GraphQL using ASP.NET Core

We are going to use the following nuget packages:

- HotChocolate
- HotChocolate.AspNetCore
- HotChocolate.AspNetCore.Playground
- HotChocolate.Types
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.Tools
- Npgsql.EntityFrameworkCore.PostgreSQL
- GraphQL.Client
- GraphQL.Client.Serializer.Newtonsoft

The first step will be to create the database. You can use code first or database first approach.

At the end you should create a table that matches the following POCO:

```
    public class Company
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public decimal Revenue { get; set; }
    }
```

I'm using PostgreSQL and Entity Framework Core as my ORM of choice. Which will work with the Npgsql.EntityFrameworkCore.PostgreSQL nuget package.

Next we'll need to create the following DbContext class:

```
    public class CompanyDbContext : DbContext
    {
        public CompanyDbContext(DbContextOptions<CompanyDbContext> options) : 
base(options)
        {

        }

        public DbSet<Company> Companies { get; set; }
    }
```

We'll need to add the **DbContext** in the **Startup.cs** **`ConfigureServices`** method with this line of code:

```
services.AddDbContext<CompanyDbContext>(options =>
options.UseNpgsql(Configuration.GetConnectionString("DefaultConnection")));
```

We'll need to create a basic repository class to interact with the Db. We'll call it **CompanyService**.

Also create interface ICompanyService and two input classes CreateCompanyInput and DeleteCompanyInput.

```
    public interface ICompanyService
    {
        IQueryable<Company> GetAll();
        Company Create(CreateCompanyInput inputCompany);
        Company Delete(DeleteCompanyInput inputCompany);
    }

    public class CompanyService : ICompanyService
    {
        private readonly CompanyDbContext _context;

        public CompanyService(CompanyDbContext context)
        {
            _context = context;
        }

        public IQueryable<Company> GetAll()
        {
            return _context.Companies;
        }

        public Company Create(CreateCompanyInput inputCompany)
        {
            var company = _context.Companies.Add(new Company {Name = inputCompany.Name,
Revenue = inputCompany.Revenue});
            _context.SaveChanges();
            return company.Entity;
        }

        public Company Delete(DeleteCompanyInput inputCompany)
        {
            var company = _context.Companies.Find(inputCompany.Id);
            _context.Companies.Remove(company);
            _context.SaveChanges();
            return company;
        }
    }


    public class DeleteCompanyInput
    {
        public int Id { get; set; }
    }

    public class CreateCompanyInput
    {
        public string Name { get; set; }
        public decimal Revenue { get; set; }
    }
```

We’ll now create the classes that are need to create the GraphQL server.

CompanyType, Mutation and Query.

The CompanyType class will describe the company data type in GraphQL.

```
using HotChocolate.Types;
using HotChocolateSample.Core;

namespace HotChocolateSample.GraphQL
{
    public class CompanyType : ObjectType<Company>
    {
        protected override void Configure(IObjectTypeDescriptor<Company> descriptor)
        {
            descriptor.Field(a => a.Id).Type<IdType>();
            descriptor.Field(a => a.Name).Type<StringType>();
            descriptor.Field(a => a.Revenue).Type<DecimalType>();
        }
    }
```

The Query class we’ll be used to define the queries for GraphQL.

```
using System.Linq;
using HotChocolate.Types;
using HotChocolateSample.Core;

namespace HotChocolateSample.GraphQL
{
    public class Query
    {
        private ICompanyService _companyService;

        public Query(ICompanyService companyService)
        {
            _companyService = companyService;
        }
        [UsePaging(typeof(CompanyType))]
        [UseFiltering]
        public IQueryable<Company> Companies => _companyService.GetAll();
    }
```

Also we'll describe a mutation class to deifne the possible CRUD operations that we can execute using GraphQL.

```
using HotChocolateSample.Core;

namespace HotChocolateSample.GraphQL
{
    // sample use:
    /*mutation {
        createCompany(inputCompany: {
            name: "Tnuva",
            revenue: 210000
        })
        {
            id
            name
            revenue
        }
    }
    */

    public class Mutation
    {
        private readonly ICompanyService _companyService;

        public Mutation(ICompanyService companyService)
        {
            _companyService = companyService;
        }

        public Company CreateCompany(CreateCompanyInput inputCompany)
        {
            return _companyService.Create(inputCompany);
        }

        // sample use:
        /*mutation {
            createCompany(inputCompany: {
                name: "Tnuva",
                revenue: 210000
            })
            {
                id
                name
                revenue
            }
        }
        */
        //test
        public Company DeleteCompany(DeleteCompanyInput inputCompany)
        {
            return _companyService.Delete(inputCompany);
        }

        // sample use:
        /*
          mutation {
              deleteCompany(inputCompany: {
                id: 71
              })
              {
                id
                name
                revenue
              }
            }
         */
    }
}
```

The next step will be to configure GraphQL in the project. We'll do so by adding couple of lines of code in Startup.cs.

Update the ConfigureServices method like so:

```
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddDbContext<CompanyDbContext>(options =>

options.UseNpgsql(Configuration.GetConnectionString("DefaultConnection")));



            services.AddScoped<ICompanyService, CompanyService>();

            services.AddScoped<Query>();
            services.AddScoped<Mutation>();
            services.AddGraphQL(s => SchemaBuilder.New()
                .AddServices(s)
                .AddType<CompanyType>()
                .AddQueryType<Query>()
                .AddMutationType<Mutation>()
                .Create());
            services.AddControllersWithViews();
        }
```

```
And also update the Configure method like so:
```

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
                app.UsePlayground();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                // The default HSTS value is 30 days. You may want to change this for
production scenarios, see https://aka.ms/aspnetcore-hsts.
                app.UseHsts();
            }
            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Company}/{action=Index}/{id?}");

                endpoints.MapGraphQL();
            });

        }
```

We added the needed services in the ConfigureServices method. And also configured GraphQL using the AddGraphQL extension method.

Then we added app.UsePlayground() and endpoints.MapGraphQL() in the Configure method. This is enough to have a running GraphQL server. If you run the app and go to localhost:[port]/graphql. You'll be able to run GraphQL queries and mutations in the GraphQL IDE provided by Hotchocolate package.

After we have our Server up and running we'll create a simple Controller called CompanyController to test the functionality of this server. I'll do it by creating a controller and using the scaffolding features provided by ASP.NET Core MVC.

The index action will look as follows:

```
public async Task<ActionResult> Index()
        {
            var query = @"
                query {
                    companies {
                        nodes {
                            id
                            name
                            revenue
                        }
                    }
                }
            ";
              HttpClient client = new HttpClient{BaseAddress = new
              Uri(BackEndConstants.GraphQLUrl)};
            var response = await client.GetStringAsync($"?query={query}");
            var json = JObject.Parse(response);
            var companiesJson = json["data"]["companies"]["nodes"];
            List<Company> companies = new List<Company>();
            foreach (var obj in companiesJson)
            {
                companies.Add(new Company()
                {
                    Id = int.Parse(obj["id"].ToString()),
                    Name = obj["name"].ToString(),
                    Revenue = decimal.Parse(obj["revenue"].ToString())
                });
            }
            return View(companies);
        }
```

Here I'm using HttpClient to query for companies and I have the GraphQL server address stored as a constant BackEndConstants.GraphQLUrl in the class BackEndConstants.

The View for the action will be as follows:

```
@model IEnumerable<HotChocolateSample.Core.Company>

@{
    ViewData["Title"] = "Index";
}

<h1>Index</h1>
```

```
<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            <th>
                @Html.DisplayNameFor(model => model.Id)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Name)
            </th>
            <th>
                @Html.DisplayNameFor(model => model.Revenue)
            </th>
            <th></th>
        </tr>
    </thead>
    <tbody>
@foreach (var item in Model) {
        <tr>
            <td>
                @Html.DisplayFor(modelItem => item.Id)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Name)
            </td>
            <td>
                @Html.DisplayFor(modelItem => item.Revenue)
            </td>
            <td>
                @Html.ActionLink("Edit", "Edit", new { /* id=item.PrimaryKey */ }) |
                @Html.ActionLink("Details", "Details", new { /* id=item.PrimaryKey */ })
|
                @Html.ActionLink("Delete", "Delete", new { id=item.Id })
            </td>
        </tr>
}
    </tbody>
</table>
```

And as for the Create action:

```
[HttpPost]
        [ValidateAntiForgeryToken]
        public async Task<ActionResult> Create(IFormCollection collection)
        {
            string mutation = $@"
                mutation {{
                    createCompany(inputCompany: {{
                        name: ""{collection["name"]}"",
                        revenue: {collection["revenue"]}
                    }})
                    {{
                        id
                        name
                        revenue
                    }}
                }}
            ";

            GraphQLHttpClient client = new
            GraphQLHttpClient(BackEndConstants.GraphQLUrl,new
            NewtonsoftJsonSerializer());
            GraphQLHttpRequest request = new GraphQLHttpRequest(mutation);
            await client.SendMutationAsync<Company>(request);

            try
            {
                return RedirectToAction(nameof(Index));
            }
            catch
            {
                return View();
            }
        }
```

Here I'm using GraphQLHttpClient to send the createCompany mutation.

The create view will look as follows:

```
@model HotChocolateSample.Core.Company

@{
    ViewData["Title"] = "Create";
}

<h1>Create</h1>

<h4>Company</h4>
<hr />
<div class="row">
    <div class="col-md-4">
        <form asp-action="Create">
            <div asp-validation-summary="ModelOnly" class="text-danger"></div>
            <div class="form-group">
                <label asp-for="Name" class="control-label"></label>
                <input asp-for="Name" class="form-control" />
                <span asp-validation-for="Name" class="text-danger"></span>
            </div>
            <div class="form-group">
                <label asp-for="Revenue" class="control-label"></label>
                <input asp-for="Revenue" class="form-control" />
                <span asp-validation-for="Revenue" class="text-danger"></span>
            </div>
            <div class="form-group">
                <input type="submit" value="Create" class="btn btn-primary" />
            </div>
        </form>
    </div>
</div>

<div>
    <a asp-action="Index">Back to List</a>
</div>

@section Scripts {
    @{await Html.RenderPartialAsync("_ValidationScriptsPartial");}
}
```

You can view the full code of the project at: https://github.com/rezomegrelidze/HotChocolateSample