

# ***Analiza Aplicatie***

**Prof. Coordonator Lectr. dr.Gabriela Mihai**

**Grupa 506**

**Studenti:**

**Badea Bogdan-Andrei**

**Balica Adrian-Claudiu**

**Mancila Doru-Bogdan**

**Universitatea din Bucuresti**

**2023**

## Cuprins

<b>1. Scopul aplicației.....</b>	<b>3</b>
<b>2. Modul de conectare .....</b>	<b>3</b>
<b>3. Swagger API .....</b>	<b>4</b>
<b>4. Structura și elemente .....</b>	<b>10</b>
<b>4.1. Entitati .....</b>	<b>10</b>
<b>4.2. Mappere .....</b>	<b>11</b>
<b>4.3. Dtos.....</b>	<b>12</b>
<b>4.4. Repository .....</b>	<b>13</b>
<b>4.5. Services .....</b>	<b>14</b>
<b>4.6 . Controllers .....</b>	<b>15</b>
<b>5. Verificarea modului în care datele sunt stocate în cele două baze de date distribuite .....</b>	<b>16</b>
<b>5.1. Types.....</b>	<b>16</b>
<b>5.2. Cities .....</b>	<b>18</b>
<b>5.3. Restaurants.....</b>	<b>23</b>
<b>5.4. Employees .....</b>	<b>27</b>
<b>5.5. Menus .....</b>	<b>32</b>
<b>5.6. Drinks.....</b>	<b>34</b>
<b>5.7. Dishes.....</b>	<b>35</b>
<b>5.8. Orders, Orders_drinks, Chefs_orders_dishes.....</b>	<b>36</b>

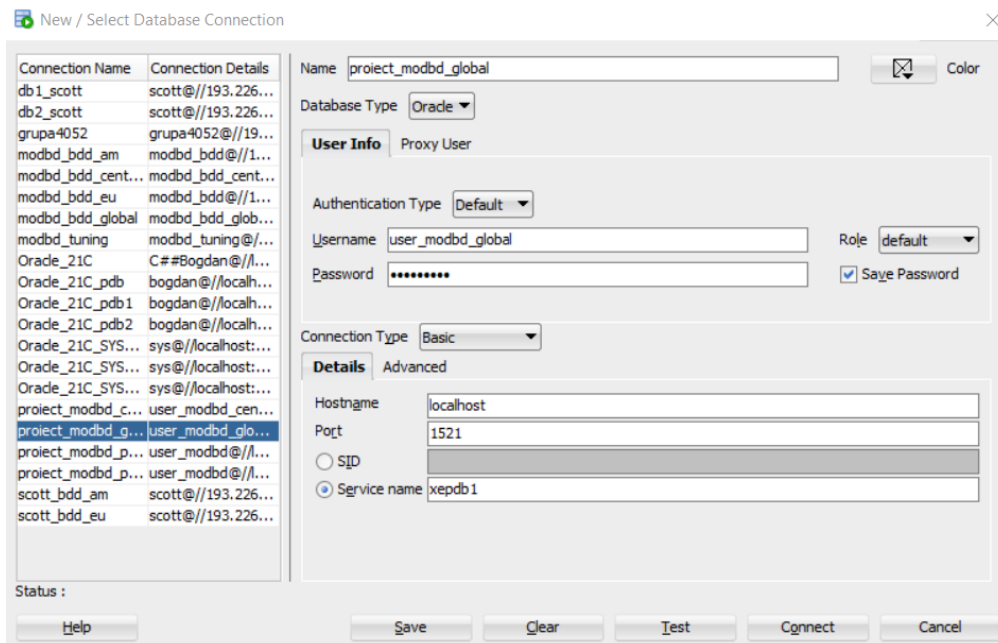
## 1. Scopul aplicației

Aplicația este destinată unui lanț de restaurante cu scopul de a gestiona atât angajații cât și meniul și activitatea lor din restaurantele respective. Această va avea la baza un sistem REST API creat cu limbajul java folosind springboot, ce se va baza pe request-uri http pentru a comunica cu baza de date pentru a putea efectua operații precum GET, UPDATE, DELETE, POST echivalente instrucțiunilor din oracle de SELECT, INSERT, UPDATE. Mai concret, aplicația va gestiona locația restaurantelor, tipurile de angajați care pot exista, angajați care fac parte dintr-un anumit restaurant, meniurile alături de băuturile și mâncărurile aferente acestuia și în final gestionarea dintre angajați și comenzi, angajatul putând fi chelner și a prelua o comandă sau bucătar și să aibă misiunea de a găti felul de mâncare corespunzător.

## 2. Modul de conectare

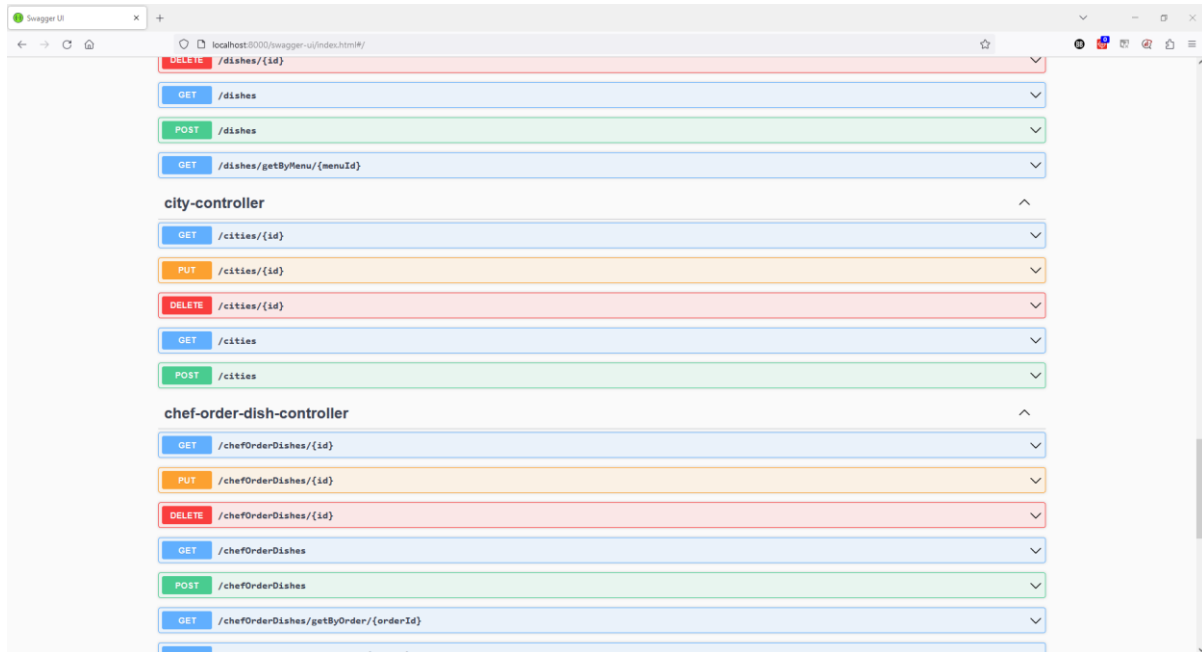
Aplicația se conectează la baza de date globală realizată în Oracle 21C separat. Această baza de date este folosită ulterior că fiind o baza de date pluggable către aplicația noastră. Această folosește un driver oracle jdbc, Oracle Driver și se conectează la baza de date prin service name – xepdb1, parolă fiind Password1 iar usernamenul care stă la baza conexiunii user\_modb\_global. Aplicația vede această conexiune ca o baza de date normală, neștiind că este fragmentată.

```
1 server.port=8000
2
3 # OracleDB connection settings
4 spring.datasource.url=jdbc:oracle:thin:@//localhost:1521/xepdb1
5 spring.datasource.username=user_modbd_global
6 spring.datasource.password=Password1
7 spring.datasource.driver-class-name=oracle.jdbc.OracleDriver
```

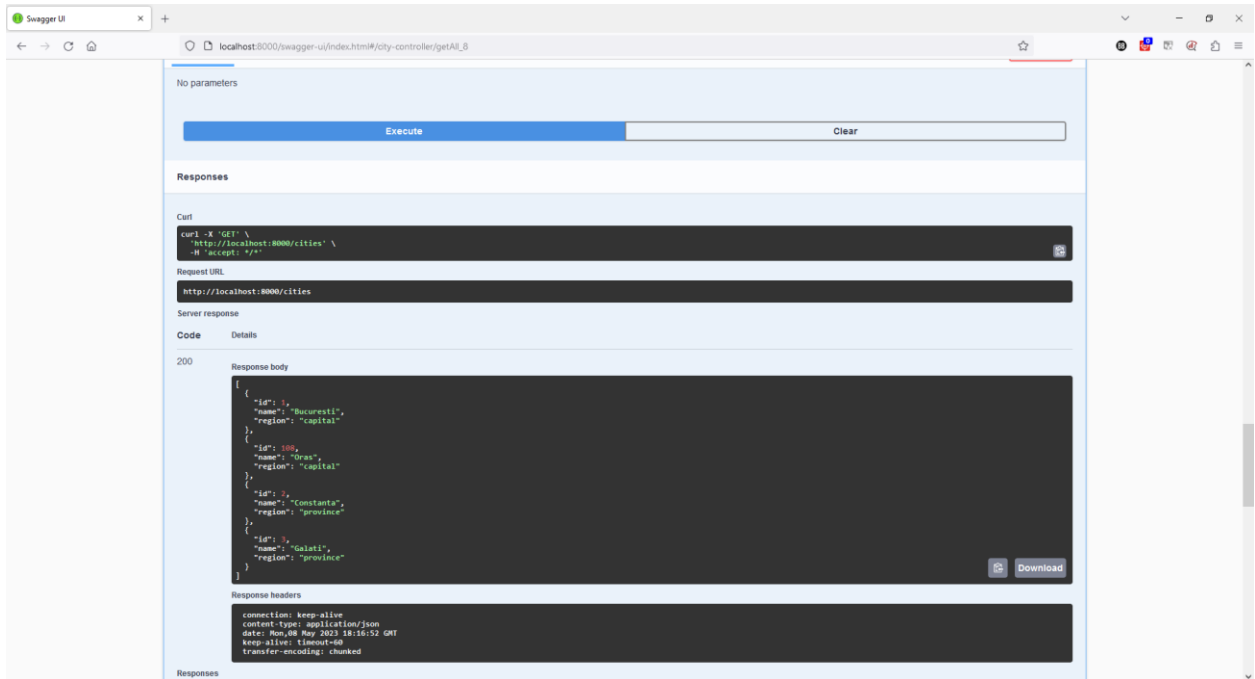


### 3. Swagger API

Pentru a folosi logică din back-end și a folosi logică create în baza de date (triggeri, secvențe, view pentru a prelua informațiile din ambele baze de date) se folosește un Swagger care expune metodele http disponibile pentru a fi folosite și a vedea funcționalitatea acestora. Aceleași metode pot fi definiție și folosite și într-o aplicație precum Postman.

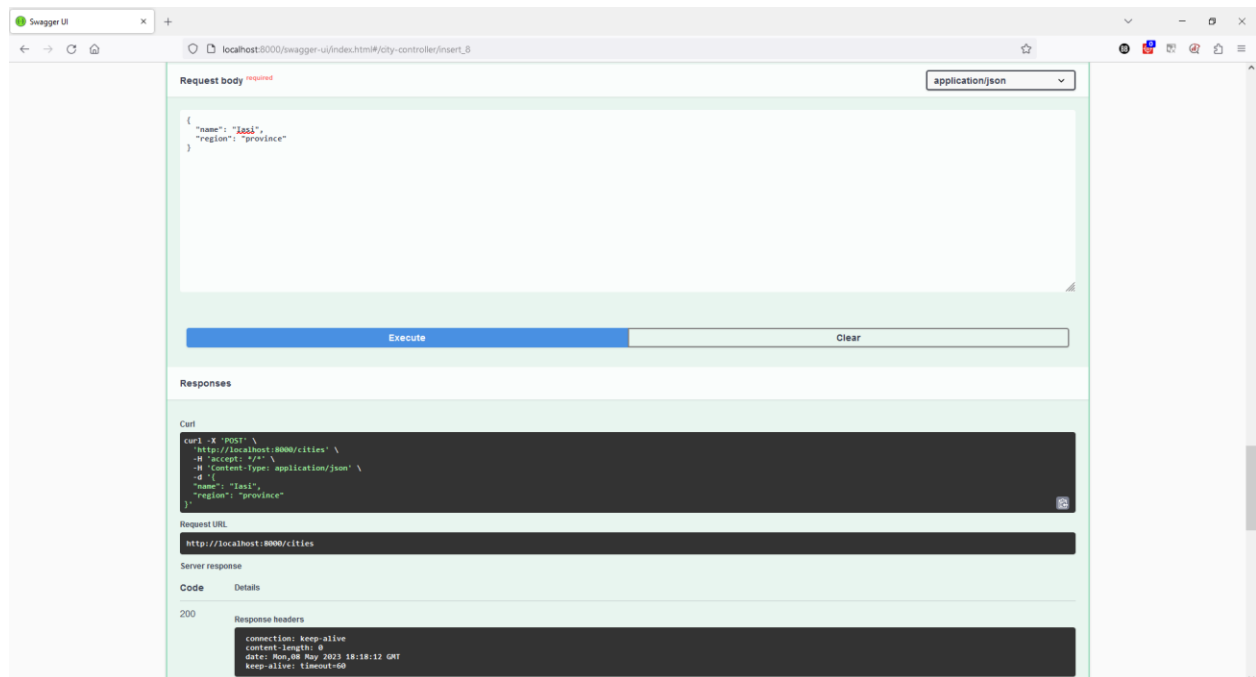


Structura swagerr-ului împărțit pe controller



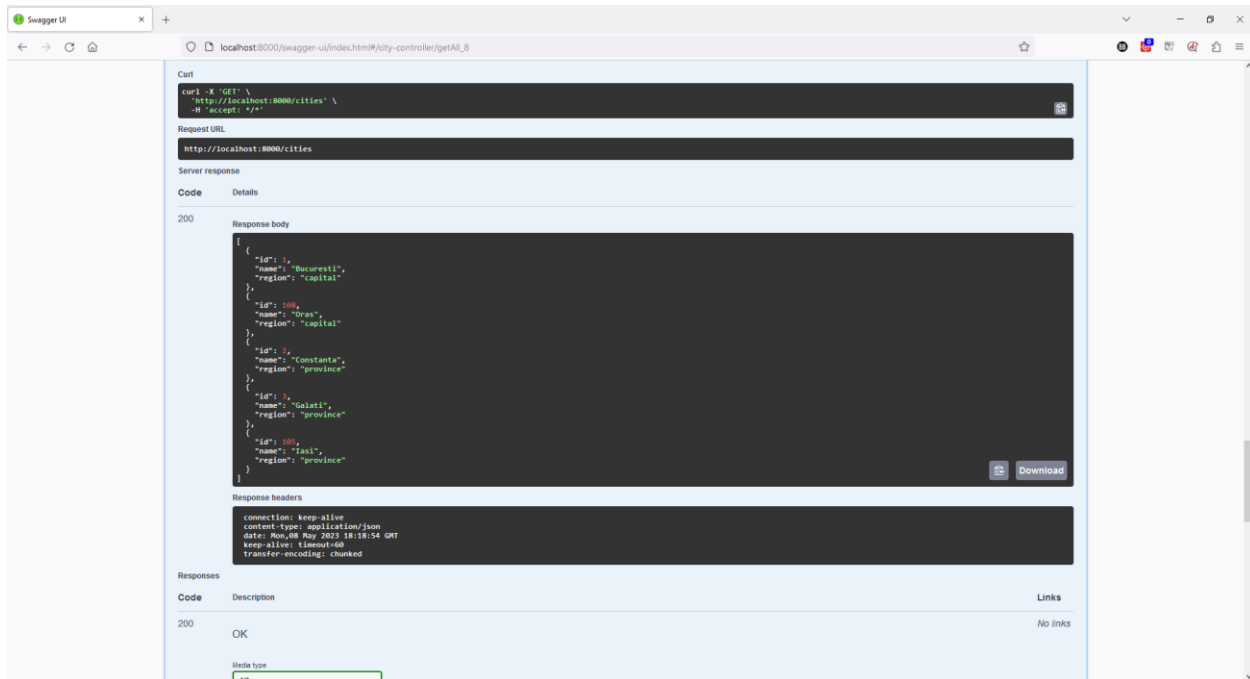
GetAll pentru toate orasele din baza de date

După cum se poate vedea în screenshot-ul de mai sus, metoda get aduce într-un vector toate orașele disponibil în baza de date, patru la număr. Acesta accesează în spate un view care face merge la informațiile fragmentate.

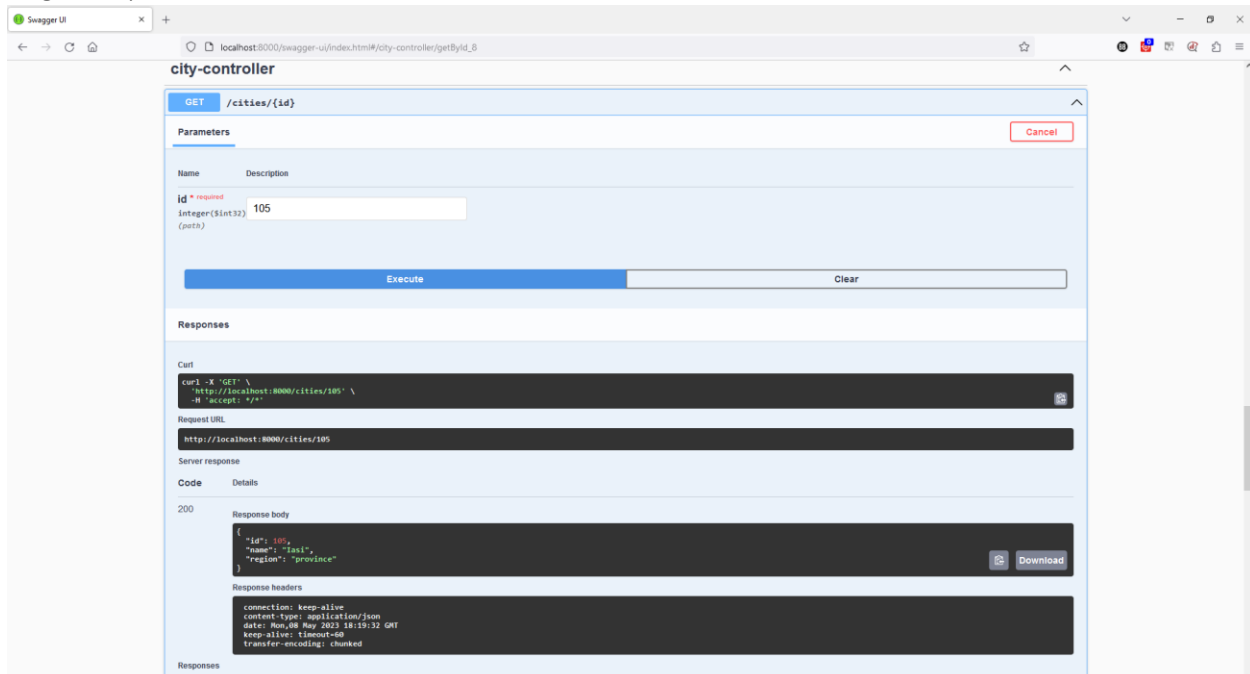


Insert prin metoda POST

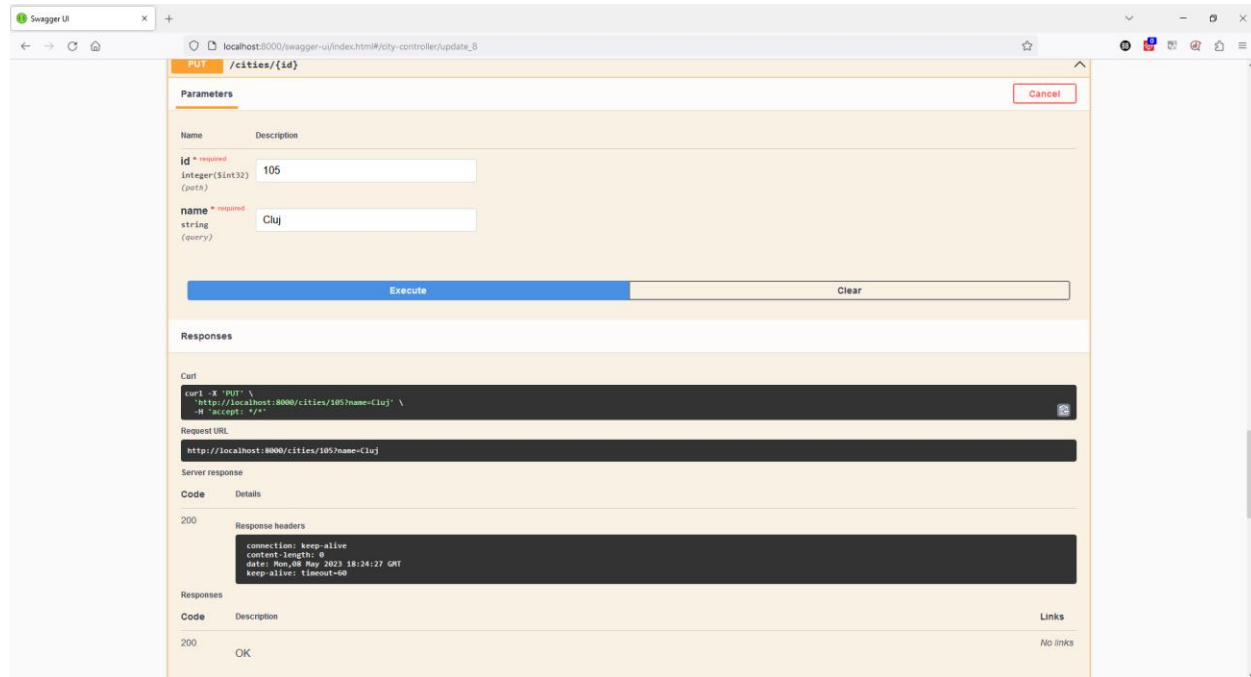
Prin metodă POST și oferirea corectă a unui body de tip json oferind numele și regiunea putem adăuga un nou oraș. Rezultatul îl putem vedea în screenshot-ul următor în care am adăugat orașul iași cu id-ul 105.



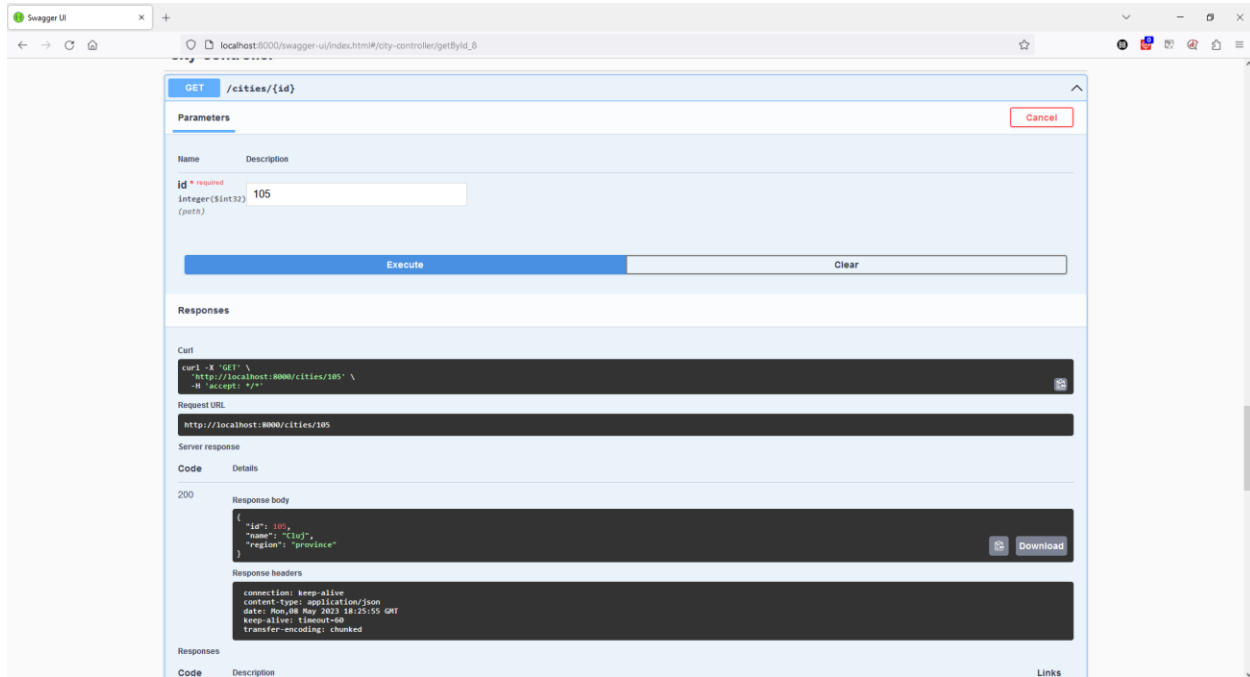
De asemenea, avem posibilitatea de a aduce și orașul după Id în caz de ne dorim să vedem doar un singur oraș



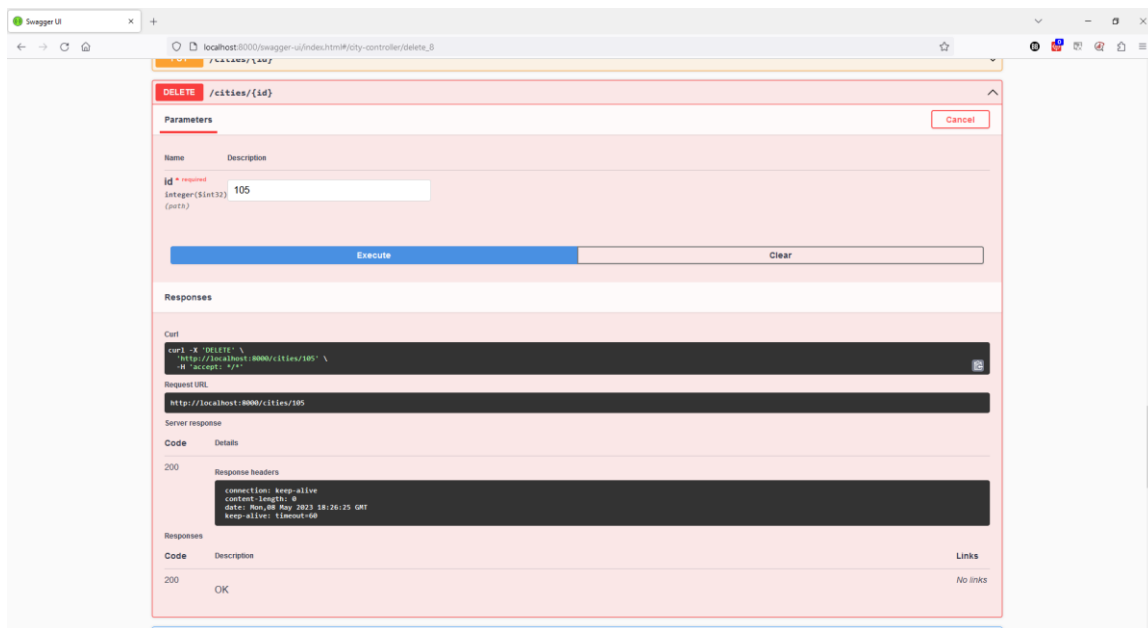
Putem de asemenea, să actualizăm și să ștergem elemente din baza de date.



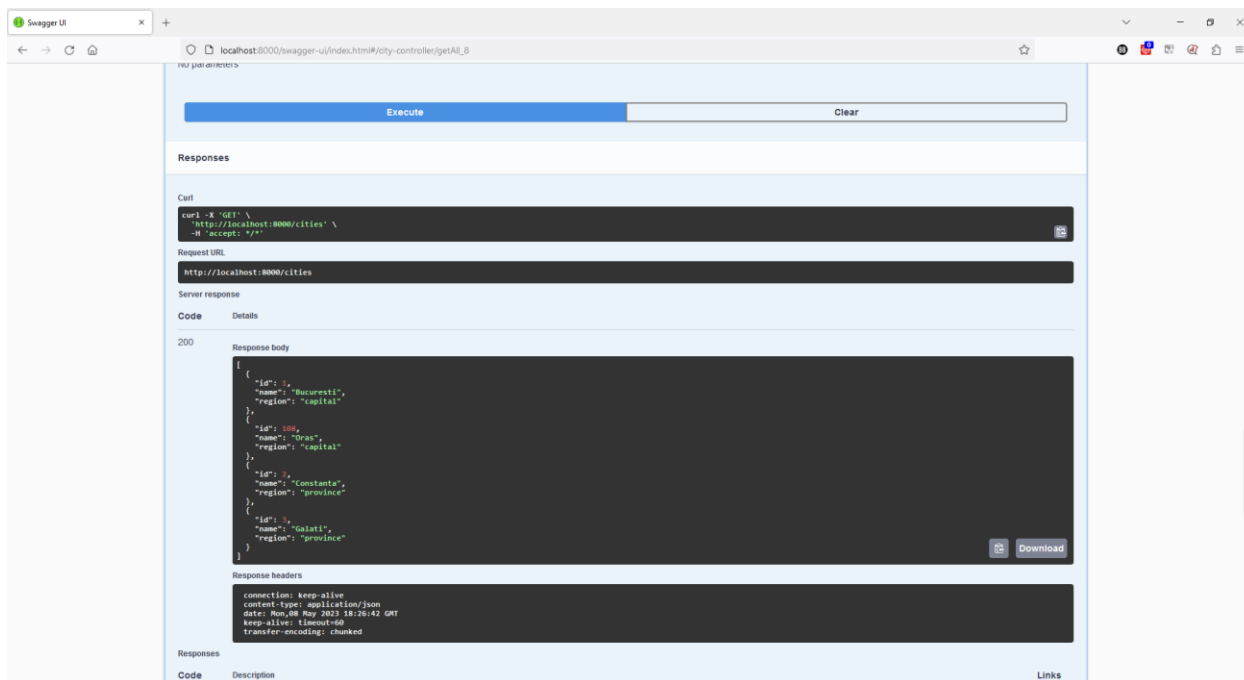
Metoda Put ce actualizeaza orasul cu id-ul 105 care initial era Iasi in Cluj



Rezultatul după actualizare



Ștergerea orasului prin metoda Delete

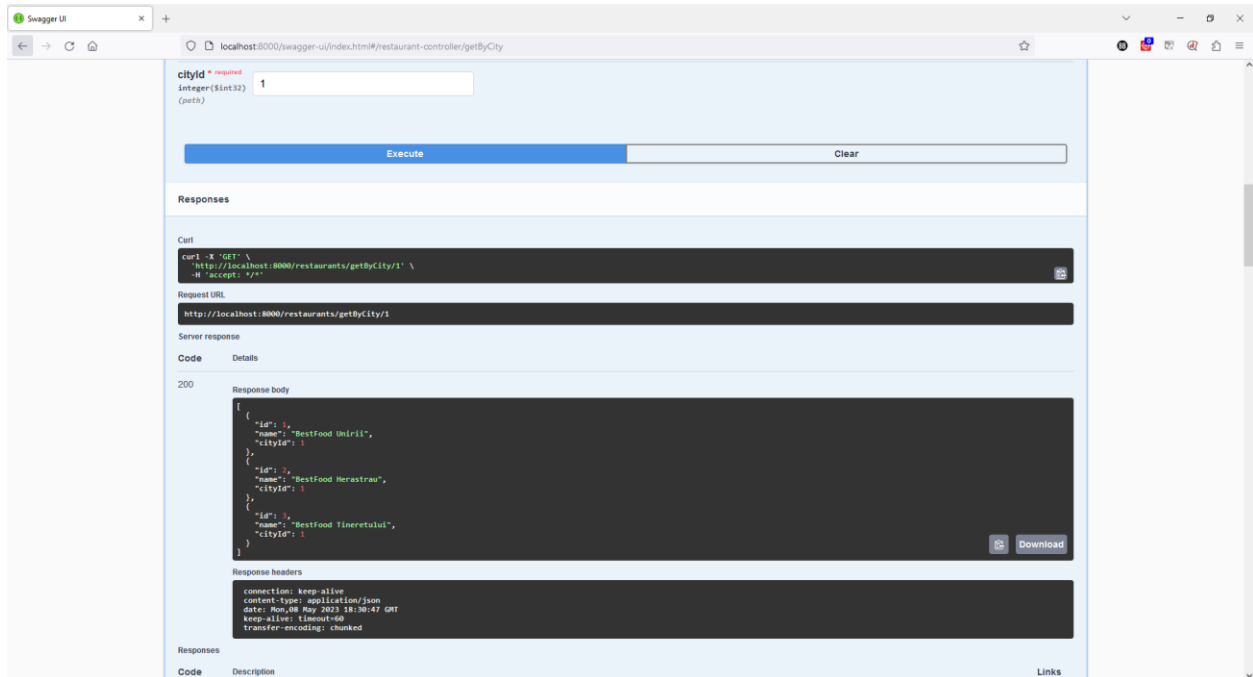


Rezultat după ștergere

Aceste operații prezentate mai sus se aplică pentru toate entitățile create pentru aplicație. În plus mai sunt metode GET care se bazează pe cheile străine precum: aducerea tuturor comenzilor care au chelnerului cu ID-ul X sau pentru un bucătar, aducerea tuturor restaurantelor dintr-un oraș, aducerea



băuturilor sau a mâncării care se află în meniu și așa mai departe. Un astfel de exemplu îl regăsim în metoda GET următoare:



Aducerea restaurantelor pentru orasul cu ID-ul 1

## 4. Structura și elemente

### 4.1 Entități

Entitățile sunt create cu adnotări specific precum `@Column` ce specifică numele coloanei. De asemenea, avem și relații precum `@OneToMany`, `@ManyToOne` pentru a ilustra relațiile pe care le avem între tabele și cum mapăm datele, iar în plus avem și `@JoinColumn()` unde specificăm detaliile pentru foreign key.

```
@Data
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
@Entity
@Table(name = "EMPLOYEES")
public class Employee {

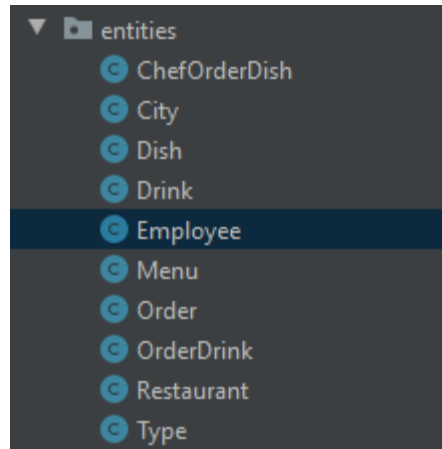
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "ID")
    private Integer id;
    @Column(name = "NAME")
    private String name;
    @Column(name = "SALARY")
    private Integer salary;
    @Column(name = "HIRE_DATE")
    private LocalDate hireDate;
    @Column(name = "TYPE_ID")
    private Integer typeId;
    @Column(name = "RESTAURANT_ID")
    private Integer restaurantId;

    @ManyToOne
    @JoinColumn(name="TYPE_ID", insertable = false, updatable = false, nullable = false)
    private Type type;

    @ManyToOne
    @JoinColumn(name="RESTAURANT_ID", insertable = false, updatable = false, nullable = false)
    private Restaurant restaurant;

    @OneToMany(mappedBy="employee")
    private List<Order> orders;
    @OneToMany(mappedBy="employee")
    private List<ChefOrderDish> chefOrderDishes;
}
```

Acceași structura ești folosită și pentru restul entităților create.



## 4.2 Mappere

Mapper-ele sunt un mecanism prin care se face maparea dintre dto-uri, obiectele trimise pe request și modul și formă în care acestea trebuie să ajungă în baza de date. Există mapere pentru fiecare entitate în parte.

```
package com.example.modbd.mappers;

import ...

@Mapper(componentModel = "spring")
public interface CityMapper {

    @Mapping(target = "id", source = "city.id")
    @Mapping(target = "name", source = "city.name")
    @Mapping(target = "region", source = "city.region")
    CityDto mapToDto(City city);

    @Mapping(target = "id", source = "cityDto.id")
    @Mapping(target = "name", source = "cityDto.name")
    @Mapping(target = "region", source = "cityDto.region")
    City mapToEntity(CityDto cityDto);
}
```

## 4.3 Dtos

DTO- Data transfer object reprezintă structura care mulează practice informația transmisă pe body într-un request de tip http POST sau PUT pentru a putea ulterior folosi și accesa informația respective.

```
package com.example.modbd.dtos;

import lombok.*;

@Data
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
@Builder
public class CityDto {

    private Integer id;
    private String name;
    private String region;

}
```

- ChefOrderDishDto
- CityDto
- DishDto
- DrinkDto
- EmployeeDto
- MenuDto
- OrderDrinkDto
- OrderDto
- RestaurantDto
- TypeDto

## 4.4 Repository

```
package com.example.modbdd.repository;

import ...

@Repository
public interface RestaurantRepository extends JpaRepository<Restaurant, Integer> {

    @Query(value = "SELECT * FROM restaurants", nativeQuery = true)
    List<Restaurant> findAll();

    @Query(value = "SELECT * FROM restaurants WHERE id = :id", nativeQuery = true)
    Optional<Restaurant> findById(Integer id);

    @Query(value = "SELECT * FROM restaurants WHERE city_id = :cityId", nativeQuery = true)
    List<Restaurant> findByCity(Integer cityId);

    @Modifying
    @Query(value = "INSERT INTO restaurants (name, city_id) VALUES (:name, :cityId)", nativeQuery = true)
    void insert(String name, Integer cityId);

    @Modifying
    @Query(value = "UPDATE restaurants SET name = :name WHERE id = :id", nativeQuery = true)
    void updateById(Integer id, String name);

    @Modifying
    @Query(value = "DELETE FROM restaurants WHERE id = :id", nativeQuery = true)
    void deleteById(Integer id);
}
```

Pe partea de repository avem metodele necesare pentru a comunica cu baza de date, insert, select, update și delete. Acestea sunt scrise în sql nativ folosind flagul `nativeQuery = true`. Practic acestea comunica direct cu baza de date globală urmând că ulterior această să distribuie datele sau să le adune în funcție de necesitate, în acest mod practice aplicația folosește baza de date distribuită că o oricare altă baza de date. Ele sunt marcate cu adnotarea `@Repository` pentru a marca obiectivul acestora.

## 4.5 Services

```
@Service
public class DishService {

    @Autowired
    DishRepository dishRepository;

    @Autowired
    DishMapper dishMapper;

    public List<DishDto> getAll() {

        List<Dish> dishes = dishRepository.findAll();
        List<DishDto> dishDtos = new ArrayList<>();
        for (Dish dish : dishes) {
            DishDto dishDto = dishMapper.mapToDto(dish);
            dishDtos.add(dishDto);
        }
        return dishDtos;
    }

    public DishDto getById(Integer id) {

        Optional<Dish> dish = dishRepository.findById(id);
        DishDto dishDto = dishMapper.mapToDto(dish.get());
        return dishDto;
    }

    public List<DishDto> getByMenu(Integer menuId) {

        List<Dish> dishes = dishRepository.findByMenu(menuId);
        List<DishDto> dishDtos = new ArrayList<>();
        for (Dish dish : dishes) {
            DishDto dishDto = dishMapper.mapToDto(dish);
            dishDtos.add(dishDto);
        }
        return dishDtos;
    }
}
```

```
@Transactional
public void insert(DishDto dishDto) {

    Dish dish = dishMapper.mapToEntity(dishDto);
    dishRepository.insert(dish.getName(), dish.getPrice(), dish.getMenuId());
}

@Transactional
public void update(Integer id, String name, Integer price) {

    dishRepository.updateById(id, name, price);
}

@Transactional
public void deleteById(Integer id) {

    dishRepository.deleteById(id);
}
```

Serviciile sunt locul în care se folosesc metodele din repository și mapperele pentru a crea logică care este necesară pentru realizarea operațiilor. La rândul lor acestea vor fi folosite în controllere. Acestea sunt marcate cu adnotarea @Service

## 4.6 Controllers

În controllere aveam practic partea aplicației care generează swagger-ul și expune endpoint-uri pentru endapps. Această este marcată prin anotarea `@RestController` iar path pentru endpoint-ul respective se definește prin `@RequestMapping`. De asemenea se folosește și `@Autowired` pentru a putea inițializa serviciul necesar în controlerul aferent

```
@RestController
@RequestMapping("/employees")
public class EmployeeController {

    @Autowired
    EmployeeService employeeService;

    @GetMapping()
    public List<EmployeeDto> getAll() {

        return employeeService.getAll();

    }

    @GetMapping("/{id}")
    public EmployeeDto getById(@PathVariable Integer id) {

        return employeeService.getById(id);

    }

    @GetMapping("/getByType/{typeId}")
    public List<EmployeeDto> getByType(@PathVariable Integer typeId) {

        return employeeService.getByType(typeId);

    }

    @GetMapping("/getByRestaurant/{restaurantId}")
    public List<EmployeeDto> getByRestaurant(@PathVariable Integer restaurantId) {

        return employeeService.getByRestaurant(restaurantId);

    }

    @PostMapping()
    public void insert(@RequestBody EmployeeDto employeeDto) {

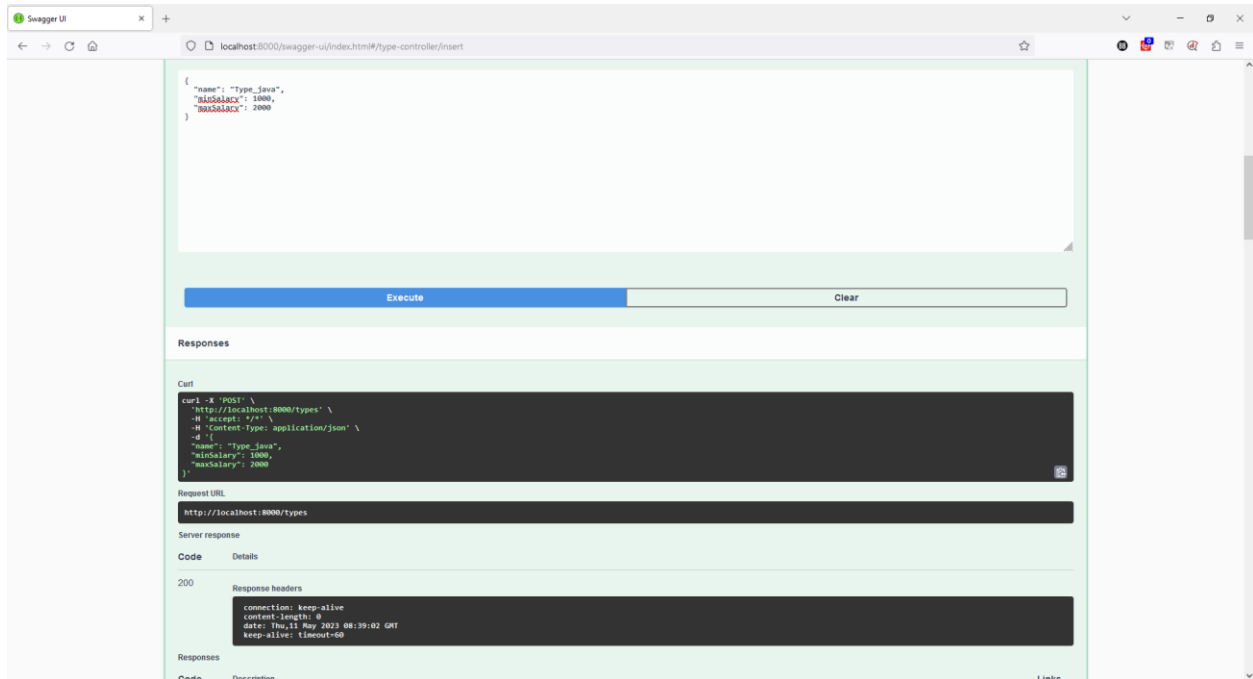
        employeeService.insert(employeeDto);

    }
}
```

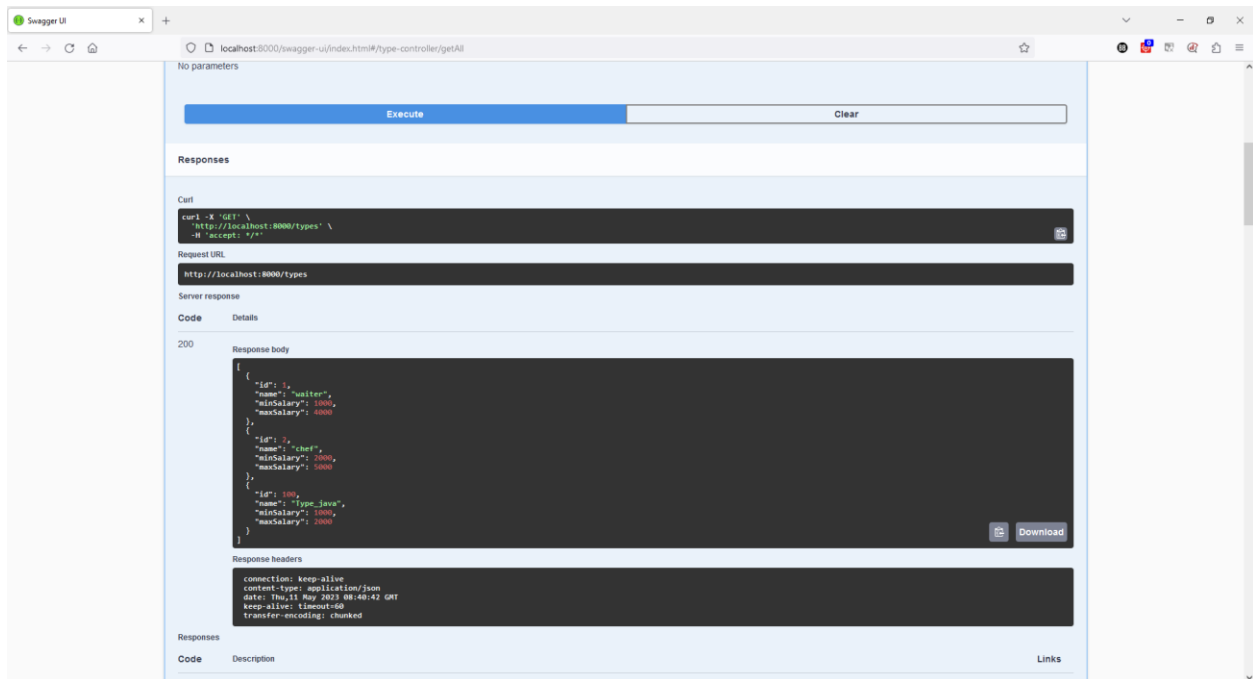
## 5. Verificarea modului în care datele sunt stocate în cele două baze de date distribuite

### 5.1. Types

Prima dată, vom crea un tip nou de angajat. Tabela types este fragmentată vertical în baza de date.



Apoi vom afișa toate valorile din tabela types pentru a vedea noile valori salavate.





Actualizăm obiectul proaspăt inserat.

The screenshot shows the Swagger UI interface for a PUT endpoint. The URL bar indicates the endpoint is `localhost:8000/swagger-ui/index.html#/type-controller/update`. The request parameters are as follows:

Name	Description
<b>id</b> * required	integer(\$int32) (path)
<b>name</b> * required	string (query)
<b>minSalary</b> * required	integer(\$int32) (query)
<b>maxSalary</b> * required	integer(\$int32) (query)

The input fields contain the following values:

- id: 100
- name: Type\_java\_2
- minSalary: 1001
- maxSalary: 2001

Buttons for "Execute" and "Clear" are visible. Below the request section, the "Responses" section shows a 200 status code with the following headers:

```
connection: keep-alive
content-length: 0
date: Thu, 11 May 2023 08:45:36 GMT
keep-alive: timeout=60
```

Verificăm că obiectul a fost actualizat.

The screenshot shows the Swagger UI interface for a GET endpoint. The URL bar indicates the endpoint is `localhost:8000/swagger-ui/index.html#/type-controller/getById`. The request parameter is as follows:

Name	Description
<b>id</b> * required	integer(\$int32) (path)

The input field contains the value: 100. Buttons for "Execute" and "Clear" are visible. Below the request section, the "Responses" section shows a 200 status code with the following response body:

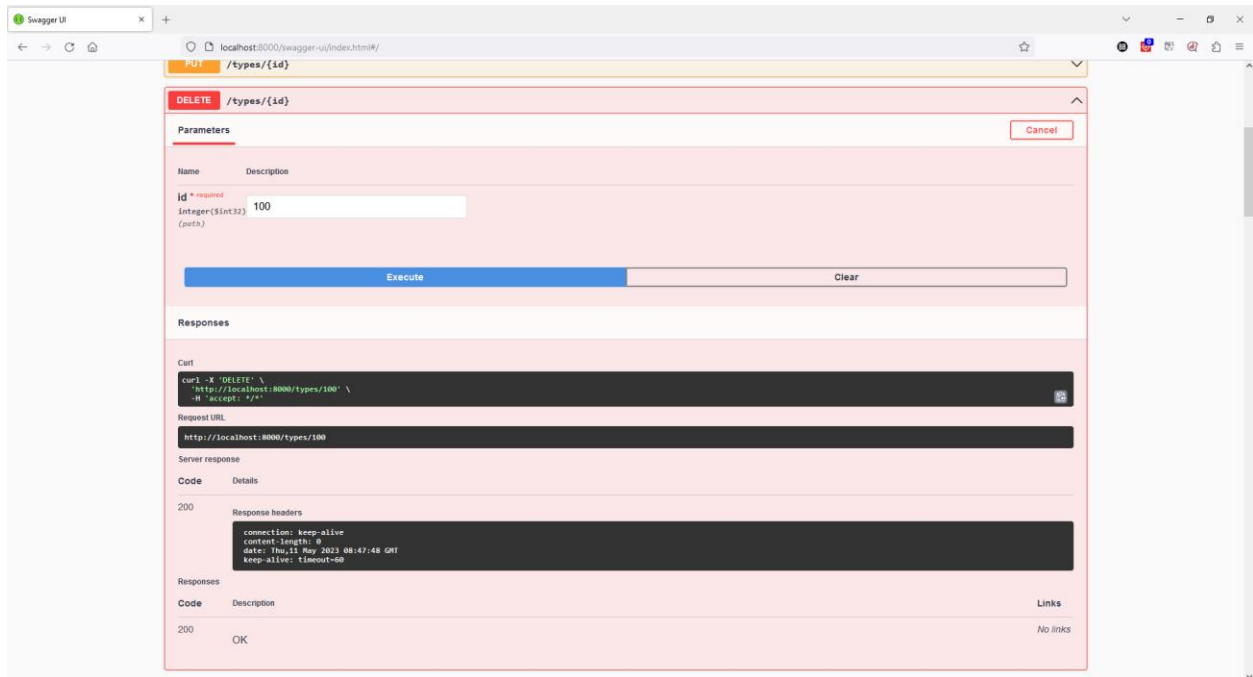
```
{
  "id": 100,
  "name": "Type_java_2",
  "minSalary": 1001,
  "maxSalary": 2001
}
```

The response headers are:

```
connection: keep-alive
content-type: application/json
date: Thu, 11 May 2023 08:47:13 GMT
keep-alive: timeout=60
transfer-encoding: chunked
```

The "Responses" table at the bottom shows a 200 status code with the description "OK".

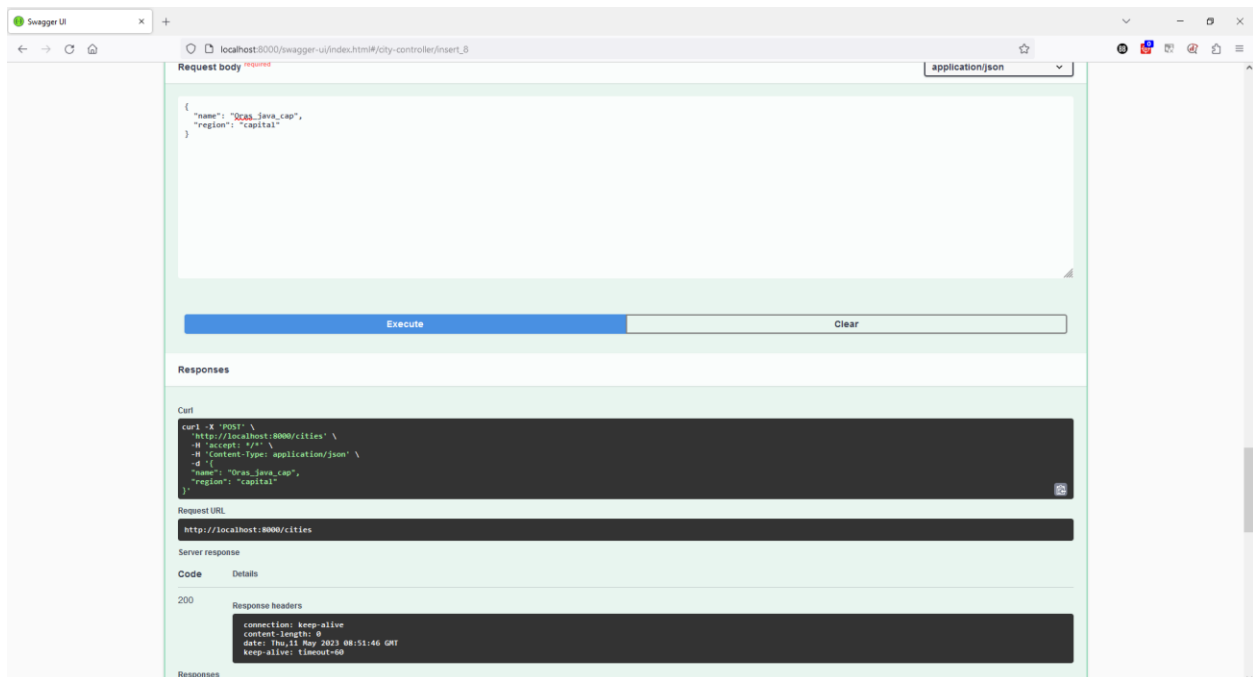
Stergem obiectul creat.

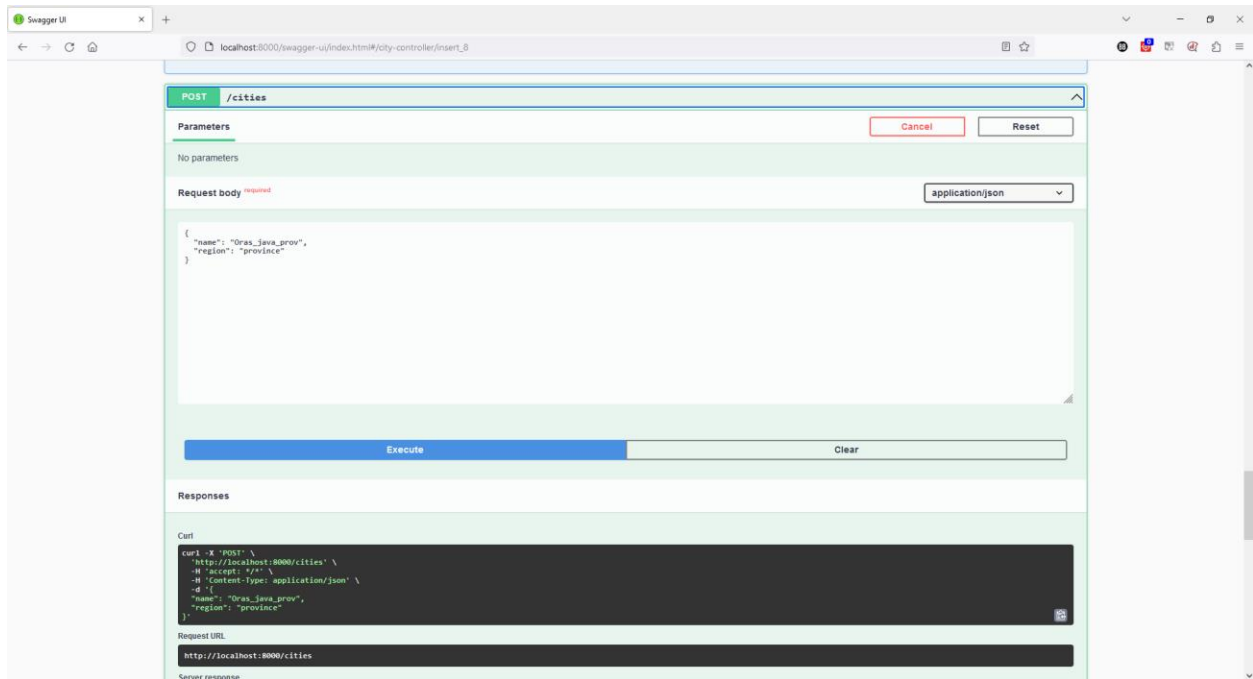


## 5.2. Cities

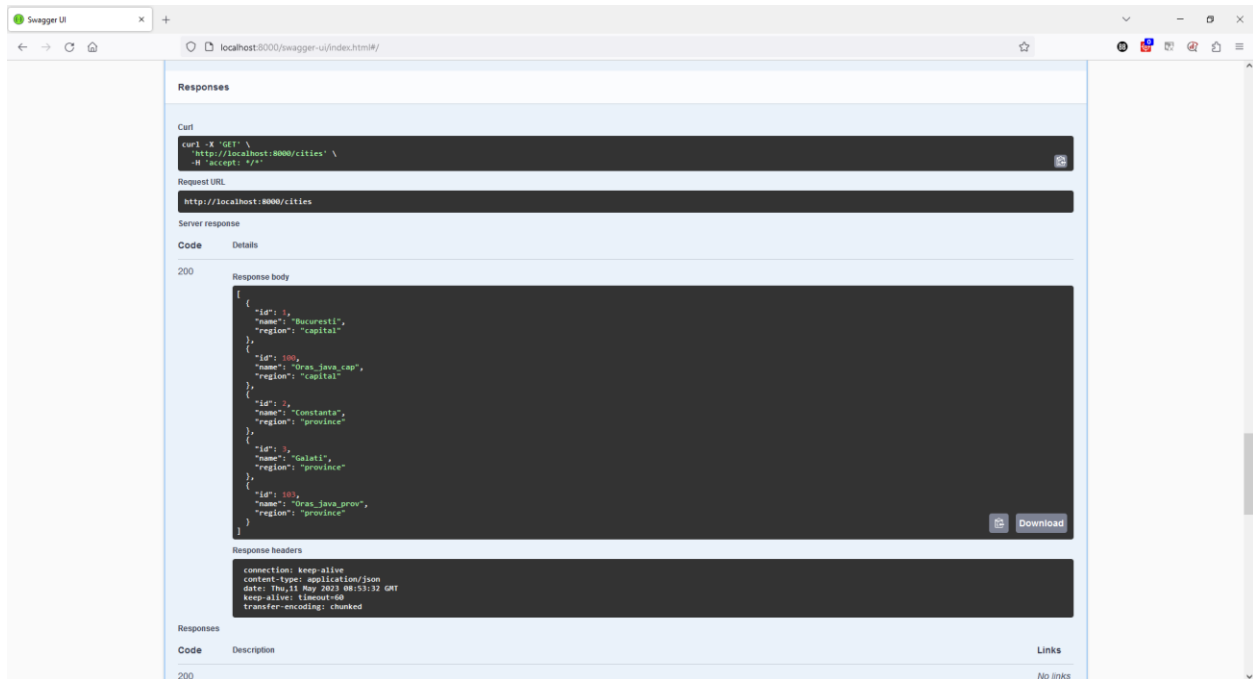
Tabela cities este fragmentată orizontal primar. Astfel, orașele capitale vor fi stocate în pdb1, iar cele din provincie în pdb2.

Inserăm două orașe pentru a testa conexiunea la ambele baze.





Verificăm că cele două orașe au fost inserate. Observăm că mai întâi sunt afișate orașele din pdb1, iar apoi cele din pdb2.



Actualizăm cele 2 orașe.

Swagger UI

localhost:8000/swagger-ui/index.html#/city-controller/update\_5

**Parameters**

Name	Description
<b>id</b> *required integer(int32) (path)	100
<b>name</b> *required string (query)	Oras_java_cap_2

**Execute** **Clear**

**Responses**

**Curl**

```
curl -X 'PUT' \
  'http://localhost:8000/cities/100?name=Oras_java_cap_2' \
  -H 'accept: */*'
```

**Request URL**

```
http://localhost:8000/cities/100?name=Oras_java_cap_2
```

**Server response**

Code	Details
200	<b>Response headers</b> <pre>connection: keep-alive content-length: 0 date: Thu, 11 May 2023 08:55:12 GMT keep-alive: timeout=60</pre>

**Responses**

Code	Description	Links
200	OK	No links

Swagger UI

localhost:8000/swagger-ui/index.html#/city-controller/update\_5

**Parameters**

Name	Description
<b>id</b> *required integer(int32) (path)	103
<b>name</b> *required string (query)	Oras_java_prov_2

**Execute** **Clear**

**Responses**

**Curl**

```
curl -X 'PUT' \
  'http://localhost:8000/cities/103?name=Oras_java_prov_2' \
  -H 'accept: */*'
```

**Request URL**

```
http://localhost:8000/cities/103?name=Oras_java_prov_2
```

**Server response**

Code	Details
200	<b>Response headers</b> <pre>connection: keep-alive content-length: 0 date: Thu, 11 May 2023 08:55:50 GMT keep-alive: timeout=60</pre>

**Responses**

Code	Description	Links
200	OK	No links

Verificăm că s-au realizat cele două update-uri.

Swagger UI interface showing a successful GET request to the endpoint `/cities/100`. The request is executed, and the response is displayed.

**Name:** `id` (required)  
**Description:** `integer($int32)` (path)

**Execute** **Clear**

**Responses**

**curl**  
`curl -X 'GET' \`  
`'http://localhost:8000/cities/100' \`  
`-H 'accept: */*'`

**Request URL**  
`http://localhost:8000/cities/100`

**Server response**

**Code** **Details**

**200** **Response body**  
`{`  
 `"id": 100,`  
 `"name": "Oras_Java_cap_2",`  
 `"region": "capital"`  
`}`

**Response headers**  
`connection: keep-alive`  
`content-type: application/json`  
`date: Thu, 11 May 2023 08:56:35 GMT`  
`keep-alive: timeout=60`  
`transfer-encoding: chunked`

**Responses**

Code	Description	Links
200	OK	No links

**Media type**

Swagger UI interface showing a successful GET request to the endpoint `/cities/103`. The request is executed, and the response is displayed.

**Name:** `id` (required)  
**Description:** `integer($int32)` (path)

**Execute** **Clear**

**Responses**

**curl**  
`curl -X 'GET' \`  
`'http://localhost:8000/cities/103' \`  
`-H 'accept: */*'`

**Request URL**  
`http://localhost:8000/cities/103`

**Server response**

**Code** **Details**

**200** **Response body**  
`{`  
 `"id": 103,`  
 `"name": "Oras_Java_prov_2",`  
 `"region": "province"`  
`}`

**Response headers**  
`connection: keep-alive`  
`content-type: application/json`  
`date: Thu, 11 May 2023 08:56:47 GMT`  
`keep-alive: timeout=60`  
`transfer-encoding: chunked`

**Responses**

Code	Description	Links
200	OK	No links

**Media type**

La final ștergem cel două orașe.

The image displays two sequential screenshots of the Swagger UI interface, demonstrating the execution of a DELETE API endpoint. Both screenshots show the endpoint **DELETE /cities/{id}** with a parameter **id** of type **integer(\$int32)** located at the path.

**Top Screenshot (ID 100):**

- The **id** parameter is set to **100**.
- The **Execute** button is highlighted in blue.
- The **Responses** section shows a **200 OK** status.
- The **Server response** section displays the following headers:

```
connection: keep-alive
content-length: 0
date: Thu, 11 May 2023 08:57:35 GMT
keep-alive: timeout=60
```

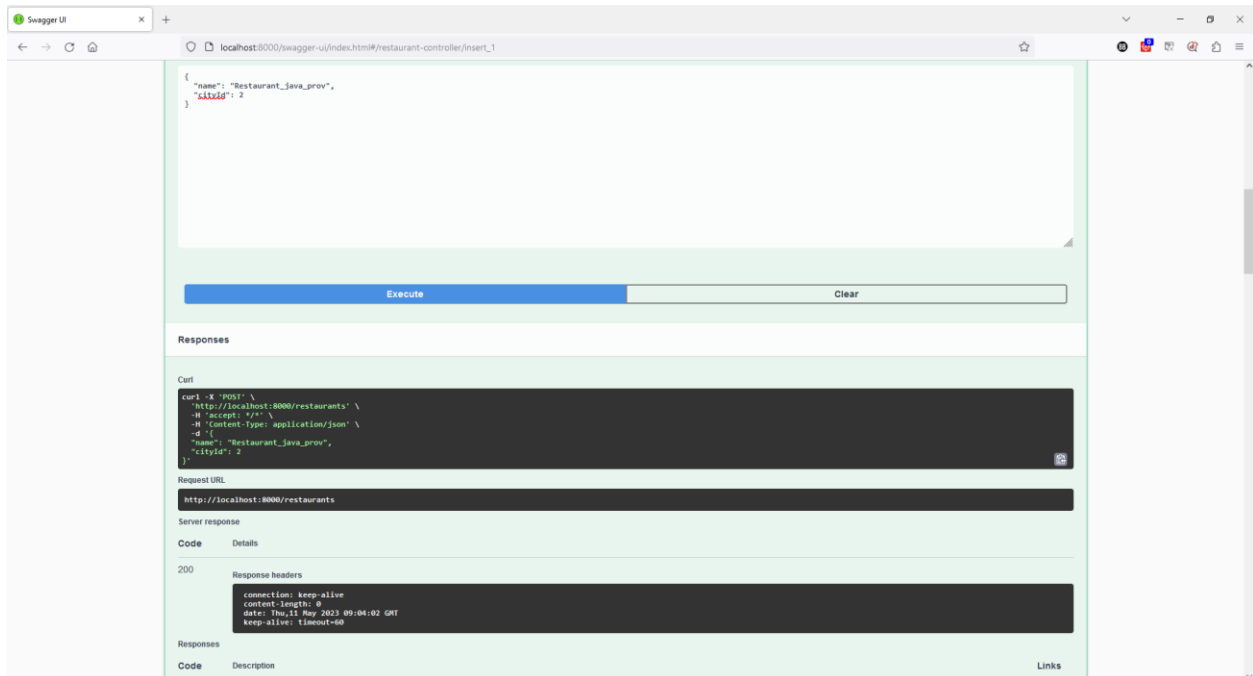
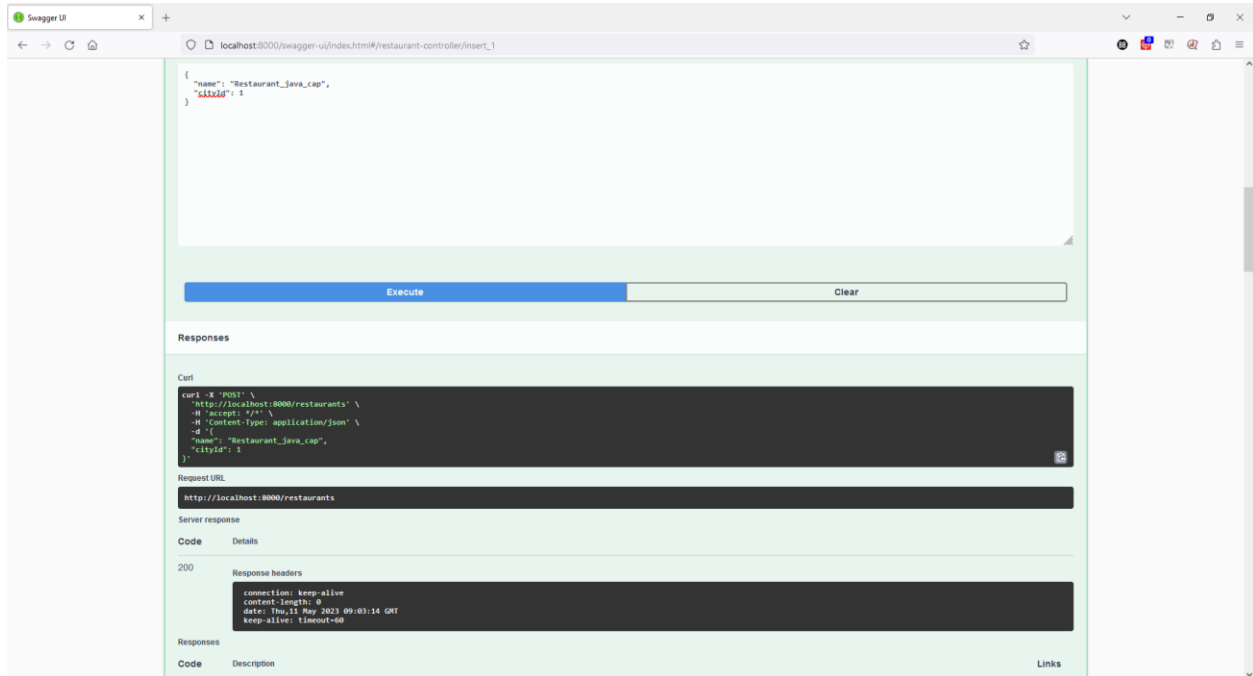
**Bottom Screenshot (ID 103):**

- The **id** parameter is set to **103**.
- The **Execute** button is highlighted in blue.
- The **Responses** section shows a **200 OK** status.
- The **Server response** section displays the following headers:

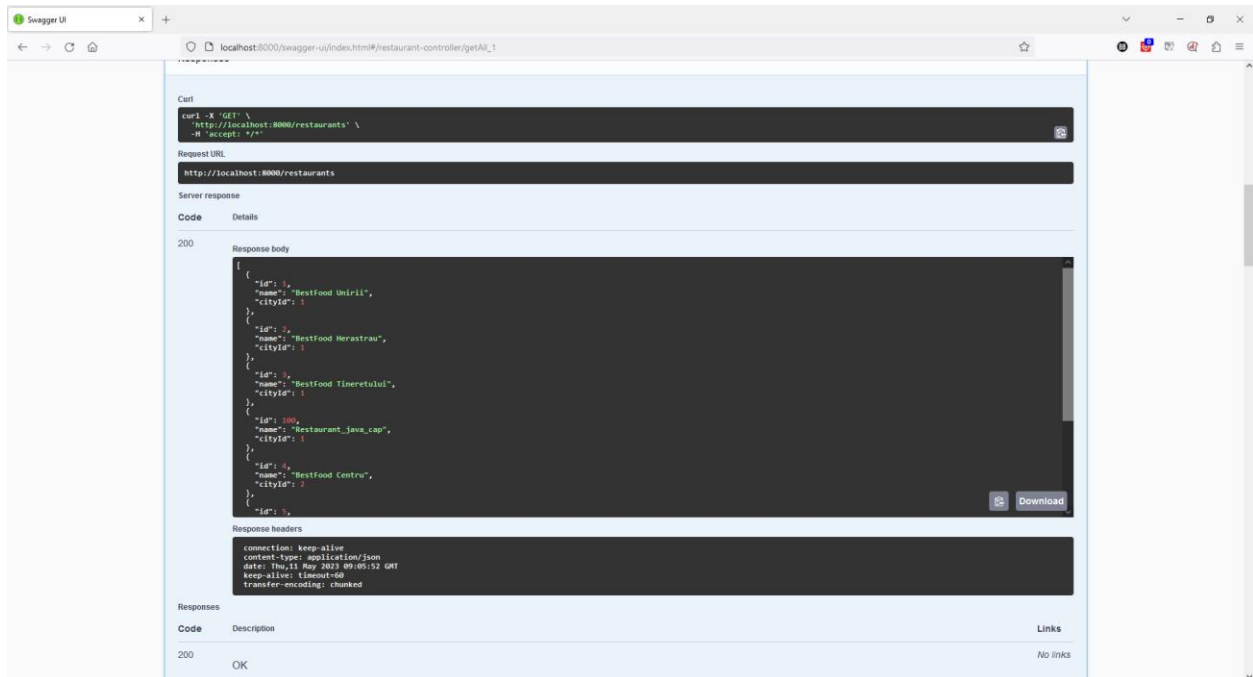
```
connection: keep-alive
content-length: 0
date: Thu, 11 May 2023 08:57:52 GMT
keep-alive: timeout=60
```

### 5.3. Restaurants

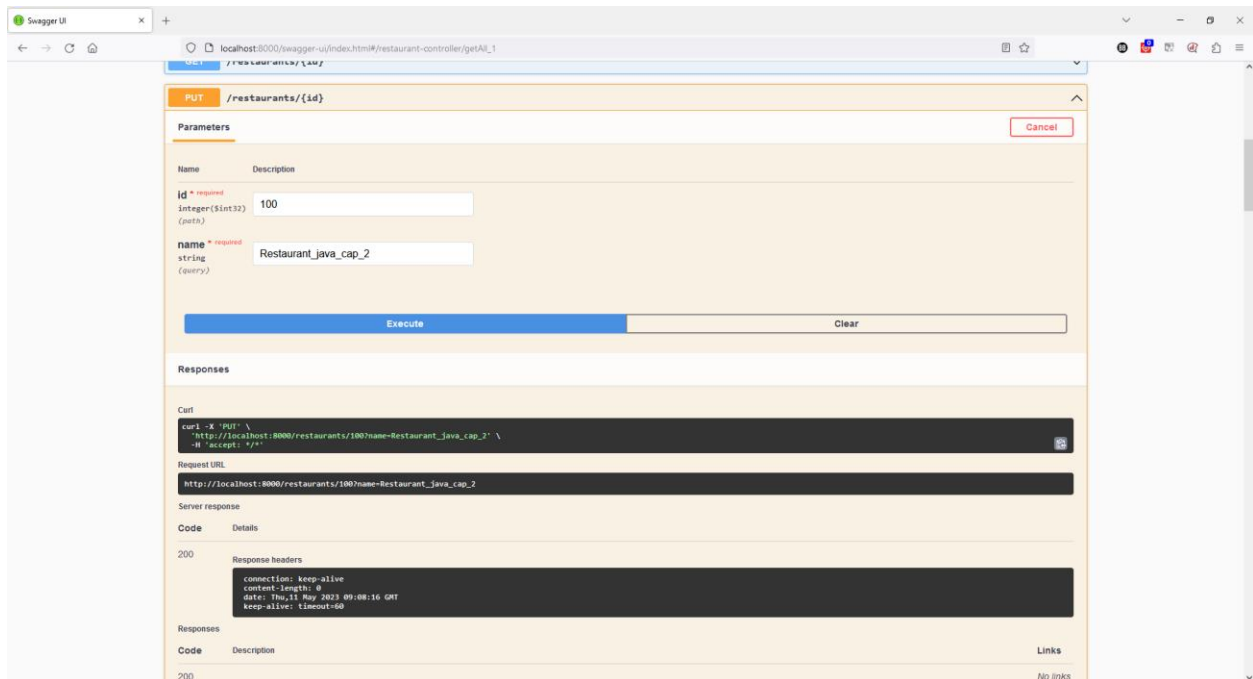
Tabela restaurants este prima tabelă fragmentată orizontal derivat în proiectul nostru. Astfel că vom insera câte 2 obiecte pentru fiecare bd.



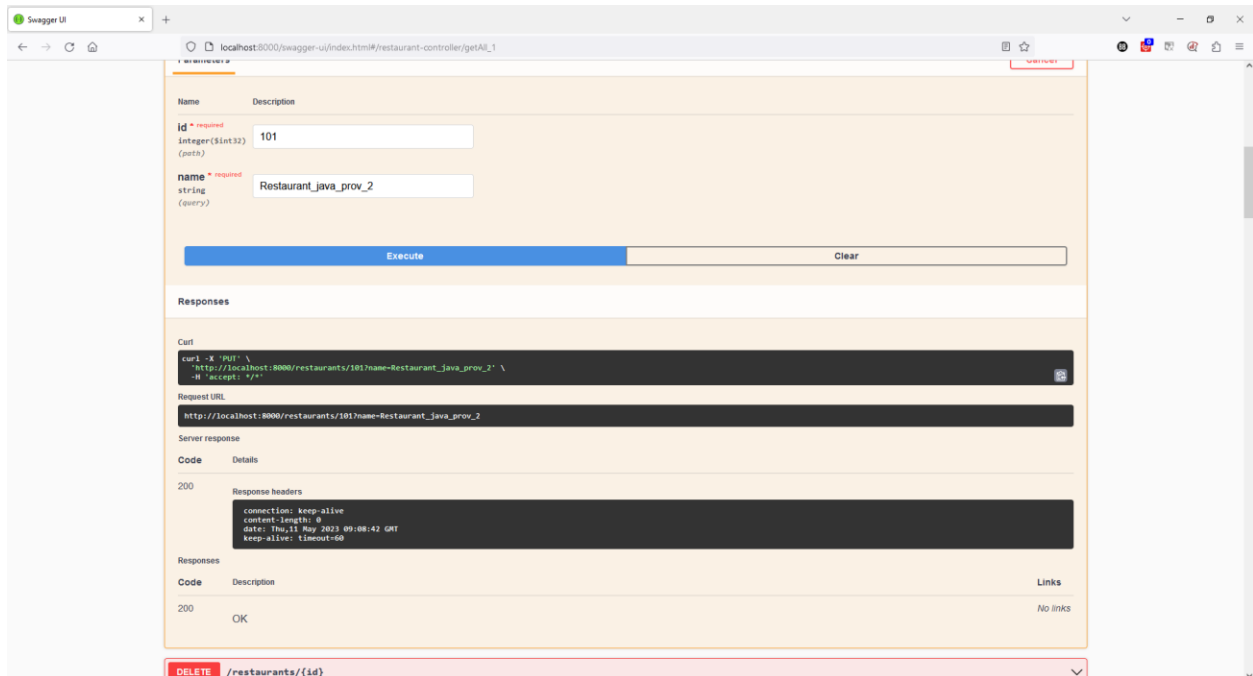
Verificăm că cele două restaurante au fost inserate.



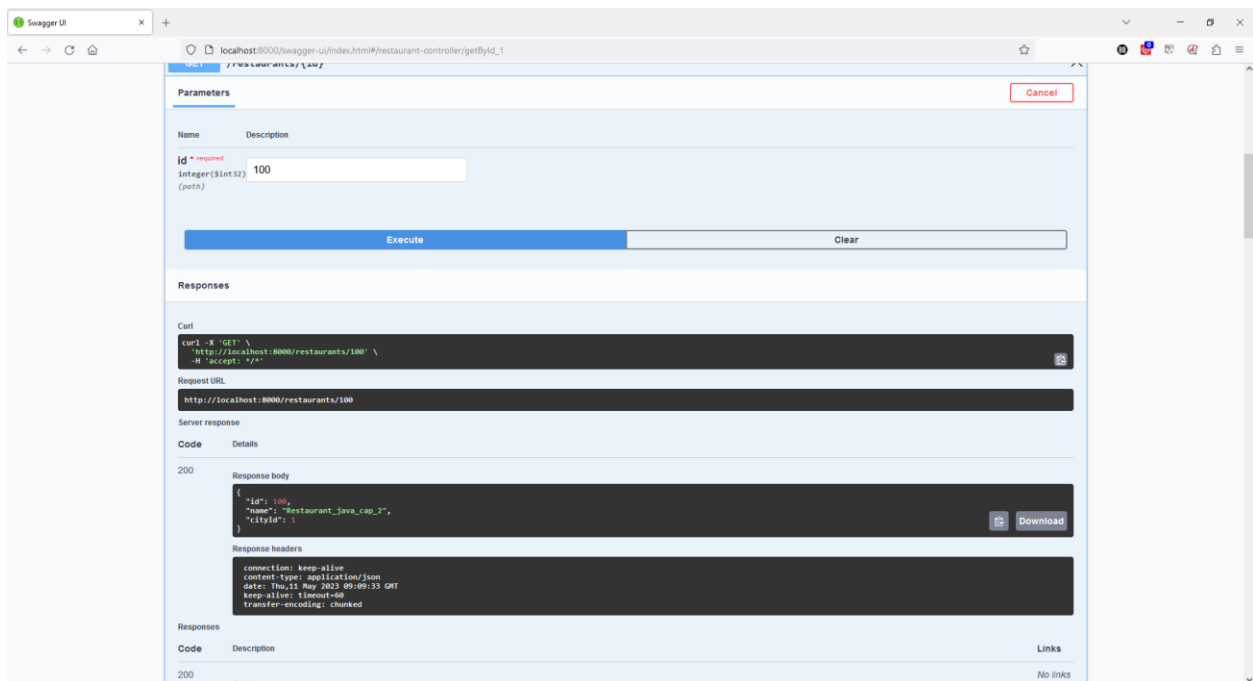
Actualizăm cele două restaurante.

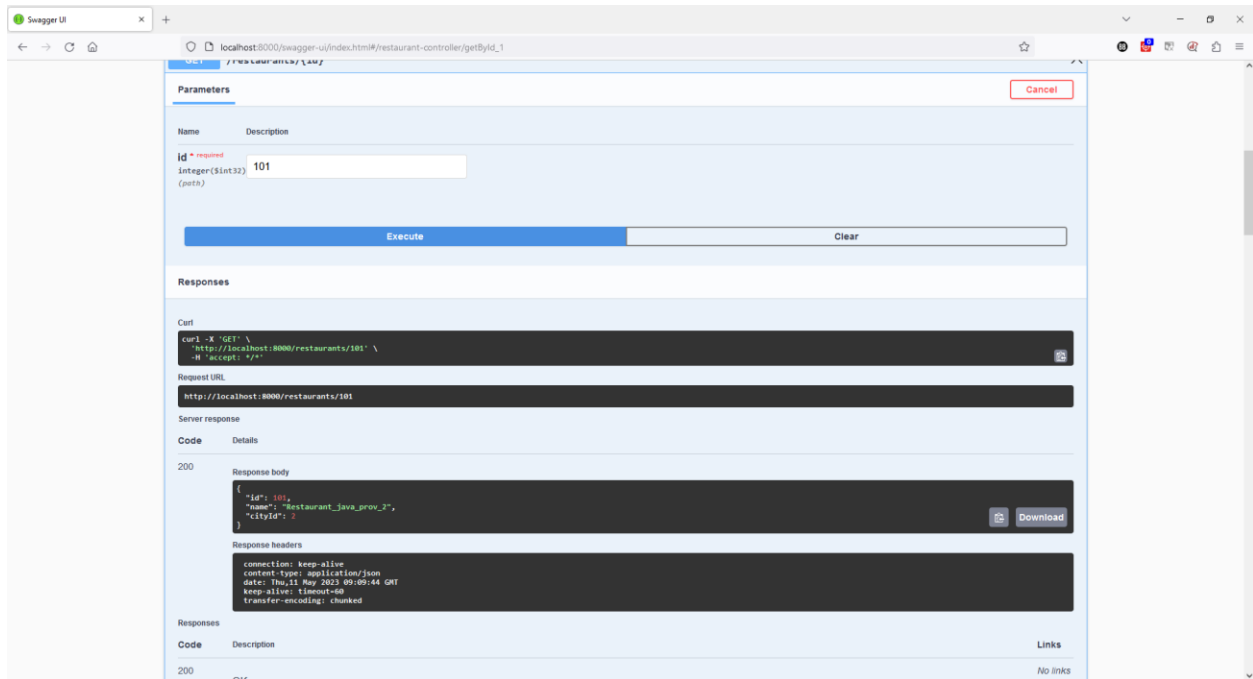




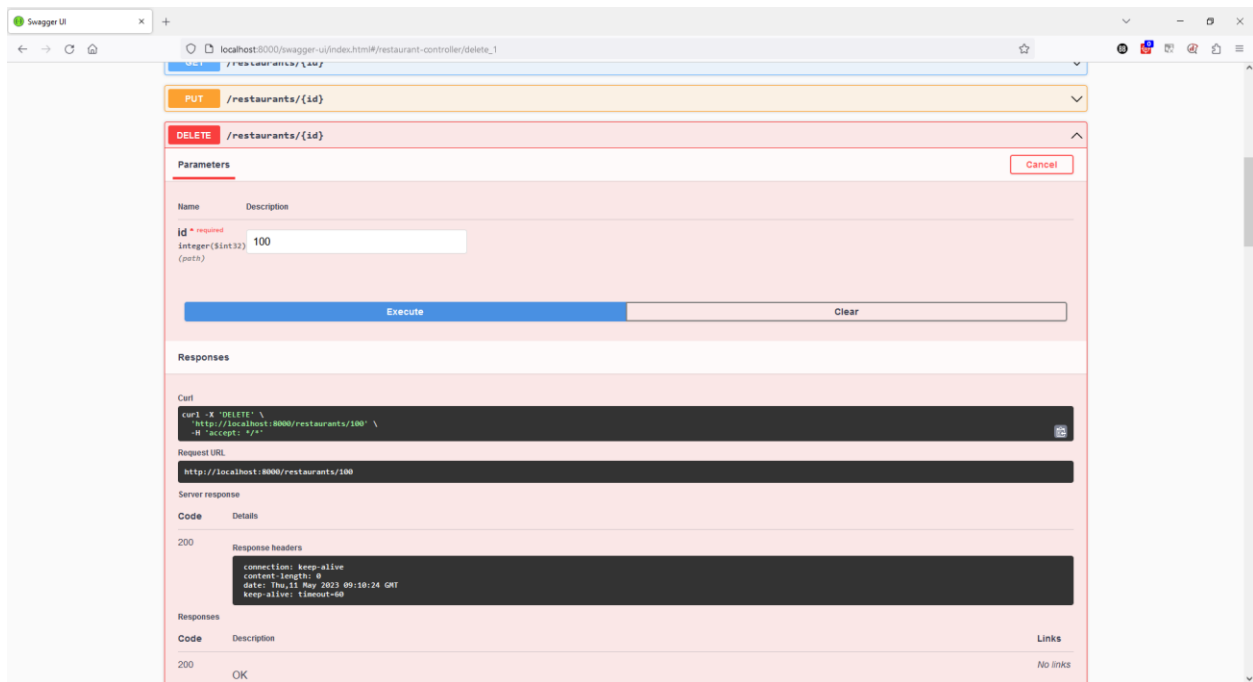


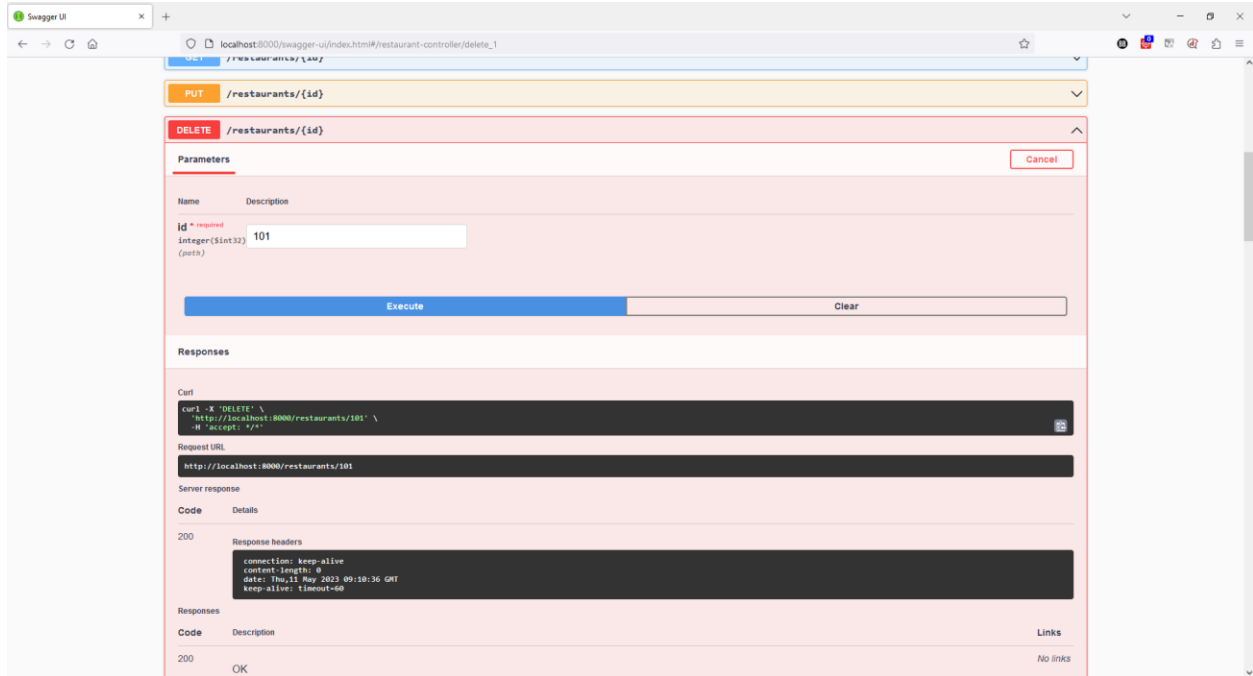
Verificăm că au fost actualizate.





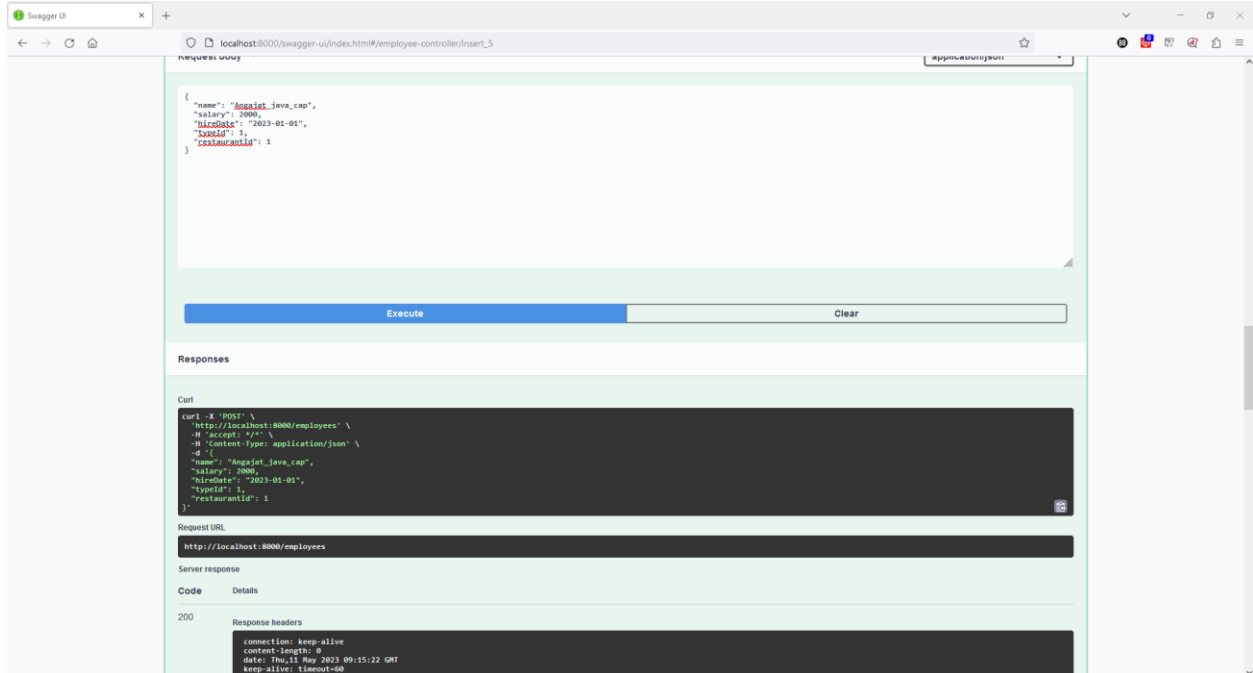
Ștergem cele două orașe.

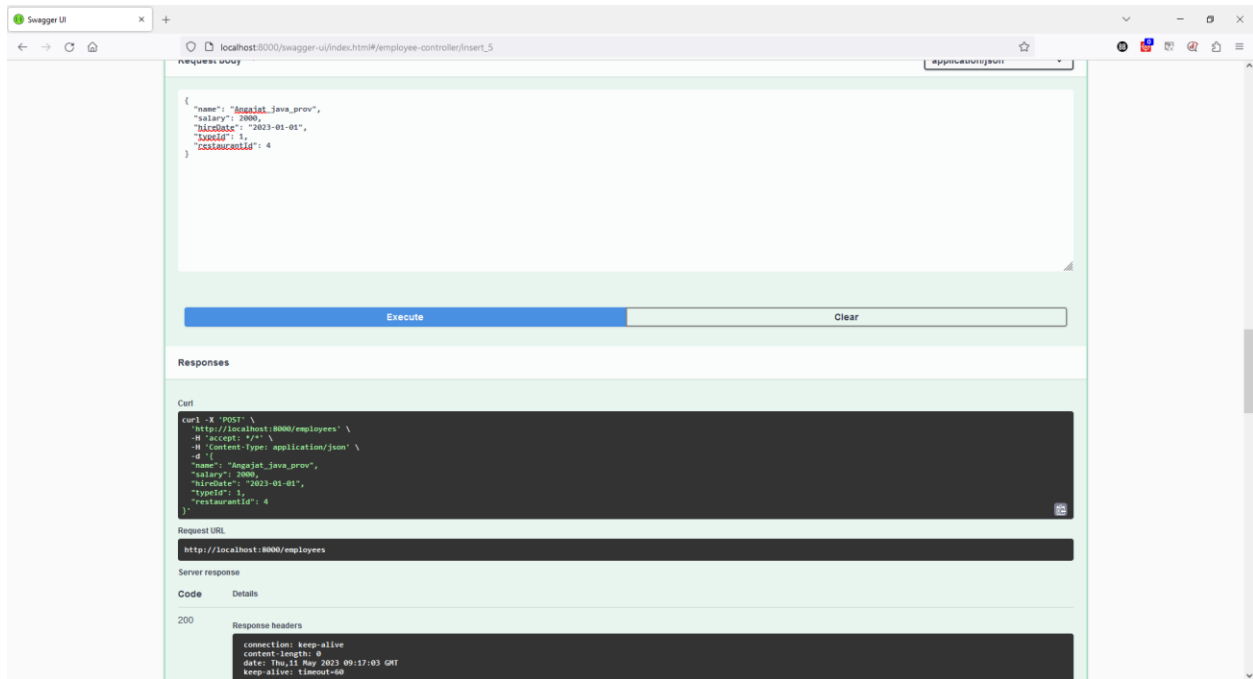




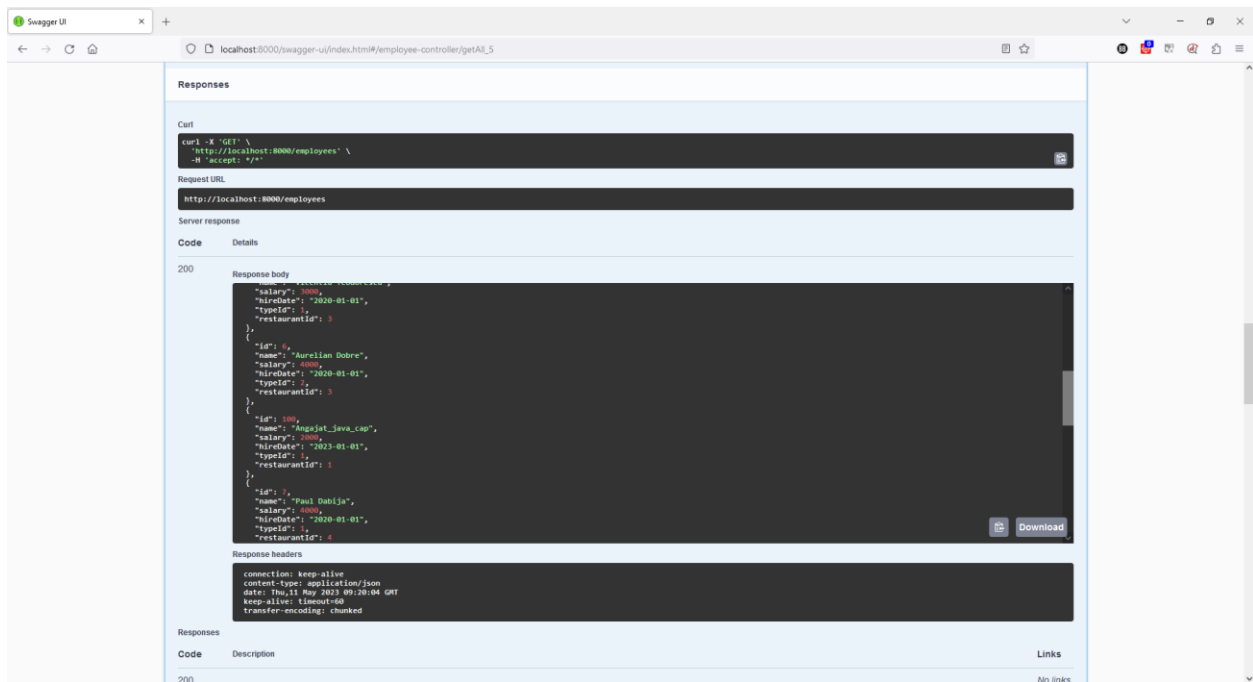
## 5.4. Employees

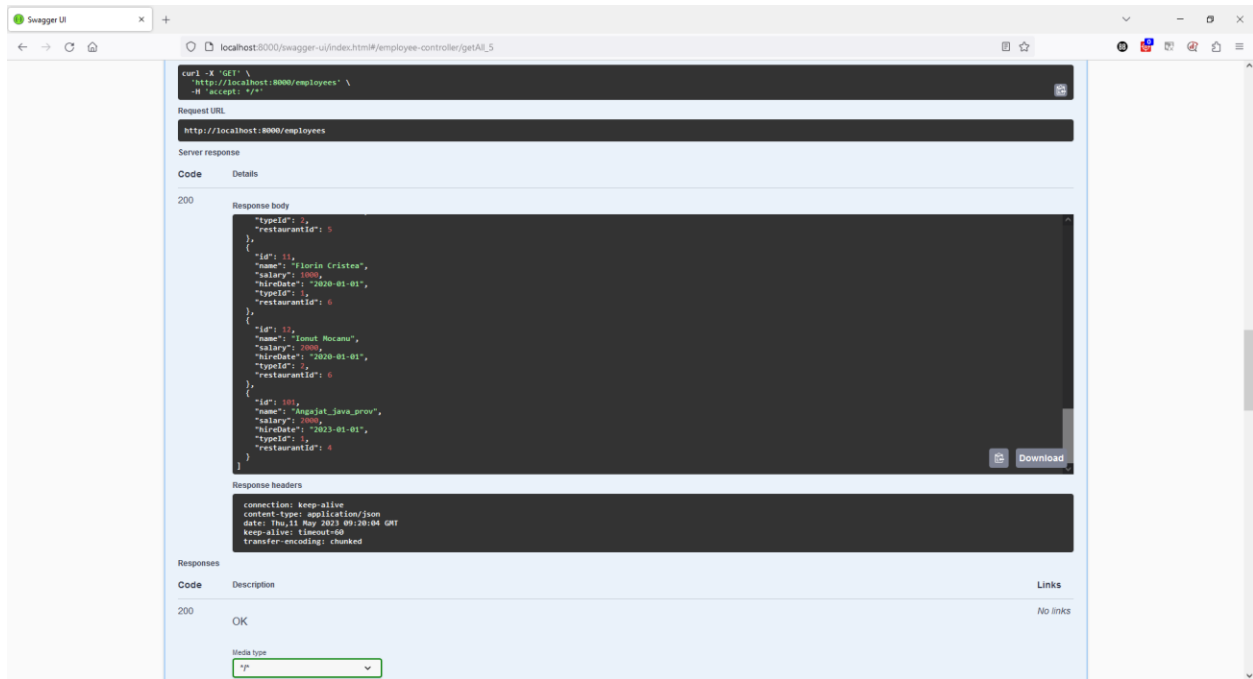
Fiind tot o tabelă fragmentată derivat, vom include tot câte două obiecte.



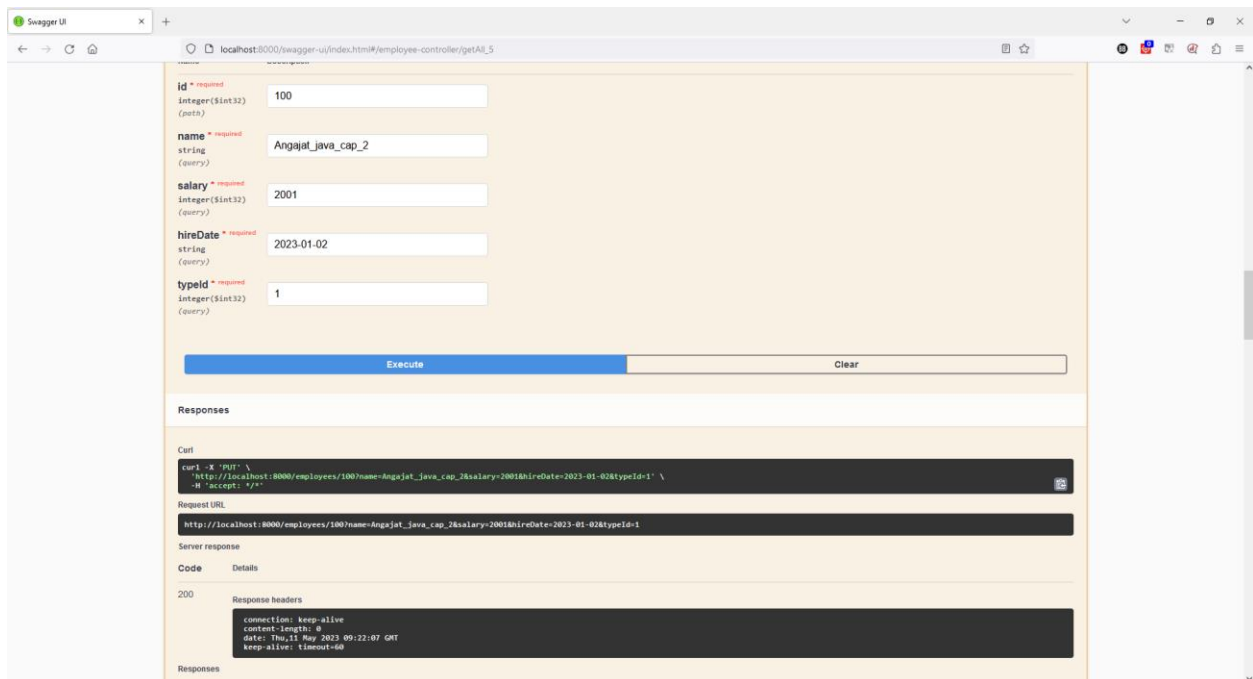


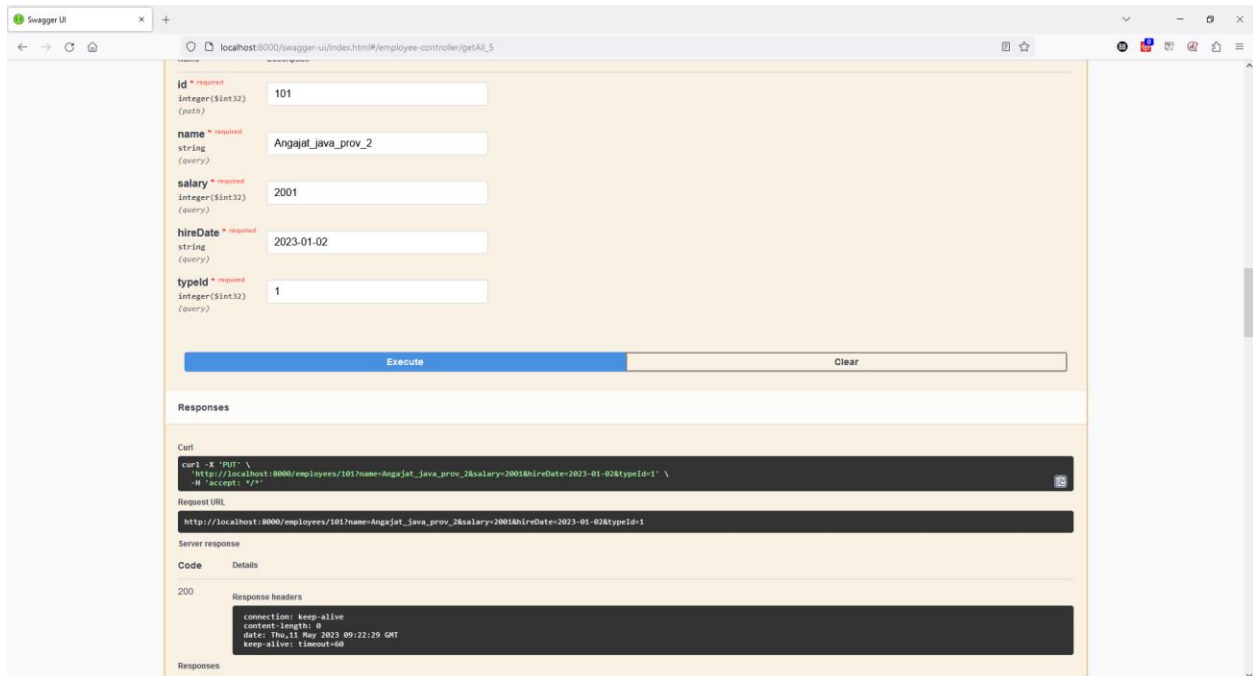
Verificăm că au fost inserați cei doi angajați.



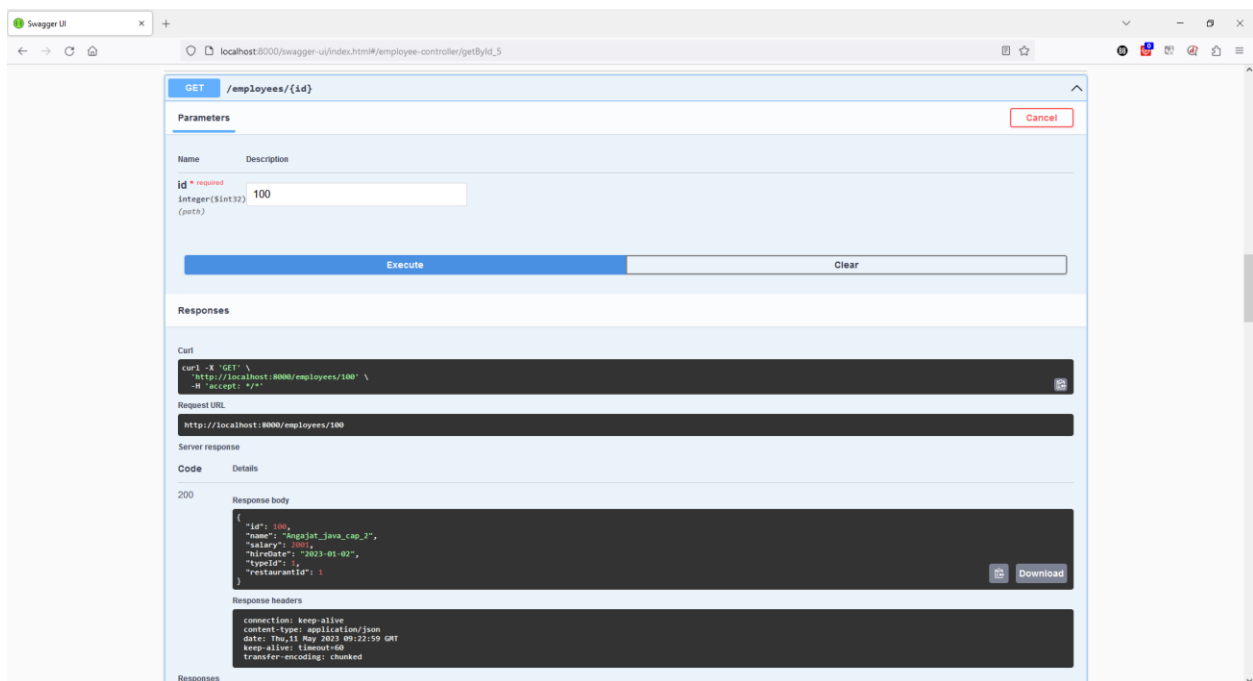


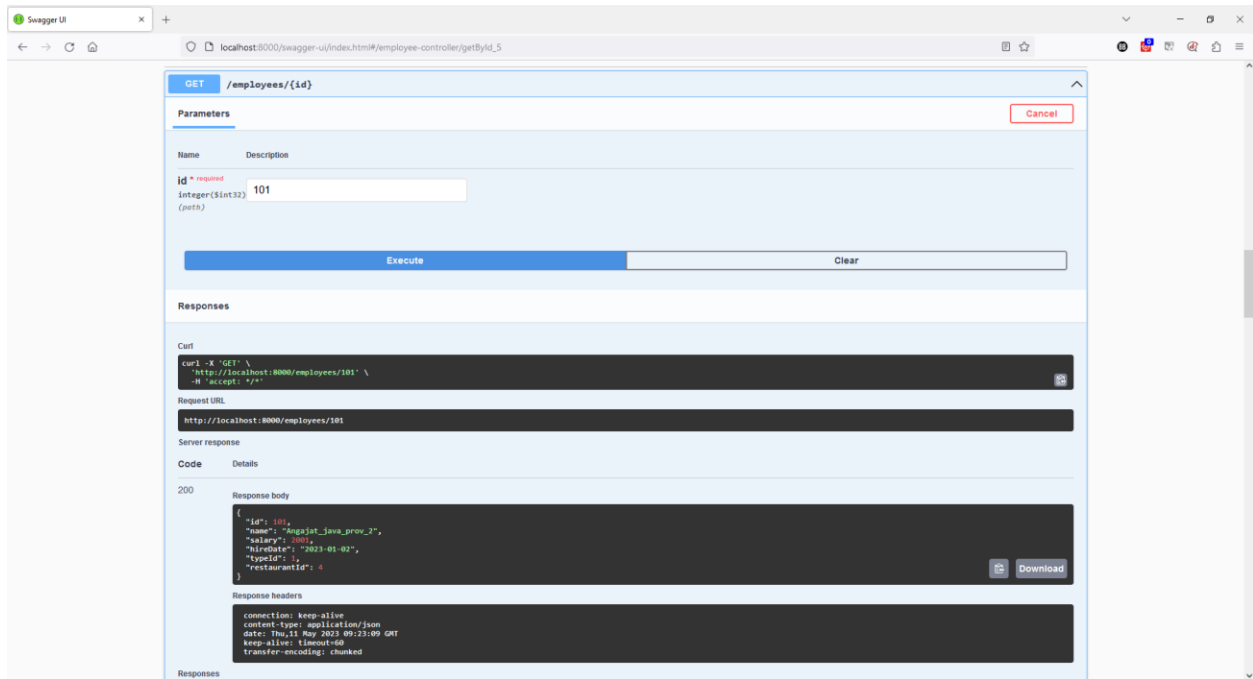
Actualizăm cei 2 angajați.



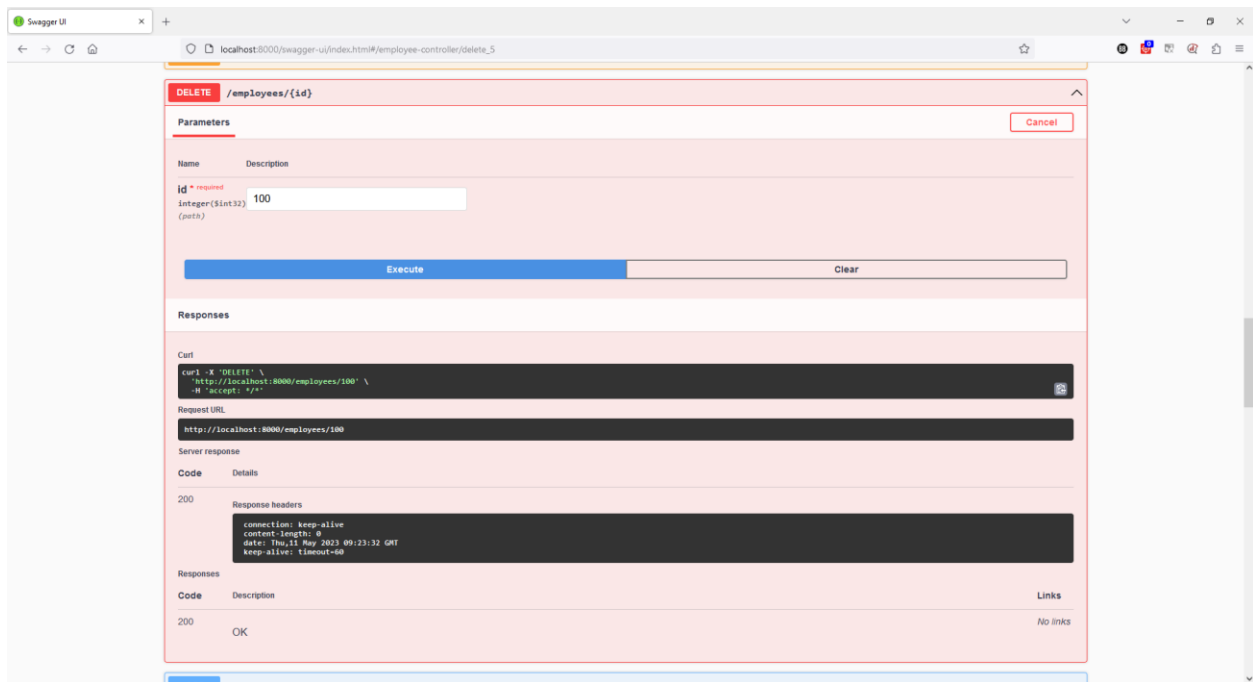


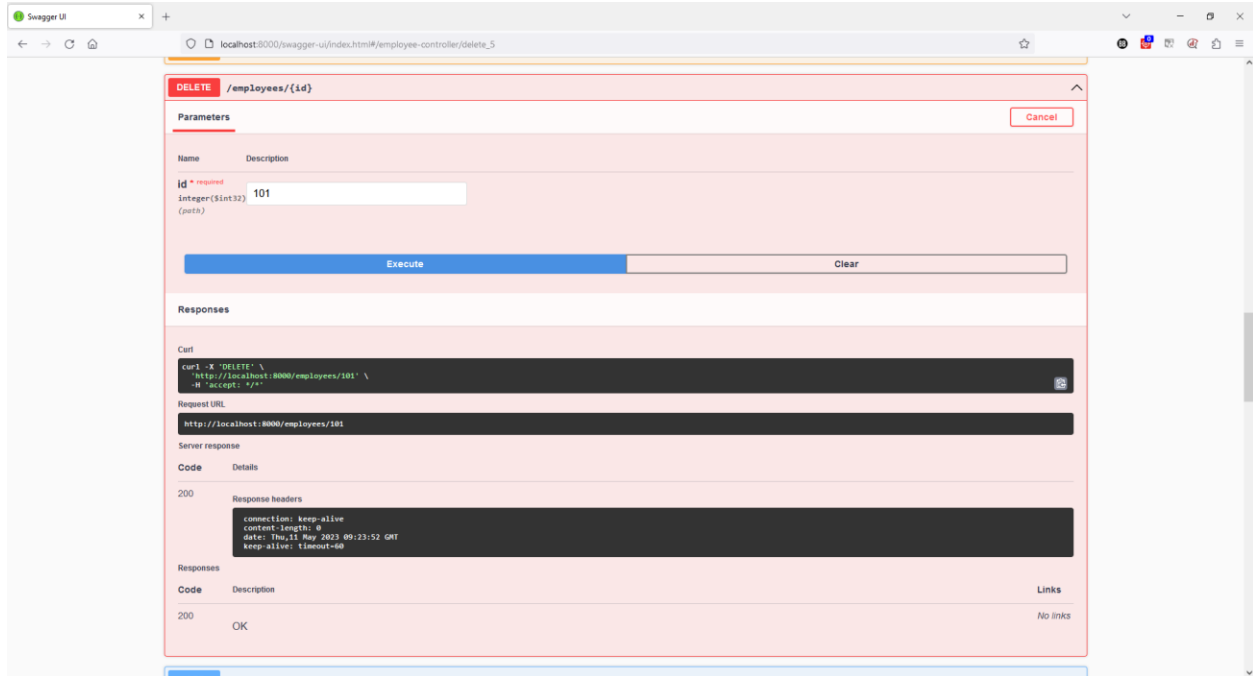
Verificăm că au fost acutalizați.





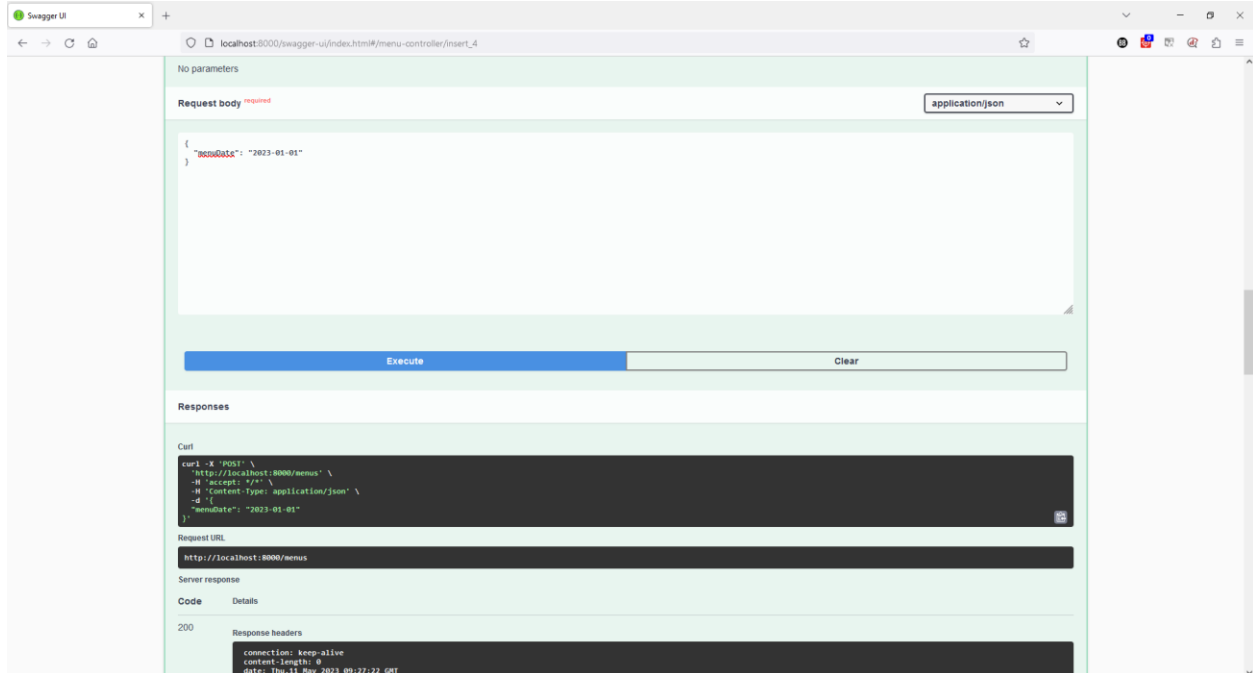
Ştergem cei 2 angajaŃi.





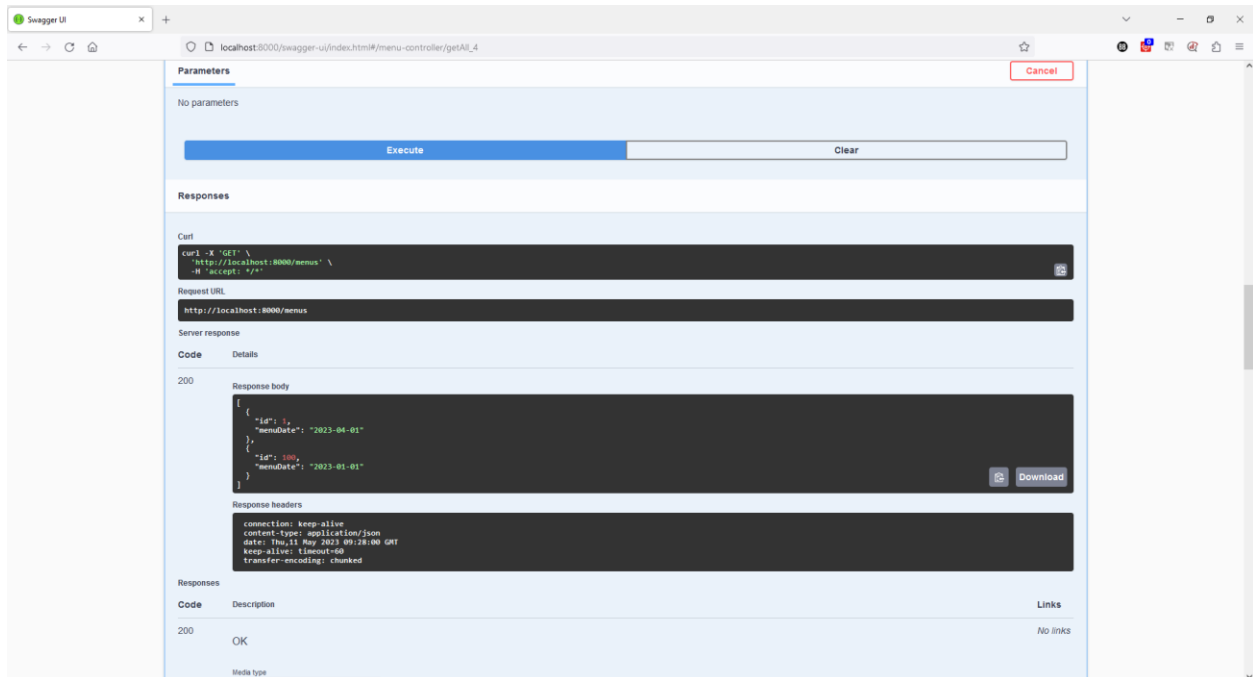
## 5.5. Menus

Tabela menus este o tabelă replicată. Astfel, vom insera un singur obiect.





Verificăm că a fost inserat.

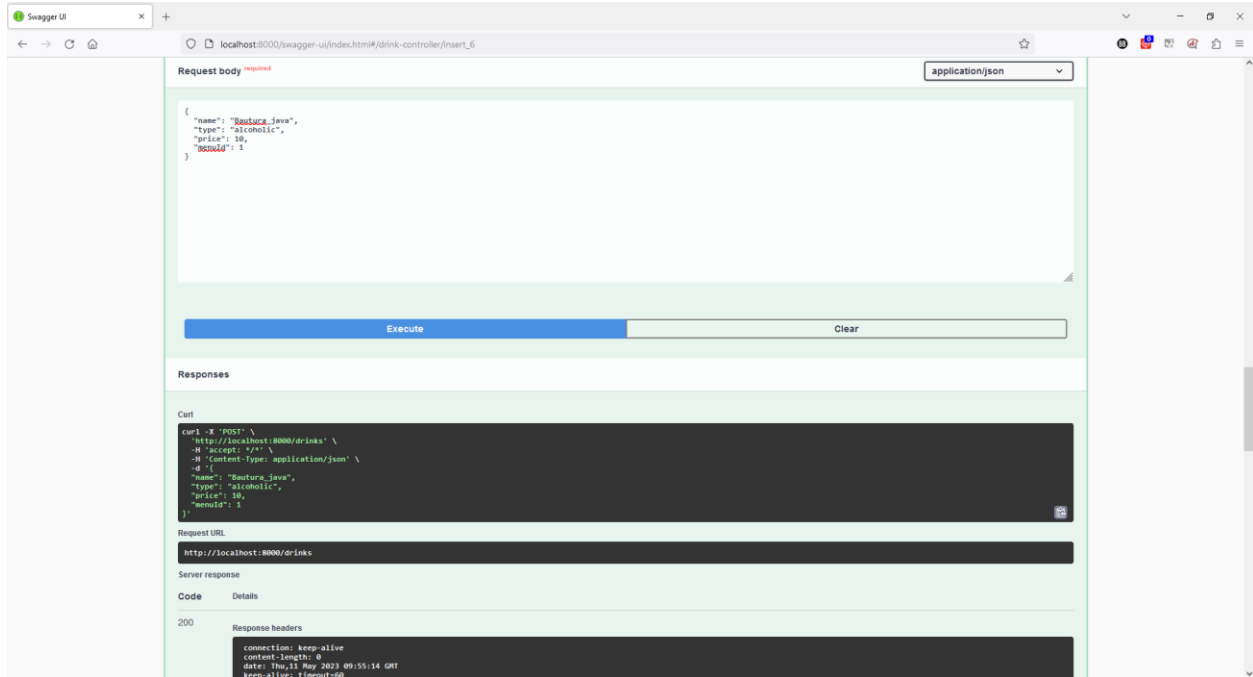


Pentru a demonstra că datele au fost replicate în ambele baze, am decis să facem câte un select la nivel local.

Worksheet	Query Builder	Worksheet	Query Builder
1	<code>SELECT * FROM user_modbd.menus_cap;</code>	1	<code>SELECT * FROM user_modbd.menus_prov@pdb2;</code>
Query Result x		Query Result x	
All Rows Fetched: 2 in 0.011 seconds		All Rows Fetched: 2 in 0.018 seconds	
ID	MENU_DATE	ID	MENU_DATE
1	01-APR-23	1	01-APR-23
2	100 01-JAN-23	2	100 01-JAN-23

## 5.6. Drinks

La fel ca tabela precedentă, tabela drinks este și ea replicată.

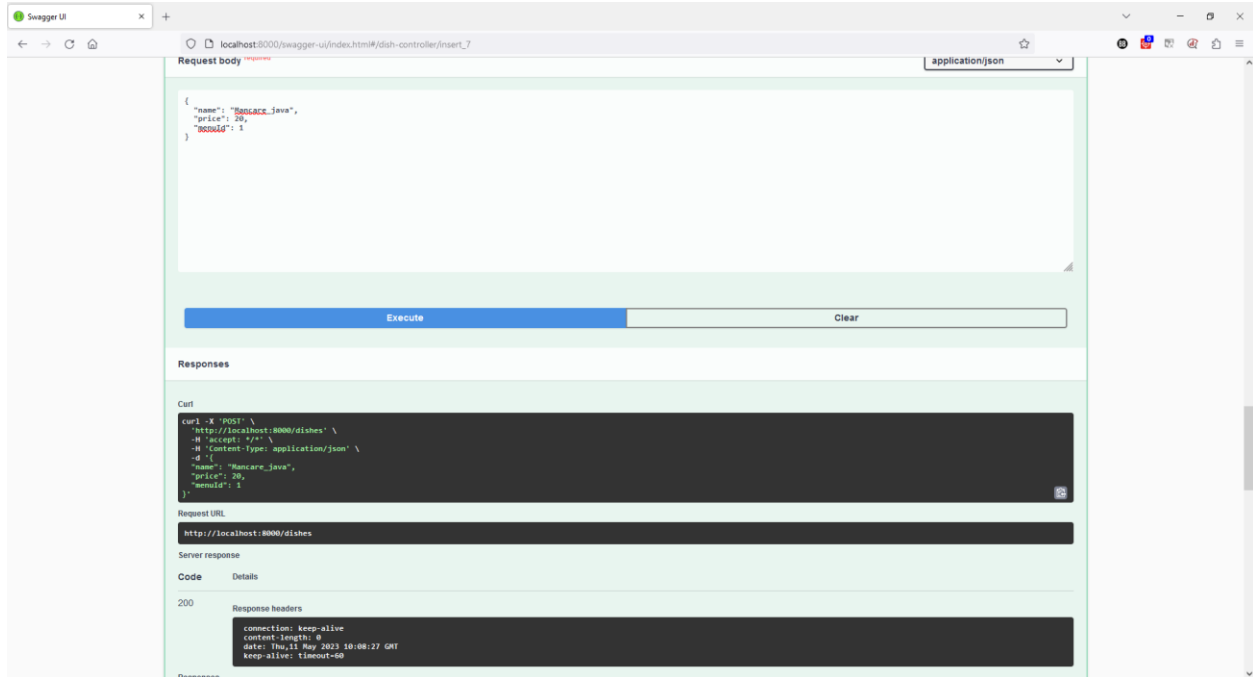


Verificăm apoi că datele au fost replicate.

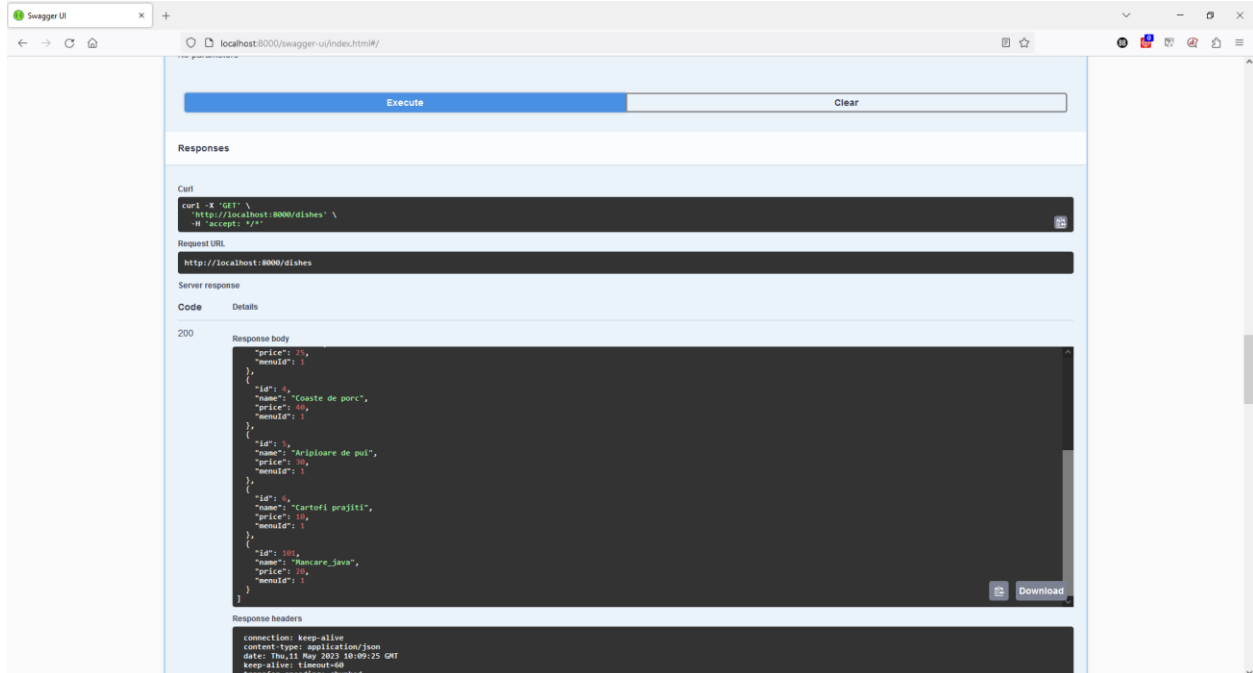
ID	NAME	TYPE	PRICE	MENU_ID
1	1 Apa	non-alcoholic	5	1
2	2 Bere	alcoholic	10	1
3	3 Vin	alcoholic	15	1
4	4 Cola	non-alcoholic	10	1
5	5 Whiskey	alcoholic	20	1
6	6 Fresh	non-alcoholic	15	1
7	100 Bautura_java	alcoholic	10	1

## 5.7. Dishes

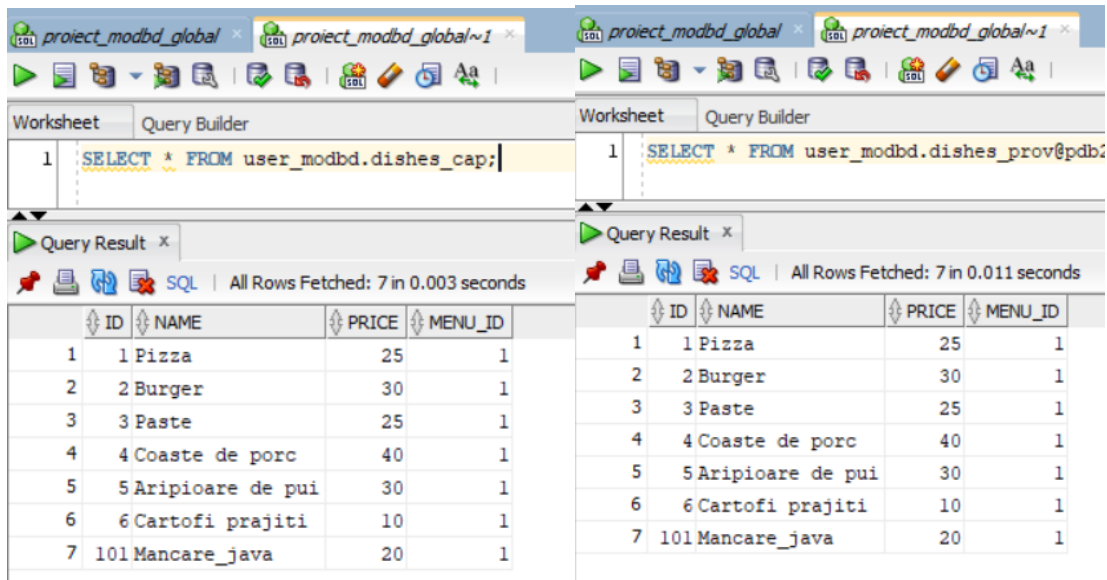
La fel, tabela dishes este replicată.



Verificăm că a fost inserat noul fel de mâncare.



Și verificăm și în baza de date.



The image shows two side-by-side screenshots of a database query tool. Both screenshots show a 'Query Builder' window with a SQL query and a 'Query Result' window displaying the results of the query.

**Left Screenshot:**

- Query: `SELECT * FROM user_modbd.dishes_cap;`
- Result: A table with 7 rows and 4 columns: ID, NAME, PRICE, MENU\_ID.

ID	NAME	PRICE	MENU_ID
1	1 Pizza	25	1
2	2 Burger	30	1
3	3 Paste	25	1
4	4 Coaste de porc	40	1
5	5 Aripioare de pui	30	1
6	6 Cartofi prajiti	10	1
7	101 Mancare_java	20	1

**Right Screenshot:**

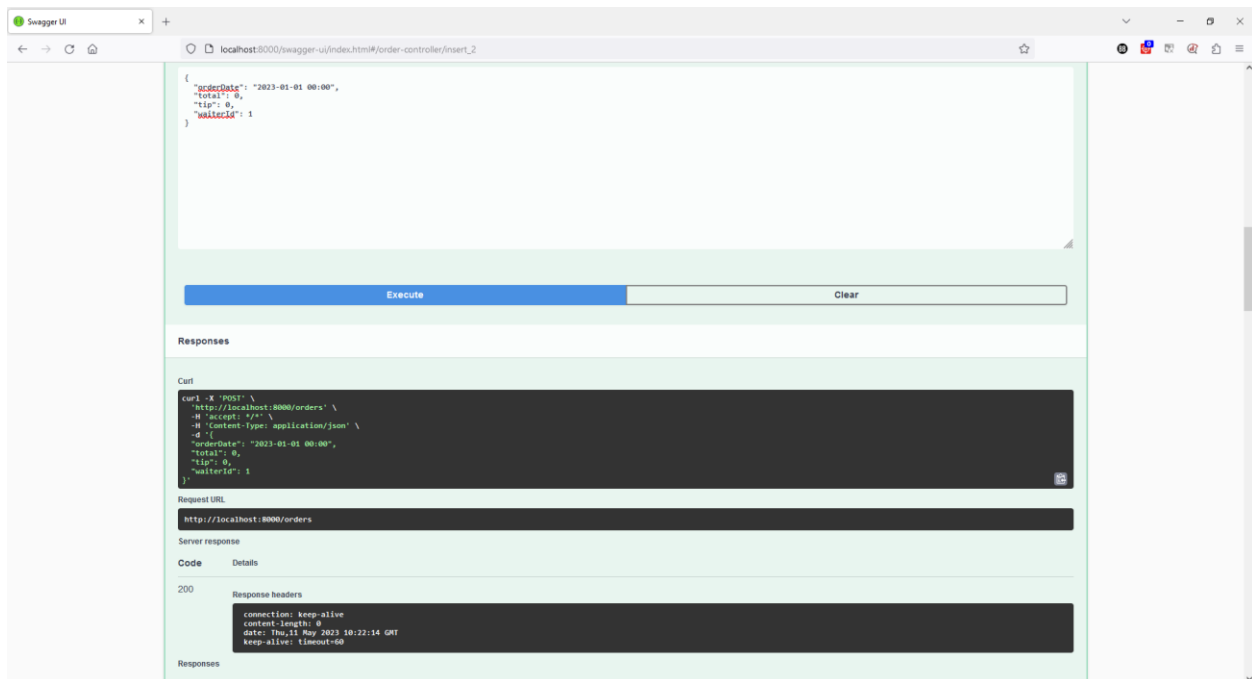
- Query: `SELECT * FROM user_modbd.dishes_prov@pdb2`
- Result: A table with 7 rows and 4 columns: ID, NAME, PRICE, MENU\_ID.

ID	NAME	PRICE	MENU_ID
1	1 Pizza	25	1
2	2 Burger	30	1
3	3 Paste	25	1
4	4 Coaste de porc	40	1
5	5 Aripioare de pui	30	1
6	6 Cartofi prajiti	10	1
7	101 Mancare_java	20	1

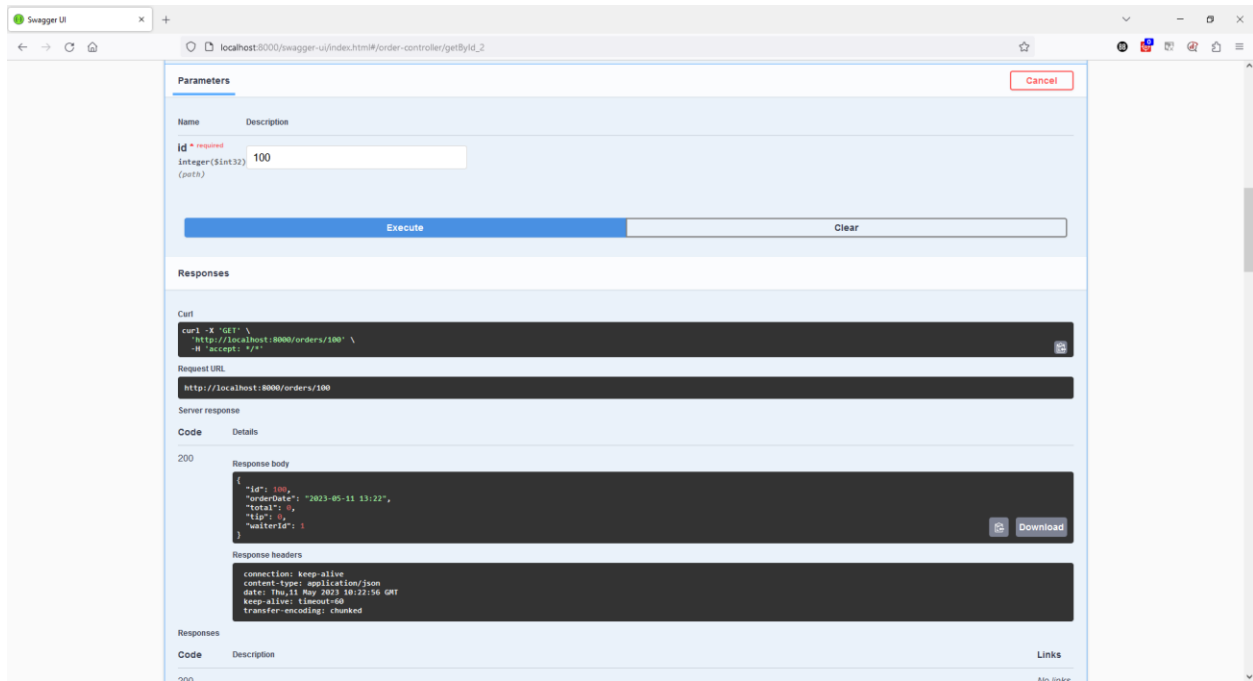
## 5.8. Orders, Orders\_drinks, Chefs\_orders\_dishes

Am decis să testăm aceste 3 tabele împreună, deoarece prețurile din Orders\_drinks și Chef\_orders\_dishes sunt calculate în funcție de cantitate. Apoi totalul din Orders este calculat în funcție de prețurile din cele 2 tabele. Mai mult, order\_date din Orders este actualizat cu fiecare băutură sau fel de mâncare adăugat.

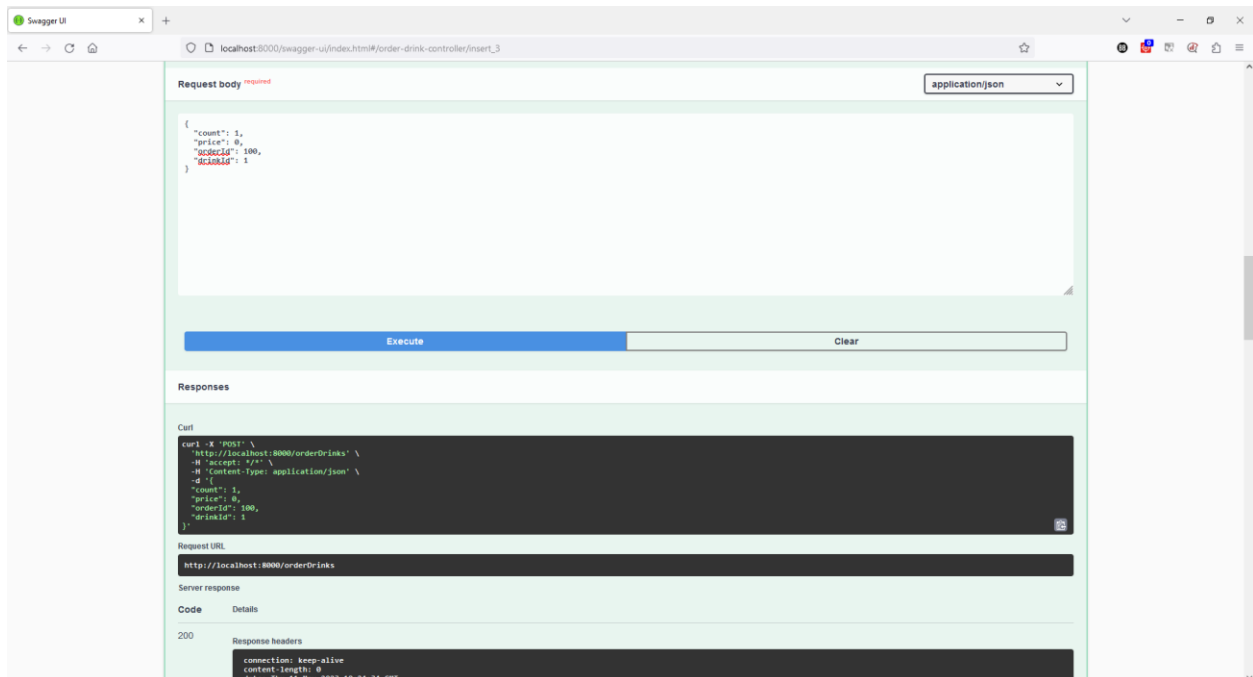
Creăm o comandă.



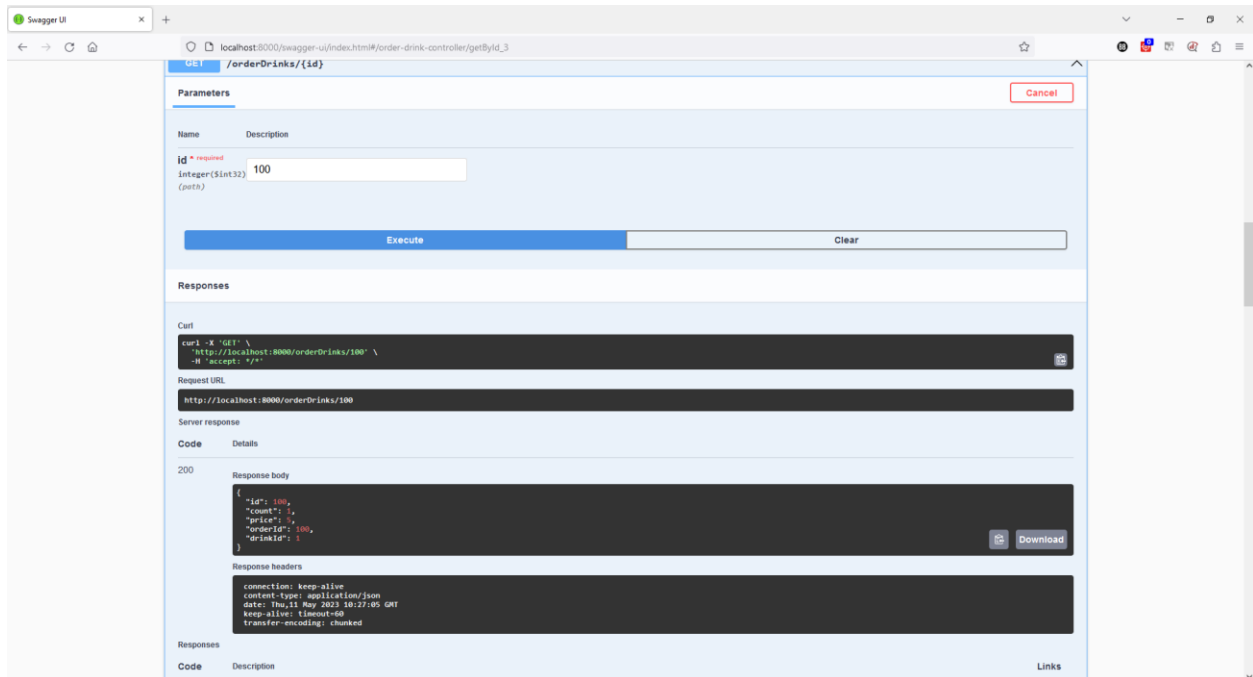
Observăm că data comenzii a fost setă automat ca fiind data curentă.



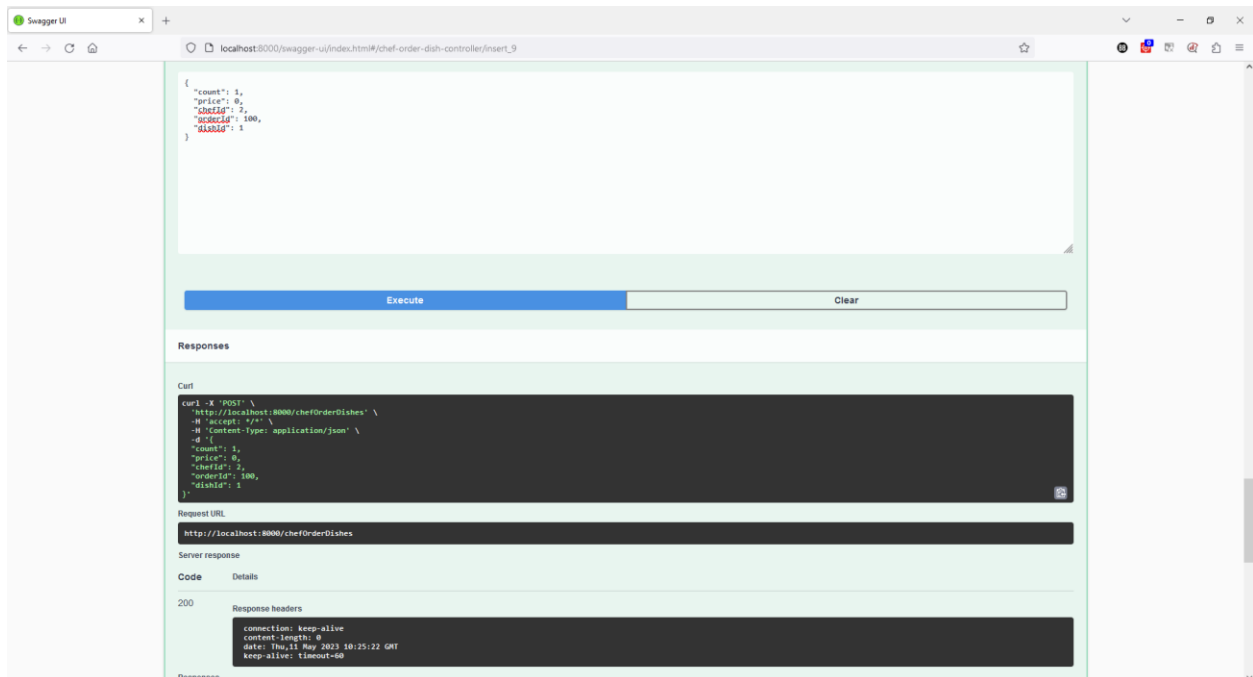
Adăugăm apoi o băutură și un fel de mâncare. Prețul din request va fi 0 și vom vedea că acesta se va actualiza automat. Vom alege băutura cu id-ul 1, al cărei preț este 5.



Vedem că a fost adăugat la comandă o bătură cu prețul 5.



Adăugăm apoi și un fel de mâncare. Prețul din request va fi 0 și vom vedea că și acesta se va actualiza automat. Vom alege dish-ul cu id-ul 1, al cărui preț este 25.



Vedem că a fost adăugat la comandă un fel de mâncare cu prețul 10.

Swagger UI interface showing the GET endpoint `/chefOrderDishes/{id}`. The parameter `id` is set to 100. The response status is 200, and the response body is a JSON object:

```
{  "id": 100,  "count": 1,  "price": 10,  "chefId": 1,  "orderId": 100,  "dishId": 1}
```

The response headers are:

```
connection: keep-alivecontent-type: application/jsondate: Thu, 11 May 2023 10:31:22 GMTkeep-alive: timeout=60transfer-encoding: chunked
```

La final, vedem că și totalul a fost actualizat în comandă și este acum egal cu 30.

Swagger UI interface showing the GET endpoint `/orders/{id}`. The parameter `id` is set to 100. The response status is 200, and the response body is a JSON object:

```
{  "id": 100,  "orderDate": "2023-05-11 13:25",  "total": 30,  "tip": 0,  "waiterId": 1}
```

The response headers are:

```
connection: keep-alivecontent-type: application/jsondate: Thu, 11 May 2023 10:33:17 GMTkeep-alive: timeout=60transfer-encoding: chunked
```