Steps for creating a new gRPC service in Trachea

0. Identify which services to convert to gRPC.

---

1. Creating the protobuffer

1.1 Translate the request and response models into a protobuf file. [Protobuf message file]

1.1.1 Protobufs should be stored in `Common.ProtoBuffers` with a directory name that is the same name as the microservice

> *e.g. for the EMR microservice, the directory is named EMR.*

1.1.2 Please use the correct naming conventions. Protobuf .proto files are snake case.

> *e.g. I would name my file `vida_authentication.proto`*

1.1.3 The `package` keyword will be the C# namespace of the compiled protobuf file. The convention I am currently using is the following: `package common.grpc.<MICROSERVICE_NAME>.<MODEL_NAME>`

> *e.g. `package common.grpc.emr.authentication`*

1.1.4 Request/Response messages (i.e. the models) for a given service method should be stored in its own protobuf file. For other service methods, store them in a separate protobuf file. This is for the sake of clarity; it has no effect on the performance or functionality.

1.1.5 **This part is up for discussion**. For the sake of clarity so we do not confuse models that we made with the models auto-generated by the gRPC compiler, all gRPC messages MUST be suffixed with a `_g` . Side-note: This convention is accepted in the ASP NET Core style guidelines, especially since these messages will be compiled into auto-generated C# code made by the gRPC compiler.

> *Instead of naming my message* `message AuthenticationRequest { ... }` *I will instead name it* `message AuthenticationRequest_g { ... }`

1.2 Create the protobuf service file. [Protobuf RPC file]

1.2.1 **This part is up for discussion**. All RPC services, per microservice, should be stored in a single protobuf file.

> *e.g. For all services in the EMR microservice, I made an* `emr_main.proto` *file.*

1.2.2 Since the RPC protobuf file does NOT have any messages, all needed messages must be imported.

> *e.g. In my* `emr_main.proto` *file, I need a message I stored inside of* `vida_authentication.proto` *. Therefore, I import the protobufer within* `emr_main.proto` *and call it through its fully qualified name. For example this fully qualified name would be* `.common.grpc.emr.authentication.AuthenticationRequest_g`

1.2.3 The `package` keyword will be the C# namespace of the compiled protobuf file. The convention I am currently using is the following: `package common.grpc.<MICROSERVICE_NAME>.<SERVICE_NAME>`

1.3 Setting up `Common.csproj` file to auto-generate compiled protobuf files.

1.3.1 For protobuffers that **only contain messages**, add the following line of code:

```
<ItemGroup>
    ...
    <Protobuf Include="ProtoBuffers\<MICROSERVICE_NAME>\<SERVICE_NAME>\*.proto" CompileOu
tputs="true" GrpcServices="None" />
    ...
</ItemGroup>
```

This will ensure that all protobufs in this directory are compiled immediately after saving. Side-note: the auto-generated code can be found in the *obj* directory of the project.

1.3.2 For protobufs that **only contain RPC services**, the step is similar to 1.3.1, but we need to set the `GrpcServices` to `Both` instead of `None`:

```
<ItemGroup>
    ...
    <Protobuf Include="ProtoBuffers\<MICROSERVICE_NAME>\<SERVICE_NAME>\*.proto" CompileOu
tputs="true" GrpcServices="Both" />
    ...
</ItemGroup>
```

1.3.3 **Note that sometimes the IDE does not immediately build the gRPC files. In order fix this, manually build the `Common` project and that should resolve the issue.** If this does not resolve the issue still, restart the Visual Studio.

---

2. Creating the gRPC service in API gateway

2.1 Create gRPC service that implements the interface associated with the relevant microservice

2.1.1 Similar to RESTful service implementations, this follows virtually all of the same steps, but we will register the gRPC clients via dependency injection instead of using `HttpClient`. Note that the gRPC client (and base) are auto-generated by the gRPC compiler (only if the protobuf files are valid). This gRPC service client implementation will serve as the gateway between the API gateway and the microservice that implements these gRPC services.

> *gRPC client --> auto-generated by gRPC compiler. Needed for API gateway.*
> *gRPC base --> auto-generated by gRPC compiler. Needed for the microservice.*
> *gRPC service client --> you make this and it uses the gRPC client via dependency injection*

2.1.2 Register the gRPC service client in `Startup.cs`. This can be done as either `transient` or `AddHttpClient` if you require the HTTP client for other sources that are not using gRPC.

> *e.g. `services.AddHttpClient<IEmrService, EmrGrpcService>(client => { ... });`*

2.1.3 Register the gRPC clients in order for the dependency injection to work properly. All options (i.e. URI, SSL certificates, etc...) is configured here. Dummy SSL certs are required since gRPC is designed only to transmit and receive via HTTPS/2.

> *This can be done with the via the* `GrpcClientServiceExtensions` *methods with* `.AddGrpcClient` *. e.g.* `services.AddGrpcClient<AuthenticationServices.AuthenticationServicesClient>(options => { ... })`

2.1.4 Write the implementation code in the gRPC service client that calls the gRPC client which calls the microservice it is associated with. Please note at this point we have not finished the gRPC service base yet in the microservice. This is a later step.

---

3. Creating the gRPC implementation file within a microservice

3.1 Create gRPC service base implementation

3.1.1 Within the microservice, this is the implementation code for the gRPC service that is called from the API gateway. The naming convention for these files is typically `<Name>ServiceImpl.cs` . So for example, in the EMR microservice, one of the gRPC service bases is named `AuthenticationServiceImpl.cs` .

3.1.2 Within the gRPC service base implementation file, create a class that implements the auto-generated gRPC base class and make overrides to all the classes that are defined within it. Note: all these definitions come from your protobuf RPC file.

3.1.3 Every method comes with 2 parameters. First is the parameter is defined in the protobuf RPC file. This parameter is ALWAYS a class. Even primitive types like an integer must be made into a class via protobuf messaages. The second parameter is the `ServerCallContext` which carries all of the meta data for this request (such as HTTP headers).

3.2 Register gRPC service base

3.2.1 Register the gRPC service base in the relevant microservice's `Startup.cs` file. This is done in the `Configure` stage via `app.UseEndpoints` . For example:

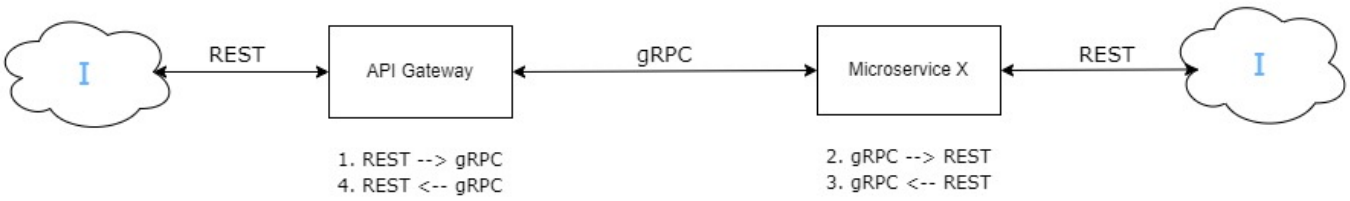
```
app.UseEndpoints(e => {
    e.MapGrpcService<AuthenticationServiceImpl>();
});
```

3.2.2 Enable gRPC in the microservice within the `Startup.cs` file via `services.AddGrpc()` in the `ConfigureServices` method. E.g.

```
services.AddGrpc(options => {
    options.EnableDetailedErrors = true;
});
```

3.2.3 At this point that project should be ready for testing.

Challenges



Challenges:

Mapping has to be done in 4 separate instances.

How can we do mapping automatically?

Mapping steps:

1. Request from REST to gRPC.
2. Request from gRPC to REST.
3. Response from REST to gRPC.
4. Response from gRPC to REST.

Useful links, guides, and articles

- Why use gRPC?

- gRPC scalar value types
- gRPC JSON mapping
- grpc-dotnet source code
- Microsoft guide on gRPC implementation with .NET 6
- Visual Studio 2022 - Mapping Generator <-- So helpful!!!