

---

## gRPC Elixir



An Elixir implementation of gRPC.

### Table of contents

- Installation
- Usage
  - Simple RPC
  - HTTP Transcoding
  - Start Application
- Features
- Benchmark
- Contributing

### Installation

The package can be installed as:

```
1 def deps do
2   [
3     {:grpc, "~> 0.7"},
4     # We don't force protobuf as a dependency for more
5     # flexibility on which protobuf library is used,
6     # but you probably want to use it as well
7     {:protobuf, "~> 0.11"}
8   ]
9 end
```

### Usage

1. Write your protobuf file:

```
1 syntax = "proto3";
2
3 package helloworld;
4
5 // The request message containing the user's name.
6 message HelloRequest {
```

---

```
7   string name = 1;
8 }
9
10 // The response message containing the greeting
11 message HelloReply {
12   string message = 1;
13 }
14
15 // The greeting service definition.
16 service Greeter {
17   // Greeting function
18   rpc SayHello (HelloRequest) returns (HelloReply) {}
19 }
```

2. Then generate Elixir code from proto file as `protobuf-elixir` shows (especially the [gRPC Support](#) section) or using `protobuf_generate` hex package. Example using `protobuf_generate` lib:

```
1 mix protobuf.generate --output-path=./lib --include-path=./priv/protos
  helloworld.proto
```

In the following sections you will see how to implement gRPC server logic.

## Simple RPC

1. Implement the server side code like below and remember to return the expected message types.

```
1 defmodule Helloworld.Greeter.Server do
2   use GRPC.Server, service: Helloworld.Greeter.Service
3
4   @spec say_hello(Helloworld>HelloRequest.t, GRPC.Server.Stream.t) ::
5     Helloworld>HelloReply.t
6   def say_hello(request, _stream) do
7     Helloworld>HelloReply.new(message: "Hello #{request.name}")
8   end
9 end
```

2. Define gRPC endpoints

```
1 # Define your endpoint
2 defmodule Helloworld.Endpoint do
3   use GRPC.Endpoint
4
5   intercept GRPC.Server.Interceptors.Logger
6   run Helloworld.Greeter.Server
7 end
```

---

We will use this module in the gRPC server startup section.

**Note:** For other types of RPC call like streams see here.

## HTTP Transcoding

1. Adding grpc-gateway annotations to your protobuf file definition:

```
1 import "google/api/annotations.proto";
2 import "google/protobuf/timestamp.proto";
3
4 package helloworld;
5
6 // The greeting service definition.
7 service Greeter {
8     // Sends a greeting
9     rpc SayHello (HelloRequest) returns (HelloReply) {
10         option (google.api.http) = {
11             get: "/v1/greeter/{name}"
12         };
13     }
14
15     rpc SayHelloFrom (HelloRequestFrom) returns (HelloReply) {
16         option (google.api.http) = {
17             post: "/v1/greeter"
18             body: "*"
19         };
20     }
21 }
```

2. Add protoc plugin dependency and compile your protos using protobuf\_generate hex package:

In mix.exs:

```
1 def deps do
2     [
3         {:grpc, "~> 0.7"},
4         {:protobuf_generate, "~> 0.1.1"}
5     ]
6 end
```

And in your terminal:

```
1 mix protobuf.generate \
2   --include-path=priv/proto \
3   --include-path=deps/googleapis \
4   --generate-descriptors=true \
5   --output-path=./lib \
6   --plugins=ProtobufGenerate.Plugins.GRPCWithOptions \
```

---

```
7 google/api/annotations.proto google/api/http.proto helloworld.proto
```

### 3. Enable http\_transcode option in your Server module

```
1 defmodule Helloworld.Greeter.Server do
2   use GRPC.Server,
3     service: Helloworld.Greeter.Service,
4     http_transcode: true
5
6   @spec say_hello(Helloworld>HelloRequest.t, GRPC.Server.Stream.t) ::
7     Helloworld>HelloReply.t
8   def say_hello(request, _stream) do
9     %Helloworld>HelloReply{message: "Hello #{request.name}"}
10  end
11 end
```

See full application code in `helloworld_transcoding` example.

## Start Application

### 1. Start gRPC Server in your supervisor tree or Application module:

```
1 # In the start function of your Application
2 defmodule HelloworldApp do
3   use Application
4   def start(_type, _args) do
5     children = [
6       # ...
7       {GRPC.Server.Supervisor, endpoint: Helloworld.Endpoint, port:
8         50051, start_server: true}
9     ]
10
11     opts = [strategy: :one_for_one, name: YourApp]
12     Supervisor.start_link(children, opts)
13   end
14 end
```

### 2. Call rpc:

```
1 iex> {:ok, channel} = GRPC.Stub.connect("localhost:50051")
2 iex> request = Helloworld>HelloRequest.new(name: "grpc-elixir")
3 iex> {:ok, reply} = channel |> Helloworld.Greeter.Stub.say_hello(
4   request)
5 # With interceptors
6 iex> {:ok, channel} = GRPC.Stub.connect("localhost:50051", interceptors
7   : [GRPC.Client.Interceptors.Logger])
8 ...
```

---

Check the examples and interop directories in the project's source code for some examples.

## Features

- Various kinds of RPC:
  - Unary
  - Server-streaming
  - Client-streaming
  - Bidirectional-streaming
- HTTP Transcoding
- TLS Authentication
- Error handling
- Interceptors (See [GRPC.Endpoint](#))
- Connection Backoff
- Data compression
- gRPC Reflection

## Benchmark

1. Simple benchmark by using ghz
2. Benchmark followed by official spec

## Contributing

Your contributions are welcome!

Please open issues if you have questions, problems and ideas. You can create pull requests directly if you want to fix little bugs, add small features and so on. But you'd better use issues first if you want to add a big feature or change a lot of code.