Laboratorium 8. *Spring Boot* i *DTO*

Cel zajęć

Realizacja zadań z laboratorium pozwoli studentom poznać zasady stosowania obiektów transferu danych *DTO* [2] i wykorzystać je w *Spring REST API*.

Zakres tematyczny

- Przygotowanie *REST API* za pomocą narzędzia *Spring Initializr*, bez zastosowania obiektów *DTO* (jak w poprzednim laboratorium).
- Definicja klas encji z wykorzystaniem relacji *JPA*.
- Modyfikacja aplikacji i zastosowanie obiektów transferu danych *DTO* (ang. *Data Transfer Object*).
- Zastosowanie w aplikacji biblioteki *MapStruct*, w celu mapowania obiektów encji w obiekty *DTO*.

Wprowadzenie

W aplikacjach w języku *Java* z współpracujących *Hibernate* czy *Java Persistence API*, aby użyć danych korzysta się z różnych obiektów (*POJO*, J*avaBean*, klas encji, *DAO*, *DTO*). Warto usystematyzować, co charakteryzuje te obiekty:

- Klasa ***JavaBean*** musi przestrzegać pewnych konwencji dotyczących nazewnictwa, budowy i zachowania metod:
  - powinna implementować interfejs *Serializable*,
  - posiada konstruktor bezargumentowy,
  - umożliwiać dostęp do właściwości pól prywatnych za pomocą metod *get* i *set*.
- Klasa ***POJO*** (ang. *Plain Old Java Object*) jest podobna do *JavaBean*, przechowuje atrybuty i pozwala je modyfikować (za pomocą metod ***get*** i ***set***), ale nie musi implementować żadnego interfejsu i posiadać konstruktora bezargumentowego. Według definicji klasa *POJO* nie powinna odnosić się do żadnego szkieletu programistycznego.
- Klasa ***encji*** jest modyfikowalna (zawiera metody ***set*** do ustawiania wartości pól reprezentujących kolumny tabeli), ale użycie adnotacji ***@Entity***, ***@Table***, ***@Column*** z *JPA* łamie zasady definicji *POJO*, uzależniając się od *Spring JPA*. Jednak w dokumentacji *Hibernate* znajduje się odniesienie do klas encji, jako do obiektów *POJO* z adnotacjami *javax.persistence.Entity*. Jeżeli logika encji jest prosta, opiera się na danych z encji, nie zawiera skomplikowanej logiki to można traktować encję jako *POJO*.

- Klasa ***DAO*** (ang. *Data Access Object*) zapewnia dostęp do źródła danych (plik, baza danych). Najczęściej jest to interfejs, ponieważ łatwo można wtedy zmienić źródła danych.
- Klasa ***DTO*** (ang. *Data Transfer Object*) definiuje obiekt transferu danych i w programowaniu obiektowym oznacza obiekt, który przechowuje tylko pola publiczne, bez metod. Często używany jest do komunikacji z bazami danych w bibliotekach *ORM*. Podstawowym zadaniem *DTO* jest transfer danych pomiędzy systemami, aplikacjami lub też pomiędzy warstwami w danej aplikacji. *DTO* to bardziej rozbudowany kontener na dane, uzupełniony o dodatkowe możliwości ułatwiające dostęp, ochronę oraz przesyłanie danych.

Zadanie 8.1. Aplikacja bez obiektów *DTO*

Do inicjalizacji projektu o nazwie np. ***pai_dto*** zastosuj *Spring Initializr*, dodając następujące zależności:
- *Spring Web*,
- *Spring Data JPA*,
- *H2 Database*,
- *Lombok*.

Po wygenerowaniu projektu – rozpakuj go, otwórz w swoim IDE i zbuduj. Plik ***application.properties*** może pozostać pusty.

8.1.1. Klasy encji w relacji 1:1

W projekcie utwórz pakiet *domain*, a w nim dwie klasy ***Student*** i ***Address***. Definicja klasy ***Student*** pokazana jest na rysunku 8.1, a klasy ***Address*** na rysunku 8.2 (dodaj odpowiednie importy klas).

Obie klasy są oznaczone adnotacją ***@Entity*** oraz adnotacjami z biblioteki *Lombok*. Dodatkowo klasy te są ze sobą **w relacji 1:1**. Każdy student ma swój własny adres, niezależnie czy istnieje już taki w bazie. Relację 1:1 opisuje adnotacja ***@OneToOne(cascade = CascadeType.ALL)*** z *JPA*. Parametr ***cascade = CascadeType.ALL*** określa, że wszystkie operacje wykonywane na klasie ***Student*** (i tabeli ***student*** w bazie danych) będą automatycznie powtarzane (kaskadowo) na powiązanej z nią klasie ***Address*** (i tabeli ***address*** w bazie danych).

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Student {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY )
    private Long id;
    private String name;
    private String surname;
    private Integer age;

    @JoinColumn(name="address_id")
    @OneToOne(cascade = CascadeType.ALL)
    private Address address;
}
```

Rys. 8.1. Klasa *Student*

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Entity
public class Address {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY )
    private Long id;

    private String street;
    private String city;
    private String state;
    private String zip;

    @OneToOne(mappedBy = "address")
    private Student student;

}
```

Rys. 8.2. Klasa *Address*

8.1.2. Repozytoria i serwis

W głównej paczce aplikacji utwórz pakiet ***repositories***, a w nim dwa interfejsy: ***StudentRepository*** (Rys. 8.3) oraz ***AddressRepository***. Oba rozszerzają interfejs ***JpaRepository*** i są poprzedzone adnotacją ***@Repository***.

```java
package bp.pai_dto.repositories;

import bp.pai_dto.domain.Student;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface StudentRepository extends JpaRepository<Student, Long>{

}
```

Rys. 8.3. Interfejs *StudentRepository*

Repozytorium ***AddressRepository*** jest analogiczne.

Następnie utwórz kolejny pakiet ***services***, a w nim:

- interfejs ***StudentService*** z metodą ***getAllStudents()***, zwracającą listę studentów,
- klasę ***StudentServiceImpl***, implementującą ***StudentService*****.** Klasę ***tudentServiceImpl*** (Rys. 8.4) należy oznaczyć adnotacją ***@Service***, aby *Spring* zarejestrował ją w kontekście. Dodatkowo do tej klasy należy „wstrzyknąć" utworzone wcześniej repozytorium ***StudentRepository***, przy pomocy konstruktora (można skorzystać z biblioteki *Lombok* i adnotacji ***@RequiredArgsConstructor***). Obiekt repozytorium i jego metoda ***findAll()*** zostanie wykorzystana dalej w metodzie zwracającej listę studentów.

```java
@RequiredArgsConstructor
@Service
public class StudentServiceImpl implements StudentService{
    private final StudentRepository studentRepository;
    @Override
    public List<Student> getAllStudents() {
        return studentRepository.findAll();
    }
}
```

Rys. 8.4. Klasa *StudentServiceImpl* implementująca interfejs *StudentService*

8.1.3. Kontroler

Mając serwis, utwórz klasę kontrolera, który będzie z niego korzystał. Klasę ***StudentController*** umieść w pakiecie ***controllers*** i oznacz adnotacją ***@RestController*** oraz adnotacjami z biblioteki *Lombok*, w celu utworzenia konstruktora, który „wstrzyknie" utworzony wcześniej serwis. Do klasy kontrolera dodaj metodę zwracającą wszystkich studentów. Wykorzystaj adnotację ***@RequestMapping***, dzięki której wszystkie akcje kontrolera będą miały ścieżkę domyślnie zaczynającą się od ***/students*** (Rys. 8.5).

```java
@RequiredArgsConstructor
@RequestMapping(value="/students")
@RestController
public class StudentController {
    private final StudentServiceImpl studentService;

    @GetMapping
    public List<Student> getAllStudents(){
        return studentService.getAllStudents();
    }
}
```

Rys. 8.5. Klasa kontrolera *StudentController*

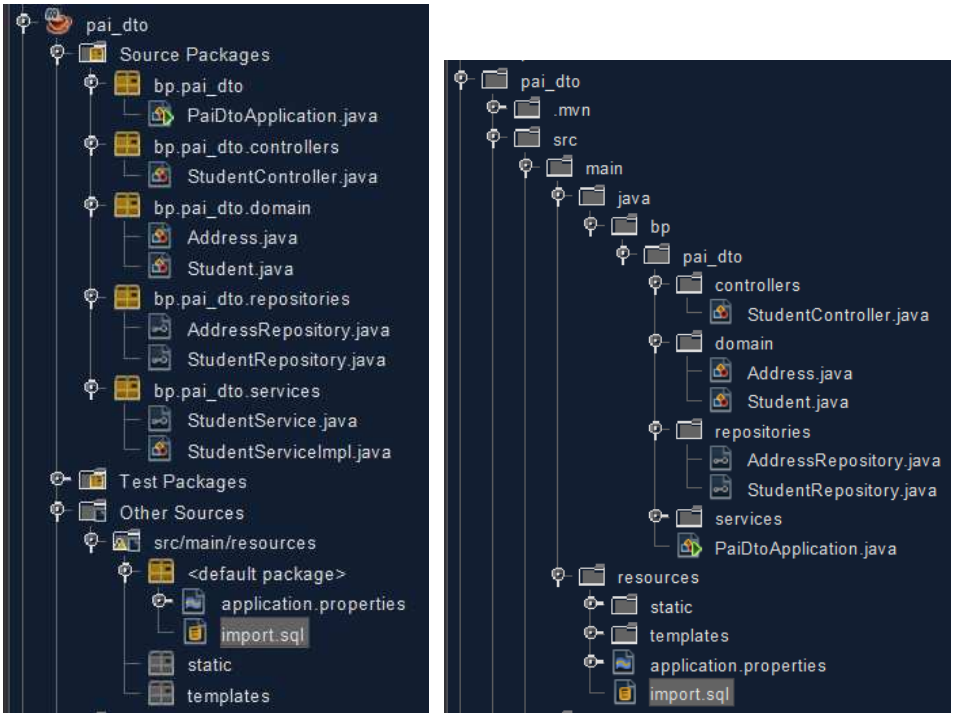
8.1.4. Plik *import.sql*

Przed przetestowaniem, czy wszystko jest prawidłowo zaimplementowane, do folderu ***resources*** dodaj plik ***import.sql*** (Przykład 8.1), który wypełni danymi bazę *H2*. **Właściwa nazwa i lokalizacja pliku jest istotna, aby *Spring* załadował go podczas startu.**

Struktura plików projektu oraz lokalizacja pliku ***import.sql*** przedstawiona jest na rysunku 8.6.

Przykład 8.1. Plik import.sql

```
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'Al.
Szucha Jana Chrystiana 1038', 'Warszawa', 'Polska', '00-582');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Bracka 84', 'Tarnowskie Góry', 'Polska', '42-605');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Ratajczaka Franciszka 90', 'Szczecin', 'Polska', '61-816');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Projektowa 142', 'Katowice', 'Polska', '91-493');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Tymiankowa 57', 'Lublin', 'Polska', '20-221');
INSERT INTO ADDRESS(id, street, city, state, zip) VALUES(null, 'ul.
Zgierska 27', 'Łódź', 'Polska', '91-468');
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Adam', 'Nowak', 18, 1);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Bartosz', 'Pawlak', 20, 2);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Kinga', 'Kowalczyk', 21, 3);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Wiktor', 'Kowalski', 20, 4);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Piotr', 'Szczepański', 22, 5);
INSERT INTO STUDENT(id, name, surname, age, address_id) VALUES (null,
'Filip', 'Chmielewski', 19, 6);
```



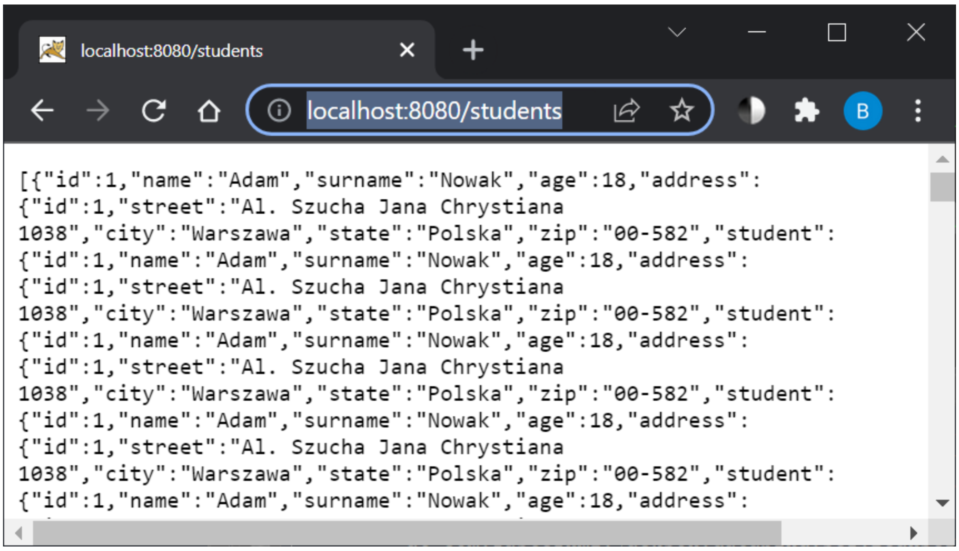
Rys. 8.6. Widok projektu w zakadce *Project* i *Files* w *Netbeans*

Zbuduj projekt i uruchom aplikację. Przy pomocy narzędzia *Postman* lub przeglądarki pobierz listę studentów, podając adres: ***localhost:8080/students*** (Rys. 8.7).

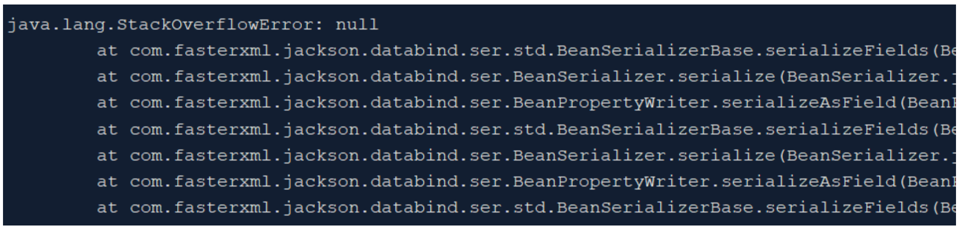
Rezultat powinien być taki, jak na rysunku 8.7, a dodatkowo w konsoli aplikacji pojawi się komunikat o błędzie (Rys. 8.8).

Na pierwszy rzut oka wszystko wyglądało dobrze, a jednak pojawił się problem, ponieważ doszło do zapętlenia się programu. Obiekt klasy ***Student*** posiada referencję do obiektu klasy ***Address***, który z kolei posiada referencję do obiektu ***Student.*** Z tego powodu obie te instancje w kółko siebie wywołują, aż do całkowitego zapełnienia stosu.

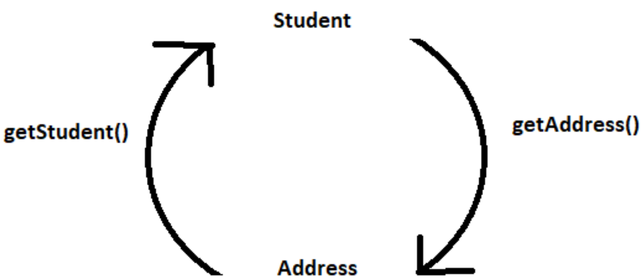
Graficznie można to zinterpretować rysunkiem 8.9.



Rys. 8.7. Widok w przeglądarce listy studentów z bazy *H2*



Rys. 8.8. Widok komunikatu o błędzie na konsoli po uruchomieniu aplikacji



Rys. 8.9. Problem z referencjami w obiektach *Student* i *Address*

Zadanie 8.2. Aplikacja z *DTO*

Problem z zadania 8.1, jak i wiele podobnych, można rozwiązać na kilka sposobów. W tym zadaniu będą zastosowane najbardziej elastyczne i najczęściej używane obiekty *DTO* (ang. *Data Transfer Object)*. Jak sama nazwa wskazuje, obiekty *DTO* są wykorzystywane przy transferze danych pomiędzy klientem a serwerem. Niezależnie od metody transferu (*GET*, *POST* czy *PATCH*), zasada działania jest taka sama. Tworzone są klasy zawierające tylko te pola, które mają być pokazane przez *API*. Do takiej klasy *DTO* potrzebny jest konwerter, przypisujący polom wartości z odpowiednich encji. Utworzona w zadaniu 8.1 aplikacja zostanie teraz zmodyfikowana, przez wprowadzenie obiektów *DTO* i odpowiednich do nich konwerterów.

8.2.1. Klasa *DTO*

Aby uzyskać listę studentów wraz z ich adresami (tak jak w poprzednim przypadku), w nowym pakiecie o nazwie ***dtos*** utwórz klasę ***StudentDto***. Klasa ta powinna zawierać pola zarówno z klasy ***Student***, jak i ***Address*** (Rys. 8.10). Klasa poprzedzona jest znanymi adnotacjami z biblioteki *Lombok*. Na szczególną uwagę zasługuje adnotacja ***@Builder***, pochodząca z tej samej biblioteki. W celu implementacji wzorca projektowego *builder* wystarczy adnotować klasę jako ***@Builder***, a kompilator wygeneruje w klasie głównej statyczną, zagnieżdżoną klasę ***builder*** oraz niezbędny konstruktor. Wzorzec projektowy *builder* to prawdopodobnie najczęściej stosowany wzorzec projektowy w *Javie.* Umożliwia on wieloetapowe budowanie obiektu, które wykonywane jest przez specjalnie przygotowany do tego zadania obiekt budujący (ang. *builder*).

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class StudentDto {
    private String name;
    private String surname;
    private Integer age;
    private String street;
    private String city;
    private String zip;
    private String state;
}
```

Rys. 8.10. Klasa *StudentDto*

8.2.2. Klasa konwertera

Dla klasy *DTO* potrzebny jest mechanizm, który zamieni obiekty klas ***Student*** i ***Address*** na obiekt ***StudentDto***. W pakiecie o nazwie ***converters***, utwórz klasę ***StudentConverter***, dziedziczącą po istniejącej klasie ***Converter*** dostępnej w odpowiednim pakiecie *Spring* (Rys 8.11).

```
import bp.pai_dto.domain.Student;
import bp.pai_dto.dtos.StudentDto;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

@Component
public class StudentConverter implements Converter<Student, StudentDto>{

    @Override
    public StudentDto convert(Student source) {
        return null;
    }
}
```

Rys. 8.11. Klasa *StudentConverter*

Klasa ***StudentConverter***:

- oznaczona jest adnotacją ***@Component***, aby była dostępna w kontekście aplikacji *Spring* i można było ją „wstrzykiwać" do innych klas komponentów.
- implementuje interfejs ***Converter<S, T>***, gdzie ***S*** (ang. *source*) jest klasą źródłową (z której dane są pobierane), a ***T*** (ang. *target*) jest klasą docelową *DTO* (udostępnianą z naszego *API*).
- nadpisuje metodę ***convert()*** interfejsu ***Converter***, w której tworzona jest logika konwersji (Rys. 8.12). Wybrano tu odpowiednie wartości pól z instancji klasy ***Student*** i przypisano je obiektowi klasy ***StudentDto*** za pomocą metody ***builder()***, dzięki której kod jest znacznie czytelniejszy.

```
@Override
public StudentDto convert(Student source) {
    return StudentDto.builder()
            .name(source.getName())
            .surname(source.getSurname())
            .age(source.getAge())
            .street(source.getAddress().getStreet())
            .city(source.getAddress().getCity())
            .zip(source.getAddress().getZip())
            .state(source.getAddress().getState())
            .build();
}
```

Rys. 8.12. Implementacja metody *convert()* w klasie *SudentConverter*

8.2.3. Wykorzystanie *DTO* w serwisie i kontrolerze

W projekcie wykorzystaj utworzoną klasę i konwerter. W tym celu:

1. W klasach ***StudentService***, ***StudentServiceImpl*** i w kontrolerze ***StudentController*** zmień zwracany typ ze ***Student*** na ***StudentDto***.
2. Do klasy ***StudentServiceImpl*** „wstrzyknij" komponent konwertera ***StudentConverter*** oraz w metodzie ***getAllStudents()*** zastosuj strumień ***stream()*** i metodę ***map()*** do zmapowania obiektów ***Student*** na ***StudentDto***.

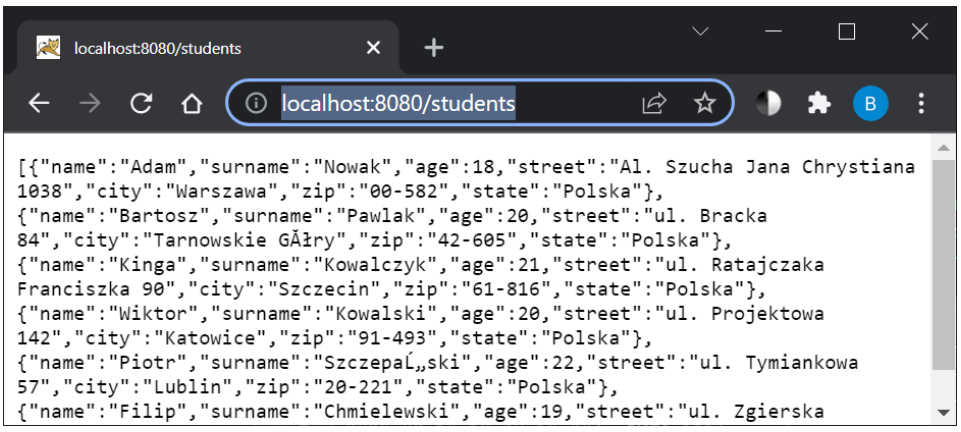
Zmodyfikowany serwis powinien wyglądać jak na rysunku 8.13.

```
@RequiredArgsConstructor
@Service
public class StudentServiceImpl implements StudentService{
    private final StudentRepository studentRepository;
    private final StudentConverter studentConverter;

    @Override
    public List<StudentDto> getAllStudents() {
        return (List<StudentDto>) studentRepository.findAll().stream()
                .map(studentConverter::convert)
                .collect(Collectors.toList());

    }
}
```

Rys. 8.13. Klasa *StudentServiceImpl* po modyfikacjach

8.2.4. Test działania obiektu *DTO*

Ponownie pobierz listę studentów w przeglądarce (Rys. 8.14) lub w *Postman*.



Rys. 8.14. Efekt działania aplikacji z obiektami *DTO* w przeglądarce

Tym razem uzyskano poprawne dane, bez wyrzucenia błędów na konsoli serwera.

Zadanie 8.3. *DTO* w repozytoriach

Przykład z poprzedniego zadania demonstruje jedynie podstawowe możliwości *DTO*. *Spring* zapewnia dużo większe wsparcie dla tego rodzaju mechanizmów, np. związane z ograniczeniem liczby pobieranych bajtów.

Dodaj dodatkowe pole ***attachment*** typu ***byte[]*** do klasy ***Student*** z dodatkową adnotacją ***@Lob***. Z założenia, pole to będzie przechowywało jakiś załącznik (np. dyplom, świadectwo) w postaci pliku ***pdf*** lub ***png***. Pola tego typu zajmują wiele bajtów, ale dla ujednolicenia, każdy taki załącznikw przykładzie niech zajmuje 1MB. Przy 5 studentach pobieranie tego pola z bazy nie stanowi jeszcze problemu. 5MB jest do przyjęcia dla małych aplikacji, ale przy 100 studentach jest to już 100MB, co może być problematyczne. Jeśli pól tego typu w tabeli byłoby kilka, mogą wystąpić poważne problemy z szybkością działania aplikacji. Korzystając z domyślnych metod repozytorium dostarczanych przez *Spring Data*, pobieranych będzie dodatkowe 100MB za każdym razem, niezależnie od tego, że dodatkowe załączniki nie są potrzebne (a i tak będą pobierane z bazy).

Spring pozwala ograniczyć liczbę danych i kontrolować, jakie dane należy pobrać z bazy i można to zrealizować przy użyciu utworzonej już klasy *DTO*.

W interfejsie dziedziczącym po repozytorium *Spring Data JPA* można dodawać własne metody poprzedzone adnotacją ***@Query***, które pozwalają zbudować własne zapytania do bazy za pomocą języka *JPQL* (ang. *Java Persistance Query Language)*. *JPQL* przypomina składnię *SQL*, ale pracuje z obiektami klas w aplikacji *Java*. Przykłady zastosowania zapytań wbudowanych lub metod z adnotacją ***@Query*** realizowane były w zadaniach z laboratorium 5. Na przykład metoda:

```
@Query("select s from Student s")
List<Student> findAllStudents()
```

pozwala pobrać wszystkich studentów z bazy, dzięki zastosowaniu składni języka *JPQL*.

Obiekt *DTO* (ograniczający liczbę pobieranych danych z bazy) w zapytaniu *JPQL* można wykorzystać w klasie ***StudentRepository*** w sposób pokazany na rysunku 8.15.

```
@Repository
public interface StudentRepository extends JpaRepository<Student, Long>{
    @Query("select new bp.pai_dto.dtos.StudentDto(s.name, s.surname, s.age, "+
    "s.address.street,s.address.city, s.address.state, s.address.zip) from Student s")
    List<StudentDto> findAllNoAttachment();
}
```

Rys. 8.15. *@Query* w klasie *StudentRepository*

Sztuczka w zapytaniu polega na użyciu konstruktora ***StudentDto***, generowanego dzięki adnotacji ***@AllArgsConstructor*** przez bibliotekę *Lombok* (Przykład 8.2).

Przykład 8.2. Konstruktor z parametrami generowany przez Lombok

```
public StudentDto(final String name, final String surname,
                  final Integer age, final String street,
                  final String city, final String state,
                  final String zip){
       this.name=name;
       this.surname=surname;
       this.age=age;
       //pozostałe przypisania ...
}
```

Wewnątrz zapytania ***@Query*** podano **całą ścieżkę do pakietu z klasą *DTO***. Do pól odwołano się poprzez ***alias.pole***, a ich typy muszą pasować do typów parametrów konstruktora. Dla jednego *DTO* można tworzyć wiele konstruktorów i odpowiednio ich używać dla każdej metody ***@Query***. Ważne jest, że w tym przypadku **nie stosuje się już konwertera**, ponieważ zwracane są gotowe obiekty klasy ***StudentDto*** (metoda ***getAllStudentsNoAttachment()*** w klasie ***StudentService*** i ***StudentServiceImpl*** – Rys. 8.16), więc **typ zwracany w repozytorium musi być zmieniony na klasę *DTO***. Wywołując tak zdefiniowaną metodę w kontrolerze, uzyskuje się takie same dane, lecz tutaj nie są pobierane wszystkie dane z bazy, a tylko potrzebne wartości. Dzięki takiemu podejściu można znacznie zoptymalizować i przyspieszyć działanie aplikacji.

```
@Override
public List<StudentDto> getAllStudentsNoAttachment() {
    return studentRepository.findAllNoAttachment();
}
```

Rys. 8.16. Metoda *getAllStudentsNoAttachment()* w klasie *StudentServiceImpl*

Zadanie 8.4. Biblioteka *MapStruct*

Biblioteka ***MapStruct*** jest darmową biblioteką znacznie usprawniającą mapowanie encji na klasy *DTO*. Niestety nie da się jej wskazać podczas tworzenia projektu w *Spring Initializr* i trzeba ręcznie dodać zależność do pliku ***pom.xml:***

```
<dependency>
       <groupId>org.mapstruct</groupId>
       <artifactId>mapstruct</artifactId>
       <version>1.4.2.Final</version>
</dependency>
```

Dodatkowo w sekcji `<plugins>` w elemencie `<build>` należy dodać kolejny `<plugin>` z przykładu 8.3.

Przykład 8.3. Dodatkowy plugin do pliku pom.xml

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>3.8.1</version>
    <configuration>
        <annotationProcessorPaths>
            <path>
                <groupId>org.mapstruct</groupId>
                <artifactId>mapstruct-processor</artifactId>
                <version>1.4.2.Final</version>
            </path>
            <path>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok</artifactId>
                <version>1.18.20</version>
            </path>
            <path>
                <groupId>org.projectlombok</groupId>
                <artifactId>lombok-mapstruct-binding</artifactId>
                <version>0.2.0</version>
            </path>
        </annotationProcessorPaths>
        <compilerArgs>
            <compilerArg>-Amapstruct.defaultComponentModel=spring
        </compilerArg>
        </compilerArgs>
    </configuration>
</plugin>
```

Po przebudowaniu projektu aplikacja jest gotowa do korzystania z biblioteki ***MapStruct***.

Aby przetestować działanie biblioteki, w istniejącym pakiecie ***converters*** utwórz interfejs o nazwie ***StudentMapper*** **z** metodą ***mapStudentToDtoStudent()***, jak na rysunku 8.17.

```java
package bp.pai_dto.converters;

import bp.pai_dto.domain.Student;
import bp.pai_dto.dtos.StudentDto;
import org.mapstruct.Mapper;
import org.mapstruct.Mapping;

@Mapper
public interface StudentMapper {
    @Mapping(target="name", source="student.name")
    @Mapping(target="surname", source="student.surname")
    @Mapping(target="age", source="student.age")
    @Mapping(target="street", source="student.address.street")
    @Mapping(target="city", source="student.address.city")
    @Mapping(target="state", source="student.address.state")
    @Mapping(target="zip", source="student.address.zip")
    StudentDto mapStudentToStudentDto(Student student);
}
```

Rys. 8.17. Klasa *StudentMapper* w pakiecie *converters*

Adnotacja:

- ***@Mapper*** oznacza interfejs jako źródło, na podstawie którego ***MapStruct*** ma utworzyć implementację dla nowego konwertera.
- ***@Mapping*** pozwala wskazać ***target*** (nazwę pola w klasie docelowej) oraz ***source*** (nazwę pola w klasie źródłowej).

Sama nazwa metody nie ma znaczenia, **musi się jedynie zgadzać zwracany typ** (***StudentDto***) oraz **typ parametru metody** (***Student***). Znaczenie ma natomiast **nazwa argumentu** w parametrze ***source*** (***source = "nazwaArgumentu.pole"***). Jedną z ważniejszych zalet tej biblioteki jest to, że jeżeli pola nie są zagnieżdżone oraz nazwa i typ pola z *DTO* zgadza się z nazwą i typem pola encji, to nie trzeba definiować dodatkowego mapowania. ***MapStruct*** sam zorientuje się, co ma wziąć i do czego przypisać.

Interfejs z rysunku 8.17 można zatem uprościć do postaci jak na rysunku 8.18.

```java
@Mapper
public interface StudentMapper {
    @Mapping(target="street", source="student.address.street")
    @Mapping(target="city", source="student.address.city")
    @Mapping(target="state", source="student.address.state")
    @Mapping(target="zip", source="student.address.zip")
    StudentDto mapStudentToStudentDto(Student student);
}
```

Rys. 8.18. Interfejs *StudentMapper* po uproszczeniu

Po tej modyfikacji można „wstrzyknąć" zdefiniowany maper do serwisu studenta (***StudentServiceImpl***) i wykorzystać go zamiast starego konwertera (Rys. 8.19).

```
final private StudentMapper studentMapper;
@Override
public List<StudentDto> getAllStudents() {
return studentRepository.findAll().stream()
            .map(studentMapper::mapStudentToStudentDto)
            .collect(Collectors.toList());
}
```

Rys. 8.19. Metoda *getAllStudents* z maperem

Testując działanie aplikacji po raz kolejny, wynik powinien być taki sam jak w przypadku poprzedniego konwertera, ale tym razem zautomatyzowano proces przekształcania encji w obiekt *DTO* (oszczędność czasu).
Biblioteka *MapStruct* ma jeszcze wiele innych możliwości, z których korzysta się w bardziej złożonych aplikacjach.

Więcej na temat tej biblioteki można znaleźć na stronach:

- dokumentacja *MapStruct*: MapStruct 1.5.0.Beta2 Reference Guide
- krótki poradnik z podstawami biblioteki: https://www.baeldung.com/mapstruct

Laboratorium opracowane zostało we współpracy ze studentami Informatyki: Kamilem Jaskotem i Sebastianem Iwanowskim.