Multicloud API binding generation from documentation

Michał J. Gajda[1], Vitor Vitali Barrozzi[1], and Gabriel Araujo[1]

Migamake Pte Ltd
`migamake@migamake.com`

1 Introduction

We propose a solution for low-level integration cloud service integration among different providers without additional support from the cloud API vendor.

Multicloud service orchestration allows leveraging the full power of specialized, scalable best-of-a-kind services in the cloud. Its advent coincides with the so-called *no-code* and *low-code* solutions that enable rapid prototyping of cloud applications by small teams of startup companies. While excellent in theory, but in practice SDKs usually address only a few programming languages, descriptions are untyped and incomplete, API description languages like Swagger/OpenAPI[20] are rarely used. That means that initial implementation is often an incoherent mix of incompatible scripts, deployment is only partly automated, and the source code uses a variety of platforms with varying level of automation.

An attempt to integrate these stumbles upon the barrier of cloud APIs documented in many different ways, usually *ad-hoc* and without particular rigour. The API interface bindings often contain implicit assumptions, untyped JSON or text-based bindings.

We solve this significant and essential interoperability problem by automatically parsing API documentation, and then generating API bindings in any chosen programming language[1]. The additional efforts to support cloud API bindings by code generation are limited to single API and single language[1, 5, 11], or require a significant effort of handwriting Swagger/OpenAPI declarations for entire API[30], if the vendor did not generate them.

2 Implementation

Our solution (*HurlAPI*), is a *data pipeline*, divided into three different stages: (I) data gathering from the web pages (II) parsing and analysis (III) code generation.

---

[1] We currently generate Haskell[12] code, but developing generation to any other typed language is offered as a paid service upon request.

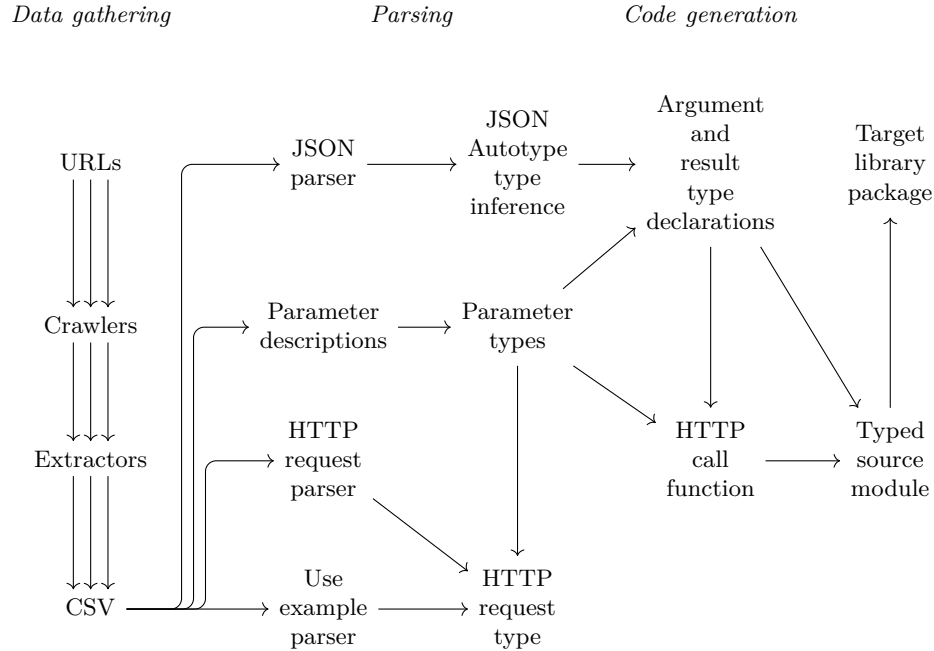*Data gathering*   *Parsing*   *Code generation*

Fig. 1: Data flow within the API binding code generation pipeline.

Data gathering At the initial stage of *data gathering*, we gain a complete description of each cloud API call with Scrapy library[24] using Python. It allows us to download HTML pages with Chrome, then we examine page structure with XPath[6], CSS[25], or JQ[14] selectors to extract data in a systematic manner. All this information is written into tabular CSV files[26], with some columns containing JSON objects, and carefully validated.

Data analysis As part of a *data analysis* stage implemented in Haskell[12], we parse many possible formats: (a) HTTP request path description with variables (b) cURL command options (3) extracts from tables of parameter descriptions. The parameter descriptions are tagged with the *parameter passing convention*[2]: (I) as part of an HTTP request path (II) URL-encoded query parameter (3) part of a request body as JSON or plain text[10] (4) HTTP header or a cookie. The content of parsed entries is carefully validated and cross-checked for possible inconsistencies. Every entry has a separate list of errors, that are reported per-record. While we only allow 100% correct records to be used for code generation, the failed records are reported in detail. Summary statistics of erroneous records

---

[2] Cloud API call parameter passing conventions differ from binary function call conventions in this respect, and many different argument passing convention may be assigned for different parameters of the same call.

reports on a *validation dashboard* that indicates percentage correctness of current data and allows us to assess the overall health of data pipeline[9].

Agile data pipeline principles Data analytics pipeline goes beyond previously described best practices in agile data science[16] and also draws from BCBS 239 best practices for risk management in the financial industry[4].

The principles of our data pipeline development process are (1) *judge by a final impact* – prioritize development of an entire pipeline to judge issues by their impact on a final product using data processing dashboard; it allows us to focus our efforts on few issues that have a significant impact on final data[3] (2) *record never disappears* – trace flow of records over the entire data pipeline with unique record identifers[4]; when filtering out records, put them to alternate output, so you can examine impact of each filter (3) *error is a tag or an alternate output* – assign error as one of many *tags* of the record, and then filter by sorting error records to an alternative output that requires similar examination as final product; multiple errors and warnings are attached to each record that elucidate co-occurrence of data quality and handling issues (4) *late filtering* – delay filtering when you have multiple data quality criteria that can be run in parallel on a single record; this makes it easy to examine issue co-occurrence that is common for faulty data (5) *universal data formats* – data at any stage of the pipeline is available for examination as CSV files (6) *gradual record enrichment* with additional information, so we can examine all data related to the record in a single row; preserve of existing information, so inputs and outputs can be quickly examined at any stage of processing (7) an *iteration throughput* is considered as important as an iteration speed, since number of successfully processed records increases a number of issues discovered during each iteration, and we try to make the processing of different records and categories as independently as possible (8) *tagging potential gaps* with errors or warnings, instead of assuming total correctness of the input data and the processing pipeline; this facilitates data-based assessments of completeness of the analysis (9) use excerpts from *real data as unit tests* whenever possible to avoid testing for issues rarely or never occur in practice.[5]

The principles (1), (7-9) are all guided by *Zeno's principle*[13] of extensive data processing, where sorting data quality and processing issues by the final impact on the final product will show that most of the issues occur in a relatively small number of records. Fixing the first issues gives big improvements, but getting to 100% accuracy needs much more work. We can easily observe that moving from 80% to 90% of correctly processed records takes about the same time as moving from 98% to 99%. Still, the gain in the former case is more

---

[3] Only in the absence of more precise goals, the primary measure of impact is a frequency of an issue.

[4] In case records are merged, we also merge the identifiers.

[5] It also allows for principled treatment of the potential problems for which we do not have any practical test examples. We firmly anchor our analysis in naturally occurring data.

substantial.[6] The principles (3-5) aim to increase iteration throughput in terms of simultaneously detected data quality and processing issues.

Code generation As the final stage, we *generate code* in a typed programming language. We start with a reference data structure that lists necessary functions and type declarations. This part is language-independent except for function generating the language-dependent declaration identifiers themselves. The binding generation proceeds with templates that use these identifiers to generate full code modules, and then entire API binding package along with its metadata; we use techniques described in [2, 8, 15, 18]. Following the best current practice, we also attach links to the original documentation website, which allows user to cross-reference the information with the original API documentation.

3   Conclusion

Limitations When proposing a code generation solution instead of handwritten code, it is important to consider limitations compared to manual processes. For a few APIs, we need to implement specialized components like AWS S3 chunked signatures[27], custom authentication rolled out for TransferWise API[29].[7] There are also few (less than 2% in entire MS Graph API[19]) of API calls that use custom argument passing. For example, MS Graph uses custom DSL for filtering by customizable *extended properties*[22]. Another example is non-standard retry behaviour of Backblaze API[3], that requires to replace the access token upon receiving 503 response (service unavailable). There are also sometimes bugs in the documentation, which will cause the generation of incorrect code. Some companies provide only language-specific SDKs[28], instead of publicly documenting their REST interfaces. Luckily the situation improves with bigger companies even providing live debugging or live sandbox functionality for the REST interface[7, 17, 21].

Summary We implemented the retargetable code generator for cloud API bindings that presents the following benefits: (1) provide a binding for thousands of API calls within months; (2) language retargeting with little effort; (3) the systematic approach allows easy scaling to a number APIs; (4) removes a dependency on the cloud API provider support; (5) it significantly reduces maintained code base as compared with handwritten cloud API bindings. We offer to generate cloud API bindings for other programming languages and other cloud APIs as a paid service.

---

[6] Which is kind of inverse Pareto[23] principle in data analytics. We can call it *Zeno's principle* after the behaviour of the turtle in the Zeno's paradox[13], since we move in smaller steps the closer we are to the complete correctness.

[7] We agree with TransferWise[29], that this is more secure than using plain secret like an access token, but each new authentication method needs special support code.

Bibliography

[1] Amazonka – A comprehensive Amazon Web Services SDK for Haskell: 2013. *https://github.com/brendanhay/amazonka.*

[2] Art of industrial code generation: *https://www.migamake.com/presi/art-of-industrial-code-generation-mar-6-2019-uog-singapore.pdf.*

[3] B2 Integration Checklist, Uploading files: *https://www.backblaze.com/b2/docs/integration_checklist.html.*

[4] Basel Committee on Banking Supervision 2013. BCBS 239: Principles for effective risk aggregation and risk reporting.

[5] Boto 3 - The AWS SDK for Python, release 1.12.9: 2020. *https://github.com/boto/boto3.*

[6] Clark, J. and (eds.), S.D. 1999. *XML path language (XPath) version 1.0.* W3C.

[7] Full API Reference, Simulation: *https://api-docs.transferwise.com/#simulation.*

[8] Gajda, M.J. Do not give us a bad name. Work in progress.

[9] Gajda, M.J. Guidelines for agile data pipelines. Work in progress; Migamake Pte Ltd.

[10] Gajda, M.J. Towards a more perfect union type. 10.5281/zenodo.3929473.

[11] Gogol – A comprehensive Google Services SDK for Haskell: 2015. *https://github.com/brendanhay/gogol.*

[12] Haskell 2010 Language Report: *https://www.haskell.org/definition/haskell2010.pdf.*

[13] Huggett, N. 2019. Zeno's Paradoxes. *The Stanford Encyclopedia of Philosophy.* Edward N. Zalta, ed. https://plato.stanford.edu/archives/win2019/entries/paradox-zeno/; Metaphysics Research Lab, Stanford University.

[14] jq is a lightweight and flexible command-line JSON processor: 2012. *https://stedolan.github.io/jq/manual/.*

[15] JSON Autotype: Presentation for Haskell.SG: 2015. *https://engineers.sg/video/json-autotype-1-0-haskell-sg--429.*

[16] Jurney, R. 2017. *Agile data science 2.0: Building full-stack data analytics applications with spark.* O'Reilly Media, Inc.

[17] Mailgun Sandbox Domain Explained: 2020. *https://blog.mailtrap.io/mailgun-sandbox-tutorial/.*

[18] Michal J. Gajda, D.K. 2020. Fast XML/HTML tools for Haskell: XML Typelift and improved Xeno. Manuscript in review, 10.5281/zenodo.3929548.

[19] Microsoft Graph REST API v1.0 reference: *https://docs.microsoft.com/en-us/graph/api/resources/domain?view=graph-rest-1.0.*

[20] OpenAPI 3.0.2 Specification: 2018. *https://swagger.io/docs/specification/about/.*

[21] OpenAPI Explorer: *https://api.alibabacloud.com/.*

[22] Outlook extended properties overview: 2020. *https://docs.microsoft.com/en-us/graph/api/resources/extended-properties-overview?view=graph-rest-1.0.*

[23] Pareto, V. 1896. *Cours d'Economie politique.* Droz.

[24] Scrapy: *https://scrapy.org/.*

[25] Selectors Level 3: 2011. *https://www.w3.org/TR/2011/REC-css3-selectors-20110929/*.

[26] Shafranovich, Y. 2005. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180; RFC Editor.

[27] Signature Calculations for the Authorization Header: Transferring Payload in Multiple Chunks (Chunked Upload) (AWS Signature Version 4): 2016. *https://docs.aws.amazon.com/AmazonS3/latest/API/sigv4-streaming.html#sigv4-chunked-body-definition*.

[28] SmartFoxServer 2X documentation central: 2020. *http://docs2x.smartfoxserver.com/*.

[29] Strong customer authentication: 2020. *https://api-docs.transferwise.com/#payouts-guide-api-access*.

[30] SwaggerHub: *https://swagger.io/tools/swaggerhub/*.