Building a real-time .NET GraphQL Client API

Michael Staib · November 25, 2019 · 21 min read

strawberry-shake graphql dotnet aspnetcore

We are busy, busy, busy working on version 11 of Hot Chocolate and *Strawberry Shake*.

In this post I want to explore the client side of GraphQL on .NET more with a special emphasis on subscriptions.

Since, with the new version of *Strawberry Shake* our initial blog has become kind of invalid I also will walk you trough the basics again before heading into subscriptions and what lies beyond.

Getting Started

Let us have a look at how we want to tackle things with *Strawberry Shake*. For this little walk-through I will use our *Star Wars* server example.

If you want to follow along then install the .NET Core 3 SDK . We are also supporting other .NET variants but for this example you will need the .NET Core 3 SDK.

Before we can start let us clone the Hot Chocolate repository and start our *Star Wars* server.

BASH

```
git clone https://github.com/ChilliCream/hotchocolate.git
cd hotchocolate
dotnet run --project examples/AspNetCore.StarWars/
```

Now that we have our *Star Wars* server running, lets create a folder for our client and install the *Strawberry Shake* CLI tools. The *Strawberry Shake* CLI tools are

optional but make initializing the client project much easier.

BASH

```
mkdir berry
dotnet new tool-manifest
dotnet tool install StrawberryShake.Tools --version 11.0.0-preview.58 --local
```

In our example we are using the new .NET CLI local tools. `dotnet new tool-manifest` creates the tools manifest which basically is like a packages.config and holds the information of which tools in which version we are using in our directory.

This is the great thing about local tools if you think about it, you can install tools to your repository and have always the right set of tools available to you in the moment you clone that repository.

The next command `dotnet tool install StrawberryShake.Tools --version 11.0.0-preview.58 --local` installs our *Strawberry Shake* tools. Once we have a final release of *Strawberry Shake* you do not need to speciefy the version anymore.

Next we need a little project. Let's create a new console application so that we can easily run and debug what we are doing.

BASH

```
dotnet new console -n BerryClient
cd BerryClient
dotnet add package StrawberryShake --version 11.0.0-preview.58
dotnet add package Microsoft.Extensions.Http --version 3.0.0
dotnet add package Microsoft.Extensions.DependencyInjection --version 3.0.0
```

OK, now that we have a project setup lets initialize the project by creating a local schema. Like with *Relay* we are holding a local schema file that can be extended with local types and fields. Our *GraphQL* compiler will use this schema information to validate the queries we write. This makes *GraphQL* query documents a part of the compilation process and with that a first-class citizen of our C# library.

For the next step ensure that the *Star Wars GraphQL* server is running since we will fetch the schema from the server.

If you want to check out what commands are available with the tools just run `dotnet graphql` and the CLI tools will output the available commands.

BASH

```
dotnet graphql init http://localhost:5000/graphql -n StarWars -p ./StarWars
```

The init command will download the schema as GraphQL SDL and create a config to re-fetch the schema. Also, the config contains the client name. The client name defines how the client class and interface shall be named.

Note: You can pass in the token and scheme if your endpoint is authenticated. There is also an update command to update the local schema.

The configuration will look like the following:

JSON

```
{
  "Schemas": [
    {
      "Name": "StarWars",
      "Type": "http",
      "File": "StarWars.graphql",
      "Url": "http://localhost:5000/graphql"
    }
  ],
  "ClientName": "StarWarsClient"
}
```

OK, now let's get started by creating our first client API. For this open your editor of choice. I can recommend using *VSCode* for this at the moment since you will get GraphQL highlighting. As we move forward, we will refine the tooling and provide proper IntelliSense.

Now let us create a new file in our `StarWars` folder called `Queries.graphql` and add the following query:

The file does not necessarily have to be called queries. You can call it however you want. The GraphQL compiler will figure out what files contain queries and what files contain schema definitions.

```graphql
query getFoo {
  foo
}
```

Now build your project.

```bash
dotnet build
```

When we now compile, we get an *MSBuild* error on which we can click in *VSCode*and we are pointed to the place in our query file from which the error stems from. The error tells us that there is no field `foo` on the `Query` type.

```bash
/Users/michael/Local/play/berry/BerryClient/StarWars/Queries.graphql(2,3): err
```

Your GraphQL query document is not just a string, it properly compiles and is fully typed. Let's change our query and compile again:

```graphql
query getFoo {
  hero {
    name
  }
}
```

```bash
dotnet build
```

Now our project changes and we get a new `Generated` folder that has all the types that we need to communicate with our backend.

Let us have a closer look at our client interface for a minute.

```csharp
public interface IStarWarsClient
{
    Task<IOperationResult<IGetFoo>> GetFooAsync(
        CancellationToken cancellationToken = default);
}
```

The named operation `getFoo` has become the method `GetFooAsync` in our generated client. This is nice since we kind of control from our GraphQL document the shape of our C# API. But there is more to that. A query document can hold multiple named operations. In essence the query document describes the interface between the client and the server.

```graphql
query function_a {
  ...
}

query function_b {
  ...
}
```

```
query function_c {
  ...
}
```

Since, with GraphQL you essentially design your own service API by writing a query document you want to have control over the structure of your generated types. *Strawberry Shake* uses fragments to help you describe clean and reusable code components.

Let us redesign our query with fragments and make it a bit more complex.

GRAPHQL

```
query getHero {
  hero {
    ...SomeDroid
    ...SomeHuman
  }
}

fragment SomeHuman on Human {
  ...HasName
  homePlanet
}

fragment SomeDroid on Droid {
  ...HasName
  primaryFunction
}

fragment HasName on Character {
  name
}
```

The fragments will yield in the following type structure:

C#

```
public interface ISomeHuman
    : IHasName
{
    string HomePlanet { get; }
}
```

```
public interface ISomeDroid
    : IHasName
{
    string PrimaryFunction { get; }
}

public interface IHasName
{
    string Name { get; }
}
```

Let us reflect on that, fragments not only let us re-use type selections in our query document but also let us create and mold our C# API into a clean type structure. This puts us as the consumer of *Strawberry Shake* in the driver seat.

We decide what data we **need** and how they are **shaped**.

We are currently looking into how we can aggregate data and flatten the type structure. We initially thought about introducing some directives to flatten the type structure. But as we thought further on that and we really felt we want to have something like lodash. We are still discussing on what we want to do here. So stay tuned.

Let's make one more tweak to our query and then we get this example running.

GRAPHQL

```
query getHero($episode: Episode) {
  hero(episode: $episode) {
    ...SomeDroid
    ...SomeHuman
  }
}

fragment SomeHuman on Human {
  ...HasName
  homePlanet
}
```

```
fragment SomeDroid on Droid {
  ...HasName
  primaryFunction
}

fragment HasName on Character {
  name
}
```

By defining a variable with our operation we now can pass in arguments. This makes our operation re-usable and a good interface with the server. GraphQL servers can pre-compile and optimize those parametrized query documents.

C#

```
public interface IStarWarsClient
{
    Task<IOperationResult<IGetHero>> GetHeroAsync(
        Optional<Episode> episode = default,
        CancellationToken cancellationToken = default);
}
```

OK, let's get it running and then go into more details.

By default the generator will also generate dependency injection code for `Microsoft.Extensions.DependencyInjection`.

In order to get our client up and running we just have to set up a dependency injection container.

Note: You can shut off dependency injection generation with a *MSBuild* property. The client can also be instantiated with a builder or by using a different dependency injection container.

Replace your `Program` class with the following code.

C#

```
class Program
{
    static async Task Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddHttpClient(
            "StarWarsClient",
            c => c.BaseAddress = new Uri("http://localhost:5000/graphql"));
        serviceCollection.AddStarWarsClient();

        IServiceProvider services = serviceCollection.BuildServiceProvider();
        IStarWarsClient client = services.GetRequiredService<IStarWarsClient>(

        IOperationResult<IGetHero> result = await client.GetHeroAsync(Episode.
        Console.WriteLine(((ISomeDroid)result.Data.Hero).Name);

        result = await client.GetHeroAsync(Episode.Empire);
        Console.WriteLine(((ISomeHuman)result.Data.Hero).Name);
    }
}
```

Run the console and it will output the following;

```
BASH
```

```
R2-D2
Luke Skywalker
```

That is quite awesome. The client is easy to setup and easy to use. We just had to initialize our project and write a GraphQL query document and everything was generated so that we can focus on using our GraphQL endpoint instead of writing a bunch of code that we do not want to actually write. Moreover, we only get the types from the schema that we actually use in our query documents, that means we are not burdened with all the schema types and fields and so on that we do not need and do not want.

Strawberry Shake let`s you take all the power of GraphQL and package it up into a fully typed client that works well with .NET. It does not limit you by introducing a

new programming model like Linq or some other .NET API, instead *Strawberry Shake* makes **GraphQL a first class citizen in the .NET world**.

OK, now let us have a look at the result object since we also carefully discussed how we expose results to the consumer.

The result of an operation can be a `IOperationResult<T>` or a `IResponseStream<T>`.

The operation result represents a single result and we expose the GraphQL result structure as specified in the GraphQL spec.

This means that we give you a chance to take advantage of partial results in case of errors. However, we also make it easy to raise an exception in case of any error with the `EnsureNoErrors` method on the result object. This is kind of like with responses from a `HttpClient`.

Also, we allow you to have full access to provider specific data that is included in a dictionary called `Extensions`. This for instances is used in cases like active persisted queries or other provider specific extensions.

Renaming Type Elements

Did you notice the enum type `Episode.Newhope` in the upper example. This really is not nice to see as a C# developer :). Since the generator is built on top of the stitching API we easily can amend things like that in order to make our client API nice to use.

So, before we go into subscriptions let`s fix that :)

First, add another GraphQL file and call it `StarWars.Extensions.graphql`. Again, the name does not really matter, you could call it `Foo.graphql` and *Strawberry Shake*would also handle it correctly.

GraphQL allows to extend types with the `extend` keyword in the GraphQL SDL. In the example below we extend the `Episode` enum and add a directive (annotation) called `@name`. The `@name` directive allows us to provide the

generator with a name for a type element that we actually want to use in our C# client API.

Now add the following type extension to the GraphQL file `StarWars.Extensions.graphql`:

```graphql
extend enum Episode {
  NEWHOPE @name(value: "NewHope")
}
```

Rebuild your project and voila ... `Episode.NewHope` is now correctly cased.

The nice thing is that we are just describing what we want to change in this schema extension file, so every time you update the server schema, we will preserve this file and reapply the type extensions to the newly downloaded schema.

Subscriptions

OK, OK, most of this was already in place, so let us have a look at something more challenging like subscriptions.

Subscriptions will need a state-full connection to a server through a WebSocket. There are other ways to do this like SignalR (which essentially is a socket abstraction) or gRPC or even over a standard TCP socket.

While we are in the works to get SignalR and gRPC in let us have a look at how we can do it through WebSockets.

When we started on this we found that WebSockets should be as easy as setting up the HttpClient nowadays. So, we have introduced a new interface called `IWebSocketClientFactory`. But just having a factory is not enough since we want to maybe pool socket connections and reuse those with multiple subscriptions.

With the solution that we are introducing with version 11.0.0-preview.58 we are making WebSockets super simple to setup, and we will do all the hard parts like reusing the connection and things like that without you ever noticing it.

Let us have a look at how we can get subscriptions to work.

The first thing we have to do is going back to our query document. The *Star Wars*server has one subscription that is raised whenever a review is written. So, let's use it and add it to our query file.

GRAPHQL

```
query getHero($episode: Episode!) {
  hero(episode: $episode) {
    ...SomeDroid
    ...SomeHuman
  }
}

subscription onReviewCreated(episode: $episode) {
  onReview(episode: $episode) {
    commentary
    stars
  }
}

fragment SomeHuman on Human {
  ...HasName
  homePlanet
}

fragment SomeDroid on Droid {
  ...HasName
  primaryFunction
}

fragment HasName on Character {
  name
}
```

Now, lets rebuild our project and then look at the client interface.

C#

```
public interface IStarWarsClient
{
    Task<IOperationResult<IGetHero>> GetHeroAsync(
        Optional<Episode> episode = default,
        CancellationToken cancellationToken = default);

    Task<IResponseStream<IOnReviewCreated>> OnReviewCreatedAsync(
        Optional<Episode> episode = default,
        CancellationToken cancellationToken = default);
}
```

Our client has now a new method that returns a response stream. A response stream is essentially an `IAsyncEnumerable` that will loop over the subscription event stream until the stream completes or the client disposes the stream.

Now let us put everything together. First we need to configure the WebSocket client connection.

C#

```
services.AddWebSocketClient(
    "StarWarsClient",
    c => c.Uri = new Uri("ws://localhost:5000/graphql"));
```

This kind of looks exactly the way we would configure an HttpClient and it hides all the complex logic about connecting and pooling WebSocket connections. It also lets you easily intercept the connect process to include authentication logic.

The next thing we need to do to consume data from subscriptions is to read from our event stream.

C#

```
class Program
{
    static async Task Main(string[] args)
    {
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddHttpClient(
            "StarWarsClient",
            c => c.BaseAddress = new Uri("http://localhost:5000/graphql"));
```

```
            serviceCollection.AddStarWarsClient();

            IServiceProvider services = serviceCollection.BuildServiceProvider();
            IStarWarsClient client = services.GetRequiredService<IStarWarsClient>(

            var stream = await client.OnReviewCreatedAsync(Episode.NewHope);

            await foreach (var result in stream)
            {
                result.EnsureNoErrors();
                Console.WriteLine(result.Data!.OnReview.Commentary);
            }
        }
    }
```

If you look at the code above it looks so easy how you can use subscription with *Strawberry Shake*, it almost looks no different from fetching a simple query with the `HttpClient`. This is exactly what we want the experience to be, simple but when you want to get into the pluming then we will allow you to easily intercept and extend the whole pipeline.

So, in order to try subscriptions out in your example open a tool like playground and the fire the following query against the local GraphQL Server while your console app is running.

GRAPHQL

```
mutation {
  createReview(
    episode: NEWHOPE
    review: { commentary: "Awesome movie.", stars: 5 }
  ) {
    commentary
    stars
  }
}
```

As soon as you trigger the above mutation the client will print the commentary to the console, it is kind of like magic :)

Custom Scalars

The mean thing with all these examples that I posted in this blog is that I am only using the *Star Wars* example. The *Star Wars* uses no custom scalars and is super simple to use. That is the reason why I like to use it for demos, because people get easily on board with it. But it also is frustrating when you want to go deeper. This is especially true with custom scalars.

Strawberry Shake supports an array of built-in scalars that go beyond the GraphQL spec. But still if you download the GitHub schema for instance you will get a ton of custom scalars.

With the current version we have made dealing with custom scalars a lot easier. First, if we do not know a scalar, then we will treat it as a `String`. While this is not always what you want, it lets you get started quickly and then change things when you really need them to change.

Let us have a look at how we can bring in a custom scalar. For this example, let us assume we have a scalar called `ByteArray`. This scalar serializes a `System.Byte[]` to a base64 string. This is easy enough. So on the client side we want the generator to generate models that expose `System.Byte[]` as property type. But in the communication between server and client the type shall be serialized as base64 string.

So, in order to give the generator a hint about these things we need to extend our schema. We would need to create a GraphQL file that holds our schema extensions (basically like with the enum example, where we renamed the enum value). The same way we can extend enums we can extend other types. In this case we want to annotate a scalar type.

GRAPHQL

```
extend scalar ByteArray
  @runtimeType(name: "System.Byte[]")
  @serializationType(name: "System.String")
```

The above example declares that for the `ByteArray` scalar the runtime type (the type that is used in the C# models) shall be a `System.Byte[]` and that the serialization type (the type which client and server use to send the data) shall be a `System.String`. For the generator that is enough to generate everything accordingly.

We still have to implement an `IValueSerializer` to specify the logic how the type shall actually serialize and deserialize.

C#

```
public class ByteArrayValueSerializer
    : ValueSerializerBase<byte[], string>
{
    public override string Name => "ByteArray";

    public override ValueKind Kind => ValueKind.String;

    public override object? Serialize(object? value)
    {
        if (value is null)
        {
            return null;
        }

        if (value is byte[] b)
        {
            return Convert.ToBase64String(b);
        }

        throw new ArgumentException(
            "The specified value is of an invalid type. " +
            $"{ClrType.FullName} was expeceted.");
    }

    public override object? Deserialize(object? serialized)
    {
        if (serialized is null)
        {
            return null;
        }

        if (serialized is string s)
        {
            return Convert.FromBase64String(s);
        }
```

```
            throw new ArgumentException(
                "The specified value is of an invalid type. " +
                $"{SerializationType.FullName} was expeceted.");
        }
    }
```

The serializer can be added as a singleton and will be automatically integrated by the generated client.

C#

```
services.AddSingleton<IValueSerializer, ByteArrayValueSerializer>();
```

We are refining how those serializers are registered. This is important for cases where one wants to have multiple clients with different kinds of serializers. I know this is rare but still this should work. The coming versions of *Strawberry Shake* will fine tune this.

Digging Deeper

Apart from being able to add custom scalars we might want to dig deeper and allow new scenarios with our client like persisted queries. It is needles to say that we will add persisted query support out of the box. But it is also a good example to use to show how we can enable advance server / client protocols with *Strawberry Shake*.

The way we built-in things like that is by providing a operation middleware. This basically works like the query middleware in the server on the request level.

Strawberry Shake allows us to swap out the default operation execution pipeline and add our own custom operation execution pipeline.

In order to setup a custom operation execution pipeline you can use for instance the `HttpPipelineBuilder`. Each transport has it`s own transport specific

pipeline since the protocol between socket communication and stateless communication is quite different.

C#

```
serviceCollection.AddSingleton<OperationDelegate>(
    sp => HttpPipelineBuilder.New()
        .Use<CreateStandardRequestMiddleware>()
        .Use<CustomMiddleware>()
        .Use<SendHttpRequestMiddleware>()
        .Use<ParseSingleResultMiddleware>()
        .Build(sp));
```

C#

```
public class CustomMiddleware
{
    private readonly OperationDelegate _next;
    private readonly IOperationSerializer _service;

    public CustomMiddleware(
        OperationDelegate next,
        ISomeCustomService service)
    {
        _next = next ?? throw new ArgumentNullException(nameof(next));
        _service = service ?? throw new ArgumentNullException(nameof(service))
    }

    public async Task InvokeAsync(IHttpOperationContext context)
    {
        // the custom middleware code
        await _next(context);
    }
}
```

Generation Options

By default *Strawberry Shake* generates dependency injection code for `Microsoft.Extensions.DependencyInjection` this can be switched of by adding the following `MSBuild` property `<GraphQLEnableDI>false</GraphQLEnableDI>`.

The generator will automatically detect if you are using C# 8.0 with nullable reference types or if you are using an older version of C#.

You can use the following `MSBuild` properties to control this.

```XML
<PropertyGroup>
  <LangVersion>8.0</LangVersion>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

We also by default take the root namespace from the project for generating files. You can however override this by providing the `<BerryNamespace />` property. However, we will change this to an item group soon in order to also enable multiple clients in a single project to use different namespaces.

```XML
<PropertyGroup>
  <BerryNamespace>$(RootNamespace)</BerryNamespace>
</PropertyGroup>
```

Dependency Injection

The client API can be used with other dependency injection container and also without dependency injection at all.

We initially had a limited builder API for this but decided to give it a do over. So, at the moment you could look at the generated dependency injection code and build your own integration.

We will allow with future build to add custom generators that can provide additional code for custom use cases. The way that would work is that such a generator would sit in a NuGet package that is being added to the project. The custom generator would register its generators to an item group and *Strawberry Shake*would pick those up and integrate them. These custom generators however are somewhere in the version 12 timeframe.

Roadmap

What are our next steps on *Strawberry Shake* and when are we planning to release it?

We have some more ground to cover before we have this version complete.

1. MSBuild Integration We are working on making the *MSBuild* integration better. There are still instances with *VSCode* where you have to compile twice. This is OK for a preview,but we are on it and think that in the next view preview builds we will have this fixed. With *Visual Studio for Windows* you can already enjoy design time code generation. This means that when you save a GraphQL file the generator will update the C# files.
2. Tooling We are heavy at work on *Bananacake Pop* which is our GraphQL IDE that will help you write and analyze GraphQL queries. We plan to use what we have done for *Bananacake Pop* to create a nice integration with *VSCode*. We want to have a rich integration with which you can work on huge schemas.

   Beyond *VSCode* we are looking at writing a nice integration with *Visual Studio for Windows* and *Visual Studio for macOS* that will make *Strawberry Shake* and *GraphQL* a first-class citizen in Microsoft IDEs.

   We hope to deliver all of this in the version 11 timeframe.
3. Persisted Query Support Persisted queries are one of our very next features that we will add to the client. We want to allow the same flows that we support on the server side.
4. Batching Support Batching support with the `@export` directive is as well planned for the initial release of *Strawberry Shake*.
5. Code Generation The current code generation produces quite nice code, but it can produce issues where the types from your own project collide with the generated code. With the next view builds we will add an option to use full type names in those cases. Also, we will add the code generation attribute to the generated files. So there are refinements going on in this area.
6. Defer / Stream We are planning to add support for defer and stream to the client. This feature depends on our server implementation so we will have to see how

far we are on execution plans before we can start on it for the client.

I hope you enjoy what we are building. We are tying to bring GraphQL on .NET to the next level. While we still are miles away from what the JavaScript world has to offer we hope to close these gaps over the next year and even pull ahead in some areas. We love GraphQL and are passionate about it. We strongly believe that with our newest member *Strawberry Shake* we really can make things like *Xamarin* and *Blazor* so much better. We have planned a lot more and hope to bring data fetching in .NET to a whole new level over the next year. Ideally you just want to decalre in your .NET component the data that you need and all the client logic is infered from that, kind of the way relay works in JavaScript.

If you want to get into contact with us head over to our slack channel and join our community.