mersive technologies inc

Software Developer Project

(Cloud Applications)

Summary

Mersive seeks developers who can work with a list of simple requirements to create a robust, safe, and performant application. Much of the code at Mersive deals heavily with networking, multimedia, and time-sensitive operations. This code project's purpose is to give the Mersive team members greater insight into a candidate's familiarity with the language of choice, code organization, and their style. Many of the current engineers at Mersive have themselves written solutions to this project, and each presented a different perspective with creative ideas.

Write a minimal HTTP server using your choice of the following languages: Kotlin, Rust, Java, C#, or Python. You may use whatever framework you wish, but you must provide unit tests. The server needs to listen for a subset of HTTP requests for arbitrary URIs. When clients request that data be posted to the server, the server should somehow store the associated MIME type and body data. This data may then be later requested by another client. The data may also be deleted, if requested by the client.

Details

- Supported HTTP Verbs
    - GET: Server should retrieve any data associated with the specified URI.
        - If the request is successful, return HTTP 200 along with any data.
        - If no associated record is found, return HTTP 404.
    - POST: Server should store the MIME type and body data associated with the specified URI. If the server already had data for this URI, it should be replaced with the new data.
        - If the request is successful, return HTTP 200 and an empty body.
        - If for some reason it is determined that the post cannot be allowed or completed, return HTTP 403.
    - DELETE: Server should remove any data associated with the specified URI.
        - If the URI exists, remove the data and respond with HTTP 200.
        - If no associated record is found, return HTTP 404.
- Supported HTTP Headers
    - Content-Type
        - Referred to also as the MIME type.
        - Specifies the format of the data being sent.
    - Content-Length
        - The length of the content to expect in the HTTP body.
- Features that should not be implemented
    - HTTPS / Security.
    - Any HTTP verb outside of the ones specified above.
    - Any HTTP header outside of the ones specified above.
    - HTTP chunked encoding.
- Additional Requirements
    - The server should ignore any additional HTTP headers provided by client requests.
    - Any unhandled HTTP verb should return HTTP 405.
    - Include instructions for building and running your solution.
    - Include unit tests and how to run them
    - Summarize your design, including any tradeoffs, possible improvements, or performance concerns which may apply.
    - Submit the project in an archive of some format (zip, tar, etc) or provide a GitHub link to your solution.
    - Please plan an appropriate amount of time to accomplish the goals given in the summary. For previous submissions, the average is about 5 hours.

Introduction to HTTP

In case you're not already familiar with the HTTP protocol, here's what you need to know.

Requests

The first line contains three tokens: the verb, the URI, and the HTTP version. All lines end in an ASCII carriage return followed by an ASCII line feed. The first line is followed by zero or more headers, followed by an empty line. If there is a request body, it will follow that empty line for as many bytes as indicated by the Content-Length header. If there is no request body (e.g. a GET or DELETE), the empty line ends the request.

Requests take the following forms:

```
VERB /URI HTTP/1.1
Header-1: Header-1 Data
Header-2: Header-2 Data

Possible HTTP body (payload) data
```

An example GET, which retrieves the resource “resource/named/foo” at the domain “example.com”, and which contains a few other irrelevant (to our purpose) headers:

```
GET /resource/named/foo HTTP/1.1
User-Agent: curl/7.35.0
Host: example.com
Accept: */*

```

An example POST, which attempts to create/update a resource named “myNewResource” which should contain 8 bytes (“TestData”) in a plain-text format:

```
POST /myNewResource HTTP/1.1
Content-Type: text/plain
Content-Length: 8

TestData
```

An example DELETE, which attempts to delete the resource named “myNewResource”:

```
DELETE /myNewResource HTTP/1.1

```

Responses

The first line contains the protocol version, response code, and a string describing the response code. Again, each line is followed by a carriage return and line feed, and there is an empty line between the headers and the body, if present.

Responses take the following form:

```
HTTP/1.1 RESPONSE-CODE RESPONSE-TEXT
Header-1: Header-1 Data
Header-2: Header-2 Data

Possible HTTP body (payload) data
```

An example response to the above example GET:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 13

Response Data
```

An example response to a failed request:

```
HTTP/1.1 404 Not Found

```

Example Sequence and Expected Responses

Using the command line utility cURL, one can test the functionality of their server with some easy commands. The below examples show a possible sequence of requests from a client, and the expected responses.

By default, the server should return "404 Not Found" for all URIs:

```
$ curl -i http://localhost:8000/firstResource
HTTP/1.1 404 Not Found
$
```

The server should accept data of any type (e.g. JSON, PNG, text, etc.):

```
$ curl -i -H "Content-Type: application/json" -X POST -d '{ "secret": 42 }'
http://localhost:8000/ firstResource
HTTP/1.1 200 OK
$
```

Once a resource exists, it should be returned by future requests:

```
$ curl -i http://localhost:8000/firstResource
HTTP/1.1 200 OK
Content-Type: application/json

{ "secret": 42 }
$
```

Other resources can be added:

```
$ curl -i -H "Content-Type: text/plain" -X POST -d 'Resource Data Payload'
http://localhost:8000/resource/number/2
HTTP/1.1 200 OK
$
```

And retrieved again:

```
$ curl -i http://localhost:8000/resource/number/2
HTTP/1.1 200 OK
Content-Type: text/plain
```

```
Resource Data Payload
$
```

Resources can be deleted:

```
$ curl -i -X DELETE http://localhost:8000/firstResource
HTTP/1.1 200 OK
$
```

And after they're deleted, they won't be shown in subsequent GET calls:

```
$ curl -i http://localhost:8000/firstResource
HTTP/1.1 404 Not Found
$
```

Unhandled requests should return appropriate errors:

```
$ curl -i -X PUT http://localhost:8000/otherResource
HTTP/1.1 405 Method Not Allowed
$
```

Extra Credit

This part is optional, so feel free to skip it. Modify your server to support non-destructive updates. Each POST or DELETE should cause a new version of the data store to be created and assigned an incrementing version number. The number for the newly-created version should be returned in a custom response header named X-Data-Version. GET requests should accept an optional version parameter indicating which version to query. If that parameter is not specified, the latest version is queried.

We initialize the data cache with a single entity:

```
$ curl -i -X POST -d "bar1" http://localhost:8000/foo
HTTP/1.1 200 OK
X-Data-Version: 1
$
```

And to retrieve it:

```
$ curl -i http://localhost:8000/foo
HTTP/1.1 200 OK

bar1
$
```

We update the same resource with new information:

```
$ curl -i -X POST -d "bar2" http://localhost:8000/foo
HTTP/1.1 200 OK
X-Data-Version: 2
$
```

We can receive the default value (the newest) or a specific version:

```
$ curl -i http://localhost:8000/foo
HTTP/1.1 200 OK

bar2
$ curl -i http://localhost:8000/foo?version=1
HTTP/1.1 200 OK

bar1
```

```
$ curl -i http://localhost:8000/foo?version=2
HTTP/1.1 200 OK

bar2
$
```

Sending a DELETE request increases the version of the data, but doesn't destroy the existing records:

```
$ curl -i -X DELETE http://localhost:8000/foo
HTTP/1.1 200 OK
X-Data-Version: 3
$
```

Trying to retrieve the default record (the newest) will fail because the latest request was a DELETE:

```
$ curl -i http://localhost:8000/foo
HTTP/1.1 404 Not Found
$
```

But the old versions still exist and can be retrieved:

```
$ curl -i http://localhost:8000/foo?version=2
HTTP/1.1 200 OK

bar2
$
```