EIDD — Année Universitaire 2025-2026

Fiabilité Systèmes Critiques

Rapport de Projet

Supervision d'un robot via API REST

Mimouna Nour Elkamar
Berne–Gallieri Tom

Février 2026

Table des matières

1 Introduction

Ce projet consiste à développer une solution logicielle permettant à un robot de communiquer avec un système de supervision via une **API REST**. L'objectif est de mettre en pratique les concepts vus en cours : architecture REST, persistance des données avec JPA, tests unitaires avec JUnit, et développement en C++.

Le système se compose de trois éléments :

— **Spring Boot** : le serveur de supervision (API REST + base de données)

— **C++** : simule le robot qui envoie sa position et récupère ses missions

— **Unity** : visualisation 3D du robot en mouvement

L'architecture globale est la suivante :

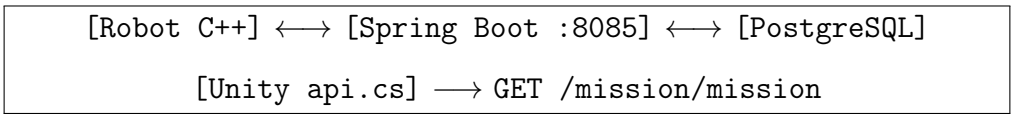

FIGURE 1 – Architecture globale du système

2 Implémentation de l'API REST

2.1 Endpoints Robot

Méthode	URL	Description
GET	`/robot`	Récupère tous les robots
GET	`/robot/{id}`	Récupère un robot par ID
POST	`/robot`	Crée un robot
PUT	`/robot/{id}`	Met à jour un robot
DELETE	`/robot/{id}`	Supprime un robot

TABLE 1 – Endpoints de l'API Robot

2.2 Endpoints Mission

Méthode	URL	Description
GET	`/mission/all`	Récupère toutes les missions
GET	`/mission/mission`	Retourne la mission suivante (cycle)

TABLE 2 – Endpoints de l'API Mission

2.3 Gestion des erreurs

Toutes les exceptions sont encapsulées dans `RobotException`, qui étend `RuntimeException`. Le service intercepte les `PessimisticLockingFailureException` (verrous DB) et les `RuntimeException` inattendues pour les convertir en exceptions métier claires.

3 Tests unitaires

3.1 Approche et outils

Les tests sont écrits avec **JUnit 5** et **Mockito**. L'approche consiste à isoler chaque composant : le repository est mocké, donc aucune base de données réelle n'est nécessaire pour les tests.

3.2 MissionServiceTest

Ce test vérifie le comportement du service de gestion des missions :

```java
@ExtendWith(MockitoExtension.class)
class MissionServiceTest {

    @InjectMocks
    private MissionService missionService;

    @Mock
    MissionRepository missionRepository;

    @Test
    void getAll() {
        when(missionRepository.findAll()).thenReturn(Arrays.asList(
                new Mission(1, 0.0, 0.0, 0.0),
                new Mission(2, 0.0, 10.0, 0.0)
        ));
        List<Mission> result = missionService.getAll();
        assertThat(result).hasSize(2);
        assertThat(result.get(0).getId()).isEqualTo(1);
    }

    @Test
    void get_all_should_catch_exception() {
        doThrow(new PessimisticLockingFailureException("DB␣lock"))
                .when(missionRepository).findAll();
        assertThatCode(() -> missionService.getAll())
                .isInstanceOf(RobotException.class);
    }

    @Test
    void getMission() {
        when(missionRepository.findAll()).thenReturn(Arrays.asList(
                new Mission(1, 0.0, 0.0, 0.0),
                new Mission(2, 0.0, 10.0, 0.0),
                new Mission(3, 10.0, 10.0, -90.0)
        ));
        missionService.createMissionMap();

        assertThat(missionService.getMission().getId()).isEqualTo(1);
        assertThat(missionService.getMission().getId()).isEqualTo(2);
        assertThat(missionService.getMission().getId()).isEqualTo(3);
        // Verification du cycle
        assertThat(missionService.getMission().getId()).isEqualTo(1);
    }
}
```

3.3 RobotApiControllerTest

Ce test vérifie les endpoints REST avec **MockMvc**, qui simule des requêtes HTTP sans démarrer un vrai serveur :

```
@ExtendWith(MockitoExtension.class)
class RobotApiControllerTest {

    @Mock
    StdRobotService robotService;

    RobotApiController robotApiController;
    MockMvc mockMvc;

    @BeforeEach
    void setUp() {
        robotApiController = new RobotApiController(
            new ObjectMapper(), new MockHttpServletRequest());
        ReflectionTestUtils.setField(
            robotApiController, "robotService", robotService);
        mockMvc = MockMvcBuilders
            .standaloneSetup(robotApiController).build();
    }

    @Test
    void robotGet() throws Exception {
        when(robotService.getAll()).thenReturn(Arrays.asList(
                new Robot(1, 0.0, 0.0, 0.0, 1.0, 2.0),
                new Robot(2, 5.0, 10.0, 90.0, 0.5, 3.0)
        ));
        mockMvc.perform(get("/robot")
                .accept(MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(jsonPath("$.length()").value(2));
    }
}
```

3.4 Résultats

L'ensemble des tests a été exécuté avec la commande `./gradlew test`. Le rapport généré montre :

Classe	Tests	Échecs	Résultat
`MissionTest`	1	0	100%
`MissionServiceTest`	5	0	100%
`RobotApiControllerTest`	5	0	100%
Total	**11**	**0**	**100%**

TABLE 3 – Résultats des tests unitaires

4 Tests de validation Postman

4.1 Démarche

Les tests Postman permettent de valider le comportement réel de l'API avec une base de données active. Spring Boot est lancé avec `./gradlew bootRun`, puis les requêtes sont envoyées depuis Postman.

4.2 Requêtes effectuées

Les captures d'écran de chaque requête sont disponibles en annexe (Figures **??** à 9).

N°	Requête	Résultat attendu	Résultat obtenu
1	`GET /mission/all`	200 + 5 missions	200 OK
2	`GET /mission/mission`	200 + 1 mission	200 OK
3	`POST /robot`	200 + robot créé	200 OK
4	`GET /robot`	200 + liste robots	200 OK
5	`GET /robot/1`	200 + robot 1	200 OK
6	`PUT /robot/1`	200 + robot mis à jour	200 OK
7	`DELETE /robot/1`	204 No Content	204 OK
8	`GET /robot/1`	404 Not Found	404 OK

TABLE 4 – Résultats des tests Postman

4.3 Exemple de réponse

La requête `GET /mission/all` retourne :

```
[
  {"id":1,"x":0.0,"y":0.0,"theta":0.0},
  {"id":2,"x":0.0,"y":10.0,"theta":0.0},
  {"id":3,"x":10.0,"y":10.0,"theta":-90.0},
  {"id":4,"x":10.0,"y":0.0,"theta":-180.0},
  {"id":5,"x":0.0,"y":0.0,"theta":-270.0}
]
```

5 Code C++

5.1 Bibliothèque utilisée : libcurl

Le choix s'est porté sur **libcurl** pour sa simplicité et sa large adoption. Elle permet d'effectuer des requêtes HTTP en C++ sans dépendances lourdes.

Installation :

```
sudo apt-get install libcurl4-openssl-dev
g++ robot.cpp -o robot -lcurl
```

5.2 Fonctionnement

Le programme simule un robot qui :

1. Appelle `GET /mission/mission` pour obtenir sa prochaine mission
2. Met à jour sa position interne avec les coordonnées reçues
3. Appelle `PUT /robot/1` pour envoyer sa nouvelle position au serveur
4. Attend 3 secondes et recommence

5.3 Fonction GET mission

```
Mission getMission() {
    CURL* curl;
    std::string response;
    Mission mission = {0, 0.0, 0.0, 0.0};

    curl = curl_easy_init();
    if (curl) {
        curl_easy_setopt(curl, CURLOPT_URL,
            "http://localhost:8085/mission/mission");
        curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION,
            writeCallback);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &response);
        curl_easy_perform(curl);

        // Parse du JSON recu
        auto extract = [&](const std::string& key) -> double {
            std::string search = "\"" + key + "\":";
            size_t pos = response.find(search);
            if (pos == std::string::npos) return 0.0;
            pos += search.length();
            return std::stod(response.substr(pos));
        };
        mission.id    = (int)extract("id");
        mission.x     = extract("x");
        mission.y     = extract("y");
        mission.theta = extract("theta");
        curl_easy_cleanup(curl);
    }
    return mission;
}
```

5.4 Fonction PUT robot

```
void updateRobot(const Robot& robot) {
    CURL* curl = curl_easy_init();
    if (curl) {
        std::string url = "http://localhost:8085/robot/"
                        + std::to_string(robot.id);
        std::string jsonBody = "{"
            "\"id\":"         + std::to_string(robot.id)        + ","
            "\"x\":"          + std::to_string(robot.x)         + ","
            "\"y\":"          + std::to_string(robot.y)         + ","
            "\"theta\":"      + std::to_string(robot.theta)     + ","
            "\"v\":"          + std::to_string(robot.v)         + ","
            "\"ultraSound\":" + std::to_string(robot.ultrasound)+
        "}";

        struct curl_slist* headers = nullptr;
        headers = curl_slist_append(headers,
                    "Content-Type: application/json");

        curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
        curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "PUT");
        curl_easy_setopt(curl, CURLOPT_POSTFIELDS,
                         jsonBody.c_str());
        curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
        curl_easy_perform(curl);

        curl_slist_free_all(headers);
        curl_easy_cleanup(curl);
    }
}
```

5.5 Résultat d'exécution

```
=== Demarrage du robot ===
Mission recue : {"id":1,"x":0.0,"y":0.0,"theta":0.0}
Robot mis a jour : {"id":1,"x":0.0,"y":0.0,"theta":0.0,"v":1.0,"
   ultraSound":2.0}
Attente 3 secondes...
Mission recue : {"id":2,"x":0.0,"y":10.0,"theta":0.0}
Robot mis a jour : {"id":1,"x":0.0,"y":10.0,"theta":0.0,"v":1.0,"
   ultraSound":2.0}
Attente 3 secondes...
...
```

6 Conclusion

Ce projet a permis de mettre en pratique l'ensemble de la chaîne de développement logiciel industriel :

— **Architecture REST** : conception et implémentation d'une API complète avec Spring Boot

— **Persistance des données** : utilisation de JPA et PostgreSQL pour le stockage des entités

— **Tests unitaires** : 11 tests avec JUnit 5 et Mockito, 100% de réussite
— **Tests de validation** : 8 requêtes Postman validant le comportement réel de l'API
— **Code C++** : communication REST avec libcurl, simulation du comportement du robot

Les compétences acquises couvrent aussi bien la conception d'architecture que les bonnes pratiques de test. Une amélioration possible serait d'ajouter une gestion d'erreurs plus fine côté C++ (retry automatique en cas d'échec réseau) et d'intégrer des tests d'intégration avec une base de données en mémoire (H2) pour valider l'ensemble de la chaîne sans PostgreSQL.

A Annexe : Captures d'écran Postman

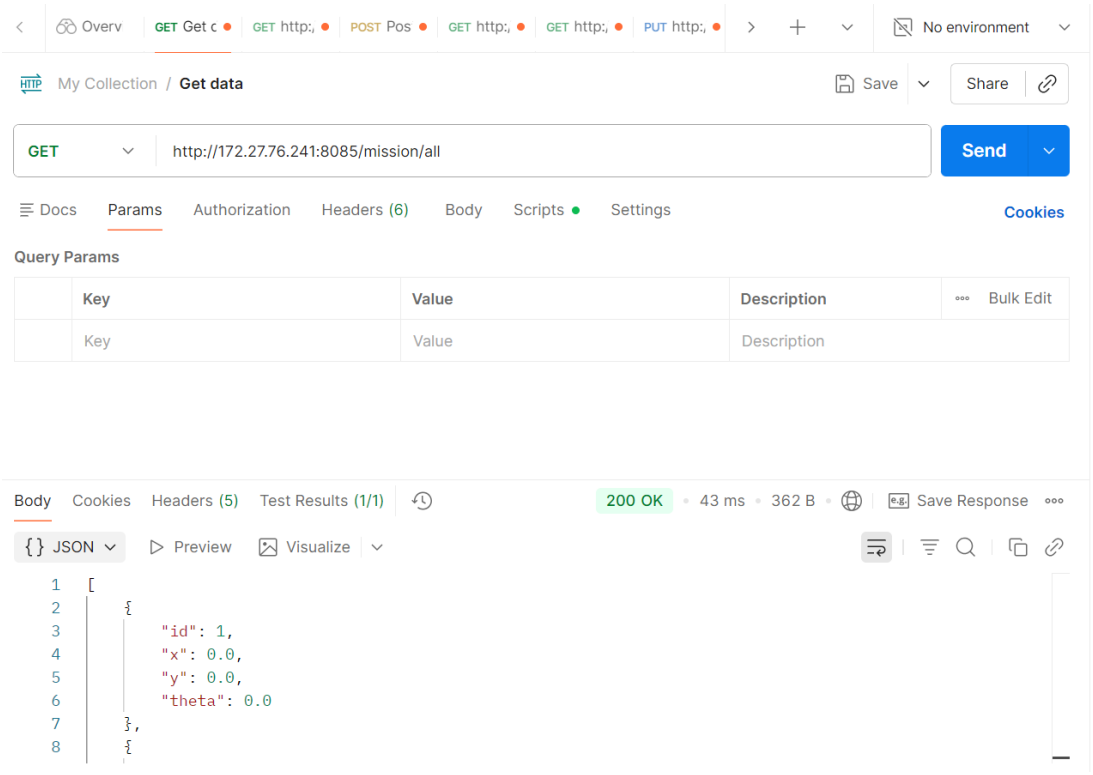

FIGURE 2 – GET /mission/all — 200 OK

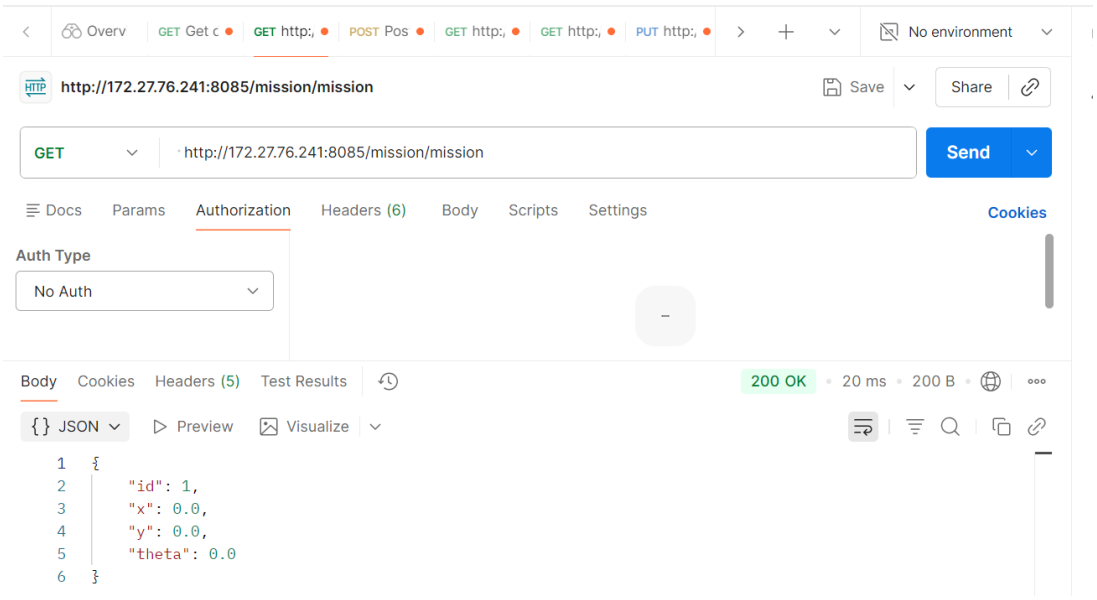

FIGURE 3 – GET /mission/mission — 200 OK

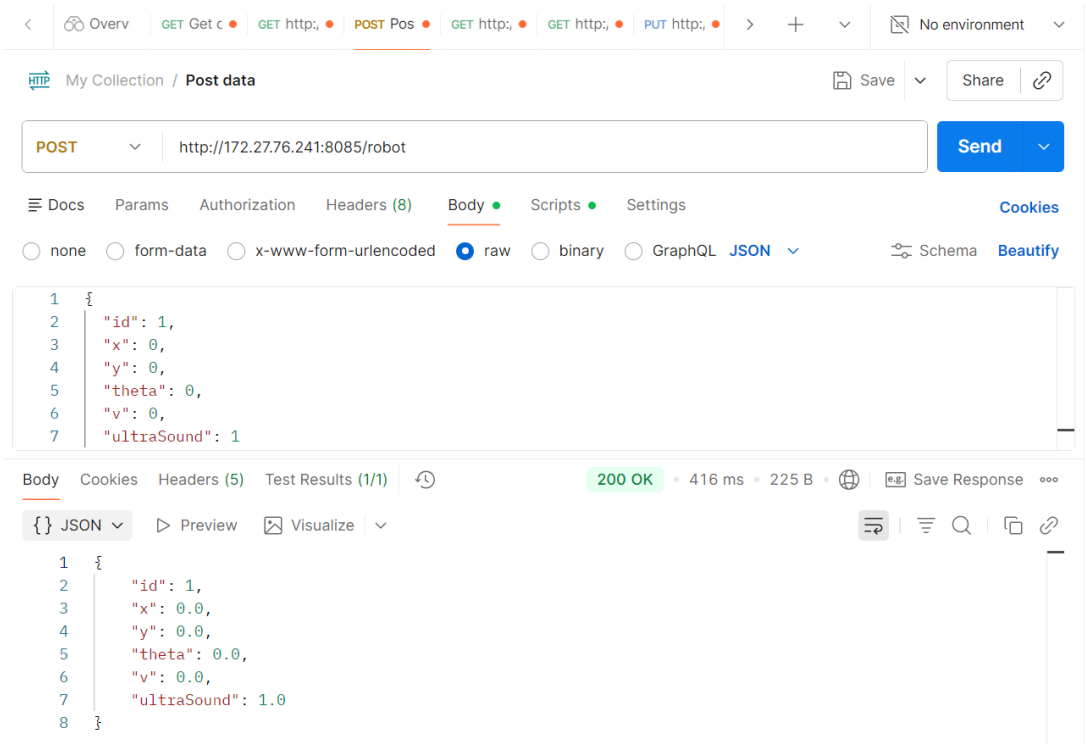

FIGURE 4 – POST /robot — 200 OK

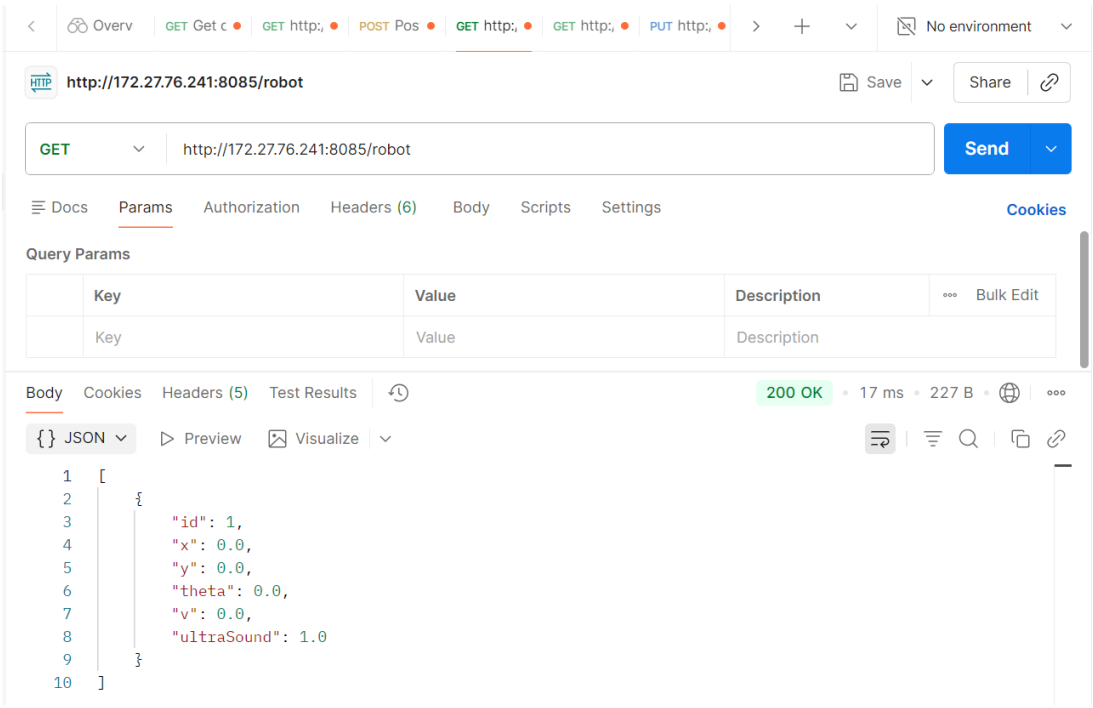

FIGURE 5 – GET /robot — 200 OK

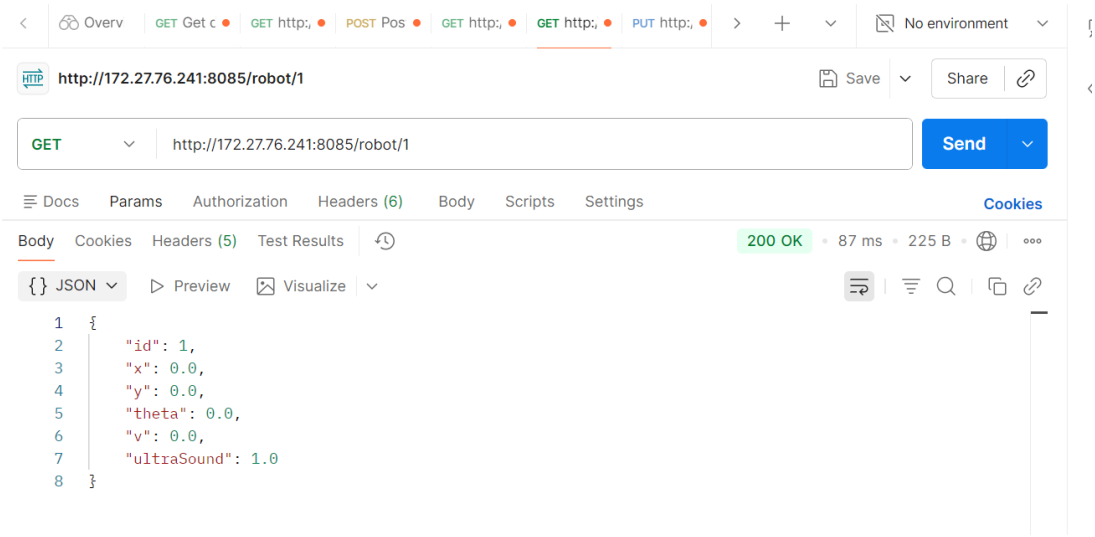

FIGURE 6 – GET /robot/1 — 200 OK

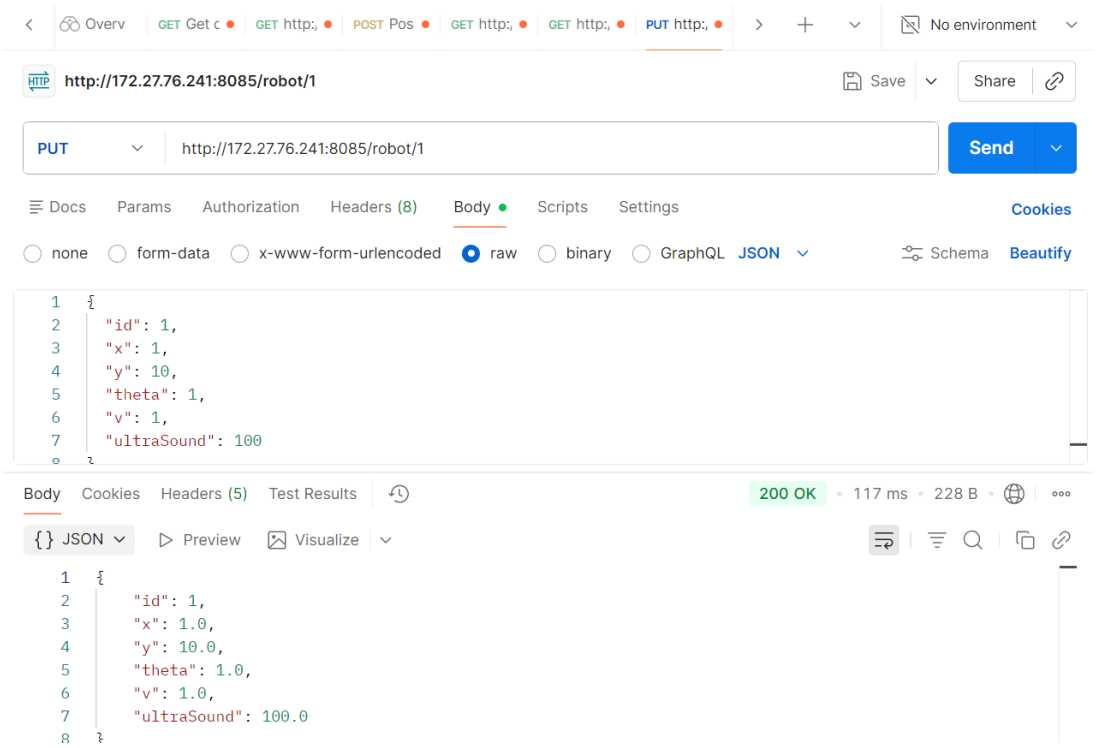

FIGURE 7 – PUT /robot/1 — 200 OK

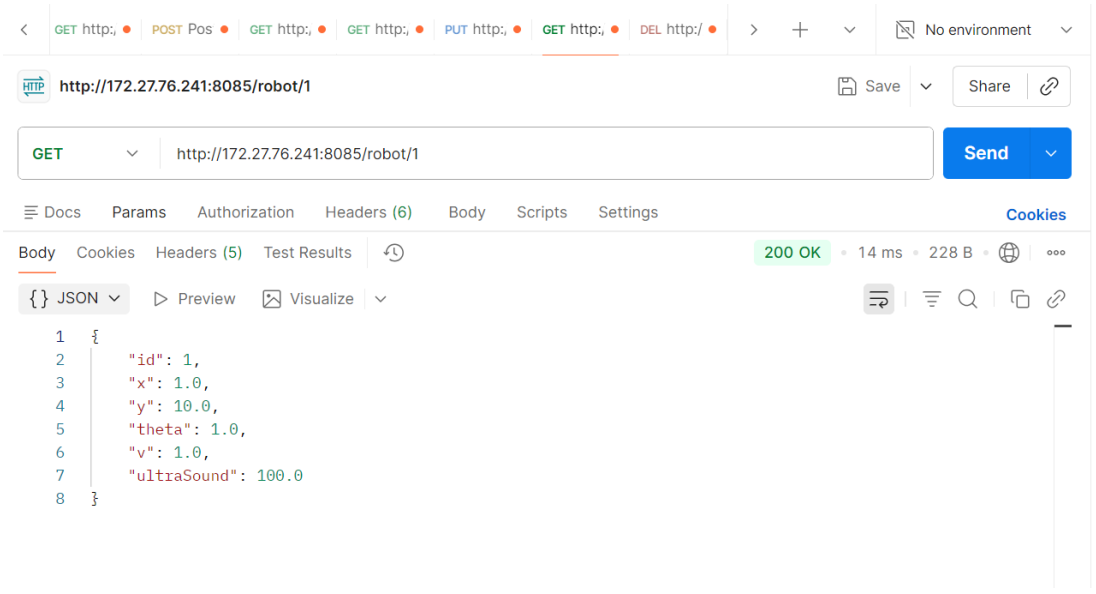

FIGURE 8 – DELETE /robot/1 — 204 No Content

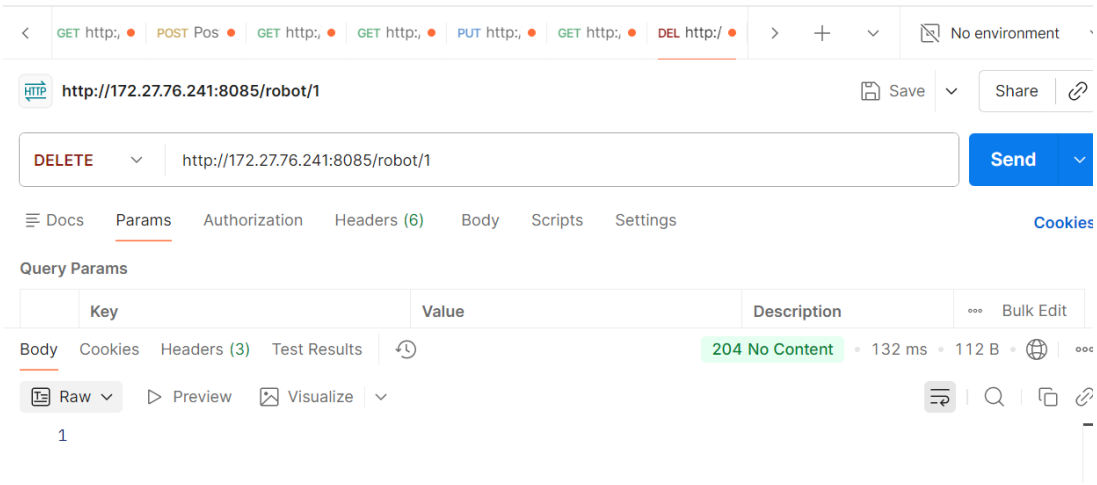

FIGURE 9 – GET /robot/1 après suppression — 404 Not Found