Implementing Code Generators

Luke Sneeringer
lukesneeringer@google.com

APIStrat 2018
Implementing Code Generators

Problem statement:
Get high-quality client libraries into the hands of your API’s customers.

Easy, right?

```python
from googlecloudpubsubapi.googlecloudpubsubapi_client import googlecloudpubsubapiClient

```

Fifteen constructor arguments, ah ah ah!
Easy.

(unless we make it hard)
Two steps to code generation:

1. Schema
2. Output

(also the agenda for this talk)
Protocol buffers

For this talk, I will use **protocol buffers** to describe APIs. The principles are universal.

Protocol buffers is a wire format used at Google and elsewhere.

It is also a *minimalist* API specification format.
Protocol buffers

Concepts:

- Messages (Structs)
- Fields
- Services
- RPCs (Methods)

(There are also enums, which I am not covering in this talk.)

// Provides language translation operations.
service TranslateService {
  // Translates input text, returning translated text.
  rpc TranslateText(TranslateTextRequest)
      returns (TranslateTextResponse) {
    option (google.api.http) = {
      get: "/language/translate/v2/"
    };
  }

  // Returns a list of supported languages.
  rpc GetSupportedLanguages(GetSupportedLanguagesRequest)
      returns (GetSupportedLanguagesResponse) {
    option (google.api.http) = {
      get: "/language/translate/v2/languages"
    };
  }

  // Detects the language of text within a request.
  rpc DetectLanguage(DetectLanguageRequest)
      returns (DetectLanguageResponse) {
    option (google.api.http) = {
      get: "/language/translate/v2/detect"
    };
  }
}
// The main translation request message for the Cloud Translation API.

message TranslateTextRequest {
  // The input text to translate. Repeat this parameter to perform translation operations on multiple text inputs.
  repeated string q = 1;

  // The language of the source text, set to one of the language codes listed in Language Support. If the source language is not specified, the API will attempt to identify the source language automatically and return it within the response.
  string source = 2;

  // Required. The language to use for translation of the input text, set to one of the language codes listed in Language Support.
  string target = 3;

  // ...

}
At a high level, every API has the same structure.
At a high level, every API has the same structure.
At a high level, every API has the same structure. Can be (often is) a resource. Can be primitives or resources. Has attrs just like resources. URI + method routes to a function.
At a high level, every API has the same structure.
Data Model

The key to quality code generation is a simple, minimalist schema.

*Everything* in your data model is a mandate.

Your greatest nemesis: YAGNI.
Focus on preserving and modeling ontological relationships.

Multiple, focused, high quality generators are probably better than one generator that tries to do everything.
  - Distinct generators can have distinct internal schema (better yet, distinct supersets of a common schema).
  - Do not try to cover every target environment or use case.

Language idiomaticity is mostly a distraction at this stage.
  - ...but schema objects can have properties that compute difficult roll-up information (e.g. imports).
Output

Output is easier than schema.

Multiple approaches:

- Abstract syntax tree
- Templates
- Print statements
- ???

All of these choices are good ones (if your generator has a reasonably small target domain).

Easy to refactor.
Output

Design for a world where the output has a different set of maintainers.

Regardless of what output mechanism you use, output code should receive consistent data.

Learning to maintain any one part of the output should be sufficient to maintain all of it.

```render(api) { }
...
}```
Output can generally be procedural ("top-to-bottom").

Individual methods are generally straightforward:

- Data transformation, if any.
- Make a service call.
- Return the response.

Really. It is simpler than it seems.
Output: Tips

- All output-related code should be given the same data.
  - "If you understand any of the templates, you understand them all."
  - Slight exception: Output code that runs multiple times (in a loop) also must be told what is being iterated over.

- Use tooling designed for your target language. (Liberally!)

- Avoid unnecessary layers of indirection.

- Idiomaticity: Sweat the details here.
  - Rely on popular tooling (e.g. code formatters, linters) to help you.
  - Avoid being more opinionated than the "least common denominator" in the ecosystem (unless necessary).
Small.
Focused.
Implementing Code Generators

Luke Sneeringer
lukesneeringer@google.com

APIStrat 2018