Project M

Egor Semenov

February 2025

1 Introduction

Dans un établissement d'enseignement moderne, les services numériques jouent un rôle clé dans l'organisation de l'apprentissage et la gestion du temps. La commodité et la fiabilité d'une application pour consulter l'emploi du temps permettent aux étudiants de planifier leur temps efficacement, tandis que le confort d'utilisation minimise le risque de manquer des cours.

L'objectif de ce projet académique est de développer un système de consultation des emplois du temps basé sur une architecture microservices. Ce projet met en pratique des approches avancées de conception de systèmes distribués, incluant l'utilisation d'un APIGateway, de microservices pour le contrôle d'accès, le traitement des requêtes et la mise à jour des données.

Au cours du développement, l'architecture du système a été modifiée à plusieurs reprises afin de s'adapter aux changements des exigences fonctionnelles et d'améliorer les performances. Une attention particulière a été portée à la création d'un système de sécurité avancé, incluant le stockage sécurisé des mots de passe, la gestion des sessions et l'utilisation de tokens JWT.

Ce projet a été réalisé en dehors du cadre du programme académique de l'université, en tant que travail autonome visant à approfondir les connaissances sur l'architecture microservices, le travail avec les bases de données, ainsi que les mécanismes d'authentification et d'autorisation.

2 Premiers pas dans la création de l'architecture

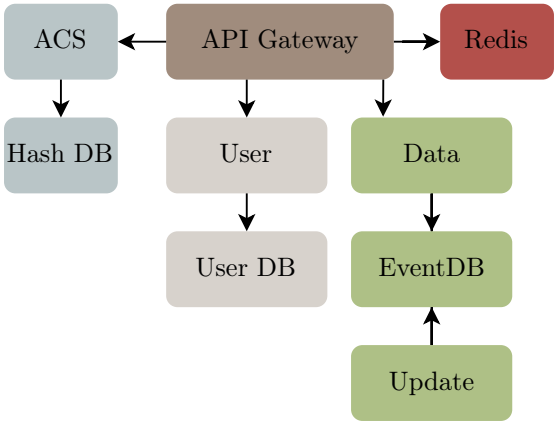


Figure 1: Architecture actuelle des microservices

2.1 Première version

La première version de l'architecture ne comprenait que deux composants: un service de traitement des requêtes et une base de données des emplois du temps. La mise à jour des données devait s'effectuer via une simple fonction dans le service de traitement des requêtes. Il était prévu d'utiliser Python pour développer le service et SQLite pour le stockage des données.

Cette version n'a pas été mise en œuvre en raison de l'évolution rapide des exigences du projet. Après avoir réussi à configurer le transfert de ports et à recevoir des requêtes depuis le réseau externe, il est devenu évident que l'application devait prendre en charge l'accès de plusieurs utilisateurs. Cela nécessitait l'intégration d'un mécanisme d'authentification, ce que l'architecture initiale ne permettait pas.

2.2 Service de contrôle d'accès

Pour prendre en charge l'authentification, un service de contrôle d'accès a été ajouté. Ce service:

- Reçoit le nom d'utilisateur et le mot de passe de l'utilisateur
- Compare les informations avec celles stockées dans la base de données
- Transmet la requête au service de traitement si l'authentification est réussie

Cependant, l'ajout de ce service a entraîné de nouveaux défis:

- Sécurité du stockage des mots de passe dans la base de données

- Routage des requêtes: tous les endpoints des services ne doivent pas être accessibles publiquement
- Inconvénient de la saisie répétée du mot de passe à chaque requête, ce qui affecte négativement l'expérience utilisateur
- Difficulté à journaliser les requêtes en raison de l'absence d'un point d'entrée unique dans le système
- Complexité dans la gestion de la vérification des tokens

2.3 API Gateway

Pour résoudre ces problèmes, le pattern API Gateway a été utilisé. Celui-ci:

- Proxie toutes les requêtes externes vers les microservices internes
- Cache l'architecture interne en ne rendant visibles que les endpoints publics
- Associe à chaque requête un identifiant unique pour uniformiser la journalisation
- Assure un contrôle d'accès centralisé

L'API Gateway a été implémentée en utilisant Python et FastAPI, car ce framework offre des performances élevées et facilite la configuration des routes. PostgreSQL a été choisi comme SGBD pour stocker les données des emplois du temps en raison de sa fiabilité et de sa flexibilité dans la gestion des données.

Pour résoudre le problème de la répétition constante de l'authentification, un système de tokens JWT a été mis en place, car ils:

- Permettent de stocker des informations supplémentaires dans le corps du token (par exemple, le rôle de l'utilisateur)
- Disposent d'un mécanisme intégré d'expiration
- Sont faciles à traiter côté client et côté serveur

Pour le stockage sécurisé des mots de passe, la bibliothèque passlib a été utilisée afin de hacher les mots de passe avec un "sel", renforçant ainsi la sécurité des données.

Il a été décidé de vérifier les tokens séparément dans chaque service, car cela réduit le couplage entre les services. Cette approche permet également de poursuivre le traitement des requêtes même lorsque le service de contrôle d'accès est indisponible.

3 Évolution de la base de données

3.1 Source des données

Le site web actuel des emplois du temps de l'université permet de télécharger les emplois du temps des groupes au format .ics. Plus précisément, le site fournit des liens vers des endpoints qui renvoient la version actuelle de l'emploi du temps pour les groupes sélectionnés.

Exemple de données issues d'un fichier .ics:

BEGIN:VEVENT
DTSTAMP:20240901T071837Z
DTSTART:20240909T081500Z
DTEND:20240909T101500Z
SUMMARY:Elec3A LOCATION:Niepce
DESCRIPTION: Parcours IE3 GOUTON
PIERRE (Exported:01/09/2024 09:18)
UID:ADE60323032342d32303235556e69766572736974e964654
26f7572676f676e652d33393431392d31322d30
CREATED:19700101T000000Z
LAST-MODIFIED:20240901T071837Z
SEQUENCE:2141403058 END:VEVENT

L'analyse de ce document a révélé les détails suivants:

- L'événement commence par BEGIN:VEVENT et se termine par END:VEVENT
- DTSTAMP n'apporte aucune information utile car il est mis à jour à chaque téléchargement
- DTSTART et DTEND indiquent respectivement l'heure de début et de fin de l'événement
- SUMMARY contient toujours le nom de l'événement
- LOCATION est généralement présent et précise le lieu
- DESCRIPTION contient le nom du parcours, du groupe, ainsi que le nom du professeur et la date du téléchargement
- UID ne contient aucune donnée significative, il semble être un identifiant d'événement. D'après l'analyse des motifs, cette chaîne semble contenir des informations sur le type de cours, le numéro de l'occurrence actuelle ainsi que le groupe, mais ces données ne sont pas fiables.
- CREATED ne contient aucune information et est toujours fixé à 01.01.1970
- LAST-MODIFIED contient toujours la date de téléchargement

- SEQUENCE, selon la documentation ICS, devrait contenir le numéro d'élément dans une série d'événements. Toutefois, cela impliquerait l'existence d'un événement principal qui n'est pas présent

Les données issues des fichiers .ics ont été choisies comme source principale, car elles contiennent des informations actualisées et complètes sur les emplois du temps, fournies par le site officiel de l'université. Cela a permis d'automatiser le processus de récupération et de mise à jour des données, garantissant ainsi l'actualité des emplois du temps pour les utilisateurs du système.

3.2 Première étape. Approche naïve

La première étape a consisté à normaliser les données. Cette approche a été choisie pour minimiser la redondance des données et offrir une flexibilité dans l'exécution des requêtes. Chaque événement pouvait inclure plusieurs professeurs et groupes, ce qui nécessitait l'utilisation de relations « plusieurs à plusieurs ». J'ai donc séparé ces relations dans des tables distinctes. Les relations « plusieurs à plusieurs » ont été simplifiées en « un à plusieurs » en ajoutant des tables intermédiaires. Le résultat est le schéma illustré à la figure 2.

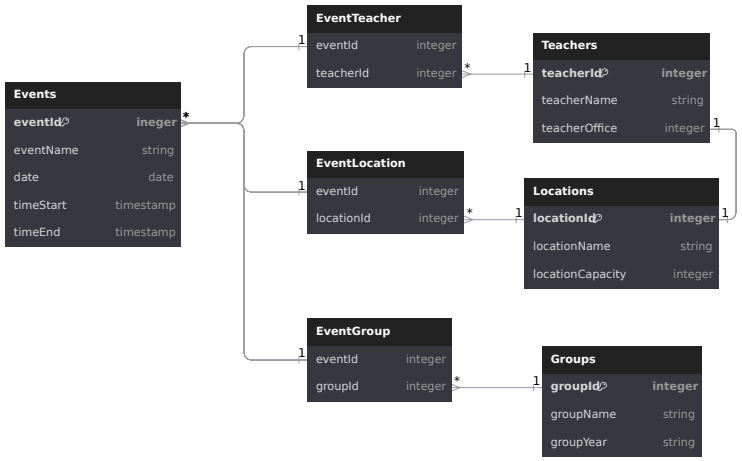


Figure 2: Première structure de la base de données

Le principal problème de cette première version résidait dans la complexité des requêtes nécessaires pour obtenir les emplois du temps. En particulier, il fallait prendre en compte non seulement les groupes spécifiques, mais aussi les flux auxquels ils appartiennent, ainsi que les sous-groupes. La structure des données ne prenait pas en charge une recherche hiérarchique de ce type, ce qui compliquait considérablement les requêtes.

3.3 Deuxième étape. Recherche de solution

La première solution envisagée a été de créer des tables distinctes pour les Groupes, les Flux et les Années. Chaque élément contiendrait alors une référence à son parent, et la recherche consisterait d'abord à récupérer l'indice de l'élément, puis à rechercher le parent via l'indice parent.

Cette solution m'a été suggérée par un motif que j'avais remarqué: dans la section DESCRIPTION des sous-groupes, il n'y avait pas de préfixe mais un chiffre était présent, le nom du flux commençait toujours par « Parcours » et contenait un chiffre, le nom de l'année d'étude contenait un espace et un chiffre, tandis que le nom du professeur ne contenait pas de chiffre. En construisant un simple automate, les entités étaient facilement triées.

Le résultat a conduit à une base de données dont une partie est illustrée à la figure 3.

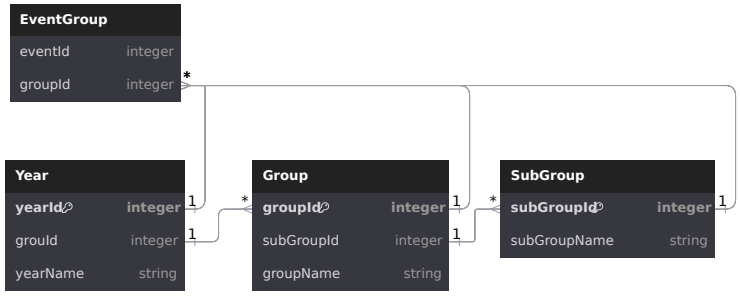


Figure 3: Deuxième structure de la base de données

Cependant, cette approche présentait également plusieurs problèmes:

- En toute rigueur, il n'était pas possible de concevoir une telle base car le type n'était pas spécifié dans la table « EventGroup ». L'ajout d'un type n'étant pas pris en charge automatiquement, la cohérence des données aurait dû être maintenue par un programme externe.
- La complexité des opérations de lecture et d'écriture.
- L'absence de support pour des arbres de groupes plus profonds.

Pour résoudre ces problèmes, j'ai évolué vers la troisième version de la base de données.

3.4 Troisième étape. Table récursive

La solution est apparue après avoir identifié le dernier problème de la liste: la profondeur variable de l'arbre des groupes. La structure des groupes forme un graphe. Il restait à comprendre comment représenter ce graphe sous forme de table. Pour cela, il suffisait d'ajouter un attribut contenant l'indice du parent. Ainsi, le niveau supérieur — l'année — n'aurait pas d'indice parent, tandis que

tous les autres en auraient un. On pourrait alors appliquer un parcours de graphe pour obtenir la liste des groupes recherchés. La structure finale de la base de données est présentée à la figure 4.

Un autre défi est apparu: la mise à jour de la table. Comment distinguer les événements déjà ajoutés, ceux à ajouter, et ceux à supprimer ? Pour cela, j'ai introduit deux nouveaux champs: « hash » et « lastFound ».

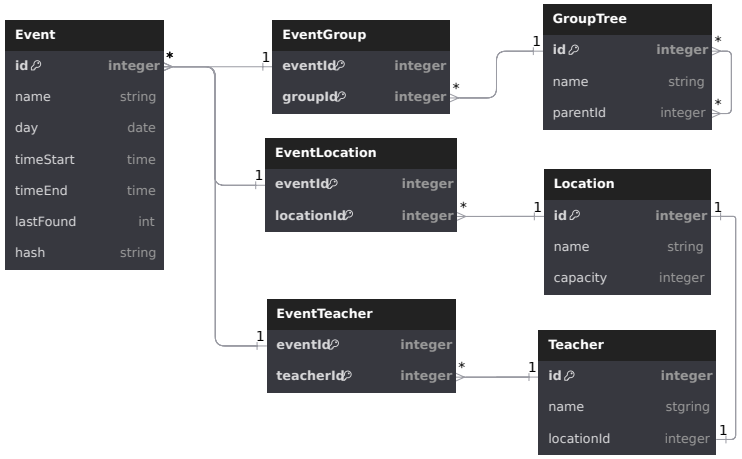


Figure 4: Structure finale de la base de données

3.5 Mise à jour des données

La mise à jour de la base de données est une tâche complexe, notamment en raison des défis liés à la suppression des données obsolètes, à l'ajout de nouvelles données et à la mise à jour de celles déjà existantes.

Dans la première version du service de mise à jour, le programme enregistrait la version précédente des fichiers .ics et comparait les deux fichiers. Toutefois, cette méthode s'est avérée inefficace en raison de l'ordre aléatoire des événements à chaque téléchargement, ainsi que de la mise à jour constante des champs DTSTAMP et LAST-MODIFIED. Cette approche a donc été abandonnée au profit d'une solution plus optimisée.

Dans la deuxième version, un champ de hachage a été ajouté à la base de données. Cet attribut indexé contenait toutes les données de l'événement en un seul endroit. Désormais, lors du téléchargement d'une nouvelle version de l'emploi du temps, le fichier n'était plus sauvegardé. À la place, un hachage de l'événement était généré, puis une recherche dans la base de données était effectuée. En l'absence de résultat, l'événement était ajouté. Cependant, un problème subsistait: la suppression des événements devenus obsolètes. Pour y remédier, il fallait conserver la liste des événements mis à jour et supprimer ceux qui n'y figuraient pas, ce qui nécessitait un second parcours complet de la table.

Dans la troisième version, un index de « dernière détection » a été ajouté à la base de données. Cet attribut est mis à jour à chaque fois qu'un événement est trouvé, par exemple par son hachage. Une fois la mise à jour terminée, il devient alors possible de supprimer tous les événements dont l'index est obsolète.

3.6 Bibliothèque pour l'interaction avec la base de données

Dans la première version du programme, SQLite a été choisi en raison de sa simplicité et de sa facilité de configuration. La communication avec la base de données s'effectuait via le support natif des requêtes SQL en Python.

Après le passage à PostgreSQL, il a fallu choisir une nouvelle bibliothèque. Le choix de SQLModel s'est imposé en raison de son intégration avec Pydantic, ce qui permet de combiner la validation des données et l'interaction avec la base de données en un seul niveau. Cette approche a simplifié le développement et amélioré la lisibilité du code, le rendant ainsi plus facile à maintenir.

En raison de sa relative nouveauté, SQLModel ne dispose pas d'une documentation complète. J'ai donc dû lire le code source de la bibliothèque et utiliser des méthodes de bas niveau issues de SQLAlchemy. Les bibliothèques utilisées ne prenant pas en charge nativement les tables récursives, j'ai dû écrire moi-même les fonctions correspondantes.

4 Poursuite du développement de l'architecture

4.1 Service des utilisateurs

Après avoir défini le scénario d'utilisation de l'application, le besoin de développer un service des utilisateurs s'est fait sentir afin d'améliorer l'expérience utilisateur. L'objectif principal était de permettre la sauvegarde des données personnelles, y compris les groupes favoris et les préférences de l'utilisateur. Cela permettrait à l'utilisateur de voir immédiatement l'emploi du temps de son groupe lors de sa connexion et de basculer rapidement entre ses groupes favoris.

Le champ « favoris » avait une structure non structurée et potentiellement récursive, ce qui le rendait difficile à stocker dans un modèle de données relationnel. En utilisant une base de données SQL, il aurait été nécessaire de créer plusieurs tables récursives, ce qui aurait considérablement complexifié les requêtes et affecté les performances du système.

Pour cette raison, MongoDB a été choisie pour stocker les données des utilisateurs. En tant que base de données orientée documents, elle est idéale pour gérer des structures imbriquées et flexibles tout en permettant une modification facile du schéma au fur et à mesure que les exigences évoluent. Cela a permis d'adapter la structure de stockage des données aux scénarios d'utilisation en constante évolution, sans avoir besoin de migrations complexes.

Dans MongoDB, un document distinct est créé pour chaque utilisateur, contenant les champs suivants:

- **UID** — identifiant unique de l'utilisateur, lié à la base de données des mots de passe.
- **CreationDate** — date d'inscription de l'utilisateur.
- **Name** — nom de l'utilisateur affiché sur la page de profil.
- **Theme** — thème du site choisi par l'utilisateur.
- **Image** — URL de l'image de profil (avatar).
- **Group** — groupe par défaut dont l'emploi du temps est affiché à la connexion.
- **History** — les 20 dernières requêtes de l'utilisateur.
- **MostSearched** — les deux requêtes les plus fréquemment utilisées.
- **Favorite** — liste des groupes, enseignants et lieux favoris, avec la possibilité de stocker des listes imbriquées pour afficher des structures complexes de favoris.

Cette structure permet de sauvegarder de manière flexible les données nécessaires pour garantir une utilisation pratique du service. Elle simplifie également l'ajout de nouveaux champs et fonctionnalités, comme des paramètres utilisateur avancés ou des catégories supplémentaires dans les favoris, sans nécessiter de modifications du schéma de la base de données.

La création des utilisateurs est étroitement liée au service de contrôle d'accès: seul ce dernier peut créer un nouvel utilisateur ou supprimer un utilisateur existant afin d'éviter des incohérences dans les données. Lorsqu'un utilisateur est supprimé, son statut passe à « inactif », mais son enregistrement reste dans la base de données pour faciliter le suivi à l'avenir.

4.2 Service de contrôle d'accès 2.0

À un stade précoce du développement, j'ai remarqué qu'après l'expiration du token d'accès (2 heures), l'utilisateur devait ressaisir son mot de passe, ce qui dégradait l'expérience utilisateur. Pour résoudre ce problème, il a été décidé de mettre en place un mécanisme de renouvellement des tokens en utilisant un Refresh Token, dont la durée de validité est fixée à 2 semaines.

Pour garantir la sécurité et la convivialité, il était nécessaire de stocker tous les tokens de rafraîchissement actifs dans la base de données. Cela permettrait de vérifier à tout moment la validité du token. Cependant, plusieurs questions se sont posées:

- Que se passe-t-il si le Refresh Token est compromis ?
- Comment l'utilisateur peut-il révoquer un token en cas de fuite ?
- Comment gérer le multi-appareil, c'est-à-dire la possibilité d'utiliser l'application sur plusieurs appareils simultanément ?

Pour répondre à ces défis, un système de sessions a été conçu (voir Fig. 5) dans lequel:

- Chaque session stocke le token d'accès actuel et le Refresh Token actuel.
- Lors de l'expiration du token d'accès, l'utilisateur envoie une demande de renouvellement et reçoit une nouvelle paire de tokens (d'accès et de rafraîchissement).
- Les tokens mis à jour sont enregistrés dans la base de données, garantissant ainsi que seuls les tokens valides sont stockés à tout moment.
- Un token individuel pour la réception de notifications est également sauvegardé dans la base de données.

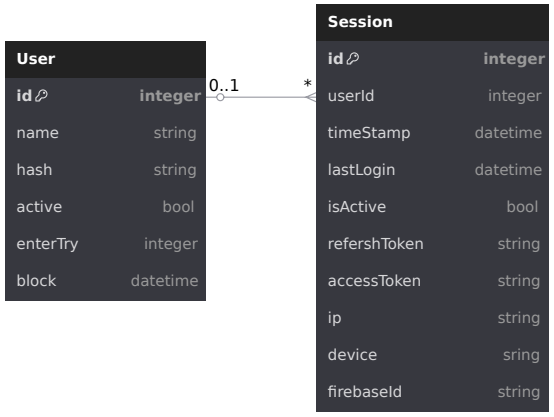


Figure 5: Structure de la base de données du service de contrôle d'accès

Pour assurer la sécurité et la flexibilité d'utilisation, la logique suivante a été mise en place:

- Un utilisateur est autorisé à avoir jusqu'à 5 sessions actives. Cela permet d'utiliser l'application sur plusieurs appareils en même temps.
- L'utilisateur peut se déconnecter à tout moment de n'importe quelle session en révoquant le token d'accès et le Refresh Token, renforçant ainsi la sécurité.

Révocation anticipée des tokens Un problème de révocation anticipée des tokens s'est posé. Si l'utilisateur termine manuellement une session, le token doit être immédiatement invalidé, même s'il n'a pas encore expiré.

Pour cela, un système de vérification des tokens révoqués a été développé:

- Tous les tokens révoqués sont stockés dans Redis — une base de données en mémoire à haute performance.

- L'API Gateway vérifie, pour chaque requête à une ressource protégée, si le token figure dans la liste des tokens révoqués. Si le token est présent, l'accès est refusé.

Choix de la technologie Redis a été choisi pour plusieurs raisons:

- **Haute performance**: La vérification des tokens est extrêmement rapide grâce au stockage des données en mémoire.
- **Expiration automatique des entrées**: Redis permet de définir une durée de vie pour chaque enregistrement, après laquelle il est automatiquement supprimé. Cela est particulièrement utile, car les tokens révoqués sont supprimés après l'expiration de leur durée de validité, ce qui empêche une accumulation excessive de données dans Redis.

Ainsi, l'utilisation de Redis permet de gérer efficacement les sessions et de garantir la sécurité des tokens tout en offrant une expérience utilisateur fluide et agréable.

4.3 Alternatives

Lors de l'analyse des interactions entre les services, deux chaînes d'actions clés ont été identifiées:

- Création d'un utilisateur dans le service de contrôle d'accès → création de l'utilisateur dans le service de données personnelles → ajout potentiel de l'utilisateur aux listes de notifications.
- Création d'un événement dans la base de données des emplois du temps → génération de la liste des destinataires des notifications.

Ces chaînes d'actions montrent un couplage étroit entre les services, car chaque service dépend directement de l'achèvement de l'opération dans le service précédent. Cela réduit la flexibilité du système et complique les modifications de la logique métier.

Pour résoudre ce problème, une architecture à base de bus d'événements a été envisagée, comme illustré à la Fig. 6.

Avantages de l'architecture événementielle

- **Découplage des microservices**: Les services interagissent via des événements plutôt que directement, ce qui améliore la flexibilité et l'évolutivité.
- **Résilience accrue**: Si un service est temporairement indisponible, l'événement peut être traité ultérieurement, réduisant ainsi les risques de défaillance.
- **Meilleure évolutivité**: Les services peuvent traiter les événements en parallèle, permettant une répartition plus uniforme de la charge de travail.

Inconvénients et raisons du renoncement à la migration Malgré les avantages énumérés, le passage à une architecture événementielle aurait nécessité des modifications substantielles dans tous les microservices, entraînant plusieurs défis:

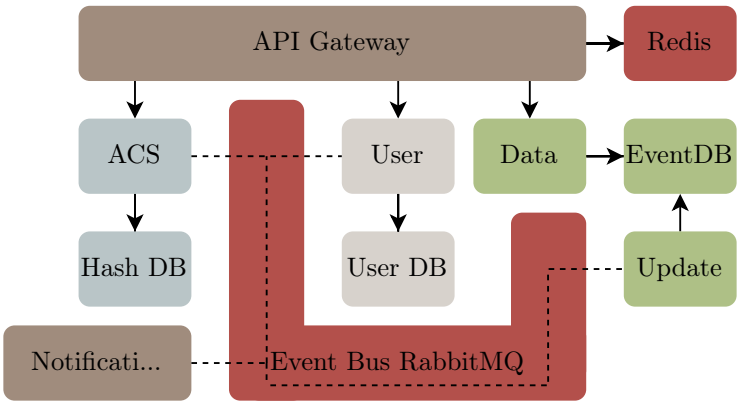


Figure 6: Architecture potentielle avec un bus d'événements

- **Temps de refactorisation élevé**: La réécriture de toutes les interactions entre les services aurait exigé un investissement considérable en temps.
- **Complexité accrue du débogage et des tests**: Le traitement asynchrone des événements complique l'analyse des erreurs et la validation des flux de travail.
- **Besoin de choix et de configuration du bus d'événements**: La sélection et l'implémentation d'une solution comme Apache Kafka ou RabbitMQ augmentent la complexité de l'infrastructure.

Après avoir pesé les gains en flexibilité et les coûts de migration, il a été décidé de conserver l'architecture actuelle tout en gardant ouverte la possibilité de passer à un bus d'événements à l'avenir si cela devenait pertinent en termes de charge ou de besoins métier.

5 Perspectives de développement de l'architecture

5.1 Service de notifications

À l'avenir, il est prévu d'intégrer Firebase pour envoyer des notifications aux utilisateurs concernant les changements d'emploi du temps et d'autres événements importants. Pour la mise en œuvre des notifications, un service dédié sera créé, qui:

- S'abonnera aux événements de modification des emplois du temps via le bus d'événements (en cas de son implémentation).
- Prendra en charge les notifications push.
- Fonctionnera de manière asynchrone afin d'améliorer les performances et de réduire la charge sur les services principaux.

Le service de notifications sera déployé en tant que microservice indépendant, lui permettant de fonctionner de manière autonome par rapport aux autres parties de l'application. On anticipe des défis liés à la création des listes de destinataires des notifications, et le travail sur cette problématique a déjà commencé.

5.2 Serveur SMTP

Dans le cadre de l'évolution du service de contrôle d'accès, l'authentification à deux facteurs par courrier électronique sera mise en place. Cette mesure renforcera la sécurité des utilisateurs.

5.3 Réécriture de certaines parties du code en Rust

Afin d'améliorer les performances et d'optimiser l'utilisation des ressources, il est prévu de réécrire en Rust les parties du code les plus critiques en termes de temps d'exécution.

Dans un premier temps, les modules de contrôle d'accès seront réécrits, car ils représentent un goulot d'étranglement en matière de performance du système.

5.4 Optimisation des interactions entre les services

Pour réduire les latences dans les échanges de données entre microservices, les actions suivantes sont prévues:

- Réécrire l'API Gateway en utilisant nginx.
- Mettre en place un mécanisme de mise en cache au niveau de l'API Gateway pour les données fréquemment demandées, telles que les emplois du temps des groupes populaires.

6 Conclusion

Au cours de ce projet, une architecture flexible et évolutive basée sur des microservices a été conçue et mise en œuvre pour un système de gestion des emplois du temps. Les principales réalisations incluent:

- La création d'une architecture de microservices permettant le développement et le déploiement indépendants des composants.
- L'optimisation des bases de données.
- La mise en place d'un mécanisme d'authentification avancé basé sur des sessions et des tokens de rafraîchissement pour garantir la sécurité.

Une analyse des architectures alternatives, comme le bus d'événements, a également été effectuée, et les perspectives d'évolution suivantes ont été définies:

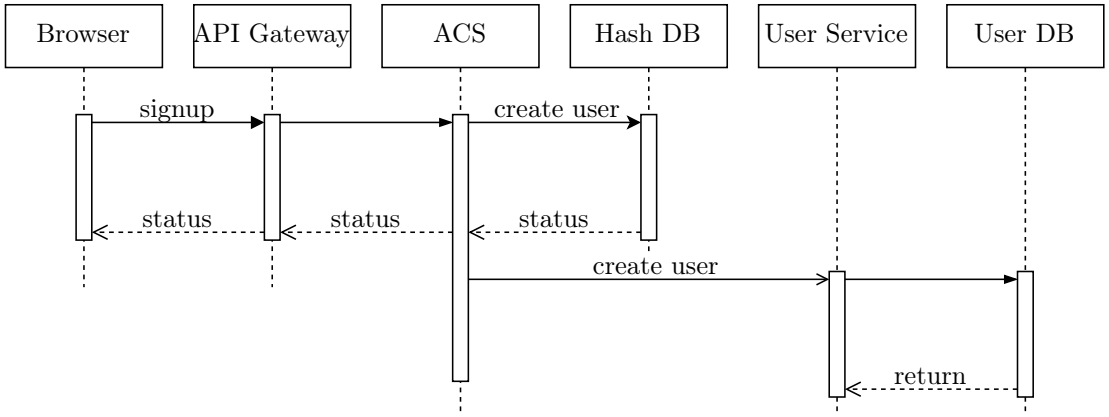
- Intégration d'un serveur SMTP et d'un système de notifications.

- Réécriture des parties critiques en Rust pour améliorer les performances.

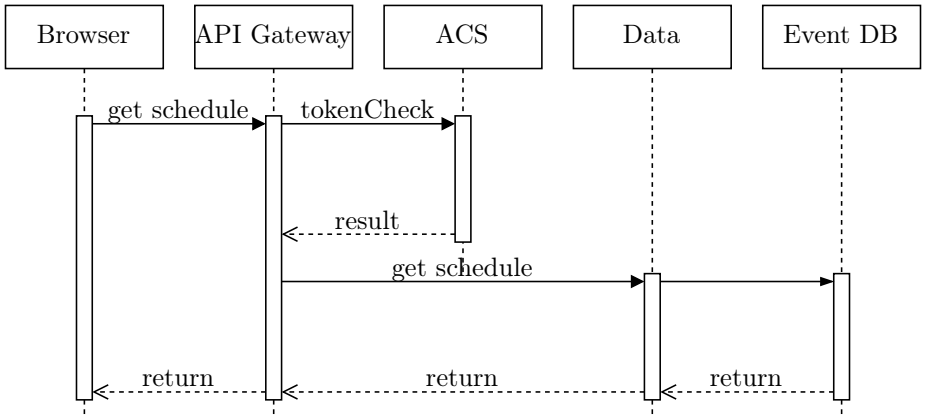
En conclusion, l'architecture du système est prête pour un futur passage à l'échelle et pour évoluer en fonction des exigences métiers et des charges à venir.

A Diagrammes de séquence

/api/v1/user/auth/signup



/api/v1/schedule/week?date=2025-02-20&group=MI4-FC



B Liste de références

References

[1] Parminder Singh Kocher *Microservices and Containers*, 2019.

[2] Ian Miell, Aidal Hobson Sayers *Docker in Practice*, 2020

[3] T. Connolly, C. Begg *Database Systems A Practical Approach to Designe, Implementation and Management*

[4] Jean-Luc Hainaut *Bases de donnees Concepts, utilisation et developpement*

[5] Mark Massé *REST API Design Rulebook*

[6] Brenda Jin, Saurabh Sahni and Amir Shevat *Designing Web APIs*