ePub^{WU} Institutional Repository

Stefan Sobernig and Michael Maurer and Mark Strembeck

RAMLFlask: Managing artifact coupling for Web APIs

Paper

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

Modern Web applications rely more and more on communication with Web services for providing their composite functionality. Currently, one of the most popular method of providing Web services is via HTTP-based APIs—a.k.a. Web APIs with or without Representational State Transfer. Such services will usually be implemented in some general purpose language (e.g., Python) and involve several stakeholders for development and for maintenance (e.g., developers, testers, architects). Enabling effective communication and development between all of these stakeholders can be difficult though. Software languages specific to describing service interfaces allow for providing a further level of abstraction, which can improve communication between stakeholders while also making development under evolution less prone to error. 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. The contributions include a systematic elicitation to requirements on interface descriptions and needs for managing coupled development artifacts from the perspective of developers of Web services and composite Web applications. This is based on an overview of relevant literature, complemented by expert interviews with professional Web developers.

KEYWORDS

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

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

Listing 1: An exemplary Web service `order_status` implemented using Flask exposed via an HTTP-based API: `@app.route()`. In Flask, `@app.route` signals the use of a *function decorator* to define a route and to bind it to a handler function.

1 INTRODUCTION

Building a Web application involves exposing server-side application functionality as services accessible via Web APIs, to be incorporated by client-side building blocks. A Web application-programming interface (Web API) defines the 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. A Web API can contract additional details depending on their architectural context. This context typically imposes additional architectural constraints on a Web service and its API. The constraints follow from the architectural styles used, such as Representational State Transfer (REST, esp. HATEOAS) or microservices.

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 to implement the API on top of HTTP, which requires highly similar code sections to be included at many places of a code document. The Python example in Listing 1 showcases a minimal Web service implementation for managing order status using Flask. As an example of boilerplate, there is annotation code (on line 4) for each and every route to enable request handling by the responsible application code (e.g., some getter and setter functions) based on the HTTP method used (GET, POST). In addition, the interface details (HTTP methods, parameter representation, content type) appear scattered over the application code (see, e.g., lines 7 and 10). Boilerplate code and code scattering are tedious to handle manually, especially for larger Web APIs. Publicly accessible Web APIs contain up to 200+ operations (routes), with a mid-range of APIs having 7–50 operations [12]. The Web APIs looked at in this paper are representative of that spectrum: between 4 (Flickr) to 223 routes (GitHub; see Section 5.1).

Also, Web APIs and their implementations are subjected to change running in parallel with changes in application code. For example, a Web API may be further developed by adding, by removing, or by altering services or their details (e.g., parameters). In addition, the details of mapping between the application code (i.e., getters and setters in Listing 1) and a resource-orientated HTTP connector may change over time (e.g., HTTP methods used, parameter representation). In these cases, 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. By *affected*, we mean that some (typically, manual) rewriting of boilerplate code (route definitions) had entered a given commit (see Section 5.2).

```
/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

On top of that, Web APIs have multiple technical and non-technical stakeholder roles during development of a Web application. Examples include developers responsible for different server-side building blocks (incl. or excl. the Web API itself), developers for client-side building blocks, developers writing tests against the Web API, documentation writers documenting the Web API, operators overseeing deployed Web applications and their API endpoints; as well as management roles to the extent the Web API must be aligned with important details of a business model (e.g., usage policies, service levels, pricing). Development involves communication and close, document-based interactions between these stakeholder roles. When the Web API is only defined in terms of its implementation in code, this raises barriers to communication and to interaction, esp. for non-technical roles being limited to code documents targeting a single Web-application platform (i.e., a specific programming language, framework, and application server). But also, technical roles may suffer from a code-only definition of a Web API, due to the API and application code being interleaved; or API code being scattered over multiple development documents (configuration files, code). This is also because of the issues of boilerplate coding and parallel code change (see above).

To tackle the above issues, the pattern—well established in distributed systems design and generative programming—of providing and maintaining interface descriptions [25] has been applied to Web APIs. Using an interface-description language (IDL) such as RAML, Swagger, API Blueprint, and WADL (among others), an interface description is a separate, well-defined, and processable development document which defines the details of a Web API only. IDL-based tooling includes code generators that produce the implementation executable on a given Web-application platform (e.g., Flask/Python code shown in Listing 2). At the same time, other documents can be derived from such an interface description in support of other stakeholder roles (test code, mark-up as API documentation). By reaching out to Web-application developers systematically in terms of expert interviews, we found out that generating test cases, route definitions, API usage snippets, and security wrappers for API endpoints are ranked at the top (see Section 3). When supported and used, this makes an interface description the single source of truth for all involved stakeholder roles.

On the downside, an interface-description document for a Web API introduces the complexity of managing co-changes [11] between documents. Changes to the interface descriptions must become tracked by the code documents (e.g., by triggering re-generation of the latter) and vice versa. At the same time, modifications to the application code should not interfere with API-related changes in unintended ways. To this end, 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).

The proof-of-concept implementation plus test suite, as well as a reproduction package incl. the collected interface descriptions is available from a supplemental Web site.[1]

2 WEB APIS

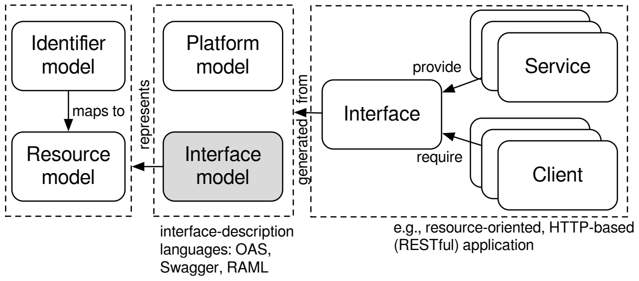

Figure 3: Main tenets of a resource-oriented distributed system: resource model, identifier model, interface model, and Web-application platform model.

Interface Descriptions. In a distributed system, an INTERFACE DESCRIPTION [25] 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. With this, an interface description aligns clients and services, implemented using different Web-application platforms, on critical invocation details. An interface description also establishes a separation of concerns: defining an interface and implementing the interface become separate actions and responsibilities. Interface descriptions reflect the type of interfaces contracted between

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

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.

A description of a resource-oriented interface is a representation of a resource model using an (typically, textual) interface-definition language (IDL). A resource model [16] defines an interface as a collection of resources, each with one or several representations, and the supported virtual instructions on the defined resources (HTTP methods). An associated identifier model defines identifiers (as URI paths and path elements) that point to resources and that contribute to establishing relationships between resources of an interface (see also Section 4).

Interface-definition languages for resource-oriented interfaces have gained attention in recent years, including Swagger [24] and the RESTful Application Modeling Language (RAML) [7]. Their uptake led to the OpenAPI Specification (OAS; [9]) as an effort towards standardized interface descriptions. More recently, development tooling supporting automated transformations from one description type (RAML) into another (OAS) or (client/ service) application development sourced by different description types (RAML, OAS) has become available. In the remainder, the focus is on interface descriptions defined using RAML 1.0 [7].

It is noteworthy that interface descriptions in these IDL are not only representations of a resource model and one identifier model, they establish views on them, often both in terms of an aggregation and as a projection. As an aggregation, an interface description adds structured and unstructured (meta-) data not present in the resource and identifier models, for example: documentation strings as in Listing 2, usage examples, invocation data for tests, and revision identifiers (see Section 3). As a projection, interface descriptions define resource relationships only indirectly and by convention through a specific interpretation of identifiers, e.g. URI path elements.

Self-description vs. explicit interface description. Developing a Web application on top of one or several Web APIs begins with a developer asking questions about the available interfaces. A *self-describing* API allows for answering questions by investigating responses to calling API endpoints (e.g., HTTP header and payload). For a self-describing API there is no explicit interface description. It rather offers a unique URI which identifies a root resource as a starting point for discovering all its resources as well as exploratory operations on these resources. Following the HATEOAS architectural style [19], the returned hypertext should contain the API details (e.g., available navigation options). Avoiding explicit interface descriptions avoids pitfalls pertaining to inconsistencies between interface description and API, managing coupled evolution between description and interface, and processing interface-description documents [17].

On the downside, in practice, there are problems pertaining to self-description. Developers must agree on one way to represent the self-describing API elements. Otherwise, exploring APIs based on different Web application frameworks will differ from implementation to implementation [17]. In HATEOAS, one has multiple options to represent navigation paths in hypertext. Furthermore, self-describing APIs fall short in providing client developers exogenous details (documentation strings, usage examples, invocation data for tests). In terms of development style, an API must be fully implemented to realise the property of self-description [17]. An explicit interface description can be useful independent from an interface implementation [20], e.g., for generating server- or client-side code skeletons, a reference manual for a Web API etc. These uses of an explicit interface description are deemed important by practitioners (see Section 3).

3 ARTIFACT COUPLING WITH INDEPENDENT DELTAS

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 [6]. This is confirmed by a recent study of ours on M2T transformations for UML-based domain-specific modeling languages [□] and a survey among experts on domain-specific modeling [□]. Code is followed by documentation, DB schema definitions, test cases, and build descriptors (in this order); and mixes thereof [6].

For Web APIs and their interface descriptions (RAML, Swagger, OAS), comparable evidence is missing. Available data does not discriminate between technical domains. For this reason, we conducted semi-structured interviews with 14 practitioners in Web application development. The practitioners were selected based on a convenience sampling. To provide a scaffold for the interviews, an interview guideline was prepared along with five "user stories" on an imagined Web application (a content management application for insurance contracts). Each user story (e.g., one entitled "Creation of frontend code for an existing REST route") documents a narrative referring to between one or three different types of generated artifacts (e.g., application code, documentation, or test code).

As for the procedure, there were three phases: (a) self-assessment, (b) interviewing, and (c) aggregation. During self-assessment, an initial selection of participants was asked to complete a self-assessment questionnaire featuring items recommended for describing the experience level of participants. Based on the self-assessments, it was established whether there is a balance between junior, experienced, and senior developers. During interviewing, each participant was asked, in turn, to study each "user story" and to rank the elements of the user story using a EUR-1000 test [1] along with a justification. The individual rankings were then aggregated as a cumulative vote to produce a final ranking of artifact types.

The main result was the ranking of artifact types according to the perceived importance: (1) test code for HTTP endpoints (routes); (2) route implementation (on top of a Web app framework); (3) client-side code snippets for HTTP endpoints; (4) security-concern implementation (on top of a Web app framework); (5) validation routines for invocation data (input, output); (6) documentation strings for routes; (7) build, packaging, and deployment descriptors; (8) version management for route implementations.

On the one hand, these results are confirmatory because they support the overall importance of test code and application code (route implementation), as found for general generative programming and for remoting middleware in general (client- and server-side stubs incl. validation, versioning). On the other hand, the results also highlight the importance of artifact types specific to Web application development using Web APIs (security concerns such as contracted authorisation and authentication schemes).

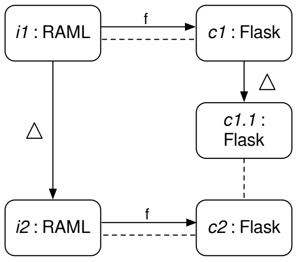

Figure 4: Rectangles represent development artifacts conforming to a software language (RAML, Flask/Python), and arrow-headed solid lines are changes. f denotes a mapping function (model-to-text transformation), δ a manual modification to an artifact, performed by a developer. Dashed lines denote a consistency relationship. Notation inspired by [11].

Artifact coupling. The use of an explicit interface description plus generator creates a collection of *coupled* development artifacts [11].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`. Consider a change which adds a new resource path (e.g., `/orders` as a collection resource) to the RAML document; or modifies an existing one (by changing the representation of a parameter); or simply removes an existing one. This raises the basic challenge of supplying transformations that establish a mapping between the revised interface description and a corresponding Flask/Python implementation, to maintain the consistency relationship within the collection, without manual intervention. This basic challenge is typically tackled by providing a code generator, and so does RAMLFlask (see Section 4). The code generator produces an API implementation from the interface description based on predefined mapping rules.

Realising the mapping between interface description and API implementation by a generator gives rise to a second challenge. This is where this paper's spot is on. The originally generated API implementation (`c1` in Figure 4) is commonly changed by a developer manually, yielding `c1.1`. This is, for one, to fully implement the API, e.g., by weaving framework code to handle compliant HTTP requests and to produce compliant HTTP responses. In addition, this is to realise the contracted application behaviour, e.g., by implementing the behaviour of a getter or setter function (see Listing 1). This code delta is created independently from any delta introduced by changes to the interface description. The key challenge is to provide for a generated API implementation `c2` that maintains consistency both to the corresponding interface description (`i2`) and the manually modified, original API implementation (`c1.1`). These two consistency relationships must be maintained even across repeated rounds of generation (*re-generation*).

There is an array of known techniques from approaches to generative programming and to model-driven software development available to tackle this second challenge [10]. Broadly, they can be grouped into language-based (e.g., design patterns like GENERATION GAP, composition filters, or mixins) or tool-based techniques (e.g., protected areas). Given a technical domain, available techniques (or, a mix thereof) must be surveyed and assessed for their fit in a systematic and critical-analytical manner (see Section 4.1).

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. `Flask` was selected for two reasons: First, it is representative for the family of microservices frameworks, so that design and implementation decisions carry over to other frameworks. Second, integration of interface-description languages with Flask has been explored before. The resulting framework extension plus documentation, examples, and tests are available from a supplemental Web site.[2]

`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 (e.g., and Flask framework elements). 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. This corresponds to the mapping f and the consistency relationship depicted in Figure 4. The mappings are encoded by two Jinja templates: one for request-handler classes, one for delegation classes.

Table 1 summarises the correspondences between RAML interface descriptions (e.g., a resource) and a RAMLFlask entity (e.g., a request-handler class). The processed RAML description is used to generate a particular Python code structure (see Figure 5).

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

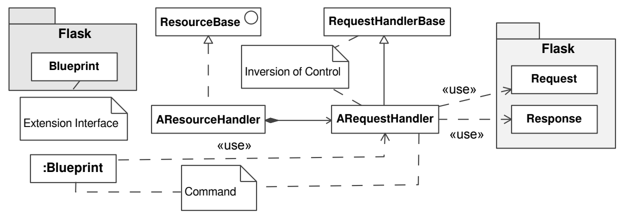

Figure 5: Structural overview of the design artifacts generated from a single RAML interface description by `RAMLFlask`. The UML comments depict the three important architectural design patterns realised by the `RAMLFlask` generator: EXTENSION INTERFACE, CONFIGURATION GROUP, COMMAND, and INVERSION OF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

Blueprint. A RAML description as a whole maps to an instance of a Flask `Blueprint` class. In Flask, a `Blueprint` acts as a CONFIGURATION GROUP [25]. A Flask app or server can have multiple blueprints active at a time, with a blueprint setting shared configuration properties for the constituents of a blueprint: route definitions. A single blueprint defines a number of route definitions that can be reused by different apps or servers. This way, a blueprint serves for implementing an EXTENSION INTERFACE[21]. A single Flask app or server providing for multiple partial Web APIs can so load (or unload) implementing endpoints, as part of a composite Web API. The exemplary RAML document in Listing 2 is turned into a `Blueprint` instance with two route definitions, with bindings to two Python functions; one implementing the getter endpoint (`GET` branch in Listing 1), the other the setter endpoint (`POST` branch).

Handlers. Resources (e.g., `/status`) and methods on resources (`get`, `post`) are turned into corresponding resource and request handlers, respectively (see Figure 5). Request handlers implement a variant of the COMMAND pattern. Each generated route definition as part of the blueprint (see above) indirects a received call to specific handler instance for a given HTTP method. The instance is created from a generated request-handler class, with a single invocation point (`handle_request`). This represents a single endpoint call as a Python object, open for refinement and for interception. It also allows for introducing alternative, runtime-configurable lifetime strategies for requests, if needed. Request handlers operate on two fundamental Flask representations—instances of `Request` and instances of `Response`—to run parameter validation and to prepare the response according to the negotiated content type. Request handlers are also the anchor for managing parameter validation (using built-in, but extensible) validation routines and authentication handlers (delegation classes). For the proof-of-concept implementation, the implementation of delegatees for handling authentication (e.g., OAuth 2.0, basic auth) are mocks. A developer can provide drop-in replacements.

All generated request handlers implement a uniform interface defined by a `RequestHandlerBase`. Also, the request handlers adhere to a number of predefined extension points on top of the single-object representation of per-endpoint calls (COMMAND, see above). Most prominently, the COMMAND method `handle_request` is provided as a TEMPLATE METHOD [5] owned by the common ancestor of all request handlers (`RequestHandlerBase`). This method lays out the overall sequence of request-handling actions and calls into deferred methods whose implementations are provided by the generated request handlers (e.g., parameter validation based on a deferred validation method). This follows the INVERSION OF CONTROL principle.

This structure of the generated code has been designed with best practises from the middleware community [25] and from implementing code generators [10]. In addition, the chosen structure opens up the opportunity to implement and to compare different artifact-coupling techniques.

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 important decision criteria (C1-C3; [8]) and against the background of a given generator architecture (RAMLFlask; see Figure 5).

Assessment criteria are [8]:

- Separation of generated and curated code (C1): Does a technique enforce a clear separation of generated and curated code into distinct source units (e.g., Python scripts) and/ or composition units (e.g., Python classes) forming the artifacts?
- Extensibility (C2):
  - Support for overriding generated parts (C2.1): Can curated code override (with or without reuse) generated parts of an artifact? For example, can the details of a generated route definition (e.g., HTTP method, representation type) be overridden by curated code?
  - Support for adding to the generated parts (C2.2): Can curated code add features to the generated parts? For example, can an extra route definition not announced by the interface description be added (for testing purposes)?
- Portability (C3): Can the technique be ported to other toolchains (Web application frameworks other than Flask) or programming languages (other than Python), or is it specific to a single tool or language environment?

A review must cover tool- and language-based techniques [10]; and account for mixed approaches. A tool-based implementation technique for mitigating unwanted artifact coupling operates on development artifacts (interface-description documents, source-code documents), that is a technique realised outside the software languages used to write the development artifacts. We considered the common techniques of `PartMerger` and of protected areas (see Table 2). A language-based technique, on the contrary, uses or devises first-class mechanisms of the software languages (RAML, Python) to maintain consistency in the presence of independent deltas in a collection of coupled artifacts. Classic language constructs such as class-based inheritance can be used in a disciplined manner (GENERATION GAP [4]), or advanced ones (e.g., composition filters and message indirection using aspects; see Table 2).

One cannot draw a sharp line between these two types of techniques, for example, a template processor is implemented as a preprocessor to a programming language while templates might be implemented by language constructs (e.g., classes). However, it is a fruitful distinction. First, it helped guide the critical-analytical review. For example, pure tool-based techniques are agnostic about the language used (see C3 portability above). Second, one can consider techniques falling into both families as a mixed technique, borrowing benefits and drawbacks from both.

	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 [8] (C1-C3). Legend: + (supported); ~ (partially supported); – (not support); T (tool-based); L (language-based)

As stated in Section 3, RAMLFlask was designed to strive for supporting critical development tasks regarding Web APIs and their interface descriptions. The fundamental tasks are generative support for route implementation (on top of Flask; ranked 2nd), for security-concern implementation (ranked 4th), and for validation routines for invocation data (input, output; ranked 5th). These tasks must be accomplished given the structural foundations of Flask (blueprints, handler functions, request and response objects; see Figure 5) and certain Python constructs, in particular the use of *function decorators* for defining views, routes and endpoints (but not methods). Therefore, certain techniques can*not* enter the shortlist. A syntax-level include-mechanism alone (e.g., in-place sourcing of a generated Python script) that treats generated route definitions as opaque, without programmatic access to them, does not allow for implementing generic validation checks on top, for example. Aspect-oriented techniques, while realisable using Python's metaprogramming facilities, will always be specific to this particular execution environment (C3). Partial classes, including open class definitions as in Python, come with different, language-specific semantics (e.g., w/ or w/wo overriding), again constraining portability (C3).

A comprehensive account of the reviewed techniques and the complete comparison in light of the project-specific requirements (RAML, Flask) are reported in [□]. In summary, we selected a mix of two reviewed techniques as the best fitting solution: GENERATION GAP and DELEGATION. More precisely, GENERATION GAP is used as part of representing and handling routes. A basic variant of GENERATION GAP separates generated code into a superclass and the curated one into a subclass. With this, the superclass can be regenerated, with the subclass picking up changes to the generated code. For the purpose of route definitions for the scope of RAMLFlask with basic refinement support (C2.1), GENERATION GAP turned out sufficient. The scheme could still be extended by a developer adopting RAMLFlask to include support for curated interface extensions. That is, support for manually added and preserved routes and route handlers (C2.2) using (i.e., EXT. GENERATION GAP).

Repeatedly required functionality (e.g., authentication procedures negotiated by a RAML description) are factored out as reusable components that can be shared between Web APIs and their implementing Web applications using DELEGATION. In DELEGATION, a delegator component (e.g., an authentication call in a generated routine) redirects authentication requests to a delegatee, based on the negotiated authentication scheme (or, defaults) and a library of concrete authentication handlers.

4.2 Design & implementation

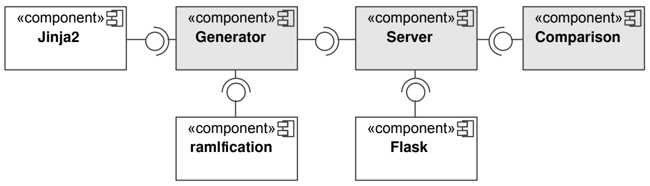

Figure 6: RAMLFlask components and their dependency structure

Structure. The main components of RAMLFlask and their relationships are depicted in Figure 6. RAMLFlask is designed as an extension to the Flask [3] Web application framework. 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[4], a third-party RAML parser. To help a developer manage artifact coupling, once code from an interface description has been generated for the first time, RAMLFlask provides services by a `Comparison` component. This component allows for comparing a potentially modified RAML description and a given revision of the generated code structure to point out inconsistencies to the developer. The `Server` component manages the lifecycle of a RAMLFlask application and is a convenience wrapper around the Flask deployment infrastructure (e.g., to automatically register the generated blueprint with the Flask app before execution). In addition, the `Server` component allows for a configuration descriptor to define a custom configuration for RAMLFlask (e.g., to customize layout and location of generated-source directories).

Behaviour: Generation. Figure 5 summarises the structure of the generated sources for a given RAML document are explained. This structure is created by RAMLFlask 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. Most of these directories are also created as Python modules. The main step consists of creating the resource and request-handler classes. This is achieved by iterating the RAML document for the defined resources, to collect the details to generate validation routines (incl. checks on return types). The extracted data are used to populate a

[3]http://flask.pocoo.org
[4]https://github.com/spotify/ramlfications

Jinja2 template for each and every handler, yielding a single Python script hosting all handler classes.

Behaviour: Comparison. The `Comparison` component allows for running different consistency checks on the family of coupled development artifacts of a RAMLFlask application (i.e., RAML document, server stub, blueprint, resource and request handlers, delegates). First, a developer can request a validation run to compare a newer revision of a generated structure with a previous one. Second, a developer can at any time verify whether the Web application under development, once having departed from the generated skeleton, still conforms to the interface description.

`Comparison` can run a structural diff analysis between two given generated source structures as produced by the RAMLFlask generator from RAML documents. This diff analysis will compare the parameter and return-type details of the two revisions. The result of this diff analysis is a notification of structural changes on parameter validations and return types. Developers can use this information as a checklist to inspect their application implementation that any handwritten code has been updated to accommodate the changes.

`Comparison` can also auto-generate *endpoint tests* and monitor their execution against the Web application under development. An endpoint tests mocks a request to an endpoint generated by RAMLFlask. The mock request is populated by exemplary invocation data provided by the RAML document. An endpoint test can indicate general availability of an endpoint and exercises parameter validation. For this, `Comparison` imports the routes generated by RAMLFlask for a given RAML document to extract the validation details. All resources are then looped through to create requests. First, an empty request is created and then filled with values from the RAML document. Once the request is built, an instance of the corresponding request handler is created. The validation method on this instance is then called to check if the created mock request passes validation.

Developing using RAMLFlask. To kick off application development using RAMLFlask, a developer first triggers generation of the basic application structure (see the code listing below). An instance of `Generator` takes the RAML interface description as input. An instance of `Server` is defined by passing the generator and a `Comparison` instance. The developer can now run the basic code generation and route binding steps using the all-in-one operation `exec_all()`. Behind the scenes, this will not only create a code skeleton, it will optionally also deploy the generated Flask app.

```
from RAMLFlask.Server import Server
from RAMLFlask.Generator import Generator
from RAMLFlask.Comparison import Comparison

gen = Generator('app.raml')
comp = Comparison()
Server(gen, comp).exec_all()
```

To actually implement the negotiated and hosted Web API, the developer can now proceed with the actual development. Most importantly, the developer must implement the mandatory `request_handler` extension point. This can be achieved by defining a subclass of the generated request-handler class (for a given resource plus HTTP method) and by overriding the `request_handler` method. This realises a concrete variant of the GENERATION GAP in RAMLFlask (see Section 4.1). Beyond this, a developer may decide to provide for custom delegation for security schemes and custom parameter-validation routines.

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. The supplemental Web site provides a replication package for this computational experiment.[5]

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. These interface descriptions were officially maintained by the corresponding projects at this time, managed via their code repositories (independent from the authors, see Table 3). For each Web API, we recorded the number of defined routes and the source lines of code (SLOC) of each RAML (YAML) document, as the two proxies for API size serving for the independent variable.

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 for time and space measurements.

Each of the 10 RAML documents was subjected to the different generative tasks offered by RAMLFlask (generation of server stub, generation of route definitions, etc.). Each task was repeated 100 times (incl. an upfront warmup of 100 runs), to obtain 100 data points per Web API, per task, and per dependent variable (execution time, RAM usage).

Figure 7 summarises the main findings on time performance. First, the RAML documents used were selected below half a second (0.5s) of total execution time (worst case) processed for all but one API: GitHub, the largest API in our corpus incl. 223 resources, ran for 2.5s (worst case, as denoted by the triangle in Figure 7) to generate a complete and operative implementation using RAMLFlask. 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 worst-case execution times for this task in Figure 7, as a reference. The remainder of the task (incl. route generation, binding, and comparison tasks) only add minimally to

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

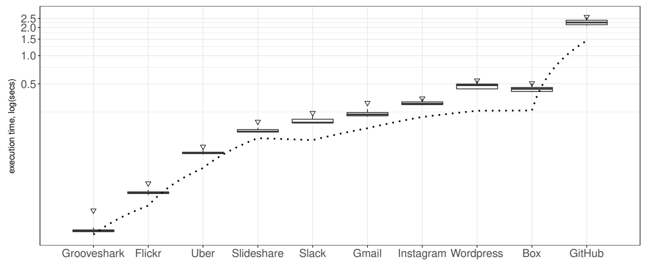

Figure 7: Comparative boxplots for the total execution times in log(seconds) for 10 different Web APIs. The dotted line represents the worst-case (maximum) execution for the basic server-generation task (incl. RAML parsing), the triangles denote the worst-case (maximum) total execution time per Web API; 100 runs

the total execution times. Third, the measured execution times per Web API show only small variance over all runs (as indicated by the "squeezed" IQR rectangles in Figure 7.

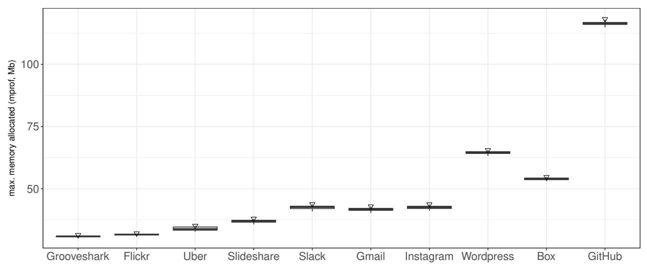

Figure 8: Comparative boxplots for the max. memory allocated by the OS when processing the 10 different Web APIs (ordered by increasing #routes from left to right). The triangles denote the largest maximum memory observed per Web API, 100 runs

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 above). This is because this maximum memory allocated represents the worst case from the blackbox perspective of the OS (neglecting a Python implementation's garbage collector). The details are reported in [□]. Figure 8 visualises the key findings. The All projects except for GitHub were observed to have the executing Python process consume less than 50MB (for Grooveshark through Instagram) and less than 75MB (for Wordpress and Box), respectively. GitHub was found to require more maximum memory, but less than 120MB.

5.2 Replay simulations

In Section 3, Figure 4 two types of change of a Web API under evolution are highlighted: changes to the interface description (RAML), changes to the generated code artifacts, and their interactions. Related work is poor on actual evidence on changes as observed on Web APIs "in the wild" (see also Section 6). How often are development artifacts (interface description, generated code, or curated code) actually changed? What is the relative importance of either type of change in real-world projects? What are characteristic changes to interface descriptions or generated code or both?

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[6], and Sync Engine[7]. HTTPbin is a service for generating HTTP requests and HTTP response for client and server developers to test their HTTP-based applications. Sync Engine offers a Web API enabling developers building apps around E-Mail. The two projects were selected based on the list of publicly known Flask projects, the accessibility of the code repositories, and their non-triviality (i.e., depth of recorded change history and SLOC size).

The objective of the subsequent repository mining was a twofold: On the one hand, a corpus of development artifacts should be 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 should be reported. Note that these two projects did not provide a RAML interface description. For this mining activity, changes to Flask route decorators as well as changes to Flask request and response objects are considered potential changes to an explicit interface description, changes to the decorated functions are changes to curated code. The procedure was as follows (details can be found in [□]):

(1) The Python scripts containing the Flask-based route implementations for each project were identified: HTTPbin (1 file), Sync Engine (4). In total, at the time, HTTPbin had defined 50 and Sync Engine 88 unique routes in these files over the reviewed period.
(2) The GIT commit history of these 5 files was then reviewed, since their initial commit (manually, at GitHub or via the command line). The file commit histories counted 151 (HTTPbin) and 234 commits (Sync Engine), respectively.
(3) Each diff was screened for changes to the resource and route definition (decorator, request, and response) and for changes to the decorated functions. The former were marked as "interface changes" plus affected route (RAML resource), the latter as "code changes". Interface changes were further characterised according to important RAML concepts (see Table 1): additions, removals and renamings (of entire routes), route modifications (parameter type, method, documentation, security scheme, logging)

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 after its first appearance would, therefore, require a re-generation. From the total of 151 commits reviewed, 26 (or, 17.2%) involved simultaneous modifications to interface description

[6]https://github.com/kennethreitz/httpbin
[7]https://github.com/nylas/sync-engine

and curated code. For Sync Engine, the number of co-changes in commits was only 17 out of 234 (7.3%). Whatever their frequency, if driven by a code generator like RAMLFlask such *co-changes* to interface description and previously generated, but by now, manually maintained code may create inconsistencies.[8] Beyond the frequency of co-occurrences of interface and code changes, this analysis did not reveal substantial types of co-occurrences (e.g., certain types of route modifications being paired with parallel code changes). Therefore, we considered all possible types of co-changes equally relevant when designing RAMLFlask.

Next, we systematically selected a set of three routes from HTTPbin and Sync Engine covering for a maximum of interface-change types (route additions, removal, modifications) among all routes. These three routes and their route change history, as established through repository mining before, had a change-type overlap of 40% of all routes (both projects combined) and more than 90% when allowing for partial overlaps. These three routes and their corresponding implementations were then turned in RAMLFlask implementations. This involved authoring a corresponding RAML document and an initial handler implementation using RAMLFlask's GENERATION GAP approach (i.e., a subclass inheriting from a generated superclass). 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 (simulation), and to continuously test RAMLFlask for support of the critical (co-)change types.

6 RELATED WORK

While a number of approaches exist that deal with the systematic development of DSLs in general; see, e.g., [15, 22, 23], the development of DSLs for describing Web APIs, in particular, has rarely been discussed in the scientific literature so far.

EMF-REST [3] is an example of a Java-based REST DSL. However, as EMF-REST is Java-based it also depends on a Java-specific language environment and is thereby inherently platform-dependent. Moreover, EMF-REST was primarily built for developers and is therefore difficult to use for non-technical stakeholders. NeoIDL [2] is another example of a REST DSL. While in principle it does not dependent on a single programming environment and enables code generation for different platforms, it was also built for software developers and is difficult to use for non-technical stakeholders. In addition, NeoIDL does not provide any (native) support for roundtrip engineering activities. In contrast, our approach provides a REST DSL that is suited for developers and non-developers alike, while also providing support for roundtrip engineering for REST-based applications. In particular, we provide integrated RTE checks that are included in the application and guide developers to avoid inconsistencies during the evolution of their API. Furthermore, while our proof-of-concept implementation is based on Python Flask, the generative process is language independent and can easily be adapted for other programming environments.

Some existing approaches address the specification of REST mashups, i.e. Web applications that are composed of two or more REST-based services. For example, Maximilien et al. [13, 14] present DSLs for Web APIs and services mashups. While Pautasso [18] presents a non-DSL approach for composing REST-based services. One commonality of such approaches is that they provide a means for integrating different APIs that have been developed independently of each other. The approach presented in this paper can be used in a similar way for integrating different REST-based applications.

7 CONCLUSION

Web applications and their architectures can benefit from using explicit and processable interface descriptions written using an interface-description language (e.g., OAS, Swagger, RAML). They pave the way for automatising Web application development via code generation. Besides, interface descriptions establish a single source of truth for all stakeholder roles in a Web application project (e.g., developers, testers, documenters). At the same time, this generative use of interface descriptions introduces challenges of managing the generated development artifacts *under simultaneous change.*

The 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 checklist) 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 and from mining of change history from two real-world Web service projects. In addition, the RAMLFlask research prototype was exercised on real-world API descriptions (incl. GitHub, Wordpress, and Instagram) for its time and space performance.

Limitations. To begin with, RAMLFlask is limited to RAML as interface-description language and its concepts. Also, some RAML concepts are not covered by the research prototype (e.g., resource types and traits). However, thanks to the clear correspondences between interface-description languages, the RAMLFlask contributions carry over (Swagger, OAS). In addition, certain code elements generated RAML description elements act only as a non-operational stub. For example, the prototype does not yet have reusable delegate implementations, e.g., for authentication schemes such as OAuth 2.0.

As future work, we will explore extending RAMLFlask to support other interface-description languages (OAS) and to test important quality attributes of RAMLFlask (ease-of use) using appropriate empirical designs involving practitioners.

REFERENCES

[1] Patrik Berander and Anneliese Andrews. 2005. *Requirements Prioritization.* Springer, 69–94. https://doi.org/10.1007/3-540-28244-0_4

[2] Rodrigo Bonifácio, Thiago M Castro, Ricardo Fernandes, Alisson Palmeira, and Uirá Kulesza. 2015. NeoIDL: A Domain-Specific Language for Specifying REST Services. *International Journal of Software Engineering and Knowledge Engineering* 25, 09n10 (November & December 2015), 613–618.

[3] Hamza Ed-Douibi, Javier Luis Cánovas Izquierdo, Abel Gómez, Massimo Tisi, and Jordi Cabot. 2016. EMF-REST: Generation of restful APIs from models. In *Proceedings of the Annual ACM Symposium on Applied Computing.* ACM, 1446–1453.

[4] Martin Fowler. 2010. *Domain Specific Languages* (1st ed.). Addison-Wesley.

[8]Note that the two repositories did not allow for verifying the existence of actual inconsistency (e.g., via referenced bug reports).

[5] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. 1995. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[6] 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

[7] github.com/raml org. [n. d.]. RAML Version 1.0: RESTful API Modeling Language. https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md. retrieved on 29.09.2016.

[8] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. 2015. Integration of Handwritten and Generated Object-Oriented Code. In *Model-Driven Engineering and Software Development*. CCIS, Vol. 580. Springer, 112–132.

[9] OpenAPI Initiative. 2019. OpenAPI Specification. Available from GitHub at https://github.com/OAI/OpenAPI-Specification, last accessed: 2.9.2019. https://github.com/OAI/OpenAPI-Specification

[10] Sven Jörges. 2013. *Construction and Evolution of Code Generators: A Model-Driven and Service-Oriented Approach*. LNCS, Vol. 7747. Springer.

[11] 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

[12] Maria Maleshkova, Carlos Pedrinaci, and John Domingue. 2010. Investigating Web APIs on the World Wide Web. In *Proc. 8th IEEE European Conference on Web Services (ECOWS 2010)*. IEEE, 107–114. https://doi.org/10.1109/ECOWS.2010.9

[13] E Michael Maximilien, Ajith Ranabahu, and Stefan Tai. 2007. Swashup: Situational web applications mashups. In *Proceedings of the ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 797–798.

[14] E Michael Maximilien, Hernan Wilkinson, Nirmit Desai, and Stefan Tai. 2007. A domain-specific language for web APIs and services mashups. In *Proceedings of the International Conference on Service-Oriented Computing*. Springer, 13–26.

[15] M. Mernik, J. Heering, and A.M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (December 2005), 316–344.

[16] A. Neumann, N. Laranjeiro, and J. Bernardino. 2018. An Analysis of Public REST Web Service APIs. *IEEE Transactions on Services Computing* (2018). https://doi.org/10.1109/TSC.2018.2847344

[17] Luca Panziera and Flavio De Paoli. 2013. A framework for self-descriptive restful services. In *Proceedings of the International Conference on World Wide Web*. ACM, 1407–1414.

[18] Cesare Pautasso. 2009. Composing restful services with jopera. In *Proceedings of the International Conference on Software Composition*. Springer, 142–159.

[19] restcookbook.com. [n. d.]. What is HATEOAS and why is it important for my REST API? http://restcookbook.com/Basics/hateoas/. retrieved on 14.08.2017.

[20] 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.

[21] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. 2000. *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd.Wiley, Chichester, England, Chapter Extension Interface, 141–174.

[22] D. Spinellis. 2001. Notable Design Patterns for Domain-specific Languages. *J. Syst. Softw.* 56, 1 (2001), 91–99.

[23] Thomas Stahl and Markus Völter. 2006. *Model-Driven Software Development*. John Wiley & Sons.

[24] swagger. [n. d.]. Swagger. http://swagger.io/. retrieved on 23.10.2017.

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