Architecture REST et architecture orientée événements (EDA)

Simulation d'un bureau de travail connecté, communicant avec un serveur utilisant l'architecture REST et orientée événements.

Travail de Bachelor

Lazar Randjelovic

Août 2023

Supervisé par :

Prof. Dr. Jacques Pasquier-Rocha
Software Engineering Group

UNI FR
UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Groupe Génie Logiciel
Département d'Informatique
Université de Fribourg (Suisse)

Software Engineering Group

Remerciements

Je remercie le professeur Dr. Jacques Pasquier-Rocha pour sa disponibilité et de m'avoir encadré tout au long de ce projet. Il m'a permis de développer un projet complet, grâce à ses précieux conseils et sa bonne supervision.

Table des matières

Liste des figures

Liste des codes source

1 Introduction

1.1. Motivation et objectifs

Motivation

L'architecture Representational state transfer (REST) est la structure de base pour la conception de services sur internet. La compréhension et la maîtrise de ce type d'architecture s'impose comme un concept indispensable pour tous les programmeurs s'intéressant à la programmation web. Toutefois, de nos jours, la conception des services a beaucoup évolué. De plus en plus de services ou de processus se déclenchent lorsqu'un événement ou une action spécifique a été effectué. Les services fonctionnant de cette manière utilisent l'architecture orientée événements. Ce type d'architecture devient de plus en plus populaire et inonde peu à peu notre quotidien, notamment par le biais des objets connectés ou des maisons connectés qui s'installent progressivement dans nos modes de vie. Comprendre ces deux types d'architectures était pour moi essentiel et sont des concepts clés de la programmation web. Avec ce travail je voulais étudier ces différentes architectures et essayer d'intégrer ces deux modèles au sein d'un même projet.

Objectifs

L'objectif de ce travail est donc de concevoir une serveur web pouvant à la fois répondre aux différentes requêtes que l'on fait habituellement sur un serveur REST, ainsi qu'un serveur qui soit capable de réagir à différents événements en direct et d'effectuer les actions appropriées pour chaque événement. En somme, concevoir un serveur web qui répond aussi bien à l'architecture orientées événements, qu'à l'architecture REST.

1.2. Organisation

Chapitre 1 : Introduction

Discussion des motivations et des différents objectifs.

Chapitre 2 : Architecture REST et architecture orientée événements

Ce chapitre décrit de manière plus théorique et détaillée les deux types d'architectures, ainsi que leur utilisation.

Chapitre 3 : Présentation du projet

Dans ce chapitre, différents scénarios sont présentés, afin de montrer l'utilisation globale du projet, les différentes fonctionnalités et les possibilités qu'il apporte.

Chapitre 4 : Détail de programmation

Présentation du serveur, de l'application mobile et du programme simulant le bureau connecté. Explication des différents points clés de chaque implémentation.

Chapitre 5 : Conclusion

Synthèse du projet, discussion des points pouvant être améliorés et conclusion personnelle.

Appendix

Contient les abréviations et les références utilisés pour ce travail.

1.3. Notations et Conventions

— Abréviations et acronymes :

  — Les acronymes et les abréviations sont définis comme suit : JavaScript Object Notation (JSON), pour le premier usage, puis JSON pour les utilisations suivantes.

— Le masculin est utilisé pour représenter le neutre dans le texte pour des raisons de facilité.

2

Architecture REST et architecture orientée événements

2.1. REST : Representational state transfer

2.1.1. Architecture

Le Representational state transfer (REST) a été défini pour la première fois par Roy Fielding, lors de sa thèse de doctorat *Architectural Styles and the Design of Network-based Software Architectures* [1], en 2000. Le principe REST propose différentes règles à utiliser pour créer des services web. Les services REST permettent aux utilisateurs de manipuler des ressources web en formulant des requêtes prédéfinies. Pour qu'une application puisse être considérée comme REST, elle doit respecter une composition architecturale spécifique :

- **Interface uniforme** : L'idée de disposer d'une interface uniforme entre tous les composants du réseau, permet un transfert standardisé des informations. Deux contraintes principales définissent cette interface uniforme :
  - **Identification des ressources dans les requêtes :** Toutes les ressources sont identifiables de manière unique. Par exemple via un URI pour les systèmes basés sur le web.
  - **Manipulation des ressources par des représentations :** La ressource retournée au client, donne suffisamment d'informations au client pour modifier ou supprimer la ressource
- **Architecture Client-Serveur** : L'architecture doit se composer d'un client, d'un serveur et de ressources. Les responsabilités doivent être séparées entre le client et le serveur
- **Communication sans état ou "stateless"** : Le serveur ne conserve aucun état concernant les différents clients. Chaque paire "requête-réponse" est considérée comme une transaction indépendante, sans lien avec les requêtes précédentes ou futures
- **Mise en cache** : Les données doivent pouvoir être mises en mémoire cache ou non. Les ressources pouvant être mises en cache peuvent servir d'intermédiaire entre les serveurs et les clients, en réutilisant les réponses des requêtes précédentes, évitant ainsi de répéter ou effectuer de multiples requêtes inutilement.

— **Système de couche** : Lorsqu'un client effectue une requête, il ne doit pas pouvoir savoir s'il communique avec un serveur intermédiaire ou le serveur final. La réponse de la requête est retournée par un ensemble de couches indépendantes et communiquant entre elles.

Les applications utilisant une architecture REST, contraignent la façon dont les serveurs répondent et traitent les requêtes des clients ; étant donné qu'ils respectent une architecture précise. Cependant, ces contraintes architecturales offrent beaucoup d'avantages et notamment une grande extensibilité, qui supporte un grand nombre de composants et d'interactions entre les composant du réseau. REST avec sa composition établit une distinction claire entre la notion de composants et leurs interactions, la notion de client-serveur et la notion de ressources. De ce fait, le style REST s'est rapidement imposé comme l'architecture par défaut pour la création de services sur le web et principalement pour la création d'interface de programmation d'application (API).

2.1.2. Pourquoi REST et son application dans les services web

Roy Fielding mentionne dans sa thèse de doctorat[1] que le "World Wide Web" (WWW) possède des fondations solides et son design assez "simple" basé sur l'"Hypertext Transfer Protocol" (HTTP), permet au plus grand nombre d'utiliser le World Wide Web de manière intuitive. La force du protocole HTTP réside dans le fait que les utilisateurs peuvent naviguer de pages web en pages web, simplement en cliquant sur des liens. Toutefois, Roy Fielding constate qu'il n'existe pas de réelle architecture définie pour ce nouveau protocole d'Internet. Ainsi, naît l'idée de construire une architecture standardisée qui permettrait aux développeurs web de créer facilement des services web. Le "World Wide Web" est donc une immense multitude de réseaux communiquant les uns avec les autres. Ces réseaux hébergent soit des ressources (textes, images, videos, ...) et sont appelés serveurs ou sont soit des clients à la recherche de ces ressources. Les clients effectuent des requêtes aux serveurs possédant la ressource qu'ils convoitent et attendent une réponse. Cette réponse peut être positive : la ressource est retournée ou négative : la ressource ne peut pas être retournée. Toutes les ressources sur Internet possèdent une adresse spécifique permettant des les identifier. Cette adresse est plus communément appelée "Universal Resource Identifiers" (URI) et contient la représentation de la ressource demandée.

L'avantage de REST dans le développement d'API web, au-delà de son architecture, est qu'il utilise uniquement les méthodes HTTP, qui sont limitées et prédéfinies. Ceci permet une communication simplifiée entre le client et le serveur. Ces méthodes HTTP permettent d'indiquer à un serveur quelle requête est demandée par le client et de ce fait, l'action appropriée à exécuter pour le serveur. Les différentes méthodes HTTP sont respectivement :

— **GET** : Récupère une représentation de la ressource demandée

— **POST** : Crée une ressource en utilisant les paramètres contenus dans le corps de la requête.

— **PUT** : Remplace une ressource existante avec les paramètres contenus dans le corps de la requête (ou créer une ressource si elle n'existe pas).

— **PATCH** : Met à jour une ressource existante

— **DELETE** : Supprime toutes les représentations de la ressource existante

En conclusion, une application web fonctionnant avec les principes REST, est une application qui respecte l'architecture REST, qui utilisent le fonctionnement de "requête-réponse" pour le client et le serveur via les méthodes définies par HTTP et qui peut identifier toutes les ressources via des URIs.

2.2. EDA : Event-Driven architectures ou Architecture orientée événements

2.2.1. Introduction

Lors du chapitre précédant, nous avons vu que les APIs utilisant les principes REST, fonctionnaient entre autre sur le principe client-serveur, où les clients effectuent des requêtes aux serveurs et attendent une réponse à leurs demandes. Pour l'"Event-Drivent architecture" (EDA) ou l'architecture orientée événements en français, la notion de client et de serveur est également présente. Toutefois la relation entre ces deux entités est totalement différente et ses contraintes architecturales diffèrent grandement de ce que peut proposer REST. L'idée principale derrière l'EDA est que les serveurs n'attendent plus qu'un client fasse une requête pour effectuer une action, mais attendent qu'un changement d'état se réalise pour exécuter différentes opérations. La conception d'application sur Internet a beaucoup évolué depuis les premières APIs REST. Malgré que la grande majorité des APIs reposent encore sur les principes REST, de plus en plus d'applications nécessitent de transmettre des informations en temps réel et ne peuvent attendre qu'un client fasse une requête spécifique pour leur bon fonctionnement. En revanche, comme les clients sont informés en direct des différents événements par le serveur, ils ont la possibilité d'agir en temps réel. par exemple, lorsque nous faisons nos courses et que nous utilisons les "auto-scanneurs", nous sommes informés tout de suite de ce qu'il y a dans notre chariot de courses, et cela sans que nous ayons besoin d'effectuer une requête à un quelconque serveur. Plus récemment, si notre maison est équipée de capteurs, nous pouvons savoir en direct, si un incendie se déclenche ou si une panne se produit quelque part et agir en conséquence sans avoir besoin d'être physiquement présent.

2.2.2. Architecture

Les applications utilisant une architecture orientée évènements, reposent principalement sur la relation "client-serveur". Contrairement à une architecture REST qui impose plusieurs contraintes précises, les APIs EDA définissent seulement le comportement du client et du serveur. Une autre différence majeure est qu'HTTP n'est pas nécessairement le protocole principal, d'autres protocoles peuvent être utilisés pour construire une API EDA. Le serveur et le client communiquent par échanges de messages, et non pas par la formulation de requêtes. De ce principe, l'architecture orientée événements se compose donc d'un serveur, d'un producteur, de souscripteurs (ou consommateurs) et fonctionne avec le principe "publish/subscribe", soit publier/souscrire en français.
Structure :

— **Producteur** : Le producteur ou "producer" en anglais, se comporte comme le composant du système qui est capable de sentir ou recevoir des événements en temps réel et de les publier aux différents consommateurs. Un événement décrit souvent le

changement ou la modification d'un état. Le producteur peut donc être vu comme un réceptacle pour les différents événements, capable de les transmettre sous forme de messages. Comme le producteur communique avec les clients uniquement par messages, l'architecture orientée événements porte parfois le nom d'architecture orientée messages (message-driven architecture).

— **Serveur ou broker** : Le serveur, plus souvent nommé broker, joue un rôle important dans le transmission des messages (asynchrones). C'est lui qui établit le lien entre les producteurs et les consommateurs. Le travaille du broker est de recevoir les messages du producteur et des les publier aux bons endroits, représentés par des "channels", afin que les différents clients puisse les recevoir. Le serveur publie les messages grâce à l'aide du protocole définit par l'API EDA. La figure 2.1 présente un schéma simplifié du fonctionnement du broker.

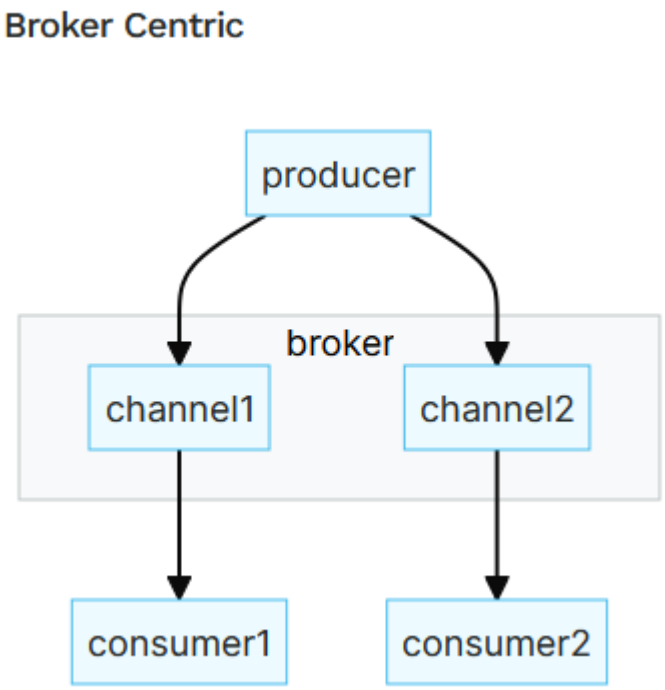


Figure 2.1. – EDA broker schema[6]

— **Channel** : Pour réceptionner les messages publiés par le producteur, les clients doivent souscrire à des "channels" (ou cannaux) pour pouvoir obtenir les messages. Les channels sont un élément essentiel de l'architecture orientée évènements. En effet, le fonctionnement de cette architecture se fait via des messages publiés par les producteurs, ensuite le broker redistribue les messages sur les channels correspondant et enfin les clients reçoivent les messages sur les canaux auxquels ils sont souscrits.

— **Consommateur** : Le consommateur ou le client est celui qui réceptionne les messages diffusés par le producteur, et qui a la possibilité de réagir à ceux-ci en direct. Pour se faire, le client souscrit à différents channels gérés par le broker et attend qu'un événement se produise sur ses canaux. Le consommateur peut s'abonner à un ou plusieurs channels, et comme les messages sont diffusés de manière asynchrone,

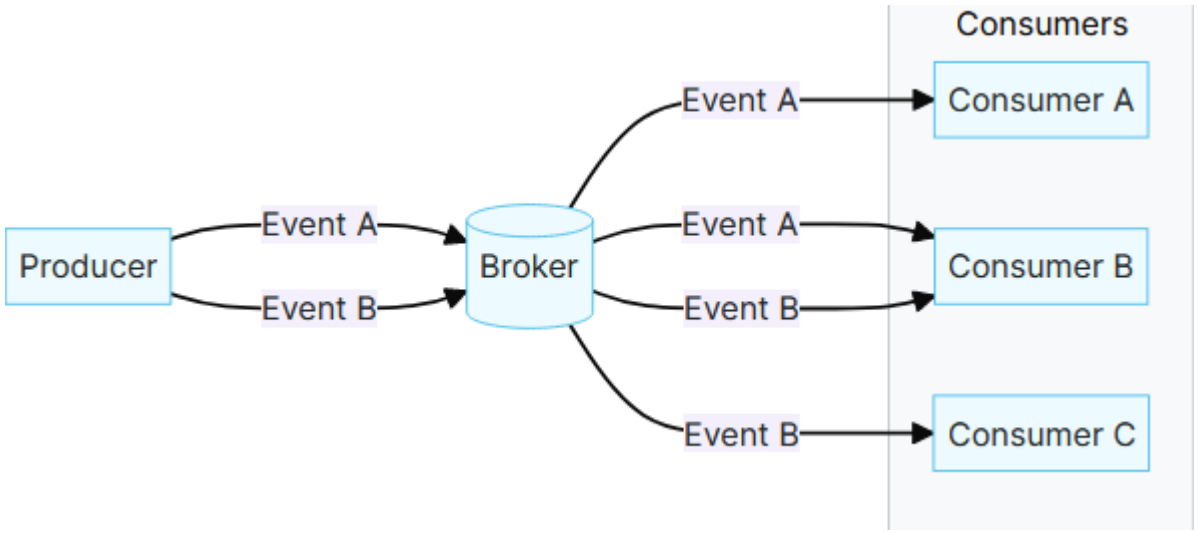


Figure 2.2. – EDA consommateur schema[7]

le client reçoit tous les messages de façon indépendante. Par exemple, si plusieurs événements se produisent en même temps le client recevra tous les messages au même moment (figure 2.2).

— **Protocole** : Le protocole est très important dans une architecture EDA. C'est l'ensemble des règles définies qui indiquent comment les informations seront échangées entre les différents composants du système. C'est lui qui va se charger de transporter les changements d'états sentis par le producteur sous le forme de messages. Le protocole transporte ensuite le message au serveur, puis jusqu'au consommateur. La figure 2.3 montre un exemple de transport de message avec le protocole MQTT [1].

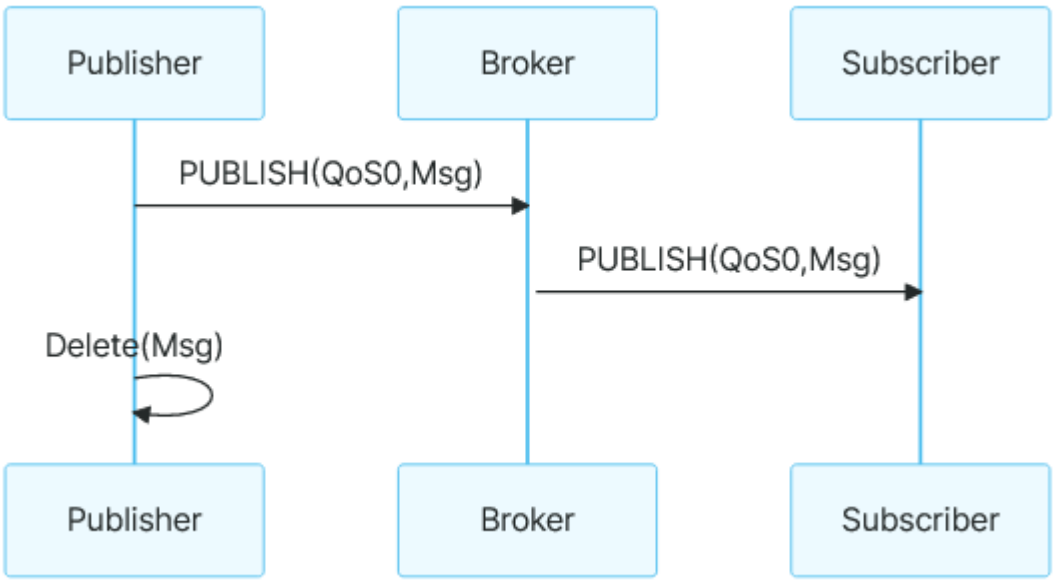


Figure 2.3. – Diffusion d'un message utilisant le protocle MQTT[22]

[1]Le protocole MQTT permet de choisir le niveau de qualité de service (QoS) lors de la publication du message, représenté par "QoS0" dans la figure 2.3.

— **Message** : Le message contient les informations captées par le producteur. Il sert à échanger les informations avec un un ou plusieurs receveurs. Le même message peut être reçu par de multiples clients en même temps.

2.3. Documentation

La documentation d'une application et plus particulièrement d'une API est presque aussi essentielle que le code lui-même. Effectivement, une API peut se composer de plusieurs milliers de lignes pour les plus complexes d'entre elles, ce qui rend parfois la lecture du code pénible et difficile. Ainsi, il peut vite devenir complexe de comprendre le fonctionnement et les possibilités qu'offre une application que nous utilisons ou souhaitons utiliser. Pour éviter ce travail fastidieux, la documentation des APIs est presque devenue obligatoire. Elle permet de comprendre en quelques lignes ce que l'API peut faire et ce pourquoi elle a été conçue. Cela est d'autant plus important dans le monde des services web, là où les applications modernes se composent de plusieurs services mis bout-à-bout, pour former de nouveaux services de plus en plus complexes.

2.3.1. OpenAPI & Swagger

OpenAPI

OpenAPI[21] naît d'une initiative de plusieurs experts de l'industrie informatique, qui ont vu la nécessité de standardiser la façon dont les APIs basées sur HTTP sont décrites et documentées. Comme cité précédemment, la documentation permet au plus grand nombre de comprendre le fonctionnement et la structure d'une API, et ce indépendamment du code et du langage de programmation utilisé. Les auteurs d'OpenAPI décident donc de créer des spécifications nommées "OpenAPI specifications" (figure 2.4), qui sont des marches à suivre pour documenter une API à chaque stade de son développement. Ce standard devient rapidement la norme pour toutes les APIs REST, au vu de son architecture et de son fonctionnement sur HTTP. OpenAPI est un langage de spécification qui définit la structure et la syntaxe d'une API. La documentation des APIs se fait sous la forme de fichiers JavaScript Object Notation (JSON) ou Yet Another Markup Language (YAML). La figure 2.5 illustre un exemple de documentation JSON générée via des annotations placées dans le code source [2].

[2]Le fonctionnement des annotations dans le code source est expliqué dans la section Swagger

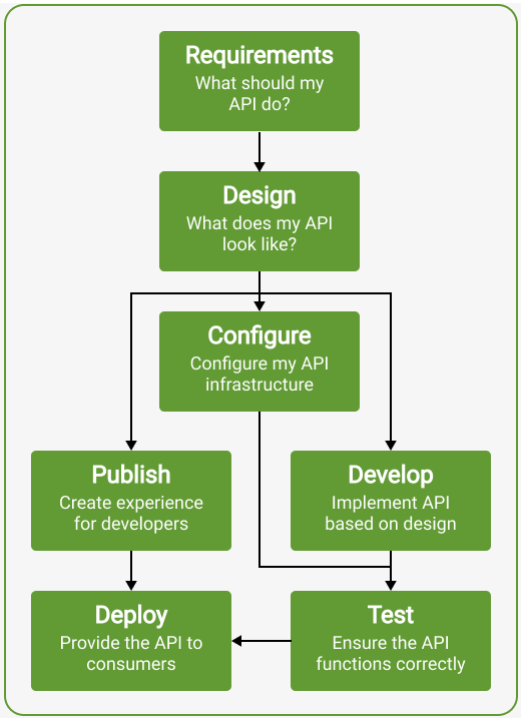


Figure 2.4. – OpenAPI spécification[21]

Swagger

L'introduction des spécifications d'OpenAPI a fortement révolutionné la façon dont les interfaces REST sont documentées. L'échange de connaissances entre les programmeurs et les clients a grandement été facilité grâce à cette standardisation de la documentation. Toutefois, même si décrire les APIs à partir d'un fichier JSON et YAML est à la portée de tous, et facilite la lecture et la compréhension de n'importe quelle API. Ce travail peut vite s'avérer laborieux et chronophage, si notre interface REST se compose de milliers de lignes de code. De plus, créer une bonne documentation n'est pas aussi évident qu'il n'y parait. Il faut souvent trouver un bon modèle ou des squelettes de documentations existants pour documenter correctement son API. Ainsi, une nouvelle initiative nommée Swagger[24] est entreprise pour donner suite au projet OpenAPI. Un outil est développé pour permettre de générer directement depuis le code source de son API une documentation. L'idée est d'insérer des annotations spéciales au sein du code source de son application dans un format basé sur eXtensible Markup Language (XML), afin de créer une documentation. Le document créé à partir de ces annotations possède plusieurs avantages. Il permet évidemment un gain de temps dans la documentation des APIs et il permet une réelle standardisation de toutes les APIs REST. Une autre particularité de l'outil mis à disposition par Swagger et qu'il offre la possibilité de directement créer la documentation de son API avec les spécifications OpenAPI dans une page HTML, sous le format d'une interface graphique, comme le montre la figure 2.6. Un des points forts de cette interface, c'est quelle donne la possibilité de directement tester les différentes méthodes REST (figure 2.7), en plus d'offrir une description de chaque méthode et du fonctionnement de l'API.

Figure 2.5. – Exemple de documentation OpenAPI en format JSON

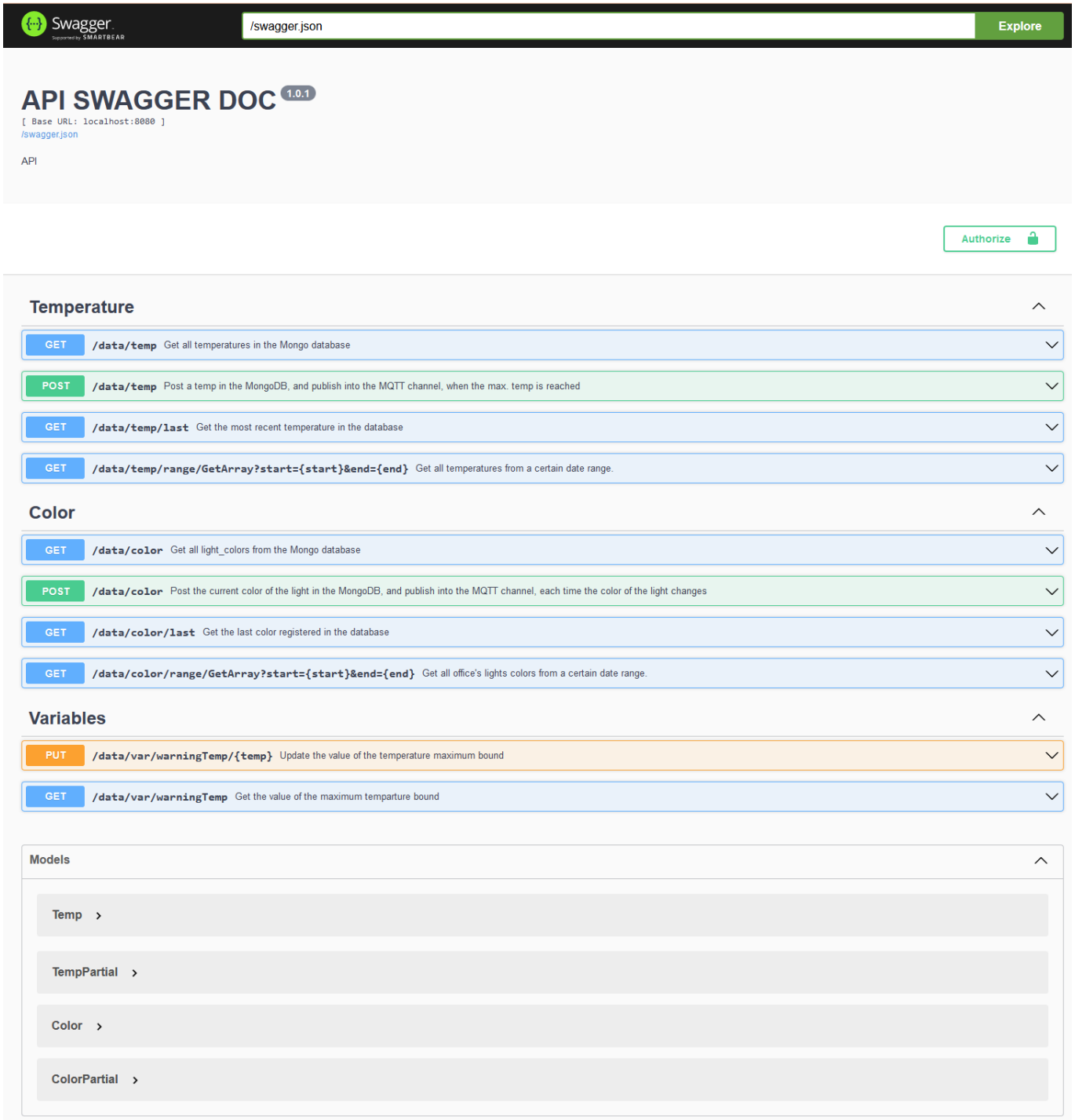


Figure 2.6. – Swagger documentation

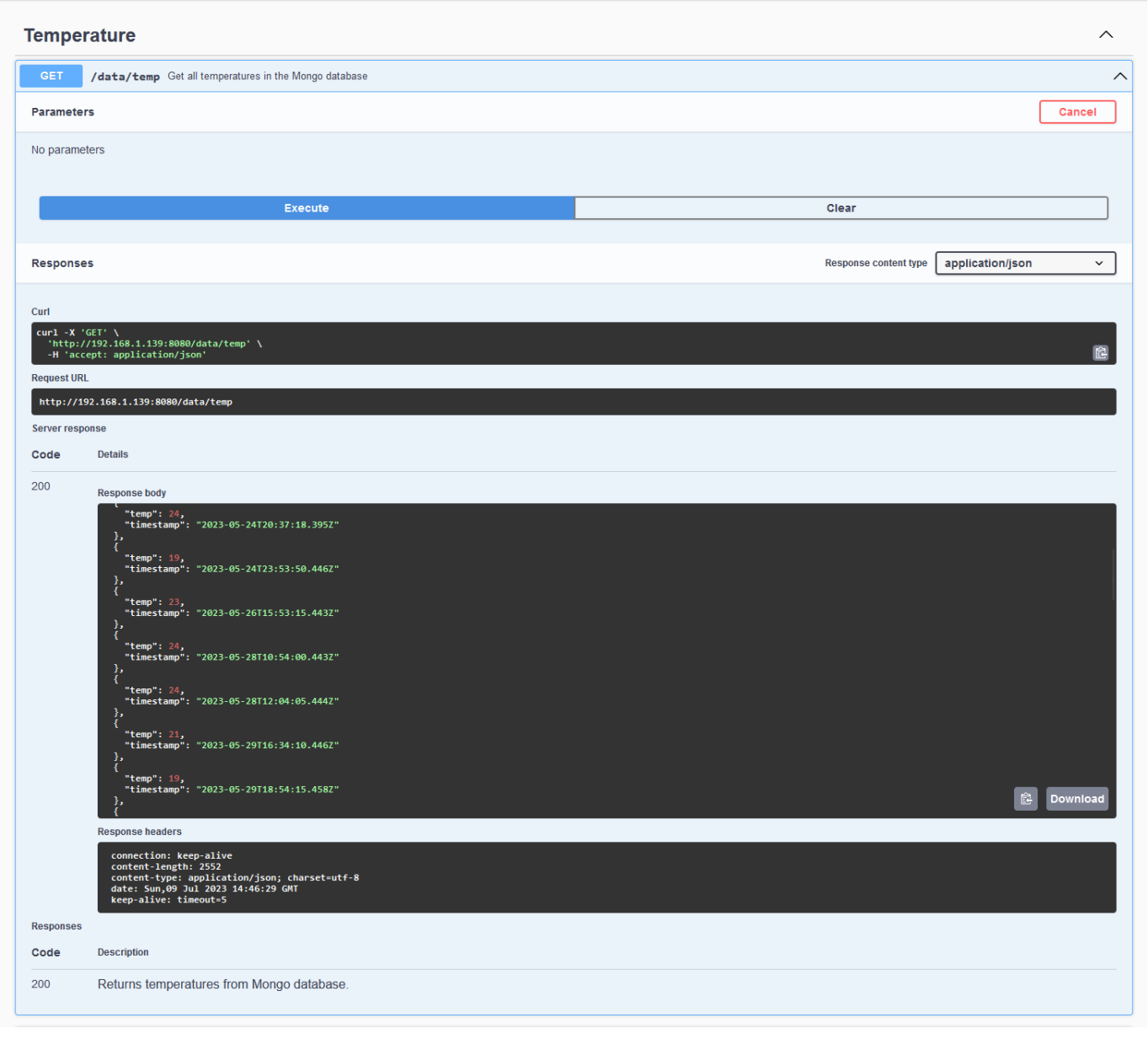


Figure 2.7. – Swagger documentation, exemple de GET

2.3.2. AsyncAPI

Comme pour les APIs basées sur HTTP, les applications utilisant l'architecture orientées événements ne font pas exception et nécessitent également une bonne documentation pour leurs interfaces. De ce constat, un autre regroupement de développeurs décide également de créer des spécifications pour les applications EDA. C'est ainsi que la plateforme AsyncAPI[5] voit le jour. Elle définit des spécifications pour documenter toutes les APIs EDA et se revendique comme le futur outil pour définir toutes les interfaces orientées événements. Elle offre sur sa plateforme la possibilité de générer une documentation pour les APIs sous la forme de fichier JSON ou YAML(figure 2.8). Cette initiative est à ses débuts et à l'heure actuelle, il n'existe pas encore d'outils sophistiqués qui permettent de générer dynamiquement une documentation via le code source de son application, comme peut le proposer la plateforme Swagger. Pour l'instant, il faut se contenter d'écrire une fichier soi-même en partant de zéro. La figure 2.9 montre un exemple de ce qui est pour le moment possible de faire grâce l'outil de AsyncAPI à partir d'un fichier JSON. La meilleure alternative à ce jour est de trouver un bon squelette et d'écrire sa documentation à partir de ce modèle. AsyncAPI part d'une bonne idée et n'est qu'aux prémices de son développement. De futurs outils émergeront certainement pour remédier au manque que présente AsyncAPI pour l'instant. Comme les APIs REST, la standardisation des interfaces EDA est nécessaire, pour résoudre les mêmes problèmes et contraintes que présentaient les applications REST avant l'apparition d'OpenAPI et de Swagger.

2.4. Synthèse

En conclusion, les deux architectures se rendent de nos jours indispensables et deviendront certainement à terme complémentaires. Même si les APIs REST restent encore le plus représentées et le plus utilisées, notamment par les utilisateurs du "World Wide Web". De plus en plus de services proposés par les industriels fonctionnent avec une architecture orientée événements, et entrent à pas de géant dans nos habitudes et notre quotidien avec l'inondation d'objets connectés en tout genre. Malgré que développer des APIs purement REST ou purement EDA reste encore envisageable, la combinaison des deux architectures me parait à mon sens être l'avenir des APIs et des services. Les deux possèdent des fonctionnalités dont ne disposent pas l'autre et une API qui se voudrait complète à l'ère de l'Internet 3.0 devra certainement contenir une concaténation des deux styles d'architectures pour pouvoir répondre à tous les enjeux et problématiques actuels.

De même pour la documentation, qui comme mentionné précédemment est essentielle pour n'importe quelle API, et évoluera sûrement dans une direction commune, en redéfinissant des nouvelles spécifications pour les deux architectures. AsyncAPI va certainement offrir de nouveaux outils et se rapprochera de la plateforme Swagger. Une fois que AsyncAPI et Swagger offriront semblablement les mêmes fonctionnalités, la dernière étape sera de développer un outil capable de combiner les deux documentations, afin d'obtenir une belle documentation uniforme.

```
"asyncapi": "2.6.0",
"info": {
  "title": "MQTT API",
  "version": "1.0.0",
  "description": "Simulation of an office.\n"
},
"servers": {
  "mosquitto": {
    "url": "tcp://localhost:9000",
    "protocol": "mqtt"
  }
},
"channels": {
  "data/sub": {
    "subscribe": {
      "summary": "Information about the avaibility of the professor",
      "message": {
        "$ref": "#/components/messages/light_color"
      }
    }
  },
  "data/sub/{light_color}": {
    "parameters": {
      "light_color": {
        "$ref": "#/components/parameters/light_color"
      }
    },
    "publish": {
      "summary": "Send the porfessor's office status",
      "message": {
        "$ref": "#/components/messages/light_color"
```

Figure 2.8. – Documentation AsyncAPI en format JSON

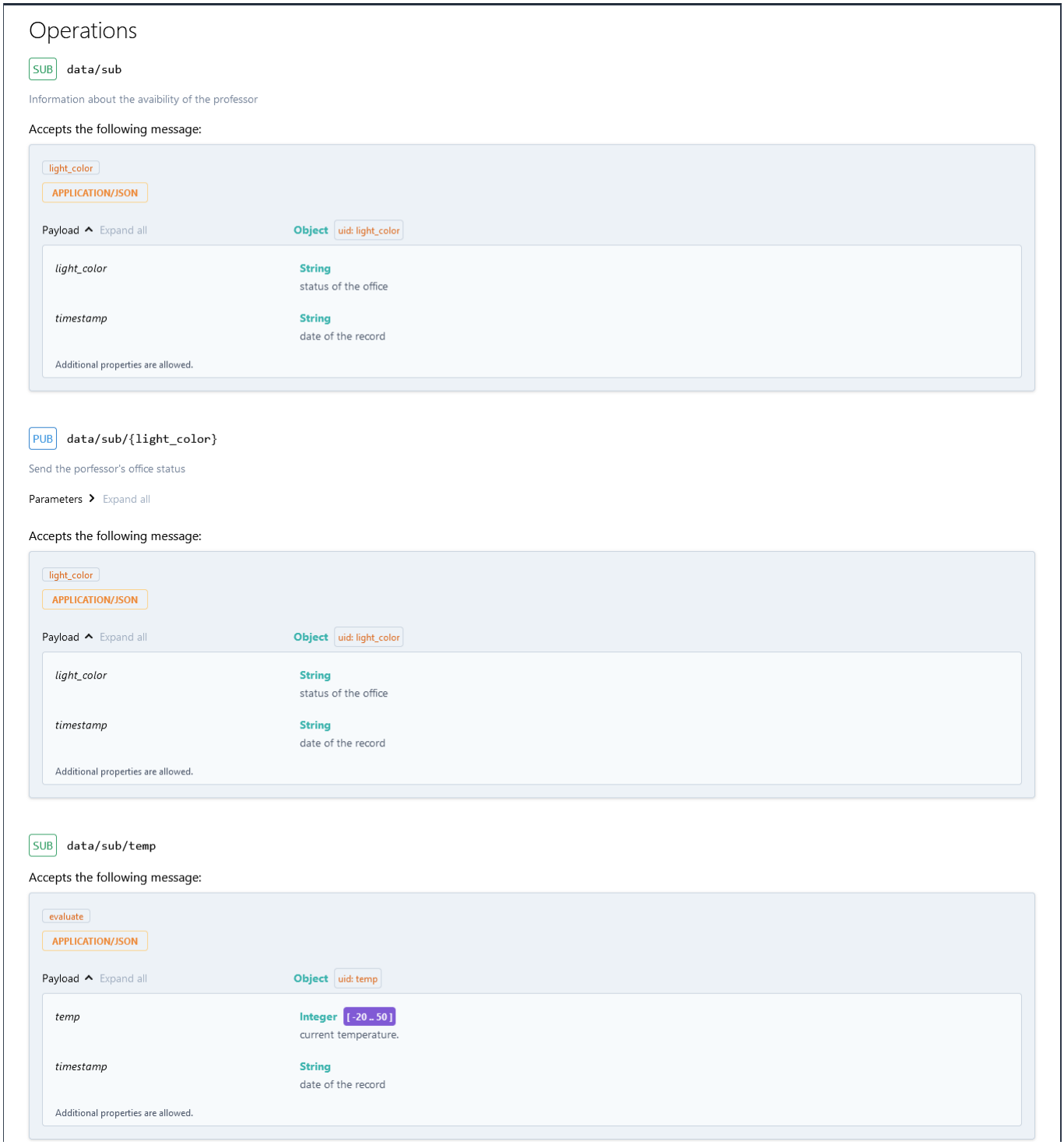


Figure 2.9. – AsyncAPI documentation

3
Présentation du projet

Cette section présente les différentes possibilités et interactions qu'offre l'application mobile.

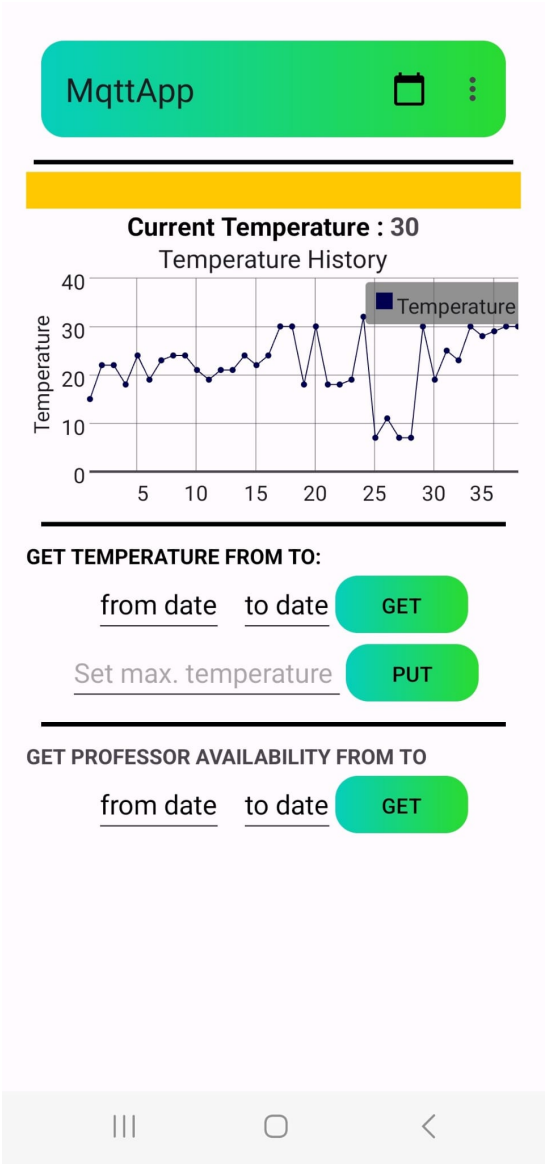
L'application mobile peut faire des requêtes HTTP au serveur principale. Elle permet également de s'abonner à différents "channels" disponibles sur le serveur, qui utilise les propriétés d'une architecture EDA, afin d'être informé en direct des événements qui se passent dans le bureau du professeur. En somme, cette interface mobile sert de lien aux clients pour communiquer avec l'API REST et EDA. L'idée de faire cette application est de pouvoir remplir l'objectif de développer une application qui respecte les contraintes REST, tout en étant capable d'intégrer des éléments d'une architecture orientée évènements.

L'application est pensée pour être destinée à deux types de clients. Premièrement aux professeurs propriétaires des bureaux et secondement aux étudiants souhaitant interagir avec les professeurs. Différents scénarios vont être présentés pour expliquer plus en détail le fonctionnement de l'application mobile.

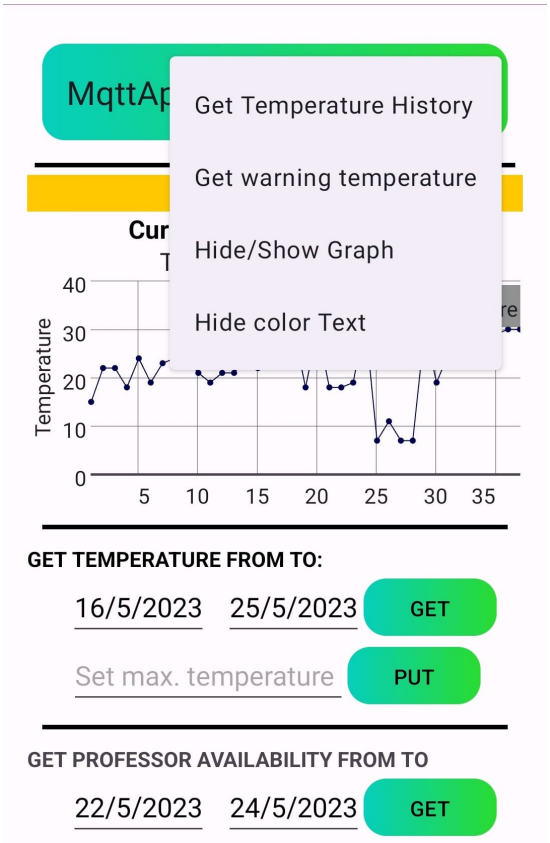
3.1. Écran d'accueil

L'écran d'accueil (figure 3.1a) montre toutes les interactions disponibles que cela soit pour le professeur ou l'étudiant. Dans une version future, l'idéale sera de séparer les "features" destinées aux professeur de celles des étudiants. L'application se compose de plusieurs parties :

- **En-Tête** : L'en-tête contient le nom de l'application, l'icône permettant d'ouvrir un nouvel écran (calendrier), ainsi que les options disponibles pour l'écran d'accueil.
- **Barre de statuts** : La barre de statuts figurant en jaune sur le figure 3.1a, indique l'état actuel de la porte du bureau. L'état de la porte décrit la position de la porte et en fonction de chaque position, différents états sont enregistrés par le bureau. La porte dispose de trois états :
  - **Vert** : La porte est grandement ouverte : Ce qui indique que le professeur est présent et disponible pour recevoir des étudiants.
  - **Jaune** : La porte est entrouverte : Le professeur se situe dans son bureau, mais n'est pas disponible pour recevoir des étudiants.



(a) Écran d'accueil application



(b) Options de l'écran d'accueil de l'application

Figure 3.1. – Écran d'accueil de l'application

  - **Rouge** : La porte est fermée : Le professeur n'est pas dans son bureau.
- **"Current Temperature"** : Affiche la température du bureau en temps réel.
- **Graphique** : Le graphique montre les différentes températures. Par défaut, il affiche toutes les températures stockées dans la base de données.
- **"GET temperature from to"** : Permet de sélectionner une plage de dates, dans laquelle nous souhaitons obtenir les températures et qui seront dessinées sur le graphique.
- **"Set max. temperature"** : Permet de définir la température maximale du bureau avant de lancer une alerte.
- **"GET professor availability from to"** : Permet de sélectionner une plage de dates qui affiche les disponibilités du professeur.
- **Options (figure 3.1b)** :
  - **"GET Temperature History"** : Affiche toutes les températures stockées dans le base de données sur le graphique. Conçu, lorsqu'une plage de dates précise a été sélectionnée et que l'on souhaite revenir au graphique par défaut.
  - **"GET warning temperature"** : Affiche une notification sur l'écran montrant la température maximale avant de lancer une alerte. Comme le présente la figure 3.2
  - **"Hide/Show Graph"** : Permet de cacher ou afficher le graphe des températures.
  - **"Hide color Text"** : Cette option cache le texte obtenu, quand une plage de dates a été sélectionnée pour les disponibilités du professeur. Le nom n'est peut-être pas très significatif et pourrait effectivement être changé.

3.2. Scénario I : Le professeur et son bureau

L'idée principale de cette application et de permettre au professeur de savoir en temps réel ce qui se passe dans son bureau. Pour ce faire, le bureau disposerait de différents capteurs mesurant diverses choses. Ainsi, il est facile d'imaginer que le bureau pourrait être équipé d'autant de capteurs que le propriétaire le souhaite, afin de satisfaire tous ces besoins et désirs. Dans ce projet, le bureau mesure uniquement la température courante et l'état de la porte d'entrée du bureau, pour des raisons de simplicité. Toutefois, si l'application fonctionne avec des capteurs mesurant la température et l'état de la porte, il ne sera pas de difficile d'implémenter d'autres mesures dans l'application pour la rendre aussi complexe que souhaité. Que cela soit la mesure du taux de CO_2, la position des fenêtres, l'intensité des différentes lumières, etc.

Dans cette version de l'application, le professeur peut déjà effectuer diverses actions. Il reçoit premièrement le température en temps réelle toutes les cinq secondes. Il modifie également la barre de statuts du bureau de façon dynamique en changeant la positon de sa porte. Le changement de température ou d'état de la porte sont automatiquement détectés par l'application qui met à jour son interface graphique instantanément.
La professeur peut également paramétrer son bureau directement depuis l'application. Il peut définir la température maximale acceptable dans le bureau avant d'être alerté. Pour

cela, le professeur peut utiliser "la ligne" *Set max. temperature* de l'application pour indiquer au serveur la nouvelle température maximale. Si le professeur ne se souvient plus de la température maximale du bureau, il peut l'obtenir via les options en cliquant sur le bouton *GET warning temperature* (figure 3.2).

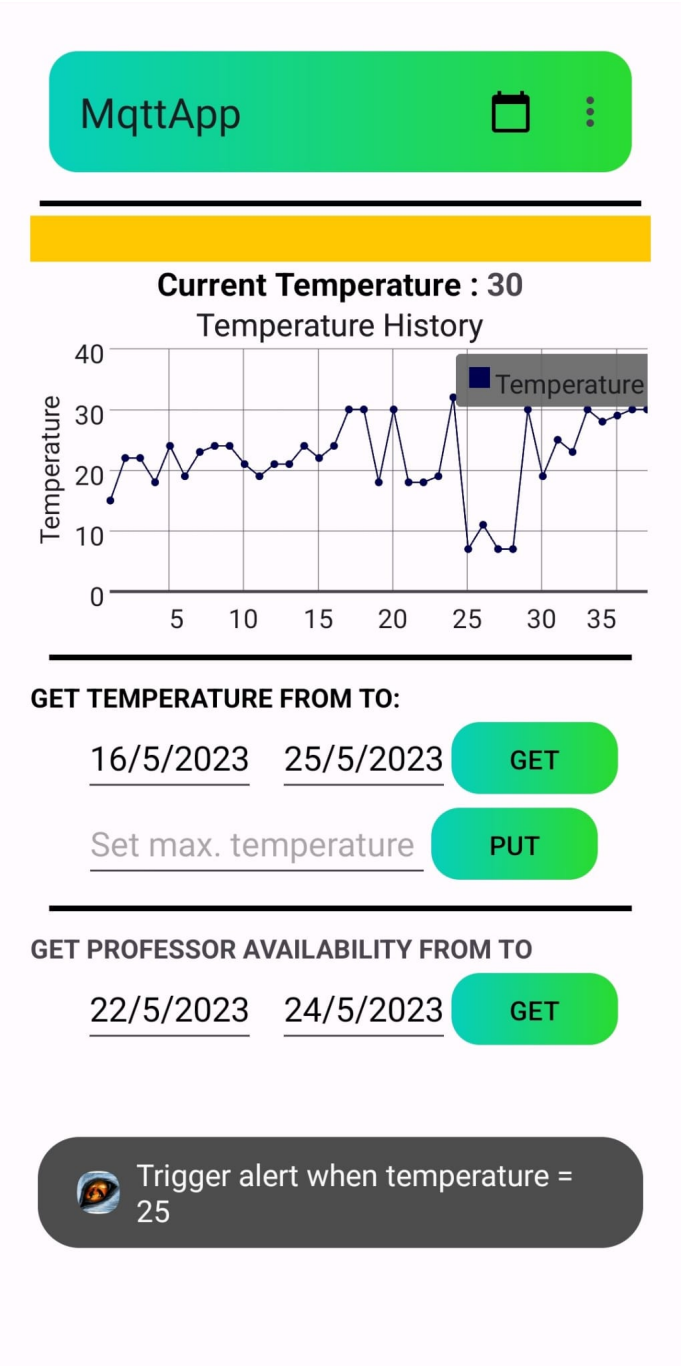


Figure 3.2. – Notification indiquant la température maximale du bureau

Le professeur se voit ensuite alerté (figure 3.3), si la température maximale configurée est atteinte ou dépassée, via une notification sur son téléphone, et cela y compris lorsque le professeur n'est pas sur l'application (l'application doit tourner en arrière-plan du téléphone pour cela).

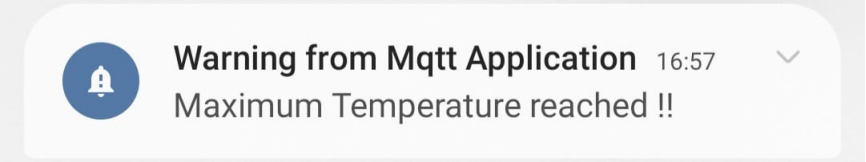


Figure 3.3. – Alerte quand la température maximale est atteinte ou dépassée

Même si le paramétrage du bureau reste pour l'instant très sommaire, il est simple d'imaginer d'autres paramétrages et alertes avec des fonctionnements similaires que pour la température ou la porte avec des capteurs supplémentaires. Par exemple, si nous pouvons obtenir une température maximale, un minimum est facilement programmable. En outre, obtenir les taux de CO_2, de la pression de l'air et leur fixer des limites ou encore ouvrir les fenêtres en fonction d'une certaine température ou d'une mesures précise, et voire même effectuer certaines actions comme appeler les pompiers si la températures est trop haute ; sont autant d'implémentations possibles que cette application pourrait intégrer, si elle était déployée dans une environnement réelle.
Le professeur peut également choisir un intervalle de dates pour regarder les températures enregistrées durant cette période, donc envisager des fonctions plus élaborées comme des moyennes globales de température ou sur certains intervalles, pourraient facilement être rajoutées.

Une version idyllique et finale de l'application serait que chaque propriétaires de bureaux, placent les capteurs qu'ils souhaitent dans leurs bureaux et l'application se modifierait de façon dynamique pour correspondre aux différents capteurs. Il serait ainsi possible de consulter, fixer des limites et utiliser diverses fonctions basiques (intervalles de temps, moyennes, actions à effectuer en cas de limite, etc) pour chacun des capteurs. De cette manière, l'application permettrait de répondre au mieux aux besoins de chaque professeur.

3.3. Scénario II : Gestion des rendez-vous

Les professeurs sont amenés à passer beaucoup de temps dans leur bureau tout au long de leur carrière. Travailler dans un cadre confortable et optimisé à leurs besoins devient un luxe à la porté du plus grand nombre grâce à l'avènement des objets connectés et de l'Internet of Things (IoT). Même si la gestion du bureau connecté reste l'élément centrale de l'application, le travail de professeur ne se résume pas seulement à enseigner et à travailler sur des projets de recherches. Ils sont régulièrement sollicités par les étudiants ou leurs collègues, qui parfois peuvent surgir à des moments inopportuns. Trouver une date de rendez-vous peut donc s'avérer fastidieux, car il est peut être nécessaire d'envoyer un mail, il faut attendre une réponse de la personne concernée, ce qui peut vite devenir une perte de temps et d'énergie pour une simple consultation. De ce constat, l'application permet également au professeur de configurer ses disponibilités directement dans l'application liée à son bureau, comme le montre la figure 3.4. Cette écran est accessible depuis l'écran d'accueil en cliquant sur l'icône de calendrier présent dans l'en-tête. La calendrier permet pour l'instant seulement de définir un événement pour un jour entier et d'indiquer le nom de l'événement. Il serait intéressant dans une version finale de permettre à l'utilisateur

de définir ces disponibilités et indisponibilités sur des plages horaires plus précises, pour que le calendrier soit vraiment complet.

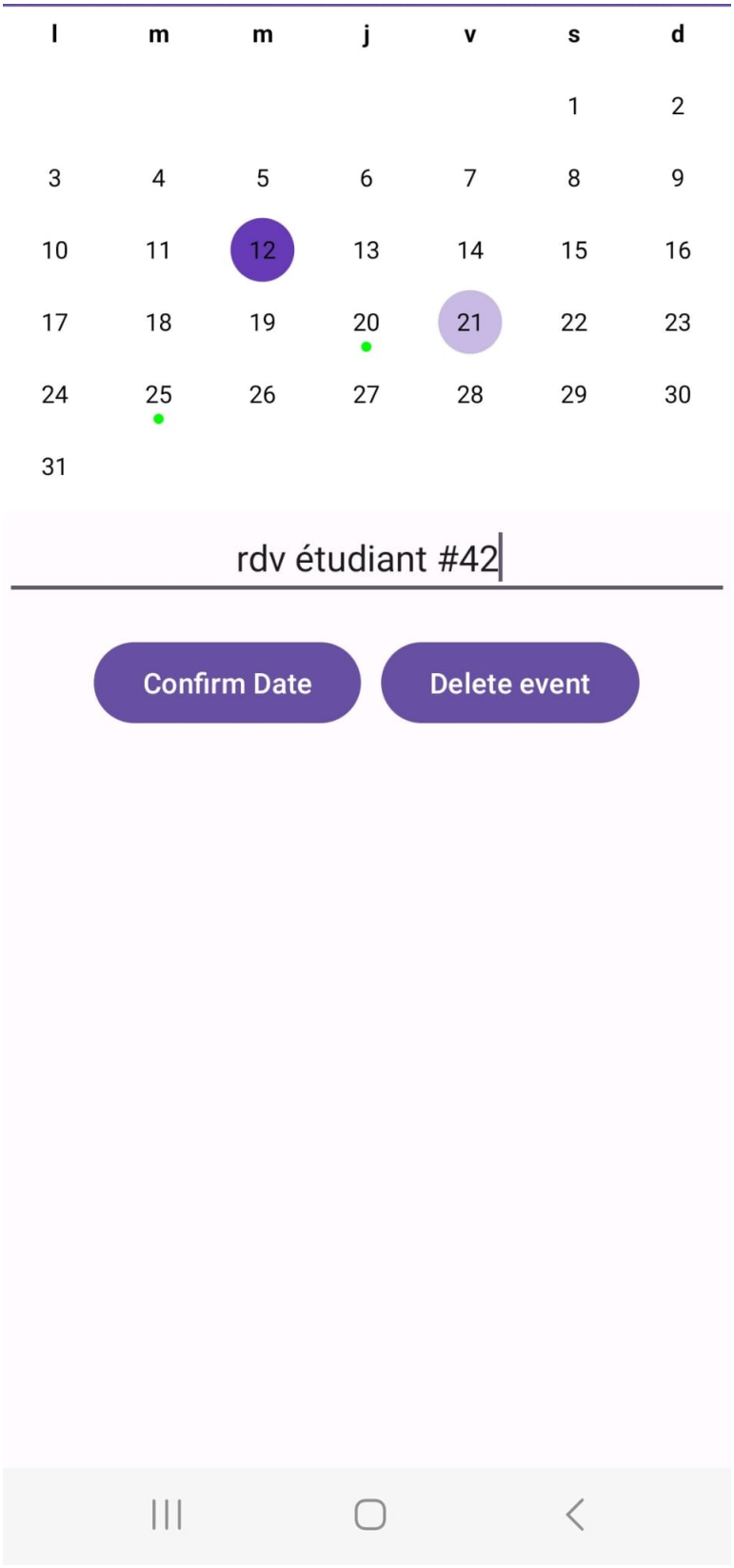
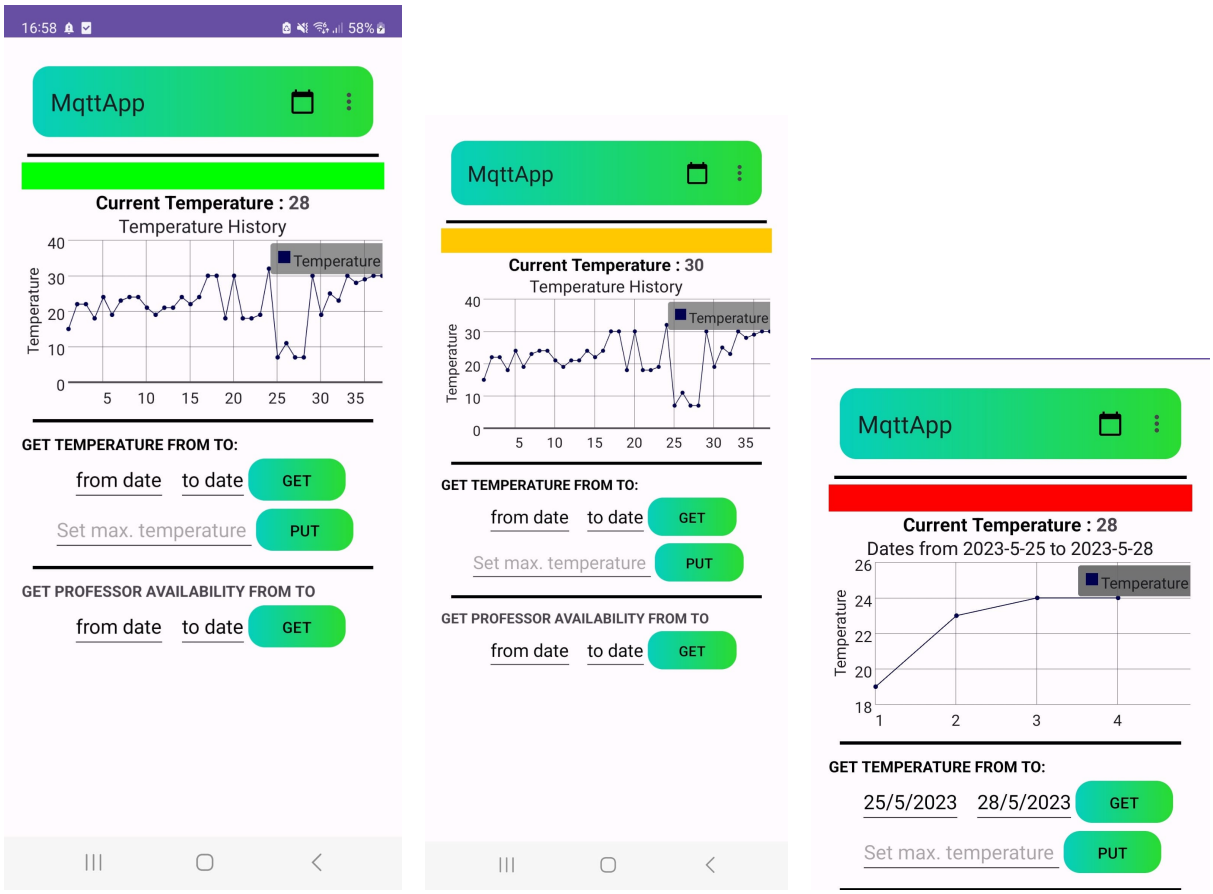


Figure 3.4. – Écran du calendrier

3.4. Scénario III : Les étudiants et le professeur

L'avantage d'une architecture orientée événements, c'est quelle permet à différent utilisateurs de s'abonner à différents canaux pour être informé en direct des changements qui se passent. De cette idée, en plus de permettre à un utilisateur de paramétrer son bureau, il peut également être intéressant d'imaginer qu'une personne extérieure au bureau souhaite connaître quelques informations concernant celui-ci, sans avoir à déranger le professeur en question. Par exemple grâce à la couleur de la barre de statut, l'utilisateur qui se connecte à l'application sait directement si le professeur est disponible ou non pour une consultation, comme l'illustre les figures 3.5. De plus, si le professeur n'était pas présent ou est indisponible, dès que la barre de statut passera au vert l'étudiant sera notifié via une notification (figure 3.6), et ce sans avoir besoin d'être sur l'application, du moment que l'application tourne en arrière plan sur son téléphone.



(a) Barre de statut verte (b) Barre de statut jaune (c) Barre de statut rouge

Figure 3.5. – Différentes barres de statuts possible

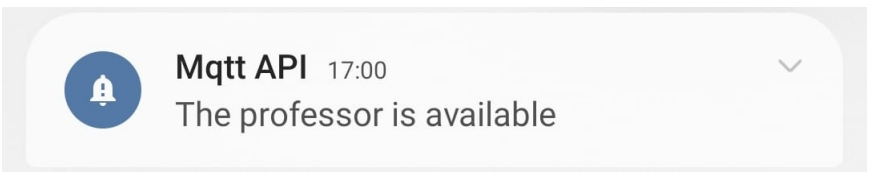


Figure 3.6. – Alerte professeur disponible

Grâce à la fonction calendrier, l'étudiant peut consulter directement les disponibilités du professeur, pour des dates précises grâce à la ligne *GET professor availability from to* (figure 3.7), sans avoir à le consulter. A partir de ces fonctions simples, il est facile d'imaginer un écran supplémentaire qui permettrait aux étudiants de s'abonner à autant de "channels" des professeurs qu'ils le souhaiteraient, accessibles via un nom et un mot de passe donnés par les professeurs. Ainsi, le côté EDA servirait aux étudiant à la fois pour savoir si un certain professeur est actuellement dans son bureau ou pas, connaître ses futures disponibilités et voire même recevoir des messages (annonces, maladie, etc), sans avoir à consulter ses mails. Avec le côté REST de l'API, les étudiants pourraient par exemple obtenir les cours du professeur ou leurs notes de cours.

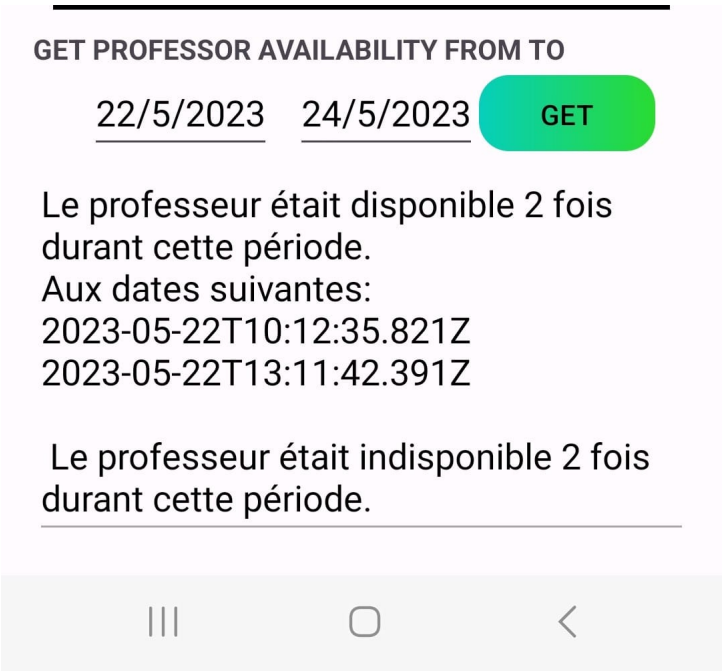


Figure 3.7. – Disponibilités du professeur dans une intervalle de dates

En conclusion, l'application est fonctionnelle et remplit l'objectif de pouvoir interagir avec un serveur REST en recevant et envoyant des requêtes HTTP. Mais aussi d'être le client souscrit aux différents "channels" du serveur EDA, en réceptionnant les messages publiés par le producteur en temps réel. Grâce à cette propriété EDA, elle met à jour son interface graphique instantanément et exécute des actions spécifiques en fonctions de différents évènements. Même si son fonctionnement reste assez simple et ne mesure que peu de choses pour l'instant, l'application démontre qu'elle peut rapidement se complexifier et reste facilement implémentable pour accueillir autant de bureaux et de capteurs que nécessaire.

4

Détails de programmation

La projet se compose de quatre éléments distincts. Une application qui envoie des données au serveur codée en "Java", un serveur principale développé avec "Node.js" et le framework Koa, une base de données et une application mobile conçue avec "Android Studio" qui permet aux clients de consommer les données. La figure 4.1 illustre les différentes interactions entre chaque composant du projet.

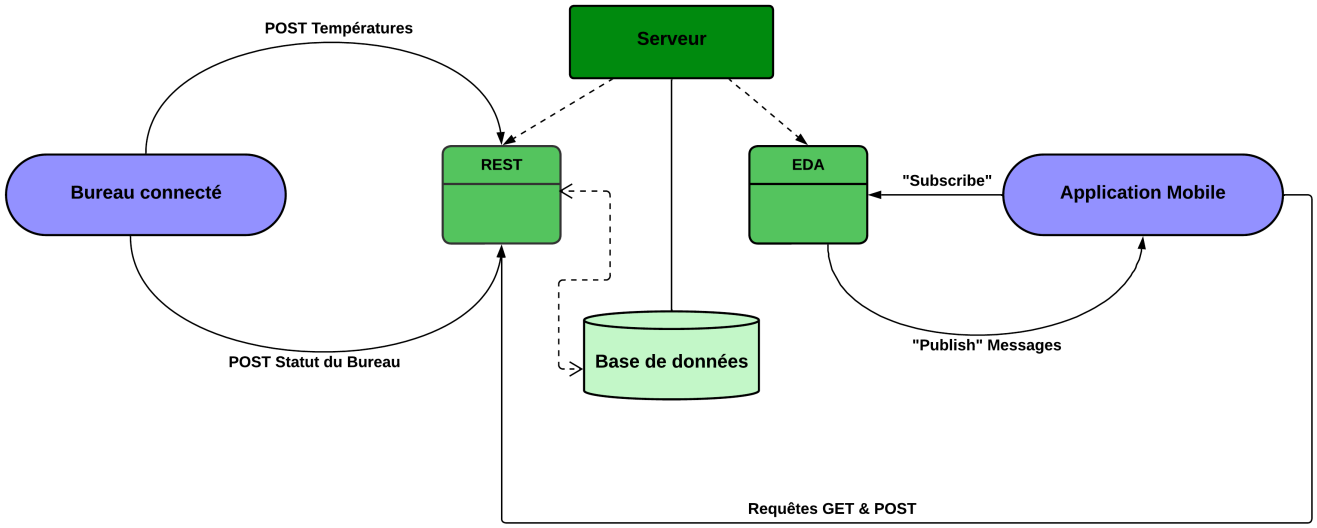


Figure 4.1. – Schéma montrant le fonctionnement des trois applications

4.1. Simulation du bureau du professeur

Premièrement, il y a une application qui simule le bureau d'un professeur et qui envoie des informations concernant celui-ci toutes les cinq secondes. L'application permet également de créer et d'envoyer des données de façon manuelles. N'ayant pas la possibilité de travailler avec de réels capteurs, cette application fait office de capteurs qui auraient

normalement été présents dans le bureau du professeur. Le rôle des capteurs est d'enregistrer les différents états du bureau et de les communiquer au serveur principale.
Le langage Java[11] a été choisi pour programmer cette partie du projet.

4.1.1. Java

Java est un langage orienté objet et possède beaucoup de libraire pour la programmation web. Il fut pendant longtemps la langage le plus utilisé pour la création d'API fonctionnant sur le web et une bonne partie des applications tournant sur le web dépendent encore de Java. Même si de nos jours, des langages comme Python[23] ou *Node.js*[18] sont de plus en plus utilisés pour la programmation d'applications web et de services. Java reste toutefois toujours très utilisé dû à son grands nombres de librairies couvrant presque tous les domaines concernant le web et son fonctionnement optimisé pour la programmation orientée objet. Ainsi, avec Java il m'était facile de créer un objet simulant le bureau et qui pouvait communiquer avec mon serveur web sans trop de difficultés.

4.1.2. Présentation de l'application

La figure 4.2 présente l'interface graphique de l'application utilisée. Java met à disposition l'utilisation de nombreux composants dans sa libraire *Java Swing*[12], ce qui permet l'implémentation rapide d'une interface graphique. L'application se compose de différentes parties :

— **Bouton Run/Stop** : Lorsque le bouton est mis en marche, l'application commence à communiquer avec le serveur. Chaque cinq secondes, elle génère aléatoirement un integer compris entre 17 et 30, ce qui représenterait les températures capturées par le capteur du bureau. Ce nombre est ensuite envoyé au serveur à l'aide d'une requête HTTP POST et attend une réponse du serveur. Si la requête a été correctement exécutée, le message 200 est retourné, autrement un message d'erreur apparaît. Quand le bouton est à nouveau cliqué, l'API cesse de générer des nombres et arrête d'envoyer des requêtes au serveur. Le listing 4.1 montre le code utilisé pour envoyer des requêtes HTTP POST.

```
1       HttpClient client = HttpClient.newHttpClient();
2       HttpRequest request = HttpRequest.newBuilder()
3           .uri(URI.create(uri))
4               .timeout(Duration.ofMinutes(2)) // Temps de timeout de la requete
5               .header("Content-Type", "application/json")
6               .POST(HttpRequest.BodyPublishers.ofString(jsonObject.toString()))
7               .build();
8
9       client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
10              .thenApply(HttpResponse::body)
11              .thenAccept(System.out::println).join();
```

Listing 4.1 – Exemple Java POST méthode

— **Bouton Send Color** : Le bouton *Send Color*, permet de simuler le capteur enregistrant la position de la porte du bureau. En fonction de la position de la porte, le capteur enregistrerait l'état vert, jaune ou rouge. Contrairement à la température, la porte bouge seulement à des instants précis et son état n'a donc pas besoin d'être

envoyé toutes les X secondes au serveur, mais uniquement lorsque un changement de position se produit. Il suffit donc de sélectionner un des trois boutons radios qui simulent l'état de la porte et d'appuyer sur le bouton *Send Color* pour envoyer une requête de type POST au serveur, afin de stocker l'état sous forme de String dans la base de données. Comme il est difficile de réellement simuler le comportement d'un professeur dans son bureau, plusieurs états ont été envoyés au serveur pour les besoins du projet, ne reflétant pas nécessairement la routine d'un professeur. Le but étant principalement de valider le fonctionnement des requêtes au serveur et de montrer que différents états pouvaient êtres obtenus par le capteur.

— **Bouton Send temperature** : Cette ligne permet d'envoyer manuellement la température voulue au serveur via une requête POST. Ce bouton sert surtout de bouton de test et de démonstration. Notamment pour vérifier si la fonction du serveur devant alerter le professeur à partir d'un certain seuil fonctionne correctement.

— **Automatic Temp send** : La dernière ligne de la figure 4.2, affiche l'integer généré (pour la température) de façon aléatoire lorsque l'application est en mode automatique.

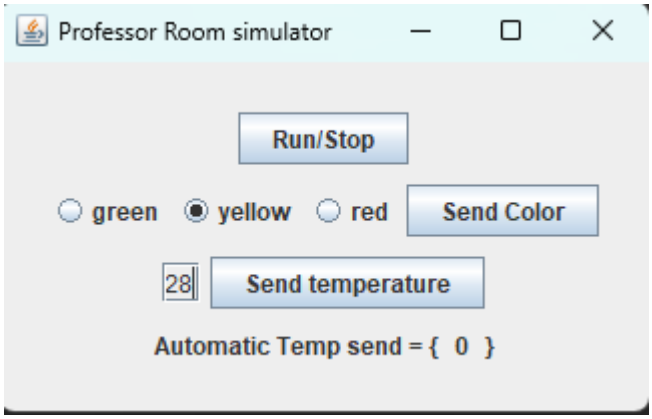


Figure 4.2. – Application simulant les capteurs du bureau

4.2. Serveur

4.2.1. Description & fonctionnement

Le serveur est l'élément central de ce projet. Il supervise le bon déroulement de tous les autres composants et sans lui, l'application mobile et l'application simulant le bureau ne fonctionneraient tout simplement pas. En effet, lorsque les températures sont envoyées depuis le bureau, c'est le serveur qui reçoit les requêtes, envoie les réponses et qui s'occupe de stocker les informations dans sa base de données. C'est également lui qui sert de producteur pour la partie orientée événements. Il détecte si un changement d'état se manifeste et si c'est la cas, il publie les messages à l'aide du protocole MQTT[17].

Le travail principale du serveur consiste essentiellement à répondre aux différentes requêtes HTTP des deux applications et c'est pourquoi il est avant tout basé sur l'architecture REST comme décrit dans le chapitre deux. Cependant, il intègre également les éléments de l'architecture orientée événements en se comportant comme le producteur

capable de publier des messages à un broker en utilisant le protocole MQTT.

Le langage de programmation Node.js a été choisi pour développer le serveur, car il permet de construire un serveur qui répond aussi bien à une architecture orientée évènements que REST. De plus, une multitude de base de données sont compatibles avec Node.js et beaucoup de frameworks existent pour faciliter le développement web. La base de données a été réalisée avec MongoDB[15] et le framework orienté web Koa[13] a été choisi pour la partie développement web.

4.2.2. Node.js

Node.js[18] est un environnement de développement javascript orienté réseau. C'est un outil puissant pour la conception d'application web. Une des forces de Node.js est qu'il permet d'effectuer des requêtes dites asynchrones, ce qui signifie que les différentes requêtes n'ont pas l'obligation d'être exécutées dans l'ordre d'arrivée. Cette fonctionnalité donne à Node.js une grande force étant donné la multitude de requêtes qu'un serveur peut se retrouver à gérer au même moment. De cette façon, il traite facilement une grande quantité de données en continue, contrairement à d'autres langages. Une autre raison d'utiliser Node.js et qu'il possède une prise en main simple et s'apprend rapidement.
Le listing 4.2 présente l'exemple "Hello World" pour Node.js[19], qui permet en quelques lignes de créer un serveur web local. Ici, un serveur affichant "Hello World" qui tourne sur l'adresse 127.0.0.1 et ouvert sur le port 3000.

```
1  const http = require('http');
2  const hostname = '127.0.0.1';
3  const port = 3000;
4  const server = http.createServer((req, res) => {
5    res.statusCode = 200;
6    res.setHeader('Content-Type', 'text/plain');
7    res.end('Hello World');
8  });
9  server.listen(port, hostname, () => {
10 console.log("Server running at http://${hostname}:${port}/");
11 });
```

Listing 4.2 – Node.js Hello Word

Le framework Koa

Koa[13] est un framework orienté web conçu pour le développement d'application web et d'API avec Node.js. Développé par la même équipe qu'Express[9], Koa se veut plus léger, plus robuste et plus flexible qu'Express. Il exploite fortement les fonctions asynchrones, manipule les callbacks à notre place et permet la gestion des erreurs avec un système de try/catch. Koa met notamment à disposition une large palette de méthodes qui facilitent la création d'un serveur web, ce qui rend Node.js plus rapide et plus simple à apprendre.

4.2.3. Endpoints

L'API respecte une architecture REST. Le client et le serveur sont donc indépendants l'un de l'autre. Pour communiquer, le client formule des requêtes au serveur et attend en retour une réponse. Très souvent, les réponses sont sous la forme d'une représentation de ressource web (une page HTML, du texte, une image, une vidéo, etc), mais cela peut aussi être simplement la création ou la modification de ressources. Afin que le serveur sache quoi faire et puisse distinguer une requête GET d'une autre, le serveur définit plusieurs *endpoints* comme illistré dans la figure 4.3. Chaque endpoint réagit d'une façon prédéterminée à chaque méthode HTTP (GET, POST, PUT, PATCH, DELETE). De cette façon, le client adresse ses requêtes aux différents endpoints et le serveur peut distinguer quelles actions il doit exécuter pour satisfaire la requête du client.

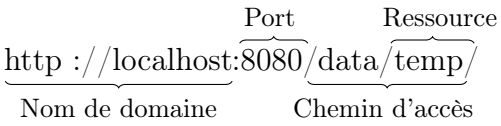


Figure 4.3. – Exemple endpoint

4.2.4. Base de données : MongoDB

Il existe deux grandes types de bases de données. Les bases de données dites relationnelles et celles dites non-relationnelles. Les premières organisent les données sous forme de tableaux en deux dimensions appelés *tables*, et chaque information dans la base données possède une relation avec les différentes tables. Les bases de données non-relationnelles ne possèdent pas cette notion de table et de relation. Les informations sont stockées sous la forme de collection ou de dictionnaire (format clé-valeur) et généralement en format JSON ou XML.

MongoDb[15] est une base de données de type non-relationnelle. Elle conserve les informations sous la forme de collections et en format JSON. Mongo est généralement la base de données utilisée avec Node.js, dû à sa faible complexité et son apprentissage rapide. L'écriture et la lecture des données se font rapidement et sans avoir à passer par des requêtes ou des tables. Le format JSON des données est également très pratique pour travailler et/ou communiquer avec d'autres APIs web ou services. La manipulation des données est aussi très facile dans ce format.
Les bases de données de types non-relationnelles sont donc très privilégiées pour le développement d'API web, car pouvant stocker une grande quantité de données rapidement et de façon dynamique.

Exemple d'opération créant une nouvelle entrée dans la base de données MongoDB. Dans cet exemple, *newListing* est l'information en format JSON à stocker et *collection* et le nom de la collection où la donnée sera conservée.

```
1 async function createListing(newListing,collection){
2     await client.connect();
3     const result = await client.db("Db_test").collection(collection).insertOne(
          newListing);
```

```
4     //console.log(`new listing inserted with id ${result.insertedId}`)
5     await client.close();
6 };
```

Listing 4.3 – Exemple d'insertion d'un objet JSON dans MongoDB (Node.js)

4.2.5. Partie orientée évènements

Le serveur se comporte également comme un producteur pour la partie orientée événements. A chaque fois, qu'une requête HTTP POST est faite par le bureau, le serveur enregistre la nouvelle information dans sa base de données, puis dans le cas de la température par exemple, le serveur lit l'information reçue et si cette information est supérieure ou égale à la valeur maximale définit par le bureau (la valeur maximale du bureau est conservée dans la base de données et lue à chaque comparaison), il publie un message à l'aide du protocle MQTT[17] au broker. Le broker va ensuite publier le message sur les bons "channels", afin que les clients puissent le recevoir. Pareillement pour l'état de la porte, le serveur possède une valeur de comparaison et si le nouvel état envoyé (par la requête POST) est différent de l'ancien, il change cette valeur dans la base de données et publie le nouvel état de la porte au broker. Comme le serveur utilise le protocole MQTT, il doit obligatoirement se connecter à un broker lors de son initialisation, autrement la partie orientée événements de l'API ne fonctionnera pas. MQTT est un protocole qui permet le transfert de message et utilise cette architecture spécifique EDA, "publish/subscribe". Le broker quant à lui, sert principalement à distribuer les messages sur les bons canaux. Il existe plusieurs agent de messages[2] qui proposent de gérer ce service. Dans ce projet Mosquitto[16] est utilisé comme intermédiaire pour le broker. Ainsi, un appareil fait tourner le broker Mosquitto et le producteur établit une communication avec lui grâce à son adresse Internet Protocol (IP). En se connectant, le producteur annonce les différents channels de publications au broker, et les clients n'auront plus qu'à s'abonner sur les bons canaux pour pouvoir recevoir n'importe quels changements d'états en temps réel.

MQTT

MQTT est un protocole de messagerie standardisé Open Artwork System Interchange Standard (OASIS)[20] pour l'IoT. Conçu comme un protocole de transport de messages de type "publish/subscribe", il est idéal pour la transmission d'informations et surtout pour la communication entre appareils à distance. Le protocole permet de se connecter à différents appareils avec une faible émission, utilise très peu de bande passante et permet une grande fiabilité dans le livraison des messages. MQTT possède un système de stockage temporaire des messages et permet de les renvoyer une ou plusieurs fois, si ceux-ci n'ont pas été reçus par les clients. C'est pourquoi il s'utilise beaucoup pour établir des connexions sur des canaux peu sûrs, de plus il peut facilement se crypter grâce à des protocoles comme Transport Layer Security (TLS).

Mosquitto

Mosquitto est un broker de messages implementé sur le protocole MQTT. Comme tous les brokers, il permet aux producteurs de publier des messages, de créer des canaux de

publication, et s'occupe de les redistribuer sur les différents "channels". Du côté des consommateurs, il permet aux clients de souscrire à différents canaux et de recevoir des messages. Mosquitto est souvent utilisé dans les petits projets comme celui-là, car il a l'avantage d'être simple à utiliser, d'être open source et permet de gérer une quantité suffisante de messages pour faire fonctionner une API comme celle-là.

Points à améliorer

A mon sens, deux points pourraient être ajoutés dans une version future pour améliorer/optimiser la partie orientée événements.

1. Utiliser un workflow pour effectuer le travail de comparaison des nouvelles informations envoyées par le bureau et publier les messages MQTT.
2. Paramétrer le bureau, pour qu'il devienne le producteur, en publiant lui-même les messages via MQTT au serveur du projet lorsqu'un changement d'état se produit. Le serveur devant ainsi simplement s'occuper de la bonne redistribution des messages sur les différents canaux.

4.3. Application mobile

Pour pouvoir recevoir les messages publiés par le producteur, il est nécessaire d'avoir un client abonné aux différent "channels". Une application mobile a donc été développée pour satisfaire ce besoin. L'application aurait pu simplement être un réceptionnaire à messages, mais des fonctionnalités supplémentaires ont été ajoutées pour rendre l'application plus intéressante et utilisable pour un potentiel déploiement. En plus d'être le consommateur, elle réalise aussi des requêtes HTTP GET et POST. Notamment pour obtenir et définir la température maximale, dessiner les températures sur le graphique ou encore obtenir le dernier état de la porte lors de l'ouverture de l'application. Toute autre application aurait été possible (web ou bureau) pour répondre à ce besoin, mais l'idée de développer une application mobile me paraissait la plus intéressante, car c'était l'occasion pour moi d'apprendre un nouvel outil de programmation, comme cela a déjà pu être le cas avec Node.js pour le serveur. Android Studio[4] a donc été choisi pour développer cette partie du projet.

Android Studio

Android Studio est l'Integrated Development Environment (IDE) ou environnement de développement intégré officiel pour le développement Android[3]. Cet environnement permet de générer un Android Package Kit (APK) qui est le format de fichiers pour le système d'exploitation Android. Ainsi, toutes les applications conçues avec cet outil fonctionneront sur n'importe quel appareil Android. Android Studio est un outil puissant très proche de la machine et laisse une liberté totale aux développeurs maîtrisant cet environnement. Le développement d'application se fait avec Java ou Kotlin[14] des langages orientés objets très employés dans la conception d'API. Toutefois, il nécessite l'apprentissage de nouvelles librairies et d'une nouvelle conception de design prévue pour les téléphones mobiles, comme la gestion de l'écran tactile et la création d'interface graphique mobile.

5
Conclusion

5.1. Conclusion

Les trois applications (le bureau, le serveur et l'application mobile) fonctionnent et sont capables de communiquer entre elles. Le serveur rempli l'objectif d'être à la fois REST et orienté évènements. Il détecte les changements d'états en temps réel, publie les messages sur les bons canaux et un client arrive à les réceptionner. Il répond également très bien aux requêtes HTTP du bureau et du client et il interagit avec la base de données sans problème.
L'application mobile effectue son rôle de consommateur en étant capable de recevoir en direct n'importe quel changement d'état, et peut effectuer différentes requêtes HTTP pour les besoins du client. Même si l'application reste simple, implémenter de nouvelles fonctionnalités pour de nouveaux capteurs, n'est pas un problème et sa complexité peut rapidement devenir élevée. Elle est pratiquement prête pour un déploiement réel et elle est capable de répondre aux besoins des professeurs et des étudiants grâce à sa conception polyvalente avec l'utilisation combinée de requêtes HTTP et de consommateur EDA.
L'application simulant le bureau quant à elle, a atteint son rôle et fournit les informations nécessaires au bon fonctionnement du projet. Elle pourrait au besoin simuler d'autres événements, mais sera de toute évidence remplacée par des réels capteurs dans une version finale.

Les différents codes sources du projet sont disponibles sur ce Google drive[8].

5.2. Points à améliorer

Le projet est fonctionnel, mais pourrait être perfectionné avant d'être mis à disposition. Il reste notamment encore quelques petits point à améliorer. L'application mobile pourrait être retravaillée, en séparant la partie consacrée aux professeurs de celle destinée aux étudiants. Pour ce faire, une interface d'authentification pourrait être rajoutée avec un nom d'utilisateur et un mot de passe, pour distinguer les étudiants des professeurs et afficher le bon écran d'accueil. Cela impliquerait d'implémenter une phase avec des tokens d'authentification dans le serveur. Une version iOS pourrait également être développée pour les utilisateurs de téléphone Apple, car pour l'instant l'application fonctionne seulement pour les appareils Android. Finalement, comme mentionné dans le chapitre 4, la partie orientée événements pourrait être un peu mieux optimisée.

5.3. Conclusion personnelle

Pour un premier projet de programmation web, je suis très satisfait du résultat. Le tout fonctionne et rempli les objectifs que je m'étais fixés au début du projet. J'ai beaucoup appris sur le monde de la programmation web et découvert des concepts passionnants. Pour mener à bien mon travail, j'ai usé des connaissances acquises tout au long de ces trois années de Bachelor et j'ai même appris à utiliser de nouveaux outils comme Node.js ou Android Studio, qui me serviront sans doute dans ma future carrière. Ce projet m'a beaucoup plu et je remercie une dernière fois le professeur Pasquier-Rocha de m'avoir guidé tout au long de ce projet.

A
Common Acronyms

API	Application Programming Interface
APK	Android Package Kit
EDA	Event-Driven Architecture
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
OASIS	Open Artwork System Interchange Standard
REST	Representational state transfer
TLS	Transport Layer Security
URI	Unified Resource Identifier
URL	Uniform Resource Locator
WoT	Web of Things
XML	eXtensible Markup Language
YAML	Yet Another Markup Language

B
Annexe

B.0.1. Documentation AsyncAPI en format JSON

```
  {
"asyncapi": "2.6.0",
"info": {
  "title": "MQTT API",
  "version": "1.0.0",
  "description": "Simulation of an office.\n"
},
"servers": {
  "mosquitto": {
    "url": "tcp://localhost:9000",
    "protocol": "mqtt"
  }
},
"channels": {
  "data/sub": {
    "subscribe": {
      "summary": "Information about the avaibility of the professor",
      "message": {
        "$ref": "#/components/messages/light_color"
      }
    }
  },
  "data/sub/{light_color}": {
    "parameters": {
      "light_color": {
        "$ref": "#/components/parameters/light_color"
      }
    },
    "publish": {
      "summary": "Send the porfessor's office status",
      "message": {
        "$ref": "#/components/messages/light_color"
      }
    }
  },
  "data/sub/temp": {
    "subscribe": {
      "message": {
        "$ref": "#/components/messages/temp"
```

```
      }
    }
  },
  "data/sub/temp/{temp}": {
    "parameters": {
      "temp": {
        "$ref": "#/components/parameters/temp"
      }
    },
    "publish": {
      "message": {
        "$ref": "#/components/messages/temp"
      }
    }
  }
},
"components": {
  "messages": {
    "light_color": {
      "name": "light_color",
      "contentType": "application/json",
      "payload": {
        "$ref": "#/components/schemas/light_color"
      }
    },
    "temp": {
      "name": "evaluate",
      "contentType": "application/json",
      "payload": {
        "$ref": "#/components/schemas/temp"
      }
    }
  },
  "parameters": {
    "light_color": {
      "description": "Status of the office.",
      "schema": {
        "type": "string",
        "description": "Status of the office"
      }
    },
    "temp": {
      "description": "current temperature",
      "schema": {
        "type": "integer"
      }
    }
  },
  "schemas": {
    "light_color": {
      "type": "object",
      "properties": {
        "light_color": {
          "type": "string",
          "description": "status of the office"
        },
        "timestamp": {
          "type": "string",
```

```
          "description": "date of the record"
        }
      }
    },
    "temp": {
      "type": "object",
      "properties": {
        "temp": {
          "type": "integer",
          "minimum": -20,
          "maximum": 50,
          "description": "current temperature."
        },
        "timestamp": {
          "type": "string",
          "description": "date of the record"
        }
      }
    }
  }
}
```

Listing B.1 – Documentation AsyncAPI en format JSON

C

License of the Documentation

Bibliographie

[1] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. 4, 5

Sites Web

[2] Agent de messages wikipédia. https://fr.wikipedia.org/wiki/Agent_de_messages (dernière consultation le 14 Juillet, 2023). 30

[3] Android URL. https://www.android.com/ (dernière consultation le 12 Juillet, 2023). 31

[4] Android Studio URL. https://developer.android.com/distribute (dernière consultation le 12 Juillet, 2023). 31

[5] Asyncapi : AsyncAPI overview. https://www.asyncapi.com/docs/concepts (dernière consultation le 08 Juillet, 2023). 14

[6] AsyncAPI Broker schema. https://www.asyncapi.com/docs/concepts/server (dernière consultation le 01 Août, 2023). iv, 7

[7] AsyncAPI Consumer schema. https://www.asyncapi.com/docs/concepts/consumer (dernière consultation le 01 Août, 2023). iv, 8

[8] Google Drive contenant le code source. https://drive.google.com/drive/folders/1NYkYOhyqD5Dk7KH83p6i1YCUDPTMhYke?usp=sharing (dernière consultation le 01 Août, 2023). 32

[9] Express Node.js URL. https://expressjs.com/ (dernière consultation le 12 Juillet, 2023). 28

[10] Free Documentation Licence (GNU FDL). http://www.gnu.org/licenses/fdl.txt (dernière consultation le 01 Juillet, 2023).

[11] Java Oracle URL. https://www.java.com/fr/ (dernière consultation le 10 Juillet, 2023). 26

[12] Java Swing. https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html (dernière consultation le 10 Juillet, 2023). 26

[13] Koa framework. https://koajs.com/ (dernière consultation le 08 Juillet, 2023). 28

[14] Kotlin URL. https://kotlinlang.org/ (dernière consultation le 12 Juillet, 2023). 31

[15] MongoDB documentation. https://www.mongodb.com/docs/ (dernière consultation le 08 Juillet, 2023). 28, 29

[16] Mosquitto URL. https://mosquitto.org/ (dernière consultation le 12 Juillet, 2023). 30

[17] Mqtt URL. https://mqtt.org/ (dernière consultation le 12 Juillet, 2023). 27, 30

[18] Node.js : About Node.js. https://nodejs.org/en/about (dernière consultation le 10 Juillet, 2023). 26, 28

[19] Node.js : Node.js Hello World. https://nodejs.org/en/docs/guides/getting-started-guide (dernière consultation le 12 Juillet, 2023). 28

[20] Oasis standard wikipédia. https://en.wikipedia.org/wiki/Open_Artwork_System_Interchange_Standard (dernière consultation le 14 Juillet, 2023). 30

[21] Openapi : What is OpenAPI ? https://www.openapis.org/what-is-openapi (dernière consultation le 08 Juillet, 2023). iv, 9, 10

[22] AsyncAPI Protocole schema. https://www.asyncapi.com/docs/concepts/protocol (dernière consultation le 01 Août, 2023). iv, 8

[23] Python URL. https://www.python.org/ (dernière consultation le 10 Juillet, 2023). 26

[24] Swagger URL. https://swagger.io/ (dernière consultation le 08 Juillet, 2023). 10