PS : J'aimerais vous prévenir en avance que j'ai refait le projet car il y avait trop de problème sur l'ancienne version, mais aussi car mon partenaire va quitter la formation car il n'a pas trouvé d'alternance avant la période de fin. De ce fait, je vous transmets ce projet solo.

Historique GitHub :

Le seul b-mol que je trouve à mon projet est que je n'ai pas du tout fait de commit car j'ai dû le refaire car j'ai eu un problème et que j'ai supprimé par maladresse mon gitignore.

De ce fait, il n'y a pas de commit sur l'historique de mon avancement. Mais pour ainsi dire, la partie authentification m'a posé beaucoup de problème. Cela m'a pris beaucoup de temps. J'ai fait appel à vous plusieurs fois pour demander des questions et conseils durant le cours. Je suis quand même ravi du contenu car cela m'a permis d'apprendre une nouvelle architectute mais aussi d'apprendre à coder en TypeScript.

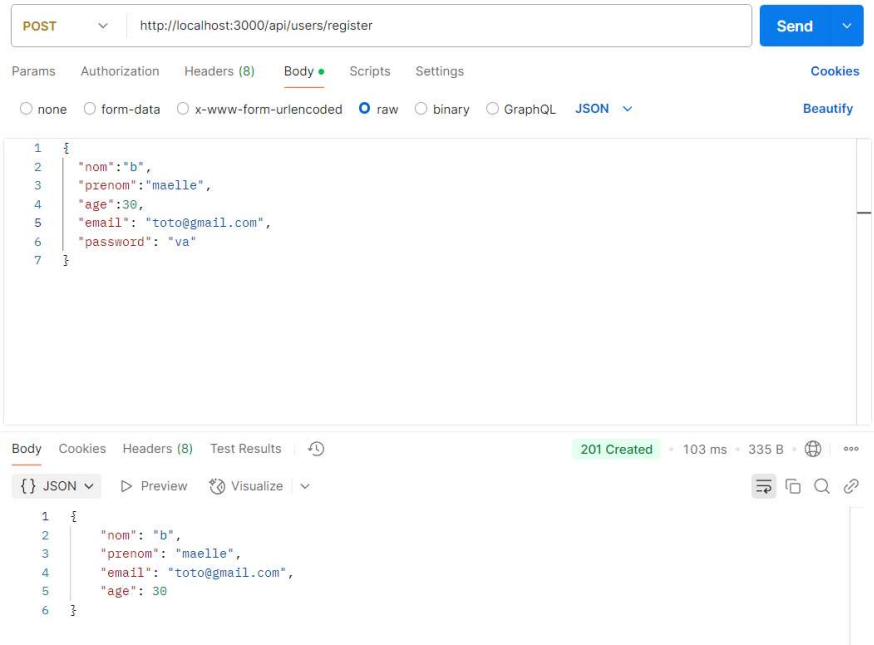
Je n'ai pas pu faire de screen car j'ai rencontré des problèmes à la fin sur mon tsconfig.json. J'ai essayé de résoudre le problème mais je n'avais pas assez de temps ce week-end car j'étais vraiment occupé. Je ne suis donc pas sûr que vous allez pouvoir tester mon code. J'ai donc préféré me concentrer sur un bon rendu de documentation.

Désolé d'avance x'(.

Pour les captures que je voulais faire à la fin du projet, j'ai décidé de reprendre quelques-unes de celle qu'on avait réalisé au préalable avec mon ancien partenaire.

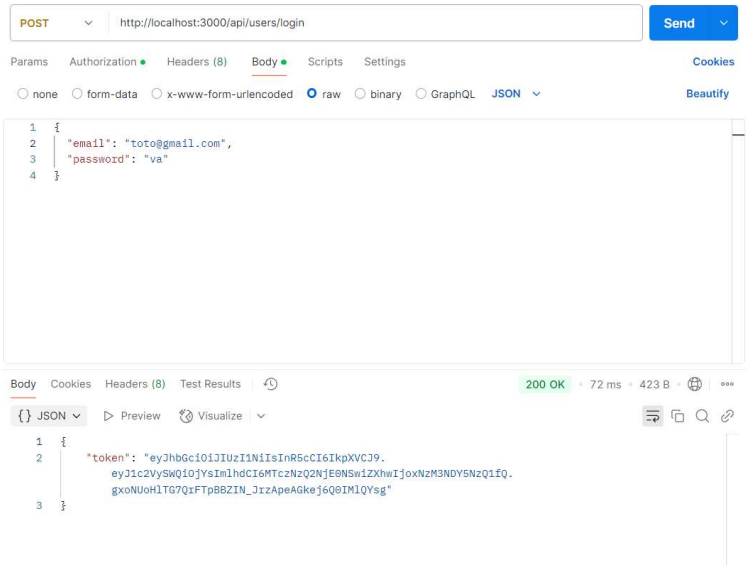
<u>Capture de l'ancien projet :</u>

Postman pour se créer un user

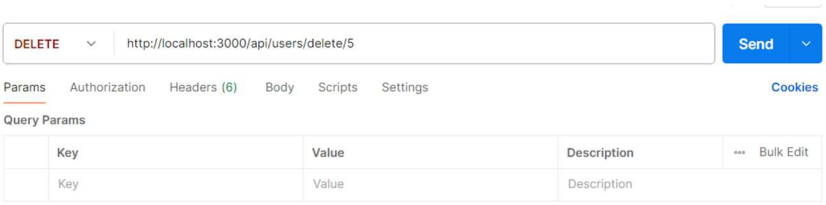


Documentation sur mon projet et mes choix

Postman pour se loger à un user



Postman pour delete un user :



Note Explicative des Choix Techniques

1. Choix entre TypeORM et Mongoose

Le choix entre TypeORM et Mongoose dépend en grande partie du type de base de données que tu utilises. **TypeORM** est une bibliothèque TypeScript/JavaScript pour interagir avec des bases de données relationnelles (telles que PostgreSQL, MySQL, etc.), tandis que **Mongoose**

est une bibliothèque spécifique à MongoDB pour gérer les données dans une base de données NoSQL.

- **TypeORM** : J'ai opté pour TypeORM pour les bases de données relationnelles, car il offre une structure très proche des objets en mémoire et permet de travailler avec des entités et des relations complexes.
- **Mongoose** : Mongoose fournit un mécanisme de validation des données, une gestion des schémas et des requêtes faciles à utiliser, tout en permettant des interactions plus flexibles avec les documents. Cependant, dans mon cas, j'ai préféré TypeORM, car le projet nécessite une base de données relationnelle.

2. Gestion des erreurs

La gestion des erreurs est un aspect crucial dans le développement d'applications robustes et fiables. Dans ce projet, j'ai opté pour plusieurs pratiques qui permettent de centraliser et de normaliser la gestion des erreurs.

- **Middleware global de gestion des erreurs** : Afin d'avoir une gestion cohérente des erreurs, j'ai implémenté un middleware global dans Express. Ce middleware intercepte toutes les erreurs qui ne sont pas traitées et les formate avant de les renvoyer à l'utilisateur sous une forme uniforme (par exemple, en JSON avec un message d'erreur et un code de statut HTTP approprié).
- **Gestion des erreurs asynchrones** : J'ai utilisé try/catch pour capturer les erreurs dans les fonctions asynchrones, particulièrement lors des interactions avec la base de données, et les transmettre au middleware de gestion des erreurs. Cela garantit qu'aucune erreur non capturée ne provoque de plantage inattendu du serveur.
- **Messages d'erreur personnalisés** : J'ai également veillé à personnaliser les messages d'erreur afin qu'ils soient clairs et précis, facilitant ainsi le débogage tout en offrant une meilleure expérience de développement.

3. Architecture n-tiers (Layered Architecture)

L'architecture n-tiers (ou architecture en couches) est un choix qui permet de structurer l'application de manière modulaire, tout en séparant les différentes responsabilités du système. Dans ce projet, j'ai appliqué une architecture en trois couches principales :

- **Couche de présentation (Frontend)** : Le frontend est séparé du backend et consomme des API via des requêtes HTTP. Cette séparation permet une flexibilité maximale, permettant de potentiellement remplacer ou modifier le frontend sans affecter la logique du backend.
- **Couche de logique métier (Backend)** : Le backend est responsable du traitement des requêtes, de la gestion des utilisateurs, et de l'interaction avec la base de données. Il expose des routes qui gèrent les requêtes HTTP (GET, POST, PUT, DELETE), et communique avec la base de données via TypeORM ou Mongoose, en fonction des

besoins. La logique métier est encapsulée dans des services qui permettent de maintenir une séparation claire des responsabilités et de faciliter les tests unitaires.

- **Couche de persistance des données (Base de données)** : Cette couche gère l'accès aux données. En utilisant TypeORM ou Mongoose, la couche de persistance est responsable des opérations CRUD (création, lecture, mise à jour et suppression) sur la base de données. Cela permet une interaction fluide avec les données tout en maintenant la logique de l'application distincte de la gestion des données.

L'architecture en couches permet de découpler les différentes parties du projet, ce qui facilite la maintenance, la scalabilité et les tests unitaires.

4. Sécurisation des routes et gestion de l'authentification

Pour sécuriser les routes, j'ai mis en place un système d'authentification basé sur **JSON Web Tokens (JWT)**. Cela permet aux utilisateurs de se connecter à l'application en utilisant leurs informations d'identification et de recevoir un token JWT qui sera utilisé pour authentifier les requêtes suivantes.

- **JWT pour l'authentification** : Le token JWT contient des informations sur l'utilisateur (par exemple, son ID et son rôle) et est signé pour garantir son intégrité. Lorsqu'un utilisateur se connecte avec succès, un token JWT est généré et envoyé au frontend. À chaque requête suivante nécessitant une authentification, ce token est inclus dans les en-têtes HTTP pour valider l'identité de l'utilisateur.
- **Middleware de validation de JWT** : Un middleware spécifique valide la présence et l'intégrité du token JWT à chaque requête protégée. Ce middleware s'assure que le token est bien signé et que l'utilisateur a les droits nécessaires pour accéder à la ressource demandée.

5. Tests et qualité du code

Pour garantir la qualité et la stabilité du code, j'ai intégré une série de tests automatisés, en particulier les tests unitaires et d'intégration.

- **Tests unitaires** : Les tests unitaires sont utilisés pour valider la logique de chaque fonction individuelle dans l'application. Par exemple, les services métiers qui interagissent avec la base de données sont testés pour garantir qu'ils retournent bien les résultats attendus.
- **Tests d'intégration** : J'ai également mis en place des tests d'intégration pour valider que l'ensemble du système fonctionne correctement. Ces tests vérifient que les différentes couches de l'application interagissent bien entre elles et que les routes HTTP renvoient les bonnes réponses.

6. Conclusion

Les choix techniques effectués pour ce projet visent à assurer une architecture robuste, scalable et maintenable, tout en garantissant une expérience utilisateur fluide et sécurisée. L'utilisation de TypeORM (ou Mongoose pour MongoDB), de JWT pour l'authentification, ainsi

que la mise en place d'une architecture en couches, d'une gestion centralisée des erreurs, et des tests automatisés, contribuent à la qualité et à la fiabilité du projet dans son ensemble.