**Pet Hospital Project – Step 1: Project Setup**

**Project Type:**

* **ASP.NET Core Empty** project (aka Minimal API)
* Keep it clean and simple – no controllers, no Razor, no Entity Framework

**Project Scope – Pet Hospital API**

In this project, we will create a simple **Pet Clinic** API using **ASP.NET Core Minimal API** and **MongoDB** (no Entity Framework).

The system will store pet data in a MongoDB collection with the following structure:

**Pet Document Structure**

{

"name": "Fluffy",

"species": "Dog",

"breed": "Golden Retriever",

"age": 5,

"color": "Golden",

"owner": {

"name": "Jane Doe",

"contact": {

"phone": "123-456-7890",

"email": "jane@example.com"

}

},

"medicalHistory": [

{

"date": "2024-03-01T00:00:00Z",

"visitReason": "Annual check-up",

"treatment": "General exam and vaccines",

"prescription": "None"

}

],

"vaccinations": [

{

"vaccineName": "Rabies",

"dateAdministered": "2023-03-01T00:00:00Z"

}

]

}

const mongopetschema=new mongoose.Schema({

    name: {

        type:String,

        required:true

    },

    species: {

        type:String,

        required:true

    },

    breed: {

        type:String,

        required:true

    },

    age: {

        type:Number,

        required:true

    },

    color: {

        type:String,

        required:true

    },

    owner: {

        name: {

            type:String,

            required:true

        },

        contact: {

            phone:String,

            email:String

        }

    },

    medicalHistory: [{

        date:Date,

        visitReason:String,

        treatment:String,

        prescription:String

    }],

    vaccinations: [{

        vaccineName:String,

        dateAdministered:Date

    }]

})

**Key Fields**

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| name | String | Pet's name |
| species | String | Type of animal (e.g., Dog, Cat) |
| breed | String | Specific breed of the pet |
| age | Number | Age in years |
| color | String | Color of the pet |
| owner | Object | Contains owner's name and contact info |
| medicalHistory | Array | List of previous visits/treatments |
| vaccinations | Array | List of administered vaccines |

**QUESTION**

should we use mongodb or mongoose?   
**Answer:**  
It depends on the **tech stack** you're using.

**Simple Rule:**

* **Use MongoDB Driver** if you're working with **ASP.NET (C#)**
* **Use Mongoose** if you're working with **Node.js (JavaScript)**

**Why?**

* **Mongoose** is a library built **specifically for Node.js**.  
  It helps manage MongoDB with schemas, validation, and easier queries in JavaScript.
* **ASP.NET** runs on **C#**, so it uses the **MongoDB .NET Driver** to talk to MongoDB directly.  
  There's no Mongoose for C#, and you don't need it — the driver does the job.

**Summary**

| **Tech Stack** | **What to Use** |
| --- | --- |
| Node.js | Mongoose |
| ASP.NET (C#) | MongoDB.Driver |

**Step-by-Step Instructions**

**Step 1 Create the Project**

**dotnet new web -n PetHospitalApi**

**cd PetHospitalApi**

We're using the **Empty template** to build a minimal and lightweight backend.

**Step 2 NuGet Packages**

**Add Necessary NuGet Packages**

Here’s what we need to get started:

| **Package** | **Purpose** |
| --- | --- |
| MongoDB.Driver | To connect and interact with MongoDB |
| Swashbuckle.AspNetCore | To add Swagger UI for testing our APIs |
| Microsoft.AspNetCore.OpenApi | For OpenAPI support (if needed for minimal APIs) |
|  |  |

**dotnet add package MongoDB.Driver**

**dotnet add package Swashbuckle.AspNetCore**

Optional: Add Microsoft.AspNetCore.OpenApi if you want more control over OpenAPI annotations.

**Not Using:**

* ❌ Entity Framework
* ❌ SQL-based DBs
* ❌ Razor Pages or MVC
* ✅ Keeping it raw and minimal, just HTTP endpoints + MongoDB

**Step 3 NuGet Packages**

Create **“Models**” folder and create the models for this project.

**Pet.cs**

//PET MODEL

//mongo Bson

using MongoDB.Bson;

// explain

using MongoDB.Bson.Serialization.Attributes;

using System.ComponentModel.DataAnnotations;

namespace PetClinic1.Models

{

    public class Pet

    {

        [BsonId]

        [BsonRepresentation(BsonType.ObjectId)]

        public string? Id { get; set; }

        [Required]

        public string? Name { get; set; }

        [Required]

        public string? Species { get; set; }

        [Required]

        public string? Breed { get; set; }

        [Required]

        public int Age { get; set; }

        [Required]

        public string Color {get;set;}="generic";

        // pet's owner

        [Required]

        public Owner? Owner {get;set;}

        //medical history

        [Required]

        public List<MedicalHistoryEntry> MedicalHistory {get;set;}=new List<MedicalHistoryEntry>();

        [Required]

        public List<VaccinationRecord> Vaccinations {get;set;}=new List<VaccinationRecord>();

    }

}

**Owner.cs**

//Owner

using System.ComponentModel.DataAnnotations;

namespace PetClinic1.Models

{

    public class Owner

    {

        [Required]

        public string? Name { get; set; }

        public Contact? Contact { get; set; }

    }

}

**Contact.cs**

namespace PetClinic1.Models

{

    public class Contact

    {

        public string? Phone { get; set; }

        public string? Email { get; set; }

    }

}

**MedicalHistoryEntity.cs**

namespace PetClinic1.Models

{

    public class MedicalHistoryEntry

    {

        public DateTime Date { get; set; }

        public string? VisitReason { get; set; }

        public string? Treatment { get; set; }

        public string? Prescription { get; set; }

    }

}

**VaccinationRecord.cs**

namespace PetClinic1.Models

{

    public class VaccinationRecord

    {

        public string? VaccineName { get; set; }

        public DateTime DateAdministered { get; set; }

    }

}

**Models Explanation**

**What is [BsonId] and [BsonRepresentation(BsonType.ObjectId)]?**

These are **MongoDB-specific attributes** from the MongoDB.Bson.Serialization.Attributes namespace. They tell the MongoDB driver how to map your C# class to a MongoDB document.

**[BsonId]**

Marks a property as the **primary key** (the \_id field in MongoDB).

Example:

[BsonId]

public string? Id { get; set; }

This says: "This property will be saved as \_id in the database."

**[BsonRepresentation(BsonType.ObjectId)]**

This allows MongoDB to store the ID as an **ObjectId** in the database, while you handle it as a **string** in C#.

So this:

[BsonRepresentation(BsonType.ObjectId)]

        public string? Id { get; set; }

lets you write code like string id = pet.Id instead of messing with ObjectId type directly.

**Why only in Pet.cs?**

Because:

* Pet is the **main document** (the actual thing stored in the collection).
* Owner, MedicalHistoryEntry, etc. are **embedded sub-documents** inside the Pet document.
* Only the **top-level document** needs an \_id field unless you plan to store sub-documents in separate collections.

**Summary**

| **Attribute** | **Use in…** | **Why** |
| --- | --- | --- |
| [BsonId] | On Pet.Id | Maps to MongoDB’s \_id field |
| [BsonRepresentation(...)] | On Pet.Id | Store as ObjectId, use as string in C# |
| Not needed in Owner, etc. | They’re sub-objects | They’re embedded, not standalone docs |

**Step 4 MongoDb Service Class**

We need a service class to create a mongo connection.

Create Service Folder and inside this folder create

**MongoDbService.cs**

The **MongoDbService** is about logic — how you connect to the database and perform operations.

**MongoDbService.cs** basic version

using MongoDB.Bson;

using MongoDB.Driver;

using PetClinic1.Models;

namespace PetClinic1.Services

{

    public class MongoDbService

    {

        private readonly IMongoCollection<Pet> \_pets;

        public MongoDbService(IConfiguration config)

        {

            var connectionString = config.GetSection("MongoDB:ConnectionString").Value;

            var databaseName = config.GetSection("MongoDB:DatabaseName").Value;

            var client = new MongoClient(connectionString);

            var database = client.GetDatabase(databaseName);

            // Use the collection for Pet documents

            \_pets = database.GetCollection<Pet>("pethospitalcollection");

            //\_collection = database.GetCollection<BsonDocument>("pethospitalcollection");

        }

    }

}

Now that we've built our models, it's time to **connect our application to MongoDB**.

**What This Service Does:**

* Reads MongoDB configuration from appsettings.json
* Creates a MongoDB client
* Connects to the specified database
* Accesses the "pethospitalcollection" where all Pet documents will be stored
* Uses a **strongly typed collection** (Pet model) instead of raw BSON documents

**Why Use Strongly Typed Models?**

Since we've already defined our Pet class (and related helper classes like Owner, Contact, etc.), using strongly typed models fits our goal better.

It gives us:

* ✔️ **Type safety**
* ✔️ **IntelliSense** (code suggestions)
* ✔️ **Cleaner, more readable code**
* ✔️ **Easier serialization/deserialization** between C# objects and MongoDB documents

**BsonDocument vs Pet Model**

| **Using BsonDocument** | **Using Pet Model** |
| --- | --- |
| Work with raw MongoDB data | Work with well-defined C# objects |
| No type checking | Type-safe access |
| Harder to read and maintain | Clean and intuitive code |
| Must access fields manually | Use dot notation (pet.Name, etc.) |

**When to Use Each**

If you **don’t have a model**, you might write something like:

**private readonly IMongoCollection<BsonDocument> \_collection;**

**\_collection = database.GetCollection<BsonDocument>("pethospitalcollection");**

But in our case, we already have models, so this approach is better:

private readonly IMongoCollection<Pet> \_pets;

\_pets = database.GetCollection<Pet>("pethospitalcollection");

This clearly reflects the structure of our data and keeps everything easier to work with.

**Step 5 Configure appsettings.json**

Before we write our CRUD operations in our MongoDbService.cs, we need to add our MongoDB **connection string** and **database name** to the appsettings.json file so our service can load them correctly.

appsettings.json

{

  "Logging": {

    "LogLevel": {

      "Default": "Information",

      "Microsoft.AspNetCore": "Warning"

    }

  },

  "MongoDB": {

    "ConnectionString": <your-atlas-connection-string>,

    "DatabaseName": "pethospital1"

  },

  "AllowedHosts": "\*"

}

**Step 6 Register MongoDb to main program**

After we set our applicationsettings.json as we did, we need to register mongodbservice to main program.

In Program.cs after

var builder = WebApplication.CreateBuilder(args);

**so now our program**

using PetClinic1.Services;

var builder = WebApplication.CreateBuilder(args);

//register MongoDbService

builder.Services.AddSingleton<MongoDbService>();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

app.Run();

**Step 7 Add CRUD methods in MongoDbService**

Now we can add our basic CRUD services to our MongoDbService.

using MongoDB.Bson;

using MongoDB.Driver;

using PetClinic1.Models;

namespace PetClinic1.Services

{

    public class MongoDbService

    {

        private readonly IMongoCollection<Pet> \_pets;

        public MongoDbService(IConfiguration config)

        {

            var connectionString = config.GetSection("MongoDB:ConnectionString").Value;

            var databaseName = config.GetSection("MongoDB:DatabaseName").Value;

            var client = new MongoClient(connectionString);

            var database = client.GetDatabase(databaseName);

            // Use the collection for Pet documents

            \_pets = database.GetCollection<Pet>("pethospitalcollection");

            //\_collection = database.GetCollection<BsonDocument>("pethospitalcollection");

        }

        public async Task<List<Pet>> GetAllPetsAsync()

        {

            return await \_pets.Find(\_ => true).ToListAsync();

        }

        public async Task<Pet> GetPetByIdAsync(string id)

        {

            return await \_pets.Find(p => p.Id == id).FirstOrDefaultAsync();

        }

        public async Task AddPetAsync(Pet pet)

        {

            await \_pets.InsertOneAsync(pet);

        }

        public async Task UpdatePetAsync(string id, Pet updatedPet)

        {

            await \_pets.ReplaceOneAsync(p => p.Id == id, updatedPet);

        }

        public async Task DeletePetAsync(string id)

        {

            await \_pets.DeleteOneAsync(p => p.Id == id);

        }

    }

}

**Step 8 Update program.cs**

Now we can write an endpoint to test our database and connection

In Program let us get all pets

using PetClinic1.Services;

using PetClinic1.Models;

using Microsoft.AspNetCore.Mvc; //optional since it is minimalist program

var builder = WebApplication.CreateBuilder(args);

//register MongoDbService

builder.Services.AddSingleton<MongoDbService>();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

//add an end-point to test our database

//get all

app.MapGet("/pets", async (MongoDbService db)=> {

    var pets=await db.GetAllPetsAsync();

    return Results.Ok(pets);

});

app.Run();

when you run program and enter endpoint address you may see an empty list, since we did not have any pet inside our database.

A screenshot of a computer

AI-generated content may be incorrect.

In this point we can connect our mongosh or compass to see if database created.

However, we look at the collections we won’t see newly created database

**MongoDB Behavior:**

MongoDB **won’t create the database or the collection** until you actually insert a document.

So even though:

* You’ve set the connection string ✅
* You’ve connected in Compass ✅
* You’ve set the database and collection names ✅

You **won’t see** anything in Compass until you call:

**await \_pets.InsertOneAsync(pet);**

**Step 9 Add the POST Endpoint to Add a New Pet**

Open your Program.cs and add this after your app.MapGet(...) lines:

//add new pet

//add [frombody ] optional since it is minimal program

app.MapPost("/pets", async ([FromBody] Pet pet, MongoDbService db) =>

{

    await db.AddPetAsync(pet);

    return Results.Created($"/pets/{pet.Id}", pet);

});

Now program.cs

using PetClinic1.Services;

using PetClinic1.Models;

using Microsoft.AspNetCore.Mvc; //optional since it is minimalist program

var builder = WebApplication.CreateBuilder(args);

//register MongoDbService

builder.Services.AddSingleton<MongoDbService>();

var app = builder.Build();

app.MapGet("/", () => "Hello World!");

//add an end-point to test our database

//get all

app.MapGet("/pets", async (MongoDbService db)=> {

    var pets=await db.GetAllPetsAsync();

    return Results.Ok(pets);

});

//add new pet

//add [frombody ] optional since it is minimal program

app.MapPost("/pets", async ([FromBody] Pet pet, MongoDbService db) =>

{

    await db.AddPetAsync(pet);

    return Results.Created($"/pets/{pet.Id}", pet);

});

app.Run();

**Step 10 Test**

Let us test our application with postman. Create a new collection, create an post request for <http://localhost:5120/pets>

**pet1**

{

"name": "Buddy",

"species": "Dog",

"breed": "Labrador",

"age": 4,

"color": "Brown",

"owner": {

"name": "Alice",

"contact": {

"phone": "1234567890",

"email": "alice@example.com"

}

},

"medicalHistory": [],

"vaccinations": []

}

**Pet2**

{

"name": "Whiskers",

"species": "Cat",

"breed": "Siamese",

"age": 3,

"color": "Gray",

"owner": {

"name": "John Smith",

"contact": {

"phone": "9876543210",

"email": "johnsmith@example.com"

}

},

"medicalHistory": [

{

"date": "2024-06-15T00:00:00Z",

"visitReason": "Fever",

"treatment": "Antibiotics",

"prescription": "Amoxicillin"

}

],

"vaccinations": [

{

"vaccineName": "FVRCP",

"dateAdministered": "2023-12-01T00:00:00Z"

}

]

}

**Check your compass again**

A screenshot of a computer

AI-generated content may be incorrect.

You should now see:

* The pethospital1 database
* The pethospitalcollection

|  |
| --- |
| SUMMARY  Now you've got:   * MongoDB connected * Models built * CRUD service written * Endpoints tested * Data showing up in Compass |

**Pet Hospital Project – Step 2: SWAGGER AND FILTERS**

In the first part of our tutorial, we built a basic MongoDB-backed service with full CRUD operations. However, those operations are not yet available as API endpoints.

In this step, we will:

1. Create RESTful endpoints that connect to our service
2. Add **Swagger** for easy API testing and documentation
3. (Optional) Add custom filters and enhancements for better usability

By the end of this section, you'll be able to test your API interactively and manage pet records directly from Swagger UI.

**Step 10 ADD RESTful endpoints**

In our program, we have only two endpoints: Get all pets and Create pet. Let us create update and delete endpoints.

**Delete**: When we create delete endpoint, we need to find a reference which we can use for deleting operation. If we used a relational database, we might use id number. But as you might notice, mongo dB keys are unique and not suitable to use directly to delete document from collection. Theoretically , we can delete by using unique Id: Here’s an example:

A screenshot of a computer

AI-generated content may be incorrect.

In our MongoDbService.cs we have already method to delete document:

        public async Task DeletePetAsync(string id)

        {

            await \_pets.DeleteOneAsync(p => p.Id == id);

        }

But it seems that we may not use this since unique ids are quite complex.

**In real applications, you don’t let users delete by \_id directly.**

Instead, you give them something **human-friendly** to delete by, like:

* Pet name
* Pet name + owner name
* Pet name + some unique field (like owner's email or phone)
* Or you **list the pets first**, let them choose, and then internally use \_id

**💡 So what can you do *now* (in a backend-only setup)?**

If you want a better experience than requiring \_id, add a delete method like this:

Add this to your “MongoDbService.cs” service file:

//delete by pet name AND owner name

        public async Task<string> DeletePetByNameAndOwnerAsync( DeleteByNameOwnerRequest deleteRequest)

        {

            var result = await \_pets.DeleteOneAsync(p =>

                p.Name == deleteRequest.Name && p.Owner.Name == deleteRequest.OwnerName);

            if (result.DeletedCount == 0)

            {

                return $"Pet with name '{deleteRequest.Name}' and owner '{deleteRequest.OwnerName}' not found.";

            }

            return $"Pet '{deleteRequest.Name}' owned by '{deleteRequest.OwnerName}' deleted successfully.";

        }

And we need a delete request:

//delete pet name and owner name request

    public class DeleteByNameOwnerRequest

    {

        public string Name {get;set;}

        public string OwnerName {get;set;}

    }

Now we write our delete endpoint and test with POSTMAN.

//delete a pet with name and owner name using REQUEST

app.MapDelete("/pets", async ([FromBody] DeleteByNameOwnerRequest request, MongoDbService db)=> {

    try {

        var result=await db.DeletePetByNameAndOwnerAsync(request);

        return Results.Ok(result);

    }

    catch(Exception ex)

    {

        return Results.Problem(ex.Message);

    }

});

Note: The method here is using DeleteByNameOwnerRequest, it is also possible to pass a “Pet” object and use (string petName, string ownerName).

TASK: Use “Pet” object instead of request and re-write service and endpoint.

**POSTMAN delete request**

DELETE : <http://localhost:5120/pets>

{ "name":"test",

"ownername": "Jone JAne"

}

**Update:** In order to update an existed “document” we can also user ownerName and pet name. First, we will create a new service method and use it in our endpoint.

Update **MongoDbService.cs**

//update by pet name AND owner name

        public async Task<string> UpdatePetByNameAndOwnerAsync(string petName, string ownerName, Pet updatedPet)

        {

            var result = await \_pets.ReplaceOneAsync(

                p => p.Name == petName && p.Owner.Name == ownerName,

                updatedPet

            );

            if (result.MatchedCount == 0)

            {

                return $"Pet with name '{petName}' and owner '{ownerName}' not found.";

            }

            return $"Pet '{petName}' owned by '{ownerName}' updated successfully.";

        }

Update **Program.cs**

//update by name and owner name

app.MapPut("/pets", async ([FromBody] Pet updatedPet, MongoDbService db) =>

{

    try

    {

        var result = await db.UpdatePetByNameAndOwnerAsync(updatedPet.Name, updatedPet.Owner.Name, updatedPet);

        return Results.Ok(result);

    }

    catch (Exception ex)

    {

        return Results.Problem(ex.Message);

    }

});

**Test with POSTMAN**

**Update 1: Change Age from 4 to 5**

POSTMAN:

Raw body:

You can send a PUT /pets request with this payload:

{

"id": "67e9bfc536456d879265e4d4",

"name": "Buddy",

"species": "Dog",

"breed": "Labrador",

"age": 5,

"color": "Brown",

"owner": {

"name": "Alice",

"contact": {

"phone": "1234567890",

"email": "alice@example.com"

}

},

"medicalHistory": [],

"vaccinations": []

}

A screenshot of a computer

AI-generated content may be incorrect.

### **Update 2: Add a New Vaccine**

Let’s say we add a **rabies vaccine** with a date:

Let’s say we add a **rabies vaccine** with a date:

Postman put request

<http://localhost:5120/pets>

{

"id": "67e9bfc536456d879265e4d4",

"name": "Buddy",

"species": "Dog",

"breed": "Labrador",

"age": 5,

"color": "Brown",

"owner": {

"name": "Alice",

"contact": {

"phone": "1234567890",

"email": "alice@example.com"

}

},

"medicalHistory": [],

"vaccinations": [

{

"vaccineName": "Rabies",

"dateAdministered": "2024-03-15T00:00:00Z"

}

]

}

And after update:

A screenshot of a computer

AI-generated content may be incorrect.

**Step 11 Filters**

**Filter 1 :“Find all pets that belong to a specific user (by owner name)”**

You already know the correct MongoDB query:

> db.pethospitalcollection.find({"Owner.Name":"Alice"})

**Let’s implement this in your code now, step by step:**

**Add a Method in MongoDbService.cs**

//find an owners all pets

        public async Task<List<Pet>> GetPetsByOwnerNameAsync(string ownerName)

        {

            return await \_pets.Find(p => p.Owner.Name == ownerName).ToListAsync();

        }

This uses a simple filter to check the nested Owner.Name field.

**Add Minimal API Endpoint**

//owner's all pet

app.MapGet("/pets/by-owner/{ownerName}", async (string ownerName, MongoDbService db) =>

{

    try

    {

        var pets = await db.GetPetsByOwnerNameAsync(ownerName);

        if (pets.Count == 0)

            return Results.NotFound($"No pets found for owner '{ownerName}'");

        return Results.Ok(pets);

    }

    catch (Exception ex)

    {

        return Results.Problem(ex.Message);

    }

});

**Postman**

GET

<http://localhost:5120/pets/by-owner/Alice>

**Filter 2: Find all pets by species (e.g., all dogs or all cats)**

/pets/by-species/{species}

**Service:**

//find all pets by species

        public async Task<List<Pet>> GetPetsBySpeciesAsync(string species)

        {

            return await \_pets.Find(p => p.Species == species).ToListAsync();

        }

**EndPoint:**

//get species

app.MapGet("/pets/by-species/{species}", async (string species, MongoDbService db) =>

{

    try

    {

        var pets = await db.GetPetsBySpeciesAsync(species);

        if (pets.Count == 0)

            return Results.NotFound($"No pets found for species '{species}'");

        return Results.Ok(pets);

    }

    catch (Exception ex)

    {

        return Results.Problem(ex.Message);

    }

});

**For example**

[**http://localhost:5120/pets/by-species/Cat**](http://localhost:5120/pets/by-species/Cat)

**will return:**

**A screen shot of a computer

AI-generated content may be incorrect.**

**Filter 3: Find all Vaccinated pets**

/pets/vaccinated

We’ll look for pets where the Vaccinations array is **not empty**.

**Service Method:**

//vaccinated pets

        public async Task<List<Pet>> GetVaccinatedPetsAsync()

        {

            return await \_pets.Find(p => p.Vaccinations.Any()).ToListAsync();

        }

**.Any()** is a LINQ-to-Mongo friendly way to check if the array has elements**.**

**Endpoint:**

<http://localhost:5120/pets/vaccinated>

app.MapGet("/pets/vaccinated", async (MongoDbService db) =>

{

    try

    {

        var pets = await db.GetVaccinatedPetsAsync();

        if (pets.Count == 0)

            return Results.NotFound("No vaccinated pets found.");

        return Results.Ok(pets);

    }

    catch (Exception ex)

    {

        return Results.Problem(ex.Message);

    }

});

**RESULT**

**A computer screen shot of white text

AI-generated content may be incorrect.**

**Adding Swagger to Your Minimal ASP.NET Core API**

Swagger helps us generate interactive API documentation — super useful for testing and showcasing endpoints.

Since we're using a **simplistic Minimal API** structure, we can't use traditional /// XML comments directly on methods. There's a clean way to do it using WithSummary() and WithDescription().

**Step 1: Install Required NuGet Packages**

Make sure you’ve installed the following NuGet packages:

**dotnet add package Swashbuckle.AspNetCore**

**Step 2** **Configure Swagger in Program.cs**

Add the following to your builder before var **app=builder.Build();**

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(options =>

{

    options.SwaggerDoc("v1", new OpenApiInfo

    {

        Version = "v1",

        Title = "Mongo Pet Hospital",

        Description = "Pet Hospital"

    });

    //enable xml

    var xmlFilename = $"{Assembly.GetExecutingAssembly().GetName().Name}.xml";

    options.IncludeXmlComments(Path.Combine(AppContext.BaseDirectory, xmlFilename));

});

This sets up Swagger and enables it to read XML comments (which we’ll configure in a moment).

**Step 3 Enable Swagger in Development Environment**

Inside your Program.cs, after var app = builder.Build();, add this:

if (app.Environment.IsDevelopment())

{

    app.UseSwagger();

    app.UseSwaggerUI();

}

This makes Swagger available when running the project in development mode.

**Step 4 Enable Swagger in Development Environment**

Edit your project file PetClinic1.csproj and add the following inside the <PropertyGroup>:

<PropertyGroup>

    <TargetFramework>net9.0</TargetFramework>

    <Nullable>enable</Nullable>

    <ImplicitUsings>enable</ImplicitUsings>

    <GenerateDocumentationFile>true</GenerateDocumentationFile>

    <NoWarn> $(NoWarn);1591</NoWarn>

  </PropertyGroup>

GenerateDocumentationFile tells the compiler to generate XML docs.  
NoWarn 1591 suppresses warnings for missing documentation on public members.

**Step 5: Add Endpoint Metadata for Swagger**

Since we're not using controllers with /// XML comments, we use WithSummary() and WithDescription() for Swagger to read.Here’s how to document an endpoint:

//get all

app.MapGet("/pets", async (MongoDbService db) =>

{

    var pets = await db.GetAllPetsAsync();

    return Results.Ok(pets);

})

.WithName("GetAllPets")

.WithSummary("Returns all pets in the system")

.WithDescription("Fetches pet records");

**Step 6: Run the App**

Now when you run the application and open your browserto:

https://localhost:{port}/swagger

You’ll see your full Swagger UI with summaries and descriptions for each endpoint.

**A screenshot of a computer

AI-generated content may be incorrect.**

**TASK**

We have seen some basic custom filters and Swagger methods . Now it’s your turn to create some new custom filters.

**First: Complete Swagger UI comments for each end point**

**1. Find all pets older than a certain age**

Create a filter that returns all pets older than a given age (e.g., 5 years). The age should be passed as a route parameter.

**2. Get all pets of a certain breed**

Allow clients to query pets by breed, such as "Labrador" or "Siamese". The result should return all pets matching that breed.

**3. Find all pets that have at least one medical history record**

Create an endpoint that returns only the pets that have any medical history entries (i.e., MedicalHistory array is not empty).

**4. List all pets that are not vaccinated**

Return all pets where the Vaccinations array is empty. These pets might need vaccination reminders!

**5. Find pets by color and species**

Combine two filters: pets that match both a specific color and species (e.g., all "Gray Cats").

**6. Get all pets with a specific vaccine**

Find pets who have received a specific vaccine (e.g., "Rabies") inside their Vaccinations array.

**7. Count how many pets each owner has**

(Advanced) Return a list of owner names along with the total number of pets they have in the system.

**Pet Hospital Project FRONT-END**

Our Back end is ready. Now we may write a front-end for this project. We will use react-vite.

**Step 1: Create new React-Vite Project**

You should create the frontend **outside your backend folder**, so your project structure will look like this:

PetClinic/

│

├── PetClinic1/ ← your ASP.NET Core backend

├── frontend/ ← your new React frontend (Vite + TS)

To create the frontend:

**npm create vite@latest**

It will ask you a few questions:

* **Project name:** frontend
* **Select framework:** React
* **Select variant:** TypeScript

This will generate a frontend/ folder with your Vite React project.

**Step 2: Move Into the Frontend Folder**

**cd frontend**

Then install dependencies:

**npm install**

Start the development server:

**npm run dev**

**You should now see a working React app at**

[**http://localhost:<port-number**](http://localhost:%3cport-number)**>**

**Step 3: Cleanup**

To simplify your starting point:

* Remove boilerplate files: logo.svg, etc.
* Replace App.tsx with a basic layout:

import "./app.css";

function App() {

  return (

    <>

    <h1>PET HOSPITAL</h1>

    </>

  )

}

export default App

**Step 4: Start Building Features**

Now you can start creating components like:

* <Button /> → to trigger API calls
* <PetTable /> → to display fetched data
* <SearchBar /> → to filter pets by name or owner

We will cover these in following part. But first we will create a navbar which uses “REACT-ROUTES”

**Step 5: REACT ROUTES AND NAVBAR**

For this project, we will create a simple navbar that has only 2 routes. “Home-Page” and “Admin-Panel”.

1. **First install router-dom**

**npm install react-router-dom**

**2. Create the Navbar Component**

Inside the src/Components/ folder, create a file called NavBar.tsx.

import { Link } from "react-router-dom";

const NavBar = () => {

  return (

    <nav style={{ padding: "10px", background: "#f0f0f0" }}>

      <Link to="/" style={{ marginRight: "10px" }}>

        Home Page

      </Link>

      <Link to="/admin">Admin Panel</Link>

    </nav>

  );

};

export default NavBar;

This will show two links at the top of the page.

**3. Set Up Routing in App.tsx**

Open src/App.tsx and configure the routes using <Routes> and <Route> from react-router-dom.

Also import the two page components you want to route to — HomePage and AdminPage.

import { BrowserRouter, Routes, Route } from "react-router-dom";

import NavBar from "./Components/NavBar";

import HomePage from "./Components/HomePage";

import AdminPage from "./Components/AdminPage";

import "./App.css";

function App() {

  return (

    <BrowserRouter>

      <NavBar />

      <Routes>

        <Route path="/" element={<HomePage />} />

        <Route path="/admin" element={<AdminPage />} />

      </Routes>

    </BrowserRouter>

  );

}

export default App;

**✅ 4. Create Placeholder Pages**

If you don’t have them already, create two simple files:

**HomePage.tsx**

const HomePage = () => {

  return (

    <>

      <h1>Home Page</h1>

      <ul>

        <li> View all pets in the system</li>

        <li> Track vaccination records</li>

        <li> Add and manage medical histories</li>

      </ul>

    </>

  );

};

export default HomePage;

**AdminPage.tsx**

import PetTable from "./PetTable";

const AdminPage = () => {

  return (

    <>

      <h1>Admin Page</h1>

      <PetTable />

    </>

  );

};

export default AdminPage;

**Button.tsx**

interface IProps {

  name: string;

  caption: string;

  callFunc?: () => void;

  buttonClass?:string;

}

const Button = (props: IProps) => {

  return (

    <>

      <button

        className={props.buttonClass}

        type="button"

        name={props.name}

        onClick={props.callFunc}

      >

        {" "}

        {props.caption}{" "}

      </button>

    </>

  );

};

export default Button;

**SearchBar.tsx**

import { useState } from "react";

import Button from "./Button";

interface SearchBarProps {

  onShowAll: () => void;

  onSearchByName: (name: string) => void;

}

const SearchBar: React.FC<SearchBarProps> = ({ onShowAll, onSearchByName }) => {

  const [searchTerm, setSearchTerm] = useState("");

  return (

    <div className="search-bar">

      <Button

        name="all-pet-button"

        caption="Get All Pets"

        callFunc={onShowAll}

        buttonClass="button-5"

      />

      <input

        type="text"

        placeholder="Search pet by name"

        value={searchTerm}

        onChange={(e) => setSearchTerm(e.target.value)}

        className="search-input"

      />

      <Button

        name="search-pet-button"

        caption="Search"

        callFunc={() => onSearchByName(searchTerm)}

        buttonClass="button-5"

      />

    </div>

  );

};

export default SearchBar;

**PetTable.tsx**

import { useState } from "react";

import SearchBar from "./SearchBar"; // Adjust path if needed

interface Owner {

  name: string;

}

interface Pet {

  id: string;

  name: string;

  owner: Owner;

}

const PetTable = () => {

  const [pets, setPets] = useState<Pet[]>([]);

  const showPets = async () => {

    const response = await fetch("http://localhost:5120/pets");

    const data = await response.json();

    setPets(data);

  };

  const searchByName = async (name: string) => {

    const response = await fetch(`http://localhost:5120/pets/by-name/${name}`);

    const data = await response.json();

    setPets(data);

  };

  return (

    <div className="pet-table-container">

      <SearchBar onShowAll={showPets} onSearchByName={searchByName} />

      <div className="table-container">

        <table className="styled-table">

          <thead>

            <tr>

              <th>Pet Unique Id</th>

              <th>Pet Id</th>

              <th>Pet Name</th>

              <th>Pet Owner</th>

            </tr>

          </thead>

          <tbody id="pet-table-body">

            {pets.map((pet, index) => (

              <tr key={index}>

                <td>{pet.id}</td>

                <td>{index}</td>

                <td>{pet.name}</td>

                <td>{pet.owner.name}</td>

              </tr>

            ))}

          </tbody>

        </table>

      </div>

    </div>

  );

};

export default PetTable;

**5. Test It Out**

Start your frontend:

npm run dev

* Go to http://localhost:5173/
* You should see the **Home Page**
* Click on **Admin Panel** to see the pet table view