**Projekt systemu realizującego operacje na bazie danych Northwind**

Informatyka (niestcjonarne), **Bazy Danych 2020**  
*AGH Wydział Informatyki, Elektroniki i Telekomunikacji*

**Skład zespołu**

* Mateusz Gałka  
  Nazwa na github: lambdaforg
* Dawid Karaś  
  Nazwa na github: dawkaras
* Kacper Kondratek  
  Nazwa na github: kKondratek
* Krzysztof Wicher  
  Nazwa na github: krwicher

**Technologie**

* Java Spring Boot
* MongoDB

**Instalacja MongoDB**

Pobieramy oraz instalujemy serwer MongoDB, który jest graficznym interfejsem do obsługi bazy danych (<https://www.mongodb.com/try/download/community>). Jest on bardzo prostym i intuicyjnym programem, który pozwala zarządzać bazami dancyh mongo. Kolejnym krokiem jest dodanie folderu bin do zmiennych środowiskowych, będzie to niezbędne do wykonania kolejnych kroków. Następnie w wierszu poleceń (najlepiej w trybie administratora) wykonujemy polecenie

mongod --dbpath F:\MongoData\ --logpath F:\MongoLogs\mongolog.log

gdzie podajemy ścieżkę, w której przechowywane będą pliki baz danych oraz drugą do pliku, w którym będą przechowywane logi. Do wystartowania serwera mongoDB posłuży nam poniśże polecenie:

net start MongoDB

od tej pory serwer mongo powinien automatycznie działać w tle przez cały czas.

**Tworzenie projektu Java Spring Boot**

Do stworzenia projektu skorzystamy z oficjalnej strony <https://start.spring.io/>, dzięki której można łatwo wygenerować podstawowy projekt. Wybieramy język Java, projekt Maven, wersję Spring Boot 2.3.6, oraz wersję Javy 11. Kolejnym krokiem jest wybranie odpowiednich zależności "dependecies", będą nam potrzebne: Thymeleaf do obsługi HTML, Spring Data for MongoDB do obsługi bazy danych oraz Spring Web, ponieważ wybraliśmy styl architektury systemu MVC.

**Konfiguracja bazy danych w projekcie**

Do edytowania projektu możemy użyć aplikacji od JetBrains pt. IntelliJ. Aby skonfigurować połączenie z serwerem mongoDB posłużyliśmy się plikiem

src/main/resources/application.properties

gdzie dodajemy kod odpowiedzialny za połączenie z bazą (nazwa bazy, port oraz host):

spring.data.mongodb.host=localhost

spring.data.mongodb.port=27017

spring.data.mongodb.database=Northwind

Baza danych zostanie utworzona automatycznie tak jak późniejsze kolekcje, dlatego do konfiguracji wystarczy samo połączenie, a później stworzenie odpowiednich kolekcji.

**Mapowanie kolekcji**

Mapowanie kolekcji w przypadku naszego projektu jest dość prostym elementem kodowo natomiast warto poświęcić więcej czasu na rozplanowanie kolekcji, ponieważ to znacznie może wpłynąć na późniejszą pracę. Trzeba dobrze przeanalizować potrzeby oraz wymagania projektu i dostosować podstawową wersję diagramu relacji bazy Northwind, ponieważ bazowe relacje mogą być nieodpowiednie do osiągnięcia zamierzonych celów. Pierwszym krokiem do odwzorowania kolekcji w Javie jest stworzenie klasy, która będzie reprezentować wybrany szablon dokumentu. Najważniejszym elementem, aby kolekcja prawidłowo była obsługiwana przez Javę jest dodanie atrybutu do klasy:

@Document(collection = "nazwa kolekcji")

dane w klasie mogą być w kilku formatach, głównie korzystamy z typów prostych (np. String), typów złożonych (np. obiekt) oraz tablic. Pola przykładowego dokumentu w Javie mogą wyglądać tak:

@Id

public String id;

public String companyName;

public String contactName;

public String contactTitle;

public String address;

public String city;

public String region;

public String country;

public String phone;

public String fax;

public ArrayList<String> orders;

public CustomerDemographic customerDemographic;

Bardzo ważnym elementem jest atrybut

@Id

który jest kluczem podstawowym w dokumencie. Możemy zauważyć kilka różnych typów pól, przykładem pola złożonego, które jest obiektem jest pole

public CustomerDemographic customerDemographic;

natomiast tablicą elementów (w tym wypadku jest to tablica ciągów znaków natomiast równie dobrze mogłaby to być tablica obiektów) jest pole

public ArrayList<String> orders;

**Zarządzanie kolekcjami**

Dzięki dodaniu podczas generowania projektu Javy zależności od Mongo jesteśmy w stanie w bardzo prosty sposób stworzyć repozytorium, które pozwoli na wykonywanie podstawowych operacji na danej kolekcji. Pierwszym krokiem będzie stworzenie interfejsu, który będzie reprezentować przykładowe repozytorium, a następnie rozszerzenie parametryzowanego interfejsu MongoRepository, którego pierwszym parameterem jest zmapowany wcześniej dokument, a drugim typ klucza podstawowego.

public interface CustomerRepository extends MongoRepository<Customer, String>

Dzięki konfiguracjom przeprowadzonym w poprzednich krokach możemy już korzystać z bazy danych, podstawowymi operacjami, które udostępnia nam rozszerzone repozytorium są:

* findAll
* insert
* saveAll
* count
* delete
* deleteAll
* deleteById
* existsById
* findAllById
* findById
* save
* exists
* findOne

a także nie musimy tworzyć połączenia z bazą, ponieważ utworzone przez nas repozytorium oraz konfiguracja połączenia z bazą robią to za nas, dlatego możemy wykonywać w/w operacje. Do wykonania operacji potrzebna nam jest instancja stworzonego przez nas repozytorium (może być stworzona dynamicznie, jednak zalecamy wstrzykiwanie zależności, aby ograniczyć zależności pomiędzy klasami), a następnie możemy wykonać przykładową operację:

repository.findAll()

Dzięki udostępnionym funkcjom przez MongoRepository możemy w szybko tworzyć proste zapytania typu:

Product findFirstById(int productId);

List<Product> findAllByNameContains(String name);

List<Product> findAllByUnitPriceBetween(double priceFrom, double priceTo);

bez implementowania ich. Repozytorium samo rozpoznaje proste zapytania i pozwala od razu je wykorzystywać.

**Obsługa widoków**

Dzięki zastosowaniu architektury MVC widoki są tworzone w projekcie Javy i w nim obsługiwane. Pierwszym krokiem będzie stworzenie kontrolera do zarządzania widokami. Będzie to klasa, która musi mieć atrybut:

@Controller

a odpowiednie metody pod odpowiednimi adresami będą zwracać wybrane widoki np.:

@GetMapping()

public String getHome(Model model){

List<Product> list = new ProductService(productRepository).getProducts();

model.addAttribute("products", list);

return "base";

}

Atrybut metody obsługuje typ zaptania (POST, GET, PUT, DELETE), natomiast w konstruktorze możemy podać ściężkę zapytania jeśli ma być inna niż domyślna. Kolejnum krokiem będzie stworzenie widoku w formacie HTML. Stosując Thymeleaf możemy w prosty sposób wysyłać stworzone widoki do odpowiednich zapytań klientów. Wymaga to jedynie dodawania do znaczników odpowiednich atrybutów np.:

<div th:fragment="products" >

dzięki wprowadzaniu do div'a nazwy products możemy korzystać z niego w funkcji stworzonej w kontrolerze.

**System rejestracji i logowania**

Do autentykacji wykorzystany został Spring Security Core. Do bazy zostały dodane dwa dokumenty User oraz Role. Gdzie jedno odpowiada za przechowanie informacji o użytkowniku, jego zabezpieczonym haśle i mailu oraz przypisanych ról. Dokument Role przechowuje dwie role zwykłego użytkownika oraz role administratora.   
Dzięki zastosowaniu ról zabezpieczyliśmy część aplikacji odpowiedzialną za tworzenie produktów, kategorii i dostawców oraz za przeglądanie raportów.