API Client Generator for Python Documentation

Release latest

Luke Sneeringer

Dec 30, 2018

Contents

A generator for protocol buffer described APIs for and in Python 3.

This program accepts an API specified in protocol buffers and generates a client library, which can be used to interact with that API. It is implemented as a plugin to `protoc`, the protocol buffer compiler.

Warning: This tool is a proof of concept and is being iterated on rapidly. Feedback is welcome, but please do not try to use this in some kind of system where stability is an expectation.

CHAPTER 1

Getting Started

This code generator is implemented as a plugin to `protoc`, the compiler for protocol buffers, and will run in any environment that Python 3.6+ and protocol buffers do.

Because dependency management and such can be a significant undertaking, we offer a Docker image and interface which requires you only to have Docker installed and provide the protos for your API.

It is also possible to install the tool locally and run it through `protoc`, and this approach is fully supported.

Note: The Docker approach is recommended for users new to this ecosystem, or those which do not have a robust Python environment available.

1.1 Docker Image

If you are just getting started with code generation for protobuf-based APIs, or if you do not have a robust Python environment already available, we recommend using our Docker image to build client libraries.

However, this tool offers first-class support for local execution using protoc: *Local Installation*. It is still reasonably easy, but initial setup will take a bit longer.

Note: If you are interested in contributing, using a local installation is recommended.

1.1.1 Installing

Docker

In order to use a Docker image, you must have Docker installed. Docker is a container management service, and is available on Linux, Mac, and Windows (although most of these instructions will be biased toward Linux and Mac).

Install Docker according to their installation instructions.

Note: This image requires Docker 17.05 or later.

Pull the Docker Image

Once Docker is installed, simply pull the Docker image for this tool:

```
$ docker pull gcr.io/gapic-images/gapic-generator-python:latest
```

1.1.2 Usage

To use this plugin, you will need an API which is specified using protocol buffers. Additionally, this plugin makes some assumptions at the margins according to Google API design conventions, so following those conventions is recommended.

Example

If you want to experiment with an already-existing API, one example is available. (Reminder that this is still considered experimental, so apologies for this part being a bit strange.)

You need to clone the googleapis repository from GitHub, and change to a special branch:

```
$ git clone git@github.com:googleapis/googleapis.git
$ cd googleapis
$ git checkout --track -b input-contract origin/input-contract
$ cd ..
```

The API available as an example (thus far) is the Google Cloud Vision API, available in the `google/cloud/vision/v1/` subdirectory. This will be used for the remainder of the examples on this page.

Compiling an API

Note: If you are running code generation repeatedly, executing the long `docker run` command may be cumbersome. While you should ensure you understand this section, a shortcut script is available to make iterative work easier.

Compile the API into a client library by invoking the Docker image.

It is worth noting that the image must interact with the host machine (your local machine) for two things: reading in the protos you wish to compile, and writing the output. This means that when you run the image, two mount points are required in order for anything useful to happen.

In particular, the input protos are expected to be mounted into `/in/`, and the desired output location is expected to be mounted into `/out/`. The output directory must also be writable.

Note: The `/in/` and `/out/` directories inside the image are hard-coded; they can not be altered where they appear in the command below.

Perform that step with `docker run`:

```
# This is assumed to be run from the `googleapis` project root.
$ docker run \
  --mount type=bind,source=google/cloud/vision/v1/,destination=/in/google/cloud/
↪vision/v1/,readonly \
  --mount type=bind,source=dest/,destination=/out/ \
  --rm \
  --user $UID \
  gcr.io/gapic-images/gapic-generator-python
```

Warning: `protoc` is *very* picky about paths, and the exact construction here matters a lot. The source is `google/cloud/vision/v1/`, and then the destination is that full directory path after the `/in/` root; therefore: `/in/google/cloud/vision/v1/`.

This matters because of how proto imports are resolved. The `import` statement imports a *file*, relative to a base directory or set of base directories, called the `proto_path`. This is assumed (and hard-coded) to `/in/` in the Docker image, and so any directory structure present in the imports of the proto files must be preserved beneath this for compilation to succeed.

1.1.3 Verifying the Library

Once you have compiled a client library, whether using a Docker image or a local installation, it is time for the fun part: actually running it!

Create a virtual environment for the library:

```
$ virtualenv ~/.local/client-lib --python=`which python3.7`
$ source ~/.local/client-lib/bin/activate
```

Next, install the library:

```
$ cd /dest/
$ pip install --editable .
```

Now it is time to play with it! Here is a test script:

```
# This is the client library generated by this plugin.
from google.cloud import vision

# Instantiate the client.
#
# If you need to manually specify credentials, do so here.
# More info: https://cloud.google.com/docs/authentication/getting-started
#
# If you wish, you can send `transport='grpc'` or `transport='http'`
# to change which underlying transport layer is being used.
ia = vision.ImageAnnotator()

# Send the request to the server and get the response.
response = ia.batch_annotate_images({
    'requests': [{
        'features': [{
            'type': vision.types.image_annotator.Feature.Type.LABEL_DETECTION,
        }],
        'image': {'source': {
```

(continues on next page)

(continued from previous page)

```
            'image_uri': 'https://s3.amazonaws.com/cdn0.michiganbulb.com'
                         '/images/350/66623.jpg',
        }},
    }],
})
print(response)
```

1.2 Local Installation

If you are just getting started with code generation for protobuf-based APIs, or if you do not have a robust Python environment already available, it is probably easier to get started using Docker: *Docker Image*

However, this tool offers first-class support for local execution using `protoc`. It is still reasonably easy, but initial setup will take a bit longer.

Note: If you are interested in contributing, setup according to these steps is recommended.

1.2.1 Installing

protoc

This tool is implemented as a plugin to the protocol buffers compiler, so in order to use it, you will need to have the `protoc` command available.

The release page on GitHub contains the download you need.

Note: You may notice both packages that designate languages (e.g. `protobuf-python-X.Y.Z.tar.gz`) as well as packages that designate architectures (e.g. `protoc-X.Y.Z-linux-x86_64.zip`). You want the one that designates an architecture; your goal here is to have a CLI command.

It is likely preferable to install `protoc` somewhere on your shell's path, but this is not a strict requirement (as you will be invoking it directly). `protoc` is also quirky about how it handles well-known protos; you probably also want to copy them into `/usr/local/include`

To ensure it is installed propertly:

```
$ protoc --version
libprotoc 3.6.0
```

pandoc

This generator relies on pandoc to convert from Markdown (the *lingua franca* for documentation in protocol buffers) into ReStructured Text (the *lingua franca* for documentation in Python).

Install this using an appropriate mechanism for your operating system. Multiple installation paths are documented on the pandoc installation page.

API Generator for Python

This package is provided as a standard Python library, and can be installed the usual ways. It fundamentally provides a CLI command, `protoc-gen-python_gapic`, (yes, the mismatch of `kebob-case` and `snake_case` is weird, sorry), so you will want to install using a mechanism that is conducive to making CLI commands available.

Additionally, this program currently only runs against Python 3.6 or Python 3.7, so you will need that installed. (Most Linux distributions ship with earlier versions.) Use pyenv to get Python 3.7 installed in a friendly way.

As for this library itself, the recommended installation approach is pipsi.

```
# Due to its experimental state, this tool is not published to a
# package manager; you should clone it.
# (You can pip install it from GitHub, not not if you want to tinker.)
git clone git@github.com:googleapis/gapic-generator-python.git
cd gapic-generator-python/

# Install the tool. This will handle the virtualenv for you, and
# make an appropriately-aliased executable.
# The `--editable` flag is only necessary if you want to work on the
# tool (as opposed to just use it).
pipsi install --editable --python=`which python3.7` .
```

To ensure the tool is installed properly:

```
$ which protoc-gen-python_gapic
/path/to/protoc-gen-python_gapic
```

1.2.2 Usage

To use this plugin, you will need an API which is specified using protocol buffers. Additionally, this plugin makes some assumptions at the margins according to Google API design conventions, so following those conventions is recommended.

Example

If you want to experiment with an already-existing API, one example is available. (Reminder that this is still considered experimental, so apologies for this part being a bit strange.)

You need to clone the googleapis repository from GitHub, and change to a special branch:

```
$ git clone git@github.com:googleapis/googleapis.git
$ cd googleapis
$ git checkout --track -b input-contract origin/input-contract
$ cd ..
```

The API available as an example (thus far) is the Google Cloud Vision API, available in the `google/cloud/vision/v1/` subdirectory. This will be used for the remainder of the examples on this page.

You will also need the common protos, currently in experimental status, which define certain client-specific annotations. These are in the api-common-protos repository. Clone this from GitHub also:

```
$ git clone git@github.com:googleapis/api-common-protos.git
$ cd api-common-protos
$ git checkout --track -b input-contract origin/input-contract
$ cd ..
```

Compiling an API

Compile the API into a client library by invoking `protoc` directly. This plugin is invoked under the hood via. the `--python_gapic_out` switch.

```
# This is assumed to be in the `googleapis` project root, and we also
# assume that api-common-protos is next to it.
$ protoc google/cloud/vision/v1/*.proto \
    --proto_path=../api-common-protos/ --proto_path=. \
    --python_gapic_out=/dest/
```

Note: A reminder about paths.

Remember that `protoc` is particular about paths. It requires all paths where it expects to find protos, and *order matters*. In this case, the common protos must come first, and then the path to the API being built.

1.2.3 Verifying the Library

Once you have compiled a client library, whether using a Docker image or a local installation, it is time for the fun part: actually running it!

Create a virtual environment for the library:

```
$ virtualenv ~/.local/client-lib --python=`which python3.7`
$ source ~/.local/client-lib/bin/activate
```

Next, install the library:

```
$ cd /dest/
$ pip install --editable .
```

Now it is time to play with it! Here is a test script:

```
# This is the client library generated by this plugin.
from google.cloud import vision

# Instantiate the client.
#
# If you need to manually specify credentials, do so here.
# More info: https://cloud.google.com/docs/authentication/getting-started
#
# If you wish, you can send `transport='grpc'` or `transport='http'`
# to change which underlying transport layer is being used.
ia = vision.ImageAnnotator()

# Send the request to the server and get the response.
response = ia.batch_annotate_images({
    'requests': [{
        'features': [{
            'type': vision.types.image_annotator.Feature.Type.LABEL_DETECTION,
        }],
        'image': {'source': {
            'image_uri': 'https://s3.amazonaws.com/cdn0.michiganbulb.com'
                         '/images/350/66623.jpg',
        }},
```

(continues on next page)

(continued from previous page)

```
    }],
})
print(response)
```

CHAPTER 2

API Configuration

This code generator relies on some configuration not specified in many published protocol buffers.

Warning: In fact, this is intended to serve as a reference implementation for proposed configuration, so as of this writing it is not published anywhere, and is subject to change.

This plugin *will* successfully publish a library on a valid protobuf API even without any additional information set, but may require some post-processing work by a human in this case before the resulting client library will work.

Look for values enclosed by <<< and >>> to quickly spot these. As of this writing, these are the `SERVICE_ADDRESS` and `OAUTH_SCOPES` constants defined in `base.py` files.

Reading further assumes you are at least nominally familiar with protocol buffers and their syntax. You may not be familiar with options yet; it is recommended to read up on them before continuing. (As a note, no need to learn about creating custom options; being able to set options that are already defined is sufficient.)

2.1 Annotations and Imports

As mentioned above, this tool uses a interface specification that is currently experimental.

When specifying an annotation, your proto will need to import the file where the annotation is defined. If you try to use an annotation without importing the dependency proto, then `protoc` will give you an error.

These protos live on the `input-contract` branch in the api-common-protos repository.

Your best bet is to likely clone this repository:

```
$ git clone git@github.com:googleapis/api-common-protos.git
$ cd api-common-protos
$ git checkout --track -b input-contract origin/input-contract
```

Once this is done, you will need to specify the root of this repository on disk as a `--proto_path` whenever invoking `protoc`.

2.2 API Client Information

The most important piece of information this plugin requires is information about the client library itself: what should it be called, what is its proper namespace, and so on.

This is rolled up into a structure called `Metadata`, and the annotation is defined in google/api/metadata.proto.

The option may be defined as a full structure at the top level of the proto file. It is recommended that this be declared other under `option` directives, and above services or messages.

This annotation is optional, and you may not need it. The generator will infer a proper name, namespace and version from the `package` statement:

```
// This will come out to be:
//   package namespace: ['Acme', 'Manufacturing']
//   package name: 'Anvils'
//   version: 'v1'
package acme.manufacturing.anvils.v1;
```

If the inferred package name is wrong for some reason, then the annotation is important.

```
package acme.anvils.v1;

// The namespace provided here will take precedence over the
// inferred one.
option (google.api.metadata) = {
  "package_namespace": ["Acme", "Manufacturing"]
};
```

Note: The `google.api.metadata` annotation can be used to specify a namespace or name, but the version *must* be specified in the proto package.

2.3 Service Information

In order to properly connect to an API, the client library needs to know where the API service is running, as well as what (if anything) else is required in order to properly connect.

This plugin understands two options for this, which are also defined in google/api/metadata.proto. Rather than being options on top level files, however, these are both options on services. If an API defines more than one service, these options do *not* need to match between them.

The first option is the **host** where the service can be reached:

```
service AnvilService {
  option (google.api.default_host) = "anvils.acme.com"
}
```

The second option is any oauth scopes which are needed. Google's auth libraries (such as google-auth in Python, which code generated by this plugin uses) expect that credentials declare what scopes they believe they need, and the auth libraries do the right thing in the situation where authorization is needed, access has been revoked, and so on.

```
service AnvilService {
  option (google.api.oauth) = {
    scopes: ["https://anvils.acme.com/auth/browse-anvils",
```

(continues on next page)

(continued from previous page)

```
                "https://anvils.acme.com/auth/drop-anvils"]
  };
}
```

2.4 Long-Running Operations

Occasionally, API requests may take a long time. In this case, APIs may run a task in the background and provide the client with a token to retrieve the result later.

The `google.longrunning.Operation` message is intended for this purpose. It is defined in google/longrunning/operations.proto and can be used as the return type of an RPC.

However, when doing this, the ultimate return type is lost. Therefore, it is important to annotate the return type (and metadata type, if applicable) so that client libraries are able to deserialize the message.

```
import "google/api/annotations.proto";

package acme.anvils.v1;

service AnvilService {
  rpc DeliverAnvil(DeliverAnvilRequest)
      returns (google.longrunning.Operation) {
    option (google.api.operation) = {
      response_type: "acme.anvils.v1.DeliverAnvilResponse"
      metadata_type: "acme.anvils.v1.DeliverAnvilMetadata"
    };
  }
}
```

CHAPTER 3

How Code Generation Works

This page gives a brief decription of how *this* code generator works. It is not intended to be the final treatise on how to write *any* code generator. It is meant to be a reference for those who wish to contribute to this effort, or to use it as a reference implementation.

There are two steps: a **parse** step which essentially involves reorganizing data to make it more friendly to templates, and a **translation** step which sends information about the API to templates, which ultimately write the library.

3.1 The protoc contract

This code generator is written as a **`protoc`** plugin, which operates on a defined contract. The contract is straightforward: a plugin must accept a `CodeGeneratorRequest` (essentially a sequence of `FileDescriptor` objects) and output a `CodeGeneratorResponse`.

If you are unfamiliar with **`protoc`** plugins, welcome! That last paragraph likely sounded not as straightforward as claimed. It may be useful to read plugin.proto and descriptor.proto before continuing on. The former describes the contract with plugins (such as this one) and is relatively easy to digest, the latter describes protocol buffer files themselves and is rather dense. The key point to grasp is that each `.proto` *file* compiles into one of these proto messages (called *descriptors*), and this plugin's job is to parse those descriptors.

That said, you should not need to know the ins and outs of the `protoc` contract model to be able to follow what this library is doing.

3.2 Entry Point

The entry point to this tool is `gapic/cli/generate.py`. The function in this module is responsible for accepting CLI input, building the internal API schema, and then rendering templates and using them to build a response object.

3.3 Parse

As mentioned, this plugin is divided into two steps. The first step is parsing. The guts of this is handled by the *API* object, which is this plugin's internal representation of the full API client.

In particular, this class has a *build()* method which accepts a sequence of `FileDescriptor` objects (remember, this is `protoc`'s internal representation of each proto file). That method iterates over each file and creates a *Proto* object for each one.

Note: An *API* object will not only be given the descriptors for the files you specify, but also all of their dependencies. `protoc` is smart enough to de-duplicate and send everything in the correct order.

The *API* object's primary purpose is to make sure all the information from the proto files is in one place, and reasonably accessible by Jinja templates (which by design are not allowed to call arbitrary Python code). Mostly, it tries to avoid creating an entirely duplicate structure, and simply wraps the descriptor representations. However, some data needs to be moved around to get it into a structure useful for templates (in particular, descriptors have an unfriendly approach to sorting protobuf comments, and this parsing step places these back alongside their referent objects).

The internal data model does use wrapper classes around most of the descriptors, such as *Service* and *MessageType*. These consistently contain their original descriptor (which is always spelled with a `_pb` suffix, e.g. the `Service` wrapper class has a `service_pb` instance variable). These exist to handle bringing along additional relevant data (such as the protobuf comments as mentioned above) and handling resolution of references (for example, allowing a *Method* to reference its input and output types, rather than just the strings).

These wrapper classes follow a consistent structure:

- They define a `__getattr__` method that defaults to the wrapped desctiptor unless the wrapper itself provides something, making the wrappers themselves transparent to templates.
- They provide a `meta` attribute with metadata (package information and documentation). That means templates can consistently access the name for the module where an object can be found, or an object's documentation, in predictable and consistent places (`thing.meta.doc`, for example, prints the comments for `thing`).

3.4 Translation

The translation step follows a straightfoward process to write the contents of client library files.

This works by reading in and rendering Jinja templates into a string. The file path of the Jinja template is used to determine the filename in the resulting client library.

More details on authoring templates is discussed on the *Templates* page.

3.5 Exit Point

Once the individual strings corresponding to each file to be generated is collected into memory, these are pieced together into a `CodeGeneratorResponse` object, which is serialized and written to stdout.

CHAPTER 4

Templates

This page provides a description of templates: how to write them, what variables they receive, and so on and so forth.

In many cases, it should be possible to provide alternative Python libraries based on protocol buffers by only editing templates (or authoring new ones), with no requirement to alter the primary codebase itself.

4.1 Jinja

All templates are implemented in Jinja, Armin Ronacher's excellent templating library for Python. This document assumes that you are already familiar with the basics of writing Jinja templates, and does not seek to cover that here.

4.2 Locating Templates

Templates are included in output simply on the basis that they exist. **There is no master list of templates**; it is assumed that every template should be rendered (unless its name begins with a single underscore).

The name of the output file is based on the name of the template, with the following string replacements applied:

- The `.j2` suffix is removed.
- `$namespace` is replaced with the namespace specified in the client, converted to appropriate Python module case. If there is no namespace, this segment is dropped. If the namespace has more than one element, this is expanded out in the directory structure. (For example, a namespace of `['Acme', 'Manufacturing']` will translate into `acme/manufacturing/` directories.)
- `$name` is replaced with the client name. This is expected to be present.
- `$version` is replaced with the client version (the version of the API). If there is no specified version, this is dropped.
- `$service` is replaced with the service name, converted to appropriate Python module case. There may be more than one service in an API; read on for more about this.

Note: `$name_$version` is a special case: It is replaced with the client name, followed by the version. However, if there is no version, both it and the underscore are dropped.

4.3 Context (Variables)

Every template receives one variable, spelled `api`. It is the *`API`* object that was pieced together in the parsing step.

Most APIs also receive one additional variable depending on what piece of the API structure is being iterated over:

- **Services.** APIs can (and often do) have more than one service. Therefore, templates with `$service` in their name are rendered *once per service*, with the `$service` string changed to the name of the service itself (in snake case, because this is Python). These templates receive a `service` variable (an instance of *`Service`*) corresponding to the service currently being iterated over.
- **Protos.** Similarly, APIs can (and often do) have more than one proto file containing messages. Therefore, templates with `$proto` in their name are rendered *once per proto*, with the `$proto``string changed to the name of the proto file. These templates receive a ``proto` variable (an instance of *`Proto`*) corresponding to the proto currently being iterated over.

4.4 Filters

Additionally, templates receive a limited number of filters useful for writing properly formatted templates.

These are:

- `rst` (`rst()`): Converts a string to ReStructured Text. If the string appears not to be formatted (contains no obvious Markdown syntax characters), then this method forwards to `wrap`.
- `snake_case` (`to_snake_case()`): Converts a string in any sane case system to snake case.
- `wrap` (`wrap()`): Wraps arbitrary text. Keyword arguments on this method such as `offset` and `indent` should make it relatively easy to take an arbitrary string and make it wrap to 79 characters appropriately.

CHAPTER 5

Features and Limitations

Nice things this client does:

- Implemented in pure Python, with language-idiomatic templating tools.
- It supports multiple transports: both gRPC and protobuf over HTTP/1.1. A JSON-based transport would be easy to add.
- It uses a lighter-weight configuration, specified in the protocol buffer itself.

As this is experimental work, please note the following limitations:

- The output only works on Python 3.5 and above.
- The configuration annotations are experimental and provided in an awkward location.
- gRPC must be installed even if you are not using it (this is due to some minor issues in `api-core`).
- No support for samples yet.

CHAPTER 6

Reference

Below is a reference for the major classes and functions within this module.

It is split into three main sections:

- The `schema` module contains data classes that make up the internal representation for an *`API`*. The API contains thin wrappers around protocol buffer descriptors; the goal of the wrappers is to mostly expose the underlying descriptors, but make some of the more complicated access and references easier in templates.
- The `generator` module contains most of the logic. Its *`Generator`* class is the thing that takes a request from `protoc` and gives it back a response.
- The `utils` module contains utility functions needed elsewhere, including some functions that are sent to all templates as Jinja filters.

Note: Templates are housed in the `templates` directory, which is a sibling to the modules listed above.

6.1 generator

The `generator` module contains the code generation logic.

The core of this work is around the *`Generator`* class, which divides up the processing of individual templates.

class gapic.generator.generator.**Generator**(*templates: str = None*)

A protoc code generator for client libraries.

This class provides an interface for getting a `CodeGeneratorResponse` for an `API` schema object (which it does through rendering templates).

Parameters **templates** (*str*) – Optional. Path to the templates to be rendered. If this is not provided, the templates included with this application are used.

get_response(*api_schema: gapic.schema.api.API*) → google.protobuf.compiler.plugin_pb2.CodeGeneratorResponse

Return a `CodeGeneratorResponse` for this library.

This is a complete response to be written to (usually) stdout, and thus read by `protoc`.

Parameters **api_schema** (*API*) – An API schema object.

Returns A response describing appropriate files and contents. See `plugin.proto`.

Return type CodeGeneratorResponse

6.2 schema

The `schema` module provides a normalized API representation.

In general, this module can be considered in three parts: wrappers, metadata, and a roll-up view of an API as a whole.

These three parts are divided into the three component modules.

6.2.1 api

This module contains the "roll-up" class, *API*. Everything else in the *schema* module is usually accessed through an *API* object.

class gapic.schema.api.**API**(*naming: gapic.schema.naming.Naming, all_protos: Mapping[str, gapic.schema.api.Proto], subpackage_view: Tuple[str] = <factory>*)

A representation of a full API.

This represents a top-down view of a complete API, as loaded from a set of protocol buffer files. Once the descriptors are loaded (see `load()`), this object contains every message, method, service, and everything else needed to write a client library.

An instance of this object is made available to every template (as `api`).

classmethod build(*file_descriptors: Sequence[google.protobuf.descriptor_pb2.FileDescriptorProto], package: str = ''*) → gapic.schema.api.API

Build the internal API schema based on the request.

Parameters

- **file_descriptors** (*Sequence[FileDescriptorProto]*) – A list of `FileDescriptorProto` objects describing the API.
- **package** (*str*) – A protocol buffer package, as a string, for which code should be explicitly generated (including subpackages). Protos with packages outside this list are considered imports rather than explicit targets.

enums

Return a map of all enums available in the API.

messages

Return a map of all messages available in the API.

protos

Return a map of all protos specific to this API.

This property excludes imported protos that are dependencies of this API but not being directly generated.

services

Return a map of all services available in the API.

subpackages
Return a map of all subpackages, if any.

Each value in the mapping is another API object, but the `protos` property only shows protos belonging to the subpackage.

class gapic.schema.api.**Proto**(*file_pb2: google.protobuf.descriptor_pb2.FileDescriptorProto, services: Mapping[str, gapic.schema.wrappers.Service], messages: Mapping[str, gapic.schema.wrappers.MessageType], enums: Mapping[str, gapic.schema.wrappers.EnumType], file_to_generate: bool, meta: gapic.schema.metadata.Metadata = <factory>*)
A representation of a particular proto file within an API.

classmethod build(*file_descriptor: google.protobuf.descriptor_pb2.FileDescriptorProto, file_to_generate: bool, naming: gapic.schema.naming.Naming, prior_protos: Mapping[str, Proto] = None*) → gapic.schema.api.Proto
Build and return a Proto instance.

Parameters

- **file_descriptor** (*FileDescriptorProto*) – The protocol buffer object describing the proto file.
- **file_to_generate** (*bool*) – Whether this is a file which is to be directly generated, or a dependency.
- **naming** (*Naming*) – The *Naming* instance associated with the API.
- **prior_protos** (*Proto*) – Previous, already processed protos. These are needed to look up messages in imported protos.

disambiguate(*string: str*) → str
Return a disambiguated string for the context of this proto.

This is used for avoiding naming collisions. Generally, this method returns the same string, but it returns a modified version if it will cause a naming collision with messages or fields in this proto.

module_name
Return the appropriate module name for this service.

Returns

The module name for this service (which is the service name in snake case).

Return type str

names
Return a set of names used by this proto.

This is used for detecting naming collisions in the module names used for imports.

python_modules
Return a sequence of Python modules, for import.

The results of this method are in alphabetical order (by package, then module), and do not contain duplicates.

Returns The package and module pair, intended for use in a `from package import module` type of statement.

Return type Sequence[Tuple[str, str]]

top
Return a proto shim which is only aware of top-level objects.

This is useful in a situation where a template wishes to iterate over only those messages and enums that are at the top level of the file.

6.2.2 metadata

The `metadata` module defines schema for where data was parsed from. This library places every protocol buffer descriptor in a wrapper class (see *wrappers*) before loading it into the *API* object.

As we iterate over descriptors during the loading process, it is important to know where they came from, because sometimes protocol buffer types are referenced by fully-qualified string (e.g. `method.input_type`), and we want to resolve those references.

Additionally, protocol buffers stores data from the comments in the `.proto` in a separate structure, and this object model re-connects the comments with the things they describe for easy access in templates.

class gapic.schema.metadata.**Address**(*name:str=''*, *module:str=''*, *module_path:Tuple[int]=<factory>*, *package:Tuple[str]=<factory>*, *parent:Tuple[str]=<factory>*, *api_naming:gapic.schema.naming.Naming=<factory>*, *collisions:Set[str]=<factory>*)

child(*child_name: str, path: Tuple[int]*) → gapic.schema.metadata.Address
Return a new child of the current Address.

Parameters **child_name** (*str*) – The name of the child node. This address' name is appended to `parent`.

Returns The new address object.

Return type Address

module_alias
Return an appropriate module alias if necessary.

If the module name is not a collision, return empty string.

This method provides a mechanism for resolving naming conflicts, while still providing names that are fundamentally readable to users (albeit looking auto-generated).

proto
Return the proto selector for this type.

proto_package
Return the proto package for this type.

python_import
Return the Python import for this type.

rel(*address: gapic.schema.metadata.Address*) → str
Return an identifier for this type, relative to the given address.

Similar to `__str__()`, but accepts an address (expected to be the module being written) and truncates the beginning module if the address matches the identifier's address. Templates can use this in situations where otherwise they would refer to themselves.

Parameters **address** (*Address*) – The address to compare against.

Returns The appropriate identifier.

Return type str

resolve(*selector: str*) → str
Resolve a potentially-relative protobuf selector.

This takes a protobuf selector which may be fully-qualified (e.g. *foo.bar.v1.Baz*) or may be relative (*Baz*) and returns the fully-qualified version.

This method is naive and does not check to see if the message actually exists.

Parameters **selector** (*str*) – A protobuf selector, either fully-qualified or relative.

Returns An absolute selector.

Return type str

sphinx
Return the Sphinx identifier for this type.

subpackage
Return the subpackage below the versioned module name, if any.

with_context(**, collisions: Set[str]*) → gapic.schema.metadata.Address
Return a derivative of this address with the provided context.

This method is used to address naming collisions. The returned `Address` object aliases module names to avoid naming collisions in the file being written.

class gapic.schema.metadata.**FieldIdentifier**(*ident:gapic.schema.metadata.Address*, *repeated:bool*)

class gapic.schema.metadata.**Metadata**(*address:gapic.schema.metadata.Address=<factory>*, *documentation:google.protobuf.descriptor_pb2.Location=<factory>*)

doc
Return the best comment.

This property prefers the leading comment if one is available, and falls back to a trailing comment or a detached comment otherwise.

If there are no comments, return empty string. (This means a template is always guaranteed to get a string.)

with_context(**, collisions: Set[str]*) → gapic.schema.metadata.Metadata
Return a derivative of this metadata with the provided context.

This method is used to address naming collisions. The returned `Address` object aliases module names to avoid naming collisions in the file being written.

6.2.3 naming

class gapic.schema.naming.**Naming**(*name: str = ''*, *namespace: Tuple[str] = <factory>*, *version: str = ''*, *product_name: str = ''*, *product_url: str = ''*, *proto_package: str = ''*)
Naming data for an API.

This class contains the naming nomenclature used for this API within templates.

An instance of this object is made available to every template (as `api.naming`).

classmethod build(**file_descriptors*) → gapic.schema.naming.Naming
Return a full Naming instance based on these file descriptors.

This is pieced together from the proto package names as well as the `google.api.metadata` file annotation. This information may be present in one or many files; this method is tolerant as long as the data does not conflict.

Parameters **file_descriptors** (*Iterable[FileDescriptorProto]*) – A list of file descriptor protos. This list should only include the files actually targeted for output (not their imports).

Returns

A ***Naming*** **instance which is provided to** templates as part of the *API*.

Return type Naming

Raises ValueError – If the provided file descriptors contain contradictory information.

long_name
Return an appropriate title-cased long name.

module_name
Return the appropriate Python module name.

module_namespace
Return the appropriate Python module namespace as a tuple.

namespace_packages
Return the appropriate Python namespace packages.

versioned_module_name
Return the versiond module name (e.g. apiname_v1).

If there is no version, this is the same as module_name.

warehouse_package_name
Return the appropriate Python package name for Warehouse.

6.2.4 wrappers

Module containing wrapper classes around meta-descriptors.

This module contains dataclasses which wrap the descriptor protos defined in google/protobuf/descriptor.proto (which are descriptors that describe descriptors).

These wrappers exist in order to provide useful helper methods and generally ease access to things in templates (in particular, documentation, certain aggregate views of things, etc.)

Reading of underlying descriptor properties in templates *is* okay, a __getattr__ method which consistently routes in this way is provided. Documentation is consistently at {thing}.meta.doc.

class gapic.schema.wrappers.**EnumType**(*enum_pb: google.protobuf.descriptor_pb2.EnumDescriptorProto*, *values: List[gapic.schema.wrappers.EnumValueType]*, *meta: gapic.schema.metadata.Metadata* = <*factory*>)
Description of an enum (defined with the enum keyword.)

ident
Return the identifier data to be used in templates.

with_context(**, collisions: Set[str]*) → gapic.schema.wrappers.EnumType
Return a derivative of this enum with the provided context.

This method is used to address naming collisions. The returned EnumType object aliases module names to avoid naming collisions in the file being written.

class gapic.schema.wrappers.**EnumValueType**(*enum_value_pb: google.protobuf.descriptor_pb2.EnumValueDescriptorProto*, *meta: gapic.schema.metadata.Metadata = <factory>*)

Description of an enum value.

class gapic.schema.wrappers.**Field**(*field_pb: google.protobuf.descriptor_pb2.FieldDescriptorProto*, *message: MessageType = None*, *enum: EnumType = None*, *meta: gapic.schema.metadata.Metadata = <factory>*)

Description of a field.

ident
Return the identifier to be used in templates.

is_primitive
Return True if the field is a primitive, False otherwise.

proto_type
Return the proto type constant to be used in templates.

repeated
Return True if this is a repeated field, False otherwise.

Returns Whether this field is repeated.

Return type bool

required
Return True if this is a required field, False otherwise.

Returns Whether this field is required.

Return type bool

type
Return the type of this field.

with_context(**, collisions: Set[str]*) → gapic.schema.wrappers.Field
Return a derivative of this field with the provided context.

This method is used to address naming collisions. The returned `Field` object aliases module names to avoid naming collisions in the file being written.

class gapic.schema.wrappers.**MessageType**(*message_pb: google.protobuf.descriptor_pb2.DescriptorProto*, *fields: Mapping[str, gapic.schema.wrappers.Field]*, *nested_enums: Mapping[str, EnumType]*, *nested_messages: Mapping[str, MessageType]*, *meta: gapic.schema.metadata.Metadata = <factory>*)

Description of a message (defined with the `message` keyword).

field_types
Return all composite fields used in this proto's messages.

get_field(**field_path*, *collisions: Set[str] = frozenset()*) → gapic.schema.wrappers.Field
Return a field arbitrarily deep in this message's structure.

This method recursively traverses the message tree to return the requested inner-field.

Traversing through repeated fields is not supported; a repeated field may be specified if and only if it is the last field in the path.

Parameters **field_path** (*Sequence[str]*) – The field path.

Returns A field object.

Return type Field

Raises `KeyError` – If a repeated field is used in the non-terminal position in the path.

ident
Return the identifier data to be used in templates.

with_context(*, *collisions: Set[str]*, *skip_fields: bool = False*) → gapic.schema.wrappers.MessageType
Return a derivative of this message with the provided context.

This method is used to address naming collisions. The returned `MessageType` object aliases module names to avoid naming collisions in the file being written.

The `skip_fields` argument will omit applying the context to the underlying fields. This provides for an "exit" in the case of circular references.

class gapic.schema.wrappers.**Method**(*method_pb: google.protobuf.descriptor_pb2.MethodDescriptorProto*, *input: gapic.schema.wrappers.MessageType*, *output: gapic.schema.wrappers.MessageType*, *meta: gapic.schema.metadata.Metadata = <factory>*)
Description of a method (defined with the `rpc` keyword).

field_headers
Return the field headers defined for this method.

grpc_stub_type
Return the type of gRPC stub to use.

ref_types
Return types referenced by this method.

signatures
Return the signature defined for this method.

with_context(*, *collisions: Set[str]*) → gapic.schema.wrappers.Method
Return a derivative of this method with the provided context.

This method is used to address naming collisions. The returned `Method` object aliases module names to avoid naming collisions in the file being written.

class gapic.schema.wrappers.**MethodSignature**(*name:str*, *fields:Mapping[str, gapic.schema.wrappers.Field]*)

composite_types
Return all composite types used in this signature.

dispatch_field
Return the first field.

This is what is used for *functools.singledispatch*.

class gapic.schema.wrappers.**MethodSignatures**(*all:Tuple[gapic.schema.wrappers.MethodSignature]*)

single_dispatch
Return a tuple of signatures, grouped and deduped by dispatch type.

In the Python 3 templates, we only honor at most one method signature per initial argument type, and only for primitives.

This method groups and deduplicates signatures and sends back only the signatures that the template actually wants.

Returns

Method signatures to be used with "single dispatch" routing.

Return type Tuple[*MethodSignature*]

class gapic.schema.wrappers.**OperationType**(*lro_response: gapic.schema.wrappers.MessageType*, *lro_metadata: gapic.schema.wrappers.MessageType = None*)

Wrapper class for `Operation`.

This exists for interface consistency, so Operations can be used alongside *`MessageType`* instances.

meta
Return a Metadata object.

name
Return the class name.

with_context(**, collisions: Set[str]*) → gapic.schema.wrappers.OperationType
Return a derivative of this operation with the provided context.

This method is used to address naming collisions. The returned `OperationType` object aliases module names to avoid naming collisions in the file being written.

class gapic.schema.wrappers.**PythonType**(*python_type: type*)
Wrapper class for Python types.

This exists for interface consistency, so that methods like *`Field.type()`* can return an object and the caller can be confident that a `name` property will be present.

ident
Return the identifier to be used in templates.

Primitives have no import, and no module to reference, so this is simply the name of the class (e.g. "int", "str").

class gapic.schema.wrappers.**Service**(*service_pb: google.protobuf.descriptor_pb2.ServiceDescriptorProto*, *methods: Mapping[str, gapic.schema.wrappers.Method]*, *meta: gapic.schema.metadata.Metadata = <factory>*)

Description of a service (defined with the `service` keyword).

has_lro
Return whether the service has a long-running method.

host
Return the hostname for this service, if specified.

Returns The hostname, with no protocol and no trailing `/`.

Return type str

module_name
Return the appropriate module name for this service.

Returns The service name, in snake case.

Return type str

names
Return a set of names used in this service.

This is used for detecting naming collisions in the module names used for imports.

oauth_scopes
Return a sequence of oauth scopes, if applicable.

Returns A sequence of OAuth scopes.

Return type Sequence[str]

python_modules
Return a sequence of Python modules, for import.

The results of this method are in alphabetical order (by package, then module), and do not contain duplicates.

Returns

The package and module, intended for use in templates.

Return type Sequence[Import]

with_context(*, *collisions: Set[str]*) → gapic.schema.wrappers.Service
Return a derivative of this service with the provided context.

This method is used to address naming collisions. The returned `Service` object aliases module names to avoid naming collisions in the file being written.

6.3 utils

gapic.utils.case.**to_snake_case**(*s: str*) → str
Convert any string to snake case.

This is provided to templates as the `snake_case` filter.

Parameters **s** (*str*) – The input string, provided in any sane case system.

Returns The string in snake case (and all lower-cased).

Return type str

gapic.utils.lines.**wrap**(*text: str*, *width: int*, *, *offset: int = None*, *indent: int = 0*) → str
Wrap the given string to the given width.

This uses `textwrap.fill()` under the hood, but provides useful offset functionality for Jinja templates.

This is provided to all templates as the `wrap` filter.

Parameters

- **text** (*str*) – The initial text string.
- **width** (*int*) – The width at which to wrap the text. If offset is provided, these are automatically counted against this.
- **offset** (*int*) – The offset for the first line of text. This value is subtracted from `width` for the first line only, and is intended to represent the vertical position of the first line as already present in the template. Defaults to the value of `indent`.
- **indent** (*int*) – The number of spaces to indent all lines after the first one.

Returns The wrapped string.

Return type str

Python Module Index

g

Index

N

O

P

R

S

T

V

W