RAMLFlask: Managing artifact coupling for Web APIs

Stefan Sobernig
WU Vienna
Vienna, Austria
stefan.sobernig@wu.ac.at

Michael Maurer
Independent Developer
mmaurer.at@gmail.com

Mark Strembeck
WU Vienna
Vienna, Austria
mark.strembeck@wu.ac.at

ABSTRACT

This paper documents a systematic approach (RAMLFlask) to extending Web application frameworks (Flask) to include support for interface-description languages (IDLs such as RAML) and code generation.

KEYWORDS

Web engineering, Web application integration, artifact coupling, interface description, application generator, Flask, RAML

ACM Reference Format:
Stefan Sobernig, Michael Maurer, and Mark Strembeck. 2020. RAMLFlask: Managing artifact coupling for Web APIs. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3341105.3374116

1 INTRODUCTION

A Web application-programming interface (Web API) defines a contract between client-side and server-side blocks of a Web application in terms of the functionality offered and used, respectively, by means of HTTP—i.e., resources and the HTTP virtual machine.

Web APIs and the so-provided Web services are typically implemented on top of a Web application framework (e.g., Flask). To this end, defining and implementing a Web API often involves code that is boilerplate and scattered. This boilerplate code is incurred by the respective framework (e.g., Flask, Eclipse Jersey) to implement the API on top of HTTP. This requires very similar or even identical code sections (Python function decorators, JAX-RS annotations) to be included at many places of a code document (see Listing 1).

Web APIs and their implementations are often subjected to change running in parallel with changes in application code. In particular upon high frequency of change and/ or for larger Web APIs, tracking changes at the level of boilerplate code does incur extra maintenance effort. For this paper, we reviewed the change history of selected Web APIs and found that, for these projects, more than 50% of commits made to API definition documents during the review periods affected the boilerplate code (decorators, annotations).

Moreover, Web APIs have multiple technical and non-technical stakeholder roles during development of a Web application. Development involves communication and close, document-based interactions between technical and non-technical stakeholder roles. To tackle the above issues, the pattern of providing and maintaining interface descriptions [9] has been applied to Web APIs. On the downside, an interface-description document for a Web API introduces the complexity of managing co-changes [4] between documents. This paper makes the following contributions:

- A critical-analytical comparison of established generative techniques for their fit to manage artifact coupling (incl. propagating changes) in an automated manner between an interface description (RAML) and the generated Web API (Flask) code (see Section 4.1).
- A proof-of-concept implementation of a *mixed* generative technique (GENERATION GAP and delegation) for RAML documents and Flask-based Web API implementations (see Section 4.2).
- A twofold validation of the proof-of-concept implementation is performed. Second, the generator implementation is tested for its space and time performance on real-world Web APIs (see Section 5.1). Second, the chosen technique is tested for its coverage of typical changes to Web APIs (see Section 5.2).

```
@app.route('/status/<order_id>', methods=['GET', 'POST'])
def order_status(order_id):
    if request.method == 'POST':
        # Retrieves the status of a specific order
        set_order_status(order_id, request.form['statusCode'])
        return
    else:
        # Sets the status of a specific order
        return Response(get_order_status(order_id),
                            mimetype='application/json')
```

Listing 1: An exemplary Web service `order_status` using `Flask` and its `@app.route` decorator.

```
/status:
  /{order_id}:
    get:
      description: Retrieve the status of a specific order
      responses:
        200:
          body:
            application/json:
    post:
       description: Sets the status of a specific order
       body:
         multipart/form-data:
          properties:
            statusCode:
              description: The status identifier to be stored
              required: true
       responses:
          200:
```

Listing 2: An interface description using RAML corresponding to the Web API defined in Listing 1

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03.
https://doi.org/10.1145/3341105.3374116

The proof-of-concept implementation plus test suite, as well as a reproduction package incl. the collected interface descriptions is available for download.[1] An extended version of this paper is also available [8].

2 WEB APIS

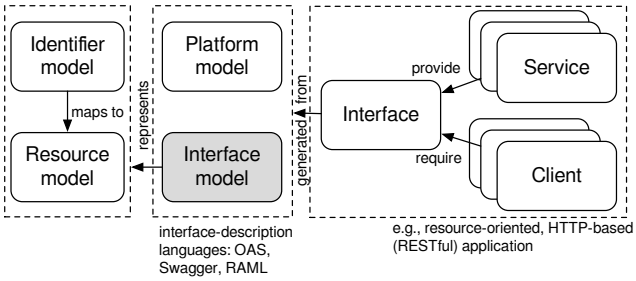

Figure 3: Main tenets of a resource-oriented Web application

Interface Descriptions. In a distributed system, an INTERFACE DESCRIPTION [9] defines the interface that is provided by service applications and required by client applications. The interface is described in a platform-independent and machine-processable manner. Interface descriptions reflect the type of interfaces contracted between clients and services, e.g., signature interfaces for W3C Web Services (based on operations as well as input and output message types) or resource-oriented interfaces for RESTful services.

Self-description vs. explicit interface description. A *self-describing* API allows for answering questions by investigating responses to calling API endpoints (e.g., HTTP header and payload). In terms of development style, an API must be fully implemented to realise the property of self-description [5]. An explicit interface description can be useful independent from an interface implementation [6], e.g., for generating server- or client-side code skeletons, endpoint tests, an API reference manual etc.

3 ARTIFACT COUPLING

Generated artifacts. An explicit interface description for a Web API written using RAML or OAS will be used to generate different types of development artifacts. From generative programming and model-driven software development, more generally, there is empirical evidence on the relative importance of certain artifact types as generator targets for practitioners. There is evidence on M2T transformations targeting source code (e.g., Flask/ Python scripts in Figure 4) being the most widely targeted artifact type for generators in model-driven approaches [1]. This is confirmed by a recent study of ours on M2T transformations for UML-based domain-specific modeling languages [7] and a survey among experts on domain-specific modeling [3].

Artifact coupling. The use of an explicit interface description plus generator creates a collection of *coupled* development artifacts [4]. The RAML document in Listing 2 and the (generated) Flask/ Python script in Listing 1 form one such a collection. A collection could also include a generated test suite, client-side stub code, and documentation strings (see above). Changing one artifact affects the other artifacts in this collection. Figure 4 illustrates such a change, turning a RAML interface description `i1` into `i2`.

[1] https://github.com/nm-wu/RAMLFlask

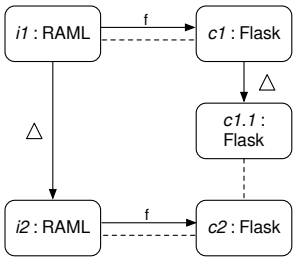

Figure 4: Rectangles represent development artifacts conforming to a software language (RAML, Flask/Python), and arrow-headed solid lines are changes. f denotes a mapping function (model-to-text transformation), $\triangle$ a manual modification to an artifact, performed by a developer. Dashed lines denote a consistency relationship. Notation inspired by [4].

4 RAMLFLASK

For the proof-of-concept implementation `RAMLFlask`, the Python Web-application framework `Flask` was adopted and refined to include support for generating API skeletons from interface descriptions written using RAML.

`RAMLFlask` realises a *template-based* code generator. As a generator, it accepts RAML documents as input and produces a Flask/ Python script implementing the HTTP-based interface described by the RAML document. The generator is based on model-to-text templates as composable and refinable code assets that encode the mapping between RAML description elements and Flask framework elements (see Table 1). The resulting code structure of the Flask-based skeleton is depicted in Figure 5. The generator templates are implemented using the Jinja2 template language and template processor.

RAML	RAMLFlask/ Flask
Interface	Blueprint
Resource	Request-handler class
HTTP method	Request-handler method
Security scheme (incl. default)	Delegation class
Data type	Validation routine (built-in)
Annotation	In-code comment

Table 1: The basic mapping between RAML description elements and Flask implementation elements.

4.1 Managing artifact coupling

The challenge is to manage the coupling of co-changing artifacts in a way allowing for concurrent changes to the interface description *and* changes to the generated code artifacts, while maintaining the overall consistency (see Section 3). The crux of this challenge is that there exist multiple different available techniques to tackle the challenge. The array of techniques must be assessed against

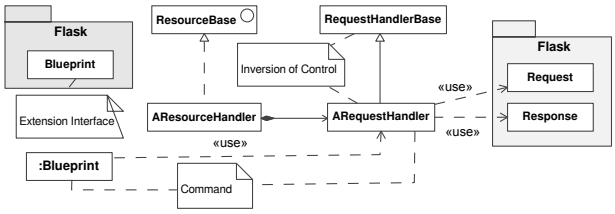

Figure 5: Structural overview of the design artifacts generated from a single RAML interface description by `RAMLFlask`. The UML comments depict the three important architectural design patterns realised by the `RAMLFlask` generator: EXTENSION INTERFACE, COMMAND, and INVERSION OF CONTROL.

important decision criteria [2]: separation of generated and curated code (C1), extensibility (C2), support for overriding generated parts (C2.1), support for adding to the generated parts (C2.2), portability (C3), and against the background of a given generator architecture (RAMLFlask; see Figure 5).

	GENERATION GAP (L)	EXTENDED GEN. GAP (L)	delegation (L)	include (L)	partial classes (L)	AOP (L)	PartMerger (T)	protected regions (T)
C1	+	+	+	+	+	+	+	–
C2.1	+	+	–	–	~	+	~	–
C2.2	–	+	–	~	~	+	~	+
C3	+	+	+	–	–	+	+	+

Table 2: Comparing language- and tool-based techniques following [2] (C1-C3). Legend: + (supported); ~ (partially supported); – (not support); T (tool-based); L (language-based)

RAMLFlask was designed to strive for supporting critical development tasks regarding Web APIs and their interface descriptions as identified by 14 expert interviews [8]. These tasks are generative support for route implementation (on top of Flask), for security-concern implementation, and for validation routines targeting invocation data.

4.2 Design & implementation

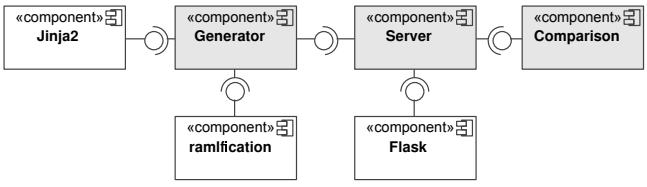

Figure 6: RAMLFlask components

Structure. The main components of RAMLFlask and their relationships are depicted in Figure 6. RAMLFlask is designed as an extension to the Web-application framework Flask[2]. Flask hosts the code generated by RAMLFlask as a Flask application. The `Generator` generates the structures detailed in Section 4: a blueprint, route decorators and handlers, and utility code. For this, `Generator` uses the Jinja2 templating engine. To process a RAML description into a Python data structure, RAMLFlask integrates with ramlfications, a third-party RAML parser.

Behaviour: Generation. The skeleton in Figure 5 is created via a few key operations. First, RAMLFlask creates three directories to host the different types of generated code. This includes folders for generated and handwritten routes, for delegates, and for version comparison artifacts. Second, the resource and request-handler classes are created. This is achieved by iterating the RAML document for the defined resources and for the validation details (incl. checks on return types).

5 DISCUSSION

5.1 Time and space performance

We measured time and space performance of the most important generative tasks supported by RAMLFlask (RAML/ YAML parsing, creation of server stubs, route implementations etc.) on a machine equipped with the Intel Core i7 CPU, a 2,8 GHz processor, and 16GB RAM, running macOS. We used CPython 2.7.16 and Flask 1.0.1 as the target platform. All test runs were performed under the CPython's default configuration.

As for the design of the computational experiment, time (execution timings) and space usage (RAM) were collected for 10 publicly documented Web APIs (incl. GitHub, Instagram, and Gmail), each described by a single RAML document (see Table 3).

Web API	#Routes	#SLOC
Grooveshark	1	143
Flickr	4	111
Uber	13	1185
Slideshare	17	3114
Slack	29	1896
Gmail	31	2120
Instagram	33	3379
Wordpress	60	2606
Box	66	4451
GitHub	223	21650

Table 3: Overview of the 10 Web APIs used.

Figure 7 (top) summarises the main findings on time performance. First, the all but RAML document were processed in below half a second (0.5s) of total execution time to generate a complete and operative implementation using RAMLFlask. GitHub, the largest API in our corpus incl. 223 resources, ran for 2.5s (as denoted by the triangle in Figure 7). Second, the total execution times are determined by a single task: the initial server generation including RAML parsing (using ramlification). See the dotted line of execution times for this task in Figure 7.

Memory usage was measured as the maximum memory allocated by the operating system (OS) over regular intervals of 5ms for the duration of all generative steps (see Figure 7, bottom). This is

[2]https://palletsprojects.com/p/flask/

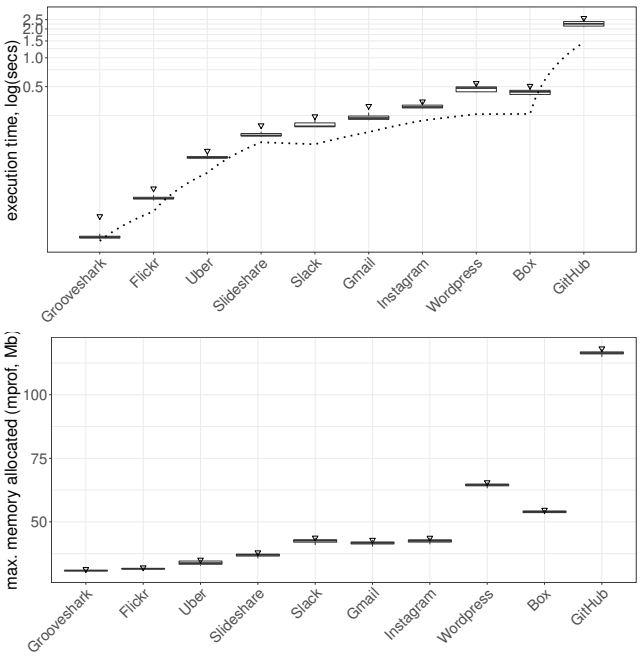

Figure 7: Comparative boxplots for the total execution times (top; in log(seconds)) and memory consumption (bottom; in MB) for processing 10 different Web APIs (100 runs). Both: Triangles denote the maximum total execution time/ memory consumed per Web API. Top: The dotted line represents the worst-case (maximum) execution time for the basic server-generation task (incl. RAML parsing).

because this maximum memory allocated represents the worst case from the blackbox perspective of the OS (neglecting garbage collection). All projects except for GitHub were observed to consume less than 75MB; GitHub required less than 120MB.

5.2 Replay simulations

To collect a corpus of changes to Web APIs—esp. their route definitions—the public source-code repositories (GitHub) of two real-world, Flask-based projects were mined for their change histories: HTTPbin and Sync Engine. This repository mining allowed for a twofold: On the one hand, a corpus of development artifacts was established, allowing for replaying critical changes on a Web API using RAMLFlask. On the other hand, first evidence on types and frequency of changes from the field of Web APIs were collected.

The coded change data was summarised. This descriptive analysis delivered strong support for RAMLFlask's features: The majority of resources (routes) has been changed at least once, 70% (35/50) for HTTPbin and 72.7% (64/88) for Sync Engine. The remainder has never experienced a change event at all, or only code changes were recorded. If defined by a RAML interface description, each change to a route requires a re-generation of the server-side skeleton. From the total of 151 commits reviewed, 26 (or, 17.2%) involved simultaneous modifications to interface descriptions and curated code. For Sync Engine, the number of co-changes in commits was only 17 out of 234 (7.3%). Such *co-changes* to interface descriptions and previously generated, but by now, manually maintained code may create inconsistencies.

We systematically selected three routes from HTTPbin and Sync Engine that had been subjected to the maximum of interface-change types (route additions, removal, modifications) among all routes. These three routes and their route change history were found to be representative of 40% of all routes (both projects combined) and of more than 90% when allowing for partial overlaps.

These three routes and their Flask implementations were then turned in RAMLFlask implementations. Then, the recorded interface changes as well as code changes were applied to create snapshots both of the generated and the source files (blueprint, handlers, and application subclasses). These can be used to replay the change history of the three routes in a stepwise manner.

6 CONCLUSION

This paper documents RAMLFlask as an extension to the Flask Web application framework. RAMLFlask provides a template-based generator capable of producing an application skeleton from RAML interface descriptions. In addition, RAMLFlask guides a developer by highlighting inconsistencies (e.g., as a todos list) when an interface description changes and/ or previously generated code has been modified. The design and implementation have been systematically derived from empirical evidence collected from 14 expert interviews [8] and from mining of change history of two real-world Web service projects. In addition, the RAMLFlask research prototype was exercised on real-world Web API descriptions for its time and space performance (incl. APIs of GitHub, Wordpress, and Instagram).

REFERENCES

[1] Generative Software. 2010. *Umfrage zu Verbreitung und Einsatz modellgetriebener Softwareentwicklung.* Survey Rep. Generative Software GmbH and FZI Forschungszentrum Informatik. http://www.mdsd-umfrage.de/mdsd-report-2010.pdf

[2] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, Bernhard Rumpe, Martin Schindler, and Andreas Wortmann. 2015. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development.* CCIS, Vol. 580. Springer, 112–132. https://doi.org/10.1007/978-3-319-27869-8_7

[3] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. 2017. Reusable and generic design decisions for developing UML-based domain-specific languages. *Information and Software Technology* 92 (July 2017), 49–74. https://doi.org/10.1016/j.infsof.2017.07.008

[4] Ralf Lämmel. 2016. Coupled Software Transformations Revisited. In *Proc. 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE'16).* ACM, 239–252. https://doi.org/10.1145/2997364.2997366

[5] Luca Panziera and Flavio De Paoli. 2013. A framework for self-descriptive RESTful services. In *Companion Proc. 22nd International Conference on World Wide Web (WWW'13).* ACM, 1407–1414. https://doi.org/10.1145/2487788.2488183

[6] Jonathan Robie, Rob Cavicchio, Rémon Sinnema, and Erik Wilde. 2013. RESTful Service Description Language (RSDL), Describing RESTful services without tight coupling. *Balisage Series on Markup Technologies* 10 (2013), 6–9.

[7] Stefan Sobernig, Bernhard Hoisl, and Mark Strembeck. 2016. Extracting reusable design decisions for UML-based domain-specific languages: A multi-method study. *Journal of Systems and Software* 113 (2016), 140–172. https://doi.org/10.1016/j.jss.2015.11.037

[8] Stefan Sobernig, Michael Maurer, and Mark Strembeck. 2019. *RAMLFlask: Managing artifact coupling for Web APIs.* Technical Report 7367. WU Vienna. https://epub.wu.ac.at/id/eprint/7367

[9] Markus Völter, Michael Kircher, and Uwe Zdun. 2005. *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware.* John Wiley & Sons.