1 - Présentation du serveur

1. Objectif du projet

Le projet consiste à concevoir un serveur miniature basé sur un microcontrôleur ESP32, capable d'héberger un site web localement et de fournir des services comme l'exploration de fichiers, l'authentification utilisateur, et la gestion de contenus via une carte SD. Ce serveur est destiné à être utilisé dans des environnements à faible consommation énergétique et pour des besoins personnels ou expérimentaux.

2. Matériel utilisé

- Carte ESP32 WROVER
- Carte microSD (FAT32, jusqu'à 32 Go)
- Alimentation 5V 1A
- Réseau WiFi local

3. Technologies embarquées

- Langage : C++ (Arduino)
- Serveur web HTTP intégré (ESPAsyncWebServer)
- Stockage via carte SD (SPI)
- Interface HTML/CSS/JS
- Authentification avec cookies SHA-256
- API REST pour les actions du serveur (upload, login, etc.)

4. Fonctionnalités principales du serveur

- Hébergement de pages web statiques (HTML/CSS/JS)
- Gestion de fichiers (explorateur, upload, renommage, suppression…)
- Authentification utilisateur par hash SHA-256 + cookie
- Interface d'administration avec gestion multi-onglets
- API REST accessible via navigateur

5. Architecture générale

Le serveur repose sur plusieurs fichiers .ino responsables de fonctions spécifiques :

- api.ino : endpoints API (login, logout, fs...)
- upload.ino : traitement des uploads
- router.ino : routage HTTP
- server.ino : serveur principal
- kvdb.ino : gestion des utilisateurs
- internal.ino : utilitaires système
- web-server.ino : point d'entrée principal
- utils.ino : fonctions utilitaires

L'ensemble est chargé depuis une carte SD, et les pages HTML sont servies à travers le réseau WiFi local.

6. Limitations rencontrées

Un test SSL (HTTPS) a été réalisé, mais il s'est révélé très instable sur ESP32 avec les bibliothèques disponibles. De plus, certains navigateurs comme Firefox bloquent les cookies utilisés pour l'authentification, ce qui provoque des échecs de connexion. Pour contourner cela, les cookies générés par le serveur ont été récupérés manuellement dans l'inspecteur de navigateur et réinsérés à la main pour valider les tests.

2 - Analyse du fichier : web-server.ino

Le fichier 'web-server.ino' constitue le point de départ de l'exécution du serveur. C'est lui qui lance les modules principaux et initialise les ressources nécessaire comme la carte SD, le réseau WiFi ou encore la base de données utilisateurs. Sans ce fichier, rien ne démarre. Il représente en quelque sorte le chef d'orchestre du projet.

1. Rôle du fichier dans le projet

Son rôle: il prépare tout ce dont le serveur a besoin pour fonctionner. Dès que l'ESP32 démarre, la fonction 'setup()' est appelée. Elle se charge d'ouvrir la communication série, de monter la carte SD, de lancer le réseau WiFi, d'activer la base de données utilisateur et enfin de démarrer le serveur web. La fonction 'loop()', quant à elle, ne contient rien car tout le serveur fonctionne en mode asynchrone, ce qui évite les blocages et rend le système plus réactif.

2. Explication du déroulement du setup

Tout commence avec 'Serial.begin(115200)', qui permet d'afficher des messages dans la console série pour suivre ce qu'il se passe pendant le démarrage. Ensuite, on monte la carte SD grâce à 'SD.begin()'. Si cette étape échoue, rien ne pourra être lu ni écrit, donc le serveur ne pourra pas charger ses pages web. Puis on configure le réseau WiFi en mode point d'accès avec 'WiFi.softAP()', ce qui permet aux utilisateurs de se connecter directement au serveur. Une fois le réseau prêt, la base de données d'utilisateurs est initialisée avec la fonction 'init_kvdb()'. Cette base contient les identifiants autorisés à se connecter. Enfin, 'init_router()' démarre le serveur HTTP et configure toutes les routes API disponibles, comme le login ou la gestion des fichiers.

3. Comment ce fichier interagit avec le reste

Le fichier 'web-server.ino' est connecté aux autres parties du projet. Il dépend du fichier 'kvdb.ino' pour gérer les utilisateurs, du fichier 'router.ino' pour créer le serveur HTTP, et indirectement de 'api.ino' et 'upload.ino' qui définissent les actions accessibles via le réseau. On peut dire que c'est ce fichier qui donne le signal de départ à tous les autres.

3 - Analyse du fichier : server.ino

Le fichier 'server.ino' joue un rôle essentiel dans le fonctionnement du serveur. C'est ici que le serveur HTTP asynchrone est déclaré, configuré, et prêt à répondre aux requêtes des utilisateurs. Sans ce fichier, aucune interaction via le navigateur ne serait possible. Il agit comme le cœur du serveur en écoutant et traitant les requêtes HTTP.

1. Mise en place du serveur HTTP

Le code commence par la déclaration d'un serveur web asynchrone, basé sur la bibliothèque ESPAsyncWebServer. Cette approche permet au serveur de répondre à plusieurs requêtes simultanément sans bloquer les autres opérations. Ensuite, plusieurs routes sont définies, permettant par exemple de gérer l'authentification, la lecture de fichiers ou encore la gestion des erreurs.

2. Exemple de code : démarrage du serveur

Voici un extrait de code qui montre comment le serveur est initialisé et comment les premières routes sont définies.

```
/*

    Web Server

    This is the main entry point of the WebStick bare metal
    web server. If you have exception rules that shall not
    be handled by the main router, you can do them here.

*/

//Check if a user is authenticated / logged in
// Serve directory entries from SD card
void HandleSDContent(AsyncWebServerRequest *request, String dirName, String dirToList) {
  AsyncResponseStream *response = request->beginResponseStream("text/html");

  // Open the directory in the SD card
  File directory = SD.open(dirToList);

  // Check if the directory is open
  if (!directory) {
    SendErrorResp(request, "Unable to open directory");
    return;
  }

  response->print("<!DOCTYPE html><html><head><meta charset=\"utf-8\"><title>Content of " + dirName + "</title></head><body style=\"margin: 3em;font-family: Arial;\">");
  response->print("<h3>Content of " + dirName + "</h3><div style=\"width: 100%;border-bottom: 1px solid #d9d9d9;\"></div><ul>");

  // List the contents of the directory from the SD card
  while (true) {
    File entry = directory.openNextFile();
```

On utilise AsyncWebServer server(80);' pour créer une instance du serveur, écoutant sur le port 80. Puis, on configure différents comportements pour les routes HTTP entrantes. Chaque route peut être reliée à une fonction ou à une réponse statique.

3. Comment ce fichier s'intègre dans le projet

Ce fichier est utilisé par 'router.ino' pour ajouter des routes supplémentaires ou modifier les existantes. Il est appelé lors de l'initialisation dans 'web-server.ino', via une fonction du type 'init_router()'. C'est grâce à cette organisation qu'il est possible de séparer la logique réseau, les API, et les utilitaires dans des fichiers distincts.

4. Conclusion

Le fichier 'server.ino' ne se contente pas de lancer un serveur ; il définit la manière dont les utilisateurs peuvent interagir avec l'appareil. Il représente l'interface réseau de tout le projet. Toute fonctionnalité visible dans le navigateur passe d'abord par ici.

4 - Analyse du fichier : router.ino

Le fichier 'router.ino' est l'un des éléments les plus importants du projet. Il sert à définir toutes les routes HTTP que le serveur pourra reconnaître. En d'autres termes, c'est lui qui fait le lien entre les requêtes du navigateur et les fonctions du serveur. Il ne traite pas directement les données, mais il indique au serveur comment réagir lorsqu'un utilisateur accède à une URL spécifique.

1. Mise en place des routes

À l'intérieur de ce fichier, on utilise la structure du serveur asynchrone pour enregistrer des chemins comme '/', '/api/auth', '/api/fs', etc. Chaque route est associée à une méthode (GET ou POST) et à une fonction de traitement. C'est aussi ici qu'on configure les erreurs 404 ou qu'on redirige l'utilisateur en fonction de son niveau d'accès.

2. Exemple de code : début du routage

Ci-dessous, on peut voir une partie du code qui montre comment les routes principales sont définies.

```
/*

    Router.ino

    This is the main router of the whole web server.
    It is like the apache.conf where you handle routing
    to different services.

    By default, all route will go to the SD card /www/ folder
*/
void serveStaticFile(AsyncWebServerRequest *request, const String &filePath, const String &mimeType) {
  File dataFile = SD.open(filePath.c_str());

  if (dataFile) {
    if (request->hasArg("download")) {
      // Set the appropriate content type for download
      request->send(SD, filePath, "application/octet-stream");
    } else {
      // Stream the file with the specified MIME type
      request->send(SD, filePath, mimeType);
    }
    dataFile.close();
  } else {
    // Handle the case where the file doesn't exist
    request->send(404, "text/plain", "File not found");
  }
}

class MainRouter : public AsyncWebHandler {
public:
```

Dans cet exemple, chaque appel à 'server.on()' correspond à une URL. La fonction passée en paramètre sera exécutée automatiquement lorsqu'un utilisateur accède à cette URL via son

navigateur. C’est une manière très directe et intuitive de contrôler le comportement du serveur.

3. Intégration dans l’ensemble du projet

Ce fichier travaille en étroite collaboration avec 'api.ino', qui fournit les fonctions de traitement des API. Il est aussi appelé depuis 'web-server.ino' grâce à une fonction comme 'init_router()', qui permet de l’activer au démarrage. Enfin, il s’appuie sur le serveur défini dans 'server.ino', puisque c’est sur cette base qu’il enregistre les routes.

4. Conclusion

Le routage est essentiel pour que le serveur sache quoi faire quand un utilisateur navigue sur une page ou déclenche une action. Ce fichier donne la carte complète des chemins possibles et permet une grande modularité. C’est grâce à lui qu’on peut gérer aussi bien l’accès aux fichiers que l’authentification ou les appels AJAX des pages HTML.

5 - Analyse du fichier : api.ino

Le fichier 'api.ino' regroupe toutes les fonctions liées aux API REST utilisées par le serveur. Il permet au serveur de répondre aux actions envoyées depuis les pages web, comme se connecter, se déconnecter, ou effectuer des opérations sur les fichiers. C'est un point central pour la communication entre l'utilisateur et le système embarqué.

1. Authentification utilisateur

L'un des éléments les plus sensibles dans ce projet est la gestion de l'authentification. Le serveur vérifie que l'utilisateur fournit un identifiant et un mot de passe (hashé en SHA-256), puis en cas de succès, il génère un cookie de session qui est envoyé au navigateur. C'est ce cookie qui permettra ensuite d'accéder aux pages protégées comme l'admin ou le gestionnaire de fichiers.

Voici un extrait de code qui montre comment le login est traité côté serveur :

```
/*

    API.ino

    This script handle API requests
    functions.
*/

/* Utilities Functions */
String GetPara(AsyncWebServerRequest *request, String key) {
  if (request->hasParam(key)) {
    return request->getParam(key)->value();
  }
  return "";
}

void SendErrorResp(AsyncWebServerRequest *r, String errorMessage) {
  //Parse the error message into json
  StaticJsonDocument<200> jsonDocument;
  JsonObject root = jsonDocument.to<JsonObject>();
  root["error"] = errorMessage;
  String jsonString;
  serializeJson(root, jsonString);

  //Send it out with request handler
  r->send(200, "application/json", jsonString);
}

void SendJsonResp(AsyncWebServerRequest *r, String jsonString) {
  r->send(200, "application/json", jsonString);
}

void SendOK(AsyncWebServerRequest *r) {
  r->send(200, "application/json", "\"ok\"");
}
```

(

On voit ici que si les identifiants sont valides, un token est généré et envoyé sous forme de cookie. C'est ce cookie qui est ensuite utilisé par les autres routes protégées pour vérifier l'accès.

2. Autres points d'entrée API

Le fichier ne se limite pas au login. Il permet aussi de gérer la déconnexion, de vérifier si une session est encore active ('/api/auth/chk'), et d'accéder à des opérations liées aux fichiers. Chaque API est définie avec une route spécifique, et le serveur renvoie une réponse JSON avec un statut clair.

3. Comment api.ino interagit avec les autres fichiers

Ce fichier repose fortement sur 'kvdb.ino' pour la vérification des identifiants, et sur 'server.ino' pour s'enregistrer dans le serveur HTTP. Il travaille aussi en collaboration avec le système de fichiers pour permettre l'exploration, l'upload ou la suppression de fichiers à travers les API REST.

4. Difficultés rencontrées

La partie la plus compliquée à comprendre a été le fonctionnement des cookies. Dans certains navigateurs comme Firefox, le cookie n'était pas accepté par défaut, ce qui bloquait l'accès à l'espace administrateur. Pour contourner cela, on a dû inspecter les cookies générés manuellement dans l'interface développeur du navigateur et les réécrire à la main pour tester les routes protégées.

5. Conclusion

Ce fichier est indispensable pour permettre à l'utilisateur de dialoguer avec le serveur. C'est lui qui transforme les actions côté client (clic sur un bouton, envoi d'un mot de passe) en traitements côté ESP32, avec des retours structurés en JSON. Il centralise toute la logique métier de haut niveau du serveur.

6 - Analyse du fichier : upload.ino

Le fichier 'upload.ino' est chargé de gérer la réception des fichiers envoyés depuis l'interface web. C'est une partie essentielle du projet puisque l'une des fonctionnalités principales du serveur est de permettre l'upload de fichiers directement via le navigateur. Le code ici permet au serveur de recevoir des fichiers multipart et de les écrire sur la carte SD.

1. Réception des fichiers côté serveur

Lorsqu'un utilisateur glisse un fichier dans le gestionnaire web, ce fichier est envoyé au serveur sous forme de requête HTTP POST. Le serveur doit alors découper cette requête, lire le nom du fichier, créer un fichier en écriture sur la carte SD, puis recevoir les morceaux de données pour les écrire au fur et à mesure. Le traitement est fait en mode événementiel.

2. Exemple de code de gestion d'upload

Voici un extrait du début du traitement de l'upload :

```
/*

    Upload.ino

    This script handles file upload to the web-stick
    by default this function require authentication.
    Hence, admin.txt must be set before use

*/

void handleFileUpload(AsyncWebServerRequest *request, String filename, size_t index, uint8_t *data, size_t len, bool final) {
  // make sure authenticated before allowing upload
  if (IsUserAuthed(request)) {
    String logmessage = "";
    //String logmessage = "Client:" + request->client()->remoteIP().toString() + " " + request->url();
    //Serial.println(logmessage);

    //Rewrite the filename if it is too long
    filename = trimFilename(filename);

    //Get the dir to store the file
    String dirToStore = GetPara(request, "dir");
    if (!dirToStore.startsWith("/")) {
      dirToStore = "/" + dirToStore;
    }

    if (!dirToStore.endsWith("/")) {
      dirToStore = dirToStore + "/";
    }
    dirToStore = "/www" + dirToStore;

    if (!index) {
      Serial.println("Selected Upload Dir: " + dirToStore);
      logmessage = "Upload Start: " + String(filename) + " by " + request->client()->remoteIP().toString();
      // open the file on first call and store the file handle in the request object
```

Ce bloc de code montre comment on identifie l'arrivée d'un fichier, comment on initialise le fichier cible sur la carte SD et comment on gère les fragments de données qui arrivent au fur et à mesure. Lorsque la dernière partie est reçue, le fichier est refermé proprement pour garantir l'intégrité des données.

3. Intégration dans le système global

Cette fonction d'upload est appelée via les routes définies dans 'router.ino' et 'server.ino'. Elle s'appuie aussi sur les fonctions internes de gestion de fichiers pour accéder à la carte SD, et sur la logique de vérification d'utilisateur pour restreindre l'accès à cette opération. Sans une session valide (cookie), l'utilisateur ne peut pas uploader de fichiers.

4. Conclusion

Le fichier 'upload.ino' rend possible une fonctionnalité avancée et utile pour l'utilisateur : envoyer des fichiers directement sur le serveur depuis une interface graphique. La gestion des morceaux de données, le contrôle d'erreurs, et l'intégration dans un système embarqué à faible ressource démontrent une compréhension solide du fonctionnement bas niveau des requêtes HTTP multipart.

7 - Analyse du fichier : kvdb.ino

Le fichier 'kvdb.ino' joue un rôle fondamental dans la gestion de la base de données utilisateur. Il s'agit ici d'une base de type clé-valeur (key-value), stockée probablement sur la carte SD ou en mémoire, et utilisée pour stocker les identifiants et les mots de passe (hashés) des utilisateurs autorisés à se connecter au serveur.

1. Chargement de la base utilisateur

Lors de l'initialisation du système, le fichier de base de données est ouvert et lu. Chaque ligne correspond à un utilisateur, avec son nom d'utilisateur et son mot de passe hashé. Le fichier 'kvdb.ino' contient les fonctions qui lisent ce fichier, stockent les paires dans une structure accessible, et permettent ensuite de valider les connexions.

2. Exemple de code : lecture des utilisateurs

L'extrait suivant montre comment sont traitées les entrées de la base de données :

```
/*

   Key Value database

   This is a file system based database
   that uses foldername as table name,
   filename as key and content as value

   Folder name and filename are limited to
   5 characters as SDFS requirements.
*/

//Root of the db on SD card, **must have tailing slash**
const String DB_root = "/db/";

//Clean the input for any input string
String DBCleanInput(const String& inputString) {
  String trimmedString = inputString;
  //Replace all the slash that might breaks the file system
  trimmedString.replace("/", "");
  //Trim off the space before and after the string
  trimmedString.trim();
  return trimmedString;
}

//Database init create all the required table for basic system operations
void DBInit() {
  DBNewTable("auth");
  DBNewTable("pref");
}

//Create a new Database table
void DBNewTable(String tableName) {
  tableName = DBCleanInput(tableName);
  if (!SD.exists(DB_root + tableName)) {
```

On observe ici que chaque utilisateur est stocké avec une clé (nom) et une valeur (hash du mot de passe). Ce type de structure permet une vérification rapide lors de la tentative de connexion. Si un nom d'utilisateur correspond à une clé et que le hash correspond, l'accès est autorisé.

3. Intégration avec les autres fichiers

Le fichier 'api.ino' fait directement appel à ces fonctions pour valider les connexions. C'est ici que les données de connexion sont vérifiées. Le fichier peut aussi permettre d'ajouter ou de supprimer des utilisateurs si la logique est prévue pour cela, même si cela dépend de l'implémentation exacte dans le code.

4. Conclusion

Même s'il est discret, ce fichier est indispensable : sans lui, aucun contrôle d'accès ne serait possible. Il agit comme un petit système d'authentification embarqué, autonome, simple mais efficace.

8 - Analyse du fichier : internal.ino

Le fichier 'internal.ino' regroupe différentes fonctions internes utilisées par le serveur pour effectuer des opérations bas niveau. Ces fonctions ne sont pas directement appelées par l'utilisateur via l'interface web, mais elles sont essentielles au bon fonctionnement global. Elles permettent par exemple de gérer le stockage, d'initier des vérifications système ou de manipuler des chemins de fichiers.

1. Fonctions utilitaires internes

Le contenu de ce fichier est constitué de fonctions de support. Elles prennent en charge des tâches comme la normalisation des chemins, la vérification de permissions ou encore la gestion de la structure des répertoires. C'est un peu la boîte à outils du serveur, sur laquelle s'appuient les autres modules pour faire leur travail.

2. Exemple de fonctions internes

Voici un extrait des premières fonctions internes définies dans le projet :

```
/*

    WiFi Setup Functions

    To setup WiFi on your Web-stick
    put a file in your SD card with filename /cfg/wifi.txt
    In the text file, write two lines which containing the
    WiFi ssid and password, seperated by a linux new linx \n
    as follows:

    [WIFI SSID]
    [WiFi Password]
*/
String loadWiFiInfoFromSD() {
  if (SD.exists("/cfg/wifi.txt")) {
    String fileContent = "";
    File configFile = SD.open("/cfg/wifi.txt", FILE_READ);

    if (configFile) {
      while (configFile.available()) {
        fileContent += char(configFile.read());
      }
      configFile.close();
    }

    return fileContent;
  }

  return "";
}

void splitWiFiFileContent(const String& fileContent, String& ssid, String& password) {
  int newlineIndex = fileContent.indexOf('\n');
  if (newlineIndex != -1) {
    ssid = fileContent.substring(0, newlineIndex);
```

Ces fonctions peuvent être appelées en interne pour vérifier l'existence d'un fichier ou d'un répertoire, ou encore pour reconstruire le chemin correct d'un fichier à partir des données de requête. On y trouve aussi potentiellement des vérifications de sécurité, comme la prévention des accès non autorisés à des dossiers sensibles.

3. Interaction avec les autres composants

Ce fichier est utilisé par les modules 'router.ino' et 'upload.ino', entre autres, pour exécuter les opérations système critiques. Plutôt que de dupliquer le code de vérification ou de manipulation de fichiers, ces fonctions sont centralisées ici. Cela rend le projet plus modulaire, plus clair et plus facile à maintenir.

4. Conclusion

Le fichier 'internal.ino' est un support discret mais essentiel à la stabilité du serveur. Il regroupe tout ce qui concerne les interactions internes avec le système de fichiers et l'environnement de l'ESP32. Il permet d'assurer que les fonctions exposées aux utilisateurs soient exécutées de manière fiable et sécurisée.

9 - Analyse du fichier : utils.ino

Le fichier 'utils.ino' contient des fonctions utilitaires générales utilisées un peu partout dans le projet. Ces fonctions sont souvent simples mais très pratiques, car elles évitent de répéter du code et permettent de centraliser certains traitements communs.

1. Fonctions définies dans ce fichier

On y trouve des fonctions qui peuvent servir à traiter des chaînes de caractères, à convertir des tailles de fichiers en format lisible, ou à exécuter des opérations spécifiques sur les chemins ou les extensions. C'est un ensemble de briques de base qui renforcent la lisibilité et la maintenabilité du code global.

2. Exemple de fonctions utilitaires

Voici un extrait des fonctions disponibles dans ce fichier :

```
/*

    Utilities

*/


void prettyPrintRequest(AsyncWebServerRequest* request) {
  if (request->method() == HTTP_GET)
    Serial.printf("GET");
  else if (request->method() == HTTP_POST)
    Serial.printf("POST");
  else if (request->method() == HTTP_DELETE)
    Serial.printf("DELETE");
  else if (request->method() == HTTP_PUT)
    Serial.printf("PUT");
  else if (request->method() == HTTP_PATCH)
    Serial.printf("PATCH");
  else if (request->method() == HTTP_HEAD)
    Serial.printf("HEAD");
  else if (request->method() == HTTP_OPTIONS)
    Serial.printf("OPTIONS");
  else
    Serial.printf("UNKNOWN");
  Serial.printf(" http:

  if (request->contentLength()) {
    Serial.printf("_CONTENT_TYPE: %s\n", request->contentType().c_str());
    Serial.printf("_CONTENT_LENGTH: %u\n", request->contentLength());
  }

  int headers = request->headers();
  int i;
  for (i = 0; i < headers; i++) {
    AsyncWebHeader* h = request->getHeader(i);
```

Dans cet exemple, les fonctions prennent en charge des conversions et des vérifications simples mais utiles, comme le formatage d'une taille de fichier pour l'affichage, ou la validation de noms de fichiers.

3. Rôle dans le projet global

Ces fonctions sont appelées depuis plusieurs autres fichiers comme 'upload.ino', 'fs.html' via l'API, ou même 'router.ino'. En regroupant les utilitaires ici, le développeur gagne en clarté et peut facilement les modifier sans impacter trop de parties du projet.

4. Conclusion

Le fichier 'utils.ino' représente une bonne pratique de programmation modulaire. Il évite la duplication de code et permet de mieux organiser les tâches communes. Même s'il n'est pas visible directement dans les fonctionnalités du serveur, il contribue fortement à sa robustesse et à sa facilité de maintenance.

Conclusion du projet

Ce projet avait pour but de créer un serveur web embarqué capable d'héberger des fichiers et de proposer une interface d'administration complète, le tout en s'appuyant sur un ESP32 et une carte SD. L'objectif principal était de proposer une solution légère, autonome et fonctionnelle, capable de remplacer des solutions plus lourdes comme un Raspberry Pi pour des usages simples.

Problèmes rencontrés

Plusieurs difficultés ont été rencontrées au cours du développement. La première concerne l'implémentation du protocole HTTPS (SSL). Bien que des tests aient été effectués avec des certificats, les résultats se sont révélés instables et inutilisables dans un environnement réel. La seconde difficulté est liée à la gestion des sites dynamiques, avec des pages qui doivent charger des contenus ou interagir avec l'utilisateur sans recharger toute la page. Cela nécessite une architecture plus avancée que celle disponible sur un microcontrôleur, ce qui a limité les possibilités.

Un autre problème important a été la gestion des cookies liés à l'authentification. Certains navigateurs, comme Firefox, bloquaient les cookies générés par le serveur, empêchant ainsi la validation de session et l'accès aux parties protégées comme le panneau d'administration. Ce comportement entraînait des erreurs de connexion difficiles à diagnostiquer. La solution trouvée a été de consulter manuellement les cookies générés dans l'inspecteur du navigateur (onglet stockage > cookies), puis de les recopier ou modifier à la main pour simuler une session active. Ce contournement a permis de tester et valider le système malgré les restrictions imposées par certains navigateurs.

Résolution marquante

Un des moments clés du projet a été la réussite de l'envoi et du téléchargement de fichiers. La fonctionnalité d'upload, notamment, a demandé un gros travail de débogage, car elle touche à la fois au front-end web, à la gestion du protocole HTTP, et à l'écriture physique sur la carte SD. Après plusieurs essais, l'ensemble a été corrigé et fonctionne maintenant de manière fluide, ce qui représente une réussite personnelle importante.

Bilan personnel

Ce projet m'a permis de comprendre comment fonctionne un vrai serveur web, mais aussi de découvrir les limites du matériel embarqué comme l'ESP32. J'ai appris à structurer un projet sur plusieurs fichiers, à utiliser des routes API, à gérer des sessions avec des cookies, et à intégrer du HTML/CSS/JS avec du C++. Même si certaines parties sont restées incomplètes, la majorité des fonctionnalités ont été implémentées avec succès. Ce projet représente donc une expérience riche, concrète et formatrice.