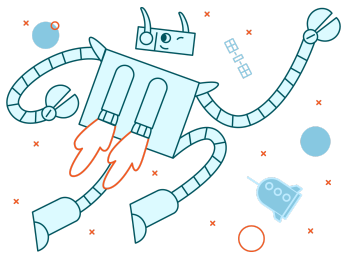
Advanced HTTP Server



Goal

In this exercise, we'll enhance our simple HTTP server by adding features such as serving static files, handling photo uploads, logging, reading configurations, and adding basic security measures. These additions will make our server more robust and functional!

Tasks

Task 1: Serve static files

- Create a directory named `www` in the same folder as your server's code.
- When a client requests `http://localhost:8000/`, the server should respond with the content of the file `www/index.html`. For a request to `http://localhost:8000/images/logo.png`, the server serves the file located at `www/images/logo.png`, and so on.
- This involves parsing the URL, constructing the file system path by appending the URL path to the root of the static files directory (`www/`), checking if the file exists, reading it if it does, and then responding appropriately with the file's content.
    - Hint: Read about MIME types and the Content-Type header to help the client's browser recognize the response's content type (HTML, image, etc.).

Task 2: Support photo uploads

- Implement functionality to handle `POST` requests for uploading JPEG and PNG photos.

- Save the uploaded photos to a specified directory in the configuration file (`upload_dir`). If the directory doesn't exist, the server should create it.
- Respond with a success message indicating the path where the photo was saved, and handle errors for unsupported file types or saving issues.

Task 3: Implement logging

- Use Python's `logging` module to set up logging for server activities, including each request and response.
- Log details like the timestamp, client IP, request method, URL, and response status.
- Configure the logging to output to the console or to a file based on settings in the configuration file.

Task 4: Use a configuration file

- Use a JSON configuration file (`server_config.json`) to manage server settings such as `port`, directories for static files and uploads, and other operational parameters like `max_request_size` and `timeout_seconds`.
- Load these settings at server startup, handling any errors in reading or parsing the configuration file by logging an error and using default settings or terminating the server.

Example configuration file (`server_config.json`)

```
{
  "port": 8000,
  "static_files_dir": "www",
  "upload_dir": "uploads",
  "max_request_size": 1048576,
  "timeout_seconds": 30
}
```

Task 5: Add security features

- Implement a maximum request size limit to prevent overly large uploads that could overload the server. This limit should be set in the configuration file (`max_request_size`).
- Set up a timeout for connections to automatically close if a client fails to complete a request within a specified duration (`timeout_seconds` in the configuration file).
- These security measures help ensure the server can handle traffic efficiently and protect against certain types of denial-of-service (DoS) attacks.

Running the server

1. Save your server code in a file called `advanced_http_server.py`.
2. Create the configuration file `server_config.json` in the same directory.
3. Open a terminal or command prompt, navigate to the directory containing your server file, and run it using Python: `python advanced_http_server.py`.
4. Use a web browser or a tool like Postman to test the server's functionality by accessing static files, uploading images, and observing the logging outputs.

To submit

- Your `advanced_http_server.py` file containing the complete code.
- The `server_config.json` configuration file.