Nantes Université

Modélisation d'une application Clients-Serveur à base de micro-services

SAE 4.1
Développement d'une application complexe (parc. REAL)
Architecture logicielle (R4.01) et Automates et Langages (R4.12)

Réalisé par : Lohan Boeglin, Théo Phan, Samuel Antunes, Sacha Chauvel, Paul-Emile Becqart

Encadré par : M. Jean-François Berdjugin (Architecture logicielle)
M. Christian Attiogbé (Automates et Langages)

Nantes – 17 avril 2025

IUT de Nantes – BUT2 Informatique 2024-2025, Groupe 1_4

Table des matières

1 Introduction

Dans le cadre de la SAE 4.1 du BUT2 Informatique, nous avons été amenés à concevoir une application reposant sur une architecture Clients-Serveur, basée sur des micro-services. L'objectif principal de ce projet est de développer plusieurs services indépendants, capables de communiquer entre eux, et facilement déployables sur les machines virtuelles de l'IUT.

Ce livrable se concentre plus particulièrement sur l'analyse et la modélisation de l'un de ces micro-services, en s'appuyant sur les enseignements du module R4.12 « Automates et Langages ». En effet, la modélisation à l'aide d'automates permet d'anticiper et de structurer le comportement d'un micro-service, tout en facilitant son implémentation, sa vérification et sa maintenance.

L'approche adoptée ici est donc double : d'une part, une conception logicielle respectant les principes de modularité, de découplage et de communication inter-services (R4.01 — Architecture logicielle) ; d'autre part, une modélisation rigoureuse des comportements grâce aux automates à états finis (R4.12 — Automates et Langages).

Ce document présente tout d'abord l'application et ses micro-services, puis détaille les modèles d'automates élaborés. Nous montrons ensuite comment ces modèles ont guidé l'implémentation et les tests, avant de conclure par une réflexion sur les apprentissages et les difficultés rencontrés.

2 Présentation de l'application

2.1 Description générale

L'application développée repose sur trois micro-services distincts, chacun ayant un rôle bien défini dans le fonctionnement global de notre système.

— **Micro-service Cartes** : Ce service est responsable de la création et du stockage des cartes dans une base de données MongoDB. Les cartes sont générées dynamiquement à partir de deux API externes : l'une fournissant des images, l'autre des prénoms. Ces données externes permettent de générer des cartes uniques, composées d'un nom, d'une image, et de certains attributs tels que la rareté. Les liens vers ces API sont répertoriés dans la section *Références*. Des exemples de réponses ou d'intégration peuvent être consultés en *Annexe*, le cas échéant.

— **Micro-service Utilisateurs** : Ce service gère la création et la gestion des utilisateurs. Chaque utilisateur possède un pseudonyme, un mot de passe, ainsi qu'un inventaire de cartes (stockées sous forme d'identifiants). Chaque compte est également associé à une quantité de *coins* (monnaie virtuelle de l'application), et à un système de *boosters*. Lors de la création d'un compte, deux boosters gratuits sont octroyés. Ces boosters peuvent ensuite être obtenus via un système de réinitialisation toutes les 12 heures ou achetés contre des coins. Les boosters permettent d'obtenir des cartes selon différents types et niveaux de rareté. Ce service est aussi en charge de l'authentification : il génère les jetons JWT, les tokens d'accès et de rafraîchissement, et gère leur renouvellement. Les données utilisateurs sont stockées dans une base MongoDB dédiée.

— **Micro-service Proxy** : Ce service agit comme point d'entrée unique de l'application. Il centralise les requêtes des clients, effectue la vérification de l'identité via les jetons JWT, et redirige les requêtes vers les micro-services appropriés. L'objectif est de garantir que seuls les utilisateurs authentifiés peuvent accéder aux services sensibles, tout en encapsulant la logique de sécurité dans une seule couche.

Cette architecture modulaire, basée sur des micro-services spécialisés, permet une meilleure maintenabilité, une répartition claire des responsabilités et une plus grande flexibilité pour les évolutions futures de l'application.

2.2 Architecture logicielle

L'application suit une architecture de type **Clients-Serveur distribuée**, basée sur une communication entre micro-services. L'objectif est de garantir une séparation claire des responsabilités, une meilleure scalabilité, et une facilité de maintenance ou de déploiement sur différentes machines virtuelles.

— Le **client** interagit uniquement avec le **micro-service proxy**, qui sert de point d'entrée unique vers l'ensemble de l'architecture.

— Le **micro-service proxy** se charge de vérifier l'authentification de chaque requête à l'aide du jeton JWT (JSON Web Token), et redirige ensuite les requêtes vers le micro-service concerné.

— Le **micro-service utilisateurs** gère l'authentification, la création de comptes, la gestion des boosters, des cartes possédées et des coins de chaque utilisateur. Il interagit avec une base MongoDB dédiée pour le stockage des données utilisateurs.

— Le **micro-service cartes** est responsable de la création, du stockage et de la génération aléatoire de cartes à partir d'APIs externes. Il stocke les cartes dans une base MongoDB distincte.

Ce découpage rend chaque service indépendant, testable et potentiellement déployable sur une VM différente. Les communications inter-services se font via des requêtes HTTP REST.

L'architecture globale de l'application est représentée sur la **Figure 1**.

2.3 Micro-services développés

Chacun des micro-services a été développé de manière indépendante, avec des technologies adaptées à ses besoins spécifiques. Cette section présente les choix techniques ainsi que les responsabilités précises de chaque service.

Micro-service Cartes

— **Rôle principal** : Générer dynamiquement des cartes et les stocker dans une base MongoDB.

— **Technologies utilisées** :

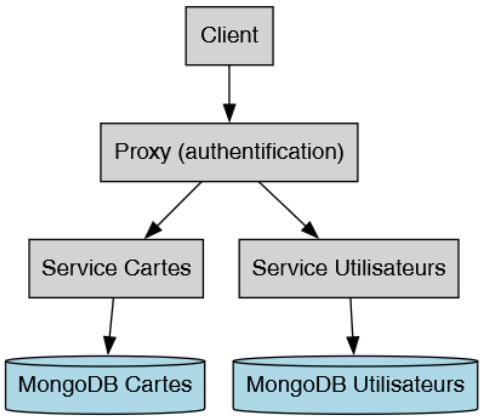

FIGURE 1 – Architecture Clients-Serveur distribuée de l'application

  - **Langage** : JavaScript (module .mjs)
  - **Environnement** : Node.js
  - **Framework** : Express
  - **Base de données** : MongoDB
  - **Appels externes** : deux APIs (noms et images) pour générer des contenus de carte aléatoires.
- **Fonctionnalités principales** :
  - Génération de cartes aléatoires (nom + image + rareté).
  - Stockage et consultation de cartes existantes.
  - Interface REST pour créer ou récupérer des cartes.

Micro-service Utilisateurs

- **Rôle principal** : Gérer les utilisateurs, leur collection de cartes, leur monnaie virtuelle et leur accès via authentification.
- **Technologies utilisées** :
  - **Langage** : JavaScript (module .mjs)
  - **Environnement** : Node.js
  - **Framework** : Express
  - **Base de données** : MongoDB
  - **Sécurité** : Hachage des mots de passe via `crypto` de Node.js et de salt
- **Fonctionnalités principales** :
  - Création et authentification d'un compte utilisateur.
  - Attribution et ouverture de boosters.
  - Gestion de la monnaie virtuelle (*coins*).
  - Association de cartes à un utilisateur (via leurs identifiants).

Micro-service Proxy

- **Rôle principal** : Servir de passerelle unique sécurisée entre les clients et les micro-services internes.
- **Technologies utilisées** :

  - **Langage** : JavaScript (module .mjs)
  - **Environnement** : Node.js
  - **Framework** : Express
  - **Sécurité** : Vérification des JWT pour l'accès sécurisé aux routes protégées.
- **Fonctionnalités principales** :
  - Vérification de l'authentification via JWT.
  - Redirection des requêtes valides vers les services utilisateurs ou cartes.
  - Gestion centralisée des erreurs et des droits d'accès.

3 Modélisation par automates

3.1 Pourquoi modéliser avec des automates ?

La modélisation par automates à états finis (ou automates finis déterministes) joue un rôle fondamental dans la conception de systèmes distribués, et en particulier dans les architectures orientées micro-services. Cette démarche permet de représenter de manière formelle et rigoureuse les comportements attendus d'un service, ainsi que les transitions possibles entre différents états du système.

Clarté et structuration du comportement L'un des avantages majeurs de l'utilisation des automates est la clarté qu'ils apportent dans la définition des scénarios d'utilisation. Un automate décrit les différents états dans lesquels un micro-service peut se trouver, ainsi que les événements (ou actions) qui déclenchent les transitions entre ces états. Cela permet de :

- mieux visualiser la logique métier d'un service,
- anticiper les comportements attendus dans différentes situations (y compris les cas limites),
- identifier plus facilement les erreurs ou incohérences dans la conception.

Aide à l'implémentation et à la vérification Les automates servent de référence pendant le développement : ils guident la structure du code, notamment dans la gestion des routes, des conditions ou des transitions d'état. Ils facilitent aussi l'écriture de tests, en permettant de générer des séquences d'actions possibles qui peuvent être directement traduites en tests unitaires ou fonctionnels.

Documentation technique claire Un automate agit comme une documentation visuelle et technique. Il peut être compris aussi bien par les développeurs que par des intervenants extérieurs (enseignants, partenaires, collègues...), même sans entrer dans les détails du code. Cela favorise la communication au sein de l'équipe.

Maintenabilité et évolutivité Lorsque de nouvelles fonctionnalités doivent être ajoutées, ou des comportements modifiés, l'automate sert de point de départ pour réfléchir aux impacts sur l'ensemble du système. Il devient alors plus facile d'ajuster ou d'étendre les transitions sans altérer les comportements déjà validés.

Adapté aux micro-services Dans une architecture distribuée, chaque micro-service a un cycle de vie bien défini et des interactions précises avec les autres composants. Les automates permettent de modéliser non seulement le comportement interne de chaque micro-service, mais aussi leurs interactions, via des modèles d'orchestration ou de coordination.

En résumé, la modélisation par automates est une méthode puissante et adaptée pour analyser, concevoir, implémenter et faire évoluer des micro-services dans une application distribuée. Elle renforce la robustesse, la lisibilité et la fiabilité globale du système.
Les images des automates sont disponibles en plus grande taille au format vertical dans la section Annexes 7. (Le format vertical peut amener à des erreurs dans l'affichage des liaisons, c'est pour cela que nous avons optés pour afficher celui horizontal dans la suite.

3.2 Automates des micro-services

Dans cette section, nous présentons les automates correspondant aux comportements principaux de chaque micro-service développé dans le cadre de l'application. Ces automates ont servi de support pour la conception et l'implémentation du code, ainsi que pour la réalisation des tests.

3.2.1 Micro-service Utilisateurs

Comportement général Ce micro-service est chargé de la gestion des comptes utilisateurs, de l'authentification, des boosters, et de la monnaie virtuelle (*coins*). Il interagit avec le client via le proxy, et stocke les données utilisateurs dans une base MongoDB dédiée. Le comportement modélisé ici concerne principalement le cycle de vie d'un utilisateur : de la création de compte à la récupération de cartes via un booster, en tenant compte des vérifications nécessaires (authentification, solde de coins...).

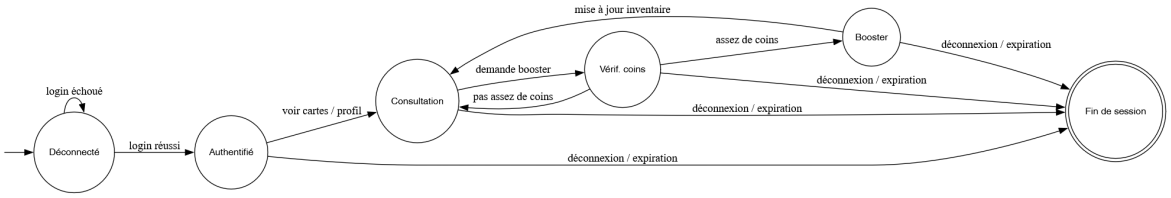

FIGURE 2 – Automate du micro-service Utilisateurs avec vérification des coins

Automate de gestion des utilisateurs Voir la Figure 2 pour une visualisation complète du comportement modélisé.

Description des états

- **Déconnecté** : l'utilisateur n'est pas encore connecté ou a été déconnecté (manuellement ou suite à l'expiration du token).
- **Authentifié** : l'utilisateur est connecté avec un jeton JWT valide.
- **Consultation** : l'utilisateur accède à son profil ou inventaire.

— **Vérification des coins** : étape intermédiaire pour contrôler si l'utilisateur peut ouvrir un booster (coins suffisants ou booster gratuit disponible).
— **Booster** : ouverture du booster et génération des cartes.
— **Fin de session** : état final, représentant la fin de l'interaction (volontaire ou non).

Transitions principales

— **Déconnecté** → **Authentifié** : suite à une connexion réussie.
— **Authentifié** → **Consultation** : l'utilisateur consulte son inventaire ou son profil.
— **Consultation** → **Vérification des coins** : l'utilisateur tente d'ouvrir un booster.
— **Vérification des coins** → **Booster** : l'utilisateur a suffisamment de coins ou un booster gratuit.
— **Vérification des coins** → **Consultation** : l'utilisateur ne peut pas ouvrir de booster.
— **Booster** → **Consultation** : les cartes sont ajoutées à l'inventaire, retour à la navigation.
— **À tout moment** → **Fin de session** : si l'utilisateur est déconnecté ou si le token expire.

Exemple de parcours d'exécution `Déconnecté → Authentifié → Consultation → Vérification des coins → Booster → Consultation → Fin de session`

3.2.2 Micro-service Cartes

Comportement général Le micro-service **Cartes** est responsable de la génération, de la récupération et du stockage des cartes dans une base MongoDB dédiée. Il est utilisé par le proxy, qui appelle le micro-service Utilisateurs en parallèle lorsqu'un booster est ouvert. Lors de la demande de cartes, le service décide dynamiquement s'il doit générer de nouvelles cartes (via des APIs externes pour les images et les prénoms) ou en récupérer depuis la base de données existante, en fonction du nombre maximum autorisé (10 000).

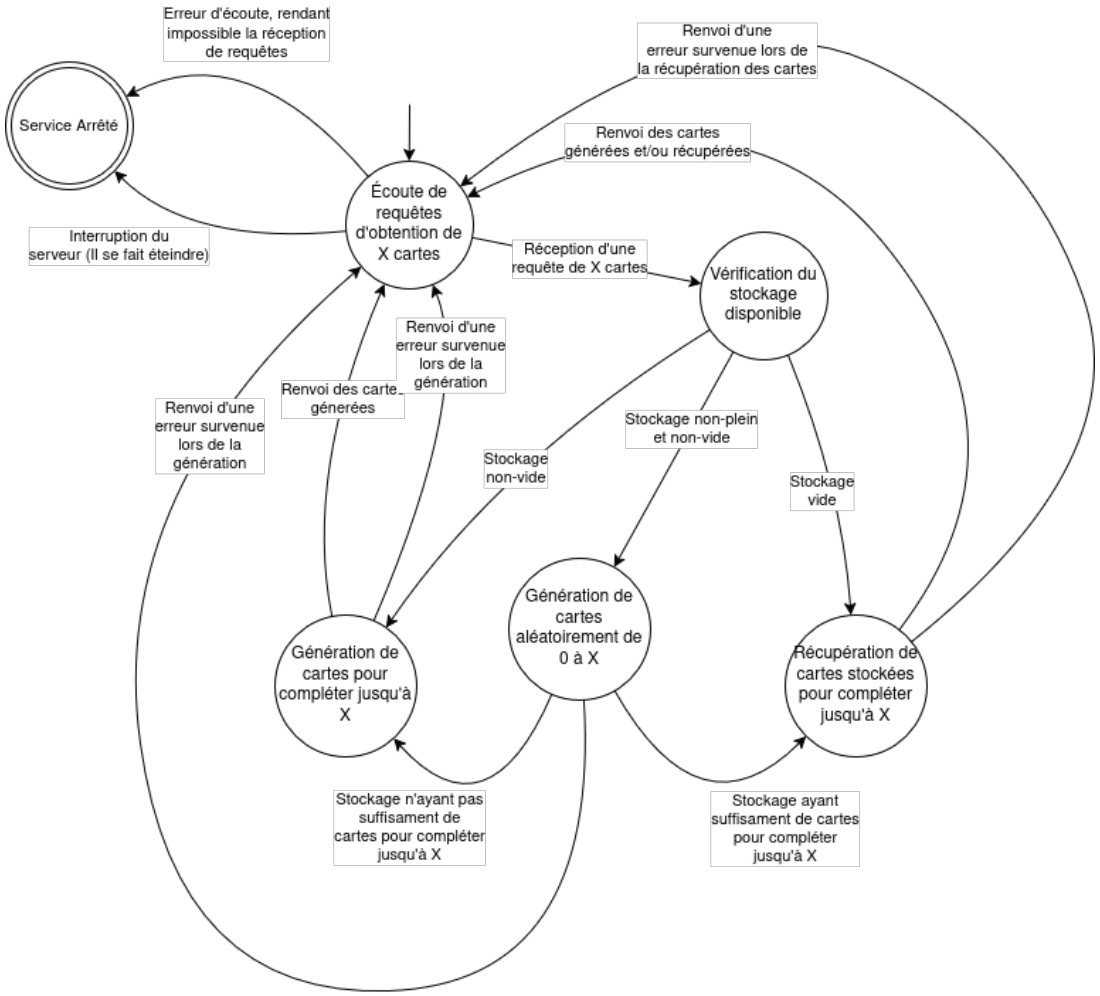

FIGURE 3 – Automate du micro-service Cartes

Automate de génération de cartes Voir Figure 3 pour une représentation du comportement.

Une remarque importante : Les cartes qui sont générées sont directement stockées dans la base de données juste après la fin de la génération, même si cela n'est pas montré explicitement par l'automate ; cela est assumé.

Description des états

- **Demande de cartes** : réception d'une requête (souvent via un booster ouvert).
- **Vérification stockage** : vérifie le nombre de cartes actuellement stockées afin de pouvoir réaliser une décision sur le nombre de cartes à générer et à récupérer de la base de données.
- **Génération aléatoire de cartes** : création de 0 à X cartes via des APIs externes (si limite de stockage n'est pas atteinte).
- **Récupération aléatoire de cartes en base** : sélection aléatoire de cartes existantes / stockées pour compléter la requête avec les cartes générées précédemment si elles n'ont pas un effectif de X.
- **Envoi de la réponse** : les cartes récupérées sont renvoyées au client.
- **Fin** : fin normale du traitement.

Transitions principales

- **Demande de cartes → Vérification du stockage** : suite à une requête de X d'un utilisateur.
- **Vérification limite → Récupération en base** : si la limite (10 000 cartes par défaut) est atteinte.
- **Vérification limite → Génération de cartes** : si la limite n'est pas atteinte.
- **Génération de cartes → Récupération en base** : équilibrage entre les cartes récupérées du stockage, et de celles générées (et donc stockées par la suite).
- **Récupération en base → Envoi de la réponse** : envoi des cartes obtenues.
- **Attente de réponse → Fin** : fin normale du cycle.
- **Erreur d'écoute / connexion → Fin** : arrêt prématuré en cas d'erreur.

Exemple de parcours d'exécution `Demande de cartes → Vérification limite → Génération de cartes → Récupération en base → Envoi de la réponse → Fin`

3.2.3 Micro-service Proxy

Comportement général Le micro-service **Proxy** agit comme un point d'entrée unique entre le client et les autres micro-services du système (Utilisateurs, Cartes, Matchs, etc.). Il est responsable de la gestion des requêtes entrantes, de la redirection vers le bon service selon l'URL ou le token JWT, et éventuellement de l'authentification préliminaire. Il permet ainsi de centraliser la sécurité, la gestion des erreurs globales et les accès.

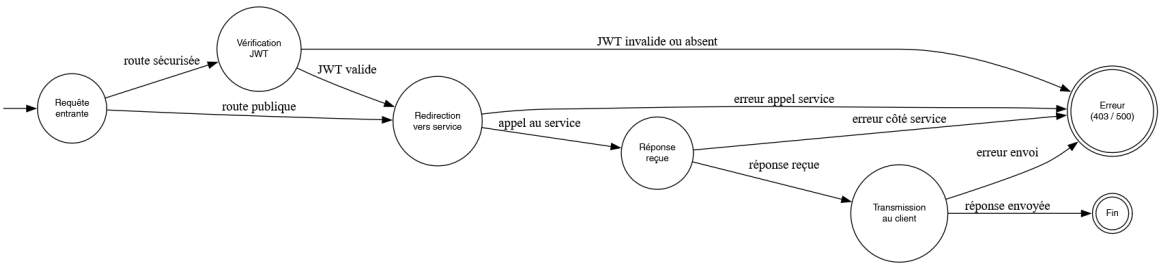

FIGURE 4 – Automate du micro-service Proxy

Automate du proxy Voir Figure 4 pour une représentation du comportement global.

Description des états

- **Requête entrante** : le proxy reçoit une requête HTTP du client.
- **Vérification JWT** : si la route nécessite une authentification, le proxy vérifie la validité du jeton.
- **Redirection vers service** : si le jeton est valide (ou non requis), la requête est redirigée vers le micro-service concerné.
- **Réponse reçue** : le proxy reçoit la réponse du micro-service appelé.
- **Transmission client** : la réponse est transmise au client.

— **Erreur (403 / 500)** : si le jeton est invalide, absent ou si une erreur serveur survient.
— **Fin** : fin normale du cycle de traitement.

Transitions principales
— **Requête entrante → Vérification JWT** : si la route est sécurisée.
— **Requête entrante → Redirection vers service** : si la route est publique.
— **Vérification JWT → Redirection vers service** : si le jeton est valide.
— **Vérification JWT → Erreur** : si le jeton est invalide ou absent.
— **Redirection vers service → Réponse reçue → Transmission client → Fin** : cycle complet.
— **À tout moment → Erreur → Fin** : si une erreur de traitement survient.

Exemple de parcours d'exécution `Requête entrante → Vérification JWT → Redirection vers service → Réponse reçue → Transmission client → Fin`

3.3 Modélisation des interactions

Lorsque plusieurs micro-services interagissent pour réaliser une fonctionnalité complète, il est essentiel de modéliser leur coordination. Cette modélisation permet de représenter la séquence des échanges et de clarifier les responsabilités entre services. Deux approches principales peuvent être utilisées : l'**orchestration**, où un service central contrôle l'enchaînement des actions, et la **coordination**, où chaque service connaît son rôle dans un échange décentralisé. Bien que cet automate est très semblable à celui de l'utilisateur figure 2, la différence est que celui là est uniquement pour un scénario d'intéraction avec les 3 services avec la précisions de chaque services.

Dans notre architecture, l'ouverture d'un booster constitue un bon exemple de coordination entre les micro-services suivants :
— **Proxy** : point d'entrée unique, il vérifie l'authentification et redirige la requête.
— **Utilisateurs** : vérifie les coins de l'utilisateur, les décrémente si nécessaire.
— **Cartes** : génère ou récupère les cartes associées au booster.

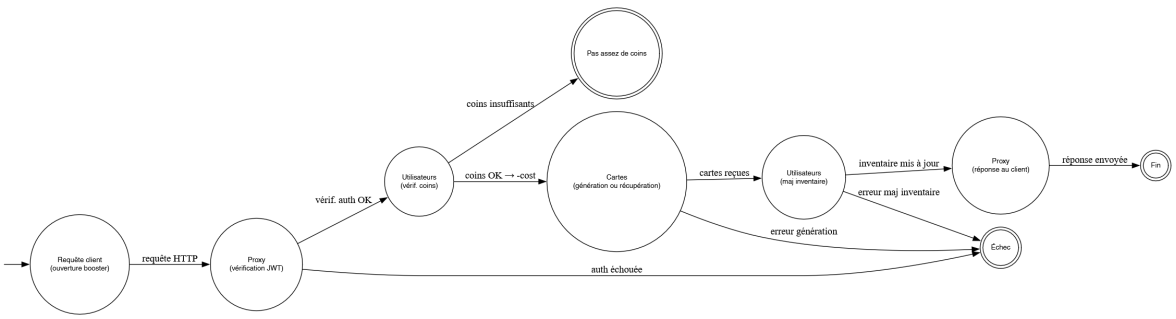

FIGURE 5 – Automate de coordination entre les micro-services

Automate d'interaction des micro-services

Description du scénario

- L'utilisateur déclenche une ouverture de booster via le client.
- Le **Proxy** vérifie le JWT et redirige la requête.
- Le service **Utilisateurs** vérifie que l'utilisateur a assez de coins.
- Si c'est le cas, il décrémente le solde, puis appelle le service **Cartes**.
- Le service **Cartes** récupère ou génère les cartes, les renvoie au service **Utilisateurs**.
- Le service **Utilisateurs** met à jour l'inventaire de l'utilisateur et retourne la réponse finale via le **Proxy**.

Exemple de parcours d'exécution `Client → Proxy → Utilisateurs (vérif. coins) → Cartes (génération) → Utilisateurs (maj inventaire) → Proxy → Client`

4 Implémentation et vérification

4.1 Lien avec les modèles

La modélisation à l'aide d'automates a permis de structurer la logique métier des micro-services dès les premières étapes de conception. Elle a servi de pont entre la phase d'analyse des besoins fonctionnels et l'implémentation concrète du code. Chaque automate représente un comportement déterministe ou conditionnel, décrivant les transitions possibles selon les actions de l'utilisateur ou les événements internes du système.

Architecture orientée micro-services La modélisation a clarifié la délimitation des responsabilités entre les micro-services. Par exemple, l'automate du micro-service *Utilisateurs* a mis en évidence les points critiques liés à l'authentification, à la gestion des jetons, et à l'utilisation de boosters, guidant ainsi le découpage en endpoints REST et en fonctions internes. De même, celui du micro-service *Cartes* a permis de distinguer les cas de génération de cartes et de récupération en base, influençant directement l'organisation des appels API et des accès MongoDB.

Logique de traitement et robustesse Les états et transitions des automates ont aussi permis de prévoir les cas d'erreur ou d'exception (authentification échouée, base inaccessible, stock insuffisant), qui ont été traduits en gestion explicite des erreurs dans le code. Cette approche a renforcé la robustesse globale de l'application, chaque scénario étant anticipé au niveau du modèle.

Coordination et orchestration L'automate d'interaction entre les services a permis de valider les enchaînements de requêtes entre micro-services : vérification des jetons côté Proxy, contrôle du solde utilisateur, appel conditionnel au micro-service *Cartes*, puis mise à jour de l'inventaire. Cette vue globale a simplifié l'implémentation du Proxy comme point central d'orchestration sans logique métier, agissant comme un répartiteur sécurisé.

Développement dirigé par les modèles En somme, les automates ont servi de support à un développement, où chaque état et transition correspond à une fonction, une route, ou un effet de bord clairement identifié dans le code. Cela a facilité la répartition du travail, la documentation technique, ainsi que les tests unitaires, chaque test pouvant être associé à une transition de l'automate.

4.2 Exemples de tests basés sur les scénarios d'executions

La modélisation par automates a permis de dériver des scénarios de tests concrets à partir des scénarios d'exécution représentatives. Chaque chemin dans un automate représente un scénario utilisateur possible, qui peut être validé par un test unitaire ou d'intégration.

Tests du micro-service Utilisateurs À partir du scénario : `Déconnecté → Authentifié → Consultation → Déconnexion`

Nous avons écrit un test intégrant les étapes suivantes :

1. Création d'un utilisateur (ou login avec compte existant)
2. Authentification et récupération du jeton
3. Appel de l'endpoint `/user` pour vérifier le profil
4. Déconnexion explicite ou expiration du token

Ce test valide le bon enchaînement des transitions et la gestion du cycle de session.

Tests du micro-service Cartes Chemin testé : `Demande → Vérification → Génération (si stock OK) → Ajout en base → Récupération → Envoi → Fin`

Un test simulateur génère une requête de cartes, en vérifiant :

— Le respect de la limite de stock (10 000)

— Le nombre de cartes générées (entre 0 et 6) si génération autorisée

— Le nombre de cartes retournées (via base + génération)

Des tests d'échec simulent aussi des erreurs réseau ou de base pour couvrir les transitions vers l'état *Échec*.

Tests du Proxy et des interactions Chemin globale : `Requête client → Authentification → Vérification coins → Appel Cartes → Mise à jour → Réponse envoyée`

Un test complet simule un utilisateur connecté ouvrant un booster :

— Le proxy valide le token (via Utilisateurs)

— Vérifie le solde de coins (>= coût)

— Décrémente le solde

— Appelle le service Cartes

— Récupère les cartes et les ajoute à l'inventaire

Chaque étape est vérifiée à l'aide de mocks ou de tests d'intégration, selon le niveau.

Avantage de cette approche L'utilisation des parcours issus des automates d'automates comme base de tests permet :

— Une couverture fonctionnelle exhaustive

— Une clarté dans les tests (chaque test correspond à un scénario)

— Une robustesse accrue (cas de succès et d'erreur modélisés)

5 Retour d'expérience

5.1 Difficultés rencontrées

La réalisation du projet a soulevé plusieurs difficultés, à la fois lors de la phase de modélisation que durant la mise en œuvre concrète du système.

Modélisation par automates La formalisation des comportements via des automates a demandé un effort important pour :

- Identifier les états réellement pertinents sans tomber dans une modélisation trop fine ou trop vague.
- Gérer les erreurs et transitions non triviales (ex. : échec de génération, état de session interrompue, limites de base de données).
- Faire correspondre précisément le modèle aux comportements réels du code, notamment lorsque des choix aléatoires ou asynchrones interviennent.

Découpage des micro-services Le découpage logique entre services a soulevé certaines interrogations :

- Où positionner certaines responsabilités (ex. : la vérification du solde de coins — côté Utilisateurs ou Proxy ?).
- Trouver un équilibre entre séparation des responsabilités et simplicité des appels inter-services.
- Gérer les dépendances tout en gardant les services relativement autonomes (ex. : éviter des appels en cascade complexes).

Coordination et orchestration Le Proxy joue un rôle central dans l'orchestration, mais cela a introduit :

- Une complexité accrue dans les tests, puisqu'il faut simuler plusieurs services.
- Un fonctionnement extrêmement dépendant des deux autres services, si l'un ne fonctionne pas, il se peut bien qu'une moitié du proxy devienne dysfonctionnelle.
- Des besoins de synchronisation pour s'assurer que tous les services réagissent de manière cohérente aux requêtes composites.

Tests et traçabilité L'usage des automates comme base des tests s'est révélé interessant, mais non trivial :

- Il fallait s'assurer que les tests reflètent bien tous les chemins pertinents dans les automates.
- Certains scénarios nécessitaient des mocks complexes ou une préparation de l'état initial de la base.
- Les erreurs réseau/API étaient parfois difficiles à simuler de manière réaliste.

Autres aspects techniques Des difficultés techniques spécifiques sont également apparues :

— La gestion du stockage en MongoDB (ex. : contraintes de taille, vérification d'unicité, accès concurrent).

— L'intégration des APIs tierces pour la génération des cartes (taux de réponse, formats d'image, erreurs).

— L'aléatoire dans la logique métier (ex. : génération de 0 à 6 cartes), qui rend difficile la reproductibilité des tests.

Malgré ces obstacles, l'approche modulaire et la modélisation formelle ont permis de construire une architecture cohérente, testable, et maintenable.

5.2 Apprentissages

Ce projet nous a permis de développer à la fois des compétences techniques concrètes et une approche plus rigoureuse en termes de conception et d'architecture logicielle. Voici les principaux enseignements tirés de cette expérience.

Approche par modélisation Utiliser des automates pour décrire le comportement des micro-services nous a appris à :

— Formaliser clairement les états possibles et les transitions dans un système distribué.

— Identifier les cas limites et les erreurs dès la phase de conception.

— Guider le découpage fonctionnel et anticiper les interactions entre services.

Architecture micro-services Nous avons acquis une meilleure compréhension de l'architecture orientée micro-services :

— Définir des responsabilités claires pour chaque service.

— Concevoir des interfaces robustes (API REST) entre composants indépendants.

— Mettre en place des mécanismes de coordination via un proxy/orchestrateur.

Développement technique Le projet nous a permis de pratiquer et consolider plusieurs aspects techniques :

— Utilisation de Node.js et des modules ES6 pour organiser le code.

— Hachage sécurisé (via `crypto`) et gestion d'authentification avec jetons.

— Manipulation de MongoDB pour stocker et interroger des données de manière efficace.

— Appels asynchrones à des APIs tierces et gestion des erreurs potentielles.

Tests et validation Nous avons appris à :

— Utiliser les automates comme base pour définir des scénarios de test pertinents.

— Rédiger des tests unitaires et fonctionnels basés sur des scénarios.

— Simuler des erreurs et tester la robustesse des services.

Documentation et rigueur Enfin, cette approche nous a sensibilisés à :

- L'importance de documenter clairement les comportements attendus dès le départ.
- L'utilité des diagrammes comme support de communication dans une équipe.
- La rigueur nécessaire pour maintenir la cohérence entre modèles et code tout au long du développement.

Au final, cette approche guidée par les modèles nous a donné une vision plus claire, plus structurée et plus professionnelle du développement logiciel distribué.

6 Conclusion

Ce projet nous a permis de concevoir, modéliser et implémenter une architecture micro-services orientée autour d'une application de gestion de cartes à collectionner. À travers l'utilisation d'automates, nous avons pu structurer et formaliser les comportements de chaque micro-service, de manière rigoureuse et visuelle.

Résumé du travail Nous avons identifié trois micro-services principaux (*Utilisateurs*, *Cartes*, *Proxy*) interagissant autour d'un cycle fonctionnel centré sur l'ouverture de boosters, la gestion de comptes, et la génération/récupération de cartes. Chaque micro-service a été modélisé sous forme d'automate pour représenter ses états et transitions, facilitant ainsi le développement, la communication d'équipe, et la couverture des cas de test.

Valeur ajoutée de la modélisation La modélisation par automates s'est révélée être un véritable atout :

- Elle a servi de base commune à toute l'équipe pour comprendre et valider les comportements attendus.
- Elle a permis d'anticiper les erreurs, de tester les limites et de structurer l'ensemble de l'architecture de façon modulaire et maintenable.
- Elle a facilité la rédaction de tests réalistes à partir de scénarios directement issues des modèles.

Perspectives et évolutions Plusieurs axes d'amélioration et d'évolution sont envisageables :

- **Déploiement** : conteneurisation via Pod ou Kubernetes
- **Observabilité** : ajout de logs centralisés, métriques, et outils de supervision pour suivre l'état des services en temps réel.
- **Sécurité** : amélioration de l'authentification, chiffrement des données sensibles, protection contre les abus d'API.
- **Expérience utilisateur** : intégration d'un front-end graphique pour interagir avec les services (projet R4.10).

En conclusion, cette démarche guidée par les modèles nous a non seulement permis de produire un système fonctionnel et modulaire, mais aussi d'adopter une posture professionnelle face à la conception logicielle distribuée. Elle constitue une base solide pour le développement futur d'applications complexes, robustes et évolutives.

7 Références

Références

[1] MongoDB Inc., *MongoDB Documentation*, https://www.mongodb.com/docs/

[2] Graphviz, *Graphviz Documentation*, 'https://graphviz.org/documentation/

[3] Node.js Foundation, *Node.js Documentation*, https://nodejs.org/en/docs/

[4] JWT.io, *Introduction to JSON Web Tokens*, https://jwt.io/introduction

[5] MongooseJs, *Mongoose Documentation*, https://mongoosejs.com/docs/

[6] TheCatAPI, *Image API Documentation*, https://developers.thecatapi.com/view-account/ylX4blBYT9FaoVd6OhvR?report=bOoHBz-8t

[7] Randommer.io, *Random Data API Documentation*, https://randommer.io/api/docs/index.html

[8] M. Jean-François Berdjugin, *Cours d'Architecture Logicielle (R4.01)*, BUT Informatique, Université de Nantes, 2025.

[9] M. Christian Attiogbé, *Cours d'Automates et Langages (R4.12)*, BUT Informatique, Université de Nantes, 2025.

Glossaire

Micro-service
: Composant logiciel indépendant effectuant une tâche précise. Il communique avec d'autres micro-services via des appels réseau.

Automate
: Modèle de comportement représenté par des états et des transitions. Utilisé ici pour modéliser les services.

État
: Situation ou configuration dans laquelle se trouve un système à un moment donné dans un automate.

Transition
: Passage d'un état à un autre selon une action ou un événement donné.

JWT (JSON Web Token)
: Format de jeton sécurisé permettant d'authentifier un utilisateur via un token signé.

Proxy
: Micro-service intermédiaire qui reçoit toutes les requêtes des clients et les redirige vers les services appropriés.

Booster
: Pack de cartes virtuelles que l'utilisateur peut ouvrir. Cela déclenche une interaction entre les services Utilisateurs et Cartes.

MongoDB
: Base de données NoSQL utilisée pour stocker les informations des utilisateurs et les cartes.

Coin
: Monnaie virtuelle utilisée dans l'application pour acheter des boosters.

API externe
: Service tiers utilisé pour enrichir les données générées (ex. : images de chats, prénoms aléatoires).

Token
: Jeton d'authentification envoyé par le client pour accéder aux ressources protégées.

Middleware Fonction dans une application Express.js qui intercepte et traite les requêtes HTTP avant qu'elles atteignent le contrôleur final.

REST API Style d'architecture web basé sur HTTP permettant aux services de communiquer via des requêtes (`GET`, `POST`, etc.).

Annexes

Cette section rassemble des ressources complémentaires : automates en grand format, extraits de code représentatifs, et liens utiles (comme le dépôt Git).

Lien vers le dépôt Git

Le projet complet est disponible ici : https://gitlab.univ-nantes.fr/pub/but/but2/sae4/sae4_class_grp1_eq4_antunes-samuel_becqart-paul-emile_boeglin-lohan_chauvel-sachphan-theo

Diagrammes des automates en grand format

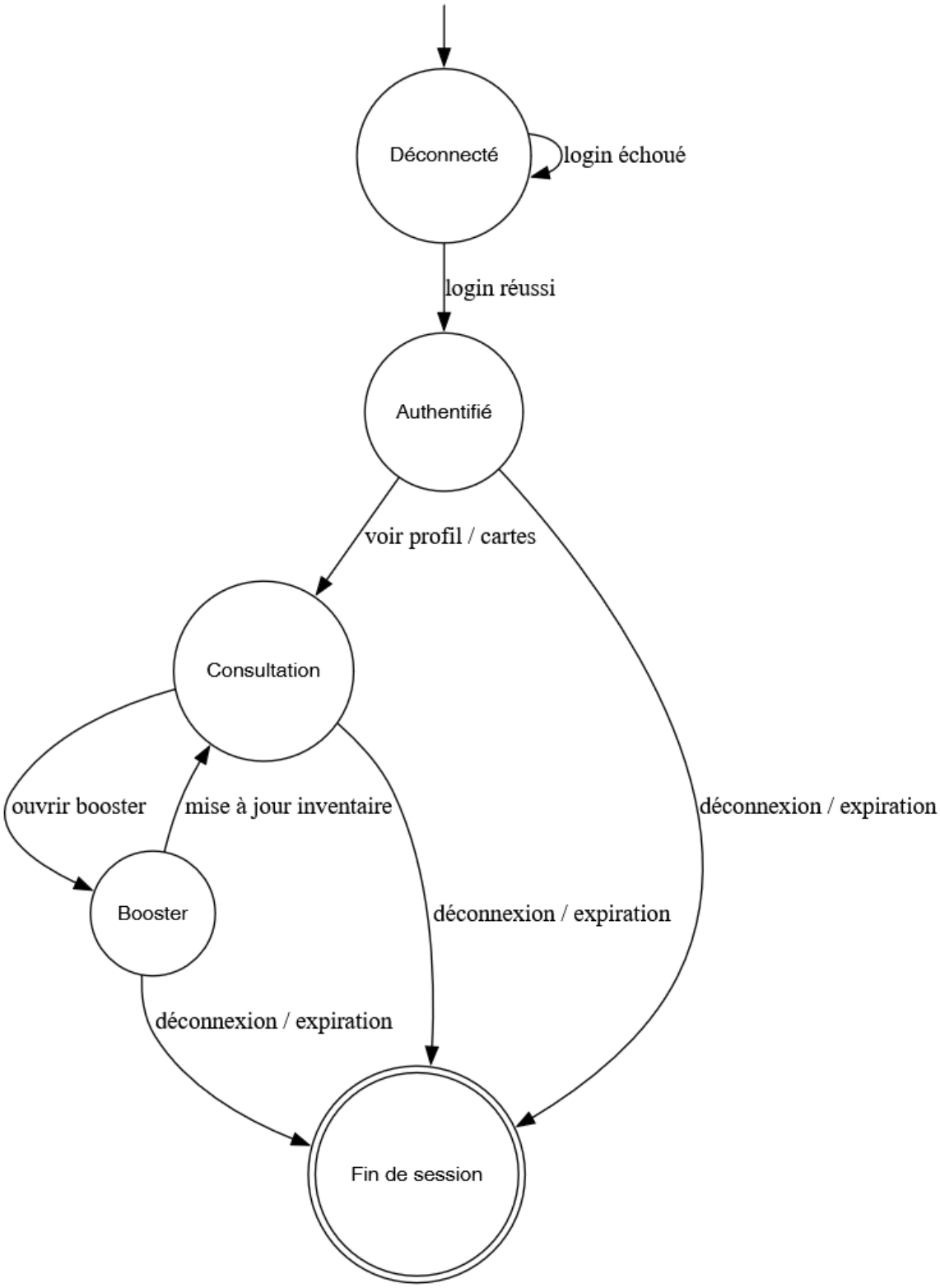

FIGURE 6 – Automate – Micro-service Utilisateurs (grand format)

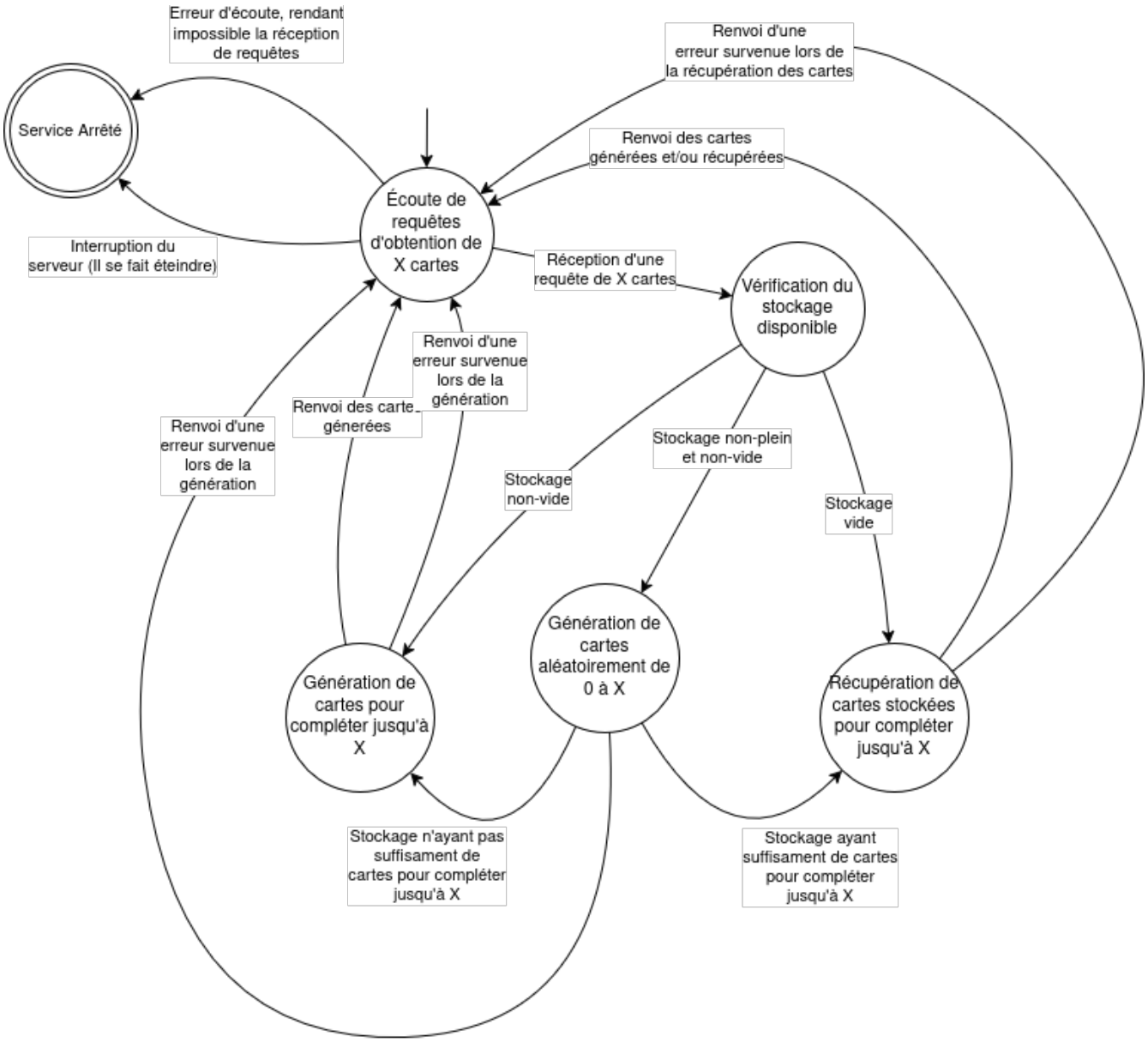

FIGURE 7 – Automate – Micro-service Cartes (grand format)

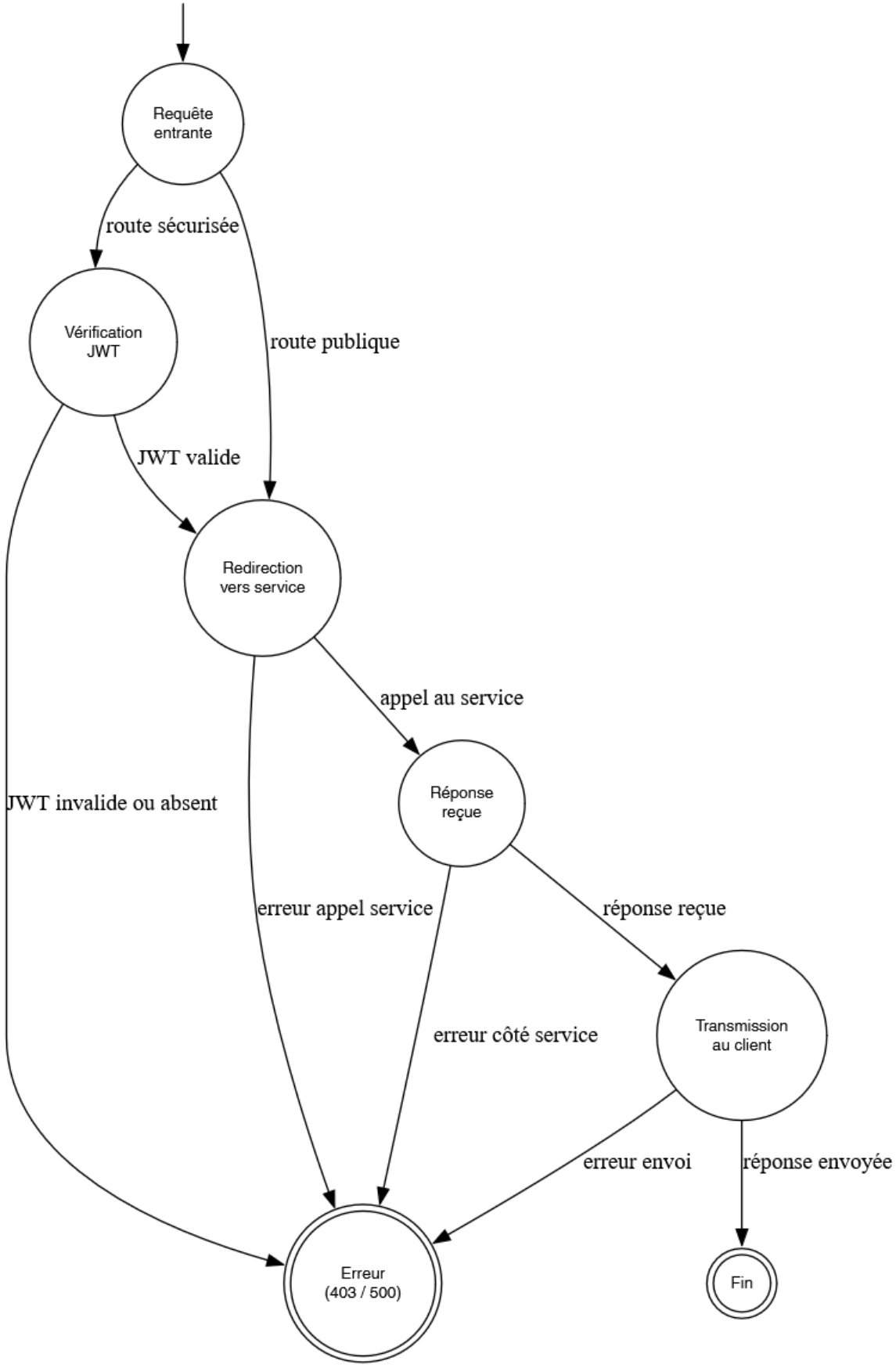

FIGURE 8 – Automate – Micro-service Proxy (grand format)

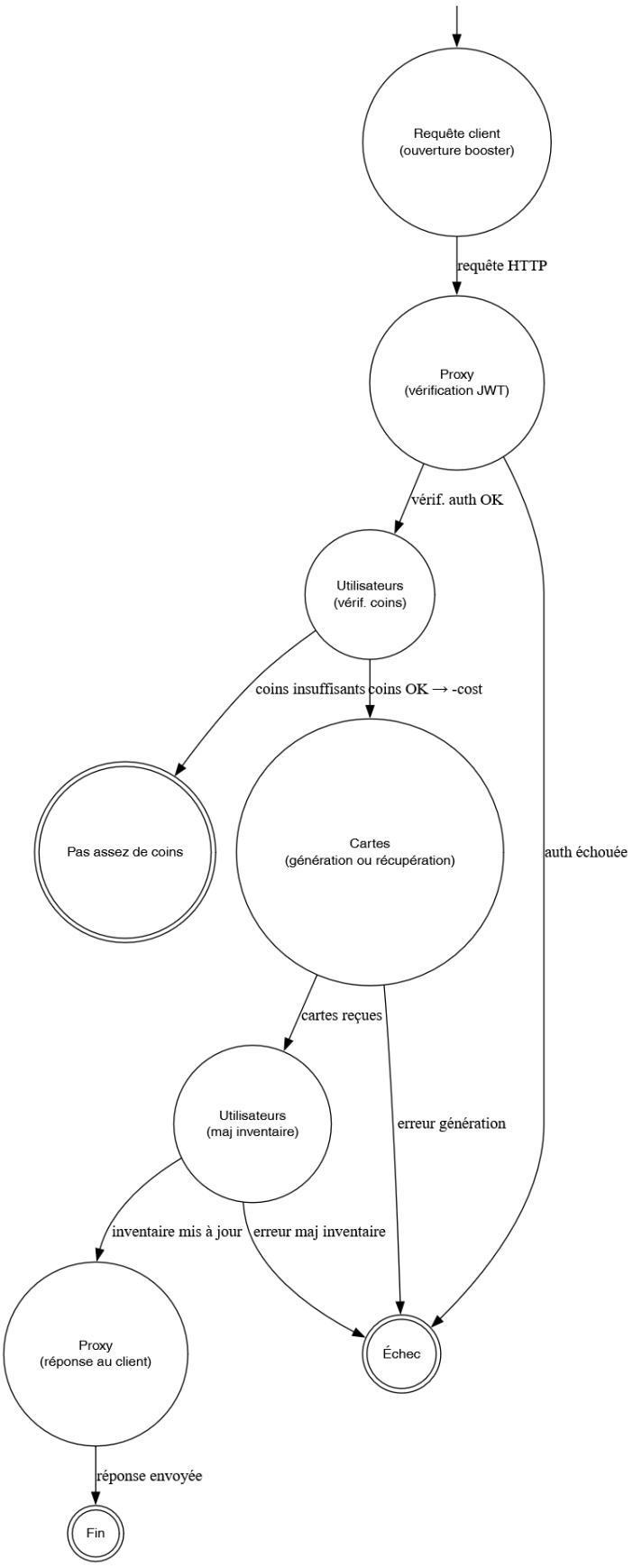

FIGURE 9 – Automate de coordination entre les micro-services (grand format)

Extraits de code représentatifs

```
'use strict'

import express from 'express'
import userRouter from './userRoute.mjs'
import authMiddleware from '../middleware/auth.mjs'

const router = express.Router()

router.use('/user', (req, res, next) => {
    // Skip JWT verification for login, register, and refresh-tokens
    if (req.path === '/login' || req.path === '/register' || req.path
       === '/refresh-tokens') {
        return next()
    }
    // Apply JWT verification for all other routes
    authMiddleware(req, res, next)
})

// Use the user and card routers
router.use('/user', userRouter)

router
    .route('/')
    .options((req, res) => {
        res.status(204)
    })

export default router
```

Listing 1 – Accès route sécurisé ou non

```
import jwt from 'jsonwebtoken'

const JWT_SECRET = process.env.JWT_SECRET

export default function authMiddleware(req, res, next) {
    const authHeader = req.headers['authorization']
    if (!authHeader) return res.status(401).json({ message: 'No token
       provided' })

    const token = authHeader.split(' ')[1]
    try {
        const decoded = jwt.verify(token, JWT_SECRET)
        req.user = decoded
        next()
    } catch (err) {
        res.status(403).json({ message: 'Invalid token' })
    }
}
```

Listing 2 – Middleware d'authentification JWT

Algorithme pour la création de carte :

Algorithm 1: Get Random Cards

Input: *amount* : Number of cards to retrieve
Output: A list of generated and/or fetched cards

```
1  amountToGenerate ← min(maxStored − existingCards, max(amount −
     existingCards, random(0, amount))) ;
2  amountToFetch ← amount − amountToGenerate ;
3  cards ← [ ] ;
4  if amountToFetch > 0 then
5      fetchedCards ← fetch random cards from database of size amountToFetch ;
6      append fetchedCards to cards ;
7  nameSource ← first name source from config ;
8  names ← fetch amountToGenerate names using nameSource ;
9  for i ← 0 to amountToGenerate − 1 do
10     imageSource ← random image source from config ;
11     imageData ← fetch one image using imageSource ;
12     image ← imageSource.cdn_root + imageData[path] ;
13     newCard ← new Card with :
14         id = existingCards,
15         name = names[i],
16         type = imageSource.source_type,
17         rarity = getRarity(),
18         image = image ;
19     add newCard to database ;
20     append newCard to cards ;
21 return cards ;
```