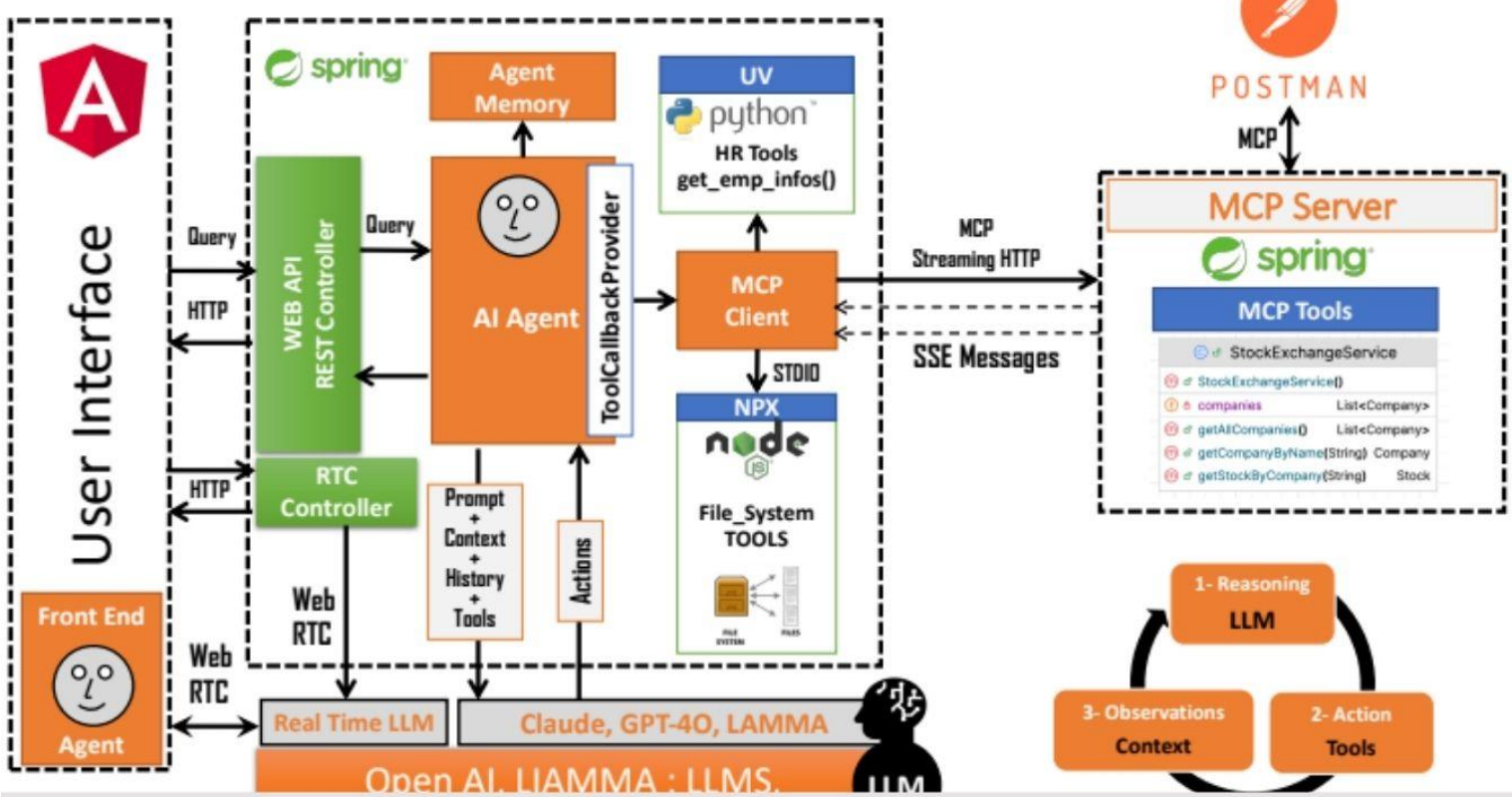

Chatbot interactif

BASÉ SUR SPRING AI ET MCP

2024/2025

SOUS LA SUPERVISION DE : MR MOHAMED YOUSSFI

RÉALISÉ PAR : BELAYACHI CHARAF

3ÈME ANNÉE: CYCLE D'INGÉNIEUR D'ÉTAT
EN GÉNIE INFORMATIQUE

06/07/2025

Table des matières

RÉSUMÉ

Ce rapport présente la conception et le développement d'une application chatbot basée sur le protocole MCP, permettant une communication en temps réel entre plusieurs serveurs (développés en Spring AI, NodeJS et Python) et un client centralisé. L'architecture mise en œuvre utilise les modes SSE et STDIO pour garantir des échanges rapides et fiables entre les différents composants.

Le client MCP orchestre les interactions avec les serveurs, tandis qu'un frontend Angular offre une interface utilisateur simple et réactive pour dialoguer avec le chatbot. Malgré quelques difficultés rencontrées dans le choix et l'intégration d'un modèle d'intelligence artificielle performant, le projet a permis de construire une solution modulaire, robuste et évolutive, posant les bases pour des améliorations futures.

I. INTRODUCTION

1. Contexte du projet

Avec l'essor des technologies conversationnelles et de l'intelligence artificielle, les applications chatbot sont devenues des outils indispensables pour automatiser les interactions entre utilisateurs et systèmes informatiques. Ce projet s'inscrit dans ce contexte, en développant une application chatbot reposant sur le protocole MCP (Multi-Channel Protocol), qui facilite la communication en temps réel entre différents serveurs et clients.

2. Objectifs du projet

L'objectif principal est de concevoir et de mettre en œuvre une architecture distribuée intégrant plusieurs serveurs MCP développés en différentes technologies (Java Spring AI, NodeJS, Python) et un client MCP centralisé, afin de garantir une communication fluide et modulable via le protocole MCP. Le projet inclut également le développement d'un frontend Angular pour permettre l'interaction utilisateur avec le chatbot.

3. Travail à faire

Pour atteindre ces objectifs, plusieurs étapes clés ont été planifiées :

- Créer un serveur MCP basé sur Spring AI.
- Tester le serveur MCP en utilisant Postman.
- Développer un client MCP capable d'intégrer :
  a. Le serveur MCP Spring de type SSE (Server-Sent Events).
  b. Un serveur MCP NodeJS utilisant le système de fichiers et communiquant en mode STDIO.
  c. Un serveur MCP Python également en mode STDIO.
- Concevoir et implémenter une interface frontend Angular pour interagir avec le client MCP.

4. Technologies utilisées

Ce projet mobilise plusieurs technologies modernes et complémentaires :

- **Spring AI** pour la création des serveurs et clients MCP Java.
- **NodeJS** et **Python** pour le développement des serveurs MCP auxiliaires.
- **Protocole MCP**, notamment les modes SSE et STDIO, pour la communication.
- **Angular** pour la réalisation de l'interface utilisateur web.
- **Postman** et **Swagger UI** pour les tests et la documentation des API.

5. Structure du rapport

Ce rapport présente dans un premier temps l'architecture globale du projet, puis détaille chaque étape du travail réalisé, de la création des serveurs MCP jusqu'au développement du frontend, avant de conclure par un bilan et des perspectives.

II. Architecture globale du projet

1. Présentation schématique de l'architecture

Le projet s'appuie sur une architecture distribuée qui permet l'interconnexion et la communication en temps réel entre plusieurs serveurs MCP développés avec différentes technologies, un client centralisé, ainsi qu'une interface utilisateur web. La communication entre ces composants repose sur le protocole MCP, qui offre des modes adaptés aux différents types d'échanges (SSE et STDIO).

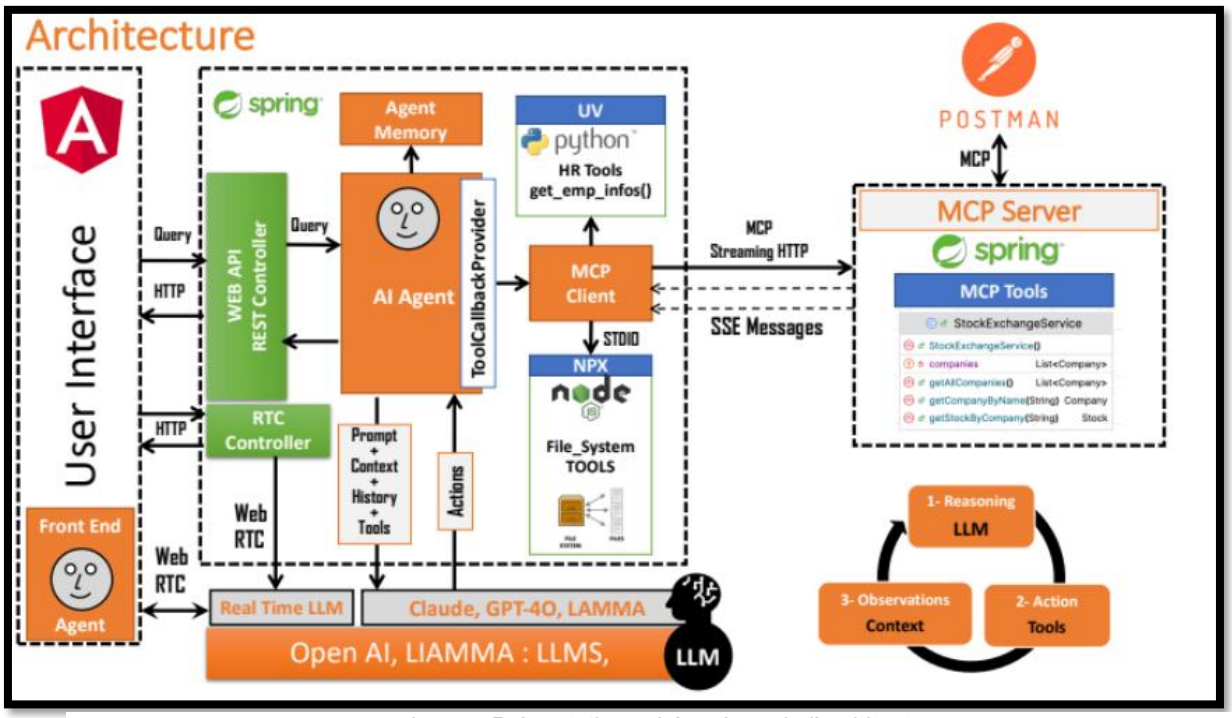

Figure 1:Présentation schématique de l'architecture

Voici une représentation simplifiée de l'architecture globale :

2. Description des composants principaux

Frontend Angular : Interface utilisateur développée avec Angular. Elle permet à l'utilisateur d'interagir avec l'application chatbot, d'envoyer des requêtes et de recevoir des réponses via le client MCP.

Client MCP Spring AI : Composant central qui sert d'intermédiaire entre le frontend et les différents serveurs MCP. Il gère la communication via le protocole MCP, notamment :

- ✓ La connexion en mode SSE avec le serveur MCP Spring AI principal.
- ✓ La communication bidirectionnelle via STDIO avec les serveurs NodeJS et Python.

Serveur MCP Spring AI (SSE) : Serveur principal basé sur Spring AI, utilisant le protocole MCP en mode SSE (Server-Sent Events). Ce serveur diffuse des événements en temps réel vers le client.

Serveur MCP NodeJS (STDIO) : Serveur auxiliaire développé en NodeJS, utilisant la communication STDIO (entrée/sortie standard). Il fournit des services spécifiques accessibles via le client MCP.

Serveur MCP Python (STDIO) : Serveur complémentaire développé en Python, communiquant également via STDIO. Il peut intégrer des fonctionnalités avancées ou spécifiques telles que des traitements AI.

3. Description du protocole MCP et modes de communication

Le protocole MCP est un protocole de communication multicanal conçu pour faciliter les échanges entre clients et serveurs dans une architecture distribuée. Il supporte plusieurs modes de communication :

- **SSE (Server-Sent Events)** : un canal unidirectionnel où le serveur envoie des événements en temps réel vers le client. Ce mode est utilisé pour le serveur MCP Spring AI, permettant la diffusion continue d'informations.
- **STDIO (Standard Input/Output)** : mode bidirectionnel basé sur les flux standard d'entrée et sortie. Il est particulièrement adapté à l'intégration de serveurs externes (NodeJS, Python) qui communiquent avec le client MCP via ces flux.

4. Flux de données et interactions

Le flux principal suit ces étapes :

- L'utilisateur saisit une requête ou un message via l'interface Angular.
- Le frontend transmet cette requête au client MCP Spring AI.
- Le client MCP oriente la requête :
  - Vers le serveur MCP Spring AI via SSE, pour les échanges en temps réel.
  - Vers les serveurs NodeJS ou Python via STDIO, selon la nature du service demandé.
- Les serveurs traitent la requête et renvoient une réponse au client MCP, qui la transmet à son tour au frontend.
- L'utilisateur visualise la réponse dans l'interface.

Cette architecture assure modularité, extensibilité et réactivité dans les échanges.

5. Diagrammes d'architecture

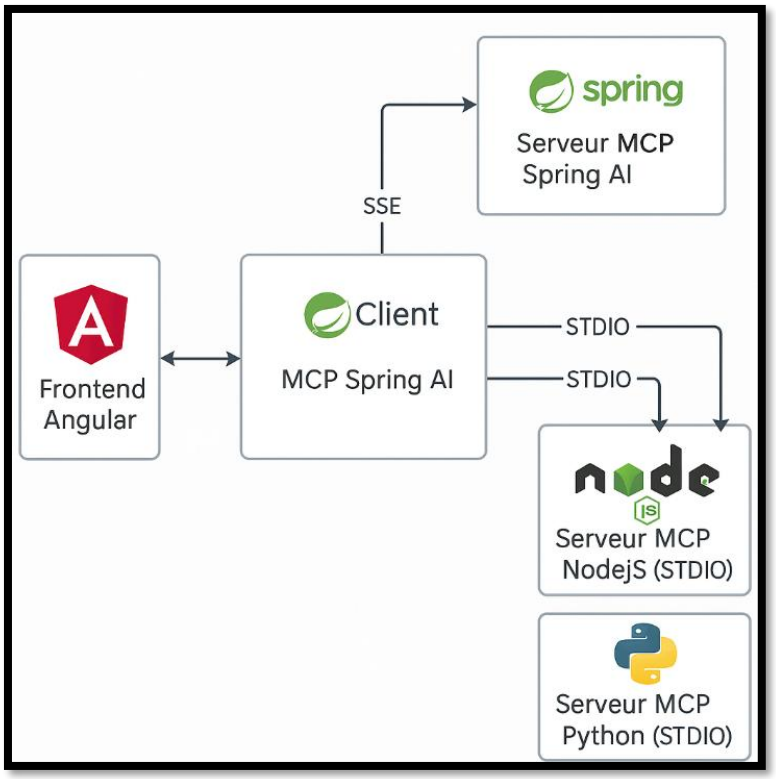

Figure 2:Diagrammes d'architecture

III. Création du serveur MCP basé sur Spring AI

1. Présentation générale

Le serveur MCP principal a été développé avec Spring AI, exploitant les fonctionnalités du protocole MCP en mode SSE (Server-Sent Events). Ce serveur assure la communication en temps réel avec les clients MCP en diffusant des événements et en exposant des outils métier.

2. Architecture et fonctionnement

L'implémentation repose sur une architecture Spring Boot classique, enrichie par les annotations spécifiques de Spring AI qui permettent d'exposer facilement des méthodes métiers en tant qu'outils MCP. Le serveur est configuré pour écouter sur un port dédié et offrir des end points SSE permettant la diffusion des événements.

3. Fonctionnalités implémentées

Le serveur expose notamment :

- Des services métiers simulant la gestion d'informations boursières pour plusieurs entreprises,
- Des méthodes accessibles via MCP qui permettent de récupérer la liste des entreprises, obtenir les détails d'une entreprise spécifique, ou obtenir la valeur boursière simulée d'une société.

4. Configuration technique

Les paramètres essentiels du serveur sont définis dans un fichier de configuration `application.properties`, notamment :

- Le nom et le type du serveur MCP,
- Les endpoints SSE utilisés,
- La gestion des notifications de changement d'outils ou de ressources,
- Le port d'écoute.

3.5 Tests et validation

5. Présentation du code

Le serveur MCP Spring AI expose plusieurs outils métier via des méthodes annotées. Voici un extrait du code principal qui définit ces outils, permettant notamment de récupérer la liste des entreprises, obtenir les détails d'une entreprise, ou la valeur boursière simulée d'une société.

mcp-server stockTools.java

```
@Service
public class StockTools {
    private List<Company> companies = List.of(
            new Company("Maroc
Telecom","Telecom",3.6,10600,"Maroc"),
            new Company("OCP","Extraction
minière",5.6,20000,"Maroc")
    );
    @Tool(description = "get all companies")
    public List<Company> getCompanies() {
        return companies;
    }
    @Tool
    public Company getCompanyByName(String name) {
        return companies.stream().filter(c-
```

```
>c.name().equals(name)).findFirst().orElseThrow(()->new
RuntimeException("Company not found"));
    }
    @Tool
    public Stock getStockByCompanyName(String name) {
        return new Stock(name,
LocalDate.now(),300+Math.random()+300);
    }
}
record Company(String name,
               String activity,
               @Description("In Milliard MA")
               double turnover,
               int employeesCount,
               String country
){}
record Stock(String companyName, LocalDate date, double stock){

}
```

McpServerApplication.java

```
@SpringBootApplication
public class McpServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(McpServerApplication.class, args);
    }

    @Bean
    public MethodToolCallbackProvider
getMethodToolCallbackProvider() {
        return MethodToolCallbackProvider
                .builder()
                .toolObjects(new StockTools())
                .build();
    }

}
```

application.properties

```
spring.application.name=mcp-server

spring.ai.mcp.server.name=stock-mcp-server
spring.ai.mcp.server.type=sync
spring.ai.mcp.server.sse-endpoint=/sse
spring.ai.mcp.server.sse-message-endpoint=/mcp/message

spring.ai.mcp.server.prompt-change-notification=true
spring.ai.mcp.server.tool-change-notification=true
spring.ai.mcp.server.resource-change-notification=true
server.port=8899
```

6. Tests :

La page liste trois outils principaux :

- **getCompanies :** récupération de la liste des entreprises.
- **getStockByCompanyName** : obtention de la valeur boursière simulée.
- **getCompanyByName** : récupération des détails d'une entreprise spécifique.

Ces outils sont accessibles par le client MCP via des requêtes conformes au protocole MCP, assurant ainsi une communication fluide et en temps réel.

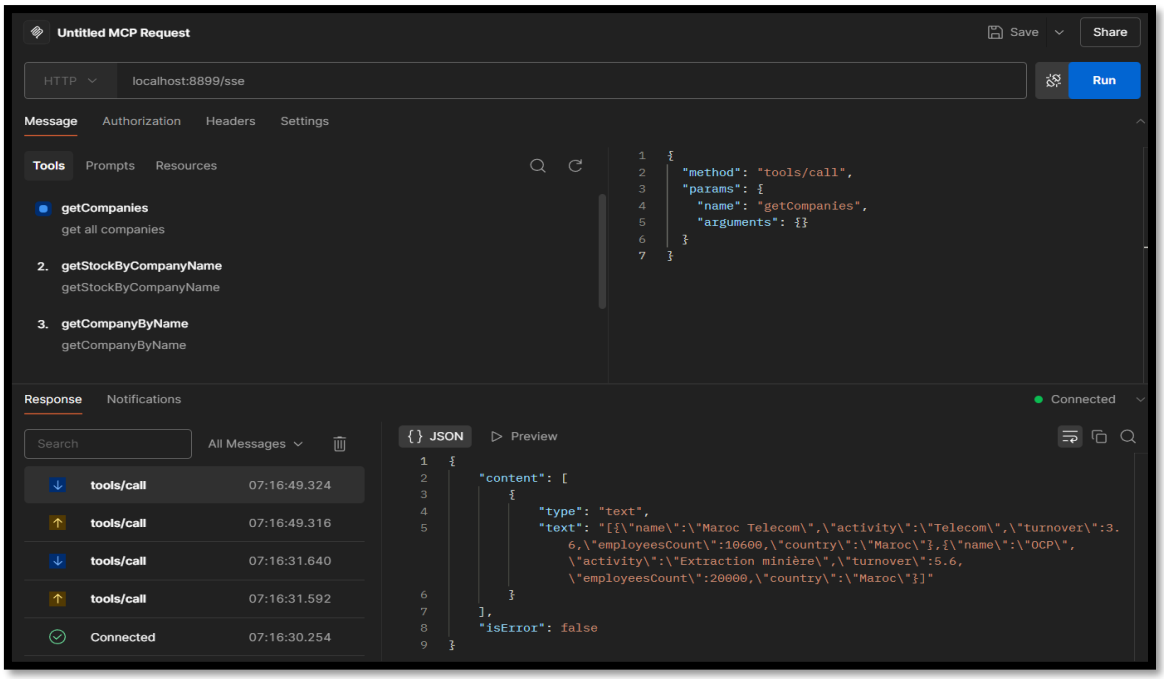

Figure 4:Test des outils exposés par le serveur MCP Spring AI via l'Endpoint SSE

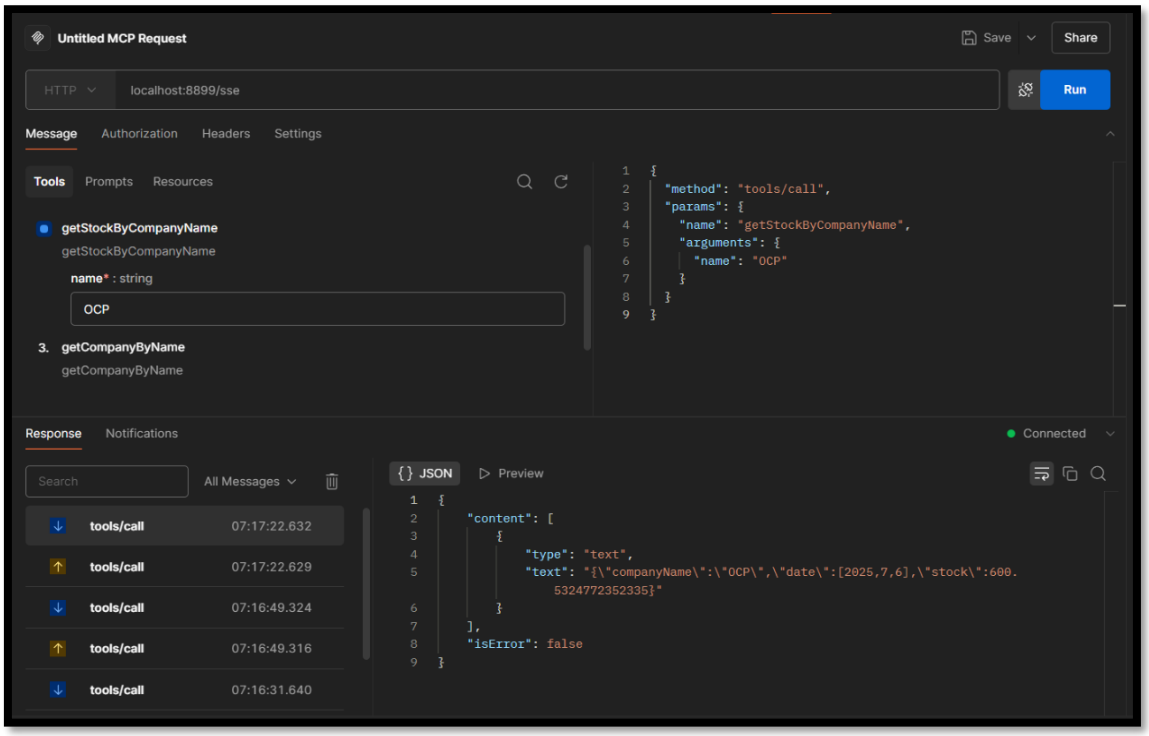

Figure 3:Test qui retourne le stock de l'entreprise par son nom

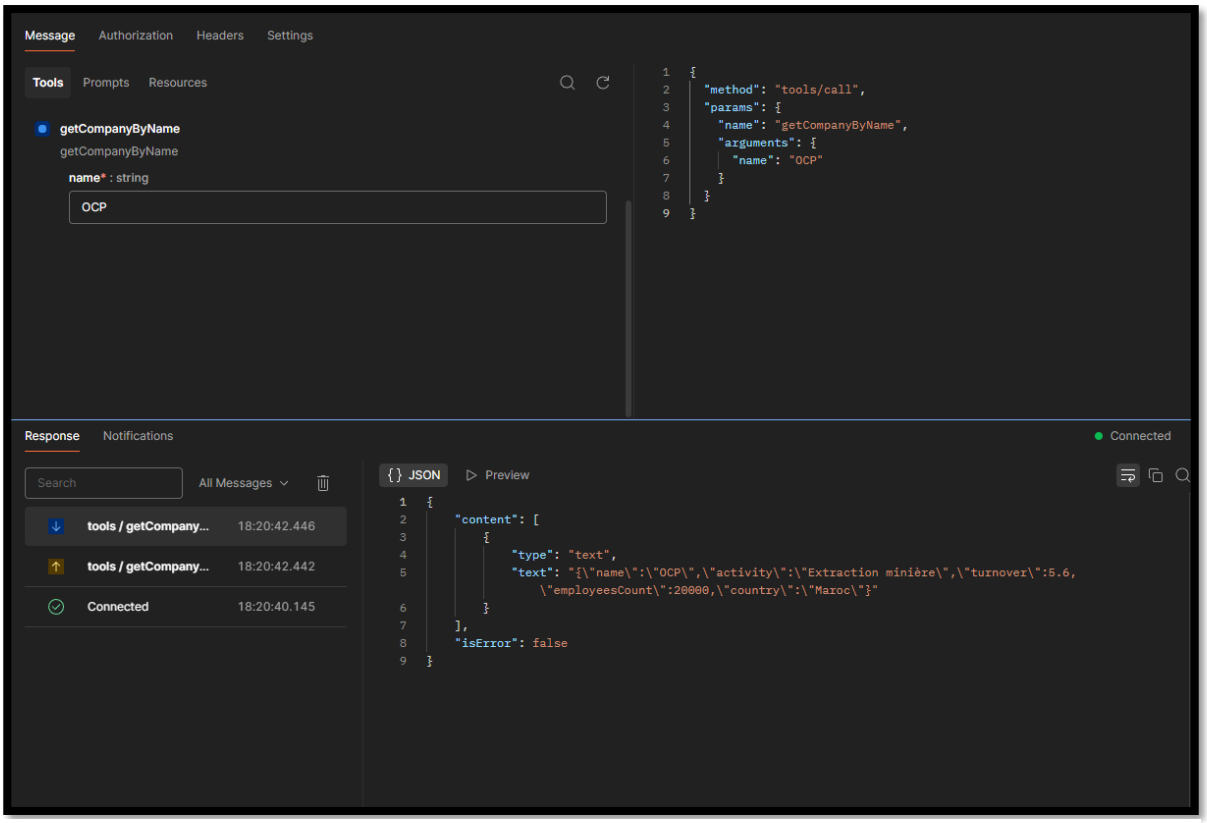

Figure 5:Test qui retourne l'entreprise par son nom

Requête GET : `http://localhost:8877/sse`

Cette requête ouvre une connexion **Server-Sent Events (SSE)** avec le serveur MCP. Elle permet au client de recevoir en temps réel les événements et notifications émis par le serveur, conformément au protocole MCP. Cette connexion reste ouverte pour assurer un flux continu de données.

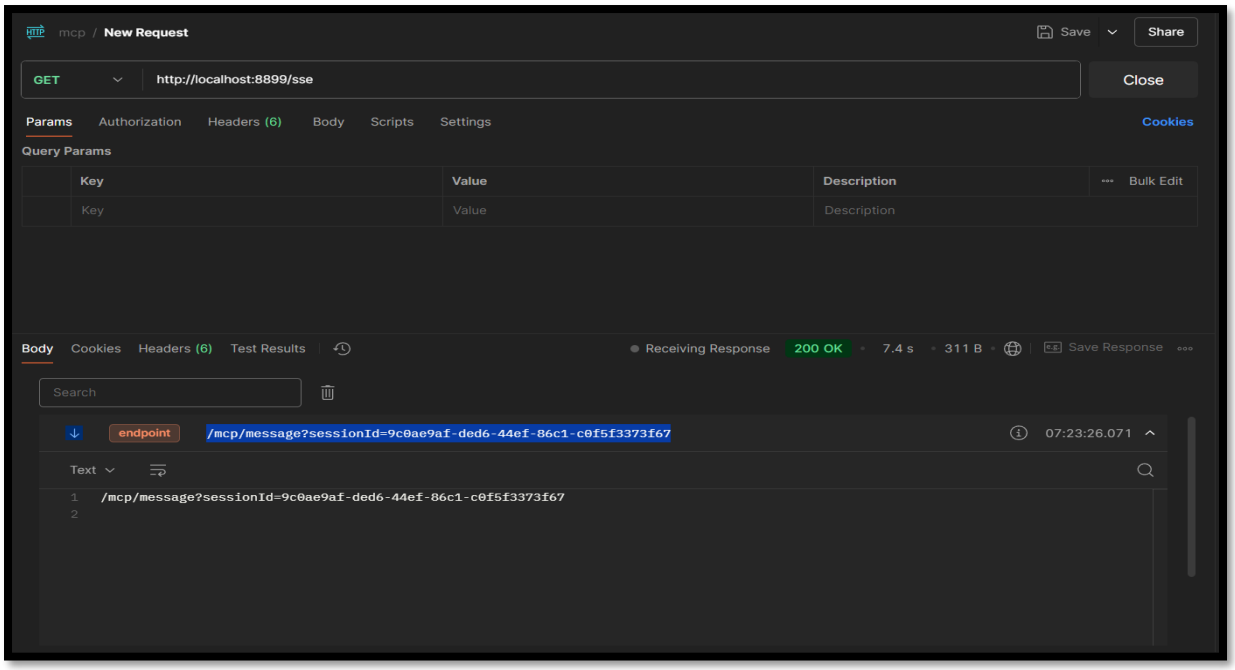

Figure 6:ouvre une connexion Server-Sent Events (SSE)

Requête POST : Initialisation de la session

URL : `http://localhost:8899/mcp/message?sessionId=9c0ae9af-ded6-44ef-86c1-c0f5f3373f67`

Cette requête initialise la session MCP entre le client et le serveur. Elle informe le serveur de la version du protocole utilisée, des capacités du client (comme la gestion des changements de liste), ainsi que des informations générales sur le client (nom et version).

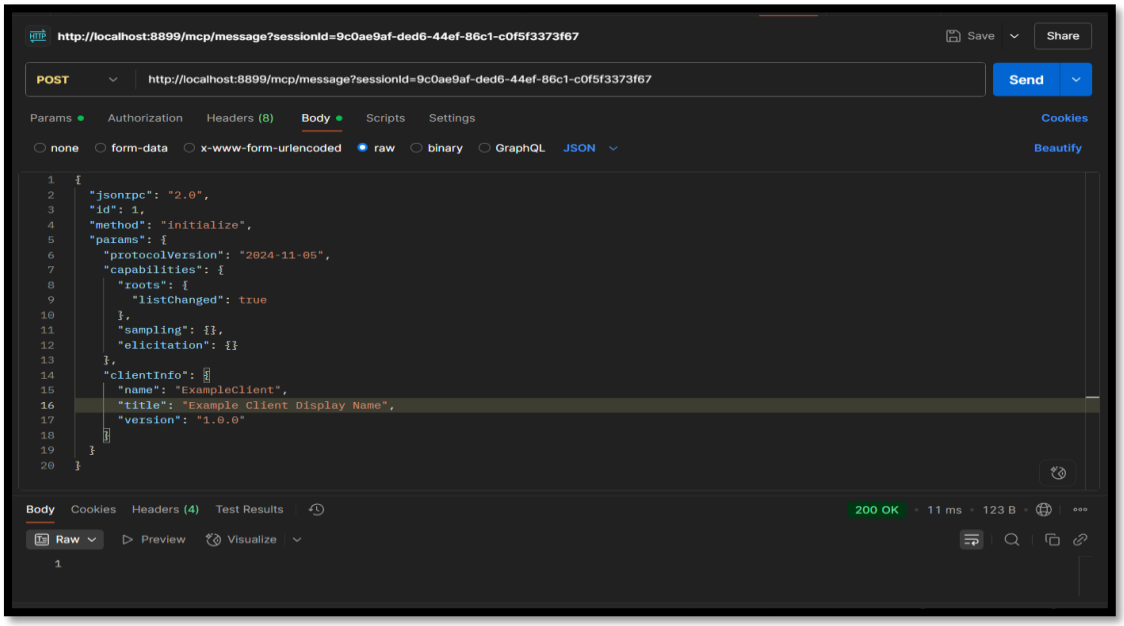

Figure 7: Initialisation de la session

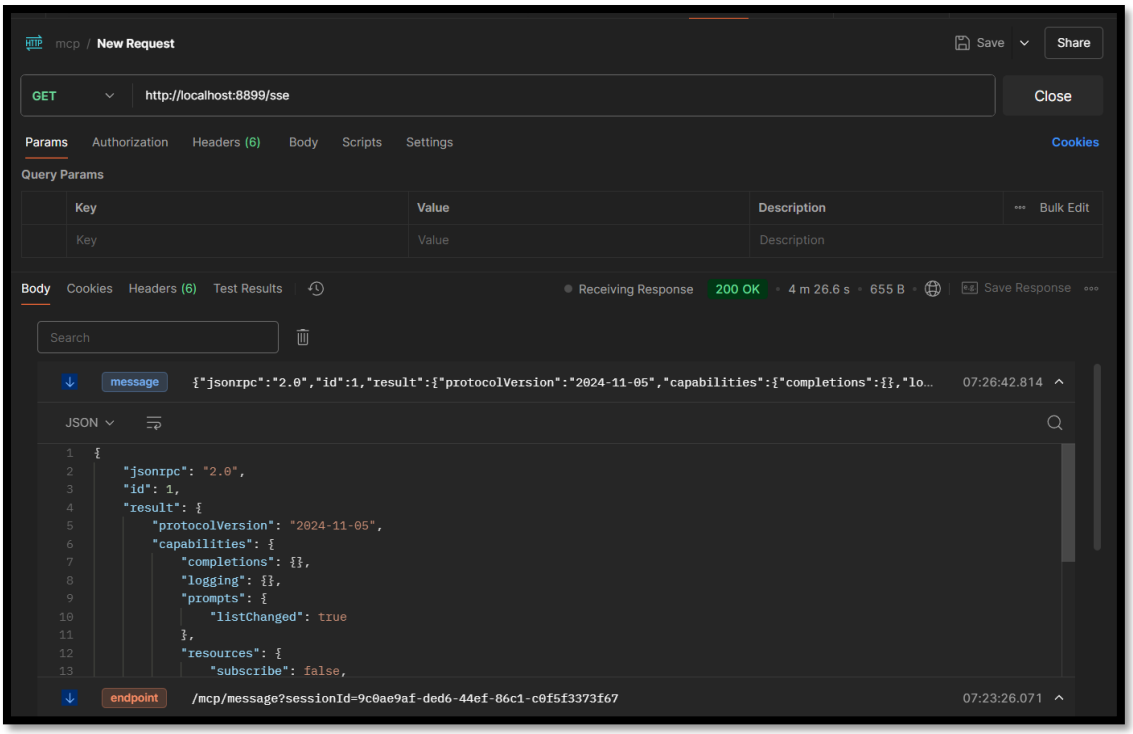

Figure 8:validation d'initialisation de la session

Requête POST : Notification d'initialisation terminée

Cette requête sert à notifier au serveur que l'initialisation du client est terminée. Elle marque le passage à l'état prêt à échanger des messages ou des commandes supplémentaires.

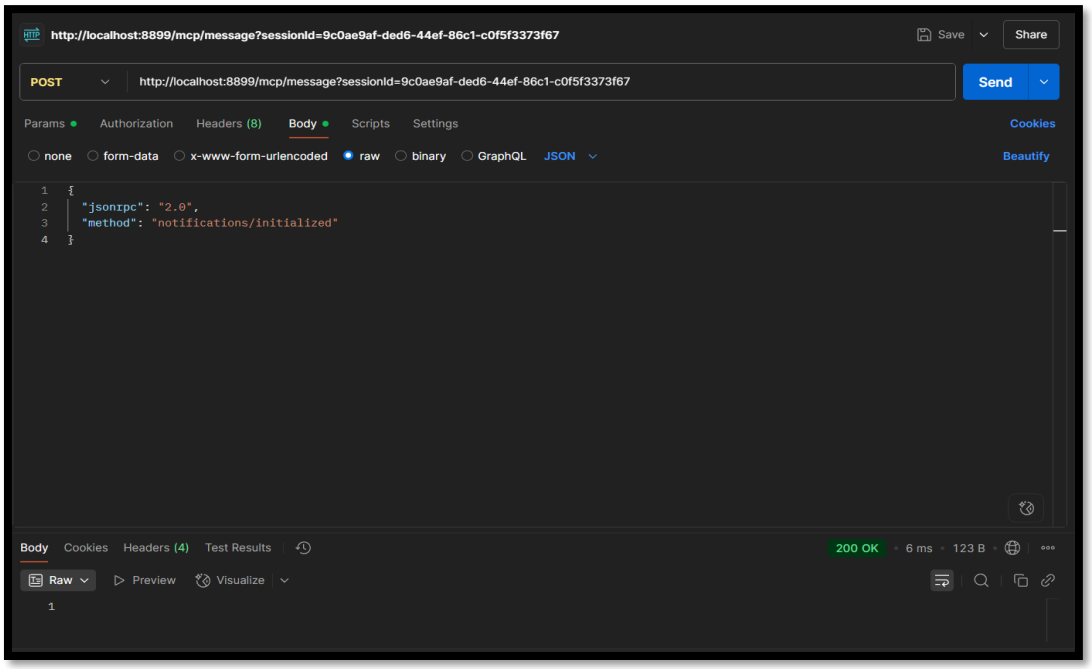

Figure 9:Notification d'initialisation terminée

Requête POST : Liste des outils disponibles

URL : `http://localhost:8899/mcp/message?sessionId=9c0ae9af-ded6-44ef-86c1-c0f5f3373f67`

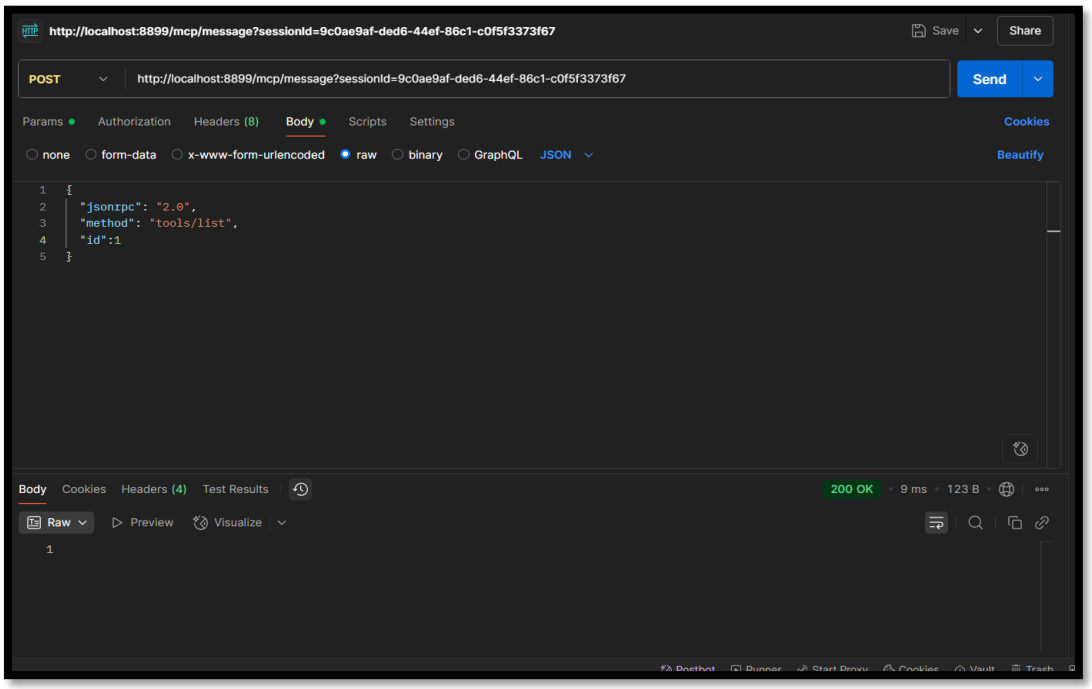

Figure 10:Demande la liste des outils disponibles

Cette requête demande la liste complète des outils (méthodes exposées) que le serveur MCP met à disposition du client. Cela permet au client de connaître les fonctionnalités accessibles.

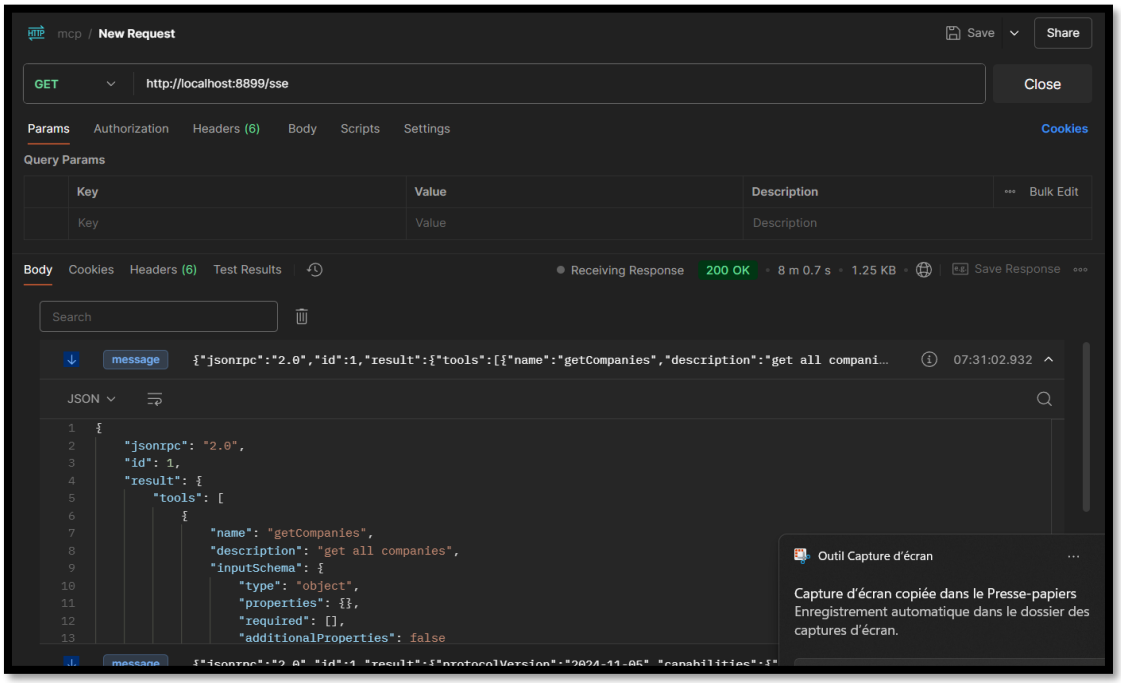

Figure 11:Liste des outils disponibles

Requête POST : Appel d'une méthode spécifique

URL : `http://localhost:8899/mcp/message?sessionId=9c0ae9af-ded6-44ef-86c1-c0f5f3373f67`

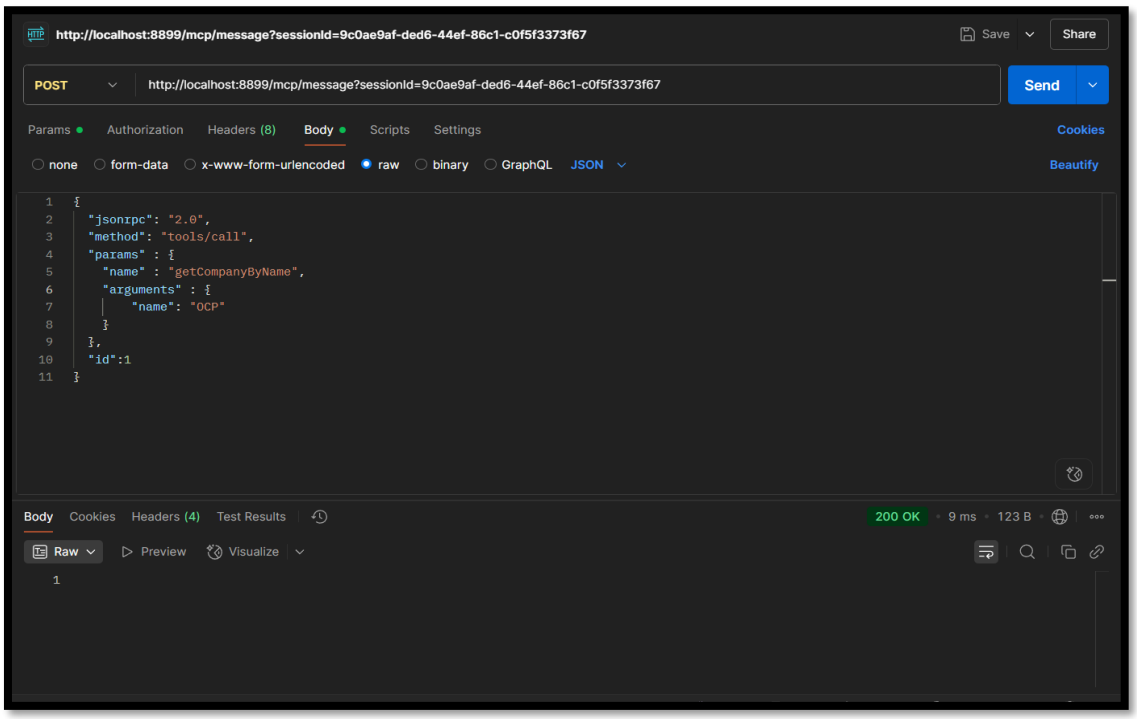

Figure 12:Appel d'une méthode spécifique

Cette requête permet d'appeler la méthode `getCompanyByName` sur le serveur MCP, en lui passant en paramètre le nom de la société recherchée, ici `"OCP"`. Le serveur traite cette requête et renvoie les informations correspondantes.

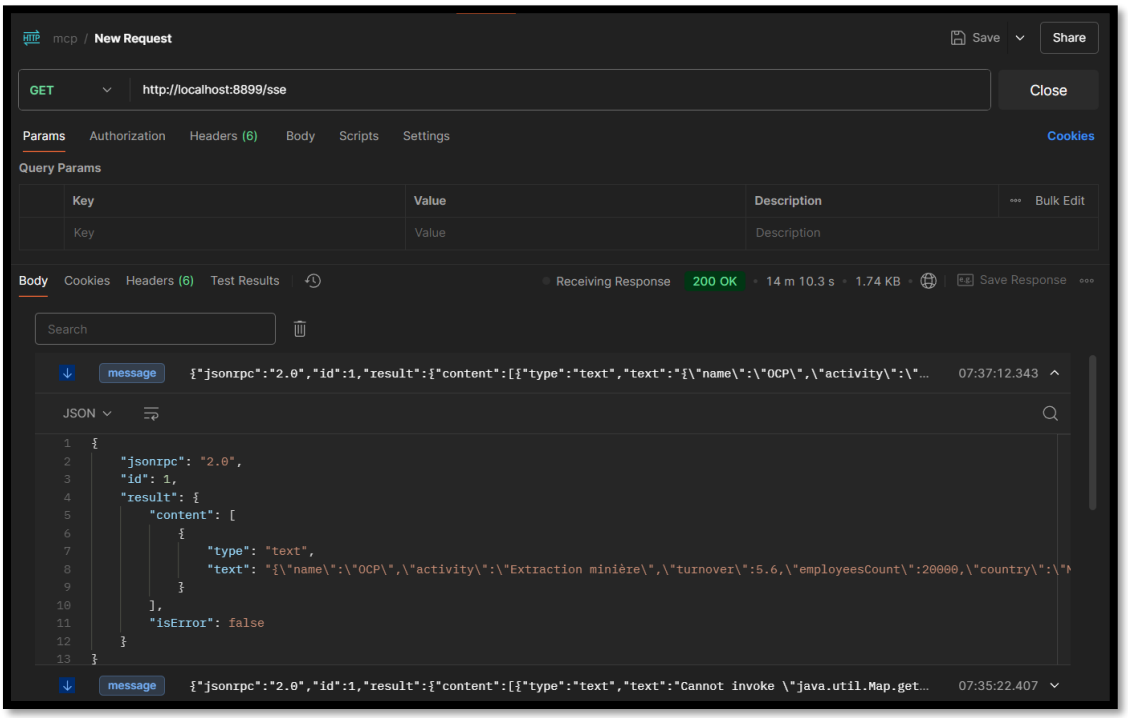

Figure 13:Résultat d'appel d'une méthode spécifique

IV. Création du client MCP

1. Présentation générale

Le client MCP constitue la couche intermédiaire entre le frontend Angular et les différents serveurs MCP (Spring AI, NodeJS, Python). Il est développé avec Spring AI et gère les communications en mode synchronisé (`sync`), en combinant des connexions SSE et STDIO. Le client centralise les interactions, permettant de simplifier la communication avec les multiples serveurs MCP.

2. Architecture et fonctionnement

Le client MCP est organisé autour de plusieurs composants clés :

- **Agent AI (`AIAgent`)** :
  Ce service encapsule un `ChatClient` configuré avec les callbacks des outils MCP disponibles, ainsi qu'une mémoire contextuelle limitée à 20 messages, facilitant des conversations cohérentes et contextualisées.
  La méthode `askLLM` permet d'envoyer une requête utilisateur au client et de récupérer la réponse textuelle générée.
- **Contrôleur REST (`AIRestController`)** :
  Expose une API REST permettant au frontend d'envoyer des requêtes au client MCP via une route POST `/chat`, qui délègue la requête à l'`AIAgent` pour traitement.
- **Classe principale (`McpClientApplication`)** :
  Initialise l'application Spring Boot et, via un `CommandLineRunner`, effectue des appels de test au serveur MCP principal :
  - Liste les outils disponibles via `listTools()`.
  - Appelle un outil spécifique (`getCompanyByName`) en lui passant un paramètre JSON.
  - Affiche la réponse reçue dans la console.

3. Configuration

- Dans le fichier `application.properties` :
  - Le client MCP est configuré en mode `sync`.
  - Une connexion SSE est établie avec le serveur MCP Spring AI (`http://localhost:8899/sse`).
  - Le port d'écoute du client est fixé à 8066.
  - La clé API pour Anthropic Claude est renseignée pour l'utilisation de modèles LLM.
  - La configuration des serveurs STDIO est externalisée dans `mcp-servers.json`.
- Le fichier `mcp-servers.json` décrit la configuration pour lancer un serveur NodeJS MCP en mode STDIO, avec la commande et les arguments nécessaires pour démarrer le serveur depuis un package npm (`@modelcontextprotocol/server-filesystem`).

4. Fonctionnalités

- ✓ Le client MCP établit et maintient une connexion SSE avec le serveur principal pour recevoir en continu les événements et notifications.
- ✓ Il peut lancer et communiquer avec des serveurs MCP externes en mode STDIO (ici NodeJS via le package NPM) et gérer leurs échanges JSON-RPC.
- ✓ Le client offre une API REST simple pour le frontend, facilitant l'intégration des interactions chatbot dans l'application utilisateur.
- ✓ Il utilise une mémoire conversationnelle contextuelle afin d'améliorer la cohérence des réponses générées par le LLM.

5. Présentation du code

Cette capture montre les principales dépendances utilisées dans le projet client MCP. Elles incluent les bibliothèques Spring Boot, Spring AI, la gestion des communications MCP en mode synchrone

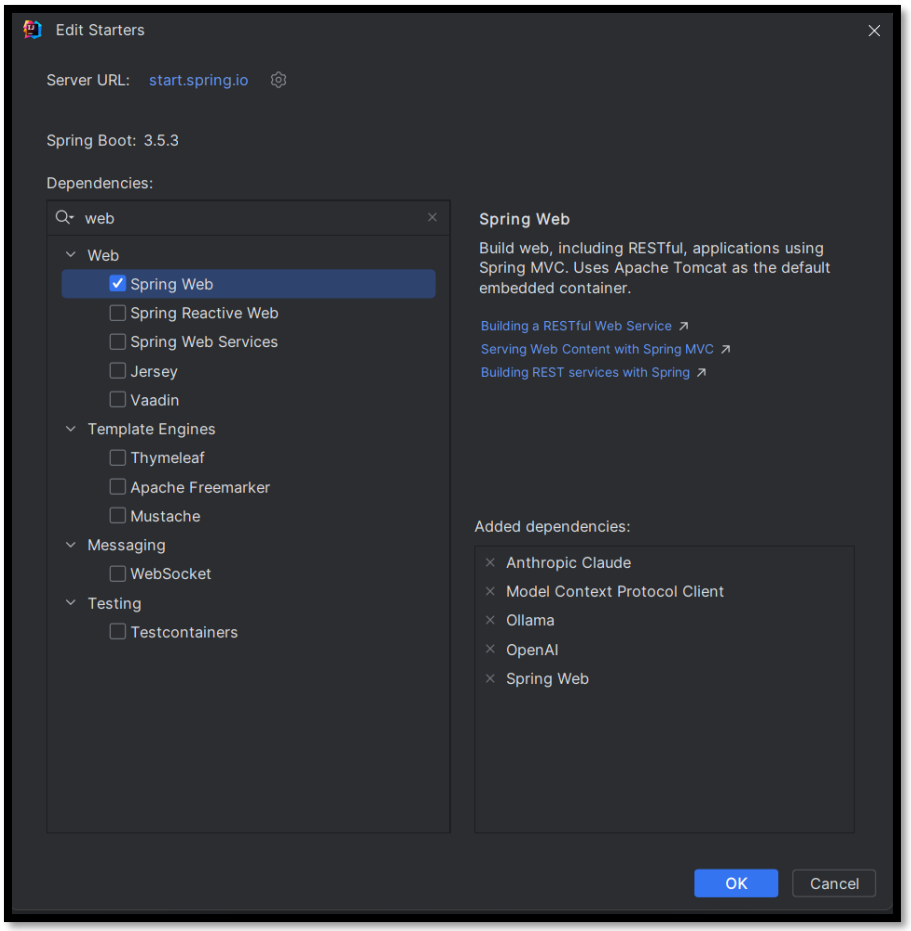

Figure 14:Dépendances du client MCP

Code AIAgent

Le service `AIAgent` est responsable de l'interaction avec le moteur de chat via Spring AI. Il configure un client de chat avec les outils MCP disponibles et une mémoire contextuelle limitée, et fournit une méthode `askLLM` pour envoyer une requête utilisateur et recevoir une réponse.

```
@Service
public class AIAgent {
private ChatClient chatClient;

    public AIAgent(ChatClient.Builder chatClient,
ToolCallbackProvider toolCallbackProvider) {
        this.chatClient = chatClient
                .defaultToolCallbacks(toolCallbackProvider)
                .defaultSystem("Answer the user question using
provided tools")
                .defaultAdvisors(MessageChatMemoryAdvisor.builder(

MessageWindowChatMemory.builder().maxMessages(20).build()
                        ).build())
                .build();
    }

    public String askLLM(String query) {
        return chatClient.prompt()
                .user(query)
                .call()
                .content();
    }
}
```

Code McpClientApplication

Cette classe principale initialise l'application Spring Boot et, via un `CommandLineRunner`, teste la connexion avec le serveur MCP en listant les outils disponibles et en appelant la méthode `getCompanyByName` avec un paramètre d'exemple, affichant les résultats dans la console.

```
@SpringBootApplication
public class McpClientApplication {

    public static void main(String[] args) {
        SpringApplication.run(McpClientApplication.class, args);
    }

    @Bean
    public CommandLineRunner commandLineRunner(List<McpSyncClient>
clients) {
        return args -> {
            clients.forEach(client -> {
                client.listTools().tools().forEach(tool -> {
                    System.out.println("**********");
                    System.out.println(tool.name());
                    System.out.println(tool.description());
                    System.out.println(tool.inputSchema());
                    System.out.println("**********");
                });
                System.out.println("**********");
                String params = """
                {
                    "name": "OCP"
                }
```

```
                """;

                McpSchema.CallToolResult result =
                        clients.get(0).callTool(new
McpSchema.CallToolRequest("getCompanyByName", params));
                McpSchema.Content content =
result.content().get(0);
                System.out.println(content);
                if (content.type().equals("text")){
                    System.out.println("**********");
                    McpSchema.TextContent textContent =
(McpSchema.TextContent) content;
                    System.out.println(textContent.text());
                }

                System.out.println(content);
            });
        };
    }
}
```

Contrôleur REST AIRestController

Ce contrôleur expose un endpoint POST `/chat` qui permet au frontend d'envoyer des messages au client MCP. Il délègue la gestion des requêtes à l'`AIAgent` et retourne les réponses générées.

```
@RestController
@CrossOrigin("*")
public class AIRestController {
    private AIAgent agent;

    public AIRestController(AIAgent agent) {
        this.agent = agent;
    }

    @PostMapping("/chat")
    @ResponseBody
    public String chat(@RequestParam String query) {
        return agent.askLLM(query);
    }

}
```

Fichier application.properties

Le fichier de configuration définit le mode de fonctionnement du client MCP (synchronisé), configure la connexion SSE vers le serveur MCP principal, le port d'écoute du client, ainsi que la clé API et les paramètres pour l'utilisation du modèle Anthropic Claude. Il référence également le fichier de configuration des serveurs STDIO.

```
spring.application.name=mcp-client

spring.ai.mcp.client.type=sync
spring.ai.mcp.client.sse.connections.server1.url=http://localhost:88
```

```
99
spring.ai.mcp.client.sse.connections.server1.sse-endpoint=/sse
server.port=8066

spring.ai.anthropic.api-key= ${key}
spring.ai.anthropic.chat.options.model=claude-sonnet-4-20250514

spring.ai.mcp.client.stdio.servers-configuration=classpath:mcp-
servers.json
```

Fichier mcp-servers.json

Ce fichier JSON configure le lancement automatique d'un serveur NodeJS MCP en mode STDIO via la commande `npx`. Il indique les arguments nécessaires pour exécuter le package `@modelcontextprotocol/server-filesystem` et définir le chemin de travail.

```
{
  "mcpServers": {
    "filesystem": {
      "command": "C:\\Program Files\\nodejs\\npx.cmd",
      "args": [
        "-y",
        "@modelcontextprotocol/server-filesystem",
        "D:\\Bureaux\\IGA\\5GI\\Interaction avancée\\Tp final\\mcp-
demo-spring-python"
      ]
    }
  }
}
```

6. Tests réalisés avec Swagger UI

Test /chat avec message "liste de entreprises" : Le client MCP répond en listant les entreprises disponibles via les outils MCP exposés.

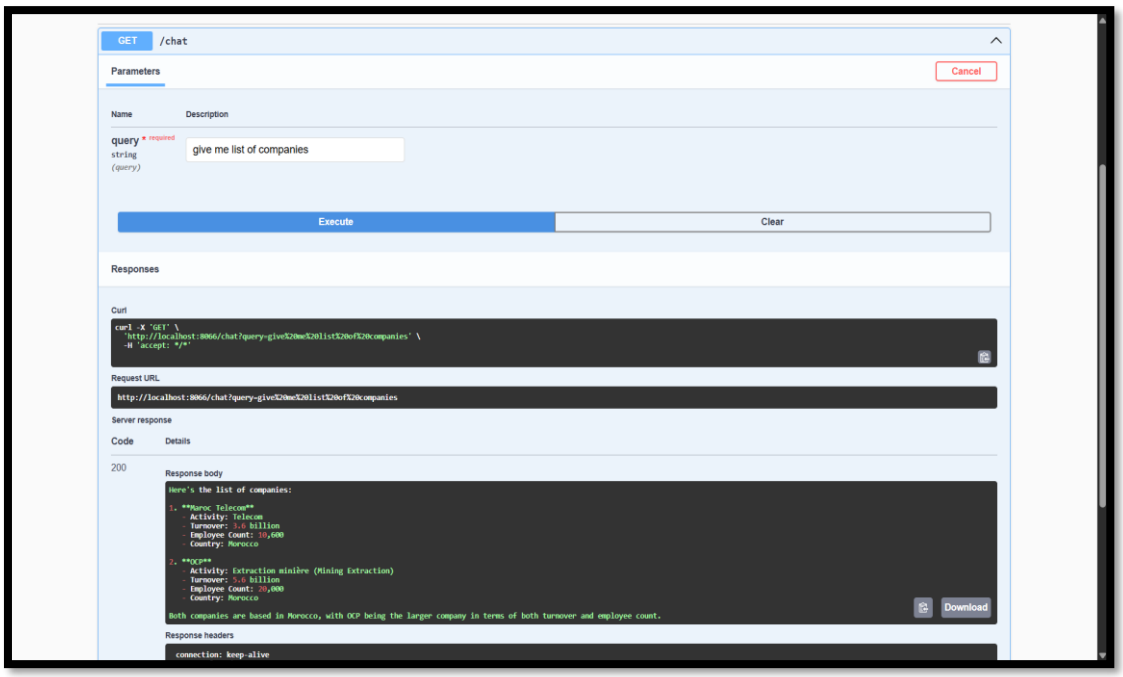

Figure 15:Test /chat avec message "liste de entreprises"

Test du serveur NodeJS via demande "nombre de fichiers" :Le client interroge le serveur NodeJS et retourne le nombre de fichiers, validant la communication STDIO.

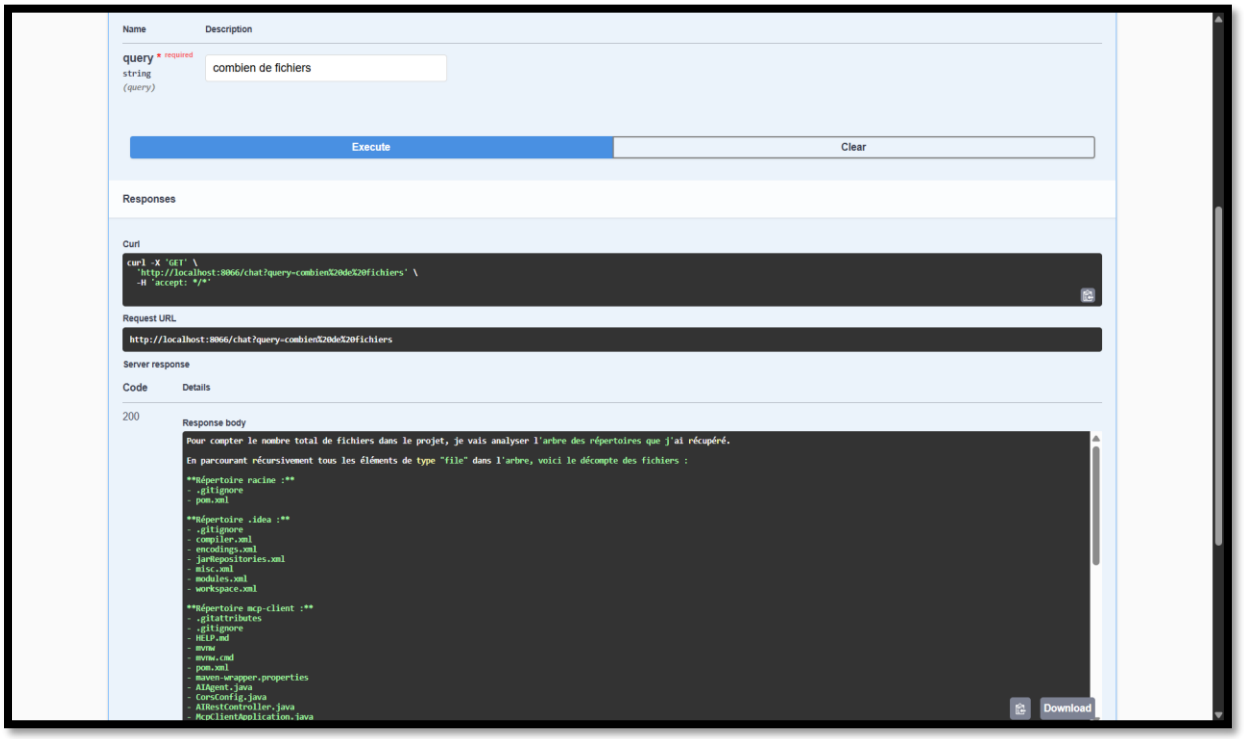

Figure 16:Test du serveur NodeJS via demande "nombre de fichiers"

Test de la mémoire conversationnelle : Après avoir fourni un nom personnel, une requête ultérieure confirme que le client se souvient et rappelle ce nom, démontrant le bon fonctionnement de la mémoire contextuelle.

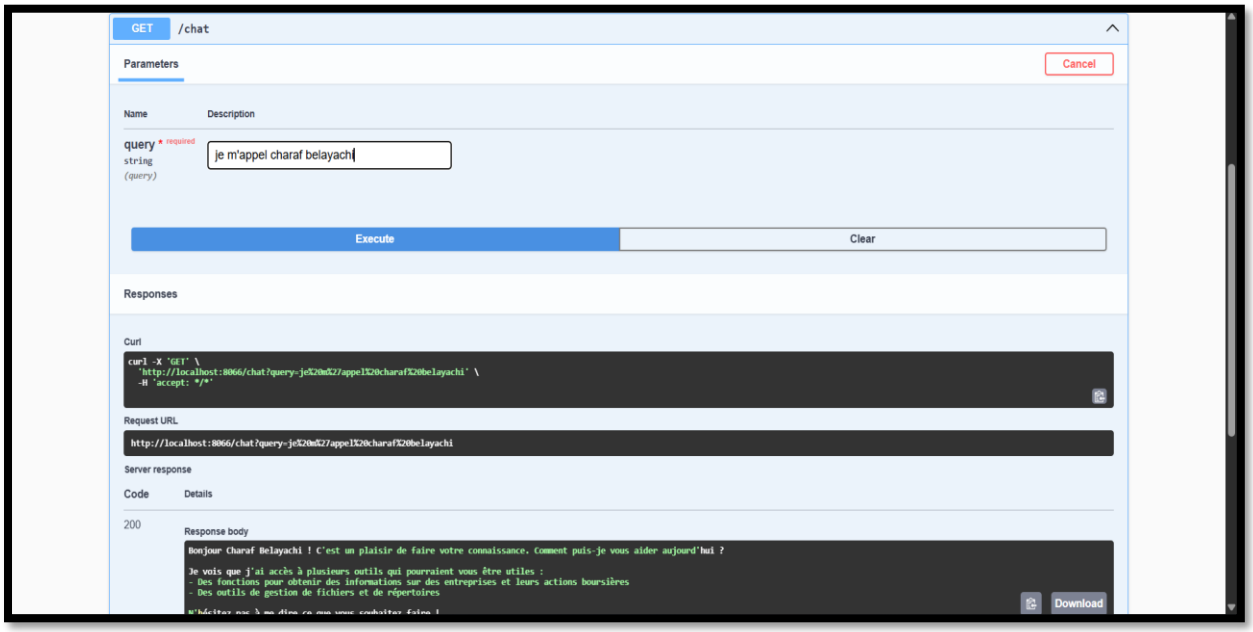

Figure 17:Test de la mémoire conversationnelle

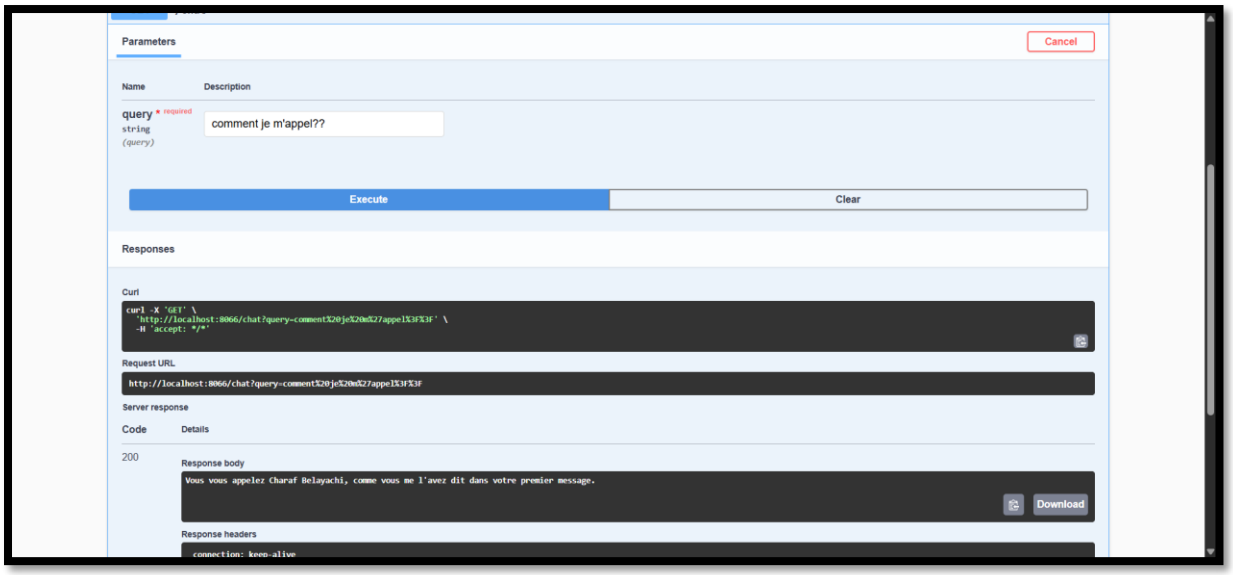

Figure 18:Résultat de test de la mémoire conversationnelle

V. Création du frontend Angular

1. Présentation générale

Le frontend de l'application est développé avec Angular, utilisant une architecture moderne basée sur des composants standalone. Il fournit une interface utilisateur intuitive pour interagir avec le chatbot via une API REST exposée par le client MCP.

2. Architecture et composants clés

- ✓ **Service `ChatServiceService`**
  Ce service gère la communication avec le backend. Il envoie les messages utilisateur à l'endpoint `/chat` via une requête HTTP POST, en utilisant un `FormData` contenant la requête. La réponse, de type texte, est retournée sous forme d'`Observable` pour faciliter la gestion asynchrone dans les composants.
- ✓ **Composant `ChatComponent`**
  Ce composant affiche l'interface de chat :
  - Une zone d'affichage des messages (utilisateur et assistant) avec un style différencié selon l'auteur.
  - Un indicateur de saisie ("AI is typing...") lorsque la réponse est en cours.
  - Un champ de texte pour saisir les messages, avec gestion du formulaire et soumission via bouton ou touche Entrée.
  - Une mise en forme soignée avec Bootstrap pour un rendu moderne et réactif.
- ✓ **Composant racine `AppComponent`**
  Utilise Angular Standalone Components pour intégrer le composant `ChatComponent` dans l'application principale. Ce composant sert de point d'entrée et de conteneur global.

3. Fonctionnement

L'utilisateur saisit un message dans le champ prévu, qui est envoyé via le service `ChatServiceService` au backend. La réponse du chatbot est affichée dynamiquement dans la zone de dialogue, avec prise en charge de l'affichage formaté (support Markdown possible). L'indicateur de saisie signale les temps d'attente.

4. Technologies et styles

Le frontend utilise :

- ✓ Angular Standalone Components pour modularité et simplicité.
- ✓ Bootstrap 5 pour la mise en page et les styles.
- ✓ RxJS pour la gestion asynchrone des appels HTTP.
- ✓ Icônes Bootstrap Icons pour enrichir l'interface utilisateur.

5. Tests et validation

Le frontend a été testé manuellement pour vérifier la fluidité des échanges, la gestion correcte des états (chargement, erreurs) et la présentation des messages. L'intégration avec le backend MCP via API REST a été validée.

Description du fichier chat-component.component.ts :

Ce composant Angular gère l'interface utilisateur du chatbot.

- Il affiche la liste des messages échangés entre l'utilisateur et l'assistant IA, avec une mise en forme différenciée selon l'émetteur.
- Il gère l'envoi des messages via un formulaire réactif, prenant en charge la saisie clavier et le clic sur le bouton d'envoi.
- Il affiche un indicateur visuel lorsque l'assistant est en train de générer une réponse.
- Le composant utilise Bootstrap pour le style et assure une bonne ergonomie mobile et desktop.

```
import { Component } from '@angular/core';

import {CommonModule, NgForOf} from '@angular/common';
import {FormsModule} from '@angular/forms';
import {ChatMessage, ChatServiceService} from '../../services/chat-
service.service';
import {marked} from 'marked';

@Component({
  selector: 'app-chat-component',
  standalone: true,
  imports: [
    NgForOf,

    CommonModule,
    FormsModule
  ],
  templateUrl: './chat-component.component.html',
  styleUrl: './chat-component.component.css'
})
export class ChatComponentComponent {
  messages: ChatMessage[] = [];
  newMessage: string = '';
  isLoading: boolean = false;
  isTyping: boolean = false;

  constructor(private chatService: ChatServiceService) {
    this.messages.push({
      content: 'Hello! I\'m your AI assistant. How can I help you
today?',
      isUser: false,
      timestamp: new Date()
    });
  }

  sendMessage(): void {
    if (!this.newMessage.trim() || this.isLoading) {
      return;
    }

    const userMessage: ChatMessage = {
      content: this.newMessage,
      isUser: true,
      timestamp: new Date()
    };
```

```
    this.messages.push(userMessage);
    const messageToSend = this.newMessage;
    this.newMessage = '';
    this.isLoading = true;
    this.isTyping = true;

    this.chatService.sendMessage(messageToSend).subscribe({
      next: (response) => {
        this.isTyping = false;
        const aiMessage: ChatMessage = {
          content: response,
          isUser: false,
          timestamp: new Date()
        };
        this.messages.push(aiMessage);
        this.isLoading = false;
      },
      error: (error) => {
        this.isTyping = false;
        console.error('Error:', error);
        const errorMessage: ChatMessage = {
          content: 'Sorry, there was an error processing your
request. Please try again.',
          isUser: false,
          timestamp: new Date()
        };
        this.messages.push(errorMessage);
        this.isLoading = false;
      }
    });
  }

  onKeyDown(event: KeyboardEvent): void {
    if (event.key === 'Enter' && !event.shiftKey) {
      event.preventDefault();
      this.sendMessage();
    }
  }

  parseMarkdown(content: string): string {
    return marked.parse(content) as string;
  }
}
```

Description du fichier chat-service.service.ts

Ce service Angular est responsable des communications HTTP avec le backend.

- Il fournit une méthode `sendMessage` qui envoie la requête utilisateur à l'endpoint `/chat` sous forme de `FormData`.
- La réponse du serveur est retournée sous forme d'`Observable<string>`, facilitant la gestion asynchrone et la mise à jour dynamique de l'interface.
- Ce service centralise la logique d'appel HTTP, favorisant la réutilisabilité et la séparation des responsabilités.

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { HttpClient } from '@angular/common/http';

export interface ChatMessage {
  content: string;
  isUser: boolean;
  timestamp: Date;
}

@Injectable({
  providedIn: 'root'
})
export class ChatServiceService {
  private apiUrl = 'http://localhost:8066';

  constructor(private http: HttpClient) { }

  sendMessage(message: string): Observable<string> {
    const formData = new FormData();
    formData.append('query', message);

    return this.http.post(`${this.apiUrl}/chat`, formData, {
      responseType: 'text'
    });
  }
}
```

Description du fichier app.component.ts

Le composant racine de l'application Angular, qui :

- Utilise les **Standalone Components** pour importer directement le composant de chat.
- Sert de conteneur principal sans logique métier particulière, facilitant l'intégration et la modularité de l'application.

```
import { Component } from '@angular/core';
import { RouterOutlet } from '@angular/router';
import {ChatComponentComponent} from "./components/chat-
component/chat-component.component";

@Component({
  selector: 'app-root',
  standalone: true,
  imports: [RouterOutlet, ChatComponentComponent],
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
export class AppComponent {
  title='chat-ui';
}
```

Capture d'écran de la boîte de chat (AI Assistant)

Cette capture montre l'interface utilisateur au démarrage, avec :

- Le titre "AI Assistant" dans un bandeau supérieur.

- Le message d'accueil de l'assistant pour inviter l'utilisateur à interagir.
- Le champ de saisie prêt à recevoir la première requête.
- Un bouton d'envoi clair et accessible.

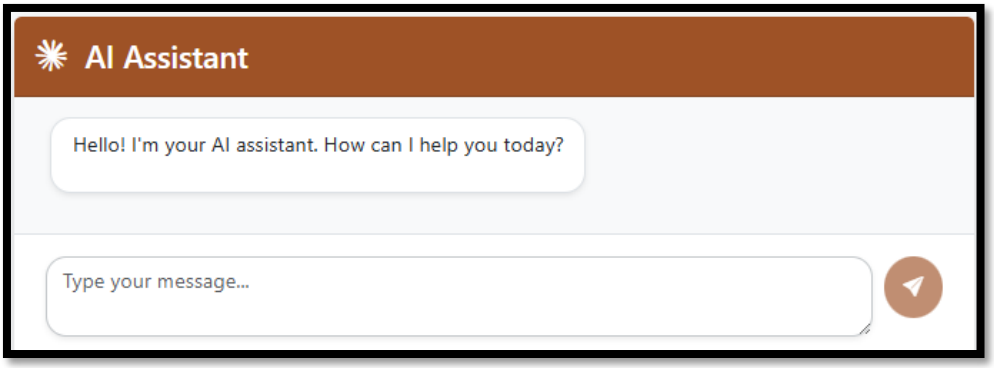

Figure 19:Capture d'écran de la boîte de chat (AI Assistant)

Capture d'écran de la réponse à la question "liste de entreprises"

Cette image illustre la réponse générée par le chatbot lorsqu'on lui demande la liste des entreprises disponibles.

- La réponse est affichée dans la zone de dialogue avec une mise en forme claire.
- Elle montre la bonne communication entre le frontend et le backend MCP, ainsi que la capacité du chatbot à traiter et retourner des données métier.

Figure 20:Capture d'écran de la réponse à la question "liste de entreprises"

Capture d’écran du test mémoire (reconnaissance du nom)

Cette capture illustre un test de mémoire conversationnelle :

- Après avoir fourni un nom personnel dans la discussion, une requête ultérieure montre que le chatbot se souvient et rappelle correctement ce nom.
- Cela démontre la fonctionnalité de mémoire contextuelle implémentée dans le client MCP, améliorant la cohérence des dialogues.

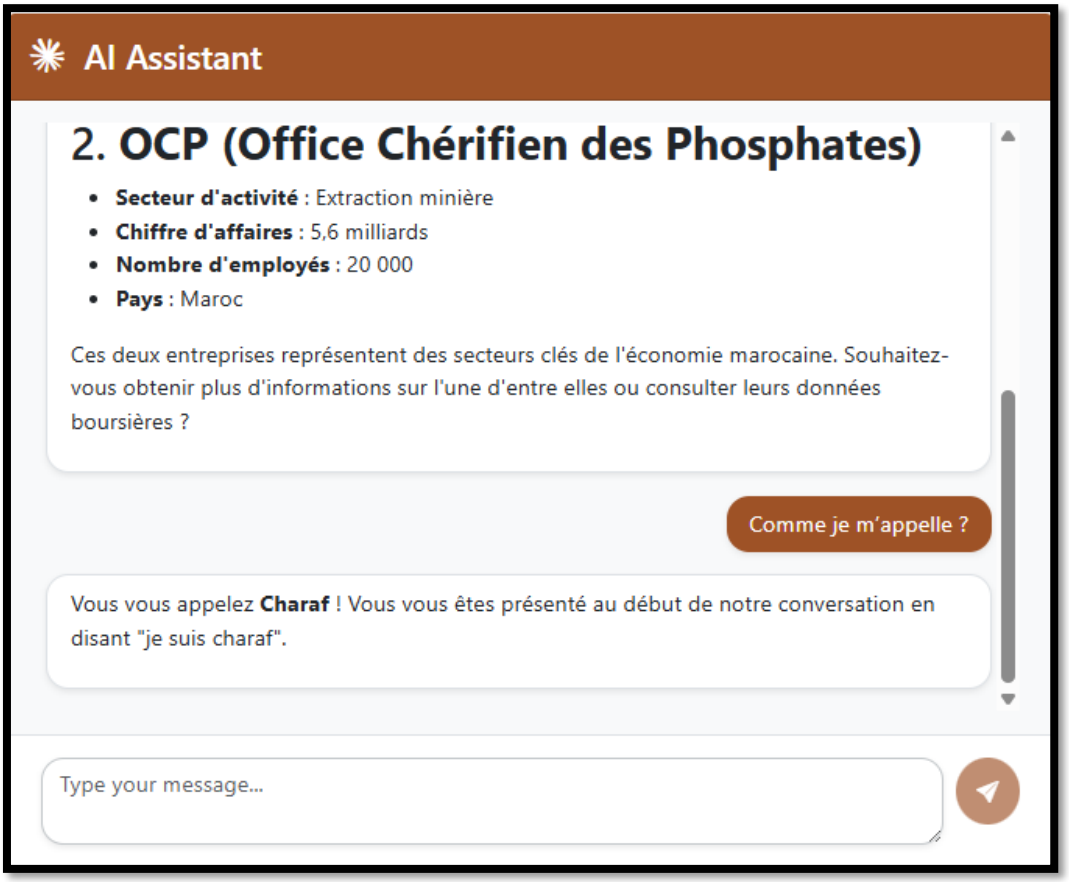

Figure 21:Capture d’écran du test mémoire (reconnaissance du nom)

VI. Conclusion générale

Ce projet a permis de concevoir et de développer une application chatbot basée sur le protocole MCP, intégrant plusieurs serveurs MCP développés en Spring AI, NodeJS et Python, ainsi qu'un client centralisé et un frontend Angular moderne. Cette architecture distribuée assure une communication fluide et en temps réel entre les différents composants, offrant une expérience utilisateur efficace et modulable.

La mise en œuvre du serveur MCP avec Spring AI a démontré la simplicité d'exposer des outils métiers via des annotations spécifiques, facilitant ainsi l'intégration avec les clients. Le client MCP a su centraliser et orchestrer les échanges entre plusieurs serveurs, tout en proposant une interface REST exploitable par le frontend Angular. Ce dernier a offert une interface claire, intuitive et réactive, favorisant une interaction naturelle avec le chatbot.

1. Difficultés rencontrées

Un des principaux obstacles rencontrés concernait le choix d'un modèle d'intelligence artificielle compatible avec l'intégration des outils MCP. En effet, la plupart des modèles testés prenaient un temps d'exécution trop long, impactant négativement la réactivité de l'application. Cette contrainte a nécessité une sélection rigoureuse des modèles et l'optimisation des échanges pour garantir un bon compromis entre performances et fonctionnalités.

2. Perspectives d'évolution

Plusieurs axes d'amélioration peuvent être envisagés pour enrichir et pérenniser ce projet :

- Intégration de modèles d'IA plus performants et adaptés à la gestion dynamique des outils MCP, avec un meilleur équilibre entre rapidité et qualité des réponses.
- Extension du système avec de nouveaux serveurs MCP spécialisés pour couvrir davantage de cas d'usage.
- Amélioration du frontend Angular avec des fonctionnalités avancées telles que la gestion multi-utilisateur, l'historique des conversations, et une interface plus riche (chat vocal, support multimédia).
- Mise en place de mécanismes de sécurité renforcés pour protéger les échanges et les données utilisateurs.
- Automatisation des tests et déploiements pour garantir la robustesse et la maintenance continue de l'application.

Ce projet a ainsi posé des bases solides, ouvrant la voie à un développement évolutif en fonction des besoins et des innovations technologiques à venir.