Code Generation

Principles & Challenges

Luke Sneeringer
lukesneeringer@google.com

OSCON 2019

Why code generation?

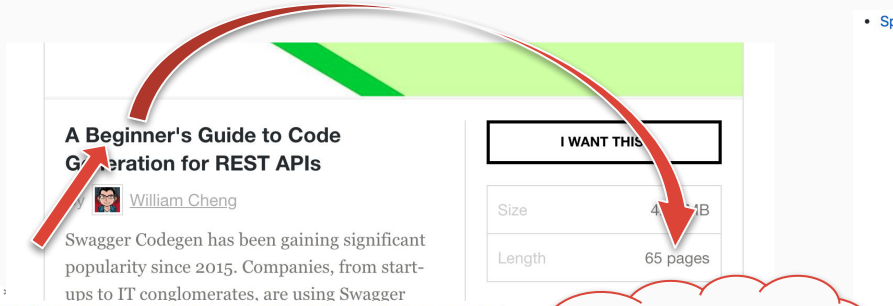
There are lots of reasons for code generation, but mine is around APIs.

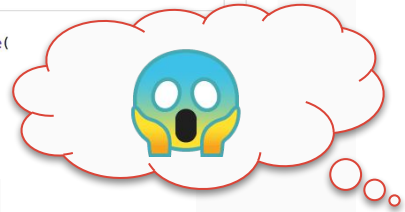
- Google produces a large number of APIs. [citation needed]
- It is prohibitively expensive to provide clients for all of them, and it leads to inconsistency and drift if we try.
- Benefits of code generation: Consistency, feature breadth, scale

Implementing Code Generators

Problem statement:

Get high-quality client libraries into the hands of your API's customers.





```
static createLongrunningrecognizeV1SpeechLongrunningrecognize(
    body,
    uploadProtocol,
    prettyPrint,
    fields,
    uploadType,
    mXgafv,
    callback,
    alt,
    accessToken,
    key,
    quotaUser,
    pp,
    bearerToken,
    oauthToken,
    callback) {
    // create empty cal
    const _callback =
```

Fifteen constructor arguments, ah ah ah!

```
def list_events(ca
  i_cal_uid: nil, max_att
  order_by: nil, page_token: nil, private_extended_property: nil,
  q: nil, shared_extended_property: nil, show_deleted: nil,
  show_hidden_invitations: nil, single_events: nil,
  sync_token: nil, time_max: nil, time_min: nil, time_zone: nil,
  updated_min: nil, fields: nil, quota_user: nil, user_ip: nil,
  options: nil, &block)
```

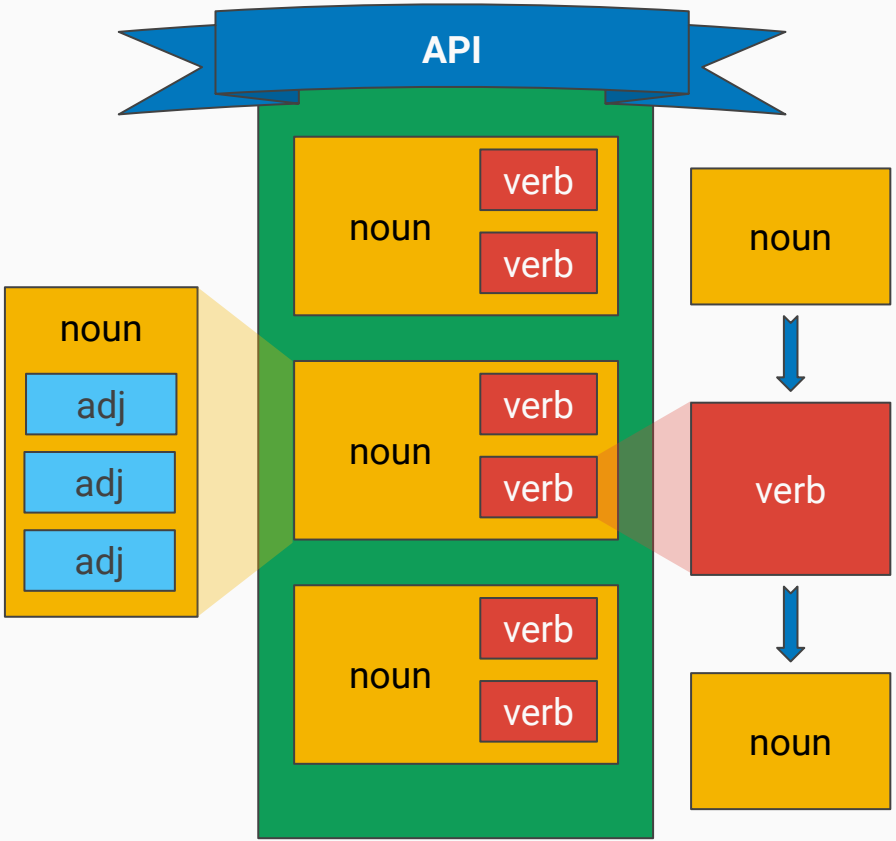
```
from googlecloudpubsubapi.googlecloudpubsubapi_client
    import googlecloudpubsubapiClient
```

topics.py 92 2 0 13:88

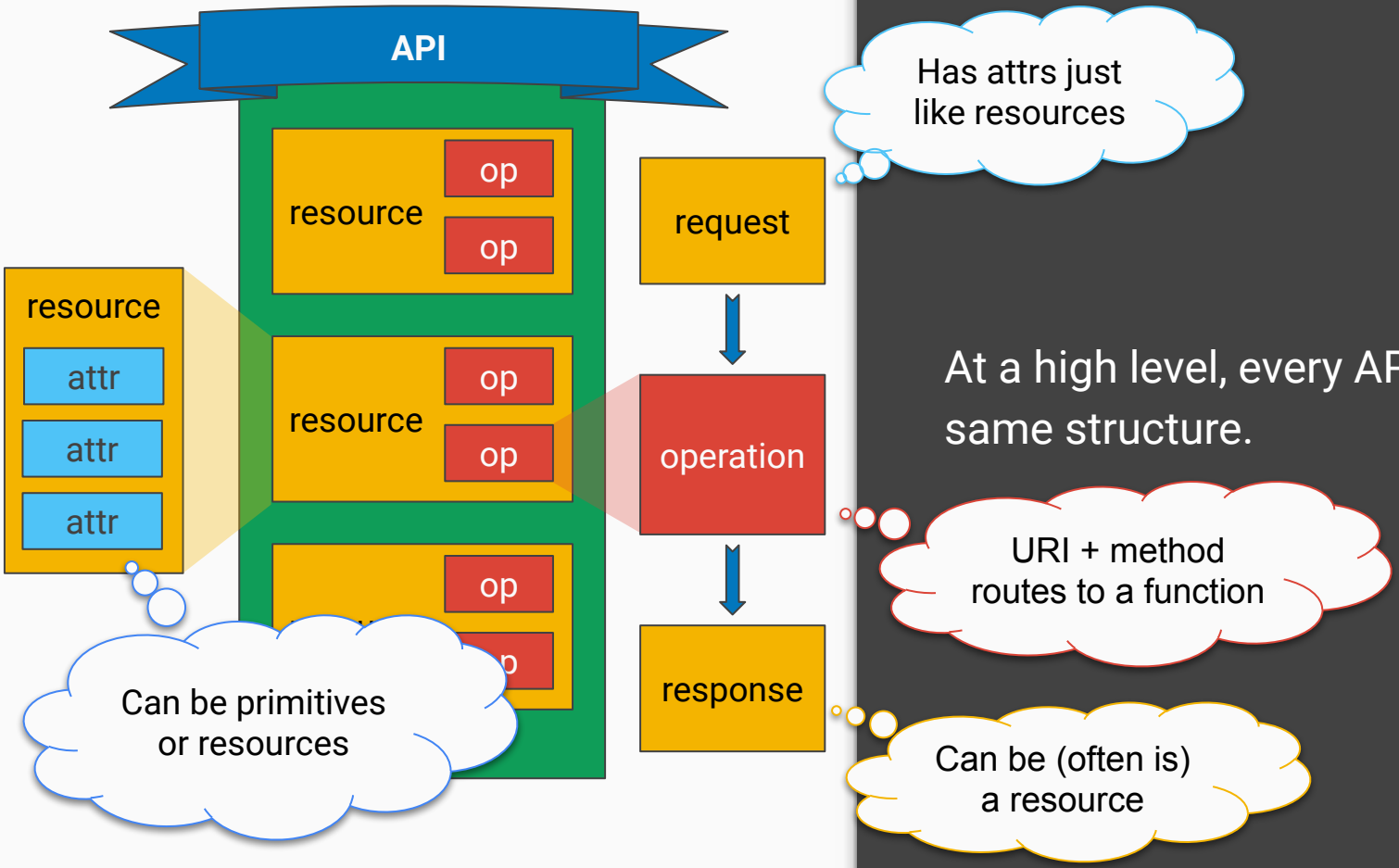
- Specification
  - Versions
  - Format
  - Document Structure
  - Data Types
  - Rich Text Formatting
  - Relative References In URLs
  - Schema
    - OpenAPI Object
    - Info Object
    - Contact Object
    - License Object
    - Server Object
    - Server Variable Object
    - Components Object
    - Paths Object
    - Path Item Object
    - Operation Object
    - External Documentation Object
    - Parameter Object
    - Request Body Object
    - Media Type Object
    - Encoding Object
    - Responses Object
    - Response Object
    - Callback Object
    - Example Object
    - Link Object
    - Header Object
    - Tag Object
    - Reference Object
    - Schema Object
    - Discriminator Object
    - XML Object
    - Security Scheme Object
    - OAuth Flows Object
    - OAuth Flow Object
    - Security Requirement Object
  - Specification Extensions
  - Security Filtering

Principle

Every API has the same structure.



At a high level, every API has the same structure.

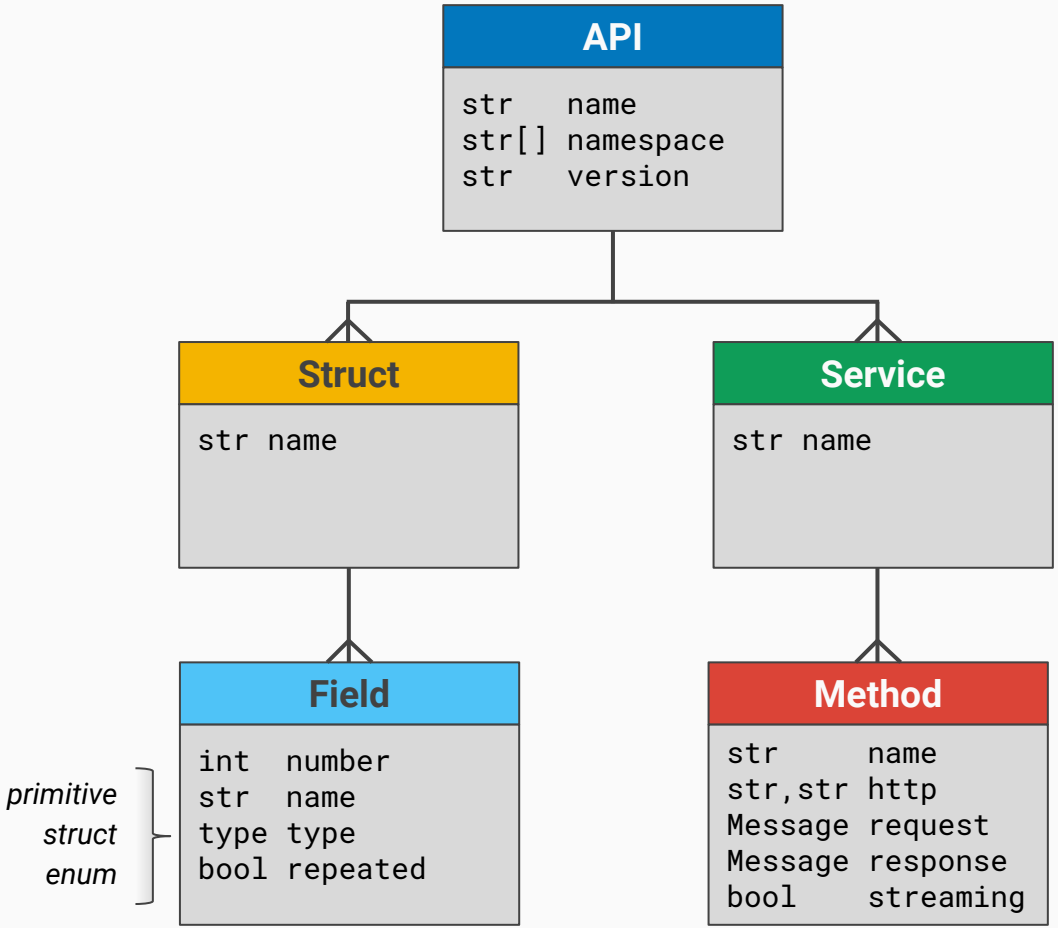


At a high level, every API has the same structure.

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.



Schema: Tips

- 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).

Principle

Separate schema from output.

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.

```
func (g *generator) clientOptions(serv *descriptor.ServiceDescriptorProto, servName string) error {
	p := g.printf

	// CallOptions struct
	{
		var maxNameLen int
		for _, m := range serv.Method {
			if l := len(*m.Name); maxNameLen < l {
				maxNameLen = l
			}
		}

		p("// %[1]sCallOptions contains the retry settings for each method of %[1]sClient.", servName)
		p("type %sCallOptions struct {", servName)
		for _, m := range serv.Method {
			p("%s%s[]gax.CallOption", *m.Name, spaces(maxNameLen-len(*m.Name)+1))
		}
		p("}")
		p("")

		g.imports[importSpec{"gax", "github.com/googleapis/gax-go"}] = true
	}

	// defaultClientOptions
	{
		eHost, err := proto.GetExtension(serv.Options, annotations.E_DefaultHost)
		if err != nil {
			return errors.E(err, "cannot read default host")
		}

		p("func default%sClientOptions() []option.ClientOption {", servName)
		p("  return []option.ClientOption{")
		p(`    option.WithEndpoint("%s:443"),`, *eHost.(*string))
		p("    option.WithScopes(DefaultAuthScopes()...),")
		p("  }")
		p("}")
		p("")

		g.imports[importSpec{path: "google.golang.org/api/option"}] = tru
	}

	// defaultCallOptions
	{
```

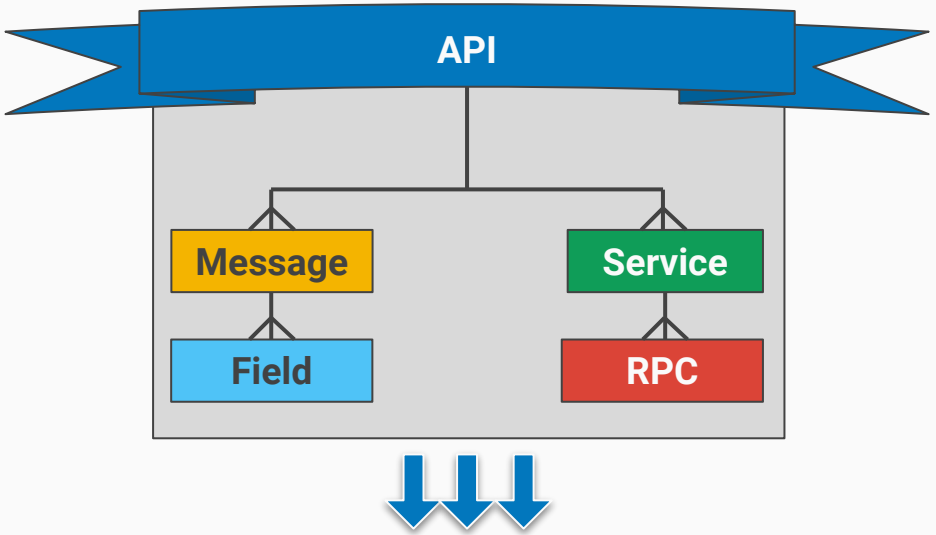
print

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

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.

```
    {% for method in service.methods.values() -%}
    {%- if method.signatures.single_dispatch -%}
    @dispatch
    {% endif -%}
    def {{ method.name|snake_case }}(self,
            request: {{ method.input.python_ident }}, *,
            retry: retry.Retry = None,
            timeout: float = None,
            metadata: Sequence[Tuple[str, str]] = (),
            ) -> {{ method.output.python_ident }}:
        """{{ method.meta.doc|wrap(width=72, offset=11, indent=8) }}

            Args:
                request ({{ method.input.sphinx_ident }}):
                    The request object.{{ ' ' -}}
                    {{ method.input.meta.doc|wrap(width=72, offset=36, indent=16) }}
                retry (~.retry.Retry): Designation of what errors, if any,
                    should be retried.
                timeout (float): The timeout for this request.
                metadata (Sequence[Tuple[str, str]]): Strings which should be
                    sent along with the request as metadata.

            Returns:
                {{ method.output.sphinx_ident }}:
                    {{ method.output.meta.doc|wrap(width=72, indent=16) }}
            """
            # Coerce the request to the protocol buffer object.
            if not isinstance(request, {{ method.input.python_ident }}):
                request = {{ method.input.python_ident }}(**request)

            # Wrap the RPC method; this adds retry and timeout information,
            # and friendly error handling.
            rpc = gapic_v1.method.wrap_method(
                self._transport.{{ method.name|snake_case }},
                default_retry=None,  # FIXME
                default_timeout=None,  # FIXME
                client_info=self.client_info,
            )
            {%- if method.field_headers %}

            # Certain fields should be provided within the metadata header;
            # add these here.
            metadata = tuple(metadata) + (
                gapic_v1.routing_header.to_grpc_metadata({
                    {%- for field_header in method.field_headers %}
                    '{{ field_header }}': request.{{ field_header }},
                    {%- endfor %}
                }),
            )
            {%- endif %}

            # Send the request.
            response = rpc(request, retry=retry,
                           timeout=timeout, metadata=metadata)
```

Output: Tips

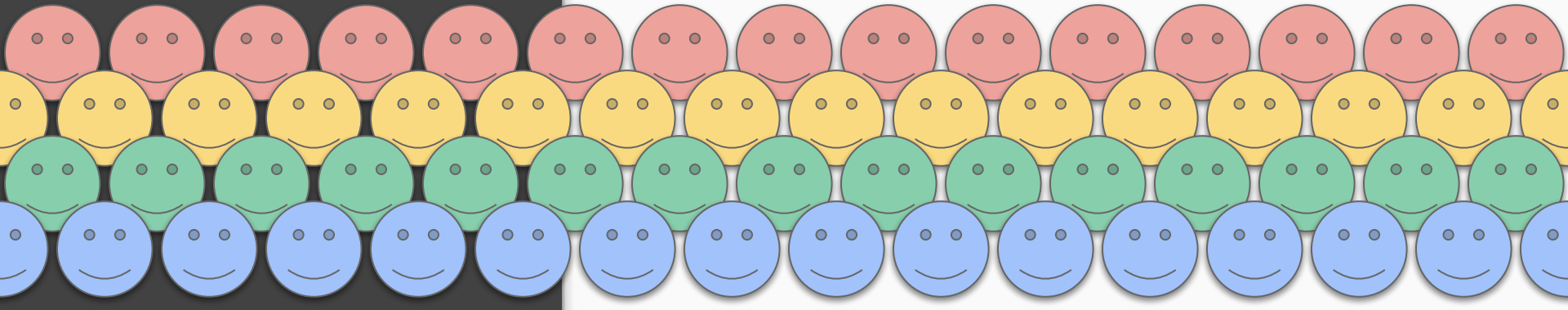
- All output-related code should be given the same data.
  - "If you understand any of the templates, you understand them all."
  - Slight exception: Output code that runs multiple times (in a loop) also must be told what is being iterated over.
- Use tooling designed for your target language. (Liberally!)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

Principle

Sanitize your inputs.

Consistency is hard.

With size comes a combinatorial explosion of communication channels.

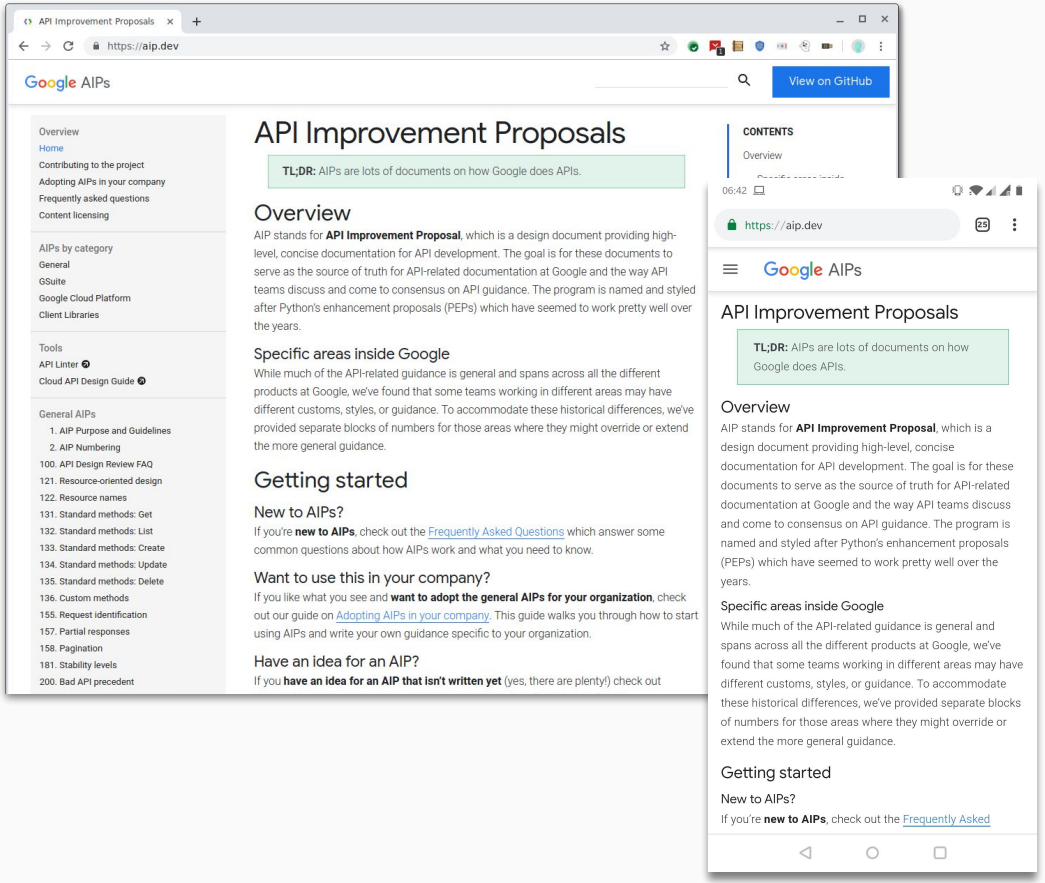


Benefits of consistent inputs

- Cognitive leverage.
- Ability to build meaningful, idiomatic features in clients that reinterpret common patterns.
- Ability to adopt new technology when it shows up and is useful.
- Learn from one another's mistakes.

Consistency: Tips

- Set up and enforce an API governance program.
- Document API standards.
- Adopt an API linter.



Challenge

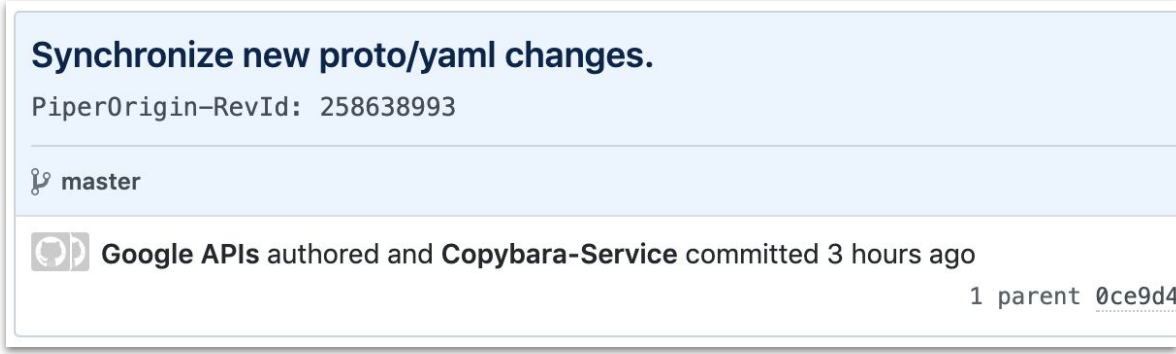
What got released anyway?

Release recording

Code generation is ordinarily part of a bigger, automated process.

The ultimate goal of that process is to go from the internal API surface to external API clients without a lot of human intervention.

But managing the sanitization and publishing of the API surface itself is difficult and error-prone.



Release recording

- Privately, surface changes are one of the **first** steps.
- Publicly, the surface change comes **last**.
- Approaches:
  - Specification changes live alongside implementation changes on branches.
  - Live-at-HEAD philosophy, with a mechanism to mark what part of the surface is at what implementation stage.

Lessons for release recording

Zero-cost principle:
At any non-trivial scale, you probably can not count on upstream providers to manually trigger any action in your system.

Challenge

Versioning is hard.

Versioning is hard.

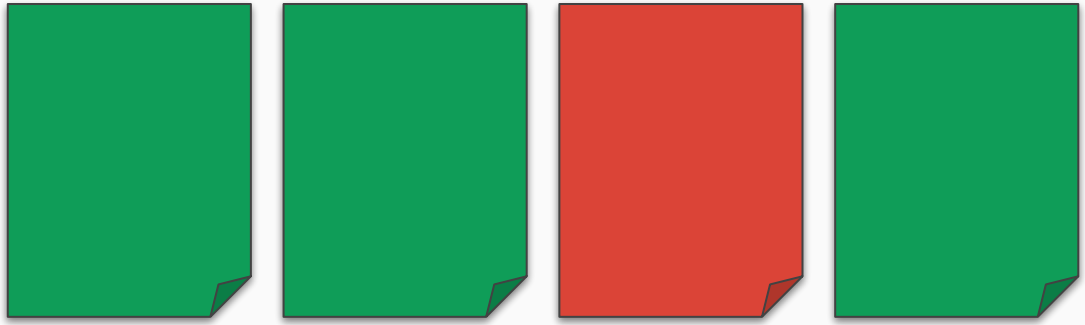
- How *do* you version automatically-generated libraries?
  - If the surface makes a backwards-incompatible change by mistake, do you make a semver-major release? (If so, how do you automate that?)
  - What about when it is correcting unusable surface?
  - Pass-through principle?
- Do you distinguish API changes from client changes?
- Common runtime dependencies can be very frustrating to upgrade, leading to release-the-world scenarios.

Lessons for versioning

If you want to use semver, you must be able to reason about the state of your releases.

You probably want to be a little bit forgiving about semver when it comes to mistakes.

Stabilize your dependencies early.



Common versioning

Is it useful to use a common version indicator across products intended for the same ecosystem?

language 1.2.0

speech 1.1.0

translate 1.6.0

video 1.9.0

vision 0.38.0

Common versioning

Is it useful to use a common version indicator for the same product across multiple ecosystems?

`translate` `1.6.0`

`translate` `4.1.1`

`translate` Versions? 🤯

`translate` `0.20.0`

`translate` `1.82.0`

Challenge

Code vs. packaging

Code vs. packaging

- In theory, a code generator can be used equally by anyone who sticks to the input format. Package generation needs seem to diverge wildly.
- Packaging decisions include:
  - Licensing
  - Formatters
  - CI/CD setup
  - ...all of which are likely to vary widely between every potential user.

Code vs. packaging

This is a classic tradeoff.

It is simpler to keep code and packaging together, but limits how many people can use the tools.

It is more complicated to separate them, but permits wider adoption.

Review

- Every API has the same structure, and features in your schema format are costly mandates.
- Schema and output are distinct concerns.
- Sanitize your inputs to promote better tools, and a richer user experience
- Automation reduces knowledge of the nature of changes to inputs, guarantee of correctness.
- Versioning is hard.
- Code generation concerns are widely reusable, package generation concerns are not.

Code Generation

Principles & Challenges

Luke Sneeringer
lukesneeringer@google.com

OSCON 2019