Wybrane elementy praktyki projektowania oprogramowania
Wykład 07/15
node.js: HTTP, HTML

Wiktor Zychla 2018/2019

1 Spis treści

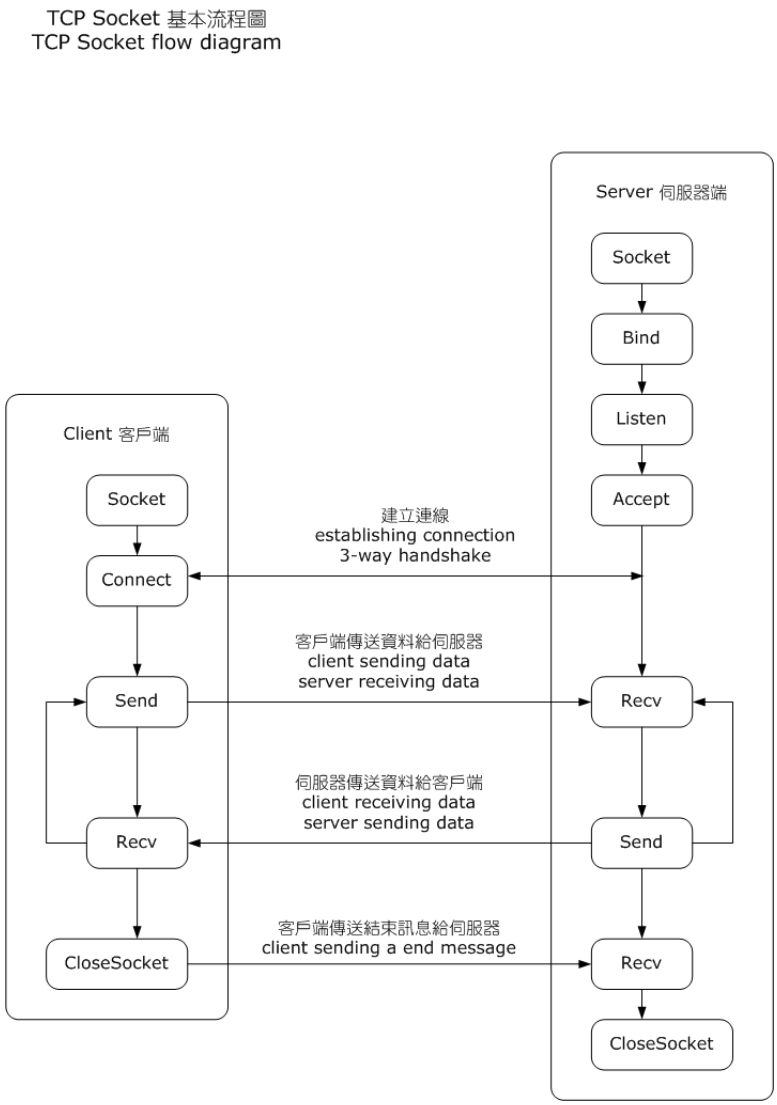
2 HTTP

2.1 Protokół

Protokół HTTP opracowano na początku lat 90tych. Jest to oparty o TCP/IP protokół komunikacyjny, zaprojektowany do przekazywania treści multimedialnych.

Przez oparty o TCP/IP rozumiemy tu protokół, w warstwie transportowej którego używane są gniazda BSD, czyli standard z początku lat 80tych, zaadoptowany do POSIX jako de facto interoperacyjny standard komunikacyjny. Interfejs gniazd przyjął się w dużej mierze dzięki swojej prostocie konceptualnej, opartej o funkcje

- **bind**, **listen** i **accept** na serwerze
- **connect** na kliencie
- **send** i **recv** na kliencie i serwerze



Rysunek 1 https://en.wikipedia.org/wiki/Berkeley_sockets#/media/File:InternetSocketBasicDiagram_zhtw.png

Jednym z podstawowych problemów do rozwiązania przy projektowaniu protokołów komunikacyjnych opartych o interfejs gniazd jest ustalenie konwencji, zgodnie z którą uczestnicy komunikacji mieliby wiedzieć jak wyznaczane są granice komunikacji (czyli kiedy wołać **send** a kiedy **recv** i z jakimi argumentami określającymi długość odczytywanego bufora).

Protokół HTTP rozwiązuje ten problem w sposób następujący:

2.1.1 Żądanie

Żądanie HTTP to tekst złożony z wielu linii, z których pierwsza określa rodzaj żądania i nazwę zasobu, a kolejne linie oznaczają tzw. nagłówki HTTP. Pusta linia oznacza koniec żądania, chyba że jest to żądanie z parametrami (np. POST czy PUT), wtedy treść żądania może być dowolnie długa. W takim przypadku obowiązkowy jest nagłówek Content-length, który określa długość tej dodatkowej opcjonalnej części.

2.1.2 Odpowiedź

Odpowiedź HTTP to tekst lub strumień binarny, interpretowany przez klienta na podstawie opisu w nagłówku Content-type, o długości zadanej przez nagłówek Content-length.

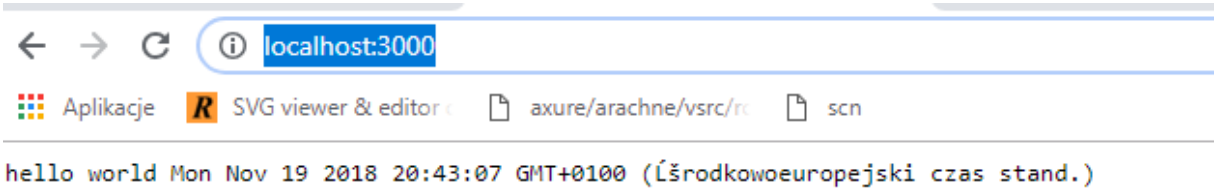
2.2 Najprostszy serwer HTTP w node.js

Node.js posiada wsparcie dla tworzenia serwerów HTTP bezpośrednio w bibliotece standardowej

```
var http = require('http');

var server =
    http.createServer(
    (req, res) => {
        res.end(`hello world ${new Date()}`);
    });

server.listen(3000);
console.log('started');
```



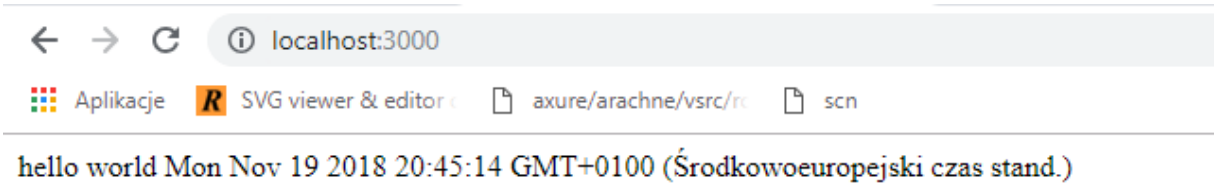
Tę niefortunną prezentację można skorygować wymuszając interpretowanie odpowiedzi w określonym kodowaniu (lub dodatkowo: typie), np.:

```
var http = require('http');

var server =
    http.createServer(
    (req, res) => {
        res.setHeader('Content-type', 'text/html; charset=utf-8');
        res.end(`hello world ${new Date()}`);
```

```
    });

server.listen(3000);
console.log('started');
```



Argumentem funkcji tworzącej serwer jest callback, przyjmujący dwa argumenty:

- Obiekt **req** typu **IncomingMessage** który jest modelem dla żądania od klienta. Z żądania można odczytać m.in. adres i typ żądania, nagłówki (w tym ciasteczka)
- Obiekt **res** typu **ServerResponse** który jest modelem dla odpowiedzi do klienta. Do odpowiedzi można skierować dowolny strumień bajtów, zgodnie z regułami protokołu

Klikając w którykolwiek w edytorze kodu prawym przyciskiem i przywołując funkcję „Go to type definition", można przejrzeć interfejs obu typów.

Callbacki kolejnych żądań trafiają do pętli zdarzeń i są obsługiwane *jednowątkowo*. Zbudowanie wielowątkowego rozwiązania wymaga użycia jawnie modułu cluster. Taka charakterystyka pracy różni się diametralnie od podejścia w innych technologiach, w których dla każdego przychodzącego żądania tworzony jest nowy wątek. Przy dużym ruchu skalowalność takiego podejścia jest zwykle *gorsza* (systemowi więcej czasu zajmuje przełączanie się między zbyt dużą liczbą wątków niż po prostu trzymanie długiej listy callbacków i obsługiwanie ich po kolei – długość kolejki nie wpływa bowiem na przetwarzanie żądań).

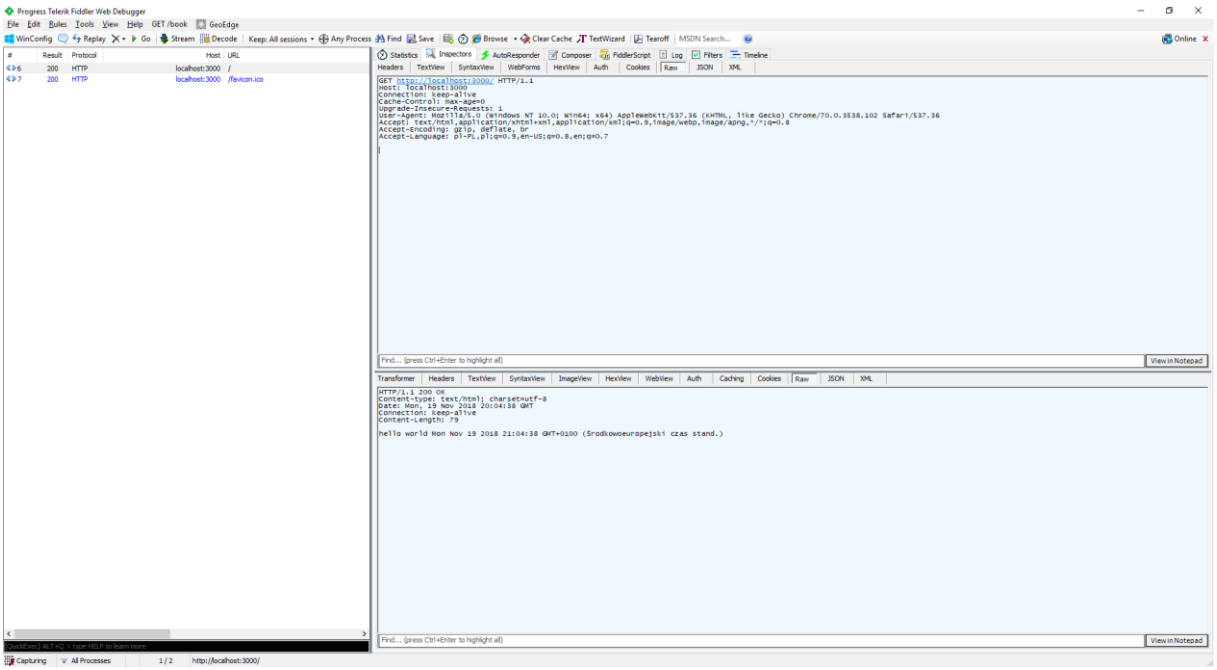
Stąd stosunkowo duża popularność środowisk opartych o libuv

2.3 Debuggery HTTP

Podgląd ruchu między klientem a serwerem możliwy jest albo za pomocą narzędzi niskopoziomowych (Wireshark) albo za pomocą tzw. debuggerów http, wykorzystujących fakt że przeglądarki pozwalają na ustawienie tzw. proxy dla ruchu HTTP

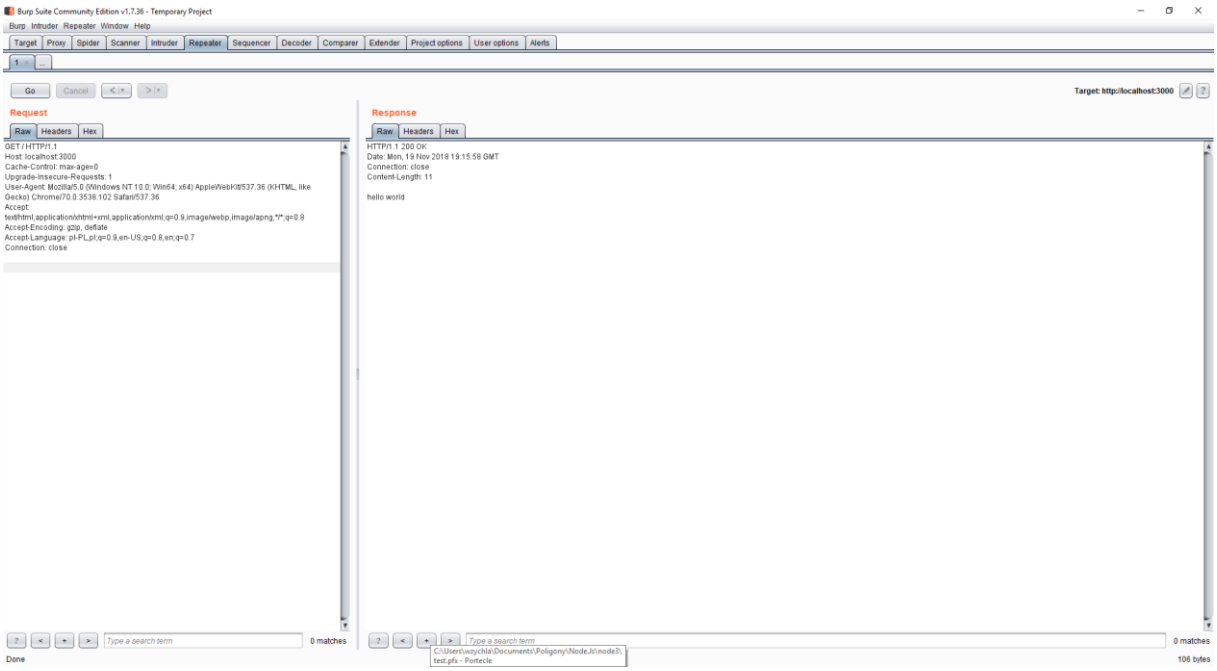
2.3.1 Fiddler

Fiddler jest darmowym debuggerem http od Telerik.



2.3.2 Burp

Burp jest darmowym debuggerem od PortSwigger Ltd.



2.3.3 Tworzenie własnego żądania

Oba narzędzia pozwalają na tworzenie własnych żądań – kopiowanie i modyfikowanie istniejących i/lub tworzenie żądań „od zera”.

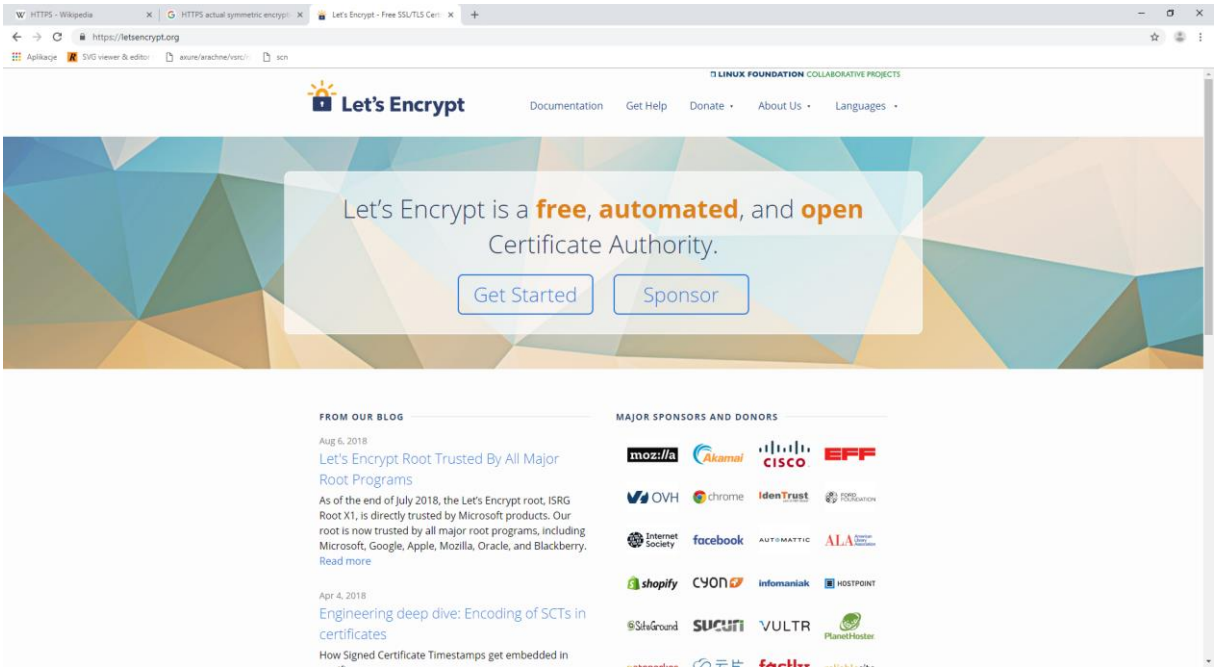
2.4 HTTPs

HTTPs to rozszerzenie protokołu HTTP w którym pod żądaniem typu CONNECT pojawia się dodatkowy tzw. handshake czyli sekwencja komunikatów zmierzająca do ustalenia klucza szyfrującego komunikację.

Ta część komunikacji wykorzystuje algorytm asymetryczny, bo klucz prywatny serwera, służący do szyfrowania jego części komunikacji, może być przez klienta zweryfikowany za pomocą klucza publicznego (certyfikatu), w ten sposób poświadczając autentyczność serwera. Sama wymiana klucza możliwa jest w sposób bezpieczny za pomocą algorytmu Diffiego-Hellmana. Po ustaleniu klucza i zweryfikowaniu autentyczności serwera, komunikacja odbywa się z wykorzystaniem tańszych algorytmów szyfrowania symetrycznego.

2.4.1 Portecle

Portecle to przykładowe narzędzie do generowania cerytfikatu. Przemysłowe certyfikaty, mające zaufane organizacje na szczycie łańcucha poświadczeń można kupić lub pozyskać bezpłatnie np. od Let's Encrypt.

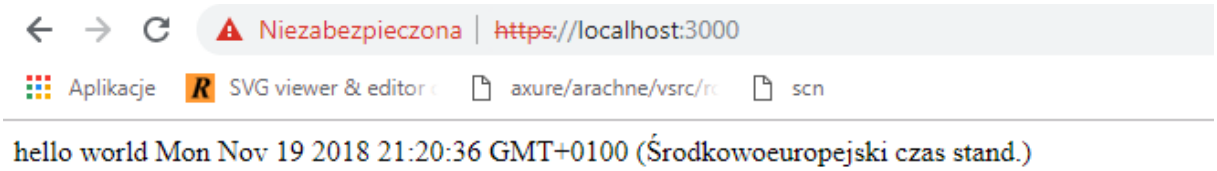


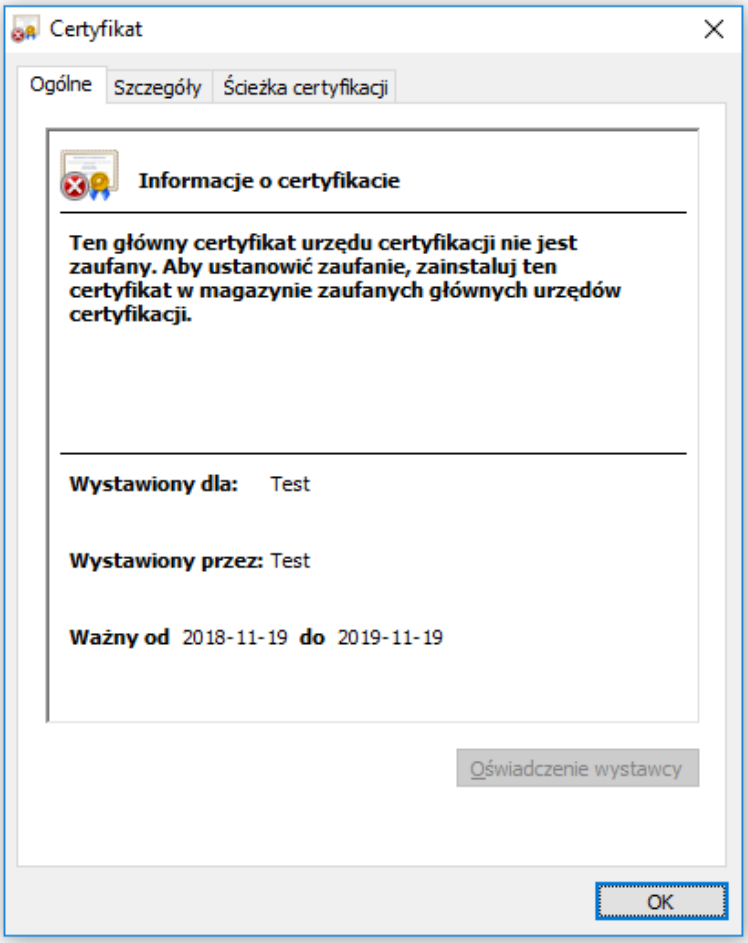
2.4.2 Najprostszy serwer HTTPs w node.js

```
var fs = require('fs');
var https = require('https');

(async function () {
    var pfx = await fs.promises.readFile('test.pfx');
    var server = https.createServer({
            pfx: pfx,
            passphrase: 'test'
        },
        (req, res) => {
            res.setHeader('Content-type', 'text/html; charset=utf-8');
            res.end(`hello world ${new Date()}`);
        });

    server.listen(3000);
    console.log('started');
})();
```





2.5 HTTP/2

HTTP/2 rozszerza protokół http o optymalizacje w warstwie transportowej:

- Otwiera zawsze jedno gniazdo miedzy klientem a serwerem
- Ruch jest kompresowany, binarny
- Możliwe jest zrównoleglanie wielu żądań i wielu odpowiedzi
- Możliwe jest przesyłanie danych z serwera do klienta **zanim** klient zrobi żądanie

```
var fs = require('fs');
var http2 = require('http2');

(async function () {
    var pfx = await fs.promises.readFile('test.pfx');
    var server = http2.createSecureServer({
            pfx: pfx,
            passphrase: 'test'
```

```
        });
    server.on('stream',
        (stream, headers) => {
            stream.respond({
                'content-type': 'text/html',
                ':status': 200
              });
            stream.end(`hello world ${new Date()}`);
        });

    server.listen(3000);
    console.log('started');
})();
```

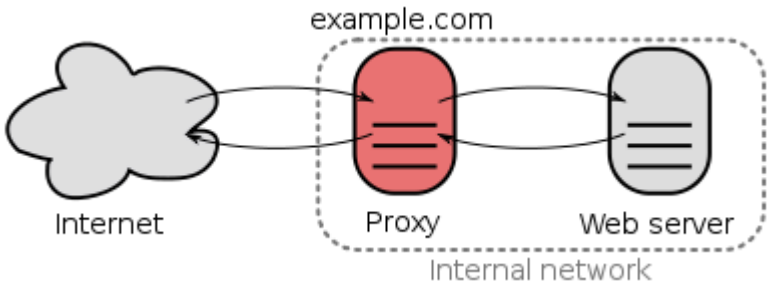
2.6 node.js a rzeczywistość

2.6.1 etc/hosts i lokalne mapy DNS

W pliku etc/host możliwe jest mapowanie DNS na potrzeby lokalnej maszyny. W praktyce umożliwia to zamapowanie na lokalny interfejs sieciowy 127.0.0.X dowolnego nagłówka hosta, czyli testowanie lokalnie witryny produkcyjnej.

2.6.2 Architektra rozwiązania serwerowego

Rzeczywista architektura aplikacji internetowych po stronie serwera wykorzystuje co najmniej dwie warstwy serwerów i tzw. mechanizm reverse proxy, w którym ruch trafia do warstwy serwerów frontowych na portach 80/443, a stamtąd do serwerów backendowych, schowanych wewnątrz sieci dostawcy.



Rysunek 2 https://en.wikipedia.org/wiki/Reverse_proxy#/media/File:Reverse_proxy_h2g2bob.svg

3 HTML

3.1 Odczyt pliku statycznego

Zamiast wysyłać klientowi tekst można odesłać „stronę”:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
```

```
<body>
    <div>
        Hello world!
    </div>
</body>
</html>
```

```
var fs = require('fs');
var http = require('http');

(async function () {
    var html = await fs.promises.readFile('test.html', 'utf-8');
    var server = http.createServer(
        (req, res) => {
            res.setHeader('Content-type', 'text/html; charset=utf-8');
            res.end(html);
        });

    server.listen(3000);
    console.log('started');
})();
```

3.2 Formularz, HTTP POST

Komunikacja zwrotna klienta z serwerem możliwa jest dzięki formularzom. Warto zwrócić uwagę na możliwą zmianę sposobu przesyłania formularza (GET vs POST).

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <form method="POST" enctype="application/x-www-form-urlencoded">
        <div>
            Login: <input type='text' name='login' /><br/>
            <button>Login</button>
        </div>
    </form>
</body>
</html>
```

Odsyłając formularz na serwer natykamy się ważne wyzwanie – bezstanowość protokołu HTTP.

3.3 Podtrzymanie stanu

Podtrzymanie stanu http możliwe jest tylko wtedy kiedy jest jawnie oprogramowane.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>Document</title>
</head>
<body>
    <form method="POST" enctype="application/x-www-form-urlencoded">
        <div>
            Login: <input type='text' name='login' value="{{name}}"/><br/>
            <button>Login</button>
        </div>
    </form>
</body>
</html>
```

```
var fs = require('fs');
var http = require('http');

(async function () {
    var html = await fs.promises.readFile('test.html', 'utf-8');
    var server = http.createServer(
        (req, res) => {
            res.setHeader('Content-type', 'text/html; charset=utf-8');
            if ( req.method == 'GET' ) {
                res.end(html.replace("{{name}}", ''));
            } else {

                var postdata = '';

                req.on('data', function(data) { postdata += data });
                req.on('end', () => {

                    // w body jest komplet zapostowanych
```

```
                    // danych w postaci klucz=wartosc&klucz2=wartosc2
                    // można np. zamienic na obiekt
                    // { klucz:wartosc, klucz2: wartosc2 } (reduce!)
                    var body =
                        postdata
                            .split('&')
                            .map( kv => kv.split('=') )
                            .reduce( (o, [k,v]) => ( o[k]=v, o ), {});

                    res.end( html.replace('{{name}}', body.login));
                });
            }
        });

    server.listen(3000);
    console.log('started');
})();
```

Taka kultura pracy w której implementacja funkcji obsługi żądania to jeden wielki **if**, w dodatku konieczna jest jawna, żmudna implementacja odczytywania nadesłanych danych, jest dalece niezadowalająca.

Użycie modułu **querystring**

```
                req.on('data', function(data) { postdata += data });
                req.on('end', () => {

                    var body =
                        qs.parse(postdata);

                    res.end( html.replace('{{name}}', body.login));
                });
```

rozwiązuje tylko niewielką część problemu.

Dlatego na kolejnych wykładach poznamy framework Express.