DOCUMENTATION: SECURE BUILD SERVER

As per the assignment, I've written out code for a build server that could potentially be exposed onto a local network, satisfies all of the requirements with room for error, and adds an extra level of protection for build records against either malicious manipulation or accidental alterations.

TECH STACK & DESIGN CONSIDERATIONS:

Frontend webpage: a simple, singular HTML file that makes an API call to a get endpoint and lists out pertinent build details in order from the most recent build to the oldest. Details can be kept up to date by refreshing the page. Efforts were made to ensure that, while the database page is barebones, it is not a mess to read and acts as a more efficient option than merely querying a database or navigating to the index endpoint directly.

Repositories: raw_data_repo and build_artifacts_repo, both hosted locally. We are not exposing either of these repositories to remote hosting in an effort to keep them as pristine and protected from alteration as possible, be it malicious or accidental.

Database: a one-table MySQL database has been created to log and track build details for viewing later.

Backend: with consideration to the preferences laid out in the assignment-as well as elements of the tech stack Michael said we'd be using at HRL which I wasn't wholly familiar with-I decided I wanted to use FastAPI and Python3 to build out this server's backend. Each endpoint is a combination of python code, MySQL queries, and bash scripts for performing efficient low-level file operations as would normally be required for working with git cli and C programs.

Three endpoints have been built out for this project: one post, one index, and one get. The design philosophy here was limited exposure of the git repos containing build data while still ensuring full functionality for a hypothetical dev team via API calls alone, thus ensuring the immutability and reliability of both repos as a system of record. This also nullifies the need for developers to ssh or otherwise directly access the build server in any capacity outside of a trusted administrator for occasional debugging. All of these endpoints and all associated functions operate on no assumptions of naming convention or directory structure outside of assuming that a Makefile is present at the top folder and that potential remnant git data may be present from a prior repo. This data will be removed by a subprocess shell script before committing to the new one.

The post endpoint takes a zip file (and only .zip files, anything else will be rejected outright). These zip files have their contents extracted to a repo containing the raw data used for builds and are then promptly removed. This allows a team to upload full C program projects into the repo, logging their git tags with git rev-parse HEAD without having direct access to git.

The aforementioned git tags can be used as a path parameter in the single get endpoint to retrieve completed build artifacts, logs, Makefile output and any executables generated from the build_artifacts repository in the form, again without the user having to directly access the repo. As is discussed further below, the git commit tag retrieved by git rev-parse HEAD is accessible from the webpage and the get-all endpoint. Put it at the end of the base URI for

FastAPI (e.g. 127.0.0.1:8000/{git-endpoint}) in your browser and a download for a zip containing all related artifacts and executables generated in the build process will begin.

Finally, the get-all/index endpoint returns a list strings containing pertinent data from the MySQL table including commit tags, build #, upload time, build start/end times, build status/success, and the name of the artifact and raw_data folders with which the project data is associated. It can be accessed either as the root route or from the HTML webpage I created earlier.

Periodic building: a python3 script called build_and_log.py has been set up to run once a minute using crontab. Unless our development team is large and our process for code review abnormally quick, this should be more than frequent enough. This job takes the most recent uploaded data and attempts a build using a Makefile after updating the associated database entry created upon initial upload to reflect its "building" status. Whether or not the build is successful, all artifacts and associated files will be saved to an appropriate artifacts folder and committed for later viewing. If the build is successful (this is confirmed by a quick search for executables at the top directory level of the data folder), it will be marked as such in the database, and vice versa for a failed build.

SETUP INSTRUCTIONS:

Contained in the zip file I sent you (repos.zip, not build_server.zip) are all of the files in the initial directory structure with which I ran the program on my end. Make sure that the files are all extracted to ~/repos with that structure intact.

To run the API, simply run `uvicorn main:app –reload` from the repos directory. The get-all endpoint can either be accessed with 127.0.0.1:8000 or more cleanly by navigating to the getall.html file in your browser once while the API is running.

Here we reach the part of setting up the project where, in my experience, there is the most potential for build failure or issues at runtime. Within build_and_log.py there's a more involved setup: crontab requires absolute paths to scripts and commands to run correctly, so you'll have to run a `readlink -f {filename}` command and swap out the uris in the subprocess lines to their absolute path equivalent. The same will have to be done for any filepaths within those called shell scripts and in the below crontab command with the location of your python3 installation.

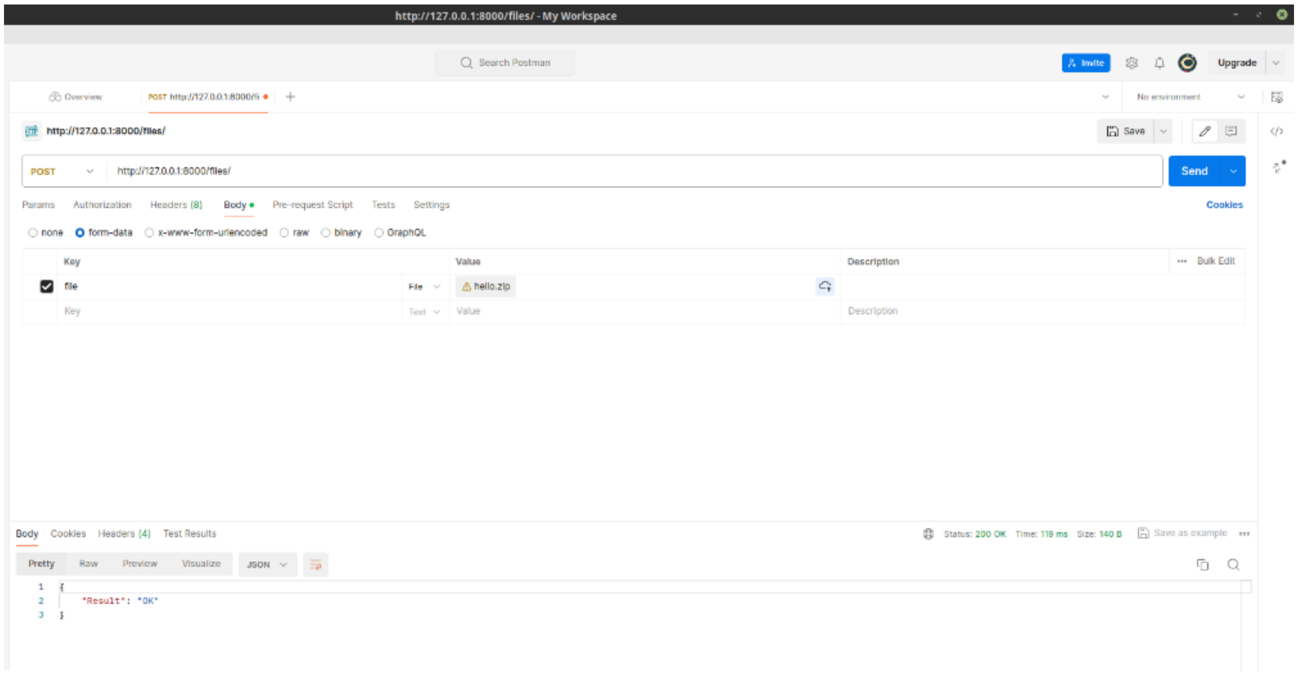
```
crontab -e
*/1 * * * * {absolute/path/to/python3} {absolute/path/to/build_and_log.py}
```

To setup the MySQL correctly for the API and associated scripts to interact with, initialize the MySQL service, change the root user password to root (this is necessary to ensure that all of the bash and SQL scripts can successfully query and update entries) and run the statements in SQL_SCRIPTS/build_server_data.sql in sequence (assuming the script itself doesn't execute properly from the CLI).

TROUBLESHOOTING

The biggest potential points of failure on setup are going to be related to shell script and dependency pathing specifically in crontab and the associated file build_and_log.py. If, while testing, you get errors like “this file not found”, double-triple check that in those files directory and script paths are absolute and not relative.

The next biggest point of failure comes in the POST testing. The two get endpoints can easily be tested via browser, but the post endpoint is a bit more fickle. I used Postman to test mine and have included a relevant screenshot below of what your workspace should look like to ensure successful API consumption. URI should read http://127.0.0.1:8000/files/ and the request sent should be form-data with key=file and value={zip file of C program}.



POST http://127.0.0.1:8000/files/

Body: form-data

Key	Value	Description
file (File)	hello.zip	

Status: 200 OK Time: 119 ms Size: 140 B

```
{
    "Result": "OK"
}
```

Finally, you may encounter a CORS related error in your browser console when accessing the HTML webpage while the API is running. The code I added to main.py fixed it on my end and should fix it on yours, but if, for some reason, this isn’t the case, I believe a few threads on StackOverflow mentioned a CORS-related Chrome extension that can be used to resolve this error.

SHORTCOMINGS AND POTENTIAL FOR EXPANSION

There’s obviously more that can be done to make this program more robust, primarily in terms of input error handling, but my main goal here was keeping uploaded data both available and immutable, automating the build process, and creating a platform upon which other features could be built to tailor to more specific projects. I tried to avoid hard coding anything wherever possible while still ensuring that it would always work for the general project structure I was provided over email.

One thing that I don’t think I’ll have the time to do-but was taken into account with the project structure-was local network hosting over a web server. Given that this is intended to be used by small-to-medium sized teams in an on-prem setting with connectivity supported via an API to allow free access with an immutable repository structure underneath, the project can be-and almost assuredly should be, were it to be put into production-expanded with an in-house

DNS server of some kind and web hosting on a linux machine otherwise air gapped from the wider internet.

The frontend could also be expanded to provide up-to-the-minute updates on builds and statuses, and another pair of HTML pages for more user-friendly uploading and downloading of data wouldn't be a bad idea, but given the requirements laid out in the PDF I was sent, I considered this a bit out of scope.