Angel Velasco

Lab Continuous testing

Objectives:

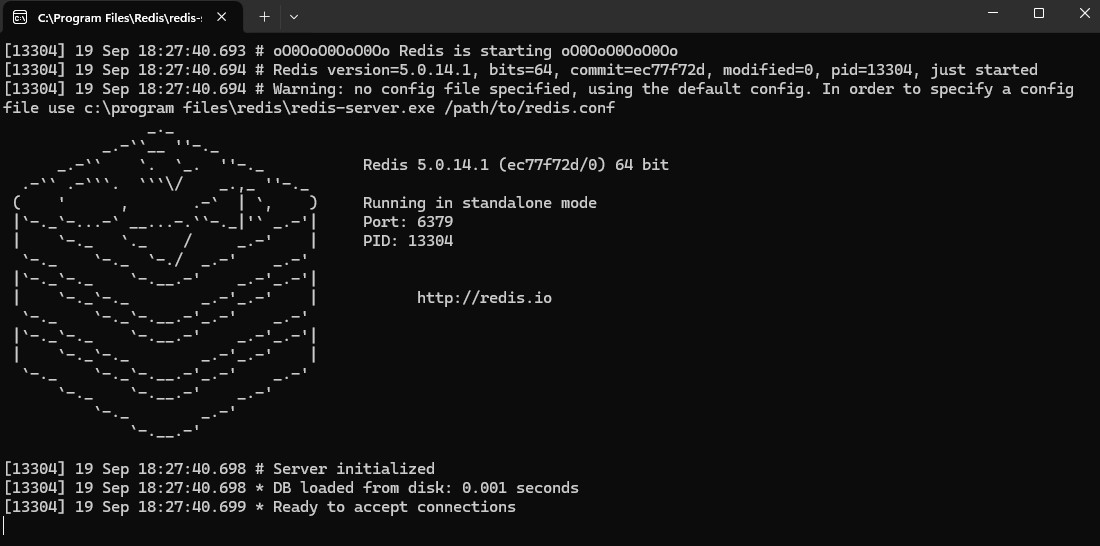
1. **Use** prepared **User API** application and run tests
2. **Using test-driven development** (TDD) create GET user functionality

Repository : https://github.com/angel0x7/angel\_DevOps\_Lab

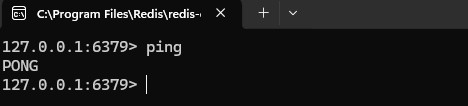
Dans ce Lab nous allons expliquer notre démarche lors de la création des méthodes REST API, accompagnés de code et d’explication de leurs rôles.

Cette application fournit une petite API REST pour créer et consulter des utilisateurs : elle sépare la configuration, le stockage et la logique métier, utilise un fallback en mémoire pour pouvoir tester sans dépendance externe, et inclut des tests automatisés pour vérifier son fonctionnement.

Start Redis Server :

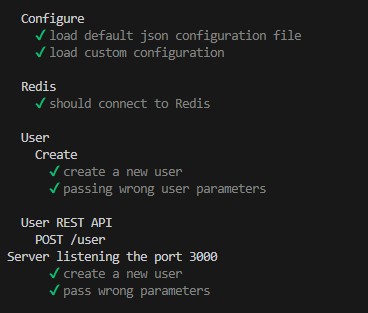


Ping Redis-cli :



Use prepared User API application and run tests

npm test :



Npm start :

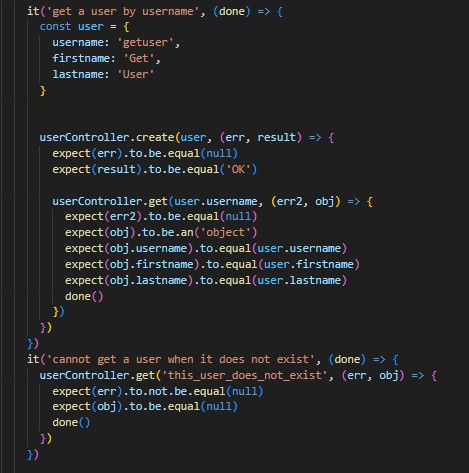


Using test-driven development (TDD) create GET user functionality

# 1. Écrire les tests unitaires (controller) — tests qui échouent d’abord

La première étape consiste à rédiger des tests unitaires ciblés pour la méthode du controller avant d’implémenter la logique. Ces tests doivent appeler directement userController.get(username, cb) sans passer par le serveur HTTP, et vérifier deux comportements:

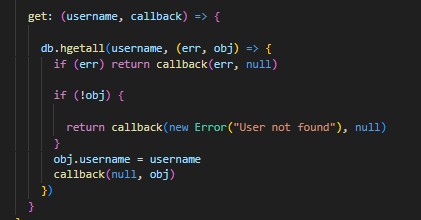
1. quand l’utilisateur existe, la fonction rappelle le callback avec err === null et un objet utilisateur contenant username et les autres champs (firstname/lastname ou email) ;
2. quand l’utilisateur n’existe pas, la fonction rappelle le callback avec une erreur (par ex. new Error('User not found')) et obj === null.



# 2. Implémenter la méthode du controller — implémentation minimale pour passer les tests

Après avoir vu les tests échouer, implémente la méthode userController.get de la façon la plus simple qui fasse passer les tests (vert). La méthode doit appeler le client Redis :

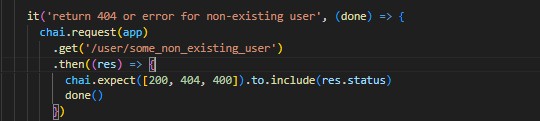
db.hgetall(username, (err, obj) => { ... }). Si err est présent, rappeler callback(err, null) ; si obj est null ou undefined, rappeler callback(new Error('User not found'), null) ; sinon, compléter obj.username = username (si besoin) et rappeler callback(null, obj).



3. Écrire les tests d’API (router) — tests d’intégration HTTP qui échouent d’abord

Ensuite, les tests d’API exercent les routes Express via l’app (ex. avec chai-http ou supertest). Les tests couvrent :

1. création + récupération — faire un POST /user pour créer un utilisateur de test, puis GET /user/:username et s’attendre à 200 et au corps JSON de l’utilisateur ;
2. utilisateur manquant — faire GET /user/:non\_existing et s’attendre à un code d’erreur approprié (404) et à un message d’erreur clair.

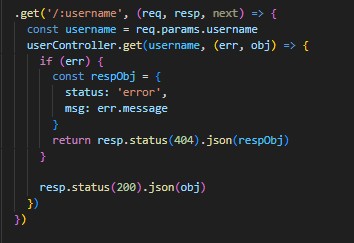


# 4. Implémenter la route GET — transformer le callback du controller en réponse HTTP

La dernière étape consiste à implémenter la route Express userRouter.get('/:username', ...) pour faire passer les tests d’API.

Dans le handler : on extrait const username = req.params.username, appeler

userController.get(username, (err, user) => { ... }) et traduit la réponse en HTTP : si err renvoyé par le controller, répond res.status(404).json({ error: err.message }), sinon res.status(200).json(user).



# 5. Run tests

Pour conclure on npm test qui nous montre bien la bonne implémentation de tout les paramètres:

