

Angel Velasco

Lab Continuous testing

Objectives:

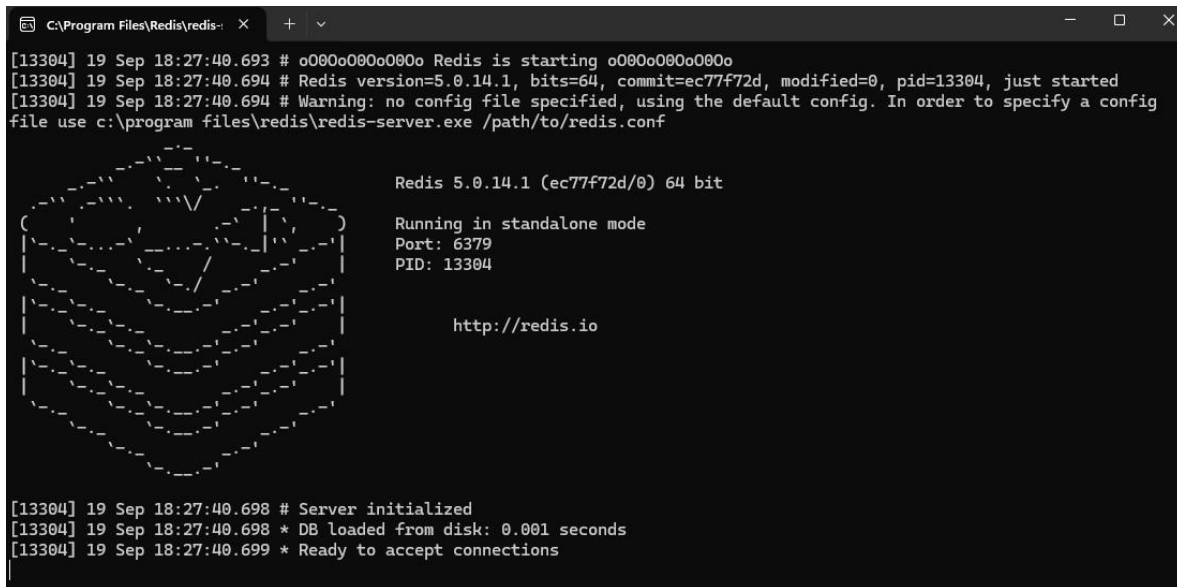
- 1) Use prepared **User API** application and run tests
- 2) Using **test-driven development** (TDD) create GET user functionality

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 :



```
C:\Program Files\Redis\redis- X + v
[13304] 19 Sep 18:27:40.693 # o000o000o000o Redis is starting o000o000o000o
[13304] 19 Sep 18:27:40.694 # Redis version=5.0.14.1, bits=64, commit=ec77f72d, modified=0, pid=13304, just started
[13304] 19 Sep 18:27:40.694 # Warning: no config file specified, using the default config. In order to specify a config
file use c:\program files\redis\redis-server.exe /path/to/redis.conf

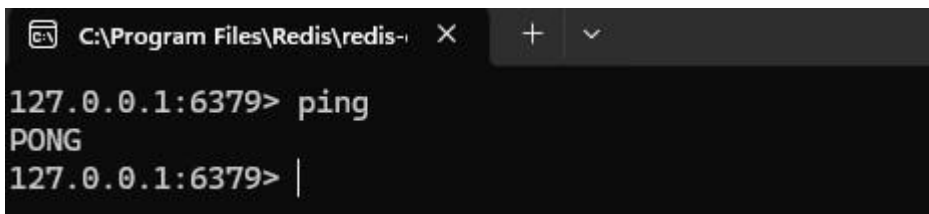
Redis 5.0.14.1 (ec77f72d/0) 64 bit

Running in standalone mode
Port: 6379
PID: 13304

http://redis.io

[13304] 19 Sep 18:27:40.698 # Server initialized
[13304] 19 Sep 18:27:40.698 * DB loaded from disk: 0.001 seconds
[13304] 19 Sep 18:27:40.699 * Ready to accept connections
```

Ping Redis-cli :



```
C:\Program Files\Redis\redis- X + v
127.0.0.1:6379> ping
PONG
127.0.0.1:6379> |
```

Use prepared User API application and run tests

npm test :

```
Configure
  ✓ load default json configuration file
  ✓ load custom configuration

Redis
  ✓ should connect to Redis

User
  Create
    ✓ create a new user
    ✓ passing wrong user parameters

User REST API
  POST /user
Server listening the port 3000
  ✓ create a new user
  ✓ pass wrong parameters
```

Npm start :

```
PS C:\Users\angel\Downloads\lab> npm start

> ece-userapi@1.0.0 start
> node src/index.js

Server listening the port 3000
█
```

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:

- (a) 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`) ;
- (b) quand l'utilisateur n'existe pas, la fonction rappelle le callback avec une erreur (par ex. `new Error('User not found')`) et `obj === null`.

```
it('get a user by username', (done) => {
  const user = {
    username: 'getuser',
    firstname: 'Get',
    lastname: 'User'
  }

  UserController.create(user, (err, result) => {
    expect(err).toBeNull()
    expect(result).toEqual('OK')

    UserController.get(user.username, (err2, obj) => {
      expect(err2).toBeNull()
      expect(obj).toBeAn('object')
      expect(obj.username).toEqual(user.username)
      expect(obj.firstname).toEqual(user.firstname)
      expect(obj.lastname).toEqual(user.lastname)
      done()
    })
  })
})

it('cannot get a user when it does not exist', (done) => {
  UserController.get('this_user_does_not_exist', (err, obj) => {
    expect(err).not.toBeNull()
    expect(obj).toBeNull()
    done()
  })
})
```

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)`.

```
get: (username, callback) => {
  db.hgetall(username, (err, obj) => {
    if (err) return callback(err, null)

    if (!obj) {
      return callback(new Error("User not found"), null)
    }
    obj.username = username
    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 :

- (a) 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 ;
- (b) utilisateur manquant — faire `GET /user/:non_existing` et s'attendre à un code d'erreur approprié (404) et à un message d'erreur clair.

```
it('return 404 or error for non-existing user', (done) => {
  chai.request(app)
    .get('/user/some_non_existing_user')
    .then((res) => {
      chai.expect([200, 404, 400]).to.include(res.status)
      done()
    })
})
```

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)`.

```
.get('/:username', (req, resp, next) => {  
  const username = req.params.username  
  userController.get(username, (err, obj) => {  
    if (err) {  
      const respObj = {  
        status: 'error',  
        msg: err.message  
      }  
      return resp.status(404).json(respObj)  
    }  
    resp.status(200).json(obj)  
  })  
})
```

5. Run tests

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

```
PS C:\Users\angel\Downloads\lab> npm test

> ece-userapi@1.0.0 test
> mocha test/*.js

Server listening the port 3000

Configure
  ✓ load default json configuration file
  ✓ load custom configuration

Redis
  ✓ should connect to Redis

User
  Create
    ✓ create a new user
    ✓ passing wrong user parameters
    ✓ avoid creating an existing user
  Get
    ✓ get a user by username
    ✓ cannot get a user when it does not exist

User REST API
  POST /user
    ✓ create a new user
    ✓ pass wrong parameters
  GET /user
    ✓ get an existing user by username
    ✓ return 404 or error for non-existing user

12 passing (64ms)
```