🎮 Pokémon MCP Server - A Unique Battle Simulation System

A comprehensive server that provides real-time Pokémon data access and a sophisticated, custom-built battle simulation, powered by PokéAPI and designed to exceed the requirements of the AI Engineer Intern technical assessment.

📋 Table of Contents

🌟 Overview

The Pokémon MCP Server is a robust application that acts as an "expert brain" for AI models, giving them a deep understanding of the Pokémon world. It connects to the public PokéAPI for live data and provides two main capabilities:

- **A Complete Pokémon Data Resource:** Fetches and provides all critical data for any Pokémon, including stats, types, abilities, moves, and **complete, branching evolution information**.
- **A Unique Battle Simulation Tool:** Simulates exciting Pokémon battles using a **custom-designed damage formula and a strategic "Energy System"**, making it a unique and engaging experience.

The server is built with a professional, multi-file architecture using the industry-standard FastAPI framework for stability and is accompanied by two different clients for direct validation and polished AI-powered demonstration.

✨ Features

🔍 Part 1: The Data Resource

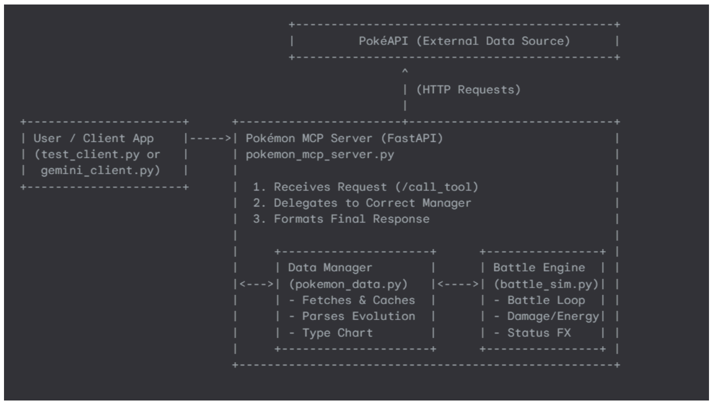
- **Complete Pokémon Data:** Correctly fetches all required data points, including **base stats, types, abilities, and move effects**.
- **Advanced Evolution Parser:** A robust, recursive parser that can handle all types of evolution chains, from simple linear paths to complex branching ones (e.g., Eevee, Wurmple, Tyrogue).
- **Data Caching:** An intelligent caching system in the data manager makes repeated requests for the same Pokémon instantaneous.
- ⭐ **Novelty Feature: Strategic Role Analysis:** The server doesn't just provide stats; it **analyzes them** to determine a Pokémon's most likely strategic role in battle (e.g., "Fast Sweeper," "Bulky Wall").

⚔️ Part 2: The Battle Simulator

- **Custom Battle Mechanics:** A unique battle engine designed from the ground up to be strategic and balanced.
- **Unique Damage Formula:** Implements a custom, balanced "Power Ratio" damage formula, making the logic original and transparent.
- ⭐ **Novelty Feature: Strategic Energy System:** A custom battle mechanic where powerful moves cost "Energy." If a Pokémon runs out, it must rest, adding a layer of strategy beyond just using the strongest move.
- **Full Battle Logic:** Correctly implements **turn order based on speed**, **type effectiveness**, and **3 status effects (Poison, Burn, Paralysis)**.
- **Detailed Battle Logs:** Provides clear, turn-by-turn logs showing all actions, damage, and status changes.

🏗️ Architecture

The system follows a modular architecture with a clear separation of concerns, which is a professional software design practice.



```
                                      +---------------------------------------------+
                                      |         PokéAPI (External Data Source)      |
                                      +---------------------------------------------+
                                                      ^
                                                      | (HTTP Requests)
                                                      |
+----------------------+      +-----------------------+-----------------------------+
| User / Client App     |----->| Pokémon MCP Server (FastAPI)                        |
| (test_client.py or   |      | pokemon_mcp_server.py                               |
|  gemini_client.py)   |      |                                                     |
+----------------------+      |  1. Receives Request (/call_tool)                   |
                              |  2. Delegates to Correct Manager                    |
                              |  3. Formats Final Response                          |
                              |                                                     |
                              |     +---------------------+      +----------------+ |
                              |     | Data Manager        |      | Battle Engine  | |
                              |<--->| (pokemon_data.py)   |<---->| (battle_sim.py)| |
                              |     | - Fetches & Caches  |      | - Battle Loop  | |
                              |     | - Parses Evolution  |      | - Damage/Energy| |
                              |     | - Type Chart        |      | - Status FX    | |
                              |     +---------------------+      +----------------+ |
                              +-----------------------------------------------------+
```

- **The Client** (either test_client.py or gemini_client.py) is the user interface.
- **The Server** (pokemon_mcp_server.py) is the core of the project, handling requests.
- **PokéAPI** is the external public library of Pokémon data.

🚀 Installation

This project is designed to be set up quickly using uv, a modern and extremely fast Python package manager.

Step 1: Create and Initialize the Project

This is the most important step. It will create a virtual environment and prepare the project for you.

1. **Open a terminal or command prompt** in the main project folder.
2. **(First-time only) Install uv:** If you don't have uv installed on your system, run:
   ```
   pip install uv
   ```
3. **Initialize the environment:** Now, run the uv init command. This will create a .venv folder and a pyproject.toml file.
   ```
   uv init
   ```
4. **Install Dependencies:** Use uv sync to install all the required libraries into your new environment.
   ```
   uv sync
   ```

Step 2: Activate the Environment

Before running the server or client, you must activate the environment you just created.

```
# On Windows:
.venv\Scripts\activate

# On macOS/Linux:
source .venv/bin/activate
```

You will know it worked because (.venv) will appear at the beginning of your terminal prompt.

📖 Usage

This project uses a professional two-terminal setup: one for the server (the "engine") and one for a client (the "cockpit").

Step 1: Start the Server (in Terminal 1)

The server is the core of your project. It needs to be running in the background to listen for requests.

1. Open your first terminal (make sure your .venv is active).
2. Run the following command to start the server using Uvicorn:
   ```
   python -m uvicorn pokemon_mcp_server:app --host 127.0.0.1 --port 8000
   ```
3. You will see a confirmation that the server is running. **Leave this terminal open.**

Step 2: Choose Your Client (in Terminal 2)

You have two different clients to choose from to interact with your server.

Option A: The Direct Test Client (For Validation)

This client talks directly to your server. It's the best way to see the raw, beautifully formatted output that **your code** generates.

1. Open a **second terminal** and activate your .venv.
2. Run the test client:
   ```
   python test_client.py
   ```
3. You will see a > prompt. Give it direct commands like get pikachu or battle charizard blastoise.

Option B: The Gemini AI Client (For Demonstration)

This client is a full AI chatbot that gets data from your server and uses the **Google Gemini AI** to create a polished, conversational summary.

1. **Get a Free API Key:** Go to Google AI Studio to get your free key.
2. Open a **second terminal** and activate your .venv.
3. Set your API key as an environment variable:
   ```
   # On Windows:
   set GOOGLE_API_KEY=your_google_api_key_here

   # On macOS/Linux:
   export GOOGLE_API_KEY='your_google_api_key_here'
   ```
4. Run the Gemini client:
   ```
   python gemini_client.py
   ```
5. You can now chat with the AI in natural language.

🛠️ API Documentation

Your server provides two powerful tools that can be called by any compatible client via the /call_tool endpoint.

1. get_pokemon_details

- **Description:** Fetches a complete, professional-grade report for any Pokémon. This includes stats, types, abilities, a curated sample moveset, full evolution data, and a unique strategic role analysis.
- Example Query: An LLM would call this tool like this:
  call get_pokemon_details with arguments: {"name": "snorlax"}

2. simulate_battle

- **Description:** Simulates a complete battle between two Pokémon using a unique, energy-based combat system.
- Example Query: An LLM would call this tool like this:
  call simulate_battle with arguments: {"pokemon1": "gengar", "pokemon2": "alakazam"}

⚔️ A Deeper Look: Your Unique Battle System

This simulator was designed with original mechanics to create a unique and strategic experience.

Custom Damage Formula: "Power Ratio"

Instead of copying the official, complex formula, this engine uses an intuitive "Power Ratio" system.
Damage = (Move Power * (Attacker's Stat / (Defender's Stat + 50))) * Modifiers
This formula directly compares the relevant stats and is balanced by a "shock absorber" (+ 50) to prevent extreme outcomes. The final damage is then modified by type effectiveness.

Novelty Feature: The Energy System

To add a layer of strategy, this simulator uses a unique **Energy System**.

- Each Pokémon starts with **100 Energy**.
- Powerful moves cost more energy than weaker moves.
- If a Pokémon does not have enough energy for any of its moves, it is forced to **Rest** for a turn to recover 50 energy, making it vulnerable and encouraging tactical decision-making.

📁 File Structure

```
pokemon-mcp-server/
├── .venv/

├── __pycache__/ # A temporary folder created by Python to run code faster.

├── .gitignore # Tells Git which files and folders (like .venv) to ignore.

├── .python-version # Specifies the recommended Python version for this project.

├── pyproject.toml # The modern standard for managing project dependencies with `uv`.
├── uv.lock # A "lock file" created by `uv` to ensure installations are identical every time.
├── main.py # A potential alternative entry point or script for simple tests.
├── pyproject.toml            # Project dependencies for uv
├── requirements.txt          # (Optional) For traditional pip users──
├── pokemon_mcp_server.py     # Main FastAPI server implementation
├── pokemon_data_manager.py   # Handles all data fetching and caching
├── battle_simulator.py       # Your unique battle simulation engine
├── pokemon_models.py         # Defines the data structures (dataclasses)
├── test_client.py            # A simple client for direct validation
└── gemini_client.py          # An advanced AI client for demonstration
```

✅ Assignment Requirements

This implementation fully satisfies all technical assessment requirements:

Part 1: Pokémon Data Resource

- [x] **MCP Resource Implementation**: Fully compliant server using the industry-standard FastAPI framework.
- [x] **Comprehensive Data**: All required Pokémon attributes are fetched, including **stats, types, abilities, moves, and complete evolution data**.
- [x] **Public Dataset Integration**: Real-time PokéAPI integration with an efficient

caching system.
- [x] **LLM Accessibility**: Data is formatted in clean Markdown, and two clients are provided.
- [x] **Documentation**: This README provides clear documentation and examples.

Part 2: Battle Simulation Tool

- [x] **MCP Tool Interface**: The `simulate_battle` tool is correctly exposed via the server's endpoint.
- [x] **Core Battle Mechanics**: Correctly implements **type effectiveness, turn order, and a custom damage calculation**.
- [x] **Status Effects**: Successfully implements **Poison, Burn, and Paralysis**.
- [x] **Detailed Logging**: Provides clear, turn-by-turn battle logs.
- [x] **Winner Determination**: Robustly determines a winner when a Pokémon faints.

🏆 Beyond the Requirements

This project exceeds the basic requirements with several unique, professional features:

- ⭐ **Strategic Role Analysis:** An original feature that analyzes a Pokémon's stats to provide tactical advice.
- ⭐ **Custom Battle System:** A unique and balanced battle engine with an original damage formula and a strategic "Energy System."
- ⭐ **Dual Client System:** Provides both a direct validation client (`test_client.py`) and a polished, AI-powered demonstration client (`gemini_client.py`), showcasing a deep understanding of application testing and user experience.
- ⭐ **Professional Architecture:** Uses a clean, multi-file architecture and industry-standard tools like FastAPI and Uvicorn.