Web Server

In this programming assignment, you will implement a simple web server to maintain your personal music library. Through this assignment, you will gain experience developing a more complex program that utilizes multiple CPU cores concurrently, implements persistent data storage, and incorporates external crates from the Rust ecosystem as dependencies. This assignment is due at **11:59 p.m.** on **Monday, December 1, 2025**.

Basic Web Server

As a starting point, your new web server, called `server` for simplicity in this assignment, should be able to accept standard `GET` HTTP requests from a web client. Your web client can be your favourite web browser or the UNIX `curl` utility. Your web server should accept requests to the local host (called `localhost`) and port `8080`, and produce a response.

For example, if you were to run the following command:

```
$ cargo build
$ target/debug/server
```

The output will be:

```
The server is currently listening on localhost:8080.
```

This implies that the web server is ready to respond to HTTP requests sent to `localhost:8080`. If a web client, such as your web browser or the command-line `curl` utility, sends a HTTP request to your new server, it will respond as follows:

```
$ curl "http://localhost:8080"
Welcome to the Rust-powered web server!
```

> If you are using `zsh` as your shell, you may notice a `%` at the end of the line. This indicates that `curl`'s output does not include a newline character. To eliminate the `%` at the end, you can use `curl -w '\n' http://localhost:8080` instead.

Site-Wide Visit Count

Now that a basic web server is functional, you can start to work on a new site-wide visit count. Every time a web client visits the URL `http://localhost:8080/count`, the server's visit count will be incremented by 1. For example, if we use `curl` as the web client, we will see the following responses:

```
$ curl "http://localhost:8080/count"
Visit count: 1
$ curl "http://localhost:8080/count"
Visit count: 2
$ curl "http://localhost:8080/count"
Visit count: 3
```

and so on, until the web server is restarted (when the visit count is reset to 0).

Please keep in mind, however, that your implementation of the web server should make every attempt to be *performant*, in that it should try to take advantage of multiple CPU cores on your computer. In the following sequence of commands, an HTTP load generator utility, called `oha`, is used to visit your server for 1000000 times concurrently:

```
$ cargo install oha
$ oha -n 1000000 "http://localhost:8080/count"
$ curl -s "http://localhost:8080/count"
Visit count: 1000001
```

In this example, the `oha` command-line utility is used to generate HTTP load on your web server, and to measure the amount of time for your web server to process 1000000 requests. On the instructor's MacBook Pro M1 Max computer, it takes about 20 seconds for `target/debug/server` and 10 seconds for `target/release/server` (after building the release version using `cargo build --release`). Depending on the speed of your computer, your measured times may vary. Needless to say, your visit count should be correctly maintained, in that your next visit will return a visit count of `1000001`.

During the time when your web server is processing these requests, you can also use the `htop` command to check whether all CPU cores on your computer have been fully utilized, so that server performance is maximized:

```
htop
```

If you do not yet have the `htop` utility installed, you can use `brew install htop` to install it on macOS, and `sudo apt install htop` to install it on Ubuntu Linux.

Personal Music Library

With some initial experience building the site-wide visit count, you should be ready to extend your web server to maintain information about your personal music library. Each entry will contain several pieces of information: a song's title, its artist, its genre (the type of music), as well as its play count. For simplicity, the song's title, artist, and genre are strings, and its play count is an integer value.

Your web server will allow you to add new songs to your personal music library, to search your personal music library for songs, and to play a song in your library.

1 Adding new songs

To add a new song, a JSON POST request should be sent to your web server using the `songs/new` route, with the title, artist, and genre included in the JSON data. Your

web server will produce a JSON response, containing a unique `id` assigned to the song, and an initial play count of 0. For example:

```
$ curl "http://localhost:8080/songs/new" \
  --json '{"title":"Bohemian Rhapsody", "artist":"Queen", "genre":"Rock"}'
{"id":1,"title":"Bohemian Rhapsody","artist":"Queen","genre":"Rock","play_co
$ curl "http://localhost:8080/songs/new" \
  --json '{"title":"Love Story", "artist":"Taylor Swift", "genre":"Country"}
{"id":2,"title":"Love Story","artist":"Taylor Swift","genre":"Country","play
$ curl "http://localhost:8080/songs/new" \
  --json '{"title":"Welcome to New York", "artist":"Taylor Swift", "genre":"
{"id":3,"title":"Welcome to New York","artist":"Taylor Swift","genre":"Pop",
```

Before adding new songs to the personal music library, the web server does not need to check for duplicates.

2 Searching for songs

A web client can send a query to the web server using the `/songs/search` route, searching for a particular title, artist, genre, or any combination of these attributes. For example:

```
$ curl "http://localhost:8080/songs/search?title=Bohemian"
[{"id":1,"title":"Bohemian Rhapsody","artist":"Queen","genre":"Rock","play_c
$ curl "http://localhost:8080/songs/search?artist=Queen"
[{"id":1,"title":"Bohemian Rhapsody","artist":"Queen","genre":"Rock","play_c
$ curl "http://localhost:8080/songs/search?genre=Rock"
[{"id":1,"title":"Bohemian Rhapsody","artist":"Queen","genre":"Rock","play_c
```

A query can also include a conjunctive combination of these attributes:

```
$ curl "http://localhost:8080/songs/search?genre=Country&artist=Taylor+Swift
[{"id":2,"title":"Love Story","artist":"Taylor Swift","genre":"Country","pla
```

Your web server's response to a query should contain all matching songs whose title, artist, or genre contains the search keyword **anywhere within the string**:

```
$ curl "http://localhost:8080/songs/search?artist=Swift"
[{"id":2,"title":"Love Story","artist":"Taylor Swift","genre":"Country","pla
```

All queries are case-insensitive. Here is an example:

```
$ curl "http://localhost:8080/songs/search?artist=taylor+swift"
[{"id":2,"title":"Love Story","artist":"Taylor Swift","genre":"Country","pla
```

If a query does not match any song in the personal music library, your web server should produce an empty JSON response:

```
$ curl "http://localhost:8080/songs/search?genre=Rock&artist=Taylor+Swift"
[]
```

3 Playing a song

A web client can play a song with a specific `id` by sending a GET request to the web server using the `/songs/play` route. When a song is played, its play count is incremented by 1. For example:

```
$ curl "http://localhost:8080/songs/play/1"
{"id":1,"title":"Bohemian Rhapsody","artist":"Queen","genre":"Rock","play_co
$ curl "http://localhost:8080/songs/play/1"
{"id":1,"title":"Bohemian Rhapsody","artist":"Queen","genre":"Rock","play_co
```

If a song is not found in the music library, the server will return an error in its JSON response:

```
$ curl "http://localhost:8080/songs/play/4"
{"error":"Song not found"}
```

> Your personal music libary should be **persistent**, in that even after your web server is terminated and then restarted, the entire content of your libary should still remain available.

Implementation Notes

As this is the final assignment in the course, you should feel free to use any third-party crate as your dependencies. However, try to avoid overly complex designs — the instructor's solution only contains 178 non-empty lines of code:

```
$ cat src/main.rs | sed '/^\s*$/d' | wc -l
178
```

When you start to work on your assignment, make sure you use the command `cargo new` to create a new project, including a `Cargo.toml` file and a `src/main.rs` file, as well as an initial git repository[1]. Take advantage of the git repository that `cargo new` creates for you, and push your local commits to GitHub. Once your project is managed by git, you can then start writing your code in the `src/main.rs` file[2]. Just like our previous assignments, you can compile and run your code using the `cargo run` command.

Testing for Correctness

For your convenience, an initial set of 6 test cases has been released to you as a compressed archive, named `a4-public-tests.tar.gz`, and reflecting what you can find in this section. Run the command `tar zxvf a4-public-tests.tar.gz` to uncompress the archive to a directory, and then move the content of the archive to your source code directory that you created using `cargo new`, so that `cargo run` works successfully. You can use these test cases to test your implementation against the expected output, and

revise minor output differences as necessary. To run the tests, you will need to start the server manually first, and run the following script in another terminal:

```
bash input.sh > my_output.txt
```

To compare the output of your program with the expected output, you can use the `diff` command:

```
diff my_output.txt output.txt
```

Please ensure that your persistent data storage has no data records before these tests commence. These tests do not check data persistence after restarting the server, but you may use the same script to test and evaluate it yourself.

Similar to previous assignments, several tests have been held back as private test cases for marking purposes.

Testing for Performance

Your solution will also be tested for performance, if it passes all test cases for correctness. The performance test suite will first create a personal music library by adding 100,000 songs, and then measure the time your server takes to respond to queries using this large library. The performance test suite has been released for your own testing purposes as well.

Notes on AI Tools

This assignment is designed for manual work without AI assistance, and you should **not** use AI tools at all for this assignment. While we recognize that the habit of using AI may be so reflective and strong that — like social media — you cannot think without AI assistance, you need to learn coding in the Rust programming language and this learning experience needs to start from somewhere. Similar to all three previous assignments, this assignment offers a sufficiently simple canvas for you to continue your learning experience and to

expand your set of skills in Rust, and using AI defeats the purpose and completely changes the learning experience. Heed our advice: do **not** use AI and you will enjoy such *"raw"* coding experience much more.

Submission

Submit your project by submitting a `.tar.gz` archive of it using Quercus, under *Assignment 4*. Use the following command to create your `.tar.gz` archive:

```
cd /path/to/parent/directory
tar zcvf 1234567890.tar.gz project_directory
```

where `1234567890` is your student number, and `project_directory` is the name of the directory containing your project files, such as `server`. This is also the project name you initially provided to the `cargo new` command. Make sure that your archive contains all the files necessary to build and run your project, including the `Cargo.toml` file and the `src` directory as shown above. Also, make sure that your project builds and runs correctly after extracted from the archive, using the `cargo run` command.

⚠️ The deadline for this assignment is **Monday, December 1, 2025**, at **11:59pm** Eastern time, and late submissions will **not** be accepted.

Marking

Your project will be built using the `cargo run --release` command, and tested against a number of test cases. The marking will be based on the correctness of your implementation in these test cases, by comparing the output of your code to the correct output. Make sure that your code is well-commented, easy to read, and that you have followed the Rust naming conventions. It goes without saying that you should also make sure that your code is free of warnings when compiled with the `cargo build` or the `cargo check` command.

One of the test cases for correctness will be designed to test your site-wide visit count. It will be used to make sure that your solution not only uses all CPU cores concurrently, but also maintains the correct visit count after 1 million concurrent visits.

If your solution passes all tests for correctness, it will be tested for performance using our performance test suite. If your solution is in the top 50% of the students in the class who qualify for performance testing, you will be awarded a **bonus mark** of 10%. If your solution is in the top 25% of the qualified students, you will be awarded an **additional bonus mark** of 10%.

1. If `cargo new` is executed within an existing git repository, no new git repositories will be initialized. ↩
2. Feel free to divide your code into multiple `.rs` files in the `src/` folder. ↩

Last updated on November 9, 2025