette fonction s'exécute au moment où la première personne essaie de se connecter au site.
Un thread est créé pour réaliser toutes les autres activités comme : les automatisations ou le warning de température.

```
main.py

@app.before_first_request
def activate_job():
    def run_job():

        (...)

    thread = threading.Thread(target=run_job)
    thread.start()
```

Et pour finir le **main**.

```
main.py

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=80, debug=True,)
    GPIO.cleanup()
users = []
```

7.1.1 - Config

Le fichier **config.csv** contient les informations pour la page settings.
Ces informations sont stockées en externe du programme pour assurer qu'elles soient sauvegardées même après un redémarrage.
Pour le moment les seules informations stockées sont :
Auto on et Auto off ; ce qui correspond à l'allumage automatique des lumières à une certaine heure.

Voici son contenu :

name	state	param1
Auto on	on	07:30
Auto off	on	18:00

7.2 - HTML

Les fichiers HTML se trouvent sous `\5_programmation\nom_de_version\templates`
On retrouve évidemment un fichier HTML par page.

Détaillons le fichier **page1.html**.

Premièrement, j'utilise une balise *meta* pour venir reload automatiquement ma page.
Ceci permet, en cas de modification d'un autre utilisateur, d'afficher ses dites modification.



Dans le header, on peut voir l'heure affichée à gauche.
Pour le faire, dans mon code python, je viens ajouter *time* quand j'appelle la fonction *render_template()*.
Et pour l'afficher sur ma page web, il suffit d'écrire `{{ time }}`



Viens ensuite le plan de l'atelier. Ce plan est une image où je viens simplement dessiner les néon allumer. Le script récupère la couleur des différent néon et créer un rectangle à la bonne place.

```
page1.html

<div class="row" id="row">
  <div class="column-1">
    
    <canvas id="myCanvas" width="356px" height="451px">
      <script>
        window.onload = function() {
          var c = document.getElementById("myCanvas");
          var ctx = c.getContext("2d");
          var img = document.getElementById("plan");
          ctx.drawImage(img, 0, 0);

          ctx.fillStyle = "{{ color[0] }}";
          ctx.fillRect(30, 30, 10, 370);

          (...)

        }
      </script>
    </canvas>
  </div>
```

Pour finir, il reste à afficher les boutons.
Pour cela j'ai réalisé une boucle *for*.
Il faut utiliser Jinja: <https://jinja.palletsprojects.com/en/3.0.x/templates/>

Puis on crée le dernier bouton pour accéder à la page suivante.
Il faut aussi créer deux faux textes pour qu'il agisse correctement avec les règles CSS.

```
page1.html

<form method="post" action="/page1" class="column-r" id="grid">
  {% for bt in button %}
    <button class="btn" type="submit" name="button_p1" value={{ loop.index }}>
      <img class="btn_img" src={{ bt }} alt="bt1" width="150px">
      {% if loop.index == 8 %}
        <p class="text_num"> P </p>
      {% else %}
        <p class="text_num"> {{ loop.index }} </p>
      {% endif %}
      {% if bt == "/static/img/img_off.png" %}
        <p class="text_on"> OFF </p>
      {% else %}
        <p class="text_on"> ON </p>
      {% endif %}
    </button>
  {% endfor %}
  <button class="btn" type="submit" name="button_p1" value="page_2">
    
    <p class="text_num" style="visibility: hidden"> + </p>
    <p class="text_on" style="visibility: hidden"> OFF </p>
  </button>
</form>
</div>
```

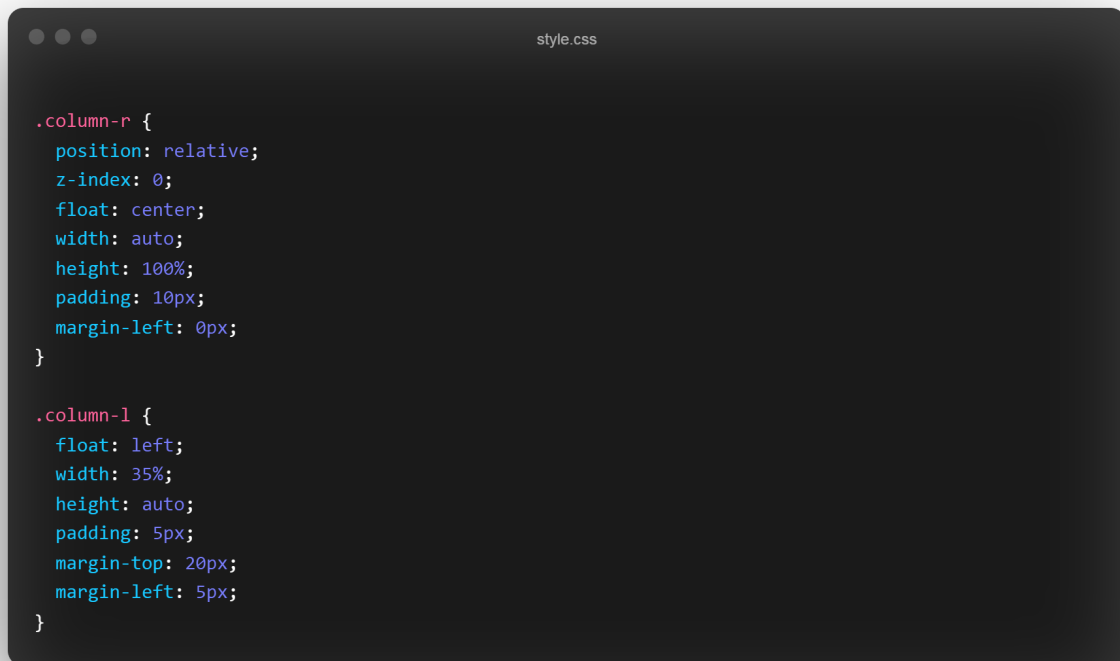

7.3 - CSS

Les fichiers CSS se trouvent dans \5_programmation\nom_de_version\static.
Il y a 3 fichier css.

style.css - utilisé pour les pages: page1 et page2.
login.css - utilisé pour la page: login.
settings.css - utilisé pour la page: settings.

Il y quelques points importants que je vais expliquer ci-dessous.

Pour l'affichage du plan a côté des boutons, j'utilise deux colonne (l et r).



```
style.css

.column-r {
  position: relative;
  z-index: 0;
  float: center;
  width: auto;
  height: 100%;
  padding: 10px;
  margin-left: 0px;
}

.column-l {
  float: left;
  width: 35%;
  height: auto;
  padding: 5px;
  margin-top: 20px;
  margin-left: 5px;
}
```

Puis pour afficher les 9 boutons, j'utilise une grille.



```
style.css

#grid {
  display: grid;
  grid-gap: 10px;
  grid-template-columns: repeat(3, 1fr);
  grid-template-rows: 15% 15% 15%;
}
```

Pour avoir un meilleur affichage sur l'écran du raspberry, j'ai créé une partie du css exprèt.

A screenshot of a code editor window titled 'style.css'. The code is a CSS media query targeting Raspberry Pi screens. It includes a comment in French and a media query rule that sets the cursor to 'none' for screens with a maximum width of 800px, a maximum height of 480px, and a minimum width of 799px.

```
style.css

/* Only for raspberry pi screen (res : 800px / 480px ) */
@media all and (max-width: 800px) and (max-height: 480px) and (min-width: 799px){

    * {
        cursor: none;
    }
}
```

Et j'ai aussi créé une partie pour les téléphones portables.

A screenshot of a code editor window titled 'style.css'. The code is a CSS media query targeting mobile phones. It includes a comment in French and a media query rule that targets only screen devices with no hover, coarse pointer, and portrait orientation.

```
style.css

/* Only for phone */
@media only screen and (hover: none) and (pointer: coarse) and (orientation: portrait){
}
```

8.0 - Instalation du raspberry

Pour l'installation du raspberry il vous suffit de récupérer cette image :

ref: OMVSERVER\formation\Projets\ControleDesLumiersKNX\lightcontrol.img

et de la flasher sur une carte SD.

Vous pourrez ensuite pull la dernière version du gitlab.
Tout et prêt a fonctionner !

8.1 - Installation

Pour réaliser une instalation complète suivez ces étapes.

8.1.1 - Image

Pour faire fonctionner votre raspberry, il vous faudra une image.

Vous pouvez simplement télécharger Raspberry Pi Imager.

<https://www.raspberrypi.com/software/>

Installer l'OS de votre choix sur une carte SD.

Configuré votre Raspberry.
vous pouvez installer **Xscreen saver** pour désactiver la mise en veille.

```
sudo apt-get install xscreensaver
```

Puis dans l'app, choisir désactiver la mise en veille.

Vous pouvez aussi activer VNC et le SSH pour vous faciliter la vie.

Pour utiliser les GPIO correctement vous devez :
sudo raspi-config

8.1.2 - Git clone

Pour cloner le gitlab copier cette commande dans votre terminal.
(vérifier s'il n'a pas été déplacé sous Projets_Ancien_Apprentis)

git clone <http://172.16.32.230/Forestier/controle-des-lumieres-knx>

8.1.3 - NGINX

Vous pouvez tester le server en exécutant le main.py dans le dossier server.

```
cd controle-des-lumieres-knx/5_Programmation/server  
sudo python3 main.py
```

Pour que le server WEB se lance automatiquement aux lancement du Raspberry Pi et pour que le server se soit plus un serveur de développement,

J'ai choisi d'utiliser un server NGINX, il peut aussi servir de proxy mais je n'est pas souhaiter aller aussi loin dans ce développement.

Pour l'installation vous pouvez suivre ce tuto : <https://www.digitalocean.com/community/tutorials/how-to-serve-flask-applications-with-gunicorn-and-nginx-on-ubuntu-18-04>
(/1_Documentation/Flask_server_Gunicorn_DigitalOcean.pdf)

Ou sinon lire la suite.
Premièrement dans ce tuto ils créent un environnement virtuel. Si vous avez cloné le GitLab alors il existe déjà.

source serverenv/bin/activate

Vous devez ensuite vérifier si le fichier wsgi.py existe.

Sinon créer le.



```
wsgi.py

from myproject import app

if __name__ == "__main__":
    app.run()
```

Utiliser la commande suivante pour lancer votre server avec uWsgi:

```
gunicorn --bind 0.0.0.0:5000 wsgi:app
```

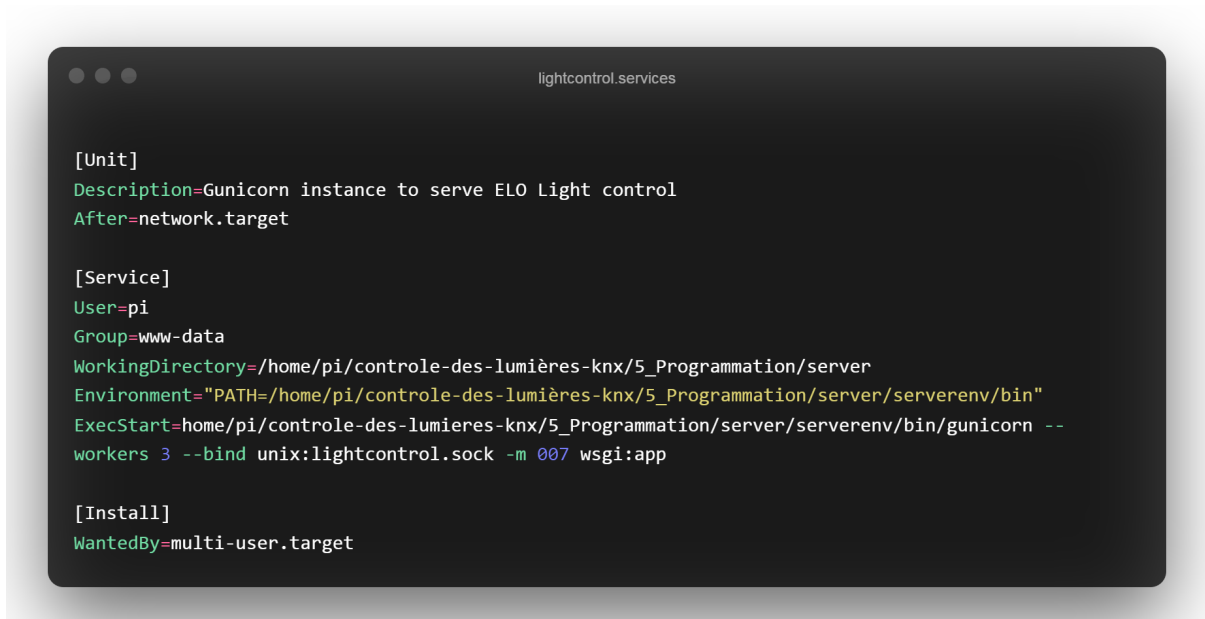
Vous pouvez maintenant sortir de l'environnement virtuel.

deactivate

8.1.4 - Configuration du server

Pour le configurer :

```
sudo nano /etc/systemd/system/lightcontrol.service
```



```
lightcontrol.service

[Unit]
Description=Gunicorn instance to serve ELO Light control
After=network.target

[Service]
User=pi
Group=www-data
WorkingDirectory=/home/pi/controle-des-lumières-knx/5_Programmation/server
Environment="PATH=/home/pi/controle-des-lumières-knx/5_Programmation/server/serverenv/bin"
ExecStart=/home/pi/controle-des-lumières-knx/5_Programmation/server/serverenv/bin/gunicorn --workers 3 --bind unix:lightcontrol.sock -m 007 wsgi:app

[Install]
WantedBy=multi-user.target
```

Now start the Gunicorn service you created:

```
sudo systemctl start lightcontrol
```

Then enable it so that it starts at boot:

```
sudo systemctl enable lightcontrol
```

Check the status:

```
sudo systemctl status lightcontrol
```

Vous devez voir:

Active: active (running)

si ce n'est pas le cas re vérifier les fichiers précédent.

Et vous avez presque fini. Il faut encore que votre navigateur se lance automatiquement au démarrage. Pour cela il faut rajouter cette ligne:

@chromium --kiosk <http://localhost>

Dans le fichier:

```
sudo nano /etc/xdg/lxsession/LXDE-pi/autostart
```

Chromium se lancera automatiquement en mode pleine écran et vous connectera sur votre site web (localhost). Avec cette simple installation vous n'avez pas de proxy et pas de sécurisation de vos packets.

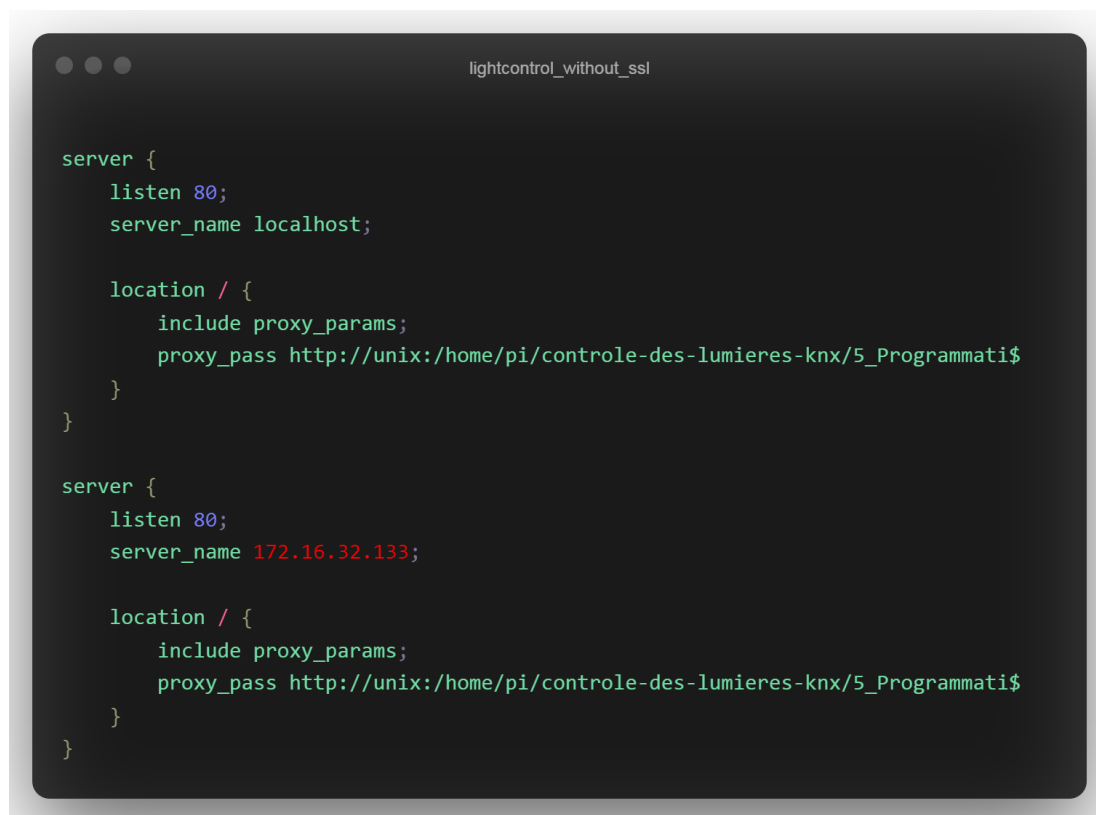
8.1.5 - Configuration du Proxy NGINX

On va maintenant mettre en place le proxy NGINX.

On commence par créer un fichier de configuration.

```
sudo nano /etc/nginx/sites-available/lightcontrol
```

Voilà comment il doit être rempli pour que toutes les requets vers l'ip du raspberry soit reçue et traitée par le serveur.



```
lightcontrol_without_ssl

server {
    listen 80;
    server_name localhost;

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/controle-des-lumieres-knx/5_Programmati$
    }
}

server {
    listen 80;
    server_name 172.16.32.133;

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/controle-des-lumieres-knx/5_Programmati$
    }
}
```

Pour autoriser, il faut créer un lien vers le répertoire "sites-enabled".

```
sudo ln -s /etc/nginx/sites-available/myproject /etc/nginx/sites-enabled
```

Maintenant vous pouvez vérifier si il n'y a pas d'erreur de syntaxe.

```
sudo nginx -t
```

Si il n'y a pas d'erreur, vous pouvez relancer NGINX.

```
sudo systemctl restart nginx
```

Votre site web avec proxy est prêt.

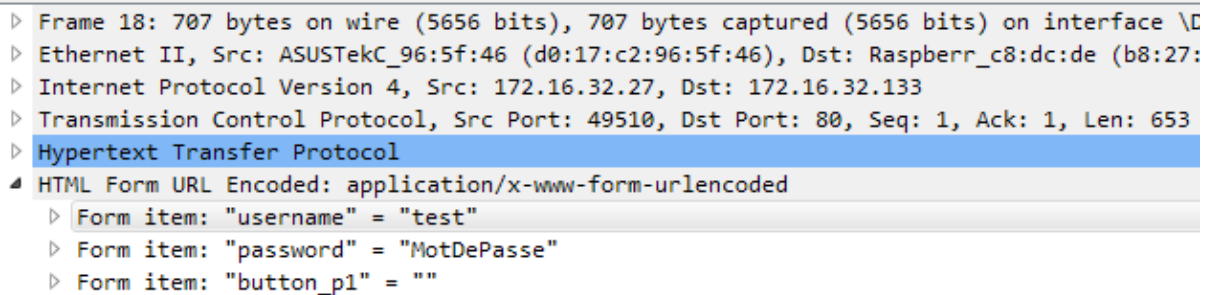
8.1.6 - SSL certificat

Vous êtes maintenant avec un serveur http fonctionnelle.

Mais le problème reste qu'en terme de sécurité, l'http ne crypte pas les communication.

La preuve, j'ai utilisé Wireshark (www.wireshark.org) pour sniffer les communication entre mon Pc et le Raspberry (server).

Voici ce que j'ai obtenu:



The image shows a Wireshark packet capture of an HTTP POST request. The packet list on the left shows 'Frame 18: 707 bytes on wire (5656 bits), 707 bytes captured (5656 bits) on interface \D'. The packet details pane on the right shows the following structure:

- Ethernet II, Src: ASUSTekC_96:5f:46 (d0:17:c2:96:5f:46), Dst: Raspberr_c8:dc:de (b8:27:0d:17:c2:96:5f:46)
- Internet Protocol Version 4, Src: 172.16.32.27, Dst: 172.16.32.133
- Transmission Control Protocol, Src Port: 49510, Dst Port: 80, Seq: 1, Ack: 1, Len: 653
- Hypertext Transfer Protocol
- HTML Form URL Encoded: application/x-www-form-urlencoded
 - Form item: "username" = "test"
 - Form item: "password" = "MotDePasse"
 - Form item: "button_p1" = ""

On peut clairement voir toutes les informations de mon login.

Pour palier à cela, on va passer notre serveur en HTTPS.

Il nous faudra un "Secure Socket Layer" ou SSL certificate, Il permet de sécuriser tous les transaction entre le serveur et le PC.

Normalement une clé SSL est payante et fonctionne avec un nom de domaine mais travaillant sans nom de domaine, je vais simplement utiliser openssl (<https://www.openssl.org>).

8.1.6.1 - Configuration du Certificat SSL

On commence par créer un dossier dans lequel on va placer les clés.

```
sudo mkdir /etc/nginx/ssl/  
cd /etc/nginx/ssl/
```

Puis, on va générer la clé privée RSA:

```
sudo openssl genrsa -des3 -out server.key 2048
```

Entrez une phrase (plus longue qu'un simple mot de passe) et répétez-la pour confirmer.
Rien ne s'affiche à l'écran lorsque vous tapez. C'est normal.

Après quelques secondes, on obtient le fichier *server.key*.

A partir de la clé privée RSA, on va maintenant générer un CSR (Certificate Signing Request).
C'est-à-dire un formulaire de demande de certificat.

```
sudo openssl req -new -key server.key -out server.csr
```



```
Country Name (2 letter code) [AU]:CH
State or Province Name (full name) [Some-State]:Switzerlan
Locality Name (eg, city) []:Payerne
Organization Name (eg, company) [Internet Widgits Pty Ltd]:ELO
Organizational Unit Name (eg, section) []:elo
Common Name (e.g. server FQDN or YOUR name) []:Robin Forestier
Email Address []:robin.forestier@zappvion.ch
```

La seule indication importante est votre nom de domaine « Common Name« .

domaine.tld protégera tous les sous-domaines. sousdomaine.domaine.tld ne protégera que le sous domaine mentionné.

Laissez les « extra attributes » vides.

Voilà, la demande de certificat server.csr est créée. Il ne reste plus qu'à la valider. Normalement, c'est une autorité de certification (CA) qui doit signer votre CSR. Elle se porte ainsi garante de l'authenticité des données qui se trouvent dans le certificat. Malheureusement, ce service est souvent payant.

Pour contourner le « problème », nous allons nous ériger en CA et signer nous même le CSR.

On commence par archiver notre clef privée :

```
sudo cp server.key server.key.tmp
```

Puis, on va décrypter la clef privée pour permettre à Nginx de s'en servir sans avoir besoin du mot de passe :

```
sudo openssl rsa -in server.key.tmp -out server.key
```

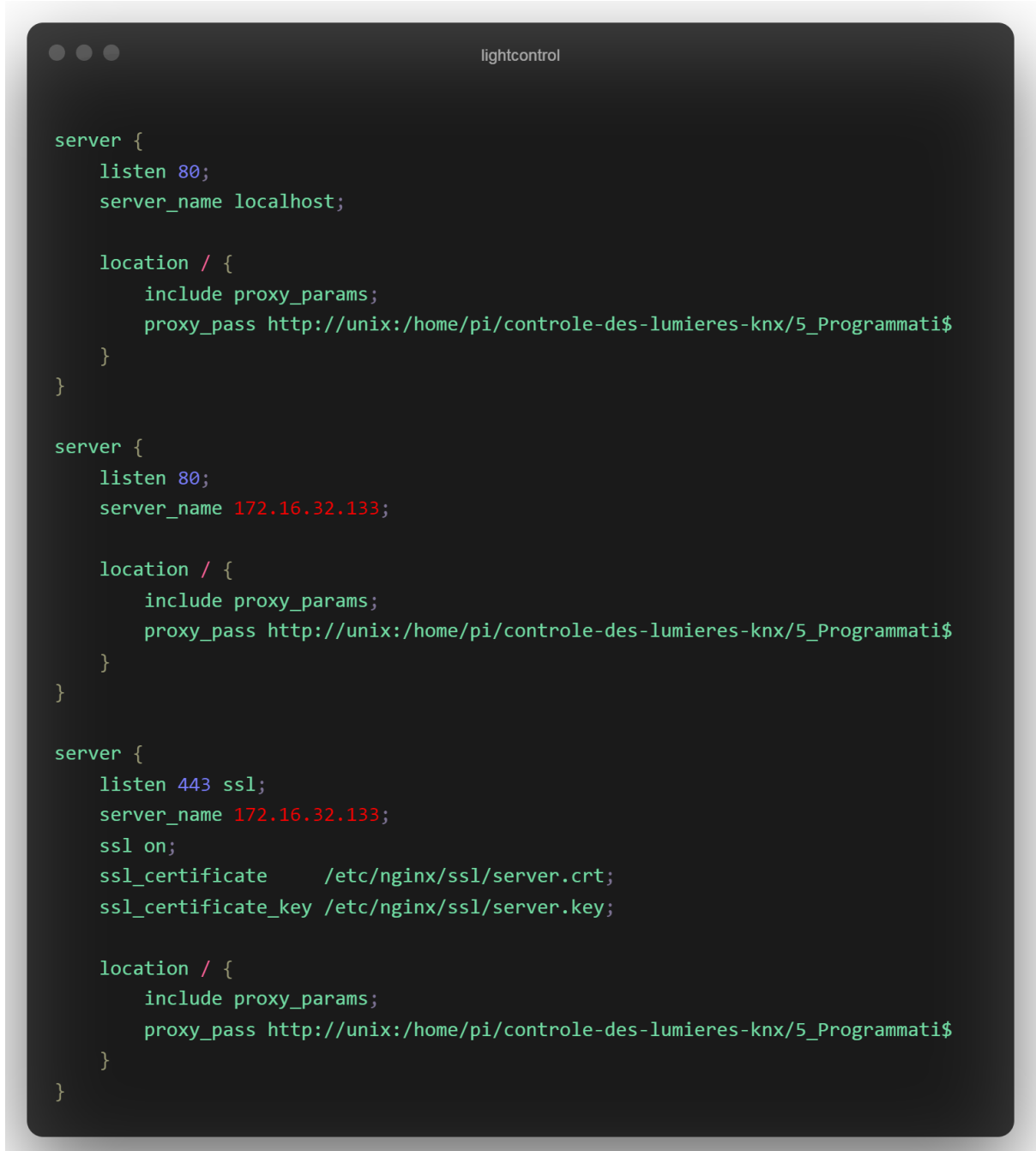
On entre la phrase choisie précédemment.

Puis, on signe le CSR avec notre nouvelle clef privée décryptée :

```
sudo openssl x509 -req -days 1000 -in server.csr -signkey server.key -out server.crt
```

Il ne reste plus qu'à ajouter la configuration pour le port 443 et le ssl dans le fichier lightcontrol.

```
sudo nano /etc/nginx/sites-available/lightcontrol
```



```
server {
    listen 80;
    server_name localhost;

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/controle-des-lumieres-knx/5_Programmati$
    }
}

server {
    listen 80;
    server_name 172.16.32.133;

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/controle-des-lumieres-knx/5_Programmati$
    }
}

server {
    listen 443 ssl;
    server_name 172.16.32.133;
    ssl on;
    ssl_certificate /etc/nginx/ssl/server.crt;
    ssl_certificate_key /etc/nginx/ssl/server.key;

    location / {
        include proxy_params;
        proxy_pass http://unix:/home/pi/controle-des-lumieres-knx/5_Programmati$
    }
}
```

sudo nginx -t

sudo systemctl restart nginx

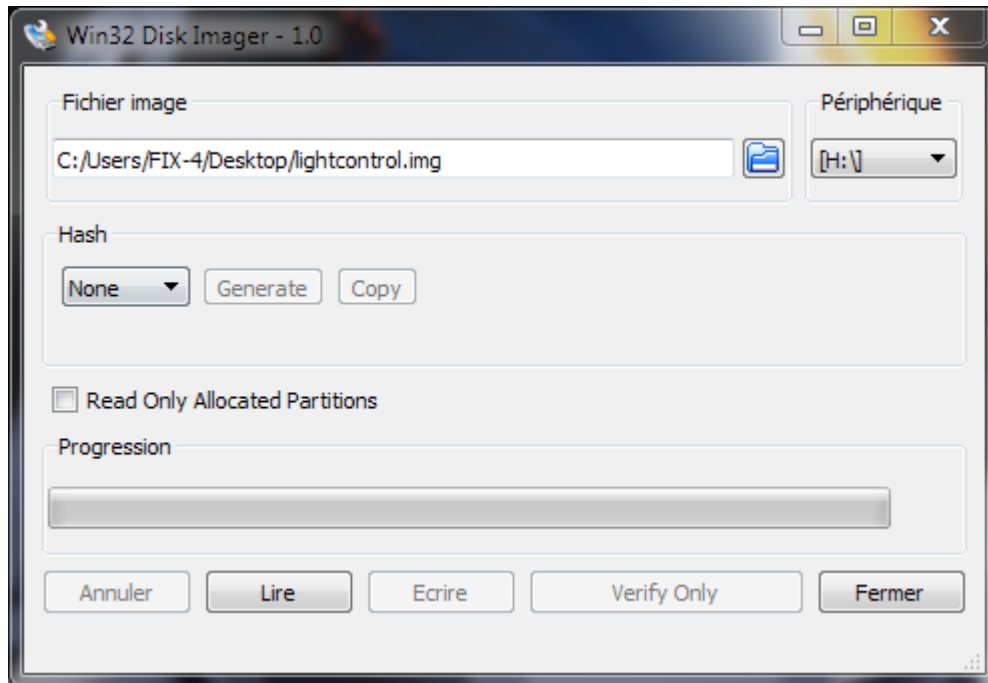
Et c'est fini, vous avez enfin complètement configuré votre serveur.

8.1.7 - Sauvgarde de l'image

Il ne vous reste plus qu'à sauvgarder l'image de votre raspberry. Pour cela il vous faudra le logiciel Win32 Disk Imager.

Bancher la carte SD sur votre PC. Dans L'interface indiquer l'emplacement ou ira se stocker votre image et son nom.

Sélectionner le périphérique, puis cliquer sur Lire. Il faudra attendre que la copie se face puis votre image serra copiée.



ref: OMVSERVER\formation\Projets\ControleDesLumiersKNX\lightcontrol.img

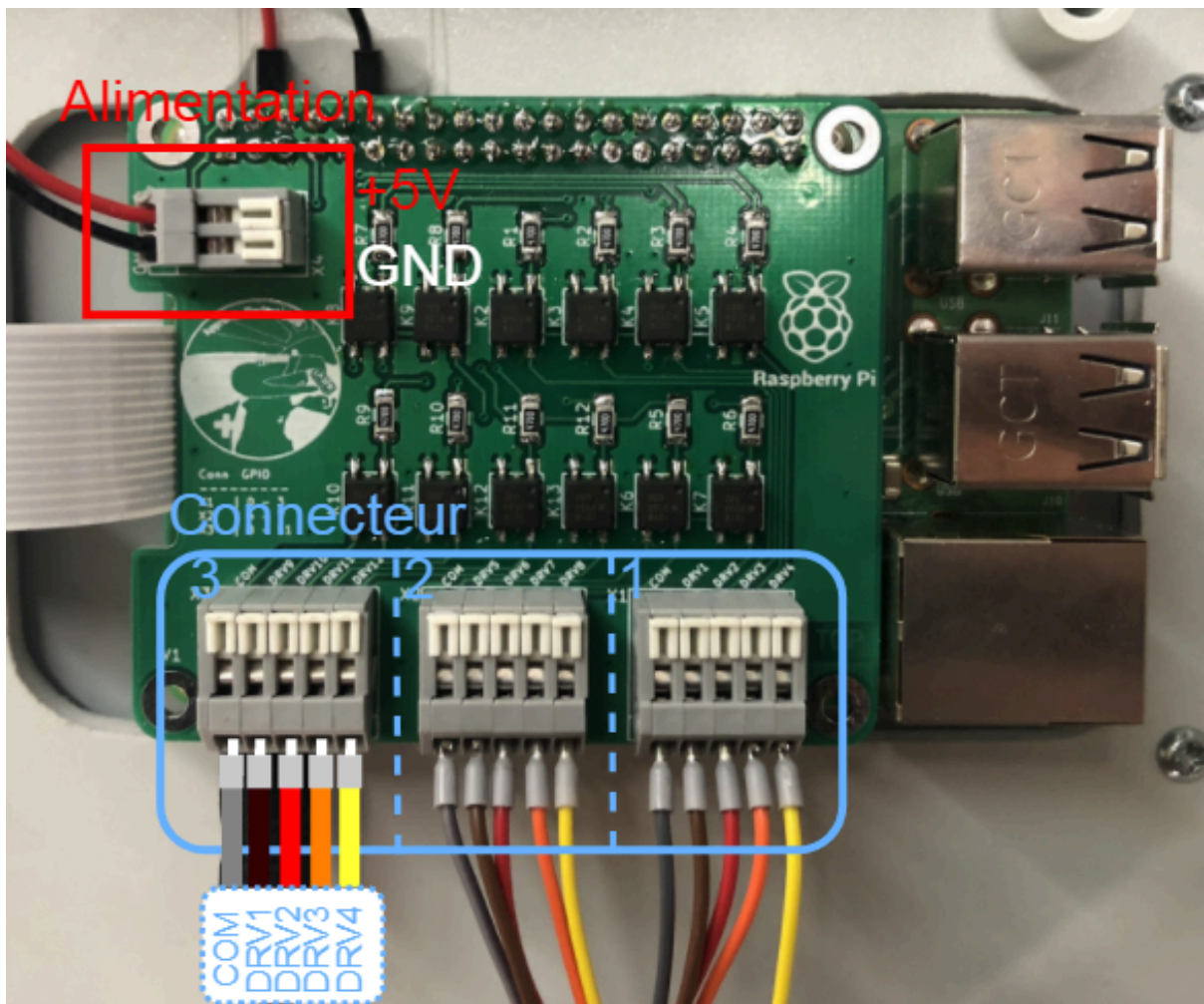
9.0 - Câblage

Pour le câblage, il faut brancher l'alimentation de l'écran sur le connecteur Wago X4 à gauche de la carte. Les US/U 4.2 se branche dans les connecteurs X1 - X3. Chaque connecteur comporte 5 pôles.

COM	DRV1	DRV2	DRV3	DRV4
-----	------	------	------	------

Les connecteurs sont connecter aux GPIO suivant :

Connecteur	GPIO
X1	0 - 3
X2	4 - 7
X3	8 - 11



10.0 - Mécanique

Pour placer l'écran sur la porte du boîtier Fibox, j'ai percé 4 trous et réalisé une fenêtre carrée pour que le raspberry puisse y passer.

ref: /1_Documentation/7-inch-display-mechanical-drawing.pdf &

/1_Documentation/raspberry-pi-3-b-plus-mechanical-drawing.pdf on GitLab.

