MODUŁ 7:
Budowa aplikacji WWW w Node.js i Express

Cel modułu

Moduł stanowi kontynuację materiału moduły 06. Zrealizujemy niektóre z przedstawionych tam wzorców projektowych, zapoznamy się z kolejnymi mechanizmami Node.js oraz udogodnieniami zapewnianymi przez ramę Express.

Efekty kształcenia

Niniejsza lekcja służy osiągnięciu następujących efektów kształcenia.

Wiedza

Po ukończonym kursie student / studentka:

- Objaśnia zasady asynchronicznego programowania w środowisku Node.js,
- Przedstawia zasady modularyzacji i hermetyzacji kodu w Node.js,
- Omawia sposób realizacji w Node.js wzorców MVC oraz PRG.

Umiejętności

Po ukończonym kursie student / studentka:

- buduje prostą funkcjonalność serwera HTTP opartą na modułach wbudowanych Node.js;
- modularyzuje implementację aplikacji WWW stosując moduły;
- definiuje projekt i stosuje w nim pakiety stworzone przez zewnętrzne podmioty;
- tworzy definicje routingu przy wykorzystaniu ramy Express;
- buduje widoki posługując się szablonami i przekazuje do nich dane.

Uzasadnienie

Implementacja pełnowymiarowych witryn / aplikacji WWW wymaga skutecznych narzędzi do panowania nad złożonością oprogramowania. W tym celu niezbędna jest m.in. konsekwentna modularyzacja kodu oraz przyjęcie spójnych konwencji w zakresie struktury i nazewnictwa. Produktywność, pielęgnacyjność oraz unikanie błędów przemawiają ponadto za dążeniem do podnoszenia poziomu abstrakcji, w czym pomagają dedykowane ramy programistyczne (frameworks) oferujące gotową implementację dla typowych zagadnień oraz wsparcie dla realizacji określonych wzorców projektowych.

Programowanie w Node.js – c.d.

Przekierowania

W poprzednim module realizowaliśmy dynamiczną konstrukcję odpowiedzi opartą na wywołaniach odpowiednich metod obiektu res.

```
res.setHeader('Content-Type', 'text/html');

res.write('<html>');

res.write( '… pozostała treść dokumentu…');
```

```
res.end();
```

Warto zaznaczyć, że zwracanie zasobu nie jest jedynym typem odpowiedzi serwera. Często (np. jako element wzorca PRG) będziemy potrzebowali sięgnąć po przekierowanie – polegające na konstrukcji odpowiedzi o odpowiednim kodzie i zaopatrzeniu jej w nagłówek zawierający docelowy adres URL – np.

```
if( warunek_wymagajacy_przekierowania_do_glownej ) {
      res.statusCode = 302;
      res.setHeader('Location', '/');
      return res.end();
}
```

Zwróćmy uwagę na obecność komendy return przy ostatnim wywołaniu operacji. Może ono być istotne przy obecności całej sekwencji takich bloków warunkowych jak powyższy, aby po wykonaniu zawartych tu instrukcji nie spowodować przetwarzania kolejnych.

Przetwarzanie żądań

Jak dotąd przetwarzaliśmy w przedstawionych przykładach wyłącznie dane zawarte w samym adresie URL (czy to w jego ścieżce, czy w *query string*). W przypadku danych przekazywanych za pomocą formularza, zwykle będziemy mieć natomiast do czynienia z wydobywaniem danych z ciała komunikatu żądania. Jest to (w przypadku bezpośredniego, niskopoziomowego oprogramowywania) o tyle bardziej skomplikowane, że, jak być może pamiętamy z protokołu HTTP, zarówno ciała żądania jak i odpowiedzi mogą być przesyłane w formie tzw. *chunked encoding*. Mamy więc do czynienia ze strumieniem danych.

Bezpośrednia implementacja będzie więc wymagała dwóch elementów:

- określenia zachowania w odpowiedzi na zdarzenie nadejścia danych;
- określenie zachowania w odpowiedzi na zdarzenie zakończenia odbierania danych.

Możemy to osiągnąć poprzez następującą strukturę kodu:

```
const body = [];      // tablica dla napływających danych

req.on('data', (chunk) => {
  body.push(chunk);  // zasilenie tablicy
});

req.on('end', () => {
      const parsedBodfy = Buffer.concat(body).toString();
      const message = parsedBody.split('=')[1];
});
```

Widzimy tutaj typową w programowaniu Node.js składnię przekazywanych jako callback funkcji, zwaną *arrow notation*. (Dla uproszczenia przykład zakłada, że przesłane dane formularzowe składają się tylko z jednej pary klucz=wartość i wydobywa wobec tego tekst stanowiący tę wartość.)

Asynchroniczne programowanie na przykładzie operacji dyskowych

Zgodnie z tym, czego dowiedzieliśmy się już o pętli zdarzeń, powinniśmy dbać, aby przekazywana serwerowi implementacja była pozbawiona blokujących oczekiwań na operacje I/O. W tym celu dysponujemy m.in. asynchronicznymi implementacjami operacji dyskowych, które umożliwiają wyodrębnienie w postaci funkcji callback kodu wykonywanego po ukończeniu operacji. Np. – przekierowanie realizowane po zapisaniu danych do pliku:

```
fs.writeFile('message.txt', dane, err => {
      res.statusCode = 302;
      res.setHeader('Location', '/');
      return res.end();
});
```

Moduły w Node.js

Przypomnijmy rozróżnienie dotyczące stosowanych w Node.js modułów. Wyróżniamy:

(1) Core modules. Wystarczy w tym przypadku funkcja `require()`. Nie musimy tych modułów instalować w naszym projekcie. Przykładami dotychczas napotkanymi są fs, http.
(2) Lokalne moduły kodu – korzystając z nich podamy w funkcji `require()` ścieżkę względną do danego pliku z kodem, pomijając jednak rozszerzenie pliku (.js)
(3) Moduły zawarte w pakietach stron trzecich. Wymagają zainstalowania np. menedżerem npm.

Modularyzacja kodu z regułami routingu

Przećwiczmy moduły (2) reorganizując nasz kod tak, aby klauzule dotyczące obsługi poszczególnych adresów i rodzajów żądań znalazły się w jednym lub większej liczbie plików zewnętrznych.

Zakładamy, że mamy już kilka bloków kodu w postaci instrukcji warunkowych `if(…) { … }`.

Zgodnie z szeroko stosowanymi konwencjami, możemy wyodrębnić plik o nazwie routes.js (albo folder o tej nazwie, jeśli zamierzamy bardziej rozbudować tę część) i umieścić w nim nasze instrukcje warunkowe opakowane w funkcję o sygnaturze (req, res) – np.

```
const reguly = (req, res) => {
      if(…) { … }
      if(…) { … }
}
module.exports = reguly;
```

Zwróćmy uwagę na właściwość exports, do której przypisujemy naszą funkcję. Gdyby nasz moduł wymagał udostępnienia większej liczby zasobów na zewnątrz, to moglibyśmy zamiast bezpośredniego przypisania zdefiniować w ramach module.exports dowolną liczbę nazwanych właściwości.

Teraz, w pliku głównym aplikacji, importujemy ten moduł następująco:

```
const routes = require('./routes');
```

zaś instrukcję tworzącą serwer zmienimy na:

```
const server = http.createServer(routes);
```

Rama Express

Tworzenie projektu i instalowanie pakietu

Stosując framework Express będziemy mieli do czynienia z instalowaniem pakietu strony trzeciej.

W tym celu powinniśmy utworzyć plik konfiguracyjny naszego projektu, w którym m.in. rejestrowane będą zależności zewnętrze w postaci nazw i wersji modułów niezbędnych do pracy naszego systemu.

Celem założenia pliku konfiguracyjnego, w folderze projekt wykonujemy w oknie terminala komendę: `npm init` , a następnie interakcyjnie wprowadzamy niezbędne opcje.

Utworzony zostanie plik package.json .

Możemy teraz zainstalować pakiet związany z tym konkretnie projektem – np.

```
npm install nodemon --save-dev
```

Komenda ta spowoduje zapisanie pakietu nodemon jako instalowanego lokalnie w bieżącym projekcie. Narzędzie nodemon będzie dla nas użyteczne jako substytut instrukcji node zapewniający bieżące aktualizowanie działającej implementacji. Oznacza to, że nie będziemy musieli restartować naszej aplikacji aby uruchomić dokonane przed chwilą zmiany w kodzie.

Zastosowana opcja sprawi, że pakied nodemon będzie instalowany tylko dla wersji deweloperskiej, ale nie dla produkcyjnej.

Zapoznaj się z aktualną zawartością pliku package.json oraz upewnij się, czy uruchomiony komendą nodemon kod aplikacji reaguje na zapisy zmian kodu dokonane w edytorze.

Wykorzystanie Express - wprowadzenie

W analogiczny sposób instalujemy Express. W tym przypadku jednak jego zastosowanie nie ograniczy się do wersji produkcyjnej, toteż wywołamy:

```
npm install --save express
```

Teraz, w kodzie aplikacji będzie niezbędny import zainstalowanego pakietu:

…

```
const express = require('express');
```

Teraz możemy go użyć to utworzenia i uruchomienia serwera.

```
const app = express();

// rezultat dostarcza nam funkcji obsługującej żądania (handlera) :

const server = http.createServer(app);

server.listen(3000);
```

Ponadto, dwa powyższe wiersze można dzięki Express skrócić do:

```
app.listen(3000);
```

Tak utworzony serwer będzie oczywiście pozbawiony logiki obsługi żądań, którą musimy wobec tego tu zdefiniować. Semantyczne będą to podobne konstrukcje jak w stosowanych przez nas wcześniej instrukcjach warunkowych, natomiast oferują one tu miejscami wyższy poziom abstrakcji i lepszą kompozycyjność. Reguły przetwarzania żądań definiowane w ramach Express noszą nazwę

middleware. Wywodzi się ona stąd, że stanowią one kod wykonywany pomiędzy przyjęciem żądania a wysłaniem przez serwer odpowiedzi.

Jak za chwilę zobaczymy, poszczególne klauzule układają się we wzorzec tzw. filtra. Polega on na tym, że jedna z klauzul przyjmuje żądanie i dokonuje jego przetwarzania (np. rejestrując otrzymane dane) a następnie może przekazać przetwarzanie do kolejnej klauzuli o analogicznej budowie.

Podstawowym konstruktem do budowy takich klauzul jest metoda use() :

```
app.use( (req, res, next)  => { … } );
```

Wnętrze tak załączonej funkcji budujemy w znany nam już sposób. Nowym elementem będzie natomiast wywołanie w kodzie takiej funkcji funkcji `next()`, które buduje scenariusz przetwarzania jako łańcuch filtrów zgodnie z ich kolejnością zdefiniowania w kodzie.

Również konstrukcja odpowiedzi jest w Express prostsza do zrealizowania. Możemy użyć metody send():

```
res.send('…kod HTML…');
```

Przy tej okazji możemy zauważyć, że pozostałe niskopoziomowe akcje (np. ustalenie kodu odpowiedzi i jej nagłówków) jest już zrealizowane automatycznie przez framework.

Zamiast zwracania zasobu

Routing żądań w Express

Routing zrealizujemy stosując wariant metody use() z szerszym zestawem parametrów. Dodatkowym, pierwszym parametrem będzie teraz wzorzec ścieżki. Odpowiednio do sposobu zorganizowania logiki przetwarzania (np. czy "wyciągamy przed nawias" jakąś generyczną część przetwarzania, czy też dedykowane dla danych URL pojedyncze middleware tworzymy jako samowystarczalne, będziemy stosować lub nie wywołanie next(). Np.

```
app.use('/dodaj-zgloszenie', (req, res, next) => { … });
```

Możemy również zróżnicować przetwarzanie zależnie od metody żądania (np. post, get) stosując zamiast `use()` metodę o takiej nazwie – np.:

```
app.post('/dodaj-zgloszenie', (req, res, next) => { … });
```

Przetwarzanie danych z żądania

Wcześniej mierzyliśmy się z przetwarzaniem surowego strumienia danych pochodzących z ciała komunikatu żądania. Express zapewnia tu wyższy poziom abstrakcji, choć nie obejdzie się bez dodatkowych udogodnień.

Będziemy potrzebowali pakietu body-parser:

```
npm install --save body-parser
```

Poi zaimportowaniu go w standardowy sposób umieszczamy jego funkcjonalność jako element middleware:

app.use(bodyParser.urlencoded({extended: false}));

Dostarczona przez pakiet funkcja przetwarzająca przebuduje ciało żądania (req.body do postaci struktury klucz-wartość oraz zadba o wywołanie funkcji next(). Tym samym będziemy teraz mogli

przetwarzać ciało żądania jako obiekt złożony z właściwości odpowiadających polom przesłanego formularza.

Jak w przypadku każdego z wspominanych tu pakietów – warto obok materiałów pochodzących z drugiej ręki przyjrzeć się samej dokumentacji pakietu dostępnej na www.npmjs.com

Obiekt routera w Express

Chcąc zmodularyzować zagadnienie routingu możemy utworzyć folder routes, a w nim pliki .js poświęcone różnym komponentom naszej aplikacji (np. funkcjonalności dostępnej dla poszczególnych ról użytkowników – np. admin.js, customer.js, guest.js)

W każdym z plików:

```
const express = require('express');

const router= express.Router();

     /* następnie definiujemy middleware dla odpowiednich URL i metod
     żądania */

router.use(…);

router.get(…);

router.post(…);

     /* oraz eksportujemy tak zdefiniowany obiekt routera */

module.exports = router;
```

W pliku głównym aplikacji (tu: app.js) skonsumujemy router następująco:

```
const nazwaKomponentu = require('./routes/nazwaKomponentu');

// jak wcześniej – bez .js w nazwie pliku

app.use(nazwaKomponentu);
```

Możliwe jest też wielopoziomowe filtrowanie ścieżek przez reguły routingu – zostaną zastosowane tylko wtedy, gdy URL żądania zawiera w sobie podany jako 1szy parametr segment adresu – np.

```
const komponentAdmin = require('./routes/admin');

app.use('/admin', komponentAdmin);
```

Prosta strona '404'

Jeśli nie zależy nam na wyrafinowanej formie strony z komunikatem o braku żądanego zasobu, możemy ją skonstruować nieomal pojedynczą linią kodu.

Zakładając, że wszystkie reguły obsługujące żądania o poprawnych / znanych nam adresach znalazły się powyżej, możemy po wszystkich klauzulach umieścić

`app.use( … )` w formie bez pierwszego parametru (będzie zatem przejmował dowolnej treści żądania, które nie zostały obsłużone wyżej. Wewnątrz funkcji obsługi możemy tam zamieścić np.:

```
res.status(404).send('<h1>Nie znaleziono strony</h1>');
```

Serwowanie widoków

W przypadku, gdy w odpowiedzi na niektóre żądania trzeba zwrócić czysto statyczny zasób, ale stosunkowo złożony, który trudno byłoby skomponować z niewielkiej liczby łańcuchów tekstowych, lepiej będzie przygotować i zapisać zwykły plik .html. Tego rodzaju widoki zgodnie z konwencją umieszczamy zwykle w folderze o nazwie views.

Pamiętajmy jednak, że żaden z plików składających się na projekt witryny działającej z Node.js nie jest domyślnie bezpośrednio dostępny do pobrania z serwera. Aby więc serwować pliki .html umieszczone w folderze views musimy jawnie uwzględnić je w regułach routingu. Różnica w tym przypadku będzie polegała na tym, że do serwowania treści użyjemy metody `sendFile()`. Np.:

```
const path = require('path');

// wykorzystanie modułu path należącego do zbioru Core Modules

res.sendFile(path.join(__dirname, 'views', 'about.html'));
```

W powyższym przykładzie wykorzystujemy predefiniowaną zmienną `__dirname` wskazującą na bieżący folder (to jest na folder zawierający bieżący plik) oraz dobudowujemy doń ścieżkę względną. Jeśli zaś ww. kod umieścimy w jednym z plików folderu routes, to potrzebny będzie jeszcze jeden argument przed ‘views’, pozwalający wynawigować do folderu nadrzędnego: '../' .

```
res.sendFile(path.join(__dirname, '../', 'views', 'about.html');
```

Udostępnianie statycznych zasobów

Osobną kategorię zasobów stanowią pliki z kodem czy np. obrazy powiązane ze stroną i konsumowane po stronie klienta. Dotyczy to m.in. plików CSS. Pożądane byłoby aby móc budować do nich intuicyjne odsyłacze oparte na ich faktycznej lokalizacji w systemie plików.

W tym przypadku konwencja przewiduje zgrupowanie ich (bezpośrednio lub w podfolderach o nazwach odpowiadających ich typowi – np. img, js, css) w folderze public.

Instrukcja, która pozwoli nam na takie adresowanie – czyniąc wnętrze folderu public korzeniem wirtualnych ścieżek to takich zasobów będzie:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Zapisywanie i przekazywanie danych

Nie będziemy tym razem rozwijać zagadnienia zarządzania danymi. Na potrzeby tego modułu warto jedynie zaznaczyć, że:

- częstokroć będzie potrzebne przekazywanie danych w postaci skonstruowanego ad-hoc obiektu: {wlasciwosc1: wyrażenie1, wlasciwosoc2: wyrażenie1} itd.
- jeśli utworzymy w naszej aplikacji zwykłą zmienną, to umieszczane w niej dane nie będą zindywidualizowane dla danej interakcji danego użytkownika z witryną (czyli nie będą miały cech tzw. zmiennych sesyjnych), ale będą dostępne w tym samym egzemplarzu dla różnych użytkowników (tj. będą zmiennymi aplikacyjnym)

Szablony w Express

Pracując z Express możemy zastosować różne technologie szablonów. Najpopularniejszymi rozwiązaniami tego typu dla Node.js/Express są EJS (*Embedded JavaScript*), Pug oraz Handlebars.

Warto tu przytoczyć historyczne rozróżnienie interfejsów programowania po stronie serwera na tzw. rodzinę CGI oraz rodzinę SSI. CGI (*Common Gateway Interface*) jest technologią która spopularyzowała dynamiczne strony WWW przez standaryzację interfejsu pomiędzy serwerem WWW a programem realizującym generowanie dynamicznej strony WWW. CGI reprezentuje wobec tego podejście, w którym program jest odpowiedzialny za skonstruowanie całej odpowiedzi do klienta (to jest całej zawartości dokumentu oraz ewentualnie nagłówków protokołowych). SSI (*Server-Side Includes*) stał się z kolei prekursorem odmiennego podejścia. Ponieważ częstokroć dynamicznie generowane muszą być tylko nieliczne elementy strony, to w SSI zaproponowano, by tworzyć strony analogicznie jak strony statyczne HTML, a zagadnienia wymagające programowania realizować przez odpowiednio "opakowane" w znaczniki wstawki kodu umieszczane wewnątrz strony. Mogły to być pojedyncze wyrażenia produkujące w dokumencie wynikowym określony rezultat, większe bloki instrukcji, czy też np. polecenia wkomponowania w danym miejscu strony zawartości innego pliku (*include*). Oczywiście z racji ww. rozszerzeń dokument w stylu SSI nie może być bezpośrednio po odczytaniu z dysku przekierowany do klienta, ale wymaga przetworzenia po stronie serwera przez tzw. silnik (*engine*) który zapewni przetworzenie dokumentu na postać wynikową.

O ile ze stylem CGI mieliśmy styczność przy pierwszych przykładach programowania odpowiedzi serwera w Node.js, o tyle przykładem stylu SSI będą właśnie szablony.

Przez szablony rozumiemy tutaj technologię dedykowaną dla budowy widoków w aplikacji, ułatwiającą konstrukcję statycznych elementów strony, osadzanie w nich elementów dynamicznych oraz wyodrębnianie elementów powtarzalnych czy konfigurowalnych (poprzez tzw. *partials*). Dobrą praktyką (przynajmniej w przypadku programowania po stronie serwera) jest konsekwentne separowanie zagadnień prezentacji od implementacji. Oznacza to u nas minimalizację kodu JS osadzanego bezpośrednio w takim widoku.

Często występującą cechą technologii templates jest wręcz zawężanie dostępnych w widoku konstrukcji programistycznych w porównaniu z pełną mocą języka programowania ogólnego przeznaczenia.

Ponadto (jak to ma miejsce np. w Pug) występują w niektórych z tych technologii rozwiązania składniowe pozwalające na bardziej zwięzłe deklarowanie znaczników (np. poprzez podanie samej nazwy elementu oraz nadanie określonego znaczenia znakom białym (np. wcięciom dokonanym za pomocą znaków tabulacji).

Szablony EJS – podstawy

EJS współdziała z Express, stanowi jednak odrębny pakiet wymagający instalacji:

```
npm install --save ejs
```

W aplikacji musimy zadbać o skonfigurowanie szablonów, to jest wskazanie stosowanego przez aplikację silnika szablonów – tu EJS oraz określenia lokalizacji plików zawierających widoki:

```
app.set('view engine', 'ejs');

app.set('views', 'views');

// powielamy tu explicite ustawienie domyślne lokalizacji foldera widoków
```

Teraz potrzebujemy przygotować dokumenty w postaci plików .ejs umieszczonych w folderze views. Wersją początkową takiego dokumentu może być zwyczajny, w pełni statyczny dokument HTML.

Teraz, w odpowiedniej regule routingu możemy zawrzeć wywołanie

```
res.render('nazwa');
```

gdzie 'nazwa' odpowiada nazwie pliku .ejs bez rozszerzenia. Zwykle natomiast będziemy przekazywać do widoku dane służące skonstruowaniu jego dynamicznych elementów. W tym przypadku drugim parametrem jest ów obiekt:

```
res.render('nazwa', {parametr: wartość, … });
```

Wyrażenia (np. służące wstawieniu jakiegoś atrybutu w strukturę wynikowego dokumentu) umieścimy w znacznikach:

```
<%= … %>
```

Natomiast instrukcje umieszczamy w znacznikach:

```
<% … %>
```

Również charakterystyczny dla stylu SSI wzorzec z większymi blokami statycznej zawartości wkomponowywanymi w instrukcję warunkową. Np.:

```
<% if(disclaimerToShow) { %>
<div> … tutaj treść HTML … </div>
<% } %>
```

Wzorzec ten może się powtarzać poprzez zagnieżdżanie. Np. często będziemy wizualizować listy danych przekazanych w postaci tablicy obiektów – wówczas powtarzalne składniki treści można będzie umieścić np. w ramach pętli – np.

```
for( let p of produkty ){ … }
```

Podsumowanie

W niniejszym module zapoznaliśmy się z udogodnieniami ramy Express dotyczącymi routingu oraz budowy widoków. W kolejnych modułach zajmiemy się zarządzaniem danymi – zarówno dotyczącymi sesji klienta w aplikacji jak i danymi trwałymi.

Zadanie

1. Sporządź bez użycia ramy Express prostą funkcjonalność złożoną z formularza z jednym polem i strony pobierającej rezultat, która wyświetli wartość przesłanego parametru.
2. Następnie utwórz drugi wariant tej funkcjonalności, w którym odebrany parametr będzie zapisany do pliku, zaś odpowiedzią do klienta będzie zlecenie przekierowania do strony z formularzem.
3. Zbuduj złożony z kilku formatek formularz oraz stronę konsumującą jego wyniki poprzez wypisanie ich jako odpowiednio zaaranżowanej treści w dokumencie HTML. Strona prezentująca wyniki powinna wykorzystywać EJS oraz prosty zewnętrzny arkusz CSS.

UWAGA! Wieloplikowe rozwiązania proszę pakować do pojedynczego .zip lub .rar !