Wybrane elementy praktyki projektowania oprogramowania

Wykład 11/15
node.js: Express (4)

Wiktor Zychla 2023/2024

1 Spis treści

2 Single-Page Applications, AJAX

Aplikacja typu Single-Page (Single Page Application) – typ aplikacji internetowej, w której po pierwszym załadowaniu strony nie ma klasycznej nawigacji (za pomocą linków lub formularzy), zamiast tego przeglądarka **dynamicznie** przebudowuje zawartość strony za pomocą operacji na drzewie DOM.

Do pozyskania danych z serwera, aplikacja typu SPA używa wbudowanego w silnik Javascript obiektu XMLHttpRequest lub, współcześnie, wygodnej fasady – funkcji **fetch**

Do operacji na drzewie DOM używa się m.in. funkcji i właściwości

- document
  - getElementById
  - querySelector
  - querySelectorAll
  - createElement
- Node
  - insertBefore
  - removeChild
- Element
  - attributes
  - innerHTML

W praktyce, obsługa modyfikacji drzewa DOM za pomocą tak niskopoziomowego interfejsu jest wyjątkowo żmudna i problematyczna. Stąd, poza nielicznymi wyjątkami, nie pracuje się na takim interfejsie – zamiast niego szuka się bibliotek/frameworków wspierających.

W trakcie naszego wykładu będzie jeszcze okazja przyjrzeć się technologiom wspierającym tworzenie aplikacji typu SPA.

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 jest stosunkowo często używany, z uwagi na wygodę – Javascript po stronie przeglądarki obsługuje serializację/deserializację (= zamianę obiektu na string i string na obiekt) w formacie JSON za pomocą metod:

- JSON.stringify
- JSON.parse

W poniższym przykładzie, po naciśnięciu przycisku, przeglądarka wysyła do serwera żądanie bez przeładowania strony (funkcja **fetch**) a następnie wynik tego żądania jest użyty do przebudowania zawartości strony (**content.innerHTML = …**).

```
// 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', 'ejs');

app.post('/ajax', upload.single(), (req, res) => {
    var txtParam = req.body.txtParam;
    res.end(`<div>dynamiczna zawartość z serwera. przysłany z serwera
parametr: ${txtParam}</div>`);
});

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

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

console.log( 'serwer działa' );
```

```
<!-- views/app.ejs -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script>
    // dodanie handlera do zdarzenia załadowania strony
    window.addEventListener('load', function() {
        // dodanie handlera do zdarzenia click przycisku
        var bt = document.getElementById('btApp');
```

```
        bt.addEventListener('click', async function() {
            var param = document.getElementById('txtParam');

            var form = new FormData();
            form.append('txtParam', param.value);

            var response = await fetch('/ajax', {
                body: form,
                method: 'post'
            });

            var responseText = await response.text();

            var content = document.getElementById('content');
            content.innerHTML = responseText;
        });
    });
    </script>
</head>
<body>
    treść statyczna strony
    <div id='content'>
        treść dynamiczna strony - będzie podmieniona po żądaniu do serwera
    </div>
    <div>
        Parametr do przesłania na serwer:
        <input type='text' id='txtParam'> <br/>
    </div>
    <button id='btApp'>podmień treść dynamiczną</button>
</body>
</html>
```

3 Representational state transfer (REST)

Skoro aplikacja typu SPA komunikuje się z serwerem za pomocą żądań przesyłanych za pomocą **fetch** to można myśleć o jakiejś strukturyzacji takich żądań.

Jednym z pomysłów na architekturę jest REST (Representational state transfer).

W tej architekturze – na serwerze istnieje szereg **punktów końcowych** (adresów) obsługujących żądania do różnych kategorii danych. Aby odróżnić rodzaj operacji (odczyt, modyfikacja, usuwanie, …) przyjmuje się że żądania do tej samej kategorii danych ale o różnych operacjach różnią się użytą metodą HTTP:

Metoda HTTP	**Opis**
GET	Pobranie danych. Operacja idempotentna (jej wykonanie nie powinno zmieniać stanu danych na serwerze)
POST	Dodanie nowych danych
PUT	Aktualizacja danych Operacja idempotentna (jej wielokrotne wykonanie nie powinno zmieniać stanu danych na serwerze więcej niż raz)
DELETE	Usunięcie danych

Dodatkowo, wymagane jest określenie konwencji wskazywania danych, przykładowo

Rodzaj operacji	**Żądanie**
Pobranie wszystkich danych	GET /api/User Response: [{ id: 1, name: Jan, surname: Kowalski }, { id: 2, …]
Pobranie jednej wskazanej danej	GET /api/User/:id Response: { id: 1, name: Jan, surname: Kowalski }
Dodanie danych	POST /api/User Request: { name: Jan, surname: Kowalski }

	Response: { id: 1, name: Jan, surname: Kowalski } lub informacja o błędzie
Aktualizacja danych	PUT /api/User/:id Request: { id: 1, name: Jan2, surname: Kowalski2 } Response: { id: 1, name: Jan2, surname: Kowalski2 } lub informacja o błędzie
Usunięcie danych	DELETE /api/User/:id

W sytuacji gdy lista punktów końcowych obejmuje wiele pozycji, zdecydowanie należy rozważyć jeden ze standardów dokumentacji punktów końcowych, na przykład będący standardem przemysłowym OpenAPI/Swagger

- OpenAPI to nazwa specyfikacji
- Swagger to bodaj najpopularniejszy zestaw narzędzi implementujący tę specyfikację

Należy też zadać pytanie o to jak uwierzytelniać dostęp do usług API.

Odpowiedzi już znamy:

- Mogą być usługi API publiczne – bez uwierzytelniania
- Mogą być usługi API zabezpieczone ciasteczkiem, tak samo jak zabezpieczaliśmy ścieżki do wrażliwych zasobów na poprzednim wykładzie. Ten sposób jest wygodny w sytuacji gdy klientem jest przeglądarka
- Mogą być usługi API zabezpieczone innym **nagłówkiem HTTP** niż ciasteczko, mówi się wtedy o tzw. tokenie lub kluczu API (API key). Nagłówek może mieć dowolną nazwę (np. api_key) i dowolny format (np. znany nam już z poprzedniego wykładu JWT – to był format w jakim dostawca OpenIDConnect zwracał informacje o użytkowniku)

W poniższym, trochę bardziej rozbudowanym przykładzie, będzie można przyjrzeć się aplikacji klienckiej, która demonstruje użycie trzech akcji REST (GET, POST, DELETE) oraz serwera który przygotowany jest na wszystkie cztery (GET, POST, PUT, DELETE).

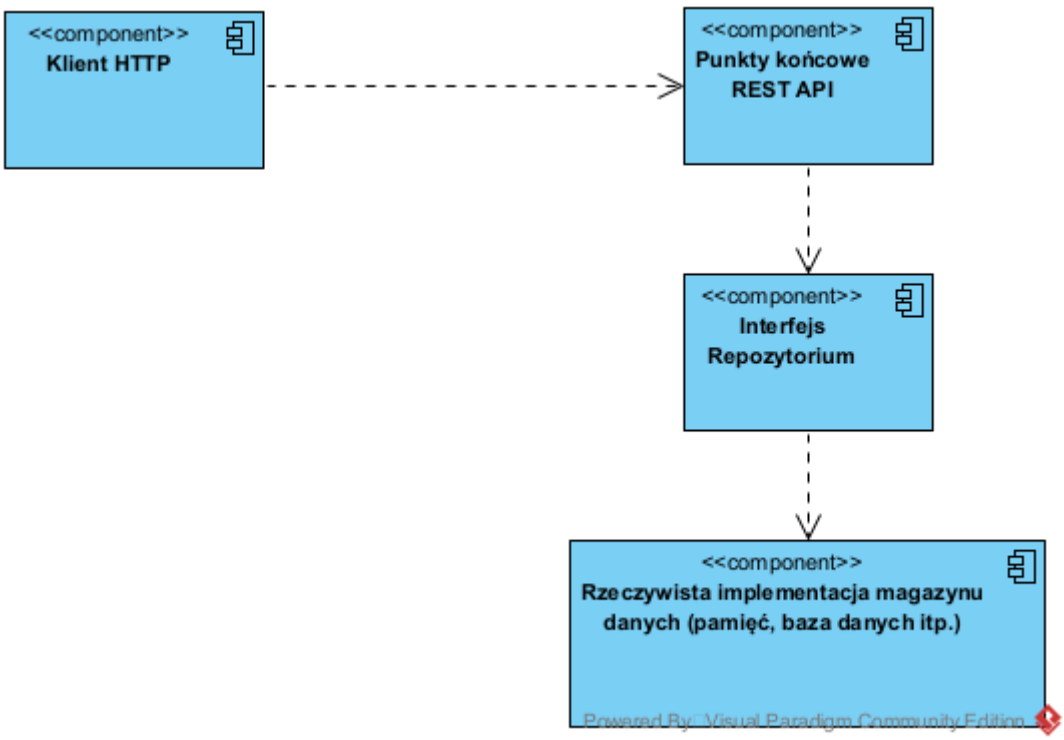
Przy okazji proszę zwrócić uwagę na oddzielenie kodu umieszczonego w middleware obsługi poszczególnych punktów końcowych API od dostępu do faktycznych danych – dostęp do danych został schowany wewnątrz obiektu usługowego typu Repository – jest to jeden z ważnych Wzorców architektury aplikacji, będzie jeszcze okazja zobaczyć go w akcji wtedy kiedy będziemy programować dostęp do bazy danych.

Na tę chwilę proszę natomiast zwrócić uwagę na ogromny zysk z użycia tego wzorca: daje on klientowi (w tym przypadku są to funkcje middleware obsługi poszczególnych punktów końcowych) **stabilny** interfejs komunikacyjny:

- getTodos
- createNewTodo
- updateTodo
- removeTodo

Interfejs ten dla swojego klienta jest mniej więcej tym samym co interfejs API dla klienta HTTP – **ukrywa** implementację za, jak to mówimy żargonowo, **stabilnym interfejsem** (czyli takim który w przyszłości mógłby być inaczej zaimplementowany ale dalej tak samo wyglądać dla klienta).

Ta ważna zasada projektowania nosi nawet nazwę: Law of Demeter.




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

var todoRepo = require('./todos');
const { application } = require('express');
var app = express();

var json = express.json();

app.disable('etag');

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

app.get( '/api/todo', (req, res) => {
    res.json( todoRepo.getTodos() );
```

```
});

app.post( '/api/todo', json, (req, res) => {
    // middleware json od razu parsuje request
    var todo = req.body;
    var description = todo.description;
    // dodawanie
    var todo = todoRepo.addTodo(description);
    // zwrócenie do klienta
    res.json( todo );
});

app.put( '/api/todo/:id', json, (req, res) => {
    // middleware json od razu parsuje request
    var todo = req.body;
    // powinna być walidacja!
    // modyfikacja
    var id = req.params.id;
    todo = todoRepo.updateTodo(id, todo.description);
    // zwrócenie do klienta
    res.json(todo);
})

app.delete( '/api/todo/:id', (req, res) => {
    // usuwanie
    var id = req.params.id;
    todoRepo.removeTodo(id);
    res.status(200);
    res.end();
});

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

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

console.log( 'serwer działa' );
```

```
// todos.js
// proste repozytorium todo
let todos = [
    { id: 1, description: 'an example todo'},
    { id: 2, description: 'another todo'},
];
let newid = todos.length + 1;
```

```
function getTodos() {
    return todos;
}

function createNewTodo(description, id) {
    var newTodo = { description };
    if ( id ) {
        newTodo.id = id;
    } else {
        newTodo.id = newid++;
    }
    return newTodo;
}

function addTodo(description) {
    var todo = createNewTodo(description);
    todos.push( todo );

    return todo;
}

function updateTodo(id, description) {
    todos.removeTodo(id);
    var todo = createNewTodo(description, id);
    todos.push( todo );

    return todo;
}

function removeTodo(id) {
    todos = todos.filter( todo => todo.id != id );
}

module.exports = {
    getTodos,
    addTodo,
    updateTodo,
    removeTodo
}
```

```
<!-- views/app.ejs -->
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <style>
```

```
        .todoItem {
            border: 1px solid black;
            margin-bottom: 10px;
            padding: 10px;
        }
    </style>
    <script>

    // pobieranie
    async function refreshTodos() {
        var todos = document.getElementById('todos');
        todos.innerHTML = '';

        var response = await fetch( '/api/todo', { method: 'get'} );
        var jsonResponse = await response.json();

        for ( var todo of jsonResponse ) {
            todos.innerHTML +=
            `<div class='todoItem'>${todo.description} <button
class='deleteButton' type='button' data-id=${todo.id}>usuń</button></div>`;
        }

        // dodanie handlerów
        document.querySelectorAll('.deleteButton').forEach( function( button
) {
            button.addEventListener('click', deleteTodo );
        })
    }

    // dodawanie
    async function addTodo() {
        var description = document.getElementById('description').value;
        if ( description ) {
            var response = await fetch(
                '/api/todo',
                {
                    method: 'post',
                    headers: {
                    'Accept': 'application/json',
                    'Content-Type': 'application/json'
                    },
                    body: JSON.stringify( { description } )
                } );
            var jsonResponse = await response.json();

            // albo dodać tylko jeden
            // albo przeładować wszystkie
            await refreshTodos();
        }
```

```
    }

    // usuwanie
    async function deleteTodo(e) {
        e.preventDefault(); // gdyby był formularz to zabezpiecza przed
odesłaniem
        var id = e.target.getAttribute('data-id');
        await fetch(`/api/todo/${id}`, { method: 'delete'});
        await refreshTodos();
    }

    // dodanie handlera do zdarzenia załadowania strony
    window.addEventListener('load', function() {
        // załadowanie danych
        refreshTodos();
        // dodanie handlera do zdarzenia click przycisku
        var bt = document.getElementById('btAdd');
        bt.addEventListener('click', addTodo);
    });
    </script>
</head>
<body>
    <div>
        <div>
            New TODO: <input id="description" type="text" />
        </div>
        <div>
            <button id='btAdd'>add todo</button>
        </div>
    </div>
    <hr />
    <div id='todos'>
    </div>
</body>
</html>
```

4 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('./views/app.ejs', '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' );
```

```
<!-- views/app.ejs -->
<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');

};
```

5 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

Istnieją narzędzia umożliwiające tunelowanie żądań z internetu do lokalnej maszyny, przykładem takiego narzędzia jest ngrok. To szybki i wygodny sposób udostępnienia aplikacji na przykład do pokazu lub testu, wada tego typu udostępnienia to oczywiście konieczność utrzymywania aplikacji na lokalnej maszynie, na tej do której tunelowane są żądania.

Darmowy hosting node.js na 01.2023 oferują m.in.

https://render.com/

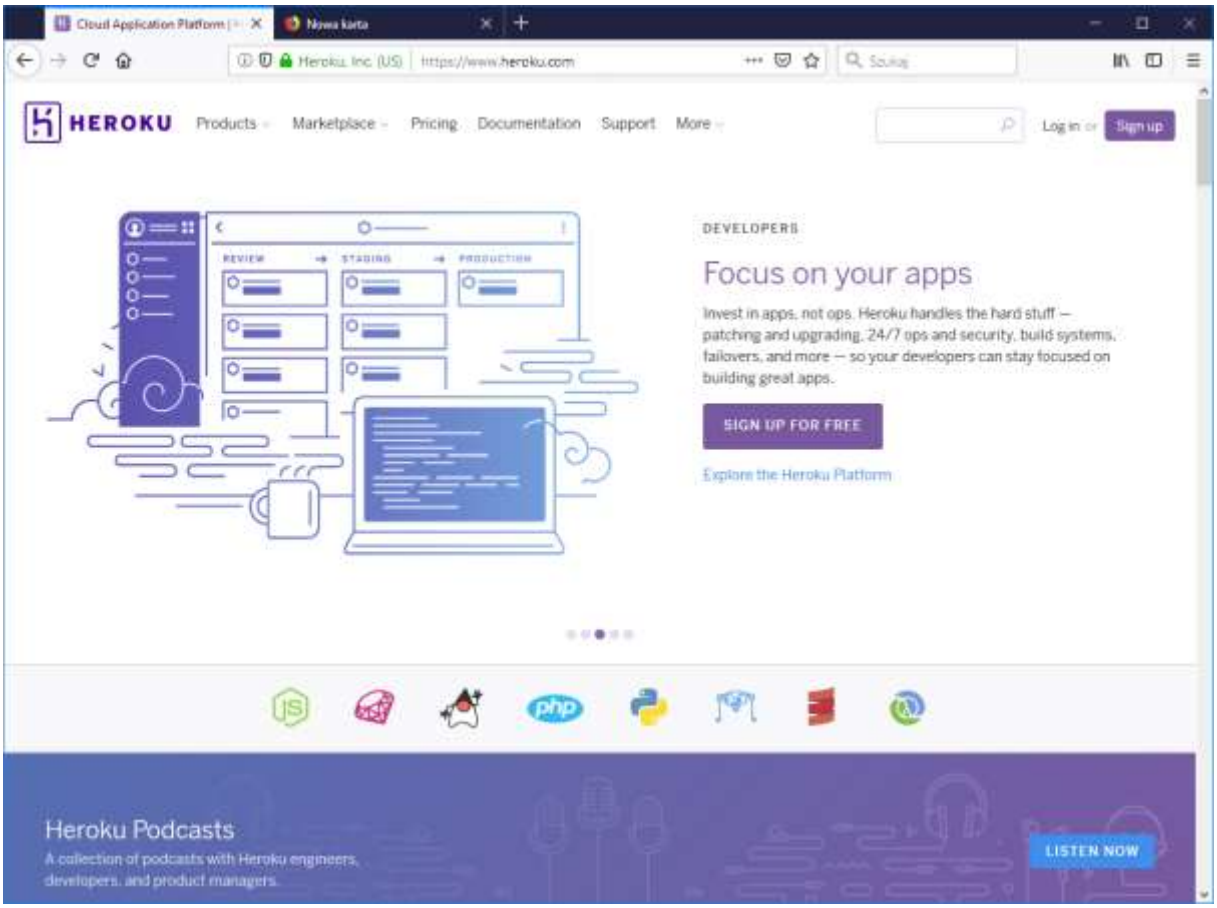
https://vercel.com/

https://www.cyclic.sh/

Państwu pozostawiam zapoznanie się z aktualnymi możliwościami hostingu we własnym zakresie. W większości przypadków należy utworzyć konto i nowy projekt, wskazujący repozytorium na GitHub.

Przykład konfiguracji dla Heroku.

Uwaga! Od jesieni 2022 Heroku nie ma darmowego hostingu node.js.



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.