Adam Kościński

Bazy danych II

Implementacja operacji CRUD na bazie danych Northwind.

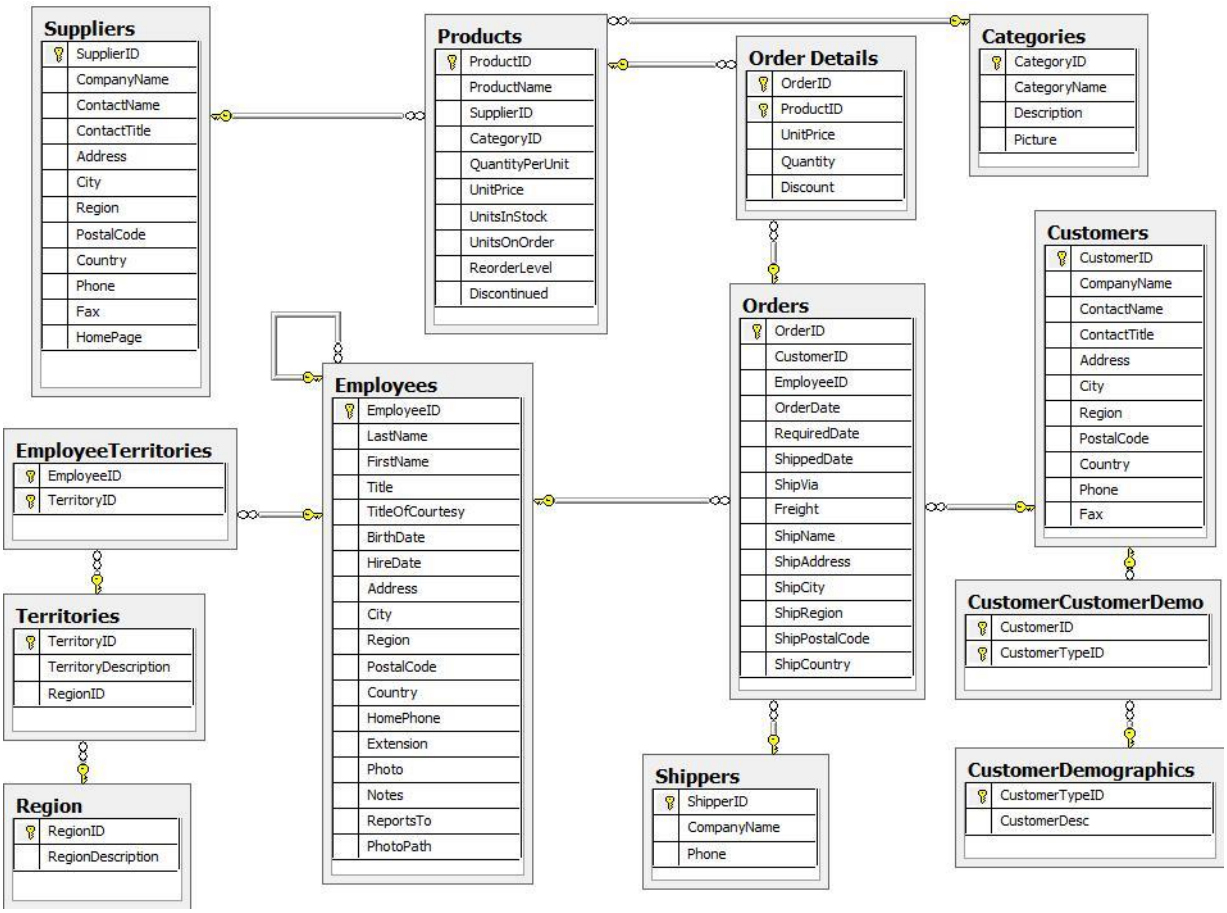
Dokumentacja tworzenia środowiska pracy, instalacja serwera bazodanowego, tworzenie połącznia między aplikacją a bazą danych, implementacja modelu oraz podstawowych kontrolerów. Testy modelu.

Użyte technologie:

- System operacyjny: Ubuntu 17.04
- Serwer bazy danych: PostgreSQL
- Język programowania: Java 8
- Implementacja JPA: Hibernate 5.2

W niniejszej dokumentacji pominięta została instalacja systemu operacyjnego.

Schemat bazy Northwind:



1. Instalacja serwera PostgreSQL, dodanie bazy danych.

Jako, że domyslne repozytoria Ubuntu zawierają paczki PostreSQL, instalacja sprowadza się do wykonania dwóch komend:

```
bazydanych@bazydanych-VirtualBox:~$
bazydanych@bazydanych-VirtualBox:~$ sudo apt-get update
```

Następnie:

```
bazydanych@bazydanych-VirtualBox:~$
bazydanych@bazydanych-VirtualBox:~$ sudo apt-get install postgresql postgresql-contrib
```

System poprosi nas o hasło roota, a następnie zainstaluje i uruchomi serwer PostrgreSQL.

Kolejnym krokiem jest dodanie użytkownika mogącego korzystać z serwera.

```
bazydanych@bazydanych-VirtualBox:~$
bazydanych@bazydanych-VirtualBox:~$ sudo -u postgres createuser --interactive
[sudo] password for bazydanych:
Enter name of role to add: bazydanych
```

Dodajemy użytkownika, ustawiamy dla niego hasło, potwierdzamy dodanie go jako superusera.

Teraz możemy stworzyć bazę danych northwind poprzez komendę:

```
bazydanych@bazydanych-VirtualBox:~$
bazydanych@bazydanych-VirtualBox:~$ sudo -u postgres createdb northwind
```

Możemy się teraz dostać do niej przez komendę:

```
bazydanych@bazydanych-VirtualBox:~$
bazydanych@bazydanych-VirtualBox:~$ psql northwind
psql (9.6.6)
Type "help" for help.

northwind=>
```

Po stworzeniu bazy danych ściągamy odpowiednio przygotowany plik .sql z którego odtworzymy bazę Northwind na naszym serwerze PostgreSQL. Dostępny jest on tutaj.

Ściągamy plik na nasz komputer, z folderu do którego go pobraliśmy wykonujemy polecenie:

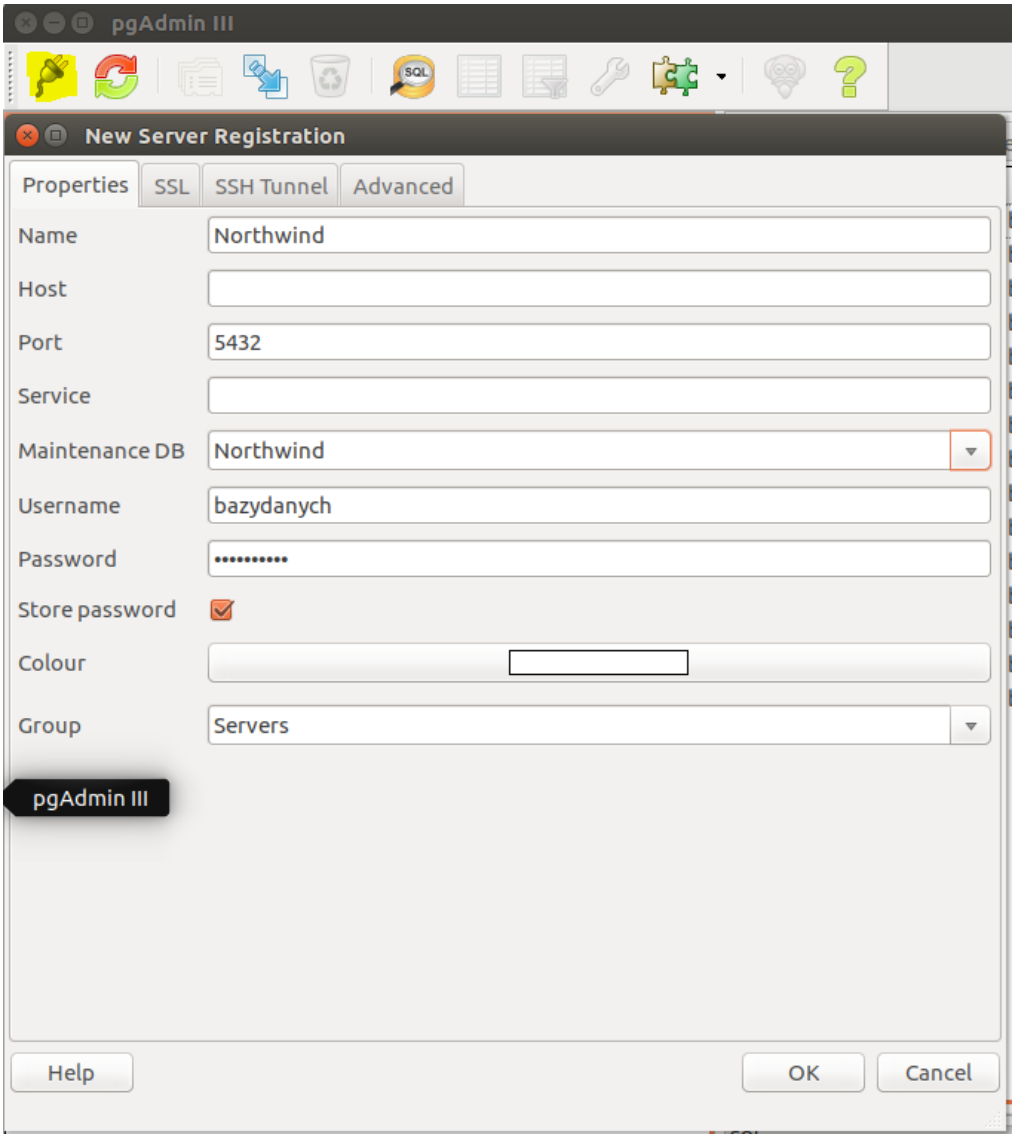
```
-rw-r--r--  1 bazydanych bazydanych 349545 paź 20 15:27 northwind.sql
bazydanych@bazydanych-VirtualBox:~/Downloads$ psql northwind < northwind.sql
```

Po chwili w bazie danych powstaje schemat i załadowane są dane z pliku northwind.sql.

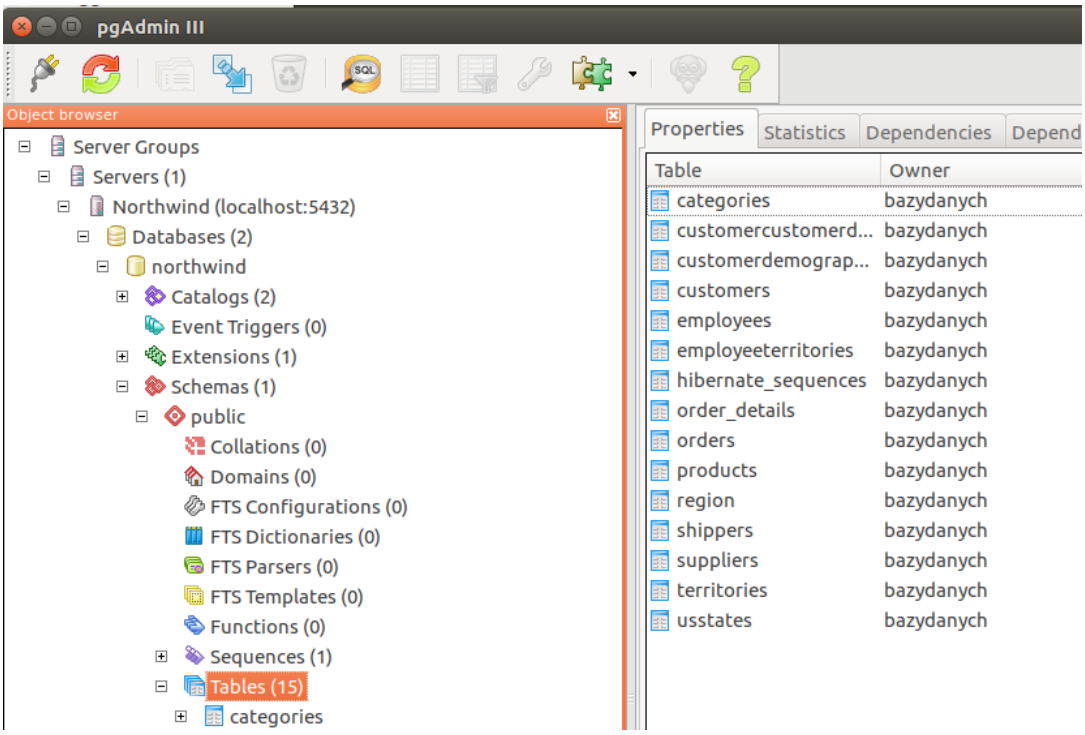
Aby ułatwić obsługę bazy, możemy zainstalować nakładkę graficzną pgAdmin III:

```
bazydanych@bazydanych-VirtualBox:~$
bazydanych@bazydanych-VirtualBox:~$ sudo apt-get install pgadmin3
```

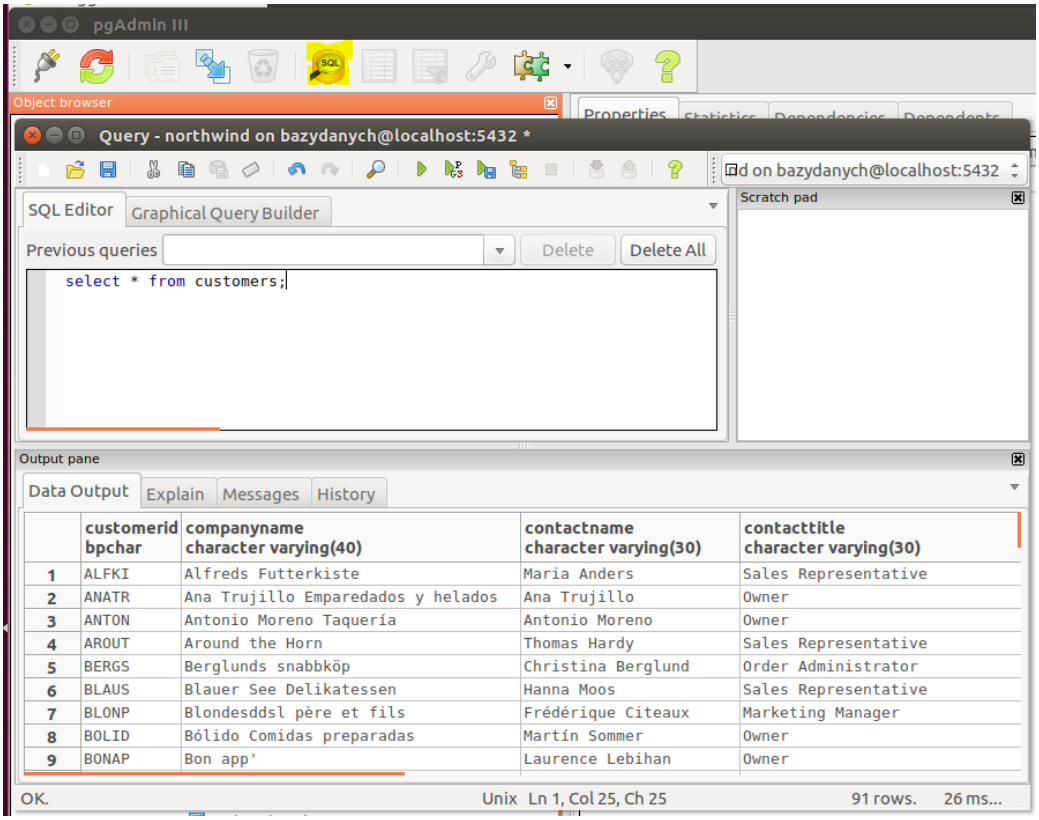
Możemy teraz uruchomić go i dodać odpowiednie połączenie:



Teraz jesteśmy w stanie sprawdzić, co znajduje się w naszej bazie danych northwind. Jak widać na poniższym obrazku, tabele zostały stworzone odpowiednio.



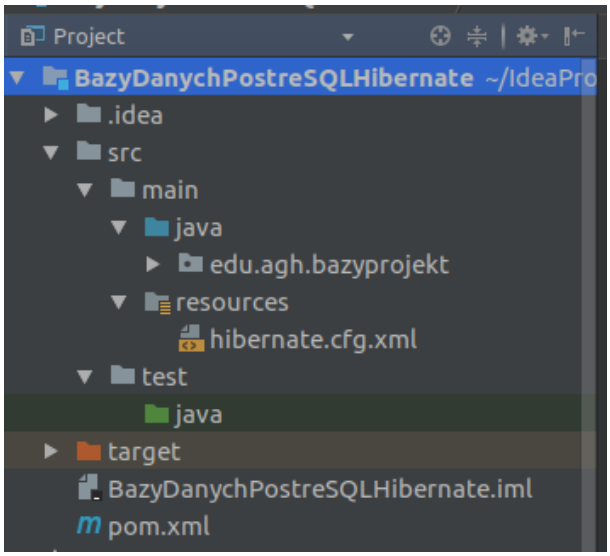
Sprawdźmy, czy baza została uzupełniona o dane poprzez wykonanie prostego zapytania na tabeli customers.



W tych prostych krokach stworzyliśmy serwer, użytkownika, samą bazę danych, a także uzupełniliśmy ją o schemat i dane. W tym momencie możemy przejść do stworzenia naszego projektu Javie oraz Hibernate.

2. Stworzenie projektu, konfiguracja Hibernate.

Używając IDE IntelliJ Idea tworzymy nowy projekt, następnie w taki sposób konfigurujemy jego strukturę:



W tym punkcie zajmę się konfiguracją połączenia z bazą danych northwind w pliku hibernate.cfg.xml. Tworzenie modelu i dodawanie go do tego pliku znajduje się w następnych punktach tej dokumentacji.

Tak wygląda plik hibernate.cfg.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.username">bazydanych</property>
        <property name="connection.password">bazydanych</property>
        <property name="connection.driver_class">org.postgresql.Driver</property>
        <property name="connection.url">jdbc:postgresql://localhost:5432/northwind</property>
        <property name="dialect">org.hibernate.dialect.PostgreSQL9Dialect</property>

        <property name="show_sql">true</property>

        <property name="format_sql">true</property>
        <property name="hbm2ddl.auto">update</property>


    </session-factory>
</hibernate-configuration>
```

Omówię teraz konfigurację która została wprowadzona do tego pliku, aby możliwe było połączenie się z bazą danych northwind:

```
<property name="connection.username">bazydanych</property>
<property name="connection.password">bazydanych</property>
```

Pola w których uzupełniamy nazwę użytkownika oraz jego hasło do połączenia się z bazą danych. Używamy poprzednio stworzonego użytkownika bazydanch z takim samym hasłem.

```
<property name="connection.driver_class">org.postgresql.Driver</property>
```

Rodzaj sterownika JDBC który zostanie użyty do połączenia z bazą danych. Używamy odpowiedniego dla serwera PostgreSQL. Dodatkowo dodajemy go w dependencjach w pliku pom.xml:

```
        <dependency>
            <groupId>org.postgresql</groupId>
            <artifactId>postgresql</artifactId>
            <version>42.1.4</version>
        </dependency>
```

```
<property name="connection.url">jdbc:postgresql://localhost:5432/northwind</property>
```

URL naszej bazy danych. W tym wypdaku znajduje się ona na tym samym serwerze co sam program, więc używamy localhost, w innym wypadku może znaleźć się tutaj IP serwera bazodanowego. Używamy tego wpisu następująco:

jdbc:nazwaSterownika:url:port/nazwaBazyDanych

```
<property name="dialect">org.hibernate.dialect.PostgreSQL9Dialect</property>
```

Konfiguracja dialektu SQL dla naszej bazy danych. Jako że wersje języka SQL używane dla poszczególnych silników bazodanowych nie są ze sobą kompatybilne, informujemy hibernate jakiego dialektu powinno użyć w komunikacji z bazą danych. W naszym wypadku jest to PostgreSQL9Dialect

```
<property name="show_sql">true</property>
```

Tutaj ustawiamy, że na standardowym wyjściu pojawią się wygenerowane przez hibernate SQLe. Przydatne jest to w wypadku środowisk developerskich lub na produkcji w wypadku problemów z bazą danych. Pozwala to przeanalizować wysyłane do bazy zapytania. Ustawienie na false powoduje ukrycie generowanych SQLi.

```
<property name="format_sql">true</property>
```

Formatuje SQLe aby ich czytanie było łatwiejsze dla ludzi.

```
<property name="hbm2ddl.auto">update</property>
```

Ustawienie co serwer wykona na schemacie bazy danych przy tworzeniu SessionFactory: Dostępne opcje:

- validate: skontroluj schemat bazy, nie wprowadzaj zmian.
- update: aktualizuj schemat bazy danych w wypadku zmiany modelu w kodzie.
- create: stwórz schemat na podstawie modelu z kodu, zniszcz poprzedni razem z danymi
- create-drop: zniszcz schemat gdy SessionFactory jest zamykane, zazwyczaj w momencie zatrzymania aplikacji.

Dla naszego serwera ustawiamy update, jako że baza jest już stworzona, ale chcemy mieć możliwość aktualizacji jej schematu w wypadku zmiany w kodzie, nie niszcząc jednach wszystkich danych w niej zawartych. Dla produkcyjnych aplikacji możemy ustawić validate, poza ich aktualizacją wymagającą zmiany schematu, wtedy ustawiamy update.

```
<property name="connection.pool_size">1</property>
```

Ogranicza ilość połączeń do bazy danych. Dla nas wystarczającą wartością jest 1.

Aby przetestować działanie aplikacji, możemy wystartować ją tworząc nowe SessionFactory w następujący sposób:

```
package edu.agh.bazyprojekt.hibernateUtils;

import ...

@Configuration
public class HibernateSessionFactory {

    @Bean
    public static SessionFactory getSessionFactory() {
        ServiceRegistry serviceRegistry = new StandardServiceRegistryBuilder().configure().build();
        return new MetadataSources(serviceRegistry).buildMetadata().buildSessionFactory();
    }

}
```

W HibernateSessionFactory tworzymy naszą SessionFactory które następnie zostanie użyte do otwarcia sesji z bazą danych na podstawie konfiguracji zawartej w pliku hibernate.cfg.xml

Po uruchomieniu następującej metody main używającej powyższego kodu do nawiązania sesji:

```
public class Main {

    public static void main(String[] args) {

        HibernateSessionFactory.getSessionFactory();
    }
```

Na standardowe wyjście wypisane zostanie potwierdzienie pozytywnego utworzenia sesji potwierdzające poprawną konfigurację w pliku hibernate.cfg.xml.

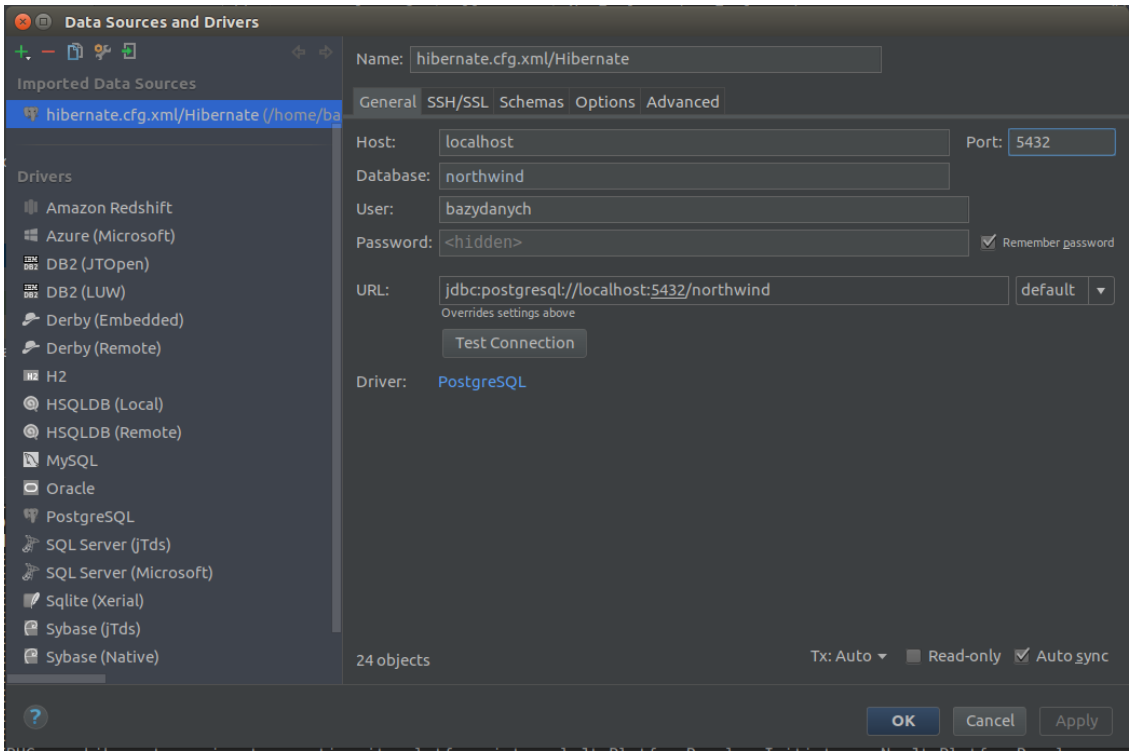
```
17:08:11.480 [main] DEBUG org.hibernate.internal.SessionFactoryRegistry - Initializing SessionFactoryRegistry : org.hibernate.internal.SessionFactoryRegistry@2e2ff723
17:08:11.482 [main] DEBUG org.hibernate.internal.SessionFactoryRegistry - Registering SessionFactory: f0d61955-f6fb-4b33-8503-b44c530cdec8 (<unnamed>)
```

Połączenie z bazą danych jest już skonfigurowane, możemy przejść do stworzenia klas reprezentujących schemat bazy danych oraz jej relacji.

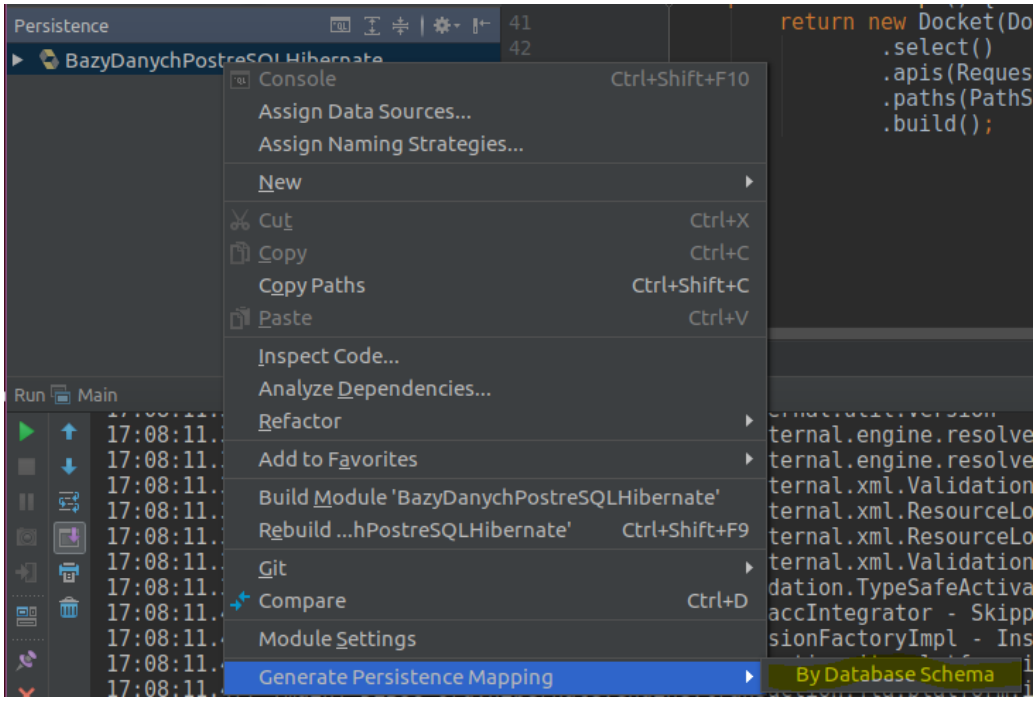
3. Automatczny import modelu przy użyciu IDE IntelliJ IDEA.

IntelliJ IDEA pozwala nam w łatwy sposób stworzyć klasy modelu na podstawie schematu bazy danych. Robimy to w następujący sposób:

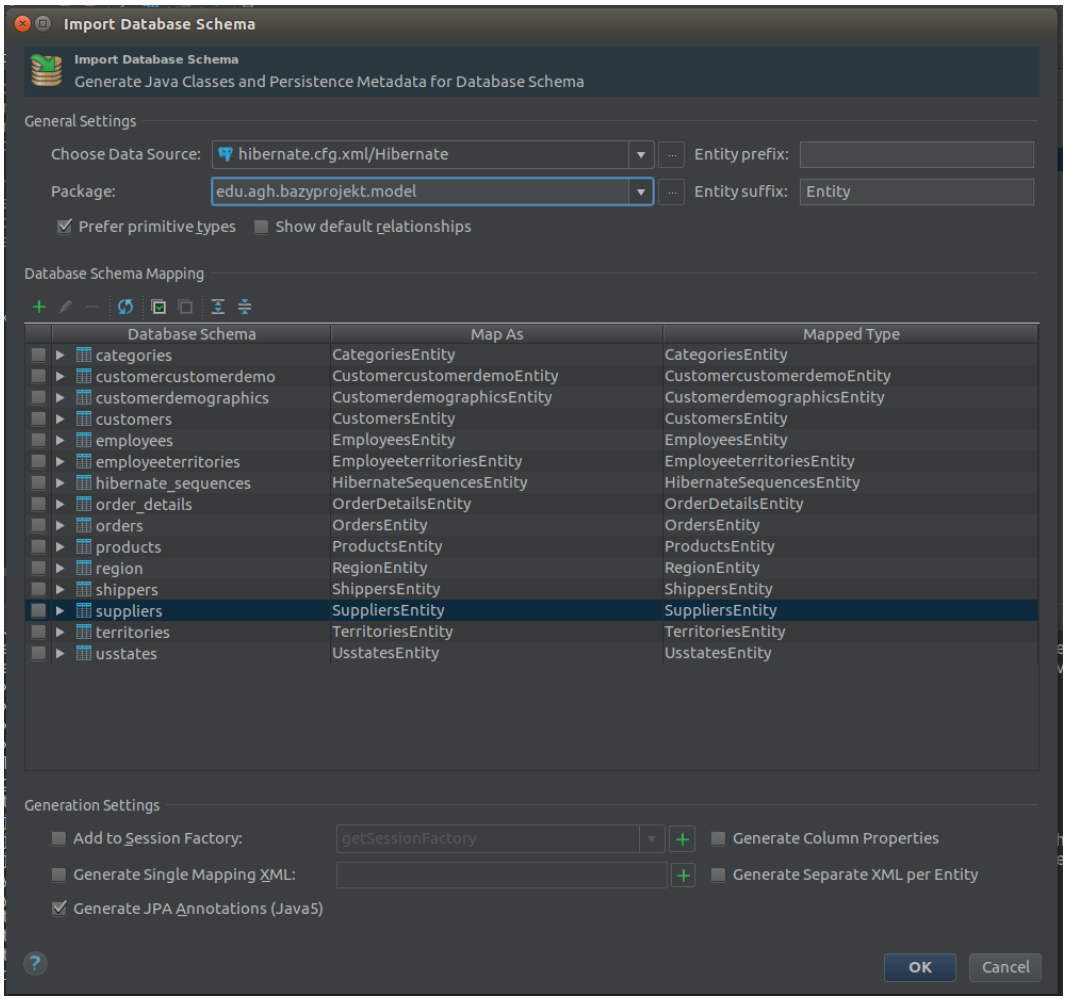
W menu Data Sources and Drivers dodajemy naszą bazę danych na podstawie pliku hibernate.cfg.xml:



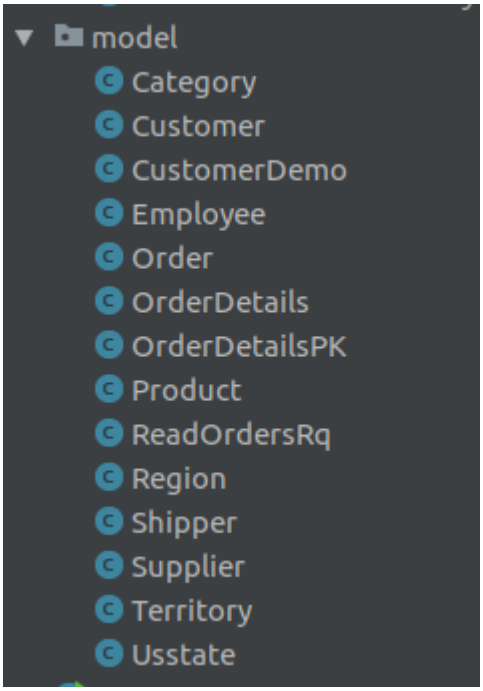
Teraz w view Persistence możemy wygenerować nasz model:



W następnym oknie wybieramy odpowiednie, dodane wcześniej przez nas Data Source, pakiet w którym zamierzamy umieścić klasy modelu i generuje nam się następujący widok:



Wygenerowały nam się następujące klasy(oprócz ReadOrdersRq, zostanie ona omówiona w innej części dokumentacji):



Wygenerowane klasy wymagają jeszcze manualnej konfiguracji, przejdę teraz po nich, opisując ich konfigurację, użyte annotacje, zależności oraz ewentualne kaskady.

4. Konfiguracja modelu.

Opiszę teraz poszczególne klasy modelu, zwracając uwagę na wystepujące w nich nowe szczegóły implementacji.
Zaczynając od góry alfabetycznie:

Category.java:

```java
@Entity
@Table(name = "categories", schema = "public", catalog = "northwind")
public class Category {
    private short categoryID;
    private String categoryName;
    private String description;
    private byte[] picture;
    private Collection<Product> productsInCategory;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "categoryid")
    public short getCategoryID() { return categoryID; }

    public void setCategoryID(short categoryid) { this.categoryID = categoryid; }

    @Basic
    @Column(name = "categoryname")
    public String getCategoryName() { return categoryName; }

    public void setCategoryName(String categoryname) { this.categoryName = categoryname; }

    @Basic
    @Column(name = "description")
    public String getDescription() { return description; }

    public void setDescription(String description) { this.description = description; }

    @Basic
    @Column(name = "picture")
    public byte[] getPicture() { return picture; }

    public void setPicture(byte[] picture) { this.picture = picture; }

    @OneToMany(mappedBy = "category", cascade = CascadeType.ALL, orphanRemoval = false)
    public Collection<Product> getProductsInCategory() { return productsInCategory; }

    public void setProductsInCategory(Collection<Product> productsByCategoryid) {
        this.productsInCategory = productsByCategoryid;
    }
```

Jako że jest to pierwsza klasa którą omawiamy, przedyskutujemy tu wszystkie występujące annotacje oraz ich konfigurację.

`@Entity`

Oznaczenie które informuje Hibernate o tym, że klasa jest kontenerem na dane z odpowiedniej tabeli w bazie danych.

```
@Table(name = "categories", schema = "public", catalog = "northwind")
```

Informuje hibernate o nazwie tabeli oraz konkretnym schemacie w bazie danych skąd tabela ta pochodzi.

```
private short categoryID;
private String categoryName;
private String description;
private byte[] picture;
```

Pola które zostały zmapowane z kolumn tabeli. Ich konfiguracja znajduje się przy getterach, ale równie dobrze mogłaby znaleźć się nad polami. Jednak przez wzgląd na łatwość czytania kodu, możliwość szybkiego sprawdzenia jakie pola znajdują się w obiekcie podjąłem konwencję annotacji nad getterami. Nie będę opisywał ich wszystkich, skupiając się nad znaczeniem wcześniej nie opisanych annotacji lub ciekawych miejscach, gdyż w wielu przypadkach ich znaczenie jest dokładnie te same przy zmienie jedynie mapowanej kolumny.

```
private Collection<Product> productsInCategory;
```

Kolekcja zawierająca Produkty, będące w relacji wiele do jednego z Kategoriami. Niżej znajdzie się opis odpowiedniej konfiguracji nad getterem.

```
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
@Column(name = "categoryid")
public short getCategoryID() { return categoryID; }
```

Annotacja @Id informuje Hibernate że pole to jest Id dla danej tabeli będąc jej PrimaryKey.

@GeneratedValue informuje, że ta wartość jest generowana automatycznie.

JPA definiuje następjące strategie generowania ID:

- AUTO – wybierany automatycznie przez Hibernate
- TABLE – tabela w bazie danej służąca do zarządzania ID
- IDENTITY – odpowiedzialność za generowanie kluczy zostaje powierzona bazie danych
- SEQUENCE – hibernate zajmuje się tworzeniem ID dla nowych encji,

W naszej aplikacji używamy GenerationType.IDENTITY, powierzając generowanie ID dla nowych encji bazie danych.

@Column mapuje pole w klasie do odpowiedniej kolumny o nazwie podanej w „name”.

```java
@Basic
@Column(name = "categoryname")
public String getCategoryName() { return categoryName; }

public void setCategoryName(String categoryname) { this.categoryName = categoryname; }

@Basic
@Column(name = "description")
public String getDescription() { return description; }

public void setDescription(String description) {
    this.description = description;
}

@Basic
@Column(name = "picture")
public byte[] getPicture() { return picture; }

public void setPicture(byte[] picture) { this.picture = picture; }
```

Mapowania pozostałych pól w klasie,

Annotacja @Basic oznacza mapowanie kolumny zawierającej dane do pola w klasie.

```java
@OneToMany(mappedBy = "category", cascade = CascadeType.ALL, orphanRemoval = false, fetch = FetchType.LAZY)
public Collection<Product> getProductsInCategory() { return productsInCategory; }

public void setProductsInCategory(Collection<Product> productsByCategoryid) {
    this.productsInCategory = productsByCategoryid;
}
```

@OneToMany oznacza relację jedna kategoria do wielu produktów. MappedBy specyfikuje pole w klasie Product które mapuje nam tę relację, cascade specyfikuję rodzaj kaskady w danej relacji. Rodzaje kaskad w JPA:

- CascadeType.PERSIST : operacje save() I persist() są wykonywane na powiązanych encjach.
- CascadeType.MERGE : encje w releacji są mergowane w wypadku mergowania głównego obiektu.
- CascadeType.REFRESH : to samo co powyżej dla refresh().
- CascadeType.REMOVE : usuwa encje w relacji w wypadku usunięcia głównej encji.
- CascadeType.DETACH : wykonuje detach na powiązanych encjach w wypadku manualnego detach na głównej encji.
- CascadeType.ALL : zawiera wszystkie powyższe.

Dla naszej relacji ustawiamy ALL, przy pomocy orphanRemoval = false zabezpieczając się jednak przed usunięciem produktów powiązanych z kategorią. Nie chcemy tracić informacji o nich w wypadku usunięcia kategorii.

Fetch określa czy obiekty w relacji będą załadowane z bazy danych przy ładowaniu kategorii czy też dopiero przy próbie dostępu do tych obiektów. Mamy opcję FetchType.LAZY która ładuje obiekty przy próbie dostępu do nich, oraz opcję FetchType.EAGER, która ładuje je przy załadowaniu głównego obiektu(kategorii).

Klasa Customer:

```java
@Entity
@Table(name = "customers", schema = "public", catalog = "northwind")
public class Customer {
    private String customerID;
    private String companyName;
    private String contactName;
    private String contactTitle;
    private String address;
    private String city;
    private String region;
    private String postalcode;
    private String country;
    private String phone;
    private String fax;
    private Collection<Order> orders;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "customerid")
    public String getCustomerID() { return customerID; }
```

W powyższej klasie nie ma na razie nic nowego, dlatego nie będę jej głębiej analizował.

```java
@JsonManagedReference
@OneToMany(mappedBy = "customer", cascade = CascadeType.ALL, orphanRemoval = true)
public Collection<Order> getOrders() { return orders; }

public void setOrders(Collection<Order> ordersByCustomerid) { this.orders = ordersByCustomerid; }
```

Annotacja @JsonManagedReference będzie opisana w innej części dokumentacji i nie będę się na niej tutaj skupiał.

Interesującym nas miejscem jest opcja orphanRemoval = true. W tym wypadku ma ona więcej sensu, gdyż usuwając z bazy klienta nie mamy potrzeby trzymać informacji o jego zamówieniach.

Klasa CustomerDemo:

```java
@Entity
@Table(name = "customerdemographics", schema = "public", catalog = "northwind")
public class CustomerDemo {
    private String customerTypeId;
    private String customerDesc;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "customertypeid")
    public String getCustomerTypeId() { return customerTypeId; }
```

Klasa nie zawiera nowych elementów, więc nie będę jej opisywał.

Klasa Employee:

```java
@Entity
@Table(name = "employees")
public class Employee {
    private short employeeId;
    private String lastName;
    private String firstName;
    private String title;
    private String titleOfCourtesy;
    private Date birthDate;
    private Date hiredDate;
    private String address;
    private String city;
    private String region;
    private String postalCode;
    private String country;
    private String homePhone;
    private String extension;
    private byte[] photo;
    private String notes;
    private String photoPath;
    private Employee reportsTo;
    private Collection<Employee> managedEmployees;
    private Collection<Order> managedOrders;
```

```java
@JsonBackReference
@ManyToOne
@JoinColumn(name = "reportsto", referencedColumnName = "employeeid")
public Employee getReportsTo() { return reportsTo; }

public void setReportsTo(Employee employeesByReportsto) { this.reportsTo = employeesByReportsto; }

@JsonManagedReference
@OneToMany(mappedBy = "reportsTo", cascade = CascadeType.DETACH)
public Collection<Employee> getManagedEmployees() { return managedEmployees; }

public void setManagedEmployees(Collection<Employee> employeesByEmployeeid) {
    this.managedEmployees = employeesByEmployeeid;
}

@JsonBackReference
@OneToMany(mappedBy = "employee")
public Collection<Order> getManagedOrders() { return managedOrders; }
```

Ciekawą sprawą jest tutaj mapowanie do tej samej tabeli, przy pomocy annotacji @JoinColumn, w której określamy kolumnę na podstawie której mapujemy i jest to kolumna „reportsto” do primaryKey „employeeId”, co pozwala nam połączyć w relacji wielu jeden zarządzający pracowanik do wielu zarządzanych.

Odwrotna sytuacja wystepuje w przypadku relacji jeden pracownik zarządzający do wielu zarządzanych poniżej.

Klasa Order

```java
@Entity
@Table(name = "orders", schema = "public", catalog = "northwind")
public class Order {
    private short orderId;
    private Date orderDate;
    private Date requiredDate;
    private Date shippedDate;
    private Float freight;
    private String shipName;
    private String shipAddress;
    private String shipCity;
    private String shipRegion;
    private String shipPostalCode;
    private String shipCountry;
    @Cascade(value= org.hibernate.annotations.CascadeType.ALL)
    private Collection<OrderDetails> orderDetails;
    private Customer customer;
    private Employee employee;
    private Shipper shippedBy;
```

```java
@JsonManagedReference
@OneToMany(mappedBy = "order", fetch = FetchType.EAGER, cascade = CascadeType.ALL, orphanRemoval = false)
public Collection<OrderDetails> getOrderDetails() { return orderDetails; }

public void setOrderDetails(Collection<OrderDetails> orderDetailsByOrderid) {
    this.orderDetails = orderDetailsByOrderid;
}

@JsonBackReference
@ManyToOne
@JoinColumn(name = "customerid", referencedColumnName = "customerid")
public Customer getCustomer() { return customer; }

public void setCustomer(Customer customersByCustomerid) { this.customer = customersByCustomerid; }

@JsonManagedReference
@ManyToOne
@JoinColumn(name = "employeeid", referencedColumnName = "employeeid")
public Employee getEmployee() { return employee; }

public void setEmployee(Employee employeesByEmployeeid) { this.employee = employeesByEmployeeid; }

@JsonManagedReference
@ManyToOne
@JoinColumn(name = "shipvia", referencedColumnName = "shipperid")
public Shipper getShippedBy() { return shippedBy; }

public void setShippedBy(Shipper shippersByShipvia) { this.shippedBy = shippersByShipvia; }
```

W tej klasie nie występują konfiguracje nie opisane wcześniej, warto jest jedynie zwrócić uwagę na annotację @JoinColumn gdzie przez name ustawiamy nazwę kolumny z ForeignKey w tabeli Order a przez referencedColumnName nazwę kolumny primaryKey w tabeli będącej w zależności z tabelą orders.

Klasa OrderDetails:

```java
@Entity
@Table(name = "order_details", schema = "public", catalog = "northwind")
@IdClass(OrderDetailsPK.class)
public class OrderDetails implements Serializable {
    private short orderId;
    private short productId;
    private float unitPrice;
    private short quantity;
    private float discount;
    private Order order;
    private Product product;
```

```java
@JsonBackReference
@ManyToOne
@JoinColumn(name = "orderid", referencedColumnName = "orderid", nullable = false, insertable = false, updatable = false)
public Order getOrder() { return order; }

public void setOrder(Order ordersByOrderid) { this.order = ordersByOrderid; }

@JsonBackReference
@ManyToOne
@JoinColumn(name = "productid", referencedColumnName = "productid", nullable = false, insertable = false, updatable = false)
public Product getProduct() { return product; }

public void setProduct(Product productsByProductid) { this.product = productsByProductid; }
```

Ta klasa jest bardzo ciekawa z powodu wykorzystanego podwójnego klucza głównego złożonego z orderId oraz productId. Hibernate nie obsługuje tego przy pomocy oznaczenia @Id, musimy więc stworzyć specjalną klasę OrderDetailsPK i dodać ją jako ID przy pomocy annotacji @IdClass. Musimy także odpowiednio oznaczyć kolumny orderId oraz productId dla klasy orderDetails. Ustawiamy opcje: nullable = false, insertable = false, updatable = false, kolejno nullable = false : nie możemy ustawić null jako wartość tej kolumny, insertable=false, nie możemy insertować wartości tej kolumny, jest ona ustawiana na podstawie wartości powiązanej encji product/order, updatable =false : nie możemy zmienić tej wartości.

Klasa OrderDetailsPK:

```
public class OrderDetailsPK implements Serializable {
    private short orderId;
    private short productId;

    @Column(name = "orderid")
    @Id
    public short getOrderId() { return orderId; }

    public void setOrderId(short orderId) { this.orderId = orderId; }

    @Column(name = "productid")
    @Id
    public short getProductId() { return productId; }

    public void setProductId(short productId) { this.productId = productId; }
```

Klasa ta jest specjalnym miejscem gdzie przechowywane jest podwójny primary key klasy tabeli orders zmapowanej do klasy Order. Przy pomocy @Column oznaczana jest nazwa tabeli a przy pomocy @Id ustawiana jest ona jako Primary Key.

Klasa Product:

```
@Entity
@Table(name = "products", schema = "public", catalog = "northwind")
public class Product {
    private short productId;
    private String productName;
    private String quantityPerUnit;
    private Float unitPrice;
    private Short unitsInStock;
    private Short unitsOnOrder;
    private Short reorderLevel;
    private int discontinued;
    private Collection<OrderDetails> orderDetails;
    private Supplier supplier;
    private Category category;
```

```
@JsonManagedReference
@OneToMany(mappedBy = "product", cascade = CascadeType.ALL, orphanRemoval = false)
public Collection<OrderDetails> getOrderDetails() { return orderDetails; }

public void setOrderDetails(Collection<OrderDetails> orderDetailsByProductid) {
    this.orderDetails = orderDetailsByProductid;
}

@ManyToOne
@JoinColumn(name = "supplierid", referencedColumnName = "supplierid")
public Supplier getSupplier() { return supplier; }

public void setSupplier(Supplier suppliersBySupplierid) { this.supplier = suppliersBySupplierid; }

@ManyToOne
@JoinColumn(name = "categoryid", referencedColumnName = "categoryid")
public Category getCategory() { return category; }
```

W klasie tej nie występuje nic nowego, więc nie będę jej omawiać.

Klasa Region:

```java
@Entity
public class Region {
    private short regionId;
    private String regionDescription;
    private Collection<Territory> territories;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "regionid")
    public short getRegionId() { return regionId; }

    public void setRegionId(short regionid) { this.regionId = regionid; }

    @Basic
    @Column(name = "regiondescription")
    public String getRegionDescription() { return regionDescription; }

    public void setRegionDescription(String regiondescription) { this.regionDescription = regiondescription; }
```

W klasie nie występuje nic nowego, więc nie będę jej omawiać.

Klasa Shipper:

```java
@Entity
@Table(name = "shippers", schema = "public", catalog = "northwind")
public class Shipper {
    private short shipperId;
    private String companyName;
    private String phone;
    private Collection<Order> orders;
```

```java
@JsonBackReference
@OneToMany(mappedBy = "shippedBy")
public Collection<Order> getOrders() { return orders; }
```

Tutaj także nie mamy nic nowego.

Klasa Supplier:

```java
@Entity
@Table(name = "suppliers", schema = "public", catalog = "northwind")
public class Supplier {
    private short supplierId;
    private String companyName;
    private String contactName;
    private String contactTitle;
    private String address;
    private String city;
    private String region;
    private String postalCode;
    private String country;
    private String phone;
    private String fax;
    private String homepage;
    private Collection<Product> products;
```

```java
@OneToMany(mappedBy = "supplier", cascade = CascadeType.ALL, orphanRemoval = false)
public Collection<Product> getProducts() { return products; }
```

Klasa ta bazuje na już omówionych konfiguracjach.

Klasa Territory:

```java
@Entity
@Table(name = "territories", schema = "public", catalog = "northwind")
public class Territory {
    private String territoryId;
    private String territoryDescription;
    private Region region;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "territoryid")
    public String getTerritoryId() { return territoryId; }

    public void setTerritoryId(String territoryid) { this.territoryId = territoryid; }

    @Basic
    @Column(name = "territorydescription")
    public String getTerritoryDescription() { return territoryDescription; }

    public void setTerritoryDescription(String territorydescription) {
        this.territoryDescription = territorydescription;
    }
```

```java
@ManyToOne
@JoinColumn(name = "regionid", referencedColumnName = "regionid", nullable = false)
public Region getRegion() { return region; }

public void setRegion(Region regionByRegionid) { this.region = regionByRegionid; }
```

Klasa ta bazuje na już omówionych konfiguracjach.

Są to wszystkie klasy które są używane w naszym modelu. Przejdźmy teraz do tworzenia podstawowego, generycznego kontrolera do wykonywania podstawowch akcji na bazie danych.

5. Stworzenie generycznego kontrolera.

```java
public class BasicGenericHibernateController<K> {

    private final Session session;

    public BasicGenericHibernateController(Session session) { this.session = session; }

    public void saveObjectsToDb(List<K> objects){
        Transaction transaction = session.beginTransaction();

        for(K object : objects) {
            session.save(object);
        }

        transaction.commit();


    }

    public void removeObjectFromDb(List<K> objects){
        Transaction transaction = session.beginTransaction();
        for(K object : objects) {
            session.delete(object);
        }
        transaction.commit();


    }

    public List<K> selectFromTable(String tableName){
        Query<K> query = session.createQuery( s: "FROM " + tableName);

        return query.getResultList();
    }
}
```

Nasz kontroler trzyma obiekt session na którym będzie wykonywał tranzakcje i tworzył proste zapytanie o zawartość tabel.

Do metod saveObjectToDb oraz removeObjectFromDb przyjmujemy jako argumenty Listy Obiektów które będzie następnie zapisywał od bazy danych. Aby zoptymalizować czas wykonania, commit transakcji odbywa się po wykonaniu wszystkich operacji session.save(object) lub session.delete(object).

Metoda selectFromTable zwraca pełną listę encji znajdującej się w tabeli o nazwie podanej jako argument tableName, jako nazwy używamy nazwy klasy, ponieważ w takiej formie Hibernate mapuje tabele z bazy danych.

6. Testowanie modelu przy użyciu generycznego kontrolera.

Przeprowadzimy teraz przykładowy test poprawności konfiguracji modelu przy użyciu Generycznego kontrolera, tabel Employee, Order, OrderDetails oraz Product.

Aby ułatwić czytanie outputu dodajemy odpowiednie metody do klas:

Employee:

```
@Override
public String toString(){
    StringBuilder sb = new StringBuilder();

    sb.append("Name: " + firstName + " " + lastName + "\n");
    sb.append("Managed orders: \n");
    managedOrders.stream().forEach(order -> sb.append(order));

    return sb.toString();
}
```

Order:

```
@Override
public String toString(){
    StringBuilder sb = new StringBuilder();
    sb.append("Order id: "  + this.getOrderId() + "\n");
    sb.append("Client name: " + this.getCustomer().getContactName() + "\n");
    sb.append("Ordered products: " + "\n");
    this.getOrderDetails().stream().forEach(orderDetails -> sb.append(orderDetails));

    sb.append("\n");
    return sb.toString();
}
```

OrderDetails:

```
@Override
public String toString(){
    StringBuilder sb = new StringBuilder();

    sb.append("Product: " + this.getProduct().getProductName() + "\n");
    sb.append("Quantity: " + this.getQuantity() + "\n");
    sb.append("Price: " + this.getUnitPrice() * (1 - this.getDiscount()) + "\n");

    return sb.toString();
}
```

Ułatwią nam one wypisanie encji razem z ich powiązaniami.

Klasa Main użyta do testów:

```
public class Main {

    public static void main(String[] args) {

        Session session =
                HibernateSessionFactory.getSessionFactory().openSession();

        BasicGenericHibernateController<Employee> employeeBasicGenericHibernateController = new
                BasicGenericHibernateController<>(session);

        List<Employee> employeeList =
                employeeBasicGenericHibernateController.selectFromTable(Employee.class.getSimpleName());

        for(Employee employee : employeeList){
            System.out.println(employee);
        }

    }


}
```

Opisując kolejne linie kodu:

```
Session session =
        HibernateSessionFactory.getSessionFactory().openSession();
```

Otwieramy nową sesję z bazą danych otrzymując SessionFactory przy pomocy: HibernateSessionFactory.getSessionFactory()

A następnie otwierając nową sesję przez użycię openSession().

```
BasicGenericHibernateController<Employee> employeeBasicGenericHibernateController =
        new BasicGenericHibernateController<>(session);
```

Tworzymy nowy kontroler dla klasy Employee przy pomocy otwartej sesji.

```
List<Employee> employeeList =
        employeeBasicGenericHibernateController.selectFromTable(Employee.class.getSimpleName());
```

Pobieramy listę Employee przy pomocy kontrolera i metody getSimpleName() aby uzyskać nazwę zmapowanej przez Hibernate tabeli.

```
for(Employee employee : employeeList){
    System.out.println(employee);
}
```

Wypisujemy naszych pracowników.

Przykładowy select użyty przez hibernate do pobrania kolekcji managedOrders dla pracownika, jest to przykład użycia FetchType.LAZY, dla każdego pracownika w momencie zgłoszenia potrzeby dostępu od kolekcji obsługiwanych przez niego Orderów:

```
19:55:48.954 [main] DEBUG org.hibernate.loader.collection.plan.AbstractLoadPlanBasedCollectionInitializer - Loading collection:
[edu.agh.bazyprojekt.model.Employee.managedOrders#10]
19:55:48.954 [main] DEBUG org.hibernate.SQL -
  select
    managedord0_.employeeid as employe13_4_0_,
    managedord0_.orderid as orderid1_4_0_,
    managedord0_.orderid as orderid1_4_1_,
    managedord0_.customerid as custome12_4_1_,
    managedord0_.employeeid as employe13_4_1_,
    managedord0_.freight as freight2_4_1_,
    managedord0_.orderdate as orderdat3_4_1_,
    managedord0_.requireddate as required4_4_1_,
    managedord0_.shipaddress as shipaddr5_4_1_,
    managedord0_.shipcity as shipcity6_4_1_,
    managedord0_.shipcountry as shipcoun7_4_1_,
    managedord0_.shipname as shipname8_4_1_,
    managedord0_.shippostalcode as shippost9_4_1_,
    managedord0_.shipregion as shipreg10_4_1_,
    managedord0_.shipvia as shipvia14_4_1_,
    managedord0_.shippeddate as shipped11_4_1_,
    customer1_.customerid as customer1_2_2_,
    customer1_.address as address2_2_2_,
    customer1_.city as city3_2_2_,
    customer1_.companyname as companyn4_2_2_,
    customer1_.contactname as contactn5_2_2_,
    customer1_.contacttitle as contactt6_2_2_,
    customer1_.country as country7_2_2_,
    customer1_.fax as fax8_2_2_,
    customer1_.phone as phone9_2_2_,
    customer1_.postalcode as postalc10_2_2_,
    customer1_.region as region11_2_2_,
    shipper2_.shipperid as shipperi1_6_3_,
    shipper2_.companyname as companyn2_6_3_,
    shipper2_.phone as phone3_6_3_
  from
    public.orders managedord0_
  left outer join
    public.customers customer1_
      on managedord0_.customerid=customer1_.customerid
  left outer join
    public.shippers shipper2_
      on managedord0_.shipvia=shipper2_.shipperid
  where
    managedord0_.employeeid=?
Hibernate:
  select
    managedord0_.employeeid as employe13_4_0_,
    managedord0_.orderid as orderid1_4_0_,
    managedord0_.orderid as orderid1_4_1_,
    managedord0_.customerid as custome12_4_1_,
    managedord0_.employeeid as employe13_4_1_,
    managedord0_.freight as freight2_4_1_,
    managedord0_.orderdate as orderdat3_4_1_,
    managedord0_.requireddate as required4_4_1_,
    managedord0_.shipaddress as shipaddr5_4_1_,
    managedord0_.shipcity as shipcity6_4_1_,
```

```
    managedord0_.shipcountry as shipcoun7_4_1_,
    managedord0_.shipname as shipname8_4_1_,
    managedord0_.shippostalcode as shippost9_4_1_,
    managedord0_.shipregion as shipreg10_4_1_,
    managedord0_.shipvia as shipvia14_4_1_,
    managedord0_.shippeddate as shipped11_4_1_,
    customer1_.customerid as customer1_2_2_,
    customer1_.address as address2_2_2_,
    customer1_.city as city3_2_2_,
    customer1_.companyname as companyn4_2_2_,
    customer1_.contactname as contactn5_2_2_,
    customer1_.contacttitle as contactt6_2_2_,
    customer1_.country as country7_2_2_,
    customer1_.fax as fax8_2_2_,
    customer1_.phone as phone9_2_2_,
    customer1_.postalcode as postalc10_2_2_,
    customer1_.region as region11_2_2_,
    shipper2_.shipperid as shipperi1_6_3_,
    shipper2_.companyname as companyn2_6_3_,
    shipper2_.phone as phone3_6_3_
  from
    public.orders managedord0_
  left outer join
    public.customers customer1_
      on managedord0_.customerid=customer1_.customerid
  left outer join
    public.shippers shipper2_
      on managedord0_.shipvia=shipper2_.shipperid
  where
    managedord0_.employeeid=?
```

Podobnie wygląda sprawa dla inicjalizacji kolekcji OrderDetails dla encji Order:

```
19:55:48.574 [main] DEBUG org.hibernate.loader.collection.plan.AbstractLoadPlanBasedCollectionInitializer - Loading collection:
[edu.agh.bazyprojekt.model.Order.orderDetails#11034]
19:55:48.575 [main] DEBUG org.hibernate.SQL -
  select
    orderdetai0_.orderid as orderid2_3_0_,
    orderdetai0_.productid as producti1_3_0_,
    orderdetai0_.productid as producti1_3_1_,
    orderdetai0_.orderid as orderid2_3_1_,
    orderdetai0_.discount as discount3_3_1_,
    orderdetai0_.quantity as quantity4_3_1_,
    orderdetai0_.unitprice as unitpric5_3_1_,
    product1_.productid as producti1_5_2_,
    product1_.categoryid as category9_5_2_,
    product1_.discontinued as disconti2_5_2_,
    product1_.productname as productn3_5_2_,
    product1_.quantityperunit as quantity4_5_2_,
    product1_.reorderlevel as reorderl5_5_2_,
    product1_.supplierid as supplie10_5_2_,
    product1_.unitprice as unitpric6_5_2_,
    product1_.unitsinstock as unitsins7_5_2_,
    product1_.unitsonorder as unitsono8_5_2_,
    category2_.categoryid as category1_0_3_,
    category2_.categoryname as category2_0_3_,
    category2_.description as descript3_0_3_,
    category2_.picture as picture4_0_3_,
    supplier3_.supplierid as supplier1_7_4_,
    supplier3_.address as address2_7_4_,
    supplier3_.city as city3_7_4_,
```

```
    supplier3_.companyname as companyn4_7_4_,
    supplier3_.contactname as contactn5_7_4_,
    supplier3_.contacttitle as contactt6_7_4_,
    supplier3_.country as country7_7_4_,
    supplier3_.fax as fax8_7_4_,
    supplier3_.homepage as homepage9_7_4_,
    supplier3_.phone as phone10_7_4_,
    supplier3_.postalcode as postalc11_7_4_,
    supplier3_.region as region12_7_4_
  from
    public.order_details orderdetai0_
  inner join
    public.products product1_
      on orderdetai0_.productid=product1_.productid
  left outer join
    public.categories category2_
      on product1_.categoryid=category2_.categoryid
  left outer join
    public.suppliers supplier3_
      on product1_.supplierid=supplier3_.supplierid
  where
    orderdetai0_.orderid=?
Hibernate:
  select
    orderdetai0_.orderid as orderid2_3_0_,
    orderdetai0_.productid as producti1_3_0_,
    orderdetai0_.productid as producti1_3_1_,
    orderdetai0_.orderid as orderid2_3_1_,
    orderdetai0_.discount as discount3_3_1_,
    orderdetai0_.quantity as quantity4_3_1_,
    orderdetai0_.unitprice as unitpric5_3_1_,
    product1_.productid as producti1_5_2_,
    product1_.categoryid as category9_5_2_,
    product1_.discontinued as disconti2_5_2_,
    product1_.productname as productn3_5_2_,
    product1_.quantityperunit as quantity4_5_2_,
    product1_.reorderlevel as reorderl5_5_2_,
    product1_.supplierid as supplie10_5_2_,
    product1_.unitprice as unitpric6_5_2_,
    product1_.unitsinstock as unitsins7_5_2_,
    product1_.unitsonorder as unitsono8_5_2_,
    category2_.categoryid as category1_0_3_,
    category2_.categoryname as category2_0_3_,
    category2_.description as descript3_0_3_,
    category2_.picture as picture4_0_3_,
    supplier3_.supplierid as supplier1_7_4_,
    supplier3_.address as address2_7_4_,
    supplier3_.city as city3_7_4_,
    supplier3_.companyname as companyn4_7_4_,
    supplier3_.contactname as contactn5_7_4_,
    supplier3_.contacttitle as contactt6_7_4_,
    supplier3_.country as country7_7_4_,
    supplier3_.fax as fax8_7_4_,
    supplier3_.homepage as homepage9_7_4_,
    supplier3_.phone as phone10_7_4_,
    supplier3_.postalcode as postalc11_7_4_,
    supplier3_.region as region12_7_4_
  from
    public.order_details orderdetai0_
  inner join
    public.products product1_
      on orderdetai0_.productid=product1_.productid
```

```
left outer join
   public.categories category2_
      on product1_.categoryid=category2_.categoryid
left outer join
   public.suppliers supplier3_
      on product1_.supplierid=supplier3_.supplierid
where
   orderdetai0_.orderid=?
```

Teraz program wypisuje nam przykładowego pracownika razem z zarządzanymi przez niego Orderami oraz ich detalami:

Name: Anne Dodsworth
Managed orders:
Order id: 10255
Client name: Michael Holz
Ordered products:
Product: Chang
Quantity: 20
Price: 15.2
Product: Pavlova
Quantity: 35
Price: 13.9
Product: Inlagd Sill
Quantity: 25
Price: 15.2
Product: Raclette Courdavault
Quantity: 30
Price: 44.0

Order id: 10263
Client name: Roland Mendel
Ordered products:
Product: Pavlova
Quantity: 60
Price: 10.424999
Product: Guaraná Fantástica
Quantity: 28
Price: 3.6
Product: Nord-Ost Matjeshering
Quantity: 60
Price: 15.525001
Product: Longlife Tofu
Quantity: 36
Price: 6.0

Order id: 10324
Client name: Jose Pavarotti
Ordered products:
Product: Pavlova
Quantity: 21
Price: 11.815
Product: Steeleye Stout
Quantity: 70
Price: 12.24
Product: Spegesild
Quantity: 30
Price: 9.6
Product: Raclette Courdavault
Quantity: 40

Price: 37.4
Product: Vegie-spread
Quantity: 80
Price: 29.835

Order id: 10331
Client name: Laurence Lebihan
Ordered products:
Product: Tourtière
Quantity: 15
Price: 5.9

Order id: 10386
Client name: Aria Cruz
Ordered products:
Product: Guaraná Fantástica
Quantity: 15
Price: 3.6
Product: Sasquatch Ale
Quantity: 10
Price: 11.2

Nie są to wszystkie Ordery dla tego pracownika, aby nie przedłużać niepotrzebnie dokumentacji skorzystałem z ograniczonej ich ilości. Demonstruje to jednak, że model jest skonfigurowany w odpowiedni sposób, i rozpoczynając od encji Employee możemy dostać się do dowolnej tabeli która jest związana z tą encją w łańcuchu relacji.

7. Podsumowanie

W dokumentacji tej zostało przedstawione krok po kroku instalacja serwera PostgreSQL, stworzenie bazy danych oraz jej użytkownika, konfiguracja Hibernate, generowanie oraz konfiguracja modelu, stworzenie prostego kontrolera obsługującego proste operacje na bazie danych oraz przetestowanie stworzonego kodu. W następnych częściach dokumentacji zostanie opisane użycie CriteriaAPI do zawężania wyników zapytań, tworzenie podstawowego serwera obsługującego żądania oraz GUI.