Namaa Backend Challenge - Python Code Execution Server

October 27, 2024

Introduction

In this challenge, you are tasked with building an HTTP server that executes Python code provided by the user, with incremental levels of complexity. You may use any language, framework, or libraries to create this server.

You will need to provide **clear instructions** for building and running your project. Please consider any potential differences between Windows and Linux environments in your setup instructions. **If we are unable to build and run your project based on your instructions, we will not consider your application.**

You are encouraged to use any resources at your disposal (documentation, online references, LLMs, etc.), but please avoid direct human assistance.

Note: Solving all levels is not mandatory. Some levels may be challenging and time-intensive, so feel free to submit partial solutions, specifying clearly which parts are complete and which are not.

Level 1: Basic Code Execution

1. **Endpoint**: Implement a POST endpoint at `/execute`.
2. **Request Schema**: The endpoint accepts a JSON payload with the following schema:

   ```
   {
       "code": "string"
   }
   ```

   - **code** (string): A Python code snippet to execute.
3. **Response Schema**: The server responds with a JSON object that conditionally includes the following keys:

```
{
    "stdout": "string",
    "stderr": "string",
    "error": "string"
}
```

Key Rules:

- If `error` is present, both `stdout` and `stderr` must be omitted.
- If either `stdout` or `stderr` (or both) are present, `error` must be omitted.

4. **Error Handling**:
    - For **internal server errors**, respond with HTTP 500 and a JSON object `{ "error": "Internal server error" }`.
    - For **user errors** (e.g., malformed JSON), respond with an appropriate HTTP status code and error message in `error`.

Example 1

Request:

```
POST /execute

{
    "code": "print('Hello, World!')"
}
```

Response:

```
{
    "stdout": "Hello, World!\n"
}
```

Example 2

Request:

```
POST /execute

{
    "code": "1 / 0"
}
```

Response:

```
{
    "stderr": "Traceback (most recent call last):\n  File \"<stdin>\"
    , line 1, in <module>\nZeroDivisionError: division by zero\n"
}
```

Level 2: Resource Limits

Enhance your server to restrict the code execution within the following limits:

1. **Time Limit**: Code execution must terminate after 2 seconds.
2. **Memory Limit**: Code execution must use no more than 100 MB of memory.

If these limits are exceeded, terminate the execution and include an appropriate error message in the `error` key.

Example

Request:

```
POST /execute

{
    "code": "while True: pass"
}
```

Response:

```
{
    "error": "execution timeout"
}
```

Level 3: Persistent Interpreter Sessions

Optimize the server by maintaining persistent Python interpreter sessions.

1. **Interpreter Session**: Instead of starting a new Python process for each request, initialize a persistent interpreter session and assign it a unique identifier (UUID).
2. **Request Schema Update**: Modify the request payload to allow specifying an existing session ID for code continuation. The updated schema is as follows:

   ```
   {
       "id": "string",
       "code": "string"
   }
   ```

   - **id** (string, optional): A unique session ID to continue code execution within an existing interpreter. If not provided, a new interpreter session is created.

- **code** (string): A Python code snippet to execute.

3. **Response Schema Update**: The server should respond with:

```
{
    "id": "string",
    "stdout": "string",
    "stderr": "string",
    "error": "string"
}
```

   - **id** (string): The session ID associated with the interpreter.
   - **stdout** (string, optional): The standard output from the executed code, if any.
   - **stderr** (string, optional): Any error output from the executed code, if any.
   - **error** (string, optional): Any server-side error message encountered while handling the request.

   Key Rules:

   - If `error` is present, both `stdout` and `stderr` must be omitted.
   - If either `stdout` or `stderr` (or both) are present, `error` must be omitted.

4. **Behavior**: When an `id` is provided in the request, execute the code within the specified interpreter session, allowing for state retention between requests. If no `id` is provided, create a new interpreter session.

Example

Request (New Session):

```
POST /execute

{
    "code": "x = 5"
}
```

Response:

```
{
    "id": "0802b8de-03a1-4108-8dd9-9a9025eded65"
}
```

Request (Using Existing Session):

```
POST /execute

{
    "id": "0802b8de-03a1-4108-8dd9-9a9025eded65",
```

```
    "code": "print(x)"
}
```

Response:

```
{
    "id": "0802b8de-03a1-4108-8dd9-9a9025eded65",
    "stdout": "5\n"
}
```

Level 4: Environment Hardening

Further enhance security by hardening the execution environment:

1. **Sandboxing**: Restrict the execution environment to prevent potentially harmful actions. Examples:
   - **Filesystem Access**: Restrict access to the filesystem to prevent reading, writing, or manipulating files.
   - **Network Access**: Disable networking capabilities to prevent outgoing or incoming network requests.

Example

Request:

```
POST /execute

{
    "code": "import os; os.remove('file.txt')"
}
```

Response:

```
{
  "id": "8ccfcbba-5a94-49a3-aecc-edb0b8273e31",
  "stderr": "Traceback (most recent call last):\n  File \"<stdin>\"
, line 1, in <module>\nPermissionError: [Errno 13] Permission
denied: 'file.txt'\n"
}
```

Submission Guidelines

- Provide clear and complete instructions to build and run your server. Consider both Linux and Windows environments, and ensure that your setup is straightforward to reproduce.

- You are not required to solve all levels. If a level is partially complete, please submit what you have achieved and clearly state which parts are functioning and which are incomplete.
- Submit your solution in a single zip file by email to `dev@namaaconsult.com` with the subject `{FIRST NAME} {LAST NAME} - Backend Challenge Solution` (e.g. Ahmed Mohamed - Backend Challenge Solution)

Evaluation Criteria (in order of importance)

1. **Correctness**: The server should execute code as described, respect the given limits, and manage interpreter sessions efficiently.
2. **Code Quality**: Code should be clean, well-organized, and follow a consistent style. Proper comments and documentation are encouraged to make the solution easy to understand and maintain.
3. **Performance**: No fancy stuff as long as the execution takes reasonable time. A clean and simple solution is preferred over a complex, highly performant one.