Wybrane elementy praktyki projektowania oprogramowania

Wykład 11/15
node.js: Express (4)

Wiktor Zychla 2021/2022

1 Spis treści

2 AJAX

AJAX = Asynchronous Javascript And XML – ogólna nazwa techniki w której przeglądarka robi dodatkowe żądania do serwera nie przez nawigację (GET) lub wysyłanie formularzy (POST) ale za pomocą Javascript i obiektu XmlHTTPRequest.

Akronim wywodzi się z tego że w oryginalnej implementacji z serwera pobierano dane w postaci XML. W praktyce jest to niewygodne, ponieważ odczyt XML nie ma wsparcia w Javascript po stronie klienta (przeglądarki).

W praktyce częściej **AHAH** lub **AJAJ**

AHAH = Asynchronous HTML and HTTP – serwer zwraca HTML który jest dynamicznie dodawany w odpowiednie miejsce drzewa DOM za pomocą Javascript w przeglądarce

AJAJ = Asynchronous Javascript and JSON – serwer zwraca dane w postaci JSON, które są dynamicznie przetwarzane i zamieniane na elementy drzewa DOM za pomocą Javascript w przeglądarce. Ten sposób przekazywania danych występuje najczęściej w aplikacjach budowanych przy użyciu frameworków typu React czy Angular.

```
// app.js
var http = require('http');
var express = require('express');
var ejs = require('ejs');
var multer = require('multer');

var app = express();
var upload = multer();

app.set('views', './views');
app.set('view engine', 'html');

app.engine('html', ejs.renderFile );

app.post('/ajax', upload.single(), (req, res) => {

    var txtParam = req.body.txtParam;
    res.end(`<div>zawartość z serwera ${txtParam}</div>`);
});

app.get('/', (req, res) => {

    res.render('app', { message : 'dynamiczne dane 2'} );

});
```

```
http.createServer(app).listen(process.env.PORT || 3000);
```

```
<!-- views/app.html -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script>
    window.addEventListener('load', function() {
        var bt = document.getElementById('btApp');
        bt.addEventListener('click', function() {
            var content = document.getElementById('content');
            var param = document.getElementById('txtParam');
            //content.innerHTML = 'nowa zawartość modyfikowana';

            var req = new XMLHttpRequest();
            req.open('post', '/ajax', true);
            req.onreadystatechange = function()
            {
                if ( req.readyState == XMLHttpRequest.DONE )
                {
                    content.innerHTML = req.responseText;
                }
            }
            var form = new FormData();
            form.append('txtParam', param.value);
            req.send(form);
        });
    });
    </script>
</head>
<body>
    kawałek statyczny
    <div id='content'>
    stara zawartość
    </div>
    <input type='text' id='txtParam'> <br/>
    <button id='btApp'>odśwież zawartość</button>
</body>
</html>
```

3 WebSockets

WebSockets – dupleksowy (dwukierunkowy) protokół komunikacyjny, oparty o TCP. Wbudowany we współczesne przeglądarki.

Protokół WebSockets inicjuje się z przeglądarki żądaniem z dodatkowym nagłówkiem

```
Upgrade: websocket
```

na który serwer reaguje odesłaniem HTTP 101 (Switching protocols) i ustanowieniem osobnego gniazda komunikacyjnego.

Tworzenie aplikacji opartych o WebSockets możliwe jest w podstawowym Javascript w przeglądarce, aczkolwiek jest dość uciążliwe, stąd szereg „opakowań" ułatwiających pisanie kodu klienta i serwera. Dla node.js najpopularniejszą biblioteką dla WebSockets jest socket.io.

Socket.io zrewolucjonizował swego czasu tworzenie interaktywnych aplikacji internetowych z uwagi na ogromną łatwość modelu programowania oraz obsługę tzw. fallback czyli sytuacji w której przeglądarka nie posiadaja wsparcia dla WebSockets (obecnie to nie ma aż takiego znaczenia).

W poniższym przykładzie zademonstrowano dwie najprostsze funkcje socket.io:

- Emitowanie spontaniczne komunikatów z serwera do klienta – powiadomienie **message** jest emitowanie z serwera co sekundę do wszystkich połączonych klientów
- Emitowanie komunikatów z klienta na serwer i ich reemisję do innych klientów – powiadomienie **chat message**

```
// app.js
var http = require('http');
var socket = require('socket.io');
var fs = require('fs');
var express = require('express');

var html = fs.readFileSync('app.html', 'utf-8');

var app = express();
var server = http.createServer(app);
var io = socket(server);

app.use( express.static('./static'));

app.get('/', function(req, res) {
    res.setHeader('Content-type', 'text/html');
    res.write(html);
    res.end();
```

```
});

server.listen(process.env.PORT || 3000);


io.on('connection', function(socket) {
    console.log('client connected:' + socket.id);
    socket.on('chat message', function(data) {
        io.emit('chat message', data); // do wszystkich
        //socket.emit('chat message', data); tylko do połączonego
    })
});

setInterval( function() {
    var date = new Date().toString();
    io.emit( 'message', date.toString() );
}, 1000 );

console.log( 'server listens' );
```

```
<!-- app.html -->
<html>
    <meta charset="utf-8" />
    <head>
        <script src="/socket.io/socket.io.js"></script>
        <script>
        window.addEventListener('load', function() {

            var socket = io();
            socket.on('message', function(data) {
                var t = document.getElementById('time');
                t.innerHTML = data;
            });

            socket.on('chat message', function(data) {
                var msg = document.getElementById('messages');
                msg.innerHTML += data + "<br/>";
            });

            var btsend = document.getElementById('btsend');
            btsend.addEventListener('click', function() {
                var txtmessage = document.getElementById('txtmessage');
                socket.emit('chat message', txtmessage.value);
            });

        });
        </script>
    </head>
```

```
    <body>
    Aktualny czas na serwerze :) <b><span id="time"></span></b>.

    <div>
        <input type='text' id='txtmessage' />
        <button id='btsend'>Wyślij</button>
        <div id='messages'></div>
    </div>
    </body>
</html>
```

Biblioteka socket.io daje stosunkowo duże możliwości jeśli chodzi o obsługę połączeń do połączonych klientów – wysyłać można nie tylko dane do wszystkich połączonych przeglądarek, ale także: grupować je w tzw. przestrzenie nazw (*namespaces*) i pokoje (*rooms*).

Organizacja kodu obsługującego większą liczbę użytkowników wygląda zwykle od strony serwera tak, że tworzy się pomocniczą strukturę danych, w której mapuje się identyfikatory połączonych gniazd na pomocnicze dane związane z konkretnym gniazdem (np. nazwa i podstawowe dane użytkownika).

Sytuacja komplikuje się jeśli proces node.js jest klastrowany (obsługiwany przez wiele procesów na jednym serwerze lub wręcz – wiele serwerów). W takiej sytuacji infrastruktura musi zapewnić trafianie klienta zawsze do tego samego procesu, który utrzymuje jego połączenie. Dodatkowa trudność polega na konieczności obsłużenia emisji z wielu serwerów do wielu połączonych klientów. Dokumentacja socket.io omawia ten scenariusz.

Pełen zestaw możliwości komunikacyjnych podsumowuje następujący wyciąg z dokumentacji:

```
io.on('connect', onConnect);

function onConnect(socket){

  // sending to the client
  socket.emit('hello', 'can you hear me?', 1, 2, 'abc');

  // sending to all clients except sender
  socket.broadcast.emit('broadcast', 'hello friends!');

  // sending to all clients in 'game' room except sender
  socket.to('game').emit('nice game', "let's play a game");

  // sending to all clients in 'game1' and/or in 'game2' room, except sender
  socket.to('game1').to('game2').emit('nice game', "let's play a game (too)"
);

  // sending to all clients in 'game' room, including sender
  io.in('game').emit('big-announcement', 'the game will start soon');

  // sending to all clients in namespace 'myNamespace', including sender
```

```
  io.of('myNamespace').emit('bigger-
announcement', 'the tournament will start soon');

  // sending to a specific room in a specific namespace, including sender
  io.of('myNamespace').to('room').emit('event', 'message');

  // sending to individual socketid (private message)
  io.to(`${socketId}`).emit('hey', 'I just met you');

  // WARNING: `socket.to(socket.id).emit()` will NOT work, as it will send t
o everyone in the room
  // named `socket.id` but the sender. Please use the classic `socket.emit()
` instead.

  // sending with acknowledgement
  socket.emit('question', 'do you think so?', function (answer) {});

  // sending without compression
  socket.compress(false).emit('uncompressed', "that's rough");

  // sending a message that might be dropped if the client is not ready to r
eceive messages
  socket.volatile.emit('maybe', 'do you really need it?');

  // specifying whether the data to send has binary data
  socket.binary(false).emit('what', 'I have no binaries!');

  // sending to all clients on this node (when using multiple nodes)
  io.local.emit('hi', 'my lovely babies');

  // sending to all connected clients
  io.emit('an event sent to all connected clients');

};
```

4 Deployment

Deployment = umieszczenie aplikacji w środowisku serwerowym z którego jest hostowana w sieci

Continuous Deployment = technika organizacji sposobu publikacji aplikacji, w którym dostarczenie aplikacji do środowiska z którego jest hostowana jest zautomatyzowane (i być może wplecione w proces kompilacji)

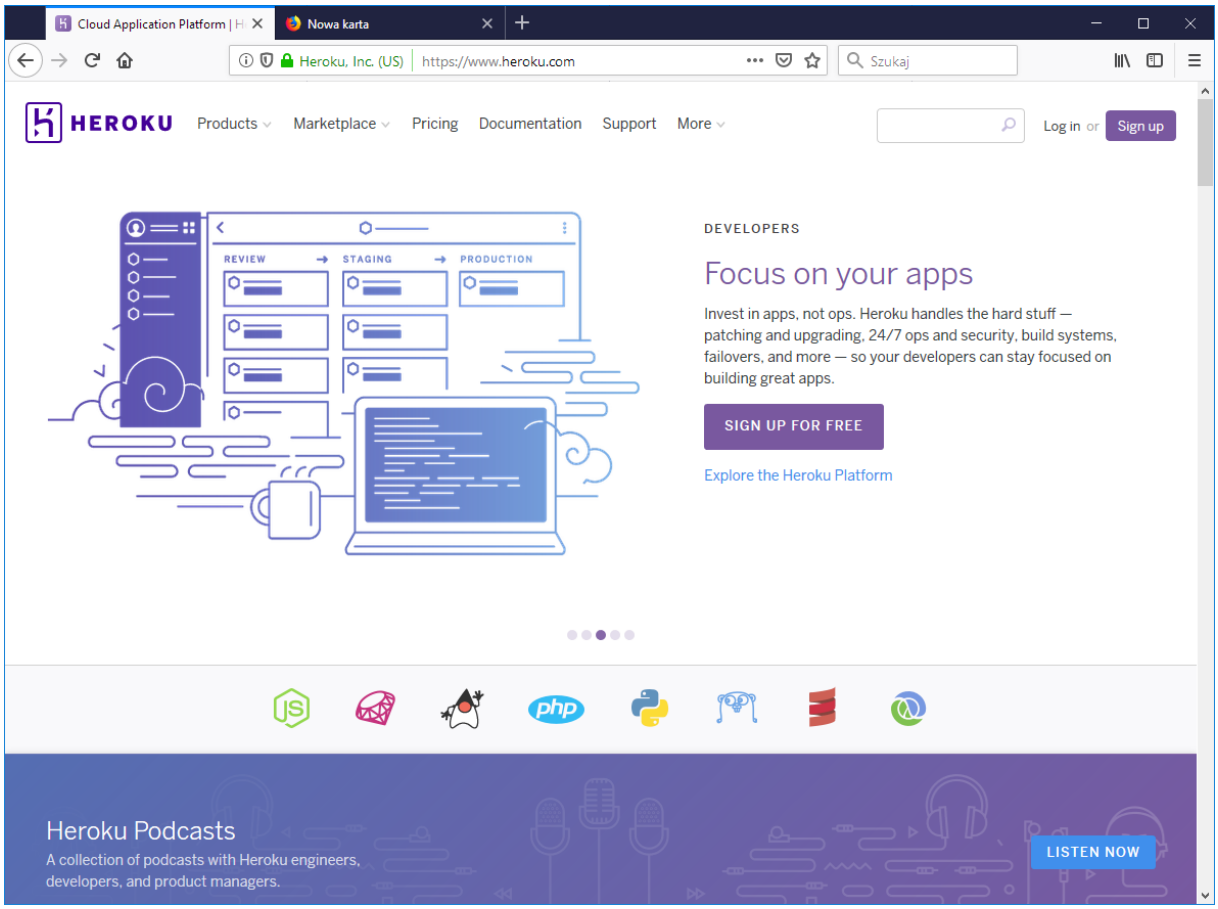
Z punktu widzenia aplikacji node.js, łatwość Continuous Deployment polega na tym, że Javascript nie wymaga kompilacji. W efekcie, w przeciwieństwie do wielu innych języków / technologii, na serwer trafia kod źródłowy.

Umożliwia to ciekawe podejście w którym zasób sieciowy repozytorium kodu (np. GIT) jest jednocześnie miejscem z którego aplikacja jest udostępniana.

Hosting node.js świadczą obecnie wszyscy duzi i wielu małych dostawców:

- Microsoft w chmurze Azure
- Google w chmurze App Engine
- Amazon w AWS

My poznamy możliwość darmowego hostingu node.js na Heroku.



Do zahostowania aplikacji na Heroku niezbędne są:

- Klient Git
- Heroku CLI

Instrukcja

1. W package.json dodać scripts/start

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node app.js"
  }
```

2. Utworzyć plik tekstowy **.gitignore** z wpisanym **node_modules** (folder ignorowany przy commit/push)

```
node_modules
```

3. Zmodyfikować kod dodając możliwość przekazania numeru portu do nasłuchu lokalnego (serwer Heroku przekazuje numer portu w zmiennej środowiskowej):

```
server.listen(process.env.PORT || 3000);
```

4. Z linii poleceń wykonywać skrypt (linia po linii)

```
git init
git add .
git commit -m "initial commit"
heroku login
heroku create
git push heroku master
```

Po wykonaniu poszczególnych kroków, aplikacja działa na Heroku. Pełna dokumentacja integracji z node.js.