Architecture micro-service

TP FINAL - IRC

Philippe Roussille

3iL INGENIEURS
</les sciences informatiques>

1 Contexte : ce n'est qu'un aurevoir…

Le célèbre service IRC de CanaDuck, joyau artisanal de Roger et Ginette, tire aujourd'hui sa révérence. Après des années à patcher, ajouter des fonctionnalités « vite fait », et maintenir la bête en vie par la seule force de leur caféine… nos deux vétérans prennent une retraite bien méritée.

Mais avant de partir, ils ont eu une idée brillante : **redécomposer proprement CanaDuck**, en respectant cette fois-ci **les bonnes pratiques** et **les principes de conception des micro-services**.

Fini le gros serveur monolithique illisible pseudo-distribué ! Désormais :

- Chaque domaine fonctionnel (utilisateur, message, canal, statistique) est **autonome**, **cohérent**, et **dockerisé**
- Les services communiquent entre eux **via des API REST simples**
- L'authentification est **déléguée** à un service JWT unique
- L'entrée unique passe par un **reverse proxy dynamique Traefik**
- Et bien sûr… **les responsabilités sont séparées proprement**

À la base, ce travail devait être encadré quasiment entièrement par **Philibert Roquart**, censé à l'origine vous donner le découpage fonctionnel… Malheureusement, Philibert est **en congé maladie** et a quitté l'entreprise, son certificat mentionne une mystérieuse **« maladie vétérinaire »** : d'après lui, son **poisson rouge stresse quand il code**, et il a tenté de *noyer le poisson*… avec lui (*sic.*).

Bref, le planning a pris un petit coup. Ginette et Roger, pris de court, sont un peu confus… mais ils comptent sur **vous** pour faire avancer le projet. Le découpage en 4 services a été **réorganisé à la dernière minute**, et (miracle ou coïncidence pédagogique bien sentie ?) il correspond **pile aux 4 groupes de ce TP final**.

C'est à vous de jouer. Ne soyez pas comme Philibert. Vous voilà donc **héritiers du code** de l'IRC de CanaDuck. À vous d'en assurer la transmission… et peut-être de faire mieux encore.

2 Le fonctionnement

2.1 Micro-services : robustesse, résilience et clarté des rôles

Dans une architecture micro-services digne de ce nom, **chaque service doit être à la fois robuste et résilient** :

- Robuste, car il doit **traiter les entrées attendues** avec fiabilité.
- Résilient, car il doit **survivre aux erreurs des autres** : une dépendance temporairement indisponible ne doit **pas faire planter l'ensemble**.
- Poli, aussi : en cas de mauvaise utilisation (JWT invalide, paramètre manquant, rôle insuffisant…), un service bien conçu **rejette clairement la demande** avec un code d'erreur HTTP approprié (401, 403, 422…), **sans crasher**.

Autrement dit : *fail gracefully*, et soyez un bon voisin HTTP.

2.2 Gardez votre cap : quelle est votre responsabilité ?

Chaque groupe n'est responsable **que d'un micro-service**. Même si vous êtes tentés d'ajouter des vérifications, des stockages ou des routes qui sortent de votre périmètre : **ne le faites pas**. Faites-en ***peu***, mais ***bien***. Je veux, demain soir, un service ***fonctionel***, pas un projet commercial qui est du vent.

Concentrez-vous sur **ce que votre service est censé faire** :

- Le `user-service` ne gère pas les canaux, ni les messages.
- Le `message-service` ne modifie pas les utilisateurs.
- Le `channel-service` n'authentifie personne.
- Le `gateway` ne stocke rien.

Respecter cette séparation, c'est garantir :

- Une API claire et maintenable
- Un déploiement indépendant de chaque composant
- Une communication cohérente entre les services

2.3 Attention : vous êtes dans une architecture décentralisée

2.3.1 ATTENTION DANGER

Dans un système monolithique, tout le monde partage les mêmes variables, les mêmes objets, les mêmes fonctions, donc chacun fait sa cuisine. Mais **en micro-services**, chaque groupe code **dans son coin**, avec son propre Flask, sa propre base, sa propre vérité.

Cela signifie une chose essentielle : **L'information que vous échangez entre services doit être claire, complète, et convenue à l'avance (échange minimal, revoir le cours de FONDADEV sur le couplage... !).**

Vous ne pouvez pas "deviner" comment l'autre va appeler votre service. Il faut que **les noms des champs, leur type, leur structure JSON soient connus et partagés**.

2.3.2 Concrètement

- Si le `message-service` attend `"channel"` et `"text"` dans un POST, il faut que le client (`gateway`) les envoie tels quels.
- Si le `channel-service` retourne `"moderators"` sous forme de liste, alors le `gateway` ou le `user-service` ne doivent pas s'attendre à un dictionnaire.
- Si le JWT contient `"roles": ["admin"]`, chaque service doit comprendre que `"admin"` suffit, et ne pas attendre `"is_admin": true`.
- etc.

2.3.3 LA DOCUMENTATION EST PRIMORDIALE

Avant de commencer le code :

- Relisez ensemble les structures JSON attendues (exemples fournis)
- Mettez-vous d'accord **au sein du groupe ET entre groupes**
- Tenez à jour une fiche commune si besoin (ex. : `structures.md`)

C'est le **prix de la liberté** en micro-services : on est indépendants, certes, mais **l'interopérabilité repose sur une bonne entente** !

3 Mais comment on fait bien les choses entre nous ? (dernier point de cours)

En architecture micro-services, lorsqu'un service a besoin de connaître **qui est l'utilisateur** ou **quels sont ses droits**, deux grandes stratégies existent :

3.1 Appels internes (`chained requests`)

Chaque service interroge un autre via HTTP. Exemple : `message-service` appelle `user-service` pour vérifier si "`roger`" est admin.

Avantages

- Forte séparation des responsabilités
- Données toujours **à jour** (si rôle ou statut change)
- Pas de duplication de logique

Inconvénients

- Risque de **cascade d'indisponibilité** si un service ne répond plus
- Couplage fort entre services
- Plus complexe à tester (nécessite plusieurs services lancés)
- Peut provoquer des boucles de dépendance mal maîtrisées

3.2 Jetons JWT portés dans les requêtes

À la connexion (`/login`), le `user-service` génère un **JWT signé** contenant l'identité, les rôles, etc. Ce jeton est ensuite transmis par le client dans chaque requête HTTP :

```
Authorization: Bearer <votre_token>
```

Chaque service peut **vérifier ce token localement**, sans appel externe.

Avantages
- Aucune dépendance externe nécessaire → plus **résilient**
- Très facile à implémenter avec Flask et PyJWT
- Toutes les infos sont déjà disponibles dans le token
- Favorise un **découplage fort entre services**

Inconvénients
- Le JWT peut contenir des infos **périmées** si un rôle change (avant expiration)
- Besoin de gérer proprement la **signature et la durée de validité**
- Si mal conçu, on peut **mettre trop de choses** dans le token (risque sécurité ou performance)

3.2.1 Le choix de ce projet : JWT

Pour ce projet, je… pardon, Ginette et Roger vous imposent le choix **du JWT**, car il permet :

- De garder chaque micro-service **indépendant et testable seul**
- De simplifier les appels HTTP inter-services
- D'avoir une architecture plus proche des usages modernes

Le `user-service` est donc responsable de générer et signer les tokens. Tous les autres services doivent les **vérifier** avant d'accepter une requête authentifiée.

4 Découpage fonctionnel

4.1 Groupe 1 : `user-service` - gestion des utilisateurs, authentification et rôles globaux

4.1.1 Objectif du service

Votre mission est de développer le **micro-service chargé de l'authentification et de la gestion des utilisateurs** dans l'architecture IRC distribuée. Ce service est **central** : tous les autres services dépendent de lui pour valider l'identité des utilisateurs et connaître leurs rôles globaux (`admin`, `user`).

Le `user-service` est **le seul à gérer les comptes utilisateurs**, les statuts de connexion, les avatars, et les rôles globaux. Il émet les **jetons JWT** lors de la connexion et vérifie leur validité pour sécuriser les accès aux autres services. Il est donc un **point de vérité** pour l'identité et les droits des utilisateurs.

Ce service fonctionne de manière **stateless** pour les endpoints exposés : chaque appel authentifié passe par un token JWT transmis dans les headers. Le stockage des comptes se fait en mémoire ou via une base de données légère (BD MySQL) selon votre avancement. Il est entièrement **dockerisé** et exposé via HTTP/JSON.

4.1.2 Tâches à réaliser, organisation conseillée

Basique : Mise en place du micro-service
- Créer un projet Python basé sur **Flask** (`app.py` + structure minimale)
- Prévoir un fichier `requirements.txt` avec `flask`, `pyjwt`, etc.
- Dockeriser le service avec un `Dockerfile`
- Exposer le service sur un port interne (`5001` par exemple)
- Prévoir un volume pour stocker les données utilisateurs (BD MySQL au choix, avec SQLAlchemy)

Premiers points : Gestion des comptes
- Implémenter la route `POST /register` : création de compte (pseudo, mot de passe hashé, email)
- Implémenter la route `POST /login` : authentification et génération d'un **JWT**
  - Utiliser un secret JWT dans les variables d'environnement
  - Durée d'expiration du token configurable

Un peu plus en détail : Informations sur les utilisateurs
- `GET /whois/<pseudo>` : informations publiques (status, canaux, rôles)
- `GET /seen/<pseudo>` : dernière activité horodatée
- `GET /ison?users=roger,ginette` : utilisateurs actuellement connectés

Compléter : Gestion des profils
- `PATCH /user/<pseudo>/password` : changement de mot de passe (ancien + nouveau)
- `POST /user/status` : changement de statut (`away`, `idle`, etc.)
- `GET /user/avatar/<pseudo>` : retourne l'URL (ou un avatar de test en base64)
- `DELETE /user/<pseudo>` : suppression d'un compte

Finaliser : Gestion des rôles
- `GET /user/roles/<pseudo>` : liste des rôles globaux (`admin`, `user`)
- `POST /user/roles/<pseudo>` : ajoute un rôle à un utilisateur (**admin uniquement**)
- Faire une fonction (ou une route particulière `/make-admin/<pseudo>` accessible uniquement aux admins via JWT)

Bonnes pratiques pour l'authentification sécurisée
- Utiliser **JWT** pour toutes les routes qui nécessitent une authentification
- Créer un décorateur Flask pour vérifier le JWT (authentification + rôle si besoin)
- ***Ne stocker aucun mot de passe en clair*** (hash obligatoire, ex : `bcrypt` ou `werkzeug.security`)

4.1.3 Tests & documentation

- Écrire un fichier Markdown ou Swagger (Flasgger) documentant chaque route

— Fournir des exemples d'appel avec `requests` ou `curl` ou `Hoppscotch`
— Prévoir des cas de test pour : login invalide, token expiré, rôle manquant…
— Faire quelques tests unitaires *judicieux* avec `unittest`

4.1.4 Conseils pour ne pas vous perdre

— Ne cherchez pas à tout faire dès le début : commencez par les routes `POST /register` et `POST /login`, puis `GET /whois`.
— Isolez bien la logique d'authentification dans un fichier `auth.py` ou `utils.py`
— Assurez-vous que le JWT encode au minimum : le pseudo, les rôles, la date d'expiration
— Utilisez une clé secrète stockée dans `docker-compose.yml` (pas en dur dans le code)

4.2 Groupe 2 : `message-service` - gestion des messages publics, privés et réactions

4.2.1 Objectif du service

Votre mission est de développer le **micro-service chargé de la gestion des messages IRC**, qu'ils soient publics (dans un canal) ou privés (entre deux utilisateurs). Ce service gère également les **réactions emoji**, les **modifications de message**, les **messages épinglés** et la **recherche**.

Le `message-service` ne s'occupe **que du contenu des messages** : il suppose que les utilisateurs et les canaux existent, et il peut interroger d'autres services (comme `user-service` ou `channel-service`) pour valider certaines informations (pseudo valide, appartenance à un canal…).

Ce service fonctionne exclusivement via des **routes HTTP REST JSON**, est **stateless** côté API, et doit **vérifier le JWT** reçu dans les requêtes (pour connaître l'expéditeur). Les données peuvent être stockées en mémoire ou dans une base de données légère (BD MySQL), avec pagination facultative.

Il est entièrement **dockerisé** et exposé sur un port interne, sans interface utilisateur directe.

4.2.2 Tâches à réaliser, organisation conseillée

Basique : Mise en place du micro-service
— Créer un projet Python basé sur **Flask** (`app.py` + structure minimale)
— Prévoir un fichier `requirements.txt` avec `flask`, `pyjwt`, `sqlalchemy`, etc.
— Dockeriser le service avec un `Dockerfile`
— Exposer le service sur un port interne (`5002` par exemple)
— Prévoir un volume de données (BD MySQL recommandé)

Premiers points : Gestion des messages publics
— `POST /msg` : envoyer un message dans un canal
  — Payload : `{ "channel": "tech", "text": "coucou tout le monde" }`

  - Facultatif : `reply_to` pour un message en réponse
- `GET /msg?channel=tech` : récupérer les messages d'un canal
  - Ajouter la pagination en bonus via `offset` et `limit`

Compléter : Réactions et gestion des messages
- `POST /msg/reaction` : ajouter une réaction (`emoji`) à un message
- `DELETE /msg/reaction` : retirer une réaction
- `PUT /msg/<id>` : modifier le contenu d'un message
- `DELETE /msg/<id>` : supprimer un message (si émetteur = utilisateur connecté)

Avancé : Fonctions supplémentaires
- `GET /msg/thread/<id>` : récupérer les réponses à un message donné
- `GET /msg/pinned?channel=tech` : messages épinglés dans un canal
- `GET /msg/private?from=roger&to=ginette` : messages privés entre deux utilisateurs
- `GET /msg/search?q=moteur` : rechercher un mot-clé dans tous les messages

Bonnes pratiques pour l'authentification et la cohérence
- Vérifier le JWT dans toutes les requêtes d'écriture ou de suppression
- S'assurer que l'utilisateur émetteur correspond au JWT (et non un champ libre)
- Vérifier que le canal existe (en interrogeant le `channel-service`, ou avec un mock si besoin)
- Ne pas autoriser à modifier ou supprimer un message d'autrui
- Ajouter un champ `timestamp` pour chaque message

4.2.3 Tests & documentation

- Écrire une documentation Markdown ou Swagger (Flasgger) des routes
- Donner des exemples d'appel avec `requests`, `curl` ou `Hoppscotch`
- Prévoir des cas de test :
  - envoi de message sans canal
  - modification d'un message d'un autre utilisateur
  - réaction multiple au même message par un même utilisateur
- Faire des tests unitaires avec `unittest` pour :
  - Création d'un message
  - Ajout d'une réaction
  - Refus de modification par un autre pseudo (simulation JWT)

4.2.4 Conseils pour ne pas vous perdre

- Commencez par `POST /msg` et `GET /msg?channel=...` pour valider le cœur du service
- Créez une structure de données claire : chaque message a un `channel`, `from`, `text`, `timestamp`, `id`, `reactions`

— Séparez bien les fonctions métiers (ex : envoyer un message) de l'authentification (décorateurs)
— Mettez des `print()` ou des logs côté console pour tracer les appels pendant les tests
— Préparez des données de test simples : 2 canaux, 3 messages, 2 utilisateurs (fictifs)

4.3 Groupe 3 : `channel-service` - gestion des canaux IRC (création, rôles locaux, ACLs)

4.3.1 Objectif du service

Votre mission est de développer le **micro-service chargé de la gestion des canaux IRC** : création, configuration, sujets, invitations, rôles locaux, bannissements, etc. C'est le service qui contrôle la **structure des salons de discussion**, ainsi que les **droits locaux** des utilisateurs dans chaque canal.

Le `channel-service` ne gère **ni les utilisateurs globaux** (ça, c'est le `user-service`), **ni les messages** (c'est le `message-service`), mais il doit être capable de :

— vérifier l'existence d'un canal,
— savoir **qui a le droit** d'y accéder,
— maintenir une liste des **membres, modérateurs, invités, bannis**,
— et **enregistrer les réglages locaux** (mode, topic…).

Ce service fonctionne exclusivement via des **routes HTTP REST JSON**, est **stateless** côté API, et utilise les **JWT** pour valider les identités et les rôles des appelants. Il peut stocker les canaux dans une base locale (BD MySQL).

4.3.2 Tâches à réaliser, organisation conseillée

Basique : Mise en place du micro-service
— Créer un projet Python basé sur **Flask**
— Ajouter un `requirements.txt` avec `flask`, `pyjwt`, `sqlalchemy`, etc.
— Dockeriser le service (`Dockerfile`)
— Exposer le service sur un port interne (`5003` par exemple)
— Prévoir un fichier ou une base BD MySQL pour stocker la structure des canaux

Premiers points : Création et lecture des canaux
— `GET /channel` : liste les canaux publics
— `POST /channel` : créer un nouveau canal
  — Payload : `{ "name": "tech", "private": false }`
  — Vérifier que le nom est unique
— `GET /channel/<nom>/users` : liste des utilisateurs dans un canal

Compléter : Configuration et droits locaux
— `PATCH /channel/<nom>` : modifier le mode et/ou le sujet
  — Payload : `{ "topic": "développement", "mode": "+r" }`
  — Vérifier que seul un owner ou modérateur peut le faire
— `POST /channel/<nom>/topic` : changer le sujet

— `POST /channel/<nom>/mode` : changer les modes (`lecture seule`, `privé`, etc.)
— `GET /channel/<nom>/config` : renvoyer la configuration complète du canal (topic, modes, rôles…)

Gérer l'accès : rôles locaux, invitations, bans
— `POST /channel/<nom>/invite` : inviter un utilisateur
  — Payload : `{ "pseudo": "roger" }`
  — Vérifier que seul un modérateur ou owner peut inviter
— `POST /channel/<nom>/ban` : bannir un utilisateur
  — Payload : `{ "pseudo": "roger", "reason": "spam" }`
— ACLs locales à maintenir :
  — `owner` (créateur du canal)
  — `moderators` (peuvent modifier le sujet, inviter, bannir)
  — `invited` (accès aux canaux privés)
  — `banned` (ne peut pas entrer)
— `DELETE /channel/<nom>` : supprimer un canal (owner only)

Bonnes pratiques pour l'authentification et la validation
— Toutes les modifications doivent être protégées par JWT
— Le JWT doit permettre d'identifier l'appelant (`pseudo`) pour vérifier ses droits dans le canal
— Le service peut (facultatif) interroger le `user-service` pour vérifier que l'utilisateur existe
— Le service ne doit pas gérer les messages : seulement la structure et les droits

4.3.3 Tests & documentation

— Écrire une documentation (Markdown ou Swagger/Flasgger) listant toutes les routes
— Donner des exemples d'appels avec `curl`, `requests`, `Hoppscotch`
— Prévoir des cas de test :
  — Création de canal déjà existant
  — Modification du sujet par un utilisateur non modérateur
  — Tentative de rejoindre un canal privé sans être invité
  — Vérification des ACLs locales
— Écrire quelques tests unitaires avec `unittest` :
  — Création de canal
  — Vérification des rôles locaux
  — Ajout d'une invitation ou d'un bannissement
  — Lecture de la configuration d'un canal

4.3.4 Conseils pour ne pas vous perdre

— Commencez par `POST /channel` et `GET /channel` pour poser les bases
— Mettez en place une structure claire par canal
— Vérifiez bien dans chaque route **si l'utilisateur a le droit d'agir**
— Stockez les données dans une base BD MySQL (avec SQLAlchemy)

— Vous pouvez créer un fichier `acl.py` ou `utils.py` pour centraliser les vérifications de droits

4.4 Groupe 4 : `gateway-service` + `stats-service` - point d'entrée unique et statistiques

4.4.1 Objectif du service

Votre mission est double :

1. Créer un **point d'entrée unique (`gateway-service`)** qui sert d'interface unique entre les clients (navigateur, front-end, outils comme Hoppscotch) et les micro-services internes (`user`, `message`, `channel`).
2. Développer un **micro-service de statistiques (`stats-service`)** capable d'agréger les données (messages, utilisateurs, canaux) et de produire des métriques utiles.

Le `gateway-service` est le **seul service exposé publiquement** via le reverse proxy **Traefik**. Tous les appels passent par lui. Il agit comme un **proxy intelligent** : il relaie les requêtes vers les bons services, vérifie les droits via les JWT, et peut enrichir ou combiner les réponses.

Le `stats-service`, lui, n'est **pas directement accessible depuis l'extérieur** : il communique avec les autres services (notamment `message-service`) en interne, via HTTP.

4.4.2 Tâches à réaliser, organisation conseillée

Basique : Mise en place du gateway

— Créer un projet Python basé sur **Flask**
— Ajouter un `requirements.txt` avec `flask`, `requests`, `pyjwt`, etc.
— Dockeriser le service (`Dockerfile`)
— Exposer le service en interne sur un port (`5004` par exemple)
— Configurer les **labels Traefik** pour exposer le gateway sur `/` en HTTP

Exemple (dans `docker-compose.yml`) :

```
labels:
  - "traefik.enable=true"
  - "traefik.http.routers.gateway.rule=PathPrefix(`/`)"
  - "traefik.http.services.gateway.loadbalancer.server.port=5004"
```

Première mission : reverse proxy intelligent

— Accepter des requêtes entrantes (`/register`, `/login`, `/msg`, etc.)
— Détecter la route et **rediriger dynamiquement** la requête vers le bon service :
  — `/register`, `/login` → `user-service`
  — `/msg`, `/msg/private` → `message-service`
  — `/channel` → `channel-service`
— Utiliser **`requests`** pour relayer les appels REST
— Conserver les **headers (Authorization)** pour que les autres services puissent lire le JWT

Sécurité et contrôle d'accès

- Analyser le JWT avant de transmettre certaines requêtes
- Vérifier :
  - l'utilisateur est connecté,
  - l'utilisateur est autorisé à modifier un message ou canal (si l'info est vérifiable à ce niveau),
  - en cas d'échec : retourner une réponse `403 Forbidden` sans appeler le service cible.
- Ajouter un décorateur Flask `@require_jwt()` pour valider les jetons

Route spéciale : agrégation multi-service

- `GET /fullinfo?user=roger&channel=tech` → doit interroger :
  - `user-service` pour `whois`
  - `channel-service` pour la config du canal
  - `message-service` pour les derniers messages du canal
- Fusionner les réponses et retourner un seul JSON :

```
{
  "user": { "pseudo": "roger", "roles": ["user"] },
  "channel": { "name": "tech", "topic": "dev", "moderators":
  ↪  [...] },
  "last_messages": [{ "from": "roger", "text": "Hello world" },
  ↪  ...]
}
```

Deuxième mission : statistiques Créer un micro-service `stats-service` :

- Exposer les routes suivantes :

Route	Description
`GET /stats/active-channels`	Canaux les plus actifs
`GET /stats/hourly-activity`	Nombre de messages par heure
`GET /stats/messages-per-user`	Volume total de messages par utilisateur
`GET /stats/top-reacted-messages`	Messages les plus réactés

- Ce service peut récupérer les données :
  - soit en interrogeant le `message-service` via API interne,
  - soit via un stockage partagé si vous mettez en place une base (facultatif).
- Le `gateway-service` peut aussi relayer les appels `/stats/...` vers ce service.

4.4.3 Tests & documentation

- Rédiger une documentation claire des routes (Markdown ou Swagger/-Flasgger)
- Exemples de scénarios à tester :
  - proxy d'une requête simple (`/register`)

  - échec si token invalide
  - `/fullinfo` avec aggregation inter-service
  - appel à `/stats/messages-per-user` avec données réalistes
- Écrire des tests unitaires pour le `gateway-service` :
  - Validation du JWT
  - Vérification de la redirection correcte des appels
  - Traitement d'erreurs (`401`, `403`, `500`…)

4.4.4 Conseils pour ne pas vous perdre

- Commencez par les redirections simples : `POST /login`, `POST /register`
- Mettez un `print()` pour chaque appel proxifié, avec nom de route → destination
- Isolez votre logique dans une fonction `proxy_request(service_url)` ou équivalent
- Testez avec `curl -H "Authorization: Bearer <...>"` en local
- Ajoutez un `config.py` ou `.env` avec les URLs des services internes
- Pour le `stats-service`, commencez par du **faux calcul** (ex. données en dur), puis branchez l'API réelle plus tard

5 Tenue de groupes - à formaliser dès le début du projet

Vous avez **6h en tout**. C'est **court**. La priorité absolue est la **clarté** dès la première heure : vous devez **vous accorder sur les structures**, les **routes clés**, et **prévoir des appels croisés**. Ce TP est aussi un **travail de coordination**.

5.1 À faire dans la première heure

- Chaque groupe doit proposer et documenter **une structure JSON claire** pour ses objets principaux (utilisateur, message, canal).
- Un format de **JWT unique** doit être utilisé par tous.
- Tous les groupes doivent partager :
  - les **noms des routes**,
  - les **champs attendus dans les requêtes et réponses**,
  - et les **codes HTTP standards** (`200`, `201`, `400`, `403`, `404`, etc.).

Conseil : créez un fichier partagé (Markdown ou tableau sur papier) récapitulant toutes les routes des services. Ce document vous servira pour la documentation OpenAPI à faire avec Flasgger !

5.2 Structures JSON recommandées (exemples)

Ces structures ne sont **pas obligatoires** mais **fortement recommandées** pour éviter les malentendus entre groupes.

5.2.1 Token JWT

Mettez-vous d'accord sur la clé JWT à utiliser, du genre `ce-projet-est-horrible`, `on-ny-arrivera-jamais`, etc.

Contenu typique (payload) du JWT à transmettre dans `Authorization: Bearer ...` :

```
{
  "pseudo": "roger",
  "roles": ["user", "admin"],
  "exp": 1720917617
}
```

5.2.2 Utilisateur (`user-service`)

Doit comprendre :

- un pseudo
- un email
- un status (en ligne, absent, en train de manger, barbote dans son bain, etc.)
- une ***liste*** de ses rôles (par défaut, `utilisateur`)
- une indication de sa dernière activité (dernière fois vu)
- une url de son avatar (ou une représentation en base64)

5.2.3 Message (`message-service`)

Doit comprendre :

- une id
- l'utilisateur qui l'a écrit
- le canal (éventuel)
- le contenu (texte)
- l'horodatage du message
- l'id du message de réponse si réponse (pour faire des fils de discussion)
- un dictionnaire de réactions (emoji → liste de pseudos)

5.2.4 Canal (`channel-service`)

Doit comprendre :

- le nom du canal
- si le canal est privé
- le sujet (description) du canal
- le propriétaire du canal
- les invités
- les bannis
- les modes (`modes` est une liste d'indicateurs personnalisés : lecture seule, modéré, etc.)

5.2.5 Whois (`user-service`)

Doit contenir :

— le pseudo
— le statut
— les rôles
— les canaux
— la dernière fois où il a été vu

5.3 Rôles dans chaque groupe

Pour éviter le chaos, **je vous conseille de désigner les rôles suivants dans chaque groupe** :

Rôle	Mission
Responsable doc	Écrit un fichier clair (`README.md` ou `routes.md`) décrivant les routes REST du service
Responsable tests	Prépare les appels `curl`, `Hoppscotch`, et quelques tests unitaires (`unittest`)
Dev principal Python	Se concentre sur le code et la conteneurisation
Coordinateur API (groupe 4)	Pour le **gateway-service**, s'assure que les **routes de tous les services sont bien accessibles** via le gateway.

ATTENTION, pour le**groupe 4** : Vous êtes responsables de la fusion dans `docker-compose.yml`. C'est à vous de vérifier que tous les services tournent et que le reverse proxy Traefik redirige correctement.

5.4 Rappels importants

— **Tous les appels REST doivent renvoyer du JSON propre**, avec :
  — un code HTTP correct (`200`, `201`, `400`, `403`, `404`)
  — une structure simple : `{ "error": "Texte explicite" }` ou `{ "result": ... }`
— **Pas de gros frameworks** : restez avec Flask et `requests`, tout simple.
— **Pas de micro-optimisations** : priorité à des routes fonctionnelles et claires.

5.5 Liste des routes par micro-service et responsabilités

5.5.1 `user-service` - Gestion des utilisateurs

Méthode	Route	Description
POST	`/register`	Créer un compte utilisateur
POST	`/login`	Authentifier et retourner un JWT
GET	`/whois/<pseudo>`	Infos publiques sur un utilisateur
GET	`/seen/<pseudo>`	Dernière activité horodatée
GET	`/ison?users=roger,ginette`	Liste des utilisateurs actuellement connectés
PATCH	`/user/<pseudo>/password`	Changer le mot de passe
POST	`/user/status`	Changer le statut (online, away, etc.)

Méthode	Route	Description
GET	`/user/avatar/<pseudo>`	Récupérer l'avatar
DELETE	`/user/<pseudo>`	Supprimer un utilisateur
GET	`/user/roles/<pseudo>`	Récupérer les rôles globaux
POST	`/user/roles/<pseudo>`	Ajouter un rôle global (admin only)
POST	`/make-admin/<pseudo>`	Donner le rôle admin (admin only, facultatif route)

5.5.2 `message-service` - Gestion des messages

Méthode	Route	Description
POST	`/msg`	Envoyer un message dans un canal
GET	`/msg?channel=tech`	Récupérer les messages d'un canal
POST	`/msg/reaction`	Ajouter une réaction (emoji)
DELETE	`/msg/reaction`	Retirer une réaction
PUT	`/msg/<id>`	Modifier le contenu d'un message
DELETE	`/msg/<id>`	Supprimer un message
GET	`/msg/thread/<id>`	Récupérer les réponses à un message
GET	`/msg/pinned?channel=tech`	Messages épinglés d'un canal
GET	`/msg/private?from=a&to=b`	Messages privés entre deux utilisateurs
GET	`/msg/search?q=mot`	Recherche par mot-clé dans les messages

5.5.3 `channel-service` - Gestion des canaux

Méthode	Route	Description
GET	`/channel`	Liste des canaux publics
POST	`/channel`	Créer un nouveau canal
GET	`/channel/<nom>/users`	Liste des utilisateurs d'un canal
PATCH	`/channel/<nom>`	Modifier le sujet / mode du canal
POST	`/channel/<nom>/topic`	Modifier uniquement le sujet
POST	`/channel/<nom>/mode`	Modifier les modes (lecture seule, privé, etc.)
GET	`/channel/<nom>/config`	Récupérer toute la config d'un canal
POST	`/channel/<nom>/invite`	Inviter un utilisateur
POST	`/channel/<nom>/ban`	Bannir un utilisateur
DELETE	`/channel/<nom>`	Supprimer un canal (owner only)

5.5.4 `gateway-service` - Point d'entrée unique (via Traefik)

Méthode	Route	Redirigé vers...	Description
*	`/register`, `/login`	`user-service`	Routes d'authentification
*	`/msg`, `/msg/*`	`message-service`	Tous les appels liés aux messages

Méthode	Route	Redirigé vers…	Description
*	`/channel`, `/channel/*`	`channel-service`	Tous les appels liés aux canaux
GET	`/fullinfo?user=roger&channel=tech`	(multi-services)	Agrégation `whois`, config canal, messages
GET	`/stats/*`	`stats-service`	Délégation aux statistiques

5.5.5 `stats-service` - Statistiques internes

Méthode	Route	Description
GET	`/stats/active-channels`	Canaux les plus actifs
GET	`/stats/hourly-activity`	Nombre de messages par heure
GET	`/stats/messages-per-user`	Nombre de messages par utilisateur
GET	`/stats/top-reacted-messages`	Messages les plus réactés

6 Rendu attendu de chaque groupe

Une fois n'est pas coutume, **le rendu ne se fait pas uniquement sur GitHub**. Je vous demande également de déposer votre projet **sur Moodle**, pour expérimenter un mode de collecte plus propre.

6.1 Chaque groupe doit rendre :

- Un **`README.md` clair, propre, à jour**
  - Objectif du service
  - Instructions de lancement
  - Exemple(s) d'appel (avec JWT si besoin)
- Un **`authors.md`** avec l'URL du dépôt sur GitHub (ajoutez PhilR3iL comme collaborateur) et les membres de votre groupe.
- Une **documentation d'API** complète au **format OpenAPI** (via Flasgger ou fichier YAML/JSON proprement structuré)
- Un fichier **`group.md`** indiquant :
  - Le **rôle de chaque membre** (ex : X = doc, Y = dev principal, Z = tests)
  - Le **déroulé du TP** (log heure par heure des avancées et soucis rencontrés)
    - Exemple :

```
9h00 : finalisation de /register (X)
9h30 : erreur 500 sur /msg, contact avec groupe 2 (Y)
10h00 : résolution en ajoutant vérif sur champ "channel"
```

- Un **`Dockerfile`** pour votre service - si vous utilisez une base de données (MySQL), ajoutez un `docker-compose.yml` fonctionnel.
- Tout fichier utile à la compréhension : jeux de test, exemples d'appels `curl`, captures, fichiers de conf, etc.

6.2 Modalité de dépôt : GitHub et Moodle

Une fois votre service terminé et *pushé proprement* sur GitHub, vous devez créer une **archive `.tar.gz`** et la déposer sur Moodle. Voici comment faire :

```
# 1. Cloner le dépôt si besoin
git clone https://mon.url/mon-depot.git
cd mon-depot
# 2. Remonter d'un cran pour archiver proprement
cd ..
tar -czf mon-depot.tar.gz mon-depot/
```

> **Nom du fichier :** `groupe-1-user.tar.gz`, `groupe-2-message.tar.gz`, etc.

6.3 Barème indicatif (20 pts par groupe)

Critère	Points
Fonctionnalité des routes	/8
JWT et vérifications d'accès	/3
Dockerisation fonctionnelle	/2
README clair et utile	/2
Documentation API (Swagger ou .md)	/2
group.md bien renseigné	/2
Bonus / esprit d'initiative	+1

> Des points peuvent être retirés en cas de service mal séparé, routes instables, ou code fouillis.

6.4 Défi demi-classes - 2 points bonus potentiels (pour chaque groupe de la demi-classe)

Je demande à **chaque demi-classe** (représentées par leurs groupe 4) de rendre **un dépôt Git commun supplémentaire** contenant :

- Un `docker-compose.yml` complet
- Tous les services branchés ensemble
- Le reverse proxy **Traefik** en configuration minimale
- Un petit script de test ou `curl` automatisé qui prouve que ça tourne

Indices pour décrocher les 2 pts bonus :

- Nommez vos conteneurs proprement dans `docker-compose.yml`
- Utilisez bien les labels Traefik (`PathPrefix`, ports internes…)
- Le gateway ne doit rien stocker mais tout relayer
- Un README global avec instructions de lancement fait toute la différence

7 Et voilà ! :)

Deux mots : bon courage, et amusez-vous bien !