Web Client

In this programming assignment, you are expected to implement a command-line utility that makes HTTP requests, similar in spirit to the `curl` UNIX utility. This assignment will help you gain more experiences on using external crates in the Rust ecosystem as dependencies, and on handling errors. This assignment is due at **11:59 p.m.** on **Monday, November 10, 2025**.

Basic Features

As a starting point, your new command-line utility — called `curl` for simplicity in this assignment — should be able to make a standard `GET` HTTP request to a web server, and print out the response.

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

```
$ cargo build
$ target/debug/curl "https://www.eecg.toronto.edu/~bli/ece1724/assignments/files/
```

The output will be:

```
Requesting URL: https://www.eecg.toronto.edu/~bli/ece1724/assignments/files/lab3.
Method: GET
Response body:
<html>
<body>
<h1>
Hello, World!
</h1>
</body>
</html>
```

Now that the basic functionality has been built, you can start adding more features to your command-line web client. Here are some of the features you should implement:

1 Handling Errors in URLs

If there are errors in URLs, your utility should be able to handle them gracefully by printing more details about these errors. Your solution will be tested with three kinds of malformed URLs:

- The URL does not have a valid base protocol. As examples, if you were to run the following commands:

```
$ target/debug/curl "www.eecg.toronto.edu"
$ target/debug/curl "data://www.eecg.toronto.edu"
$ target/debug/curl "http//www.eecg.toronto.edu"
```

The output will be:

```
Requesting URL: www.eecg.toronto.edu
Method: GET
Error: The URL does not have a valid base protocol.
```

```
Requesting URL: data://www.eecg.toronto.edu
Method: GET
Error: The URL does not have a valid base protocol.
```

```
Requesting URL: http//www.eecg.toronto.edu
Method: GET
Error: The URL does not have a valid base protocol.
```

- The URL contains an invalid IP address. As examples, if you were to run the following commands:

```
$ target/debug/curl "https://[ ...1]"
```

```
$ target/debug/curl "https://255.255.255.256"
```

The output will be:

```
Requesting URL: https://[...1]
Method: GET
Error: The URL contains an invalid IPv6 address.
```

```
Requesting URL: https://255.255.255.256
Method: GET
Error: The URL contains an invalid IPv4 address.
```

- The URL does not have a valid port number (between 0 and 65535, inclusive). For example, if you were to run the following command:

```
$ target/debug/curl "http://127.0.0.1:65536"
```

The output will be:

```
Requesting URL: http://127.0.0.1:65536
Method: GET
Error: The URL contains an invalid port number.
```

2 Handling Errors When Making Requests to the Web Server

When your utility makes a request to the web server, several different kinds of errors may occur. Your utility will be tested with the following two types of errors:

- The host address cannot be resolved (or the network is offline). For example, if you were to run the following command:

```
$ target/debug/curl "https://example.rs"
```

The output should be:

```
Requesting URL: https://example.rs
Method: GET
Error: Unable to connect to the server. Perhaps the network is offline or th
```

- The web server returns a status code. For example, if you were to run the following command:

```
$ target/debug/curl "https://www.eecg.toronto.edu/~bli/ece1724/assignments/f
```

The output should be:

```
Requesting URL: https://www.eecg.toronto.edu/~bli/ece1724/assignments/files/
Method: GET
Error: Request failed with status code: 404.
```

3 Supporting the `POST` Method

By default, your utility supports the `GET` method when requesting from a web server. It is useful to support the `POST` method as well. Using the `POST` method, your utility sends the specified data in a `POST` request to the web server, in the same way that a browser does when a user has filled in an HTML form and presses the submit button.

Just like the `curl` UNIX utility, you should support two new command-line arguments, `-X POST` and `-d`. `-X POST` specifies that the `POST` method should be used, and `-d` is followed by a string that contains one or multiple key-value pairs separated by `&`, each in the form of `key=value`.

To print the response body from the web server a bit more nicely, you should first check if the response body is JSON formatted. If this is the case, print it with keys sorted in alphabetical order. Otherwise, print the response body directly.

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

```
$ target/debug/curl "https://jsonplaceholder.typicode.com/posts" -d "userId=
```

The output will be:

```
Requesting URL: https://jsonplaceholder.typicode.com/posts
Method: POST
Data: userId=1&title=Hello World
Response body (JSON with sorted keys):
{
  "id": 101,
  "title": "Hello World",
  "userId": "1"
}
```

4 Sending JSON Formatted Data to a REST API Server

Recently, `curl` 7.82.0 introduced the `--json` option as a new way to send JSON formatted data to web servers using `POST`. This is handy when testing REST API servers that respond to JSON data. In `curl` 7.82.0, this option is equivalent to combining the `-X POST` option, the `-d` option with the JSON formatted data, as well as `--header "Content-Type: application/json"` to add a custom header that lets the web server know that the data is sent in JSON format[1]. For example, if you were to run the following command:

```
$ target/debug/curl --json '{"title": "World", "userId": 5}' "https://dummyj
```

Your output will be:

```
Requesting URL: https://dummyjson.com/posts/add
Method: POST
JSON: {"title": "World", "userId": 5}
Response body (JSON with sorted keys):
```

```
{
  "id": 252,
  "title": "World",
  "userId": 5
}
```

If the JSON formatted data provided at the command line is not valid JSON, your program should panic:

```
$ target/debug/curl --json '{"title": "World"; "userId": 5}' "https://dummyj
```

```
Requesting URL: https://dummyjson.com/posts/add
Method: POST
JSON: {"title": "World"; "userId": 5}
thread 'main' panicked at src/main.rs:84:58:
Invalid JSON: Error("expected `,` or `}`", line: 1, column: 18)
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrac
```

Implementation Notes

Use the `reqwest` crate to make requests to the web server. You may also find three more crates useful: `url` to parse URLs, `structopt` to help process command-line arguments more easily, and `serde_json` to help parse JSON formatted data.

You only need to explicitly handle errors that have been mentioned in this assignment. You can assume that the test cases will not contain errors that have not been explicitly mentioned. For these errors that have not been mentioned, feel free to print your own error messages or simply panic at run-time.

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[2]. 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[3]. Just like our previous assignments, you can compile and run your code using the `cargo run` command.

Testing

For your convenience, an initial set of 5 test cases has been released to you as a compressed archive, named `a3-public-tests.tar.gz`, and reflecting what you can find in this section. Run the command `tar zxvf a3-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 test, you can use the following command:

```
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
```

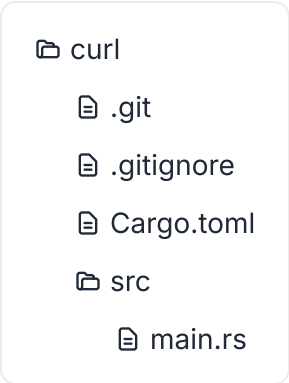
Notes on AI Tools

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 the previous assignment on the Reversi board game and the search utility, 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 3*. 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 `curl`. 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, November 10, 2025**, at **11:59pm** Eastern time, and late submissions will **not** be accepted.

Marking

Your project will be built and tested using the `cargo run --` command against a set of 10 test cases[4], including 5 public test cases that are released to you, and 5 more hidden test cases that we use internally for testing your solutions. 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. You should also make sure that your code is free of warnings when compiled with the `cargo build` or the `cargo check` command.

1. You do not need to implement the `--header` option; only `--json` is needed. ↩
2. If `cargo new` is executed within an existing git repository, no new git repositories will be initialized. ↩
3. Feel free to divide your code into multiple `.rs` files in the `src/` folder. ↩
4. `--` in this command implies that all subsequent arguments are passed to the program being run, rather than to `cargo`. ↩

Last updated on October 18, 2025