Rendu IoT

PELLETIER-MULSANT Nicolas
MAGOUS Baptiste

Lien Github Arduino: https://github.com/NicolasPelletierMulsant/IoT_Arduino
Lien Github API: https://github.com/NicolasPelletierMulsant/IoT_API
Lien Github Android: https://github.com/BaptisteMagous/SmartLightsAndroid

Notre projet est un ensemble de lumières d'ambiance intelligentes pour décorer l'intérieur d'une maison. Les lumières, discrètes, s'adaptent à toutes les situations et réagissent au bruit et à la lumière naturelle. Lorsque l'ambiance est calme, les lumières seront douces. Et durant les soirées mouvementées, les lumières seront plus vives.

Phase 1

Liste du matériel

- Kit Feather HUZZAH32
- LEDS
- Capteur Son
- Capteur de Présence
- Capteur luminosité

Technologies

- C++ / Arduino
- FreeRTOS

Scénarios

Des personnes mangent le soir. Les ampoules de la salle à manger sont réglées pour s'allumer surtout avec le détecteur de présence. Lorsque toutes les personnes quittent la pièce, les lumières s'éteignent.
Les ampoules du salon sont réglées de la même façon. Mais lorsque le salon est vide, les lumières ne s'éteignent pas totalement, pour ne pas être plongé dans la pénombre. Après 22h, la pièce s'éteint s' il n'y a plus de présence.
Si l'utilisateur le souhaite, l'intensité des ampoules peut être adaptée en fonction de la luminosité extérieure. Par exemple, en pleine journée, l'intensité des ampoules sera plus faible qu'en fin de journée.

Des personnes regardent un film, le capteur détecte peu de mouvement mais du son. Les ampoules sont réglées sur l'ambiance film, ce qui signifie que les lumières vont s'adapter en fonction du son pour donner la meilleure expérience. Dans cette configuration, les ampoules pourront potentiellement varier de couleurs.

Diagramme déploiement

https://lucid.app/lucidchart/9f17564b-d910-40ad-83b5-be609219c7d2/edit?invitationId=inv_5ddbfe87-842f-478a-943f-87ad7f61a3d0

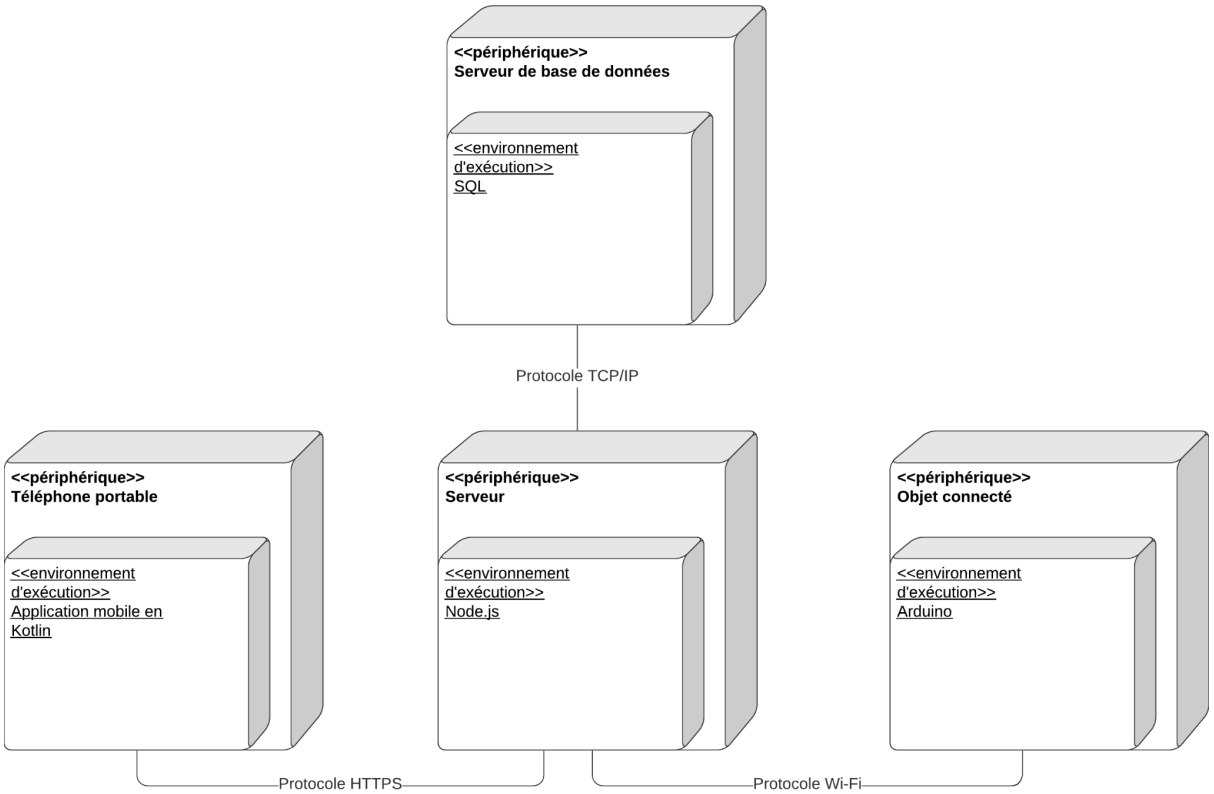
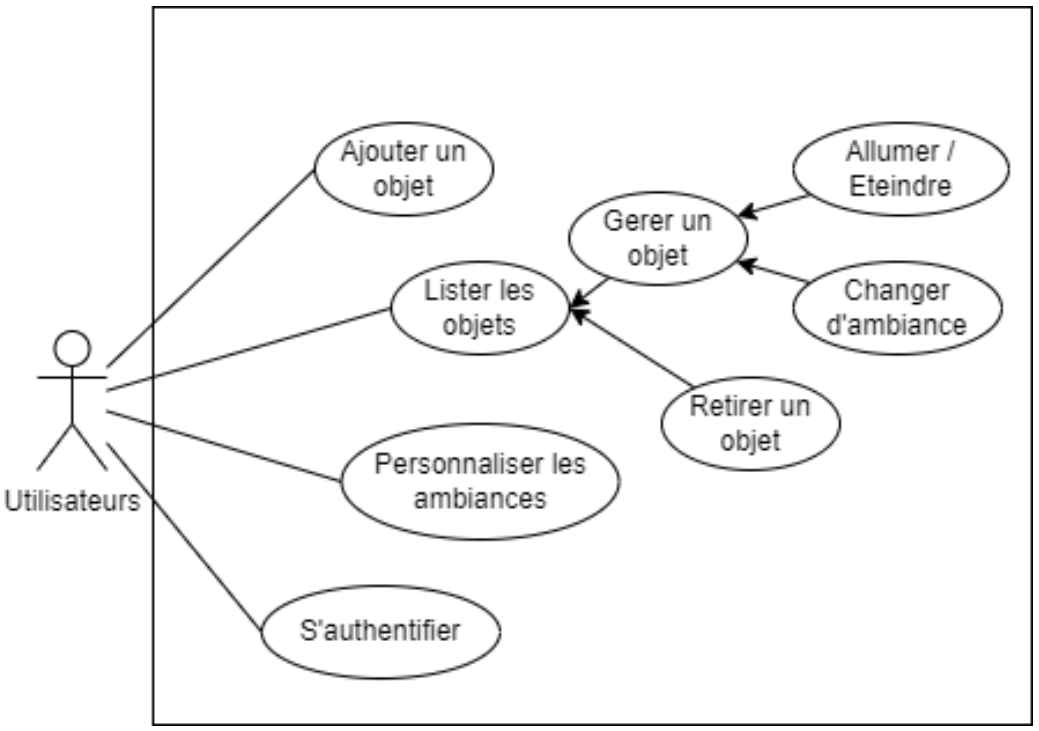
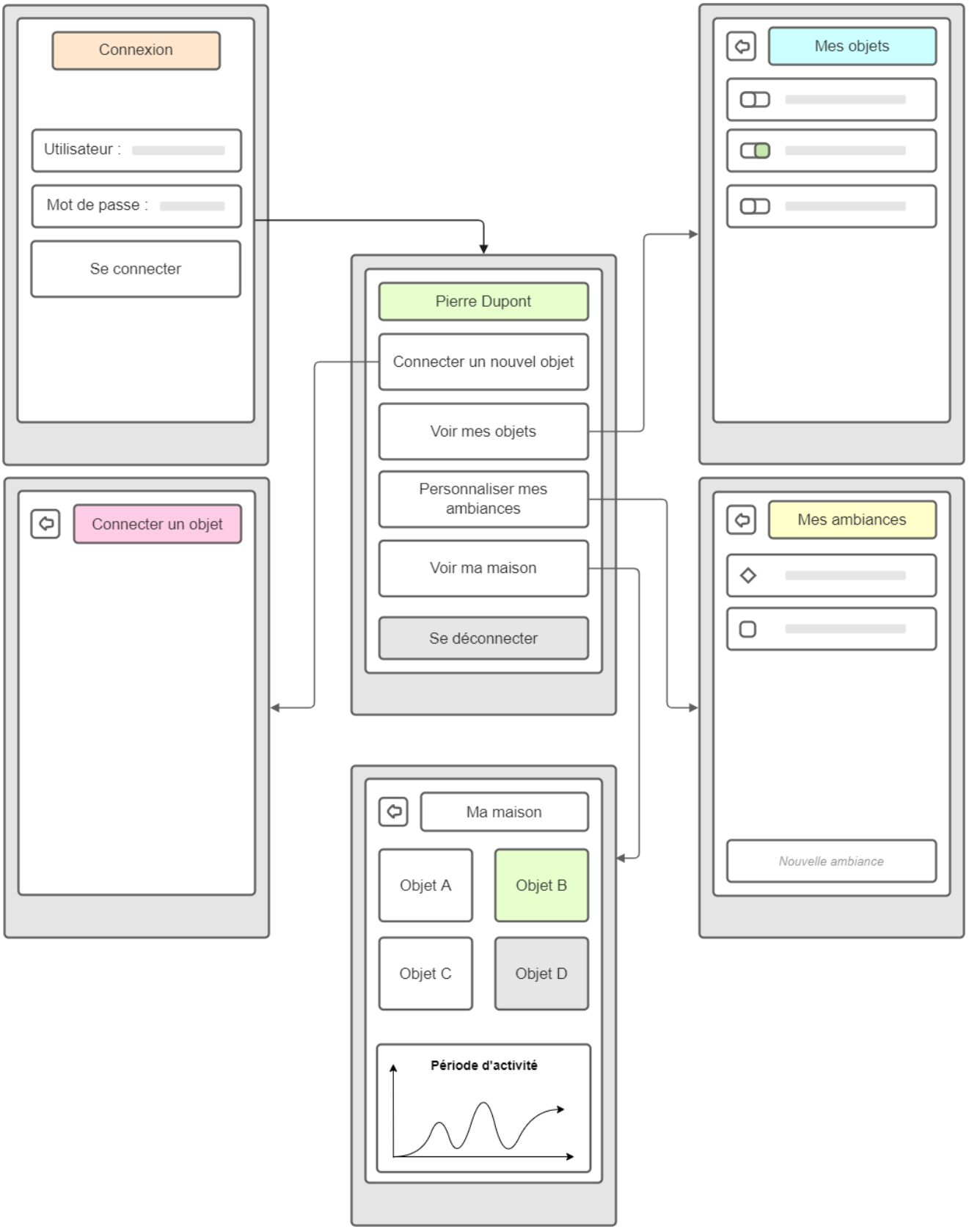


Diagramme cas d’utilisation



Maquette de l’application mobile

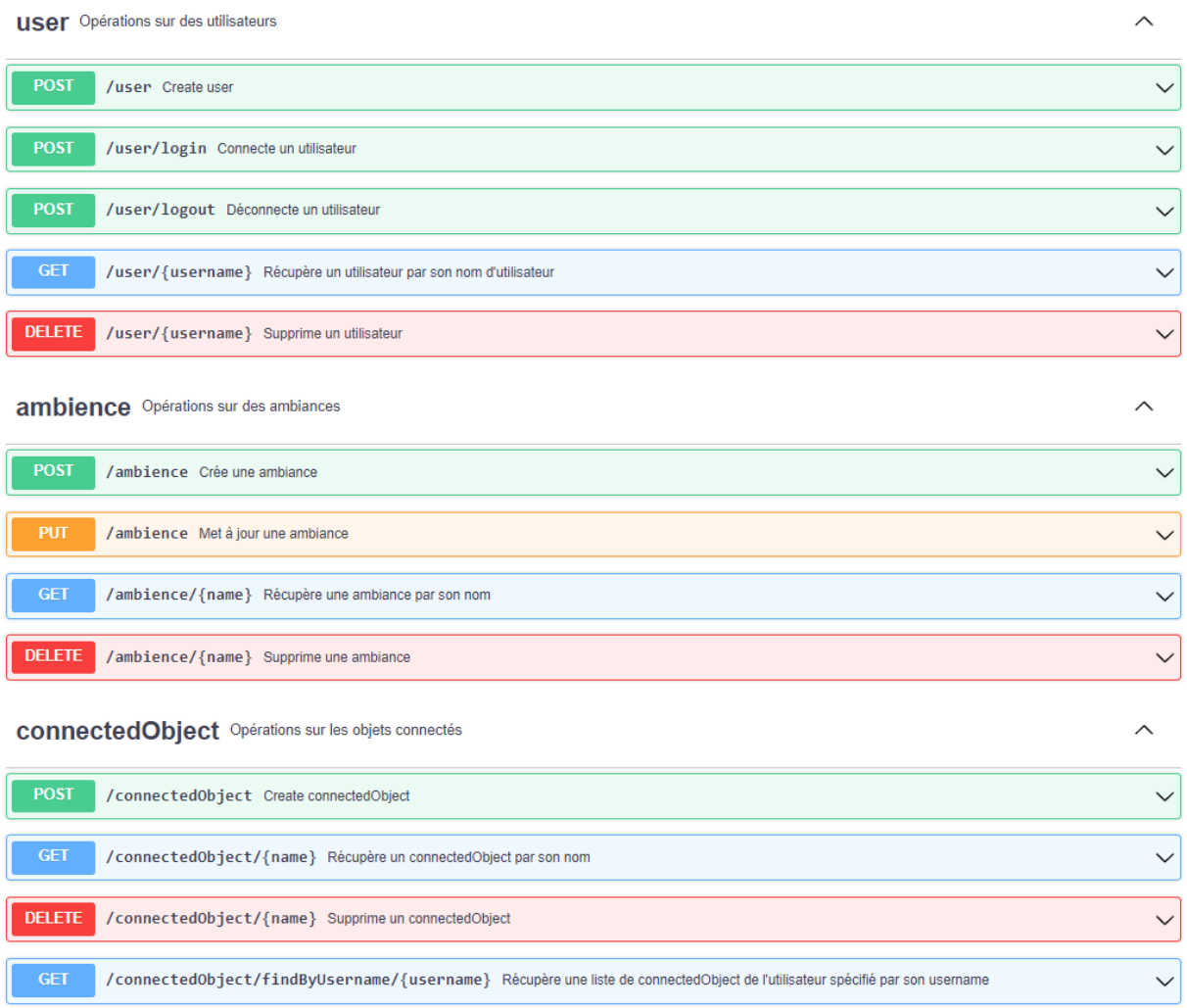


API

Pour la définition de l'API des Web Services, nous avons décidé de la manière dont allaient être échangés les données et du format de celles-ci.
Nous avons utilisé Swagger Editor pour créer une documentation de qualité et lisible.

Pour voir l'ensemble de la documentation, il suffit de se rendre sur https://editor.swagger.io/ et d'importer le fichier swagger.yaml qui se trouve dans le dossier Postman de notre repository Github.

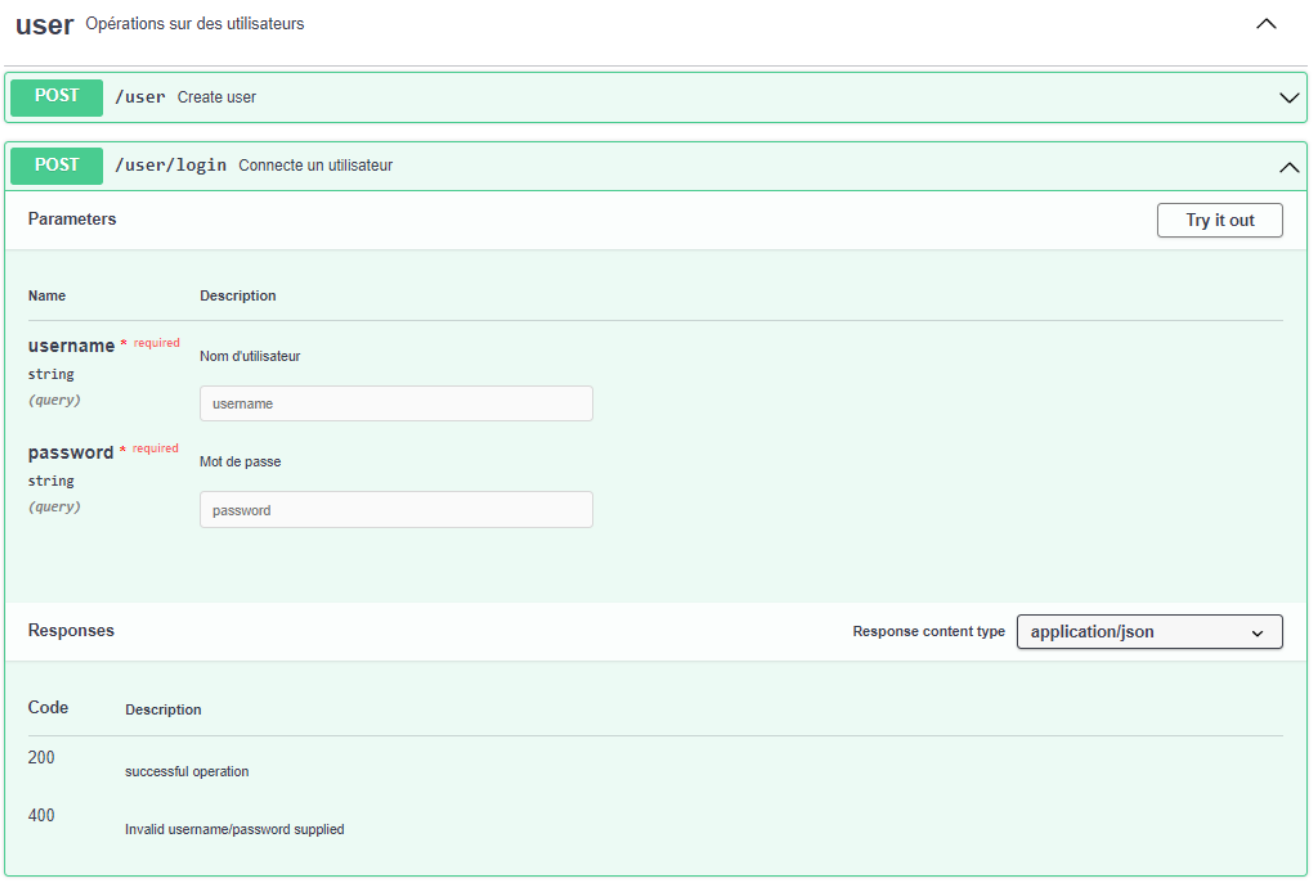
Ci-après, une capture d'écran qui donne une idée générale du fonctionnement de notre API :



On retrouve trois catégories principales:

- User: Pour gérer l'ensemble des requêtes liées aux utilisateurs de l'application
- Ambience: Pour gérer les ambiances qui seront utilisées par notre objet connecté et qui seront modifiables depuis l'application
- ConnectedObject: Pour gérer un objet connecté, ce qui permet à un utilisateur d'avoir plusieurs objets connectés et de les gérer indépendamment

Finalement, un exemple d'une route un peu plus détaillée qu'on peut retrouver sur l'éditeur en ligne :



Le format de données utilisé dans l'ensemble de l'API est le format JSON, que ce soit pour les données renvoyées par l'API ou les données que l'API reçoit en entrée des requêtes POST.

Quant à l'implémentation de l'API, nous avons choisi d'utiliser Node JS avec la librairie Express pour créer le serveur. Les différentes catégories de route (User, Ambiance, ConnectedObject) sont réparties dans plusieurs dossiers pour en faciliter la maintenance.

Pour tester les différentes routes de l'API, nous avons utilisé le logiciel Postman, dans lequel nous avons défini une collection de requêtes contenant toutes les routes implémentées dans l'API.
Pour utiliser la collection, il suffit d'ouvrir le logiciel Postman, d'importer la collection (qui est stockée au format JSON dans le dossier Postman de notre repository Github) puis de choisir la requête à tester.

Objet connecté (FreeRTOS)

Pour la mise en place des différentes fonctionnalités de l'objet connecté, nous avons utilisé le noyau FreeRTOS. Le fonctionnement de celui-ci est simple, il faut répartir les différentes fonctionnalités en "tâches" qui seront appelées les unes après les autres.

Pour la communication entre l'objet et l'extérieur, nous avons choisi d'utiliser un protocole de communication par WiFi.

```
WiFi.begin(username, password);
while (WiFi.status() != WL_CONNECTED) {
  Serial.println("Trying to connect to WiFi");
  delay(500);
}
Serial.print("Connected to WiFi ! IP : ");
Serial.println(WiFi.localIP());

server.begin();
```

Finalement, nous avons choisi de répartir le fonctionnement de l'application en deux tâches principales.

```
xTaskCreate( TaskUpdateLight,  "Update lights",  2048, NULL, 2, &Task_Handle1);
xTaskCreate( TaskListenToServer,  "Listen to server",  2048, NULL, 4, &Task_Handle2);
```

La première tâche nommée "Update Lights" permettra comme son nom l'indique de gérer l'ensemble des fonctionnalités relatives aux lumières. Cela passe notamment par la vérification de différentes conditions qui indiqueront à l'objet si il doit changer de "State" et donc d'appliquer de nouvelles couleurs et intensités aux LEDs.

```
//If ALL conditions are met, we shall use that state
if (conditionMet) {
  float colorToUse = 0.0f;
  //For each possible factors
  for (int k = 0; k < 3; k++) {
    switch (states[i].factors[k].type) {
      case LUMINOSITY:
        colorToUse += luminosity * states[i].factors[k].value;
        break;
      case SOUND:
        if (noise)
          colorToUse += states[i].factors[k].value;
        break;
      case PRESENCE:
        if (presence) {
          colorToUse += states[i].factors[k].value;
        }
        break;
    }
  }
```

La seconde tâche nommée Listen to server, permet d'agir comme un serveur distant et donc de recevoir des requêtes provenant de l'extérieur. C'est notamment à l'aide de cette tâche que l'objet est capable de recevoir des requêtes provenant du serveur et par extension, de l'application Android.

```
while (client.connected()) {
  if (client.available()) {             // if there's bytes to read from the client,
    char ch = client.read();             // read a byte, then
    Serial.write(ch);
    data += ch;

    if (ch == '{' && !jsonStarted) jsonStarted = true;
    if (jsonStarted) JSONInput += ch;
  }
```

Exemple de réception d'une requête extérieure

L'objet peut ensuite effectuer différentes actions en fonction de la requête reçue, comme par exemple, modifier l'ambiance actuelle à l'aide de celle fournie dans la requête.

```
if (String(action) == "setAmbiance") {
  changeAmbiance();
}
```

```
void changeAmbiance() {
  JsonArray newStates = JSONDocument["states"].as<JsonArray>();
  states[0] = emptyState;
  states[1] = emptyState;
  states[2] = emptyState;

  int stateI = 0;
  for (JsonObject newState : newStates) {
    Factor newFactors[3] = {Factor(NONE, 0.0f), Factor(NONE, 0.0f), Factor(NONE, 0.0f)};
    Condition newConditions[3] = {Condition(NONE, EQUAL, 0.0f), Condition(NONE, EQUAL, 0.0f), Condition(NONE, EQUAL, 0.0f)};

    int factorI = 0;
    for (JsonObject newFactor : newState["factors"].as<JsonArray>()) {
      newFactors[factorI] = Factor(typeToEnum(newFactor["type"]), newFactor["value"]);
      factorI++;
    }
    int conditionI = 0;
    for (JsonObject newCondition : newState["conditions"].as<JsonArray>()) {
      newConditions[conditionI] = Condition(typeToEnum(newCondition["type"]), operatorToEnum(newCondition["operator"]), newCondition["value"]
      conditionI++;
    }

    states[stateI] = State(newState["name"], newState["colors"][0], newState["colors"][1], newFactors, newConditions);
    stateI++;
  }
}
```

Il est également capable de répondre à des requêtes pour envoyer des informations aux utilisateurs, on peut par exemple demander à l'objet de nous envoyer l'état de ses différents capteurs.

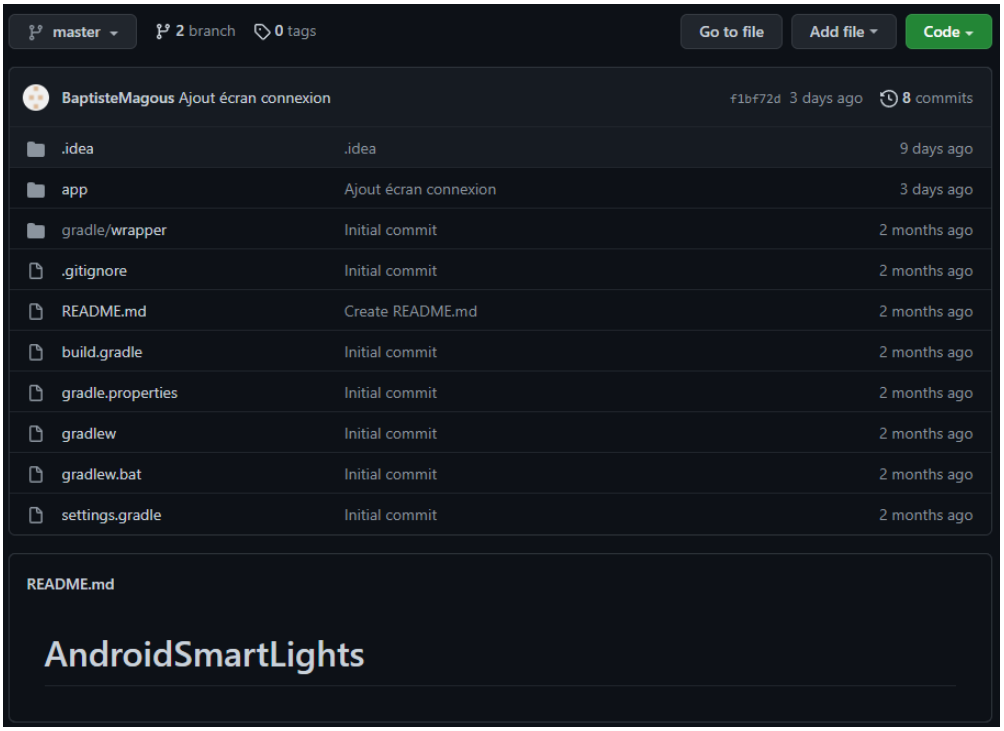
```
client.println("HTTP/1.1 200 OK");
client.println("");
if (String(action) == "getData") {
  serializeJson(storedDatas, client);
  setupStoredDatasDocument();
}
```

Tout ce qui a été montré à présent est un résumé de l'application, le code complet permettant de faire fonctionner l'objet connecté étant disponible sur notre repository GitHub.

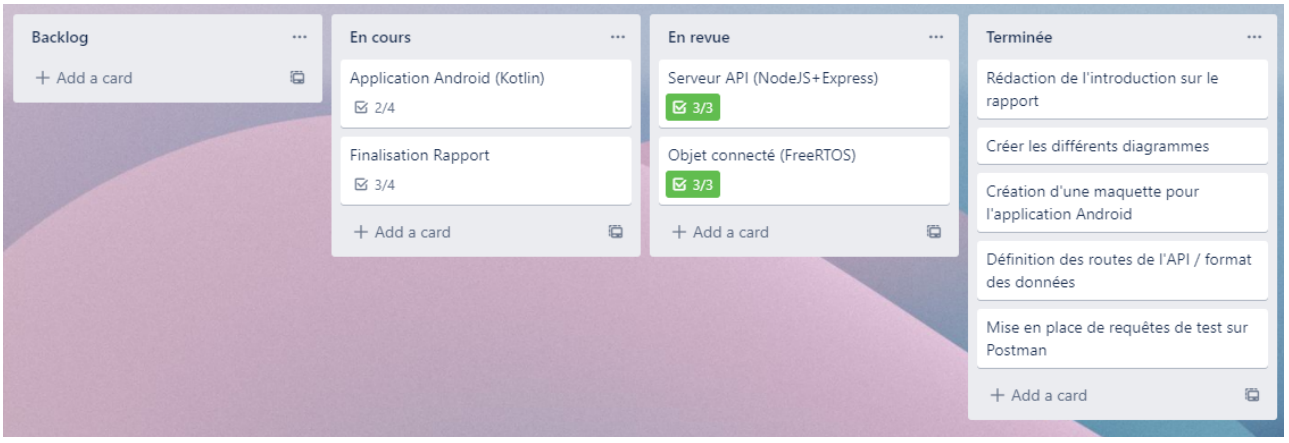
Outils Collaboratifs

Pour faciliter la collaboration, nous avons utilisé plusieurs outils pour partager les différentes tâches à effectuer et le code de nos différents services.

Pour le gestionnaire de version, nous avons utilisé l'outil GitHub.



Pour visualiser et répartir efficacement les tâches, nous avons utilisé Trello.



Finalement, nous avons utilisé le site LucidChart pour créer nos différents diagrammes de manière collaborative.

Bilan et Conclusion

Finalement, nous avons réussi à obtenir le résultat que nous souhaitions.
Nous avons réussi à mettre en place une API fonctionnelle et à faire communiquer l'application Android et l'objet connecté avec l'API. L'application prend en charge la gestion des utilisateurs et permet de modifier le fonctionnement de l'objet connecté.

Ce projet nous a permis d'approfondir nos connaissances tant au niveau théorique que pratique. Nous avons pu mettre en place une vraie application Android en utilisant une architecture qui a fait ses preuves.
De plus, nous avons pu voir comment il était possible de faire communiquer un objet connecté avec un serveur distant en créant notre propre API.

Malgré tout, notre projet était plutôt une Proof Of Concept qu'un réel produit livrable au grand public. Il serait donc intéressant d'étoffer l'application avec plus de fonctionnalités et de mettre en place une interface graphique plus ergonomique pour l'utilisateur. Il serait également intéressant avec des objets connectés de plus grandes tailles, on pourrait penser à de vraies lumières de taille réelle.