

# Proiectarea și implementarea unui mini server web HTTP

Pericleanu Alexandra

Grupa C113D

Academia Tehnică Militară "Ferdinand I" —  
București, România  
alexandra.pericleanu@stud.mta.ro

Talpă Denisa

Grupa C113D

Academia Tehnică Militară "Ferdinand I" —  
București, România  
denisa-nicoleta.talpa@stud.mta.ro

**Rezumat**—Dezvoltarea aplicațiilor web reprezintă un element central al sistemelor informatici moderne. Lucrarea prezintă proiectarea și implementarea unui mini server web HTTP, realizat de la zero în limbajul C, cu accent pe comunicarea prin socket-uri, concurență, sincronizare și suport pentru conexiuni persistente.

**Index Terms**—server web, HTTP, socket-uri, thread pool, concurență, keep-alive

## I. INTRODUCERE

Dezvoltarea aplicațiilor web a devenit un element central al sistemelor informatici moderne, fiind utilizată într-o gamă largă de domenii, de la platforme de conținut și servicii online, până la aplicații distribuite complexe. La baza acestor aplicații se află protocoale standardizate de comunicare, dintre care protocolul HTTP (HyperText Transfer Protocol) joacă un rol esențial în schimbul de date dintre clienti și servere.

Înțelegerea în profunzime a modului de funcționare a unui server web, precum și a mecanismelor de comunicare dintre procese, fire de execuție și resursele sistemului de operare, reprezintă un obiectiv important în cadrul disciplinei Proiectarea Sistemelor de Operare. Din acest motiv, proiectul de față își propune realizarea unui mini server web HTTP, implementat de la zero în limbajul C, care evidențiază concepte fundamentale precum comunicarea prin socket-uri, concurență, sincronizarea thread-urilor și gestionarea resurselor.

Motivația alegerii acestei teme constă în dorința de aprofunda interacțiunea directă dintre un server și sistemul de operare, fără a apela la framework-uri sau biblioteci de nivel înalt. Prin implementarea manuală a unui server HTTP, se obține o înțelegere clară a modului în care sunt tratate cererile clientilor, cum sunt gestionate conexiunile multiple și cum sunt livrate resursele statice sau conținutul dinamic.

Proiectul abordează un scenariu practic de utilizare, și anume dezvoltarea unui mini blog web, care permite afișarea de conținut static (pagini HTML, fișiere CSS și imagini), precum și interacțiunea utilizatorilor prin trimiterea de comentarii folosind cereri de tip POST.

Obiectivele principale ale proiectului sunt:

- Proiectarea și implementarea unui server HTTP capabil să gestioneze conexiuni multiple simultan;
- Utilizarea unui mecanism de *thread pooling* pentru tratarea eficientă a clientilor;

- Implementarea metodelor HTTP de bază și a codurilor de stare corespunzătoare;
- Implementarea suportului pentru conexiuni persistente (*keep-alive*) și gestionarea controlată a durei acestora;
- Evidențierea aspectelor de securitate, validarea cererilor și protecția împotriva accesului neautorizat la resurse.

Prin atingerea acestor obiective, proiectul demonstrează aplicarea practică a conceptelor studiate în cadrul cursului și oferă o bază solidă pentru înțelegerea arhitecturii și funcționării serverelor web în contextul sistemelor de operare moderne.

## II. LUCRĂRI CONEXE

În literatura de specialitate și în practica industrială există numeroase implementări de servere web care au ca scop gestionarea eficientă a cererilor HTTP și a conexiunilor concurențe. Printre cele mai cunoscute exemple se numără Apache HTTP Server și Nginx, care oferă suport avansat pentru concurență, conexiuni persistente și optimizarea utilizării resurselor sistemului. Aceste soluții sunt concepute pentru a funcționa în medii de producție și utilizează arhitecturi complexe bazate pe procese multiple, thread-uri sau mecanisme asincrone.

Din punct de vedere educațional, există numeroase lucrări și proiecte care propun implementarea unui server HTTP minimal, cu scopul de a ilustra concepțele fundamentale ale comunicării prin socket-uri, gestionării conexiunilor TCP și sincronizării thread-urilor.

Un mecanism frecvent discutat în lucrările existente este menținerea conexiunilor persistente (HTTP *keep-alive*), introdusă pentru a reduce latența și costurile asociate stabilirii repetate a conexiunilor TCP. Implementările moderne tratează acest mecanism ca o responsabilitate a serverului, care trebuie să decidă durata de viață a conexiunii, numărul maxim de cereri acceptate și gestionarea timeout-urilor.

Comparativ cu soluțiile existente, proiectul de față adoptă o abordare simplificată, orientată spre înțelegerea mecanismelor interne ale unui server web. Prin implementarea manuală a suportului pentru *thread pooling* și conexiuni persistente, fără utilizarea unor biblioteci sau framework-uri de nivel înalt, se evidențiază în mod clar interacțiunea dintre aplicație și

sistemul de operare, precum și compromisurile realizate între performanță, complexitate și control asupra resurselor.

### III. ARHITECTURA ȘI DESIGNUL SISTEMULUI

Aplicația dezvoltată este un mini server web HTTP implementat în limbajul C, proiectat pentru a gestiona cereri concurente printr-o arhitectură de tip *thread pool*. Designul urmează un model *Producer-Consumer*, în care thread-ul principal acționează ca producător de conexiuni (acceptă clientii), iar un set fix de thread-uri *worker* acționează ca consumatori (preiau conexiunile și procesează cererile HTTP).

Comunicarea dintre producător și consumator este realizată prin intermediul unor cozi partajate, sincronizate cu mutex-uri și variabile de condiție, pentru a evita condițiile de cursă (*race conditions*) și pentru a elimina așteptarea activă (*busy waiting*).

#### A. Componente principale

1) *Modul de rețea (socket server)*: Serverul utilizează un socket TCP (AF\_INET, SOCK\_STREAM) legat la un port fix (8080). Thread-ul principal initializează socket-ul, efectuează operațiile bind() și listen(), apoi intră într-un ciclu infinit în care acceptă conexiuni prin accept().

Fiecare conexiune acceptată este reprezentată de un descriptor de socket (*file descriptor*), care este transferat către thread-urile worker pentru procesare.

2) *Coada de conexiuni cu priorități (myqueue)*: Pentru decuplarea dintre acceptarea conexiunilor și procesarea efectivă a cererilor, conexiunile sunt introduse într-o coadă partajată. În implementarea actuală, elementele din coadă includ pointerul către socket și nivelul de prioritate asociat cererii.

Thread-urile worker extrag elementele din coadă în ordine, permitând serverului să răspundă mai rapid cererilor considerate mai importante. De exemplu, cererile de tip POST pot fi tratate cu prioritate mai mare decât cererile GET.

3) *Sincronizare (mutex și variabilă de condiție)*: Accesul la coada partajată este protejat printr-un mutex, deoarece aceasta este utilizată simultan de mai multe thread-uri. Pentru a evita consumul inutile de CPU atunci când nu există cereri disponibile, thread-urile worker așteaptă pe o variabilă de condiție.

Astfel, atunci când coada este goală, worker-ul intră în stare de așteptare, iar thread-ul principal îl trezește în momentul adăugării unei conexiuni noi în coadă.

4) *Modul de parsare HTTP*: După preluarea unei conexiuni, worker-ul citește datele din socket și parsează cererea HTTP. Procesul de parsare extrage metoda (GET, POST, HEAD, OPTIONS), calea solicitată, versiunea protocolului și delimită zona de headere de corpul mesajului (*body*).

Această componentă permite validarea cererii și direcționarea acesteia către funcțiile de procesare specifice metodei utilizate.

5) *Modul de procesare a cererilor (request handlers)*: Pentru cererile de tip GET, serverul localizează resursa solicitată în directorul de conținut (de exemplu Blog/), determină tipul MIME în funcție de extensia fișierului și returnează conținutul

cu codul de stare 200 OK. În cazul în care fișierul nu există, este returnat codul de eroare 404 Not Found.

Pentru cererile de tip POST, serverul prelucrează datele din corpul cererii, decodează câmpurile transmise și salvează comentariile într-un fișier local.

6) *Modul de securitate și validare*: Pentru a preveni accesul neautorizat la fișiere din afara directorului aplicației, cererile sunt validate prin verificarea căii solicitate. Sunt blocate cazurile de tip directory traversal, iar cererile invalide sunt tratate prin trimiterea unui răspuns de eroare 400 Bad Request.

7) *Gestionarea conexiunilor persistente (keep-alive)*: Serverul include suport pentru conexiuni persistente, permitând același socket TCP să proceseze mai multe cereri HTTP successive. Pentru protocolul HTTP/1.0, menținerea conexiunii este activată doar dacă clientul trimit explicit headerul Connection: keep-alive.

În acest caz, worker-ul intră într-un ciclu de procesare repetată a cererilor, utilizând timeout-uri pentru închiderea controlată a conexiunilor inactive și o limită maximă de cereri per conexiune pentru a preveni monopolizarea resurselor.

#### B. Fluxul de procesare al unei cereri

Fluxul general de procesare a unei cereri HTTP poate fi descris prin următorii pași:

- 1) Thread-ul principal acceptă o conexiune nouă prin accept() și obține descriptorul de socket al clientului;
- 2) Conexiunea este introdusă în coada partajată împreună cu nivelul de prioritate;
- 3) Un thread worker se trezește (sau este deja activ), extrage conexiunea din coadă și o preia pentru procesare;
- 4) Worker-ul citește cererea HTTP, o parsează și identifică metoda, calea și versiunea protocolului;
- 5) Cererea este procesată prin handler-ul corespunzător (GET, POST, HEAD, OPTIONS), iar serverul trimită răspunsul HTTP (linie de status, headere și corp);
- 6) Dacă conexiunea este de tip keep-alive, worker-ul poate procesa cereri suplimentare pe același socket până la închiderea conexiunii de către client, expirarea timeout-ului sau atingerea limitei maxime de cereri. În caz contrar, conexiunea este închisă după trimiterea răspunsului.

## IV. IMPLEMENTARE

Implementarea mini serverului web a fost realizată integral în limbajul de programare C, utilizând apeluri de sistem specifice sistemelor de operare de tip Unix/Linux. Alegerea acestui limbaj a permis controlul direct asupra resurselor sistemului, precum socket-urile, thread-urile și mecanismele de sincronizare, fără a apela la biblioteci sau framework-uri de nivel înalt.

#### A. Limbaje și tehnologii utilizate

Serverul este implementat în limbajul C, folosind următoarele tehnologii:

- Biblioteca standard C (stdio.h, stdlib.h, string.h);

- Biblioteca POSIX pentru programare concurrentă (`pthread.h`);
- API-ul de retea POSIX (`sys/socket.h`, `unistd.h`);
- Mecanisme de sincronizare (`pthread_mutex_t`, `pthread_cond_t`).

Aplicația rulează pe un sistem de operare Linux și respectă modelul de programare POSIX pentru gestionarea proceselor și fiilor de execuție.

#### B. Structura proiectului

Codul sursă este organizat modular, pentru a separa responsabilitățile principale ale serverului:

- Modulul principal: inițializează socket-ul serverului, creează thread pool-ul și acceptă conexiuni;
- Modulul coadă (`myqueue`): gestionează conexiunile acceptate și prioritatea acestora;
- Modulul de parsare HTTP: extrage informațiile relevante din cererea clientului (metodă, cale, protocol, headere, body);
- Modulele de procesare GET / POST / HEAD / OPTIONS: tratează cererile și generează răspunsuri HTTP corespunzătoare;
- Modulul de logging: înregistrează activitatea serverului (cereri, răspunsuri, închiderea conexiunilor).

Această separare îmbunătățește lizibilitatea codului și facilitează extinderea ulterioară a aplicației.

#### C. Protocolul de comunicare

Comunicarea între client și server se realizează prin intermediul protocolului HTTP peste TCP. Serverul ascultă pe un port configurabil (de exemplu 8080) și procesează cereri HTTP de tip GET, POST, HEAD și OPTIONS.

Pentru fiecare cerere, serverul construiește manual răspunsul HTTP, inclusivând:

- Linia de status (de exemplu HTTP/1.0 200 OK);
- Headerele necesare (`Content-Type`, `Content-Length`, `Connection`);
- Corpul răspunsului (conținut HTML, CSS, imagini sau mesaje text).

#### D. Parsarea cererilor HTTP

Datele primite prin socket sunt citite sub forma unui flux de octeți, specific protocolului TCP. Pentru a gestiona corect cereri de dimensiuni variabile, serverul utilizează o funcție dedicată pentru citirea completă a unei cereri HTTP, până la delimitarea dintre headere și body și, în cazul cererilor de tip POST, până la citirea integrală a conținutului specificat de `Content-Length`.

Ulterior, cererea este analizată printr-o structură dedicată (`http_request_t`), care stocăază:

- Metoda HTTP;
- Calea solicitată;
- Versiunea protocolului;
- Pointeri către zona de headere și body.

Această abordare permite separarea clară între faza de citire a datelor și faza de procesare logică a cererii.

#### E. Procesarea cererilor GET, POST, HEAD și OPTIONS

Pentru cererile de tip GET, serverul caută resursa solicitată în directorul de conținut static. În funcție de extensia fișierului, este determinat tipul MIME corespunzător (HTML, CSS, JPEG etc.), iar conținutul este transmis către client. În cazul în care resursa nu există, serverul returnează codul de eroare 404 Not Found.

Cerurile de tip POST sunt utilizate pentru trimiterea de date de la client către server, în special pentru functionalitatea de comentarii din cadrul mini blogului. Datele sunt decodeate și salvate într-un fișier local, simulând un mecanism simplu de persistență.

Metoda HTTP HEAD este implementată pentru a permite clientilor să obțină metadatele unei resurse fără a primi corpul acesta. Această metodă este utilă în scenarii precum verificarea existenței unei resurse, obținerea dimensiunii unui fișier sau validarea tipului MIME, fără a transfera efectiv conținutul. În cadrul serverului dezvoltat, cererile HEAD sunt procesate prin reutilizarea logicii de tratare a cererilor GET, cu diferența că răspunsul HTTP conține doar linia de status și headerle, fără transmiterea corpului mesajului. Serverul determină tipul MIME și dimensiunea resursei, construind corect headerul `Content-Length`, conform specificației HTTP.

Metoda OPTIONS este utilizată pentru a determina capabilitățile serverului pentru o anumită resursă. În cadrul mini serverului web, această metodă este implementată pentru a răspunde cu lista metodelor HTTP suportate. La primirea unei cereri OPTIONS, serverul generează un răspuns HTTP cu codul de stare 204 No Content sau 200 OK și include headerul `Allow`, care specifică metodele acceptate, precum:

GET, POST, HEAD, OPTIONS

#### F. Implementarea conexiunilor persistente (HTTP Keep-Alive)

Serverul include suport pentru conexiuni persistente (*keep-alive*), conform specificației HTTP. În cazul cererilor HTTP/1.0, menținerea conexiunii este realizată doar dacă clientul trimită explicit headerul `Connection: keep-alive`.

Atunci când mecanismul *keep-alive* este activ, serverul nu închide conexiunea după procesarea primei cereri, ci permite procesarea mai multor cereri successive pe același socket TCP. Pentru a preveni utilizarea excesivă a resurselor, sunt implementate:

- Un timeout de inactivitate, după care conexiunea este închisă automat;
- O limită maximă de cereri per conexiune, pentru a evita monopolizarea thread-urilor.

#### G. Concurrentă și sincronizare

Gestionarea concurrentă a cererilor este realizată printr-un thread pool cu un număr fix de thread-uri worker. Threadul principal acceptă conexiuni și le introduce într-o coadă partajată, iar workerii preiau și procesează conexiunile în mod paralel.

Accesul la coadă este sincronizat printr-un mutex, iar thread-urile inactive așteaptă pe o variabilă de condiție, eliminând consumul inutil de CPU. Această abordare permite

scalarea eficientă a serverului pentru un număr mare de clienți simultani.

#### H. Aspecte de securitate și validare

Serverul validează căile solicitate pentru a preveni accesul la fișiere din afara directorului aplicației. Sunt blocate cererile care conțin secvențe de tip *directory traversal* sau caractere nepermise. În cazul cererilor invalide, serverul răspunde cu coduri de eroare corespunzătoare (400 Bad Request, 404 Not Found, 501 Not Implemented).

#### I. Autentificarea utilizatorilor (Signup și Login)

Pentru a permite interacțiunea controlată a utilizatorilor cu aplicația de tip mini blog, serverul include un mecanism simplu de autentificare, bazat pe crearea și validarea conturilor de utilizator. Acest mecanism este implementat fără utilizarea unor biblioteci externe, pentru a evidenția procesarea manuală a datelor primite prin cereri HTTP.

Crearea unui cont (signup) se realizează printr-o cerere HTTP de tip POST, care conține câmpurile `username` și `password`, codificate sub forma `application/x-www-form-urlencoded`. Serverul decodează datele primite, validează unicitatea numelui de utilizator și salvează noile credențiale într-un fișier local (`users.txt`), simulând un mecanism de persistență.

Autentificarea (login) se realizează tot printr-o cerere POST, în urma căreia serverul verifică datele introduse prin compararea acestora cu înregistrările existente în fișierul de utilizatori. În cazul autentificării reușite, serverul inițiază o sesiune pentru utilizatorul respectiv.

#### J. Gestionarea sesiunilor și utilizarea cookie-urilor

Pentru a menține starea de autentificare între cereri HTTP succesive, serverul utilizează mecanismul de sesiuni bazate pe cookie-uri. După autentificarea cu succes a unui utilizator, serverul generează un identificator de sesiune unic și îl transmite clientului prin headerul `Set-Cookie`.

Cookie-ul de sesiune este retransmis automat de client la fiecare cerere ulterioară, permitând serverului să identifice utilizatorul asociat conexiunii. Pe baza acestui mecanism, accesul la anumite funcționalități, precum trimiterea comentariilor, este restrictionat doar utilizatorilor autentificați.

La deconectare (logout), serverul invalidează sesiunea și trimit un cookie expirat către client, asigurând încheierea corectă a sesiunii de autentificare.

#### K. Controlul accesului și validarea utilizatorilor

Serverul aplică reguli de control al accesului pentru a preveni utilizarea neautorizată a funcționalităților aplicației. Cерерile de tip POST destinate trimiterii comentariilor sunt acceptate doar dacă utilizatorul este autentificat, identificat printr-o sesiune validă.

În cazul în care un client neautentificat încearcă să acceseze aceste resurse, serverul răspunde cu coduri de stare corespunzătoare, precum 401 Unauthorized sau redirecționează utilizatorul către pagina de autentificare.

Această abordare separă clar zona publică a aplicației de zona protejată, contribuind la securitatea și coerenta funcțională a mini blogului.

#### L. Persistența comentariilor

Comentariile trimise de utilizatori autentificați sunt procesate prin cereri POST și sunt salvate într-un fișier local, împreună cu numele utilizatorului și conținutul mesajului. Această soluție simplificată permite păstrarea comentariilor între reporniri ale serverului, fără a necesita o bază de date.

La accesarea paginii blogului, comentariile sunt încărcate și livrate către client printr-un endpoint de tip API, sub forma unui răspuns JSON. Această separare între continutul static și datele dinamice permite o interfață web interactivă și extensibilă.

#### M. Interfața web și integrarea client-server

Interfața web a aplicației este realizată folosind HTML, CSS și JavaScript și comunică cu serverul exclusiv prin cereri HTTP. JavaScript-ul rulează în browser și utilizează API-ul `fetch()` pentru a interacționa cu endpoint-urile serverului, inclusiv autentificarea utilizatorilor, încărcarea comentariilor și trimiterea de date.

### V. TESTARE ȘI VALIDARE

Testarea serverului web a fost realizată prin rularea directă a comenzi în terminal, utilizând utilitarul `curl` și mecanisme TCP standard. Această abordare permite validarea comportamentului serverului în condiții reale de utilizare și evidențierea clară a conformității cu specificația HTTP.

#### A. Testare de securitate – protecție împotriva directory traversal

Pentru a verifica faptul că serverul restricționează accesul la fișiere din afara directorului aplicației, au fost rulate următoarele comenzi:

```
curl -i "http://localhost:8080/.../server.log"
curl -i
  "http://localhost:8080/%2e%2e/server.log"
curl -i
  "http://localhost:8080/%2e%2e/%2e%2e/etc/
passwd"
```

#### B. Testarea metodei GET

Metoda GET a fost testată prin solicitarea unei resurse statice:

```
curl -i --http1.0 \
-H "Connection: close" \
http://localhost:8080/cats.html
```

Rezultatul obținut în urma testării metodei GET:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 10794
Connection: close
```

urmat de codul html aferent paginii cerute.

Verificare conexiune keep-alive folosind un GET

```
curl -v http://10.156.27.19:8080/
http://10.156.27.19:8080/cats.html
```

#### C. Testarea metodei POST

Metoda POST a fost utilizată pentru a testa trimiterea datelor către server și procesarea comentariilor:

```
curl -i --http1.0 \
-H "Connection: keep-alive" \
-H "Content-Type:
application/x-www-form-urlencoded" \
-d "username=Ana&message=Primul" \
http://localhost:8080/
```

Rezultatul obținut în urma testării metodei POST:

```
HTTP/1.1 200 OK
Content-Type: text/plain
Content-Length: 2
Connection: keep-alive
Keep-Alive: timeout=5, max=50
```

#### D. Testarea metodei HEAD

Implementarea metodei HEAD a fost testată prin solicitarea unei resurse existente:

```
curl -I http://localhost:8080/cats.html
```

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 10794
Connection: keep-alive
Keep-Alive: timeout=5, max=50
```

și a unei resurse inexistente:

```
curl -I http://localhost:8080/nuexista.html
```

```
HTTP/1.1 404 Not Found
Content-Type: text/html
Content-Length: 201
Connection: close
```

Pentru validarea conformității cu specificația HTTP, valorile câmpului Content-Length obținute prin metodele GET și HEAD au fost comparate:

```
curl -s -D - http://localhost:8080/cats.html
-o /dev/null | tr -d '\r' | grep -i
'^Content-Length:'
curl -s -I http://localhost:8080/cats.html |
tr -d '\r' | grep -i '^Content-Length:'
```

Alternativ, comparația automată a fost realizată prin următorul script:

```
get=$(curl -s -D -
http://localhost:8080/cats.html -o
/dev/null | tr -d '\r' | awk -F':'
'^Content-Length:/ {print $2; exit}')
head=$(curl -s -I
http://localhost:8080/cats.html | tr -d
'\r' | awk -F':'
'^Content-Length:/ {print $2; exit}')
```

```
echo "GET: $get"
echo "HEAD: $head"
if [ -z "$get" ] || [ -z "$head" ]; then
  echo "Eroare: lipseste Content-Length la
unul din raspunsuri"
elif [ "$get" = "$head" ]; then
  echo "OK: Content-Length sunt egale"
else
  echo "FAIL: Content-Length difera"
fi
```

Valorile obținute au fost identice, confirmând implementarea corectă a metodei HEAD.

#### E. Testarea metodei OPTIONS

Metoda OPTIONS a fost testată pentru a verifica metodele HTTP acceptate de server:

```
curl -i -X OPTIONS
http://localhost:8080/cats.html
```

```
HTTP/1.1 204 No Content
Allow: GET, POST, HEAD, OPTIONS
Connection: keep-alive
Keep-Alive: timeout=5, max=50
```

Răspunsul serverului a inclus headerul Allow, indicând metodele suportate.

## VI. CONCLUZII ȘI PERSPECTIVE

Proiectul a demonstrat că implementarea unui server web HTTP funcțional, folosind exclusiv limbajul C și apeluri de sistem POSIX, este realizabilă și oferă o înțelegere aprofundată a mecanismelor interne ale aplicațiilor de tip server.

Principalele rezultate obținute includ implementarea concurenței prin thread pool, suportul pentru conexiuni persistente, procesarea completă a metodelor HTTP de bază și integrarea unui mecanism simplu de autentificare și persistentă a datelor.

Dificultățile întâmpinate au fost legate în principal de pararea corectă a cererilor HTTP, sincronizarea thread-urilor și gestionarea stării aplicației într-un protocol inherent fără stare.

Din perspectivă critică, arhitectura aleasă oferă un control bun asupra resurselor, însă presupune o complexitate crescută a codului și un efort mai mare de mențenanță comparativ cu soluțiile bazate pe framework-uri moderne.

## VII. BIBLIOGRAFIE

### BIBLIOGRAFIE

- [1] <https://www.garshol.priv.no/download/text/http-tut.html>
- [2] <https://medium.com/@justup1080/tutorial-creating-a-minimalist-http-server-in-c-2303d140c725>
- [3] <https://curl.se/docs/httppscripting.html>
- [4] <https://curl.se/docs/tutorial.html>
- [5] <https://antonz.org/curl-by-example/>
- [6] [https://en.wikipedia.org/wiki/HTTP\\_cookie](https://en.wikipedia.org/wiki/HTTP_cookie)
- [7] <https://medium.com/@harshanacz/web-authentication-101-full-breakdown-75b3e6a605c5>
- [8] <https://beej.us/guide/bgnet/>