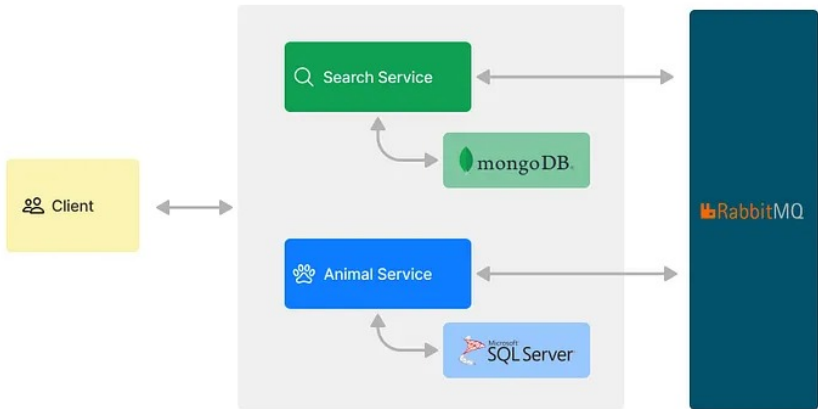


Build Microservices with .NET Core and RabbitMQ (Step-by-Step)

Adam Boucek · Follow
12 min read · Jul 30, 2023

188 3



Animal Adoption Microservices Architecture

The concept of microservices architecture has become increasingly popular in recent times as a contemporary method for constructing large and intricate applications. Instead of creating a single, extensive codebase for an application (known as a monolithic approach), microservices architecture involves breaking down the application into more minor, autonomous services that interact with each other through APIs.

Each microservice is responsible for a specific business function and can be developed and deployed independently from other services. This approach offers several advantages: enhanced flexibility, scalability, resilience, and simplified maintenance and testing procedures.

Regarding implementing microservices, .Net Core is a widely preferred option. In this blog post, we will delve into the fundamentals of microservices architecture using .Net Core and provide some illustrative code examples.

Useful links

[Postman Collection](#)

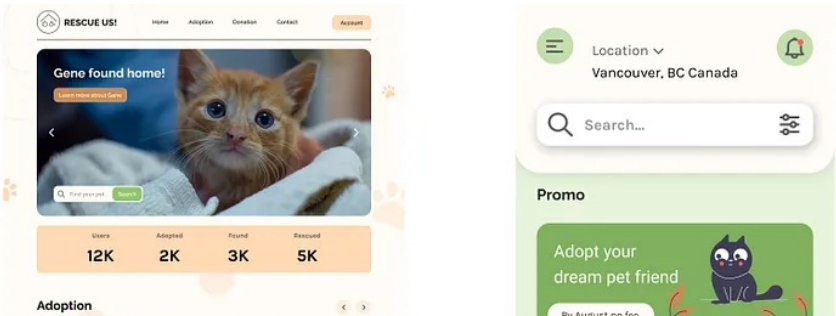
[GitHub Repository](#)

App Overview

App Overview

For this blog post, I prepared a project focusing on adoption animals. This project will be primarily a walkthrough tutorial on how to build a scalable app implementing .Net, NextJs, React Native, Docker, and more.

However, for this post, we will focus on two simple services and how to handle asynchronous communication with RabbitMQ. We will follow the [Database per Service](#) pattern to make our services more independent. Also, it allows us to create services with various databases (MSSQL and MongoDB).



Medium

Sign up to discover human stories that deepen your understanding of the world.

Free

- ✓ Distraction-free reading. No ads.
- ✓ Organize your knowledge with lists and highlights.
- ✓ Tell your story. Find your audience.

Sign up for free

Membership

- ✓ Read member-only stories
- ✓ Support writers you read most
- ✓ Earn money for your writing
- ✓ Listen to audio narrations
- ✓ Read offline with the Medium app

Try for \$5/month

we will create a solution with services and a docker-compose file in our new folder. The following commands will generate .Net boilerplates that we should delete so that they will not interfere with our project.

```
dotnet new sln
touch docker-compose.yml
mkdir Services
cd Services

dotnet new webapi -o AnimalService
dotnet new webapi -o SearchService
```

We will use Docker Compose to spin our database and RabbitMQ for our development. We can spin docker-compose with the following command

```
docker-compose -f docker-compose.yml up.
```

docker-compose.yml

```
version: '3.4'
services:
  sqlserver:
    image: 'mcr.microsoft.com/mssql/server:2022-latest'
    environment:
      ACCEPT_EULA: 'Y'
      MSSQL_SA_PASSWORD: 'Password123'
      MSSQL_PID: 'Express'
    ports:
      - '1433:1433'
    restart: always
    volumes:
      - './drive:/var/opt/mssql'
  mongodb_container:
    image: mongo:latest
    ports:
      - 27017:27017
    volumes:
      - './mongodb_data_container:/data/db'
    restart: always
  rabbitmq:
    image: rabbitmq:3-management-alpine
    ports:
      - 5672:5672
      - 15672:15672
```

Animal Service

Before we start coding anything, we need to install the necessary packages for our service. With NuGet Package Gallery, we can install these **packages**:

```
AutoMapper.Extensions.Microsoft.DependencyInjection

MassTransit.RabbitMQ

Microsoft.EntityFrameworkCore

Microsoft.EntityFrameworkCore.Design

Microsoft.EntityFrameworkCore.SqlServer

AutoMapper.Extensions.Microsoft.DependencyInjection

MassTransit.RabbitMQ

Microsoft.Extensions.Http.Polly

MongoDB.Entities
```

Once we have all packages ready, we can start shaping our service. Because I decided to do this tutorial about animal adoption, we need to begin with Entity. It is an object that we are going to store in our database.

Entities/Animal.cs

```
public class Animal
{
    [Key]
    public Guid Id { get; set; }
    public int PublicId { get; set; }
    public int Age { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public string Breed { get; set; }
    public string Sex { get; set; }
    public int Weight { get; set; }
    public string Color { get; set; }
    public string Description { get; set; }
    public string CoverImageUrl { get; set; }
    public Status Status { get; set; }
    public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
    public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;
}
```

Entities/Status.cs

```
public enum Status
{
    Available,
    Pending,
    Adopted,
    Found,
    Missing
}
```

Now we have to up Entity Framework. Thank to Nuget Packages we will use Microsoft.EntityFrameworkCore.Design and Microsoft.EntityFrameworkCore.SqlServer packages. Be careful; use the identical versions of the packages that match your project's version. Framework NuGet package Microsoft.EntityFrameworkCore.Design.

We will use Code First Migration, meaning Our database schema will be generated based on our writing.

The following step is to create a DB context class that will use DbContext from Entity Framework that makes an abstraction of our database. Notice we also added some seed initial data and outbox for cases our broker would be down while we want to publish an event.

Data/AnimalDbContext.cs

```
public class AnimalDbContext : DbContext
{
    public AnimalDbContext(DbContextOptions options) : base(options) { }

    public DbSet<Animal> Animals { get; set; }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);

        // Add in memory outbox
        builder.AddInboxStateEntity();
        builder.AddOutboxMessageEntity();
        builder.AddOutboxStateEntity();

        // Seed data
```

```
builder.Entity<Animal>().HasData(new Animal
{
    Id = Guid.NewGuid(),
    PublicId = 1,
    Name = "Dee Dee",
    Type = "Dog",
    Description = "Lorem ipsum",
    Breed = "Double doodle",
    Sex = "Female",
    Color = "White",
    Weight = 10,
    Age = 2,
    Status = Status.Available,
    CoverImageUrl = "https://placedog.net/500",
    CreatedAt = DateTime.UtcNow,
    UpdatedAt = DateTime.UtcNow,
});
builder.Entity<Animal>().HasData(new Animal
{
    Id = Guid.NewGuid(),
    PublicId = 2,
    Name = "Buttercup",
    Type = "Cat",
    Description = "Lorem ipsum",
    Breed = "Bengal cat",
    Sex = "Male",
    Color = "Beige",
    Weight = 5,
    Age = 5,
    Status = Status.Available,
    CoverImageUrl = "https://placekitten.com/200/200",
    CreatedAt = DateTime.UtcNow,
    UpdatedAt = DateTime.UtcNow,
});
}
```

Once our DB context class is ready, we have to specify a connection string for our service. For this example, let's put the connect string into `appsettings.Development.json` right behind the `Logging` brackets.

appsettings.Development.json

```
"ConnectionStrings": {
  "DefaultConnection": "Server=localhost;User Id=sa;Password=Password123;Database=AnimalDb",
},
"RabbitMq": {
  "Host": "localhost"
}
```

Logically, I don't want this tutorial to be too long; we will include our DB Context and RabbitMQ in the `Program.cs` at once.

Program.cs

```
// Connect to MSSQL with DB Context
builder.Services.AddDbContext<AnimalDbContext>(option =>
{
    option.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection"));
});

// Configure RabbitMQ
builder.Services.AddMassTransit(x =>
{
    // Add outbox
    x.AddEntityFrameworkOutbox<AnimalDbContext>(o =>
    {
        o.QueryDelay = TimeSpan.FromSeconds(10);

        o.UseSqlServer();
        o.UseBusOutbox();
    });

    x.SetEndpointNameFormatter(new KebabCaseEndpointNameFormatter("animal", false));

    // Setup RabbitMQ Endpoint
    x.UsingRabbitMq((context, cfg) =>
    {
        cfg.Host(builder.Configuration["RabbitMq:Host"], "/", host =>
        {
            host.Username(builder.Configuration.GetValue("RabbitMq:Username", "guest"));
            host.Password(builder.Configuration.GetValue("RabbitMq:Password", "guest"));
        });
        cfg.ConfigureEndpoints(context);
    });
});
```

Now we have to generate an entity migration. Remember, we have to install `dotnet-ef` globally to run the following commands.

```
dotnet ef migrations add InitialMigration -o Data/Migrations
dotnet ef database update
```

Mapper

Now we will create mapper profiles that will help us map our RabbitMQ events and DTOs. The AutoMapper will help us map the classes once we move further in our app.

Helpers/ProfileMapper.cs

```
public ProfileMapper(){
    CreateMap<Animal, AnimalDto>();
    CreateMap<CreateAnimalDto, Animal>();
    CreateMap<AnimalDto, AnimalCreated>();
    CreateMap<Animal, AnimalUpdated>();
}
```

Finally, we can provide a mapper service to our program file.

```
...
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
...
```

Add DTOs

We have to specify what shapes our app requires and what form of data it will return from our `AnimalController`, which we will create shortly.

AnimalDTO, CreatedDTO, and UpdateDto

```

public class AnimalDto
{
    public Guid Id { get; set; }
    public int PublicId { get; set; }
    public int Age { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public string Breed { get; set; }
    public string Sex { get; set; }
    public int Weight { get; set; }
    public string Color { get; set; }
    public string Description { get; set; }
    public string CoverImageUrl { get; set; }
    public string Status { get; set; }
    public DateTime CreatedAt { get; set; }
    public DateTime UpdatedAt { get; set; }
}

```

```

public class CreateAnimalDto
{
    [Required]
    public int Age { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    public string Type { get; set; }
    [Required]
    public string Breed { get; set; }
    [Required]
    public string Sex { get; set; }
    [Required]
    public int Weight { get; set; }
    [Required]
    public string Color { get; set; }
    [Required]
    public string Description { get; set; }
    [Required]
    public string CoverImageUrl { get; set; }
    [Required]
    public Status Status { get; set; }
}

```

```

public class UpdateAnimalDto
{
    public int Age { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public string Breed { get; set; }
    public string Sex { get; set; }
    public int Weight { get; set; }
    public string Color { get; set; }
    public string Description { get; set; }
    public string CoverImageUrl { get; set; }
    public string Status { get; set; }
}

```

In the last part of our Animal service, we have to create a controller to query and edit our data. Also, whenever we change the data, we publish an event that goes to the event bus (RabbitMQ). Notice publishing the event classes with `_publishEndpoint`.

Controllers/AnimalsController.cs

```

[Route("api/[controller]")]
public class AnimalsController : Controller
{
    private readonly AnimalDbContext _context;
    private readonly IMapper _mapper;
    private readonly IPublishEndpoint _publishEndpoint;

    public AnimalsController(AnimalDbContext context, IMapper mapper, IPublishEndpoint publishEndpoint)
    {
        _publishEndpoint = publishEndpoint;
        _mapper = mapper;
        _context = context;
    }

    [HttpGet]
    public async Task<ActionResult<List<AnimalDto>>> GetAllAnimals()
    {
        var animals = await _context.Animals
            .OrderBy(x => x.UpdatedAt)
            .ToListAsync();

        return _mapper.Map<List<AnimalDto>>>(animals);
    }
    [HttpGet("{id}")]
    public async Task<ActionResult<AnimalDto>> GetAnimalById(Guid id)
    {
        var foundAnimal = await _context.Animals.FirstOrDefaultAsync(x => x.Id == id);

        if (foundAnimal == null) return NotFound();

        return _mapper.Map<AnimalDto>(foundAnimal);
    }
    [HttpPost]
    public async Task<ActionResult<AnimalDto>> CreateAnimal(CreateAnimalDto createAnimalDto)
    {
        var animal = _mapper.Map<Animal>(createAnimalDto);

        _context.Animals.Add(animal);

        var newAnimal = _mapper.Map<AnimalDto>(animal);

        await _publishEndpoint.Publish(_mapper.Map<AnimalCreated>(newAnimal));

        var result = await _context.SaveChangesAsync() > 0;

        if (!result) return BadRequest("Could not save changes to the DB");

        return CreatedAtAction(nameof(GetAnimalById),
            new { animal.Id }, newAnimal);
    }
    [HttpPut("{id}")]
    public async Task<ActionResult> UpdateAnimal(Guid id, UpdateAnimalDto updateAnimalDto)
    {
        var animal = await _context.Animals.FirstOrDefaultAsync(x => x.Id == id);

        if (animal == null) return NotFound();

        animal.Description = updateAnimalDto.Description ?? animal.Description;
        animal.Name = updateAnimalDto.Name ?? animal.Name;
        animal.Status = updateAnimalDto.Status != null ? EnumHelper.EnumParse(updateAnimalDto.Status) : animal.Status;
        animal.Breed = updateAnimalDto.Breed ?? animal.Breed;
        animal.CoverImageUrl = updateAnimalDto.CoverImageUrl ?? animal.CoverImageUrl;
        animal.Color = updateAnimalDto.Color ?? animal.Color;
        animal.Type = updateAnimalDto.Type ?? animal.Type;
        animal.CoverImageCaption = updateAnimalDto.CoverImageCaption ?? animal.CoverImageCaption;
        animal.Weight = updateAnimalDto.Weight == 0 ? animal.Weight : updateAnimalDto.Weight;
        animal.Age = updateAnimalDto.Age == 0 ? animal.Age : updateAnimalDto.Age;
        animal.UpdatedAt = DateTime.UtcNow;

        await _publishEndpoint.Publish(_mapper.Map<AnimalUpdated>(animal));

        var result = await _context.SaveChangesAsync() > 0;

        if (result) return Ok();

        return BadRequest("Problem saving changes");
    }
    [HttpDelete("{id}")]
    public async Task<ActionResult> DeleteAnimal(Guid id)
    {
        var animal = await _context.Animals.FindAsync(id);

        if (animal == null) return NotFound();
    }
}

```

```
        _context.Animals.Remove(animal);

        await _publishEndpoint.Publish<AnimalDeleted>(new { Id = animal.Id.ToString() });

        var result = await _context.SaveChangesAsync() > 0;

        if (!result) return BadRequest("Could not update DB");

        return Ok();
    }
}
```

Events

We will create an empty solution containing our events that will consume our RabbitMQ broker.

Also, we need to add references to our services so that we can use these events, like so.

```
dotnet add Services/SearchService/SearchService.csproj reference Services/Events
dotnet add Services/AnimalService/AnimalService.csproj reference Services/Events
```

AnimalCreated.cs

```
public class AnimalCreated
{
    public Guid Id { get; set; }
    public int Age { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public string Breed { get; set; }
    public string Sex { get; set; }
    public int Weight { get; set; }
    public string Color { get; set; }
    public string Description { get; set; }
    public string CoverImageUrl { get; set; }
    public string Status { get; set; }
    public DateTime UpdatedAt { get; set; } = DateTime.Now;
}
```

AnimalUpdated

```
public class AnimalUpdated
{
    public string Id { get; set; }
    public int Age { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public string Breed { get; set; }
    public string Sex { get; set; }
    public int Weight { get; set; }
    public string Color { get; set; }
    public string Description { get; set; }
    public string CoverImageUrl { get; set; }
    public string Status { get; set; }
    public DateTime UpdatedAt { get; set; } = DateTime.Now;
}
```

AnimalDeleted

```
public class AnimalDeleted
{
    public string Id { get; set; }
}
```

Search Service

This service is straightforward. We will receive events from RabbitMQ, which will mimic our database from Animal Service, and we will query the data in MongoDB.

Packages

```
AutoMapper.Extensions.Microsoft.DependencyInjection
```

```
Microsoft.Extensions.Http.Polly
```

```
MassTransit.RabbitMQ
```

```
MongoDB.Entities
```

We will create an object that will help us search through our database. Notice that we also implement a simple pagination.

Helpers/SearchParams.cs

```
public class SearchParams
{
    public string SearchTerm { get; set; }
    public int PageNumber { get; set; } = 1;
    public int PageSize { get; set; } = 4;
    public string Sex { get; set; }
    public string Type { get; set; }
    public string OrderBy { get; set; }
    public string FilterBy { get; set; }
}
```

The next step is to create a MongoDB entity for our Search Service. Notice we do not need an Id property. We drive this Animal class with MongoDB Entity, which will provide ids for our animal.

Data/Animal.cs

```
public class Animal : Entity
{
    public int PublicId { get; set; }
    public int Age { get; set; }
    public string Name { get; set; }
    public string Type { get; set; }
    public string Breed { get; set; }
    public string Sex { get; set; }
}
```



```
public int Weight { get; set; }
public string Color { get; set; }
public string Description { get; set; }
public string CoverImageUrl { get; set; }
public string Status { get; set; }
public DateTime CreatedAt { get; set; } = DateTime.UtcNow;
public DateTime UpdatedAt { get; set; } = DateTime.UtcNow;
}
```

Services/AnimalServiceHttpClient.cs

We will have an HTTP client for this service, so our service can call our Animal Service.

```
public class AnimalServiceHttpClient
{
    private readonly HttpClient _httpClient;
    private readonly IConfiguration _config;

    public AnimalServiceHttpClient(HttpClient httpClient, IConfiguration config)
    {
        _httpClient = httpClient;
        _config = config;
    }

    public async Task<List<Animal>> GetAnimalsForSearchDb()
    {
        return await _httpClient.GetFromJsonAsync<List<Animal>>(_config["AnimalsS
            + "/api/animals");
    }
}
```

Data/DbInitializer.cs

Now we have to create a DB initializer that will create a collection of Animal classes named SearchDb and synchronously receive data from Animal Service and store it in the database.

```
public class DbInitializer
{
    public static async Task InitDb(WebApplication app)
    {
        await DB.InitAsync("SearchDb", MongoClientSettings
            .FromConnectionString(app.Configuration.GetConnectionString("SearchDb")));

        await DB.Index<Animal>()
            .Key(x => x.Type, KeyType.Text)
            .Key(x => x.Breed, KeyType.Text)
            .Key(x => x.Sex, KeyType.Text)
            .CreateAsync();

        var count = await DB.CountAsync<Animal>();

        using var scope = app.Services.CreateScope();

        var httpClient = scope.ServiceProvider.GetRequiredService<AnimalServiceHttpClient>();

        var animals = await httpClient.GetAnimalsForSearchDb();
        Console.WriteLine(animals.Count + " returned from the animal service");

        if (animals.Count > 0) await DB.SaveAsync(animals);
    }
}
```

Mapper

Similarly to our Animal Service, we will create a mapper class and will include event classes we will receive from RabbitMQ.

Helpers/ProfileMapper.cs

```
public class ProfileMapper : Profile
{
    public ProfileMapper()
    {
        CreateMap<AnimalCreated, Animal>();
        CreateMap<AnimalUpdated, Animal>();
    }
}
```

Program.cs

Also, we have to include our RabbitMQ, Http service, and Mapper for this service so our Search Service can work as planned.

```
// Mapper
builder.Services.AddAutoMapper(AppDomain.CurrentDomain.GetAssemblies());
// Http service
builder.Services.AddHttpClient<AnimalServiceHttpClient>().AddPolicyHandler(GetPolicy());
// RabbitMQ
builder.Services.AddMassTransit(x =>
{
    x.AddConsumersFromNamespaceContaining<AnimalCreatedConsumer>();

    x.SetEndpointNameFormatter(new KebabCaseEndpointNameFormatter("search", false));

    x.UsingRabbitMq((context, cfg) =>
    {
        cfg.Host(builder.Configuration["RabbitMq:Host"], "/", host =>
        {
            host.Username(builder.Configuration.GetValue("RabbitMq:Username", "guest"));
            host.Password(builder.Configuration.GetValue("RabbitMq:Password", "guest"));
        });

        cfg.ConfigureEndpoints(context);
    });
});
var app = builder.Build();
// Configure DB connection
app.Lifetime.ApplicationStarted.Register(async () =>
{
    try
    {
        await DbInitializer.InitDb(app);
    }
    catch (Exception e)
    {
        Console.WriteLine(e);
    }
});

...
// Add Async Policy
static IAsyncPolicy<HttpResponseMessage> GetPolicy()
=> HttpPolicyExtensions
    .HandleTransientHttpError()
    .OrResult(msg => msg.StatusCode == HttpStatusCode.NotFound)
    .WaitAndRetryForeverAsync(_ => TimeSpan.FromSeconds(3));
```

Consumers

Consumers will play a crucial role in our service. Our consumers will consume data from RabbitMQ, read the payload and mutate the data in our MongoDB.

Consumers/AnimalCreatedConsumer.cs

```
public class AnimalCreatedConsumer : IConsumer<AnimalCreated>
{
    private readonly IMapper _mapper;

    public AnimalCreatedConsumer(IMapper mapper)
    {
        _mapper = mapper;
    }

    public async Task Consume(ConsumeContext<AnimalCreated> animalCreated)
    {
        Console.WriteLine("Consuming animal created " + animalCreated.Message.Id);

        var animal = _mapper.Map<Animal>(animalCreated.Message);

        await animal.SaveAsync();
    }
}
```

Consumers/AnimalCreatedConsumer.cs

```
public class AnimalCreatedConsumer : IConsumer<AnimalCreated>
{
    private readonly IMapper _mapper;

    public AnimalCreatedConsumer(IMapper mapper)
    {
        _mapper = mapper;
    }

    public async Task Consume(ConsumeContext<AnimalCreated> animalCreated)
    {
        Console.WriteLine("Consuming animal created " + animalCreated.Message.Id);

        var animal = _mapper.Map<Animal>(animalCreated.Message);

        await animal.SaveAsync();
    }
}
```

Consumers/AnimalDeletedConsumer.cs

```
public class AnimalDeletedConsumer : IConsumer<AnimalDeleted>
{
    public async Task Consume(ConsumeContext<AnimalDeleted> animalDeleted)
    {
        Console.WriteLine("Consuming animal delete " + animalDeleted.Message.Id);

        var result = await DB.DeleteAsync<Animal>(animalDeleted.Message.Id);

        if (!result.IsAcknowledged)
            throw new MessageException(typeof(AnimalDeleted), "Problem deleting");
    }
}
```

Consumers/AnimalUpdatedConsumer.cs

```
public class AnimalUpdatedConsumer : IConsumer<AnimalUpdated>
{
    private readonly IMapper _mapper;

    public AnimalUpdatedConsumer(IMapper mapper)
    {
        _mapper = mapper;
    }

    public async Task Consume(ConsumeContext<AnimalUpdated> animalUpdated)
    {
        Console.WriteLine("Consuming animal update " + animalUpdated.Message.Id);

        var animal = _mapper.Map<Animal>(animalUpdated.Message);

        var result = await DB.Update<Animal>().Match(animal => animal.ID == animalUpdated.Message.Id).Update(
            animal => new
            {
                animal.Name,
                animal.Age,
                animal.Description,
                animal.Breed,
                animal.Sex,
                animal.Weight,
                animal.Color,
                animal.Type,
                animal.CoverImageUrl,
                animal.UpdatedAt,
            }, animal).ExecuteAsync();

        if (!result.IsAcknowledged)
            throw new MessageException(typeof(AnimalUpdated), "Problem updating");
    }
}
```

Controller

Finally, we will create a controller that will provide an endpoint `http://localhost:7002/api/search` that will allow us to query our data. Our endpoint will allow us to sort, filter, and paginate through the data.

Controllers/SearchController.cs

```
[ApiController]
[Route("api/search")]
public class SearchController : ControllerBase
{
    [HttpGet]
    public async Task<ActionResult<List<Animal>>> SearchAnimals([FromQuery] SearchParams searchParams)
    {
        var query = DB.PagedSearch<Animal>(searchParams);

        if (!string.IsNullOrEmpty(searchParams.SearchTerm))
        {
            query.Match(Search.Full, searchParams.SearchTerm).SortByTextScore();
        }

        // Sort by parameters
        query = searchParams.OrderBy switch
        {
            "age" => query.Sort(x => x.Age),
            "weight" => query.Sort(x => x.Weight),
            _ => query.Sort(x => x.CreatedAt),
        };
    }
}
```

```
});

// Filter by parameters
query = searchParams.FilterBy switch
{
    "found" => query.Match(x => x.Status == "Found"),
    "pending" => query.Match(x => x.Status == "Pending"),
    "available" => query.Match(x => x.Status == "Available"),
    "missing" => query.Match(x => x.Status == "Missing"),
    _ => query.Sort(x => x.Ascending(y => y.CreatedAt)),
};

if (!string.IsNullOrEmpty(searchParams.Type))
{
    query.Match(x => x.Type == searchParams.Type);
}

if (!string.IsNullOrEmpty(searchParams.Sex))
{
    query.Match(x => x.Sex == searchParams.Sex);
}

query.PageNumber(searchParams.PageNumber);
query.PageSize(searchParams.PageSize);

var result = await query.ExecuteAsync();

return Ok(new
{
    results = result.Results,
    pageCount = result.PageCount,
    totalCount = result.TotalCount
});
}
```

• • •

In the end, we should be able to go to the RabbitMQ dashboard on `http://localhost:15672/` and log in with username `guest` and password `guest`. Then we can go to `connections` and see our services connected like so:

Overview			Details			Network	
Name	User name	State	SSL / TLS	Protocol	Channels	From client	To client
172.23.0.1:58934 AnimalService	guest	<div></div> running	<div></div>	AMQP 0-9-1	1	2 B/s	2 B/s
172.23.0.1:60224 SearchService	guest	<div></div> running	<div></div>	AMQP 0-9-1	1	0 B/s	0 B/s

Listening services

And more importantly, on `Queues` and `Streams`, we can see the events that RabbitMQ distributes across our microservices.

Events of our cluster

• • •

Final thoughts

In this blog post, we've introduced the concept of microservices architecture and illustrated the process of constructing a basic application using .Net and RabbitMQ. We developed two microservices: one for mutating animal data and another for searching through the animal data.

Subsequently, we demonstrated how these microservices can be interconnected to create a unified application capable of providing information about animals that are missing or are available to adopt.

Although this example is straightforward, the underlying principles can be applied to far more intricate systems, offering a simplified approach to developing, deploying, and scaling large applications.

In the next blog post, we will look into how to build other services on top of this project.

• • •

Check out my other posts on [Boucek.dev](#) or projects on [GitHub](#).

Please feel free to contact me on [LinkedIn](#) to share any feedback or suggestions!


Net Core


Microservices


Rabbitmq


Docker Compose


Mssql

 188

 3







Written by Adam Boucek

96 Followers · 2 Following


Full-stack developer. (RemixJS, NestJS, NextJS) See my profile <https://www.boucekdev.com>

Follow

Responses (3)




- What are your thoughts?


Respond
-  Genya

Jun 21, 2024

...

you miss MassTransit.EntityFrameworkCore package in your list


 1

Reply
-  ahmed Zedan

Sep 27, 2024

...

It is better for the explanation to follow the natural sequence of implementation and demonstrate the goal for each step



Reply

Genya
Jun 21, 2024

...

good article, thx

Reply

More from Adam Boucek

Adam Boucek

Polymorphic React Button-or-Link Component in Typescript
To minimize the use of ternary operators across the codebase, we've created a...

Aug 5, 2023 28 1

Adam Boucek

How you can start with GPT Engineer step-by-step
We can hear about ChatGPT and AI everywhere. People discuss what is going to...

Aug 20, 2023 87

Adam Boucek

Data Model—College Athletics
In this article, I will discuss how to design a simple database model for a college athletic...

Jun 28, 2023 4

See all from Adam Boucek

Recommended from Medium

Joud W. Awad

Microservices Pattern: Distributed Transactions (SAGA)
Explore the SAGA Pattern: Ensuring Data Integrity in Microservices. Dive into its...

Jun 17, 2024 1.3K 9

DotNet Full Stack Dev

Configuration Management in .NET Microservices
Managing configurations in microservices is crucial for ensuring that each service...

Jul 28, 2024 12

Lists

Staff picks
800 stories · 1568 saves

Self-Improvement 101
20 stories · 3223 saves

Stories to Help You Level-Up at Work
19 stories · 919 saves

Productivity 101
20 stories · 2718 saves

Maulik Patel

.NET Core Worker Services
In the world of software development, background services are essential for...

Sep 8, 2024 13

In Level Up Coding by Matt Bentley

A Simple Event Sourcing Implementation in .NET
A beginners guide to Event Sourcing and how to implement Event Sourcing in .NET using...

Nov 1, 2023 623 5

Randika Hasheen

Implementing MongoDB in a .NET Core Web API Using Clean...
Introduction

Nov 12, 2024 22

Engr. Md. Hasan Monsur

Step-by-Step Kafka Integration with .NET Core 8 Web API...
Learn how to seamlessly integrate Kafka with .NET Core 8 Web API in this step-by-ste...

Oct 12, 2024 18

See more recommendations

