MODUŁ 8:
Node.js / Express - wzorzec MVC i trwałości danych

Cel modułu

Moduł stanowi kontynuację materiału modułu 07. Wprowadzimy dalsze środki porządkujące i modularyzujące aplikację oraz podstawowe narzędzia dla zarządzania danymi trwałymi (pliki, relacyjna baza danych) oraz danymi interakcji z użytkownikiem (dane sesyjne oraz cookies).

Efekty kształcenia

Niniejsza lekcja służy osiągnięciu następujących efektów kształcenia.

Wiedza

Po ukończonym kursie student / studentka:

- Przedstawia konwencje nazewnicze i szczegóły techniczne realizacji wzorca MVC w Express,
- Wskazuje dostępne w ramach Node.js / Express środki do zarządzania warstwą modelu.

Umiejętności

Po ukończonym kursie student / studentka:

- Buduje widoki z wykorzystaniem *partials*;
- Realizuje trwałość danych w aplikacji opartą na plikach oraz serializacji JSON;
- Realizuje trwałość danych w aplikacji opartą na SQL.

Uzasadnienie

Spośród nieomówionych jeszcze zagadnień budowy aplikacji w Node.js oraz Express, najistotniejsze dla osiągnięcia faktycznych kompetencji budowy aplikacji WWW jest opanowanie zagadnień zarządzania złożonością oraz unikania redundancji w kodzie oraz rozwiązań dotyczących zarządzania trwałością danych.

Widoki w EJS – c.d.

Partials

W poprzednim module wykorzystywaliśmy szablony EJS dla produktywnego połączenia składników statycznych strony z elementami dynamicznymi. Pozostaje jednak problem powielania wspólnych elementów stron w poszczególnych plikach widoków. Problem ten możemy rozwiązać za pomocą tzw. *partials* czyli zasobów reprezentujących fragment docelowego widoku.

W tym celu powołajmy folder w ramach folderu views – często stosowaną nazwą jest includes, a zatem przyjmijmy, że nasze zasoby częściowe znajdą się w: /views/includes

Możemy w ten sposób wyodrębnić np. wspólne dla wszystkich stron naszej witryny fragmenty nagłówka strony, stopki albo elementu nawigacyjnego.

Po zgrupowaniu odpowiednich treści w przygotowanych plikach *partials* – używamy do ich skomponowania w dokumencie docelowym odpowiednio rozmieszczonych wywołań dostarczanej przez EJS funkcji include.

Przypomnijmy, że wyrażenia JS mogliśmy w EJS wkomponowywać stosując znacznik wyrażenia:

```
<%= … %>
```

Zastosowanie go jednak do sprowadzenia zasobu z innego pliku EJS nie da pożądanego efektu, gdyż kod HTML wytworzony przez wyrażenia osadzone w tym znaczniku podlega tzw. *escaping-owi*. Dlatego też potrzebujemy innego wariantu znacznika pozbawionego tej właściwości:

```
<%- … %>
```

A więc np.:

```
<%- include('includes/naglowek.ejs') %>
```

Celem zapoznania się z pozostałymi wariantami znaczników stosowanych w EJS – zob. https://ejs.co/#docs

Kontrolery

Wyodrębnienie kontrolerów będzie raczej prostym zabiegiem. Potrzebujemy w tym wypadku wydobyć po prostu funkcje, które przekazywaliśmy jako callback do definicji routingu.

W tym celu załóżmy folder `controllers` i utwórzmy w nim pliki .js (nazwa zależna od obsługiwanego obszaru funkcjonalności – np. `produkty.js`) odpowiednio do potrzeb naszej aplikacji grupujące funkcje obsługi żądania.

Przypomnijmy – aby funkcja była konsumowalna spoza pliku jej implementacji, powinna być przypisana w ramach właściwości `exports`:

```
exports.funkcjaObslugi = (req, res) => { … };
```

W pliku z regułami routingu należy zaimportować ww. plik tak samo, jak wykonywaliśmy to w poprzednich przykładach:

```
const funkcjaA = require('./controllers/produkty');
```

Modele

W podobny sposób możemy wyodrębnić modele. Realizacją wzorca modelu w obiektowych językach programowania wiąże się zwykle z powołaniem, w dedykowanym jej pliku, klasy reprezentującej dane pojęcie obsługiwane przez aplikację (np. Produkt), wyposażenie jej w odpowiednie dla dziedziny problemowej atrybuty oraz obudowanie mechanizmami walidacji danych oraz metody związane z operacjami na tych danych (modyfikacja, usuwanie, listowanie,…) czy wreszcie samo utrwalanie danych. To, w jakim stopniu realizujemy te zadania explicite kodem języka programowania, a w jakim są one dla nas realizowane dzięki odpowiednim ustawieniom konfiguracyjnym czy zachowaniu odpowiedniej konwencji nazwowej, zależy od stosowanej ramy programistycznej. Tutaj większość tych zadań będziemy oprogramowywać tradycyjnie.

Z powyższego wynika, że jeśli np. tworzymy nową instancję jakiegoś pojęcia w naszej aplikacji (np. na skutek wprowadzenia przez użytkownika danych z formularza na stronie), to obsługa tej akcji będzie się po prostu wiązała z utworzeniem nowego obiektu klasy zdefiniowanej w ramach modelu.

Jako wyjściowy szkielet takiego modelu rozważmy następujący kod:

```
const products = [];
```

```
module.exports = class Produkt {
    constructor(n, c) {
        this.nazwa = n;
        this.cena = c;
    }

    save() {
        products.push(this);
    }

    static fetchAll() {
        return products;
    }
}
```

W tym wypadku dane mają charakter ulotny – są przechowywane jedynie w zmiennej typu tablicowego.

Aby zapewnić trwałość tych danych, możem skorzystać ze znanych już mechanizmów obsługi plików dyskowych. Zakładając, że plik będziemy chcieli nazwać `produkty.json` oraz przechowywać go w folderze `data`, funkcja dodająca nowy obiekt mogłaby wyglądać następująco:

```
    save(){
        const p = path.join(
                    path.dirname(process.mainModule.filename),
                    'data', 'produkty.json'
                    );
        fs.readFile(p, (err, fileContent) => {
           let products = [];
           if(!err){
               products = JSON.parse(fileContent);
           }
           products.push(this);
           fs.writeFile(p, JSON.stringify(products), (err) => {
                console.log(err); } );
           });
    }
```

W większości bazujemy tu na środkach operacji na plikach znanych już z poprzednich modułów. Zwróćmy uwagę na:

- dynamiczne ustalenie ścieżki bezwzględniej na podstawie właściwości dostępnej w obiekcie process;
- wykorzystanie parametru `err` do reagowania na sytuację, gdy plik (jeszcze) nie istnieje;
- deserializacja i serializacja danych za pomocą metod dostępnych w klasie JSON.

Budując analogiczną funkcjonalność dla metody pobierania danych, proszę zwrócić uwagę, że komplikacją w tym przypadku wyniknie z asynchronicznego charakteru metody `readFile()`. Tutaj nie będziemy w stanie wprost zwrócić odczytanych danych za pomocą instrukcji return. Zamiast tego należy wyposażyć metodę `fetchAll()` w parametr, którym będzie przekazywana jako callback funkcja: przyjmie ona jako swój argument odczytane dane i zajmie się ich przekazaniem do widoku.

Model a parametry routingu

W powyższym przykładzie nasze obiekty modelu były pozbawione numerycznych identyfikatorów. Warto zadbać o ich dodanie, gdyż przy realizacji typowego wzorca funkcjonalności CRUD (create/retrieve/update/delete) tworzymy akcje, które wymagają w większości przypadków jednoznacznej identyfikacji obiektu (wyświetlenie szczegółów, usunięcie, modyfikacja).

Zakładając, że obiekt posiada identyfikator, możemy np. zbudować funkcjonalność wyświetlania szczegółów. Ponieważ jest to operacja niemająca efektów ubocznych (nie modyfikuje stanu obiektów na serwerze) to realizowana będzie za pomocą żądania typu GET.

Nie będziemy tu omawiać rutynowej funkcjonalności takiej jak generowanie identyfikatora dla nowo dodawanych obiektów czy też generowania hiperłącza do widoku szczegółowego, w którym ów identyfikator należy dołączyć.

Natomiast istotną do wykorzystania tutaj właściwością Express będzie obsługa wydobywania parametrów z adresu URL. Mianowicie, w metodach tworzących reguły routingu, parametr URL uwzględnia znak specjalny w postaci dwukropka. W przypadku napotkania tego znaku, znak ten oraz znajdująca się za nim sekwencja znaków (aż do najbliższego znaku / albo dwukropka).

Zatem, jeśli w regule routingu mamy np.

```
router.get('/produkty/:nazwaParametru', nazwaFunkcji);
```

To reguła ta może zostać dopasowana np. do adresu:

```
/produkty/12345
```

i w funkcji obsługi żądania możemy odwołać się do tak wkomponowanego w adres URL parametru poprzez wyrażenie:

```
req.params.nazwaParametru
```

Jeśli zechcemy tę funkcjonalność wykorzystać do budowy strony podglądu szczegółów pojedynczego obiektu, to nasza klasa modelu będzie wymagała w tym celu kolejnej statycznej metody, która podobnie jako `fetchAll()` zostanie wyposażona w parametr callback.

Query string

Nieco inaczej ma się sprawa z *query string*, czyli fragmentem URL znajdującym się po znaku zapytania. Tej części URL nie reprezentujemy explicite we wzorcu ścieżki w regule routingu. Może zawierać dowolnie wiele parametrów – np. `/add-product?par1=x&par2=y;par3=z`

Możemy dane te bez dodatkowych zabiegów odzyskać, już jako strukturę obiektu, sięgając do właściwości `req.query`, a więc np.

```
req.query.par1
```

Pamiętajmy jednak, że niezależnie od ich wartości (np. czysto numeryczne, czy true / false) zawsze będą traktowane jako typu string.

Utrwalanie danych za pomocą MySQL

Zalety wykorzystania systemów zarządzania bazami danych dla utrwalania danych w aplikacji powinny nam być już znane z innych dedykowanych temu zagadnieniu przedmiotów. W zakresie relacyjnych baz danych powinny nam być znane m.in. takie pojęcia jak:

- relacja / tabela;
- klucz główny, klucz obcy;
- asocjacje (*relationships*): 1-do-wiele; 1-do-1; wiele-do-wiele;
- spójność danych i integralność referencyjna;
- normalizacja;
- schemat;
- wartości NULL.

System MySQL

Odpowiednią dla naszego systemu operacyjnego instalację MySQL możemy uzyskać na stronie: https://www.mysql.com/

Interesować nas będzie wersja Community – zależnie od systemu operacyjnego, za pomocą pojedynczego lub dwóch pakietów potrzebujemy:

- MySQL Community Server
- MySQL Workbench

Dla zgodności z interfejsami dostępnymi w Node.js, podczas instalacji powinniśmy wybrać: *Use Legacy Authentication Method*

Po zainstalowaniu systemu, jako minimalny scenariusz zmierzający do potwierdzenia, że jesteśmy w stanie posługiwać się bazą danych z poziomu naszej aplikacji, wykonajmy w ramach Workbench:

- Connect to Database
- Schemas > Create

Następnie, w ramach nazwanego schematu (np. 'sklep') spróbujmy utworzyć tabelę. Warto będzie tu zadbać m.in. o utworzenie kolumny z identyfikatorem (właściwości kolumny: PK, auto-increment, unsigned, not null).

Stwórzmy też przynajmniej jeden wiersz w ramach tej tabeli, aby móc kierować potem doń zapytania z poziomu aplikacji Node.js.

Po stronie Node.js potrzebujemy pakietu zawierającego adaptery dla MySQL.

```
npm install --save mysql2
```

Warto też będzie powołać odrębny plik `database.js`, a w nim skonstruować połączenie z bazą danych.

```
const mysql = require('mysql2');


const pool = mysql.createPool({
      host: 'localhost',
      user: 'root',
```

```
      database: 'sklep',
      password: 'root'
});
module.exports = pool.promise();
```

W powyższym kodzie założono, że schemat naszej bazy nosi nazwę sklep oraz że (co oczywiście w docelowym środowisku niedopuszczalne) hasło do bazy danych brzmi `root`.

Zamiast opierać się na każdorazowym otwieraniu i zamykaniu połączenia przy dostępie do bazy, wybrano tutaj wariant tzw. puli połączeń. Polega ona na zarządzaniu przez adapter pulą otwartych połączeń i dysponowaniu ich przy wykonywanych na bazie operacjach.

Warto też dodać, że interfejs MySQL wykorzystuje stosowany w JavaScript mechanizm tzw. obietnic (*promises*). Obietnica jest obiektem, na którym możemy wywoływać operacje `.then()` oraz `.catch()`, które odpowiednio definiują reakcje na pomyślne wykonanie operacji (i dostępność jej rezultatu) oraz na błąd (i dostępny dlań komunikat błędu). Programowanie oparte na promises pozwala uniknąć niedogodności związanych z powstawianiem rozbudowanych zagnieżdżeń funkcji callback w programowaniu asynchronicznym.

W pliku aplikacji wykorzystujemy stworzony wyżej mechanizm następująco. Zakładając, że mamy w naszym schemacie tabelę o nazwie produkty, spróbujmy następującego mimimalistycznego kodu by przekonać się, czy możemy połączyć się z bazą.

```
const db = require('./util/database');
// ...
      db.execute('select * from produkty')
      .then(res => {
            console.log(res);
      } )
      .catch(res => {
            console.log(res);
      } ) ;
```

Parametr res w funkcji przekazanej do metody `then()` będzie zawierał rezultat uzyskany z bazy danych. Warto się zapoznać z jego budową – jest on stosunkowo obszerną zagnieżdżoną tablicą.

Zbadajmy na podstawie powyższego przykładu zwracane rezultaty. Teraz możemy przystąpić do budowy wersji naszej aplikacji wykorzystującej bazę relacyjną zamiast pliku. Przytoczony wyżej przykład zapytania przyda nam się do metody `fetchAll()`. Dzięki zastosowaniu *promises* możemy tutaj uniknąć używania callback i po prostu zwrócić rezultat wywołania metody `execute()`.

Definicję obsługi rezultatu w postaci łańcucha `.then().catch()` umieścimy więc już wewnątrz wywołującej `fetchAll()` funkcji kontrolera.

Ze względu na zagnieżdżenie zwracanej struktury, możemy uprościć wyłuskanie z niej interesujących nas elementów za pomocą tzw. destrukturyzacji (*destructuring*) odniesionej w tym przypadku do tablicy:

```
.then( ([wiersze, metadaneKolumn]) => { ... obsługa rezultatu ...})
```

Taki zapis spowoduje wydobycie z rezultatu dwóch pierwszych elementów tablicy (w tym przypadku samych będących tablicami) i przypisanie ich do zmiennych `wiersze` oraz `metadaneKolumn`.

Więcej nt. destrukturyzacji - zob. np. https://codeburst.io/es6-destructuring-the-complete-guide-7f842d08b98f

Przytoczona wyżej metoda `execute()` działa również dla instrukcji typu DML, a więc m.in. możemy ją użyć z klauzulą `insert`. Zakładając, że nasza przykładowa tabela miałaby kolumny nazwa, cena, opis, zaś samo wywołanie znajdowałoby się w metodzie klasy `Produkt` będącej definicją modelu, operację taką można będzie wykonać przekazując dwa parametry jak niżej:

```
'insert into produkty (nazwa, cena, opis) values (?, ?, ?)',[this.nazwa, this.cena, this.opis]
```

Widzimy tutaj, że obsługiwane są zapytania parametryzowane. Jest to bezsprzecznie właściwsze rozwiązanie niż próba samodzielnej konkatenacji łańcuchów, rodząca częste zagrożenia typu *SQL injection* (zob. następny moduł).

Analogicznie będziemy postępować budując metodę wyszukującą pojedynczy wiersz w naszej bazie (na potrzeby wyświetlenia widoku szczegółowego). W takim przypadku jedynym przekazywanym parametrem będzie identyfikator. Tam również możemy pozwolić sobie na zwracanie promise do przetworzenia jej w funkcji kontrolera.

W tej sekcji zapoznaliśmy się z utrwalaniem danych poprzez bezpośrednie operacje na bazie danych. W praktycznych aplikacjach jest to podejście pracochłonne, stąd warto będzie zainteresować się udogodnieniami dla odwzorowań obiektowo-relacyjnych, których przykładem w ramach Node.js jest pakiet Sequelize.

Zarządzanie danymi interakcji z klientem

Cookies

Pojęcie cookies powinno nam być już znane w ramach zagadnień protokołu HTTP. Przypomnijmy, że w założeniach jest to element danych który serwer może powierzyć klientowi celem ominięcia ograniczeń bezstanowości protokołu HTTP w przypadku konieczności skorelowania pewnej sekwencji żądań (np. włożenie towaru do koszyka i następnie sfinalizowanie transakcji, uwierzytelnienie się i następnie dostęp do funkcjonalności objętej kontrolą dostępu, czy też np. zdefiniowanie preferencji a następnie wykorzystanie funkcjonalności serwisu je uwzględniających). Zatem w pierwotnym zamyśle zarówno tworzenie takich danych, jak i ich modyfikacja oraz odczyt są wykonywane przez funkcjonalność serwerową. Natomiast przeglądarki dają w określonych warunkach możliwości manipulowania ciasteczkami również skryptom działającym po stronie klienta.

Ustawić cookie możemy za pośrednictwem obiektu res (co nie powinno być niespodzianką dla osób znających sposób realizacji mechanizmu w protokole HTTP, gdzie cookie zakładane jest odpowiednim nagłówkiem odpowiedzi serwera):

```
res.setHeader('Set-Cookie', 'nazwaParametru=wartosc');
```

Z tych samych względów (nagłówek protokołowy poprzedza ciało) musimy pamiętać, by powyższą instrukcję w kontrolerze wywołać przed konstruowaniem ciała odpowiedzi. Poza samą wartością, pamiętajmy o możliwości skonfigurowania parametrów cookie, w tym:

- flaga `Secure`
- flaga `HttpOnly`
- właściwość `Expires` albo `Max-Age`

- właściwości `Domain` oraz `Path`

Wiedząc, że tak założone cookie będzie przez klienta przedkładane jako zawartość nagłówka o nazwie Cookie, możemy ją odzyskać przy kolejnych żądaniach:

```
req.get('Cookie')
```

Dane sesyjne

Innym mechanizmem, również częściowo opartym na cookies, są tzw. dane sesyjne. Są to zmienne powoływane osobno dla każdej z sesji użytkownika (rozumianej tutaj jako sekwencja żądań do naszej witryny kierowanych z tego samego okna przeglądarki) i zapisywane po stronie serwera. W tym wypadku cookie potrzebne jest nam dla określenia tzw. korelacji, czyli skojarzenia, z którą sesją (z potencjalnie wielu nawiązanych w danym okresie czasu z witryną) związane jest dane żądanie. Jest to realizowane jako tzw. identyfikator sesji.

Celem wykorzystania danych sesyjnych powinniśmy zaopatrzyć się w pakiet express-session:

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

Importujemy go w znany już sposób do aplikacji:

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

oraz wykorzystujemy jako funkcję wywołaną w ramach konstrukcji komponentu middleware obsługującego sesję:

```
app.use(session({secret: 'jakiś Łancuch Znaków', resave: false, saveUninitialized: false}));
```

Powyżej podano proponowaną dla naszych potrzeb konfigurację. Właściwość `secret` parametryzuje sposób generowania losowego indentyfikatora sesji, `resave` wskazuje, czy zapis sesji powinien następować po każdym żądaniu, czy tylko w przypadku zmiany, zaś `saveUninitialized` określa, czy zapisywać dane sesji przed ich pierwotną modyfikacją.

Dalsze szczegóły można znaleźć m.in. na: https://www.npmjs.com/package/express-session

Teraz możemy posługiwać się właściwością session w obiekcie req:

```
req.session.nazwaWlasciwosci = wartość;
```

Natomiast wyczyścić dane sesyjne można za pomocą metody `destroy()`, której podajemy w formie metody callback instrukcje przeznaczone do wykonania po dokonaniu usunięcia.

```
req.session.destroy( metodaCallback );
```

Uwaga! Powiązanie pojęć cookie i sesja jest dwojakiego rodzaju i okoliczności te nie powinny być mylone. Po pierwsze, pojęcie ciasteczka sesyjnego (session cookie) oznacza dowolne cookie bez sformułowanego explicite terminu ważności. Tym samym będzie ono "żyło" nie dłużej niż do momentu zamknięcia danego okna przeglądarki. Po drugie, jak podano wyżej, cookies (zwykle właśnie sesyjne) służą przekazywaniu identyfikatora sesji.

Podsumowanie

W niniejszym module zapoznaliśmy się ze sposobem reorganizacji kodu aplikacji zgodnym ze wzorcem MVC. Następnie poznaliśmy podstawowe mechanizmy utrwalania danych w plikach i w relacyjnych bazach danych oraz podstawowe środki dla zarządzania danymi dotyczącymi interakcji z użytkownikiem.

Zadanie

Zbuduj funkcjonalność witryny opartą na Express i zbieżną z tematem Twojego projektu semestralnego. Implementacja powinna być zorganizowana według omówionej tutaj struktury wzorca MVC. Funkcjonalność powinna dotyczyć pojedynczej wybranej encji (np. Pracownik, Produkt, Artykuł itp.) i obejmować:

a. stronę listującą zgromadzone w systemie instancje tej encji
b. stronę formularza do dodawania nowej instancji
c. stronę obsługującą żądanie dodania nowej instancji
d. stronę wyświetlającą szczegóły (wszystkie atrybuty) wybranego obiektu

Mechanizmem zapewnienia trwałości danych powinny być – do wyboru – plik JSON albo baza MySQL.

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