ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

МОСКОВСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет Информационных технологий
Кафедра Информатики и информационных технологий

направление подготовки
09.03.02 «Информационные системы и технологии»

ЛАБОРАТОРНАЯ РАБОТА № 11

Дисциплина: «Backend»

Тема: *Создание веб-API приложения на основе ASP.NET Core*

Выполнил: студент группы: **231-339**

Карапетян Нвер Каренович
(Фамилия И.О.)

Дата, подпись: 18.03.25 _____
(Дата) (Подпись)

Проверил: _____ _____
(Фамилия И.О., степень, звание) (Оценка)

Дата, подпись _____ _____
(Дата) (Подпись)

Москва
2025

Цель:

Освоить создание веб-API приложения на платформе ASP.NET Core для обработки HTTP-запросов и предоставления данных через API.

Задачи:

- Настроить проект ASP.NET Core для создания веб-API.
- Реализовать контроллеры для обработки GET, POST, PUT и DELETE запросов.
- Создать модели данных для работы с API (например, CRUD операции).
- Протестировать работу API приложения с использованием инструментов для отправки запросов (например, Postman).

Ход работы

В рамках данного практического занятия была разработана веб-API система для управления информацией о птицах и их средах обитания. В ходе работы были реализованы основные CRUD-операции, протестированы API-запросы, а также использованы мапперы для преобразования моделей.

Модели данных

В проекте были определены две основные модели данных, отображающие информацию о птицах (`Bird`) и их средах обитания или, другими словами, экологических нишах (`EcologicalNiche`). Эти таблицы связаны друг с другом отношением «многие ко многим» благодаря вспомогательной таблице `BirdEcologicalNiche`, которые хранит в себе уникальные идентификаторы (Id) птиц и сред обитания, а также соответствующие навигационные свойства для последующей работы с данными в контроллерах.

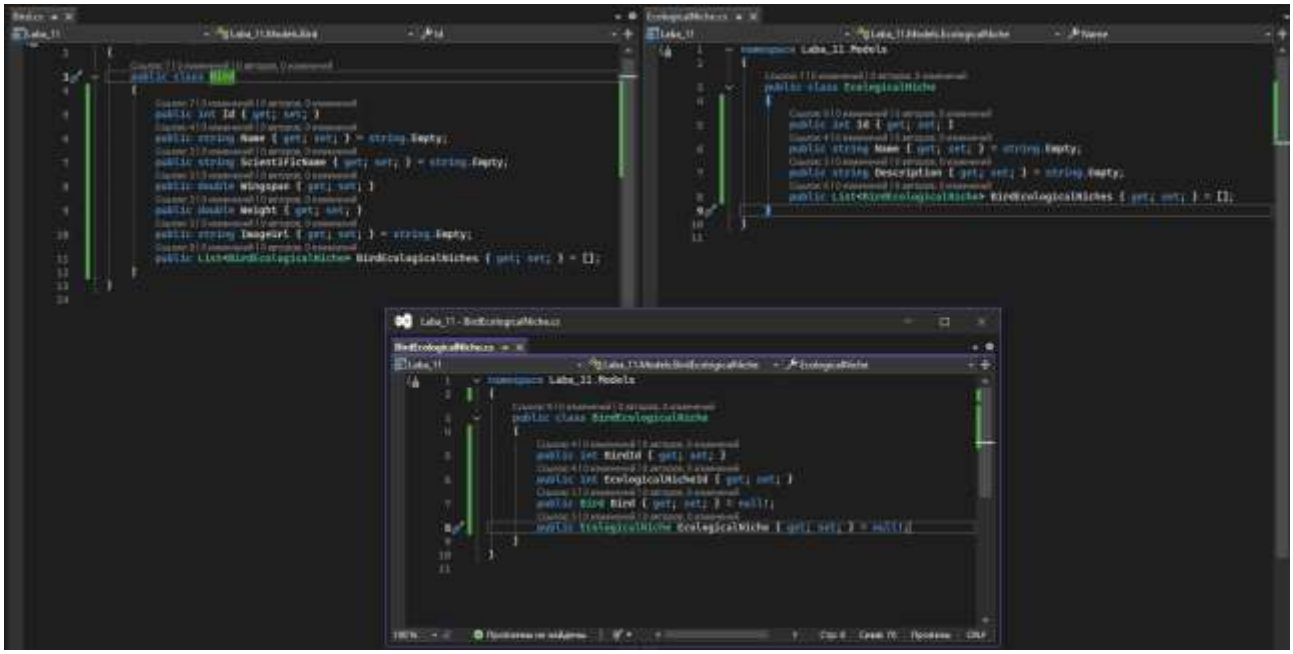


Рисунок 1. Скрипты моделей, составляющих таблицы в базе данных PostgreSQL.

Миграции

Для работы с базой данных использовался Entity Framework Core, а миграции применялись для создания и обновления схемы базы данных.

После определения моделей данных был описан контекст базы данных, в которой определим коллекции сущностей для каждой модели, которую мы разработали ранее. Также опишем в переопределенном методе `OnModelCreating` внешние ключи и в явном виде установим связь «многие ко многим» между таблицами:

```
using Laba_11.Models;
using Microsoft.EntityFrameworkCore;

namespace Laba_11.Data
{
    Ссылок: 8 | 0 изменений | 0 авторов, 0 изменений
    public class ApplicationDbContext : DbContext
    {
        Ссылок: 0 | 0 изменений | 0 авторов, 0 изменений
        public ApplicationDbContext(DbContextOptions options) : base(options) { }

        Ссылок: 6 | 0 изменений | 0 авторов, 0 изменений
        public DbSet<Bird> Birds { get; set; }
        Ссылок: 7 | 0 изменений | 0 авторов, 0 изменений
        public DbSet<EcologicalNiche> EcologicalNiches { get; set; }
        Ссылок: 1 | 0 изменений | 0 авторов, 0 изменений
        public DbSet<BirdEcologicalNiche> BirdEcologicalNiches { get; set; }

        Ссылок: 0 | 0 изменений | 0 авторов, 0 изменений
        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating(modelBuilder);

            modelBuilder.Entity<BirdEcologicalNiche>(x => x.HasKey(p => new { p.BirdId, p.EcologicalNicheId }));

            modelBuilder.Entity<BirdEcologicalNiche>()
                .HasOne(x => x.Bird)
                .WithMany(x => x.BirdEcologicalNiches)
                .HasForeignKey(x => x.BirdId);

            modelBuilder.Entity<BirdEcologicalNiche>()
                .HasOne(x => x.EcologicalNiche)
                .WithMany(x => x.BirdEcologicalNiches)
                .HasForeignKey(x => x.EcologicalNicheId);
        }
    }

}
```

Рисунок 2. Листинг скрипта ApplicationDbContext.

Следующим шагом необходимо открыть консоль диспетчера пакетов и командой «`Add-Migration`» добавляем миграцию, затем применяем миграцию («Update-Database»), после чего в pgAdmin мы можем видеть спроектированную базу данных со всеми таблицами моделей, а также с таблицей, в которой отображены истории миграций:

Рисунок 3. Таблицы в базе данных.

DTO (Data Transfer Object)

Для разделения слоев бизнес-логики и передачи данных через API использованы DTO-классы. Они обеспечивают инкапсуляцию данных и защиту от прямого изменения моделей базы. Для этого были разработаны DTO для чтения, записи или обновления данных:

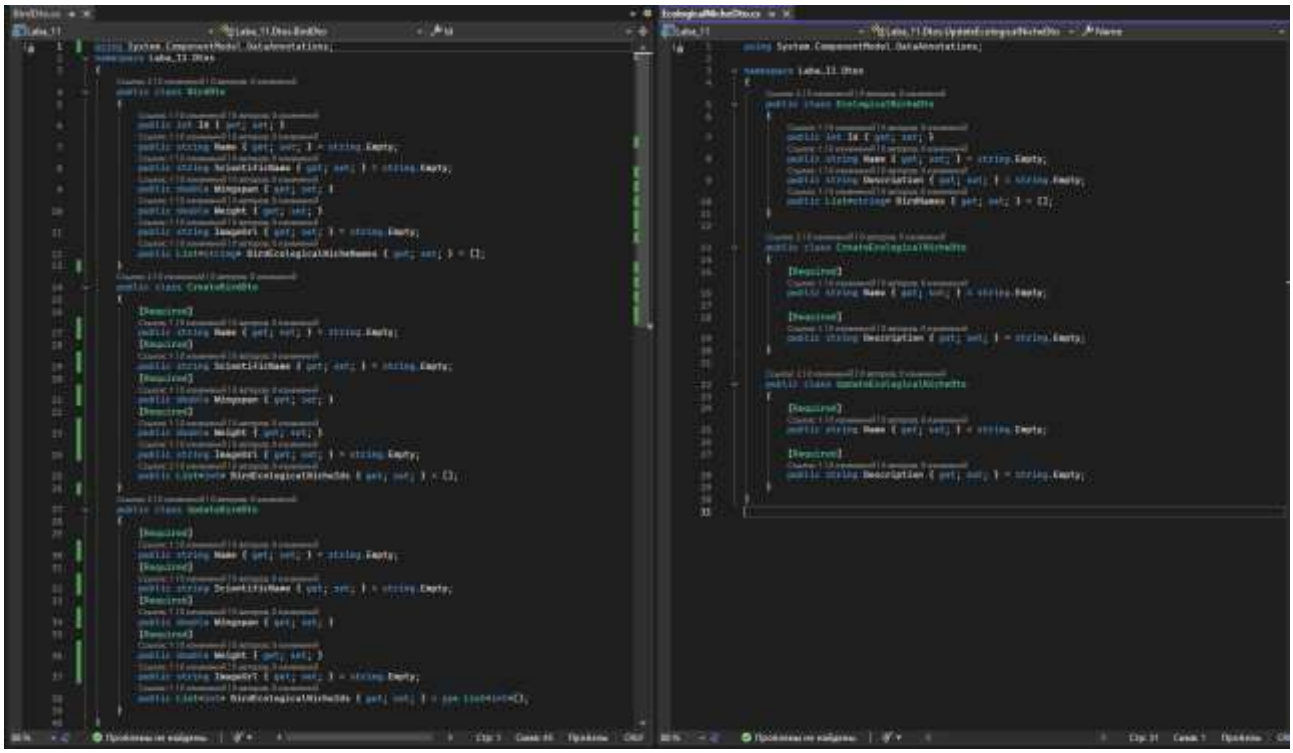


Рисунок 4. Листинги скриптов DTO.

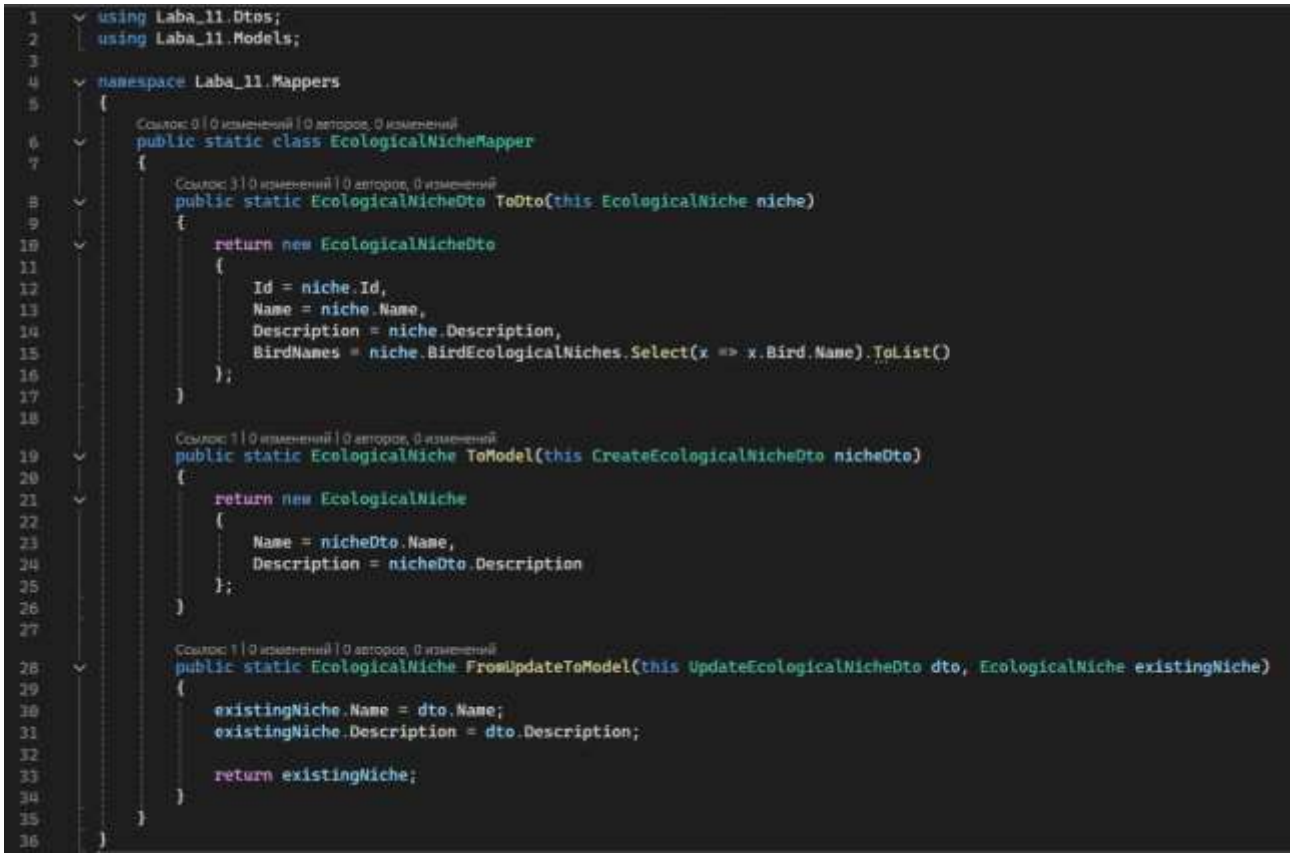
Реализация мапперов

Для преобразования моделей данных в DTO и обратно были реализованы классы `BirdMapper` и `EcologicalNicheMapper` обеспечивающий удобную конвертацию объектов.

Листинг 1. BirdMapper.cs

```
using Laba_11.Dtos;
using Laba_11.Models;

namespace Laba_11.Mappers
{
    public static class BirdMapper
    {
        public static BirdDto ToDto(this Bird bird)
        {
            return new BirdDto
            {
```

```
                Id = bird.Id,
                Name = bird.Name,
                ScientificName = bird.ScientificName,
                Wingspan = bird.Wingspan,
                Weight = bird.Weight,
                ImageUrl = bird.ImageUrl,
                BirdEcologicalNicheNames = bird.BirdEcologicalNiches
                    .Select(x => x.EcologicalNiche.Name).ToList()
            };
        }

        public static Bird ToModel(this CreateBirdDto birdDto)
        {
            var bird = new Bird
            {
                Name = birdDto.Name,
                ScientificName = birdDto.ScientificName,
                Wingspan = birdDto.Wingspan,
                Weight = birdDto.Weight,
                ImageUrl = birdDto.ImageUrl,
            };

            bird.BirdEcologicalNiches = birdDto.BirdEcologicalNicheIds
                .Select(id => new BirdEcologicalNiche { BirdId = bird.Id, Ecologi-
calNicheId = id })
                .ToList();

            return bird;
        }

        public static Bird FromUpdateToModel(this UpdateBirdDto birdDto, Bird ex-
istingBird)
        {
            existingBird.Name = birdDto.Name;
            existingBird.ScientificName = birdDto.ScientificName;
            existingBird.Wingspan = birdDto.Wingspan;
            existingBird.Weight = birdDto.Weight;
            existingBird.ImageUrl = birdDto.ImageUrl;
            existingBird.BirdEcologicalNiches = birdDto.BirdEcologicalNicheIds
                .Select(id => new BirdEcologicalNiche {
                    BirdId = existingBird.Id,
                    EcologicalNicheId = id
                })
                .ToList();

            return existingBird;
        }
    }
}
```



```
using Laba_11.Dtos;
using Laba_11.Models;

namespace Laba_11.Mappers
{
    Ссылок: 0 | 0 изменений | 0 авторов, 0 изменений
    public static class EcologicalNicheMapper
    {
        Ссылок: 3 | 0 изменений | 0 авторов, 0 изменений
        public static EcologicalNicheDto ToDto(this EcologicalNiche niche)
        {
            return new EcologicalNicheDto
            {
                Id = niche.Id,
                Name = niche.Name,
                Description = niche.Description,
                BirdNames = niche.BirdEcologicalNiches.Select(x => x.Bird.Name).ToList()
            };
        }

        Ссылок: 1 | 0 изменений | 0 авторов, 0 изменений
        public static EcologicalNiche ToModel(this CreateEcologicalNicheDto nicheDto)
        {
            return new EcologicalNiche
            {
                Name = nicheDto.Name,
                Description = nicheDto.Description
            };
        }

        Ссылок: 1 | 0 изменений | 0 авторов, 0 изменений
        public static EcologicalNiche FromUpdateToModel(this UpdateEcologicalNicheDto dto, EcologicalNiche existingNiche)
        {
            existingNiche.Name = dto.Name;
            existingNiche.Description = dto.Description;

            return existingNiche;
        }
    }
}
```

Рисунок 5. Листинг скрипта EcologicalNicheMapper.cs.

Контроллеры

Контроллер BirdController

1. GET /api/bird

Метод обработки запроса на получение всех птиц. Он выполняет запрос к базе данных, чтобы получить список всех объектов типа `Bird` из таблицы `Birds`. Все данные о птицах, включая связанные экологические ниши, возвращаются в формате JSON. Для каждого объекта птицы используется метод `ToDto`, который преобразует данные модели в DTO (Data Transfer Object). Это позволяет передавать только необходимые данные и скрывать внутренние детали реализации. В ответ на запрос возвращается список объектов птиц, сериализованный в JSON.

Листинг 2. Метод GetBirds.

```
// GET: api/birds
[HttpGet]
public async Task<IActionResult> GetBirds()
{
    var birds = await _context.Birds
        .Include(b => b.BirdEcologicalNiches)
```

```
        .ThenInclude(bn => bn.EcologicalNiche)
        .ToListAsync();

    return Ok(birds.Select(b => b.ToDto()));
}
```

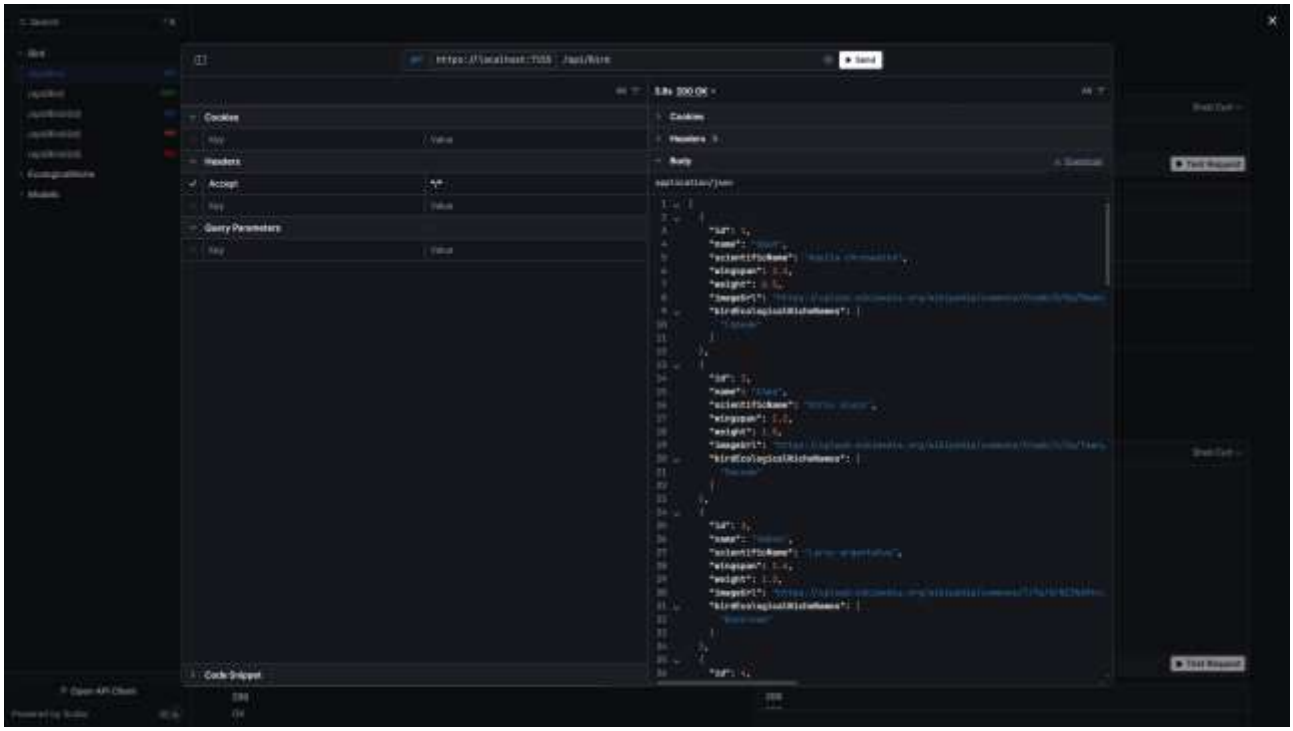


Рисунок 6. Результат метода GetBirds.

2. GET /api/bird/{id}

Метод обрабатывает запрос на получение конкретной птицы по её уникальному идентификатору. Он принимает идентификатор в URL, находит птицу в базе данных по этому идентификатору и возвращает её данные в формате JSON. Если птица с таким идентификатором не найдена, метод возвращает ошибку с кодом 404. Преобразование данных модели в DTO выполняется с помощью того же метода `ToDto`. Ответ содержит всю информацию о выбранной птице, включая её название, научное имя, размах крыльев, вес, изображение и связанные с ней экологические ниши.

Листинг 3. Метод GetBird.

```
// GET: api/birds/{id}
[HttpGet("{id}")]
public async Task<IActionResult> GetBird([FromRoute] int id)
{
    var bird = await _context.Birds
        .Include(b => b.BirdEcologicalNiches)
```

```
        .ThenInclude(bn => bn.EcologicalNiche)
        .FirstOrDefaultAsync(b => b.Id == id);

    if (bird is null)
        return NotFound();

    return Ok(bird.ToDto());
}
```

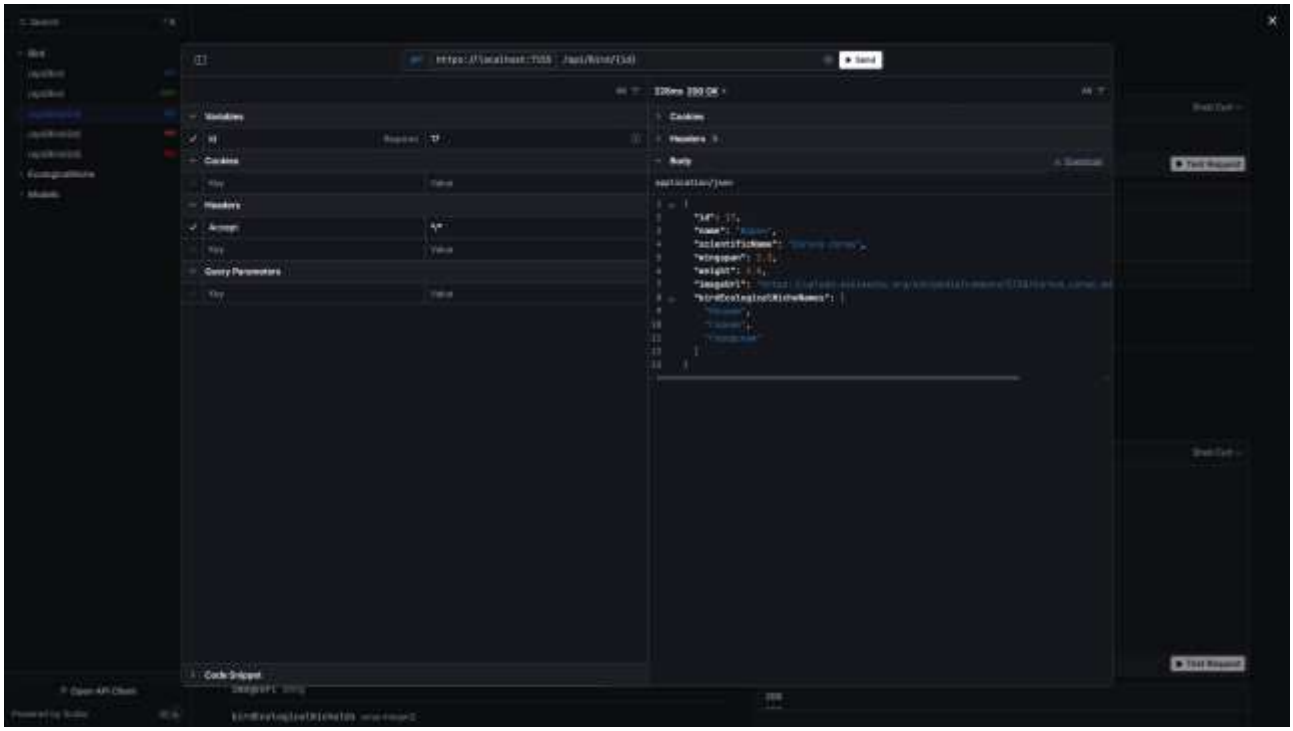


Рисунок 7. Результат метода GetBird.

3. POST /api/bird

Этот метод обрабатывает запрос на добавление новой птицы. Данные для создания новой птицы передаются в теле запроса в формате JSON. В теле запроса ожидается объект `CreateBirdDto`, который включает в себя информацию о птице и идентификаторы экологических ниш, с которыми она связана. Метод преобразует переданные данные в модель `Bird` с помощью маппера, добавляет её в контекст базы данных и сохраняет изменения. Также метод проверяет, указаны ли экологические ниши, и если да, то связывает птицу с соответствующими записями в таблице `EcologicalNiches` через вспомогательную таблицу `BirdEcologicalNiches`. После успешного добавления птицы, метод возвращает

статус 201 (`Created`) и указывает местоположение только что добавленной записи с помощью `CreatedAtAction`, возвращая данные новой птицы в формате DTO.

Листинг 4. Метод CreateBird.

```
// POST: api/birds
[HttpPost]
public async Task<IActionResult> CreateBird([FromBody] CreateBirdDto dto)
{
    var bird = dto.ToModel();

    if (dto.BirdEcologicalNicheIds.Any())
    {
        var niches = await _context.EcologicalNiches
            .Where(n => dto.BirdEcologicalNicheIds.Contains(n.Id))
            .ToListAsync();

        bird.BirdEcologicalNiches = niches
            .Select(n => new BirdEcologicalNiche { Bird = bird, EcologicalNiche = n
})
            .ToList();
    }

    await _context.Birds.AddAsync(bird);

    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(GetBird), new { id = bird.Id }, bird.ToDto());
}
```

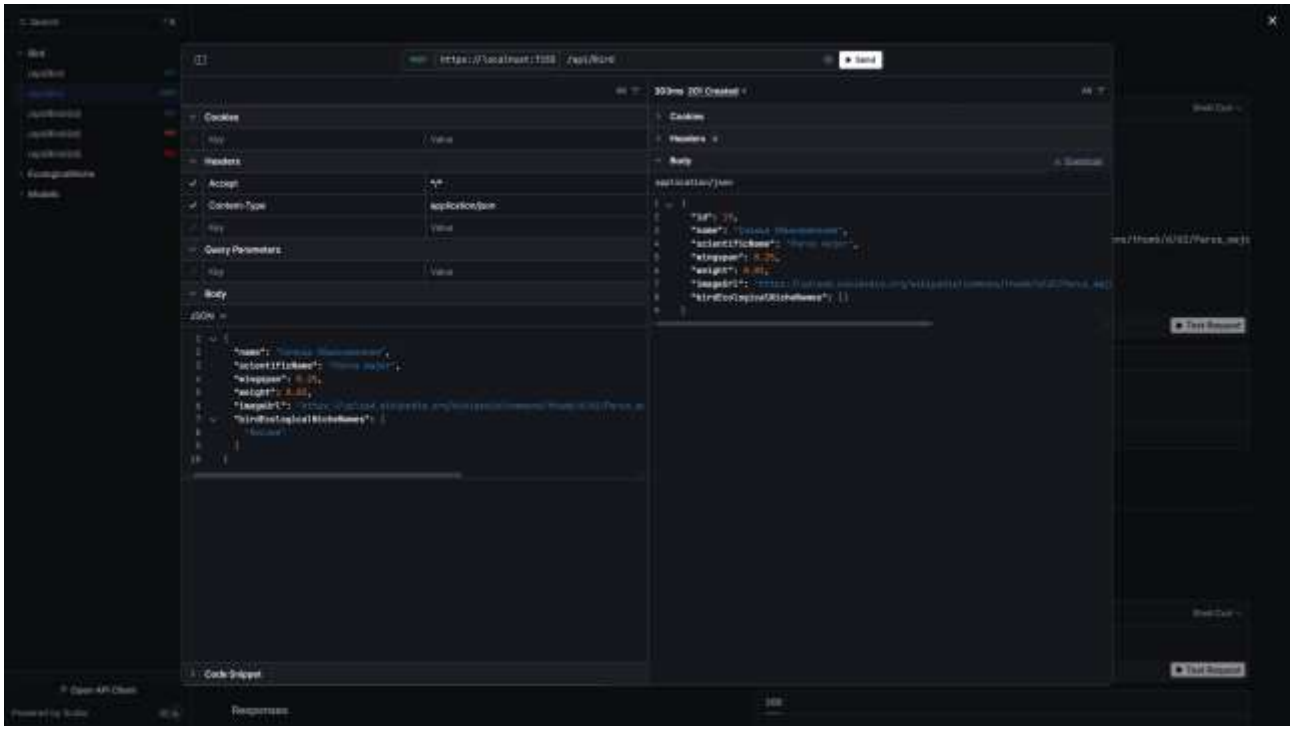


Рисунок 8. Метод CreateBird.

4. PUT /api/bird/{id}

Этот метод предназначен для обновления информации о существующей птице. Он принимает идентификатор птицы в URL и обновленные данные в теле запроса в формате JSON. Запрос содержит объект `UpdateBirdDto`, который включает измененные значения для птицы и идентификаторы её экологических ниш. Метод находит существующую птицу по идентификатору, обновляет её данные с помощью метода `FromUpdateToModel`, который принимает старую модель и новые данные, и затем сохраняет изменения в базе данных. Если птица с данным идентификатором не найдена, возвращается ошибка с кодом 404. После успешного обновления метод возвращает статус 204 (`No Content`), что означает успешное завершение операции без необходимости возвращать данные.

Листинг 5. Метод UpdateBird.

```
// PUT: api/birds/{id}
[HttpPut("{id}")]
public async Task<IActionResult> UpdateBird([FromRoute] int id, UpdateBirdDto dto)
{
    var bird = await _context.Birds
        .Include(b => b.BirdEcologicalNiches)
        .FirstOrDefaultAsync(b => b.Id == id);

    if (bird is null)
```

```
        return NotFound();

    dto.FromUpdateToModel(bird);

    await _context.SaveChangesAsync();

    return NoContent();
}
```

5. DELETE /api/bird/{id}

Метод обработки запроса на удаление птицы по её уникальному идентификатору. Он принимает идентификатор в URL, находит птицу в базе данных и удаляет её. После этого сохраняет изменения в базе данных. Если птица с таким идентификатором не существует, метод возвращает ошибку с кодом 404. После успешного удаления возвращается статус 204 (`No Content`), что свидетельствует о том, что операция завершена успешно, но ответ не содержит данных.

Листинг 6. Метод DeleteBird.

```
// DELETE: api/birds/{id}
[HttpDelete("{id}")]
public async Task<IActionResult> DeleteBird([FromRoute] int id)
{
    var bird = await _context.Birds
        .Include(b => b.BirdEcologicalNiches)
        .FirstOrDefaultAsync(b => b.Id == id);

    if (bird is null)
        return NotFound();

    _context.Birds.Remove(bird);

    await _context.SaveChangesAsync();

    return NoContent();
}
```

Аналогично был разработан контроллер `EcologicalNicheController`.

Приложение

Листинг 7. Скрипт BirdController.

```
using Laba_11.Data;
using Laba_11.Dtos;
using Laba_11.Mappers;
using Laba_11.Models;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace Laba_11.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class BirdController : ControllerBase
    {
        private readonly ApplicationDbContext _context;

        public BirdController(ApplicationDbContext context)
        {
            _context = context;
        }

        // GET: api/birds
        [HttpGet]
        public async Task<IActionResult> GetBirds()
        {
            var birds = await _context.Birds
                .Include(b => b.BirdEcologicalNiches)
                .ThenInclude(bn => bn.EcologicalNiche)
                .ToListAsync();

            return Ok(birds.Select(b => b.ToDto()));
        }

        // GET: api/birds/{id}
        [HttpGet("{id}")]
        public async Task<IActionResult> GetBird([FromRoute] int id)
        {
            var bird = await _context.Birds
                .Include(b => b.BirdEcologicalNiches)
                .ThenInclude(bn => bn.EcologicalNiche)
                .FirstOrDefaultAsync(b => b.Id == id);

            if (bird is null)
                return NotFound();

            return Ok(bird.ToDto());
        }
```

```
        // POST: api/birds
        [HttpPost]
        public async Task<IActionResult> CreateBird([FromBody] CreateBirdDto dto)
        {
            var bird = dto.ToModel();

            if (dto.BirdEcologicalNicheIds.Any())
            {
                var niches = await _context.EcologicalNiches
                    .Where(n => dto.BirdEcologicalNicheIds.Contains(n.Id))
                    .ToListAsync();

                bird.BirdEcologicalNiches = niches
                    .Select(n => new BirdEcologicalNiche { Bird = bird, Ecological-
Niche = n })
                    .ToList();
            }

            await _context.Birds.AddAsync(bird);

            await _context.SaveChangesAsync();

            return CreatedAtAction(nameof(GetBird), new { id = bird.Id },
bird.ToDto());
        }

        // PUT: api/birds/{id}
        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateBird([FromRoute] int id, Up-
dateBirdDto dto)
        {
            var bird = await _context.Birds
                .Include(b => b.BirdEcologicalNiches)
                .FirstOrDefaultAsync(b => b.Id == id);

            if (bird is null)
                return NotFound();

            dto.FromUpdateToModel(bird);

            await _context.SaveChangesAsync();

            return NoContent();
        }

        // DELETE: api/birds/{id}
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteBird([FromRoute] int id)
        {
            var bird = await _context.Birds
```

```
                .Include(b => b.BirdEcologicalNiches)
                .FirstOrDefaultAsync(b => b.Id == id);

            if (bird is null)
                return NotFound();

            _context.Birds.Remove(bird);

            await _context.SaveChangesAsync();

            return NoContent();
        }
    }
}
```

Листинг 8. Скрипт EcologicalNicheController.

```
using Laba_11.Data;
using Laba_11.Dtos;
using Laba_11.Mappers;
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;

namespace Laba_11.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class EcologicalNicheController : ControllerBase
    {
        private readonly ApplicationDbContext _context;

        public EcologicalNicheController(ApplicationDbContext context)
        {
            _context = context;
        }

        // GET: api/EcologicalNiche
        [HttpGet]
        public async Task<IActionResult> GetNiches()
        {
            var niches = await _context.EcologicalNiches
                .Include(n => n.BirdEcologicalNiches)
                .ThenInclude(bn => bn.Bird)
                .ToListAsync();

            return Ok(niches.Select(n => n.ToDto()));
        }

        // GET: api/EcologicalNiche/{id}
```

```
        [HttpGet("{id}")]
        public async Task<IActionResult> GetNiche([FromRoute] int id)
        {
            var niche = await _context.EcologicalNiches
                .Include(n => n.BirdEcologicalNiches)
                .ThenInclude(bn => bn.Bird)
                .FirstOrDefaultAsync(n => n.Id == id);

            if (niche is null)
                return NotFound();

            return Ok(niche.ToDto());
        }

        // POST: api/EcologicalNiche
        [HttpPost]
        public async Task<IActionResult> CreateNiche([FromBody] CreateEcological-
NicheDto dto)
        {
            var niche = dto.ToModel();

            await _context.EcologicalNiches.AddAsync(niche);

            await _context.SaveChangesAsync();

            return CreatedAtAction(nameof(GetNiche), new { id = niche.Id },
niche.ToDto());
        }

        // PUT: api/EcologicalNiche/{id}
        [HttpPut("{id}")]
        public async Task<IActionResult> UpdateNiche([FromRoute] int id, UpdateEco-
logicalNicheDto dto)
        {
            var niche = await _context.EcologicalNiches.FirstOrDefaultAsync(n =>
n.Id == id);

            if (niche is null)
                return NotFound();

            dto.FromUpdateToModel(niche);

            await _context.SaveChangesAsync();

            return NoContent();
        }

        // DELETE: api/EcologicalNiche/{id}
        [HttpDelete("{id}")]
        public async Task<IActionResult> DeleteNiche([FromRoute] int id)
```

```
        {
            var niche = await _context.EcologicalNiches
                .Include(n => n.BirdEcologicalNiches)
                .FirstOrDefaultAsync(n => n.Id == id);

            if (niche is null)
                return NotFound();

            _context.BirdEcologicalNiches.RemoveRange(niche.BirdEcologicalNiches);
            _context.EcologicalNiches.Remove(niche);

            await _context.SaveChangesAsync();

            return NoContent();
        }
    }
}
```