
gRPCurl



`grpcurl` is a command-line tool that lets you interact with gRPC servers. It's basically `curl` for gRPC servers.

The main purpose for this tool is to invoke RPC methods on a gRPC server from the command-line. gRPC servers use a binary encoding on the wire (protocol buffers, or “protobufs” for short). So they are basically impossible to interact with using regular `curl` (and older versions of `curl` that do not support HTTP/2 are of course non-starters). This program accepts messages using JSON encoding, which is much more friendly for both humans and scripts.

With this tool you can also browse the schema for gRPC services, either by querying a server that supports server reflection, by reading proto source files, or by loading in compiled “protoset” files (files that contain encoded file descriptor protos). In fact, the way the tool transforms JSON request data into a binary encoded protobuf is using that very same schema. So, if the server you interact with does not support reflection, you will either need the proto source files that define the service or need protoset files that `grpcurl` can use.

This repo also provides a library package, github.com/fullstorydev/grpcurl, that has functions for simplifying the construction of other command-line tools that dynamically invoke gRPC endpoints. This code is a great example of how to use the various packages of the `protorelect` library, and shows off what they can do.

See also the `grpcurl` talk at GopherCon 2018.

Features

`grpcurl` supports all kinds of RPC methods, including streaming methods. You can even operate bi-directional streaming methods interactively by running `grpcurl` from an interactive terminal and using stdin as the request body!

`grpcurl` supports both secure/TLS servers *and* plain-text servers (i.e. no TLS) and has numerous options for TLS configuration. It also supports mutual TLS, where the client is required to present a client certificate.

As mentioned above, `grpcurl` works seamlessly if the server supports the reflection service. If not, you can supply the `.proto` source files or you can supply protoset files (containing compiled descriptors, produced by `protoc`) to `grpcurl`.

Installation

Binaries

Download the binary from the releases page.

Homebrew (macOS)

On macOS, `grpcurl` is available via Homebrew:

```
1 brew install grpcurl
```

Docker

For platforms that support Docker, you can download an image that lets you run `grpcurl`:

```
1 # Download image
2 docker pull fullstorydev/grpcurl:latest
3 # Run the tool
4 docker run fullstorydev/grpcurl api.grpc.me:443 list
```

Note that there are some pitfalls when using docker: - If you need to interact with a server listening on the host's loopback network, you must specify the host as `host.docker.internal` instead of `localhost` (for Mac or Windows) *OR* have the container use the host network with `-network="host"` (Linux only). - If you need to provide proto source files or descriptor sets, you must mount the folder containing the files as a volume (`-v $(pwd) : /protos`) and adjust the import paths to container paths accordingly. - If you want to provide the request message via stdin, using the `-d @` option, you need to use the `-i` flag on the docker command.

Other Packages

There are numerous other ways to install `grpcurl`, thanks to support from third parties that have created recipes/packages for it. These include other ways to install `grpcurl` on a variety of environments, including Windows and myriad Linux distributions.

You can see more details and the full list of other packages for `grpcurl` at [repology.org](https://repology.org/project/grpcurl/information):
<https://repology.org/project/grpcurl/information>

From Source

If you already have the Go SDK installed, you can use the `go` tool to install `grpcurl`:

```
1 go install github.com/fullstorydev/grpcurl/cmd/grpcurl@latest
```

This installs the command into the `bin` sub-folder of wherever your `$GOPATH` environment variable points. (If you have no `GOPATH` environment variable set, the default install location is `$HOME/go/bin`). If this directory is already in your `$PATH`, then you should be good to go.

If you have already pulled down this repo to a location that is not in your `$GOPATH` and want to build from the sources, you can `cd` into the repo and then run `make install`.

If you encounter compile errors and are using a version of the Go SDK older than 1.13, you could have out-dated versions of `grpcurl`'s dependencies. You can update the dependencies by running `make updatedeps`. Or, if you are using Go 1.11 or 1.12, you can add `GO111MODULE=on` as a prefix to the commands above, which will also build using the right versions of dependencies (vs. whatever you may already have in your `GOPATH`).

Usage

The usage doc for the tool explains the numerous options:

```
1 grpcurl -help
```

In the sections below, you will find numerous examples demonstrating how to use `grpcurl`.

Invoking RPCs

Invoking an RPC on a trusted server (e.g. TLS without self-signed key or custom CA) that requires no client certs and supports server reflection is the simplest thing to do with `grpcurl`. This minimal invocation sends an empty request body:

```
1 grpcurl grpc.server.com:443 my.custom.server.Service/Method
2
3 # no TLS
4 grpcurl -plaintext grpc.server.com:80 my.custom.server.Service/Method
```

To send a non-empty request, use the `-d` argument. Note that all arguments must come *before* the server address and method name:

```
1 grpcurl -d '{"id": 1234, "tags": ["foo","bar"]}' \
2     grpc.server.com:443 my.custom.server.Service/Method
```

As can be seen in the example, the supplied body must be in JSON format. The body will be parsed and then transmitted to the server in the protobuf binary format.

If you want to include `grpcurl` in a command pipeline, such as when using `jq` to create a request body, you can use `-d @`, which tells `grpcurl` to read the actual request body from stdin:

```
1  grpcurl -d @ grpc.server.com:443 my.custom.server.Service/Method <<EOM
2  {
3      "id": 1234,
4      "tags": [
5          "foor",
6          "bar"
7      ]
8  }
9  EOM
```

Listing Services

To list all services exposed by a server, use the “list” verb. When using `.proto` source or proto set files instead of server reflection, this lists all services defined in the source or proto set files.

```
1  # Server supports reflection
2  grpcurl localhost:8787 list
3
4  # Using compiled proto set files
5  grpcurl -proto set my-protos.bin list
6
7  # Using proto sources
8  grpcurl -import-path ../protos -proto my-stuff.proto list
```

The “list” verb also lets you see all methods in a particular service:

```
1  grpcurl localhost:8787 list my.custom.server.Service
```

Describing Elements

The “describe” verb will print the type of any symbol that the server knows about or that is found in a given proto set file. It also prints a description of that symbol, in the form of snippets of proto source. It won’t necessarily be the original source that defined the element, but it will be equivalent.

```
1  # Server supports reflection
2  grpcurl localhost:8787 describe my.custom.server.Service.MethodOne
3
4  # Using compiled proto set files
5  grpcurl -proto set my-protos.bin describe my.custom.server.Service.
   MethodOne
```

```
6
7 # Using proto sources
8 grpcurl -import-path ../protos -proto my-stuff.proto describe my.custom
  .server.Service.MethodOne
```

Descriptor Sources

The `grpcurl` tool can operate on a variety of sources for descriptors. The descriptors are required, in order for `grpcurl` to understand the RPC schema, translate inputs into the protobuf binary format as well as translate responses from the binary format into text. The sections below document the supported sources and what command-line flags are needed to use them.

Server Reflection

Without any additional command-line flags, `grpcurl` will try to use server reflection.

Examples for how to set up server reflection can be found [here](#).

When using reflection, the server address (host:port or path to Unix socket) is required even for “list” and “describe” operations, so that `grpcurl` can connect to the server and ask it for its descriptors.

Proto Source Files

To use `grpcurl` on servers that do not support reflection, you can use `.proto` source files.

In addition to using `-proto` flags to point `grpcurl` at the relevant proto source file(s), you may also need to supply `-import-path` flags to tell `grpcurl` the folders from which dependencies can be imported.

Just like when compiling with `protoc`, you do *not* need to provide an import path for the location of the standard protos included with `protoc` (which contain various “well-known types” with a package definition of `google.protobuf`). These files are “known” by `grpcurl` as a snapshot of their descriptors is built into the `grpcurl` binary.

When using proto sources, you can omit the server address (host:port or path to Unix socket) when using the “list” and “describe” operations since they only need to consult the proto source files.

Protoset Files

You can also use compiled protoset files with `grpcurl`. If you are scripting `grpcurl` and need to re-use the same proto sources for many invocations, you will see better performance by using protoset

files (since it skips the parsing and compilation steps with each invocation).

Protoset files contain binary encoded `google.protobuf.FileDescriptorSet` protos. To create a protoset file, invoke `protoc` with the `*.proto` files that define the service:

```
1 protoc --proto_path=. \
2     --descriptor_set_out=myservice.protoset \
3     --include_imports \
4     my/custom/server/service.proto
```

The `--descriptor_set_out` argument is what tells `protoc` to produce a protoset, and the `--include_imports` argument is necessary for the protoset to contain everything that `grpcurl` needs to process and understand the schema.

When using protosets, you can omit the server address (host:port or path to Unix socket) when using the “list” and “describe” operations since they only need to consult the protoset files.