

# Wybrane elementy praktyki projektowania oprogramowania

## Wykład 07/15

### node.js: HTTP, HTML

Wiktor Zychla 2023/2024

---

#### 1 Spis treści

2	HTTP .....	2
2.1	Protokół .....	2
2.1.1	Żądanie .....	3
2.1.2	Odpowiedź.....	3
2.2	Klienci i debuggery HTTP .....	3
2.2.1	Fiddler .....	4
2.2.2	Burp .....	5
2.2.3	Tworzenie własnego żądania.....	5
2.3	Najprostszy serwer HTTP w node.js .....	5
2.4	HTTPs .....	9
2.4.1	Generowanie certyfikatów .....	9
2.4.2	Najprostszy serwer HTTPs w node.js.....	9
2.5	HTTP/2 .....	10
2.6	node.js a rzeczywistość .....	11
2.6.1	etc/hosts i lokalne mapy DNS.....	11
2.6.2	Architektura rozwiązania serwerowego .....	11
3	HTML .....	12
3.1	Odczyt pliku statycznego .....	12
3.2	Formularz, HTTP POST .....	12
3.3	Podtrzymanie stanu.....	13

## 2 HTTP

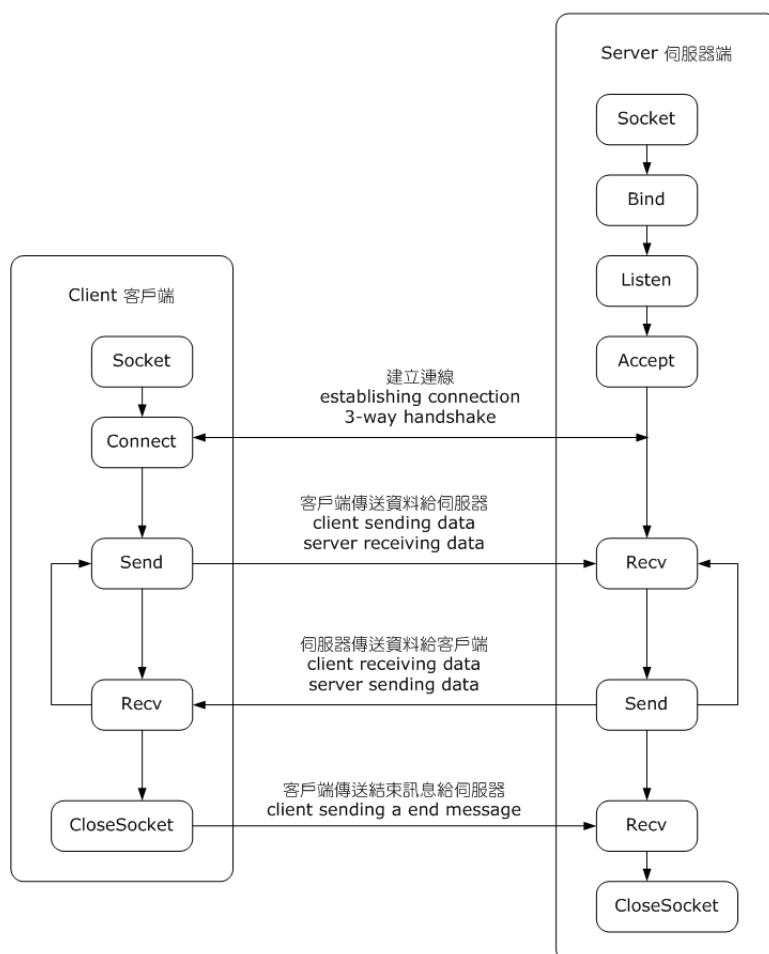
### 2.1 Protokół

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

TCP Socket 基本流程圖  
TCP Socket flow diagram



Rysunek 1 [https://en.wikipedia.org/wiki/Berkeley\\_sockets#/media/File:InternetSocketBasicDiagram\\_zhtw.png](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 Klienci i debuggery HTTP

Sama przeglądarka internetowa może posiadać funkcję podglądu ruchu sieciowego (m.in. zakładka Network w Chrome), ponieważ podgląd odbywa się w samej przeglądarce, nie ma problemu z podglądem ruchu szyfrowanego (HTTPS).

Zewnętrzny 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. Takie narzędzia mają więcej możliwości niż podgląd wbudowany w przeglądarkę (np. proxy może znajdować się fizycznie na innej maszynie).

Jak narzędzie rozwiązuje problem podglądu ruchu HTTPS? Czy to w ogóle możliwe? Sprawdźmy w trakcie wykładu podglądając ruch przykładowych witryn

- <https://reqbin.com/> - witryna z mechanizmami śledzenia komunikacji do innych witryn
- <https://httpbin.org/> - witryna do której można kierować żądania ([opis](#))

Przykładowy klient wykonujący POST do ścieżki <https://httpbin.org/post>

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

function promisedRequest() {
  return new Promise(function (resolve, reject) {

    var qs = new URLSearchParams();
    qs.append('foo', 'foo');
    qs.append('bar', 'bar');
    var postData = qs.toString();

    /**
     * @type {http.RequestOptions}
     */
    var requestParameters = {
      host: 'httpbin.org',
```

```

        port: '443',
        path: '/post',
        method: 'POST',
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded',
            'Content-Length': Buffer.byteLength(postData)
        }
    };

    var client = https.request(requestParameters, function (res) {

        var buffer = '';

        res
            .on('data', function (data) {
                buffer += data.toString();
            })
            .on('end', function () {
                resolve(buffer);
            });

        client.write( postData );
        client.end();

    });
}

(async function() {

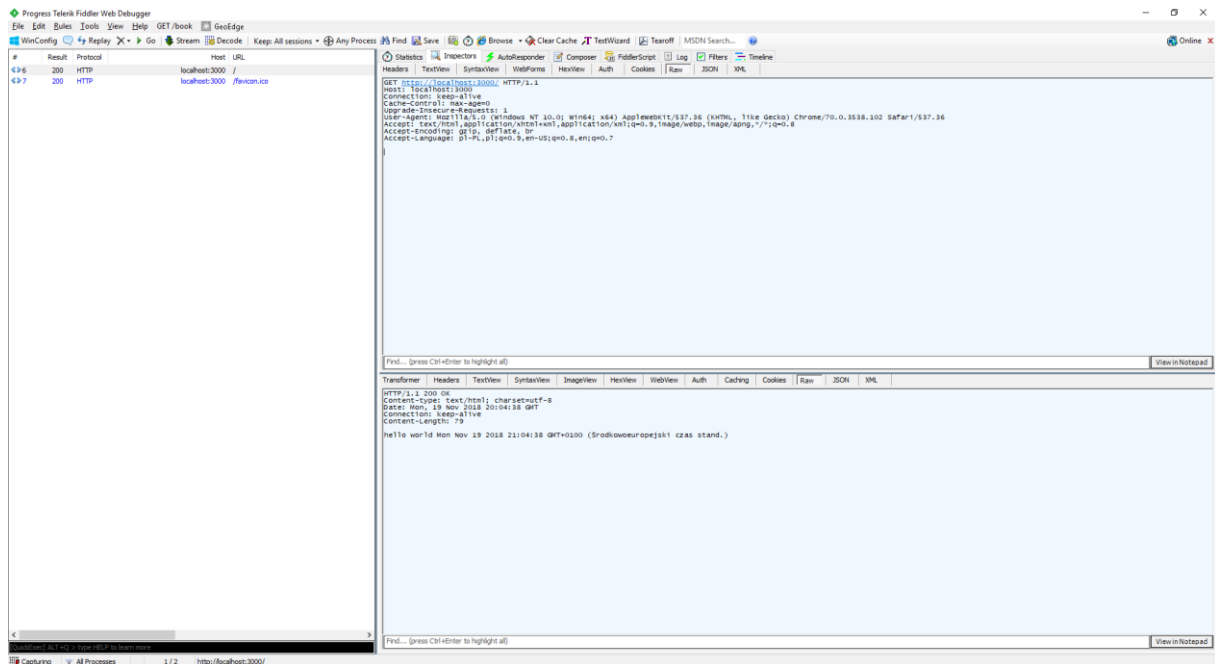
    var result = await promisedRequest();
    console.log( result );

})();

```

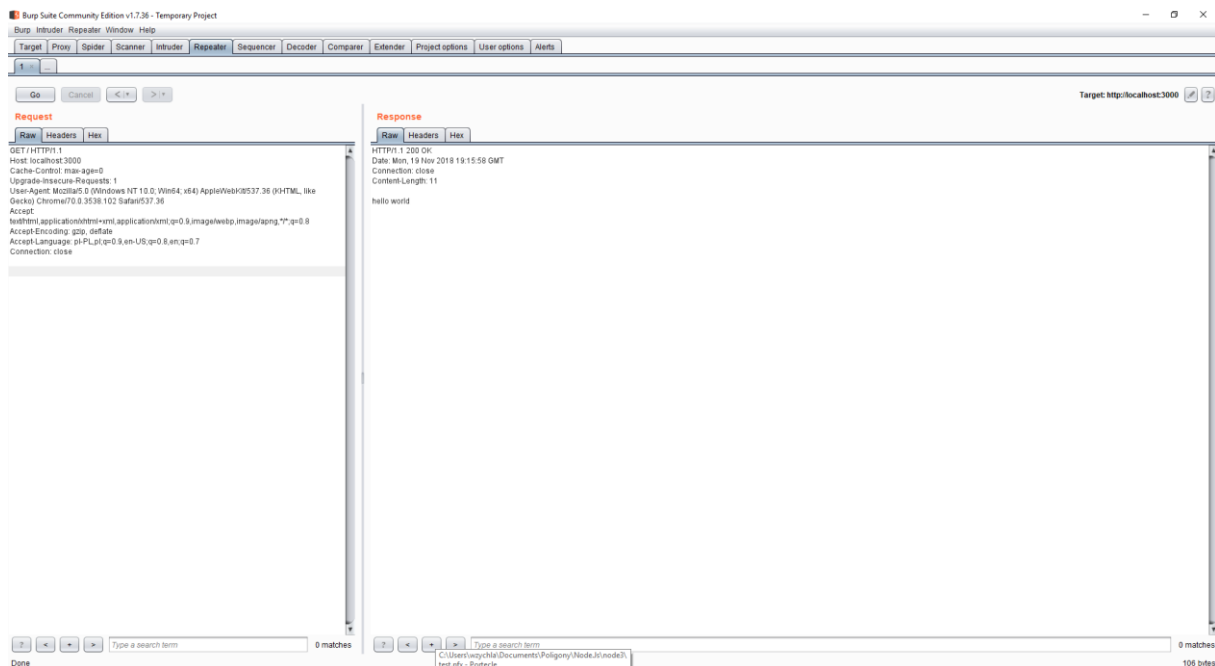
### 2.2.1 Fiddler

Fiddler jest darmowym debuggerem http od Telerik. W wersji Fiddler Classic – tylko dla Windows, w nowszej wersji Fiddler Everywhere – dla Windows/Linux/macOS.



## 2.2.2 Burp

Burp jest darmowym debuggerem od PortSwigger Ltd. Napisany jest w Javie i działa w większości dużych systemów operacyjnych.



## 2.2.3 Tworzenie własnego żądania

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

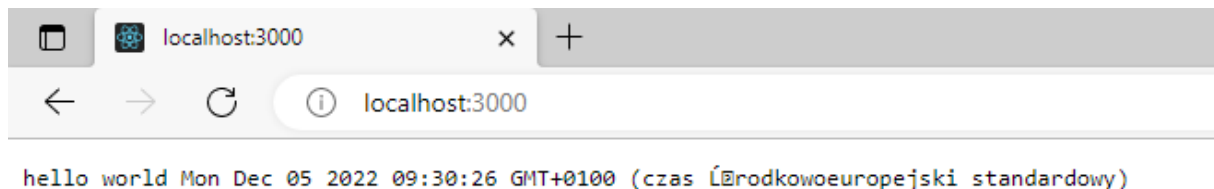
## 2.3 Najprostszy serwer HTTP w node.js

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');
```

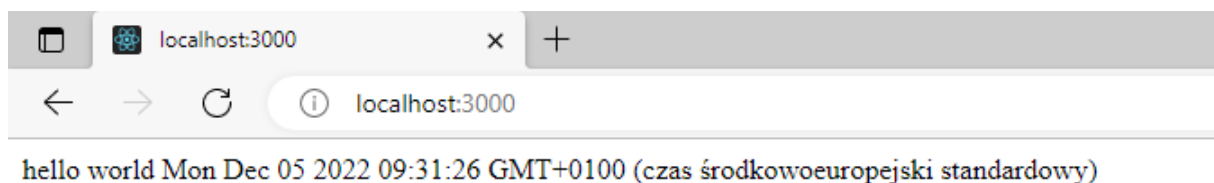


Tę niefortunną prezentację można skorygować wymuszając interpretowanie odpowiedzi w określonym kodowaniu (lub dodatkowo: typie), np.:

```
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 edytorze kodu prawym przyciskiem i przywołując funkcję „Go to type definition”, można przejrzeć interfejs obu typów.

Funkcje zwrotne kolejnych żądań (na przykład – żądań wielu użytkowników) trafiają do pętli zdarzeń i tworzą **kolejkę** oczekujących żądań. Z uwagi na charakterystykę środowiska uruchomieniowego Javascript – żądania z kolejki są obsługiwane oczywiście *jednowątkowo*. Między innymi dlatego tak ważne jest aby możliwie dużo pracy funkcja obsługująca żądanie delegowała do *asynchronicznych kontynuacji* – jeśli na przykład w trakcie obsługi żądania można wywołać asynchroniczną funkcję odczytującą plik, to takie asynchroniczne żądanie *uwalnia* pętlę zdarzeń, pozwala przetwarzać kolejne żądanie z kolejki, a to oryginalne żądanie, po zakończeniu wywołania asynchronicznego ustawia się na końcu pętli zdarzeń.

Asynchroniczność jest tu kluczowa do zwiększenia przepustowości – procesor przetwarza wyłącznie kod i można zrezygnować z kosztownego *oczekiwania* na żądanie synchroniczne do zewnętrznego zasobu.

Przykładowo – jeśli do serwera napisanego w node.js trafią równocześnie dwa żądania od klientów i w ramach każdego żądania trzeba będzie przeczytać plik, to sekwencja zdarzeń z punktu widzenia serwera może wyglądać następująco:

- Do serwera trafia żądanie A, ustawia się w kolejce zdarzeń
- Do serwera trafia żądanie B, ustawia się w kolejce zdarzeń
- Serwer rozpoczyna przetwarzanie żądania A, w ramach tego żądania zaczyna odczytywać plik
- Serwer rozpoczyna przetwarzanie żądania B, w ramach tego żądania zaczyna odczytywać plik
- ... serwer czeka w pętli zdarzeń na zakończenie odczytu plików ...
- Odczyt pliku z żądania A kończy się i do pętli zdarzeń trafia kontynuacja. Odpowiedź na żądanie A jest przesyłana do klienta
- Odczyt pliku z żądania B kończy się i do pętli zdarzeń trafia kontynuacja. Odpowiedź na żądanie B jest przesyłana do klienta

W trakcie obsługi obu żądań procesor w żadnym momencie nie czekał na odczyt danych z plików – bo w tym czasie przetwarzał inne żądania z pętli zdarzeń.

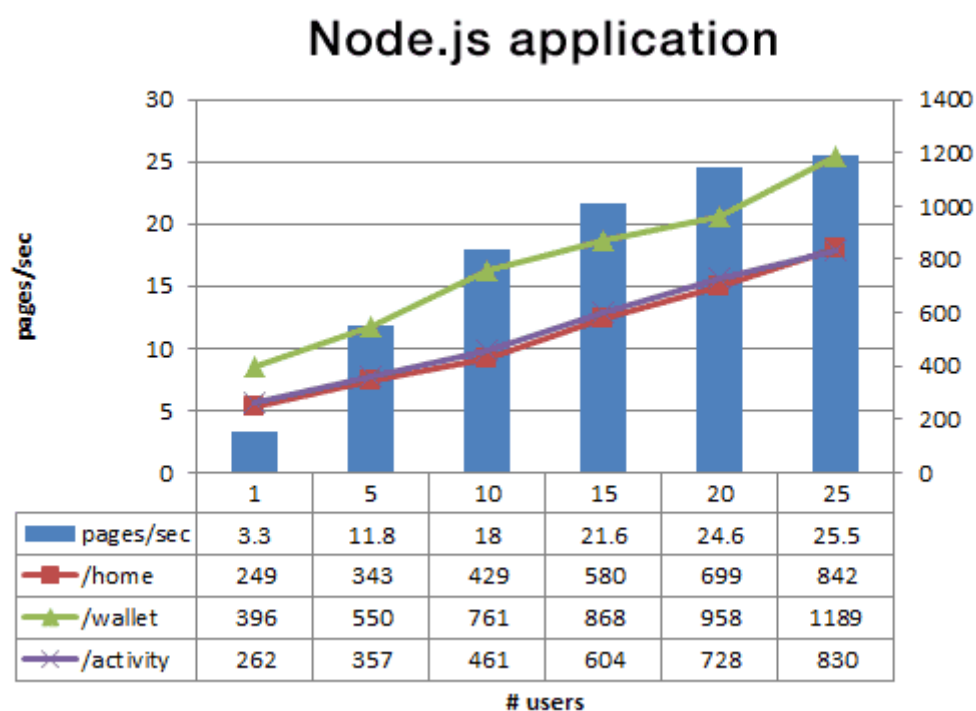
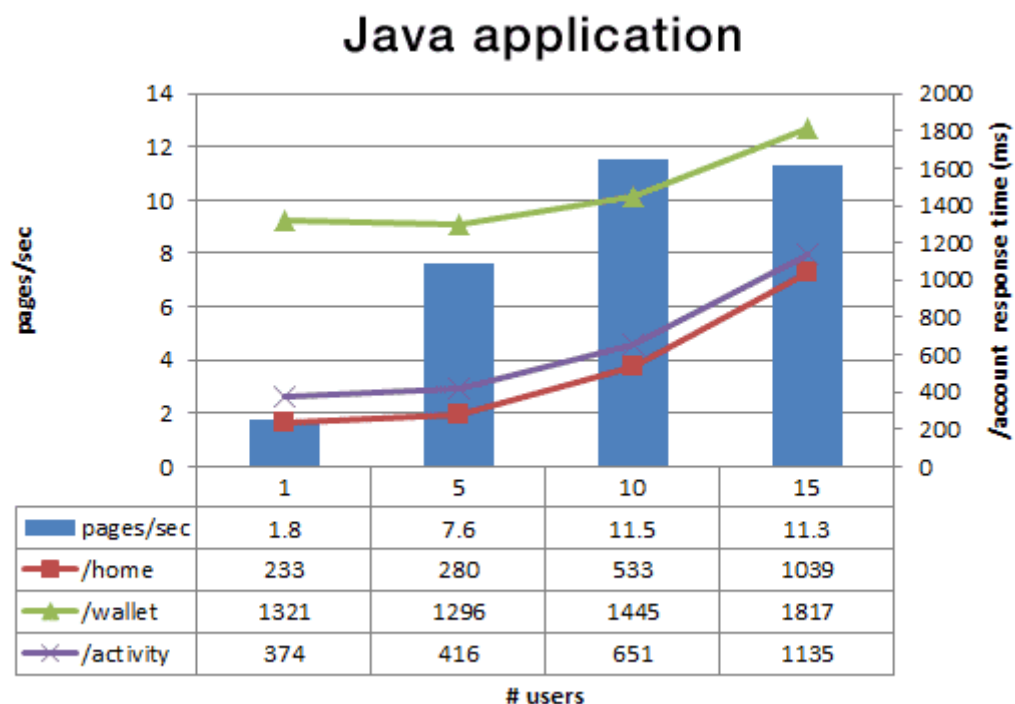
Taka charakterystyka pracy różni się diametralnie od podejścia w innych technologiach, w których dla każdego przychodzącego żądania tworzony jest **nowy wątek**. Przy dużym ruchu skalowalność takiego podejścia jest zwykle *gorsza* (systemowi więcej czasu zajmuje przełączanie się między zbyt dużą liczbą wątków niż po prostu trzymanie długiej listy callbacków i obsługiwanie ich po kolei – długość kolejki nie wpływa bowiem na przetwarzanie żądań). W takiej architekturze podobna sekwencja jak wyżej wyglądałaby tak:

- Do serwera trafia żądanie A, dla tego żądania tworzony jest nowy wątek,
- Do serwera trafia żądanie B, dla tego żądania tworzony jest nowy wątek
- Oba wątki wykonują się równolegle, oczekując synchronicznie na odczyt danych z plików, blokując na ten czas rdzenie procesora (w tym czasie nie będą wykonywać się inne wątki!)

Stąd stosunkowo duża [popularność](#) środowisk opartych o [libuv](#) czyli asynchronicznie I/O.

Można przywołać przykładowe opracowania, w których analizowano charakterystykę przepustowości dla takich dwóch typów aplikacji – aplikacji napisanej w oparciu o delegowanie asynchroniczne (node)

i klasycznej aplikacji z wieloma wątkami (np. Java). Wyniki takich porównań są zwykle na korzyść technologii wykorzystującej przetwarzanie asynchroniczne.



Rysunek 2 Przykład analizy przepustowości dwóch różnych technologii wykonany przez PayPal, za <https://medium.com/paypal-engineering/node-js-at-paypal-4e2d1d08ce4f>

Zbudowanie wielowątkowego rozwiązania wymaga użycia jawnie modułu [cluster](#). **Klaster** w terminologii node.js to grupa procesów, które współdzielą ten sam port TCP. Żądanie HTTP trafia do



**któregoś** z procesów dzielących się portem (wynika to ze sposobu w jaki w systemach operacyjnych zaimplementowano współdzielenie portów między procesami).

## 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 (RSA), w którym serwer ma klucz publiczny (publiczny certyfikat) i klucz prywatny (którego nie ujawnia). Serwer przesyła certyfikat do klienta, a klient używa certyfikatu do zaszyfrowania klucza do szyfrowania dalszej komunikacji (AES).

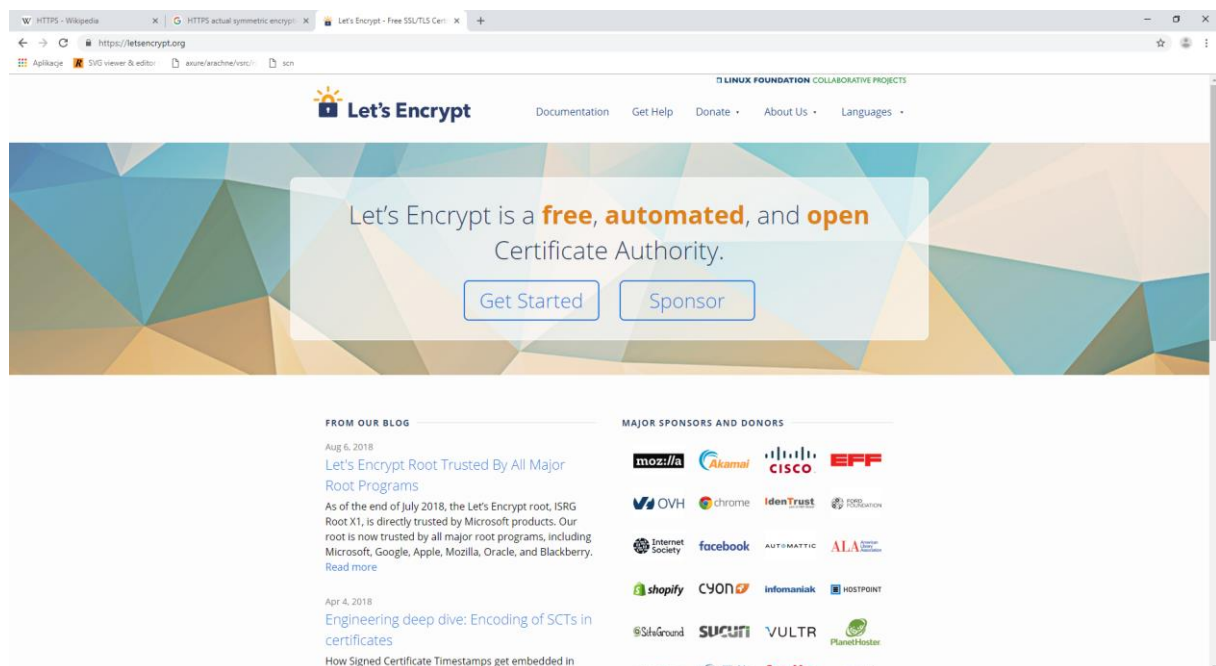
Należy zwrócić uwagę, że sama wymiana klucza możliwa jest w sposób bezpieczny za pomocą [algorytmu Diffiego-Hellmana](#), natomiast w tym przypadku odbywa się to prościej.

Po ustaleniu klucza i zweryfikowaniu autentyczności serwera, komunikacja odbywa się z wykorzystaniem tańszych algorytmów szyfrowania symetrycznego.

### 2.4.1 Generowanie certyfikatów

Certyfikat SSL typu RSA/ECDSA można wygenerować na przykład przy pomocy OpenSSL. Wygodne narzędzie z graficznym GUI to [KeyStore Explorer](#).

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,
```

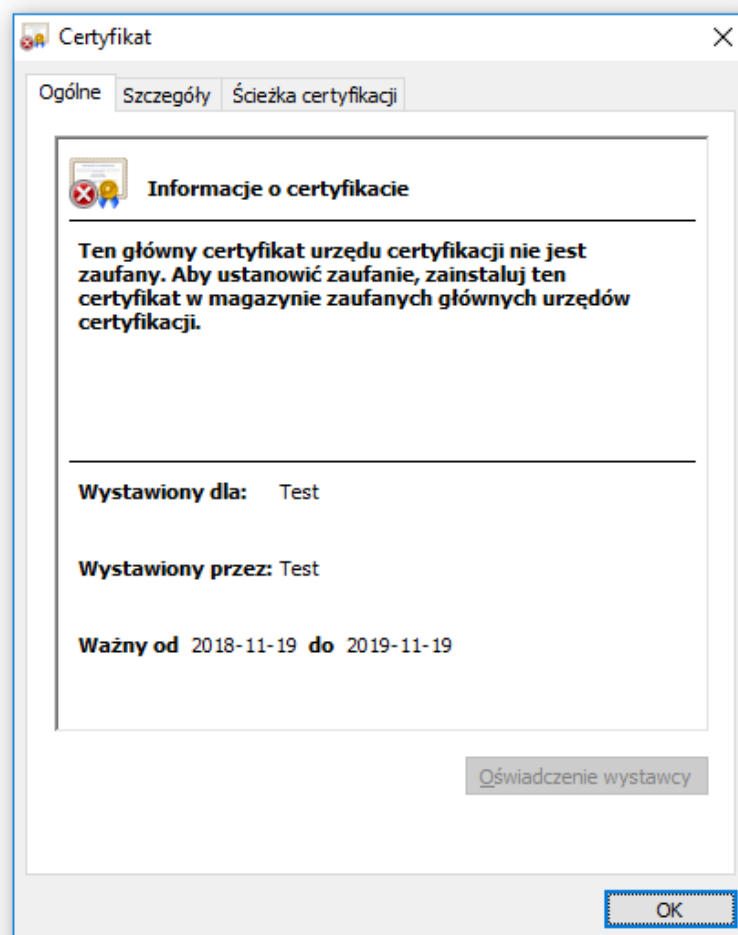
```
    passphrase: 'test'
  },
  (req, res) => {
    res.setHeader('Content-type', 'text/html; charset=utf-8');
    res.end(`hello world ${new Date()}`);
  });

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

← → ↻ ⚠ Niezabezpieczona | https://localhost:3000

Aplikacje  SVG viewer & editor < | axure/arachne/vsrc/rc | scn

hello world Mon Nov 19 2018 21:20:36 GMT+0100 (Środkowoeuropejski czas stand.)



## 2.5 HTTP/2

[HTTP/2](#) rozszerza protokół http o optymalizacje w warstwie transportowej:

- Otwiera zawsze jedno gniazdo między 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');
})();
```

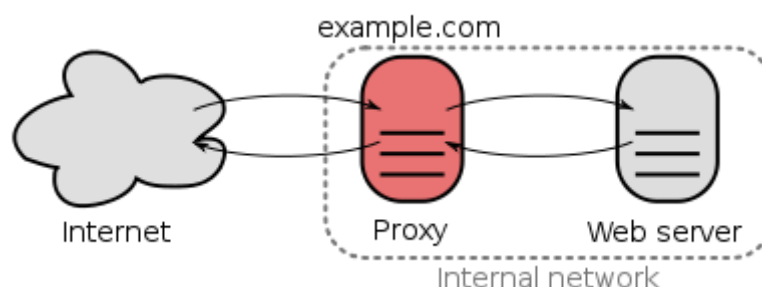
## 2.6 node.js a rzeczywistość

### 2.6.1 etc/hosts i lokalne mapy DNS

W pliku etc/hosts 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 Architektura 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 3 [https://en.wikipedia.org/wiki/Reverse\\_proxy#/media/File:Reverse\\_proxy\\_h2g2bob.svg](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).

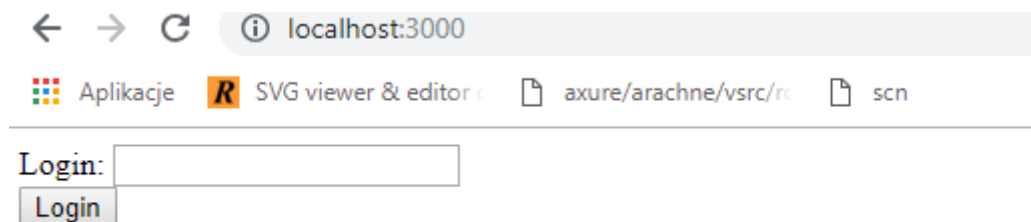
```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
```

```

    <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. zamienić na obiekt
            // { klucz:wartosc, klucz2: wartosc2 } (reduce!)
            var body =
                Object.fromEntries(
                    postdata
                        .split('&')
                        .map( kv => kv.split('=') )
                );

            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 niezadowolająca.

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.