How should we run the code provided by the user

Introduction

When we receive user submissions, we would like to be able to verify that their code fulfills the problem's requirements. To do this, we should be able to run the user's submitted code, and when running it we should have security precautions in place. We dont want the user to be able to submit an infinite loop, or a fork bomb and crash the system. We also do not want to be able to submit something that allos different submission attempts to interfere with one another, so we need to set up and then clean the environment. To do this, we can use an off-the-shelf solution to run the code, or we can build it ourselfves.

Why not build it yourself?

Whenever possible, in most contexts it's a good idea use an external library for complex parts of the system that we might not want to reinvent. See: Not Invented Here Considering that this project is supposed to take no-more than 1-2 days per week, it is not feasible to homebrew a solution for running code securely, So here are some of the criteria I have laid out for a library to cover.

Criteria for an external library

There is a need for a library that can do the following:

- have a web API (for scalability reasons)
- can run the following languages:
    - Rust
    - C#
    - Python
    - JavaScript
    - C/C++
- can easily modify the installed language version
- can easily add/remove different languages
- can spin up a clean environment in which to run the user submitted code
- the library doesnt allow execution leaks (jailbreaks), namely one run to affect another's outut
- the library has security measures implemented
- implementation should be hostable on-premises

Why Piston?

1. **Clean Language Runtime Environment**: Automatic Configuration: Piston excels in automatically configuring a clean language runtime environment. It provides a clean OS image or can be provisioned with just enough data for the desired execution, ensuring a consistent and isolated environment.
2. **Flexibility in Handling Input**:
    - **Multiple File Support**: Piston stands out by easily accepting multiple files for execution, streamlining the process for scenarios involving complex exercises input scenarios and structure bundles.
    - **Stdin Support**: The ability to accept stdin as input enhances user flexibility, allowing for dynamic program execution.
3. **Languages support**: The library provides a tool to assist the administrator with:
    - Adding new languages
    - Removing specific languages/versions
    - Changing language versions

    The library also supports all of the abovementioned languages without any problems, and if administrators want to add a future version for experimentation, they can do so quite easily.
4. **Scalability**: Piston works just off of simple HTTP requests and can easily be made to work with low and high loads spinning up and down instances as demand fluctuates.
5. **Built-in Security**:
    - **Automatic Timeouts**: Piston incorporates automatic timeouts to weed out slow-running code or infinite loops, enhancing security and resource management.
    - **Memory Limits**: Piston enforces memory limits, protecting against excessive resource consumption.
    - **Process Count Cap**: Piston includes a process count cap to control the number of concurrent executions, preventing system overload.
    - **IO Limits**: IO limits are in place to regulate input/output operations, ensuring controlled access to system resources.
    - **Host OS Protection**: Piston has security measures preventing the execution of commands that could harm the host OS.

Drawbacks:

- **No builtin way of calculating resource usage**: Compared to a solution like Domjudge or OpenJudgeSystem, there is no builtin section of the API to tell you how long it took to execute your code. There are however API properties to limit the runtime length and execution duration as outlined above. This is not going to be hard to include, it will just require modifying the piston library.

Why not Judge0?

Judge0 is interesting due to its extensive configuration options, and scalability, and commercial support.

- **Drawbacks**:
  - **Configurability**: Judge0 includes features like batching submissions, authentication, and run history that are better suited for implementation at the backend level rather than within the execution engine.
  - **Bloat**: Judge0 does not offer customization of the installed languages, meaning you will waste a lot of space per judge0 instance.
  - **Customisation** : Judge0 offers no ways for an administrator to easily change the version of the installed language. That is not a trivial task for a regular system administrator, and requires knowledge of the Judge0 codebase, which even though is open-source, should be customised using recipes like Piston does.
- **No builtin way of calculating resource usage**: Compared to a solution like Domjudge or OpenJudgeSystem, there is no builtin section of the API to tell you how long it took to execute your code. There are however API properties to limit the runtime length and execution duration as outlined above. This is not going to be hard to include, it will just require modifying the piston library.

Gutting another solution's code running internals?

Implementations that usually are included in a competing solution are not entirely straightforward to understand and port. This also does not consider the technical debt that will be incurred by including a component that is purpose-built for a different platform, ableit with similar requirements, the maintenance side will be costly.

1. **Open Judge System**
   - **Dependency on Windows Server**: The OpenJudgeSystem introduces a notable dependency on Windows Server, which may pose challenges for users seeking cross-platform compatibility. This reliance on a specific operating system can limit deployment flexibility and may not align with the preferences or infrastructure of all users.
   - **Configuration Challenges**: Configuring the Open Judge System is a cumbersome process. The dependency on Windows Server adds complexity to the setup, also adding proprietary software as a dependency, and the fact that the backend for this system is not designed to be scalable beyond a running instance make it not suitable for extraction and use just by itself.
2. **DOMJudge**

- **Manual configuration**: DOMJudge requires the user to manually configure a lot of the infrastructure around the running part of the code, allowing you to use your own compilers, interpreters, and change versions of the languages as you wish. Problem with this is the user needs to be profficient with the OS and be careful when configuring the runners, as it is quite easy to allow an execution leak and spoil the results. This is also intertwined with the system, telling it what compiler is available, and is not easy to port over to an API if it was not engineered to be taken out from the beginning.
- **Mix of programming languages**: As DOMJudge is written in PHP and C, the code running component is also written in both languages, making it a maintenance nightmare for this project.
- **Accurate runtime results**: DOMJudge provides the user with accurate resource usage information after every submission.

Conclusion

From the given options, none of them are perfect, and the closest one to the requirements given above is Piston. The fact there is no used resource information is not great, but is easy to get implemented.

While Judge0 offers more features than Piston (that are actually meant to be implemented on the backend rather than the code-execution side), it doesn't make sense as it comes with a lot of baggage around language configuration (choice), and it also suffers a similar issue around used resource information.

And looking at some of the existing "competing" solutions for running competitions, gutting one and using its execution library doesnt make sense as it will be a maintenance nightmare around either the dependence on an operating system, security, or language choices.

References

Engineer-man/piston: A high performance general purpose code execution engine. (2021, January 4). GitHub. Retrieved November 10, 2023, from https://github.com/engineer-man/piston

Hackerrank interview/competition platform. (n.d.). HackerRank. https://www.hackerrank.com/

How I built the internet's best performing code execution engine [Video]. (n.d.). YouTube. https://www.youtube.com/watch?v=SD4Kgwdjmdl

Robust, scalable, and open-source online code execution system that can be used to build a wide range of applications that need online code execution features. (2017). Judge0. https://judge0.com/

SaaS platform offering job-focused courses on algorithms in different languages. (n.d.). https://www.codechef.com/

DOMjudge/domjudge: DOMjudge programming contest jury system. (n.d.). GitHub. https://github.com/DOMjudge/domjudge/tree/main

(n.d.). DOMjudge. https://www.domjudge.org/

NikolayIT/OpenJudgeSystem: An open source system for online algorithm competitions for Windows, written in ASP.NET MVC. (n.d.). GitHub. https://github.com/NikolayIT/OpenJudgeSystem