Judge Image for Annaforces

License: MIT

A secure and isolated environment for executing untrusted code, primarily for use in competitive programming platforms or online judges. It uses Docker to create a sandbox that restricts resource usage (CPU time, memory) and prevents malicious code from affecting the host system.

1 Features

- **Language Support:** Executes code written in C, C++, and Python.
- **Resource Limiting:** Enforces time and memory limits on code execution.
- **Secure Sandboxing:** Uses Docker containers to isolate the execution environment.
- **Detailed Results:** Provides detailed information about the execution, including:
  - Standard output and standard error.
  - Execution time and memory usage.
  - Status (Success, Compilation Error, Runtime Error, Time Limit Exceeded, Memory Limit Exceeded).

2 Getting Started

2.1 Prerequisites

- **Docker:** You must have Docker installed and running on your system.

2.2 Installation

1. Clone the repository:

```
git clone https://github.com/lohitpt252003/judge-image-for-annaforces.git
```

2. Navigate to the project directory:

```
cd judge-image-for-annaforces
```

3 Examples

3.1 Example 1: Successful Python Execution

```
from good_one import execute_code
import json

python_code = """
import sys
name = sys.stdin.readline()
print(f"Hello, {name.strip()}!")
"""

result = execute_code(
    language='python',
```

```
    code=python_code,
    stdin='World',
    time_limit_s=5,
    memory_limit_mb=128
)

print(json.dumps(result, indent=2))
```

Output:

```
{
  "stdout": "Hello, World!",
  "stderr": "",
  "err": "",
  "timetaken": 0.0,
  "memorytaken": 3.68359375,
  "success": true
}
```

3.2 Example 2: C++ Code with a Runtime Error

```
from good_one import execute_code
import json

cpp_code = """
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v;
    std::cout << v.at(10); // This will throw an exception
    return 0;
}
"""

result = execute_code(language='c++', code=cpp_code, stdin='', time_limit_s=5,
    memory_limit_mb=128)

print(json.dumps(result, indent=2))
```

Output:

```
{
  "stdout": "",
  "stderr": "",
  "err": "Runtime Error (Exit Code: 134)",
  "timetaken": 0.0,
  "memorytaken": 0.0,
  "success": false
}
```

4 API Reference

4.1 `execute_code()`

Argument	Description	Default Value
`language`	The programming language (‘c‘, ‘c++‘, ‘python‘).	`'python'`
`code`	The source code to execute.	`"print(...)"`
`stdin`	The standard input for the code.	`''`
`time_limit_s`	The time limit in seconds.	2
`memory_limit_mb`	The memory limit in megabytes.	1024

Returns: A dictionary containing the execution results with the following keys:

- `stdout` (str): The standard output of the code.
- `stderr` (str): The standard error of the code.

- `err` (str): An error message if an error occurred.
- `timetaken` (float): The time taken for the code to execute in seconds.
- `memorytaken` (float): The memory used by the code in megabytes.
- `success` (bool): Whether the code executed successfully.

5 How It Works

The core logic is in the `execute_code` function within `good_one.py`. Here's a breakdown of the process:

1. **Input Validation:** Checks if the requested programming language is supported.
2. **Docker Setup:**
   - Verifies that Docker is running.
   - Checks if the required Docker image (`sandbox-image:latest`) exists.
   - If the image is not found, it builds it dynamically from a simple `Dockerfile` definition.
3. **File Preparation:**
   - Creates a temporary directory on the host machine.
   - Saves the user's source code and standard input to files within this directory.
4. **Container Management:**
   - Starts a detached Docker container from the `sandbox-image`.
   - Copies the source code and input files into the container.
5. **Code Compilation (for C/C++):**
   - If the language is C or C++, it compiles the code inside the container using `gcc` or `g++`.
   - If compilation fails, it returns a "Compilation Error."
6. **Code Execution:**
   - Executes the compiled binary (for C/C++) or the Python script.
   - The execution is wrapped with `/usr/bin/time -v` to measure resource usage and `timeout` to enforce the time limit.
7. **Result Parsing:**
   - Captures the standard output, standard error, and exit code of the process.
   - Parses the output of `/usr/bin/time -v` to extract the execution time and memory consumption.
   - Determines the final status based on the exit code (e.g., exit code 124 indicates a timeout).
8. **Cleanup:**
   - Stops and removes the Docker container.
   - Deletes the temporary directory from the host.

6 To-Do / Improvements

☐ Add support for more languages (e.g., Java, Go, Rust).

☐ Implement a more robust queueing system for handling multiple submissions.

☐ Enhance security by running the code as a non-root user inside the container.

☐ Improve the accuracy of resource measurement.

☐ Create a REST API wrapper around the execution logic.

☐ Add more comprehensive unit and integration tests.

☐ Parameterize the Docker image name and other constants.

7 Contributing

Contributions are welcome! Please feel free to submit a pull request.

8 Authors

- Lohit P Talavar
- Adarsh Mishra
- Priyanshi Meena

9 License

This project is licensed under the MIT License. See the LICENSE file for details.

10 Credits

- Python
- Docker
- Linux (Ubuntu)