Zentrum für sichere Informationstechnologie – Austria
Secure Information Technology Center – Austria

A-1030 Wien, Seidlgasse 22 / 9
Tel.: (+43 1) 503 19 63–0
Fax: (+43 1) 503 19 63–66

A-8010 Graz, Inffeldgasse 16a
Tel.: (+43 316) 873-5514
Fax: (+43 316) 873-5520

http://www.a-sit.at
E-Mail: office@a-sit.at

DVR: 1035461 ZVR: 948166612 UID: ATU60778947

SECURITY ASPECTS OF WEB-APIs

Project Report

Version 1.0, 13.3.2017

Bojan Suzic – bojan.suzic@a-sit.at

Abstract:

Web-APIs represent a significant building block of the modern Web. They enable efficient and technology neutral data and process integration between diverse entities and platforms. As an innovation driver, they facilitate the creation of new business models and products. Based on the currently diverse range of models and implementations, as well as missing standardizations, it is not easy to evaluate the current state of Web APIs and their development. Information from several web directories or API search engines can be used as partial parameters to estimate the adoption level and features of publicly exposed Web APIs. From this point, the ProgrammableWeb lists and categorizes more than 15,000 APIs which are registered, published and exposed to external parties by companies located world-wide.

The broad variety of APIs, as well as the management of their lifecycles, motivated the inception of specifications and tools to ease and accelerate their development and integration in programmatic environments. The most known examples of such frameworks are Swagger (OpenAPI), RAML and API BluePrint. These frameworks establish specifications and provide supporting tools with primary goals to support the design, implementation, maintenance, documentation and sharing of APIs. They rely on JSON or YAML-based code that describes Web API structure and is reused by supporting tools to provide a specific functionality in the domains such as API documentation or management.

Focused on practical aspects of API development and integration, these specifications do not put a particular emphasis on non-functional aspects, such as security. This work particularly addresses that aspect by evaluating existing security-related features of API-description frameworks, investigating existing gaps, as well as the application and extension of these frameworks. It further explores possible synergies and orthogonal integrations with other frameworks and tools with the potential to deliver additional benefits relevant to the security of Web APIs and overall security management in distributed environments.

Contents

Figures

Tables

1. Introduction

The data as a driver of economy and innovation is recognized by many industrial or administrative entities. While industrial players intensively rely on data to generate competitive advantage, establish new services or new markets [1], the administrative entities only recently started to integrate data in their widely reaching initiatives. One of the examples in this direction is provided by the Communication of EC [2], which identifies important building blocks and establishes the action plan to support the growth of the data-driven economy. The cloud computing is there acknowledged as one of the key enablers of growth and development in all sectors for citizens, businesses and public administrations, especially considering the enhancement and adoption of data value chain [3]. One additional example that can be partially considered as a supporting action is also a recent EU Directive on Payment Services (PSP2) [4], which among other objectives, aims to support the establishment of new and open interfaces to facilitate the exchange of data and services from the banking sector in a structured way.

In a typical data sharing scenario, the interfaces between systems on Web are established using Web Services or Web APIs, which allow the interaction, data exchange and service consumption across different platforms, systems and organizations using a set of standardized protocols and tools. In the interpretations of PSP2, Web APIs are one of the essential technologies that should be employed to establish open interfaces among the diverse interacting entities in payment service chains [5]. The Web APIs are, however, since the last decade one of the broadly adopted means to open data and interfaces to external parties. The portals such as ProgrammableWeb[1] count already more than 15,000 different APIs classified in more than one hundred of thematic categories. Although some may question the completeness of API directories, as well as the timeliness of their data, the wide presence of APIs in today's web is hard to be dissented.

There are several technologies that support the setting up of Web APIs and delivery of Web Services. Although traditional web services[2] were initially preferred by businesses, often due to their structured interfaces and strict standardization, RESTful interfaces became the dominant approach to support Web APIs during the recent years [6]. This can be easily observed on the example of ProgrammableWeb, which reports an exceptional increase in registrations of RESTful APIs since 2011.

Following the broad adoption of Web APIs and the emergence of related issues and challenges from their increased use, many approaches were incepted to provide structured descriptions of APIs or support their integration in a broader ecosystem. This report focuses on RESTful APIs and hence the solutions aimed to provide structured descriptions of interfaces based on this paradigm.

The primary goal of this report is to identify the currently adopted or emerging specifications for the structured description of API interfaces, evaluate the features of existing approaches in the terms of security, and investigate and further develop their application or use scenarios with the objective to advance the security management within and across the systems and organizations.

After this introduction, the second section of the report presents the conceptual developments and evolution of API approaches and summarizes the actual endeavors aimed to classify interfaces present on the web. Then, the third section introduces the Web API specifications currently applied web-wide, focusing on the assessment of two approaches and their security-related features. The fourth section of this document deals with the issues and challenges characterizing selected frameworks, which are then further positioned in the subsequent section, both in the terms of development, application, security relevance and potential benefits arising from integration with the other frameworks. This report then concludes with a summary.

---

[1] http://www.programmableweb.com

[2] WS-* family of standards

2. Web API Interfaces

The idea that underlies today's APIs has been existing since the practical adoption of computing. There can be recognized four significant transitions that characterize data and system integration for the purpose of data interchange, as shown in Figure 1.

The first phase, specific for the 1960s to 1980s, has been characterized by the inception of basic approaches to interconnect systems, such as ARPANET or establishment of TCP sessions. The further developments, characteristic for the 1980s to 1990s, led to the adoption of techniques such as point-to-point interfaces and electronic data interchange. The significant adoption of Web subsequently led to the development of techniques based on enterprise service bus and service-oriented architectures.

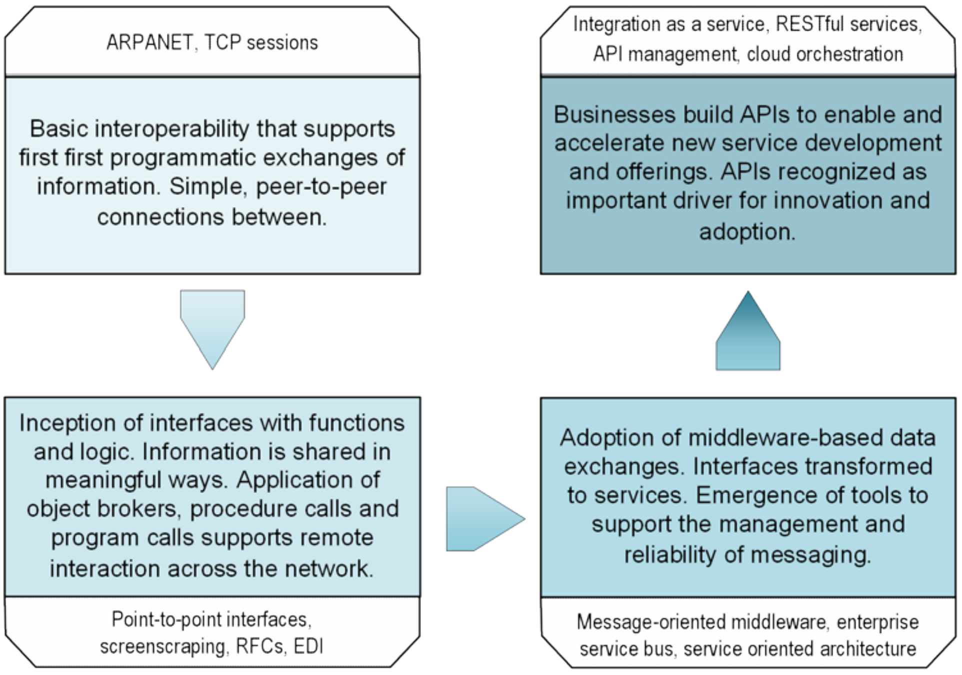
ARPANET, TCP sessions
Basic interoperability that supports first first programmatic exchanges of information. Simple, peer-to-peer connections between.
Inception of interfaces with functions and logic. Information is shared in meaningful ways. Application of object brokers, procedure calls and program calls supports remote interaction across the network.
Point-to-point interfaces, screenscraping, RFCs, EDI
Adoption of middleware-based data exchanges. Interfaces transformed to services. Emergence of tools to support the management and reliability of messaging.
Message-oriented middleware, enterprise service bus, service oriented architecture
Businesses build APIs to enable and accelerate new service development and offerings. APIs recognized as important driver for innovation and adoption.
Integration as a service, RESTful services, API management, cloud orchestration

Figure 1: The evolution of APIs

Since the 2000s we are witnessing strong developments and the emergence of the terms such as "API economy" [1]. This concept primarily emphasizes the innovation capability of networked data exchange and service consumption, backed by technologies and protocols that apply the separation of services, interfaces, and their functionality. This allowed the creation of businesses that, for instance, specialize in providing APIs to their clients, strongly base their infrastructure and business model on third party APIs, or businesses that specialize in managing, integrating, brokering, documenting or testing external APIs.

The growth of public interest in Web APIs can be noticed from search trends on Google, as presented in Figure 2, which describes the evolution of interest for term *"web api"* for the period since the beginning of 2012 and the end of 2016.

The growing number of devices that are connected to the Internet raises the significance of data sharing and integration among them. Beside traditional, user-oriented devices, such as computers, laptops, tablets or smartphones, there is an increasing number of other independently deployed devices and appliances that interact through the internetworked world. The examples are smart-home appliances, such as refrigerators, alarms, lighting and other devices that can control household equipment. In addition to them connected are vast types of Internet-of-Things devices such as electric meters, or sensors that constantly monitor their environment and collect data, up to the agricultural equipment that controls the fields and production facilities.

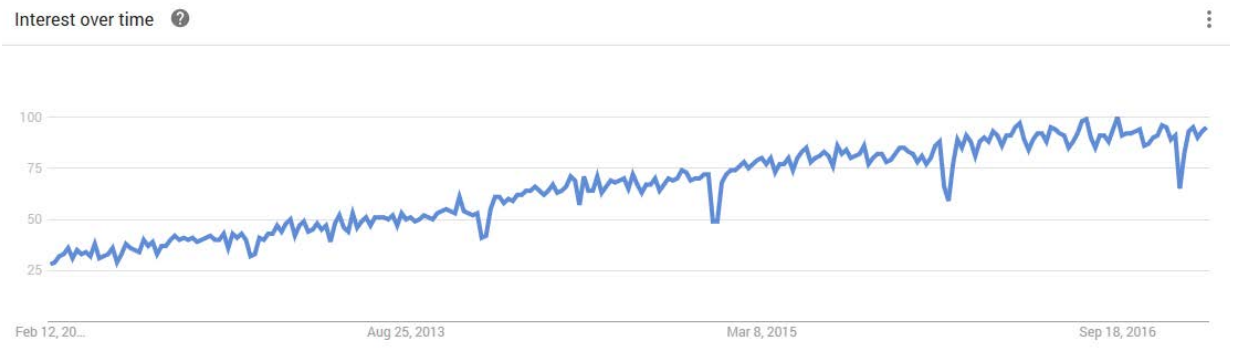

Figure 2: Google Trends: interest between 2012 and 2016 for term "web api"

In the auto industry, the recent developments aim to interconnect cars and other transportation means with the Internet and other subjects. From this point, APIs are important as they provide structured and broadly adopted means to exchange the data between devices and integrate their exposed functionality in the composite processes that cross the device, organizational or jurisdictional boundaries.

The significance of data integration and exchange as a significant innovation and growth driver, as well as a market force, has been acknowledged in the field of public authorities. In its recent communication [2], European Commission recognizes Web APIs as one of the enablers for data exchange in the industry. One of the recommendations hence suggests dedicating additional efforts to support the development of open, standardized and well-documented APIs that can serve as further provision and exchange of data in machine-readable formats with accompanying meta-data. Another particular measure from EU authorities aimed to support the development and adoption of API-based, layered service approaches can be noticed in the Payment services directive [4, 7]. This initiative intends to promote the emergence of new players and markets as well as the development of innovative mobile and internet payments to support the world competitiveness.

Several initiatives to categorize and track the API developments exist to date. One of the most known portals is ProgrammableWeb[3], which lists about 15,000 APIs in its catalogue. The APIhound[4] service implemented automated indexing machine specialized for Web APIs, which gathered over 50,000 specifications to date. Some other services, such as APIs.IO[5] and APIs.guru[6] focus on manual maintenance of API catalogues with additional metadata that enables their automated retrieval and reasoning on a structured way. Due to the involved overhead, these two approaches index a bit more than 1,000 APIs.

Due to the diverse nature of APIs, quick developments, disparate technologies used to design and provide their functionality, and a lack of standardization, it is hard to assess the precise number of the publicly available APIs, or to classify them based on their properties. The survey of Bülthoff and Maleshkova [6] however provides some indicative results on nature of top 45 Web APIs and their underlying features.

[3] http://www.programmableweb.com

[4] http://apihound.com/

[5] http://apis.io/

[6] https://apis.guru

3. Web API Description Specifications

During the last years, several approaches to describe RESTful APIs have emerged. The most known among them, Swagger, is recently adopted as a reference for OpenAPI specification, backed by several important industry players. RAML, less adopted but more expressive and complete, is the second approach that is being supported by companies focused on development of integration platforms and API management solutions. Finally, API Blueprint, is less a specification and more an effort that positively resonated in community and gained a broader acceptance.

Subsequent sections describe OpenAPI and RAML in more detail and evaluate them in the terms of security related capabilities. Aside the general comparison, API Blueprint is excluded from the more detailed analysis as it more focuses on functional features and (still) do not provide a structured mean to express security related properties or requirements.

The following table shows the basic comparison between three main API description technologies applied in the practice.

		OpenAPI	*RAML*	*API Blueprint*
Basic information	Format	JSON, YAML	YAML	Markdown
	Workgroup	Yes	Yes	No
	Institutional support	3Scale, CA technologies, Google, Microsoft, Paypal, IBM, Atlassian, Adobe	Mulesoft, AngularJS, Akamai, Cisco, VmWare, Akana	Apiary
	API design approach	Top-down Bottom-up	Top-down	Top-down
	Current version	2.0 (3.0-devel)	1.0	1A9
Security features	Authentication	HTTP Basic API keys OAuth 2 (implicit, password, client credentials, authorization code) OpenID Connect	HTTP Basic HTTP Digest OAuth 1.0 and 2.0 Pass Through	Not supported (Considered in RFC documents published separately)
	Security filtering	Filtering access to the documentation itself	-	-
Community	Stackoverflow questions	~8200	~600	~1500
	Github contributors	~90	~30	~50
	Github stars	~5700	~700	~5000
	Github forks	~1700	~200	~1500

Table 1: Basic comparison of three main frameworks

3.1. OpenAPI concepts

OpenAPI is the effort organized and backed by Linux Foundation and relevant industry players from several areas. It builds on Swagger, an initiative that has been originally started by Wordnik[7] and later donated to the community in efforts to develop and adopt an open specification for machine-readable interfaces for describing, producing, consuming and visualizing RESTful services. In this sense, OpenAPI 3.0 represents the first significant milestone after the adoption of Swagger as a base for OpenAPI specifications. Considering the fact that it is currently in beta, and expected to be published soon, in this work we rely on this specification and refer to the OpenAPI 2.0 (original Swagger) to point to significant changes.

Compared to its predecessor, OpenAPI 3.0 brings the following categories of changes:

- Structural improvements
- Extending capabilities of request parameters
- Improving protocol and payload handling
- Improving description of documentation and its integration with examples
- Advancing capability of security models
- Extending the scope of RESTful path definitions

In overall, the structure of the description document has been simplified, both with the aim to improve (human) readability and navigation but also to support definition and reusability of its common components. The change that mostly deals with the document structure is illustrated on Figure 3, which highlights the modifications between two versions of the specification.

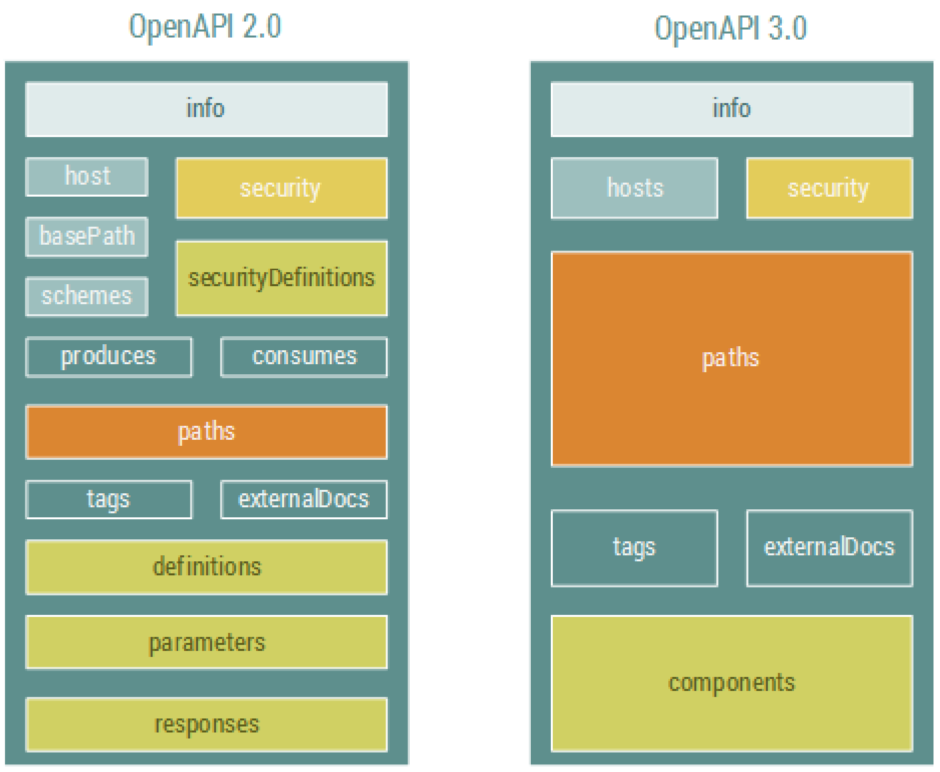

Figure 3: Differences in organization of 2.0 and 3.0 versions of OpenAPI

[7] An non-profit organization that provides language and dictionary resources

One of the most significant changes is the `Components Object`, which contains a range of other objects that establish the definitions reusable in other parts of specifications. These definitions may include the formats of responses, parameters, headers, callbacks or security parameters, which are further applied in other parts of the definitions by the means of references.

The overall structure of `Components` section is shown on the right part of Figure 4.

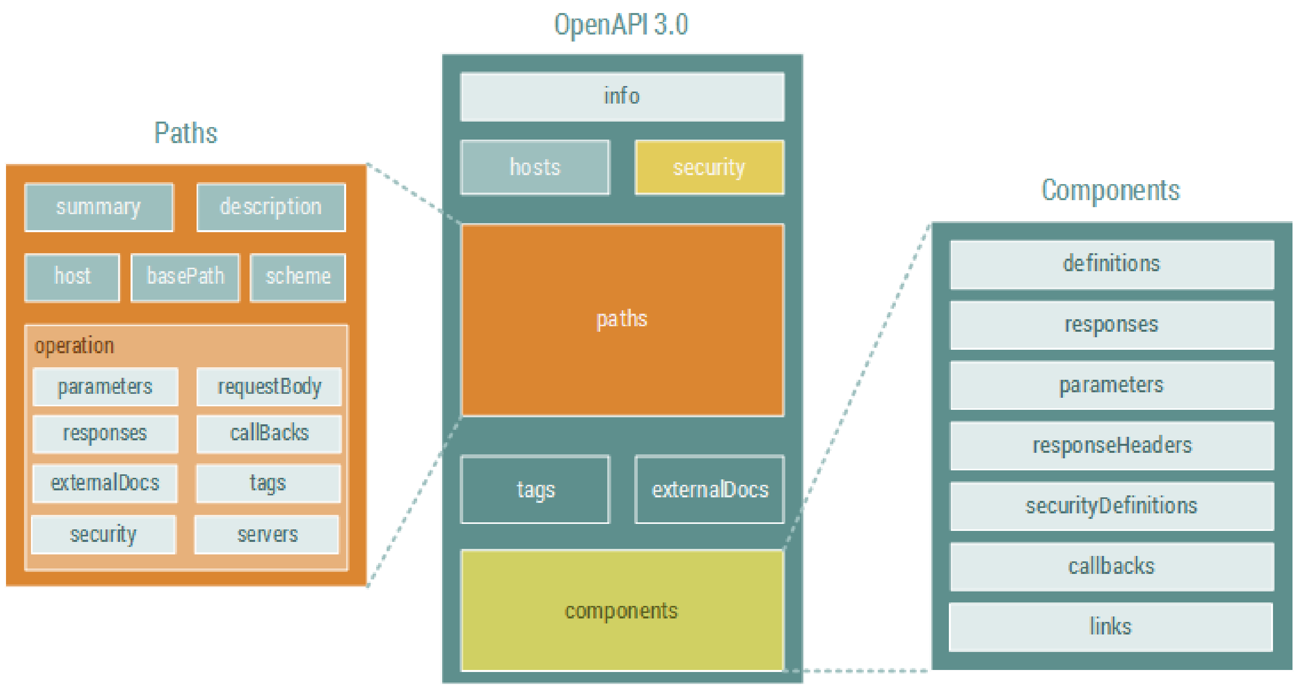

Figure 4: Important entities in OpenAPI 3.0 information model

The left part of Figure 4 shows the structure of the `Paths` section, which contains a set of elements that describe individual URL paths i.e. endpoints of a service. Based on other definitions provided in the root or other elements of document description, the paths are reused in the specification document and by the consumer client. This is done by concatenating them to the server address and base path, following their hierarchical establishment. Each of path elements is denoted as a `PathItem`, aiming to granularly describe supported operations over a particular path that are derived from standard HTTP methods. Hence, a path item delivers a particular operation and provides a structured description of supported capabilities, required parameters, as well as predefined responses and available callbacks that apply to each described operation.

Figure 5 shows the example code in YAML that describes an `Operation`, a primary element that is contained in the `PathItem` object. In this case, the operation is executed as `PUT HTTP` method (defined in the `PathItem` object, not shown here). The call expects a parameter `petId`, which is a required string contained in the path of the URL. Other parameters are provided in the request body, which expects a form that contains parameters related to the `name` and `status` of the entity. After providing the supported HTTP response codes and structures, the operation also defines the security requirements that are expected to be fulfilled for the request to be successfully executed.
In the subsequent section, the role of this capability and its conceptual dimension will be approached in more details

```yaml
tags:
- pet
summary: Updates a pet in the store with form data
description: ''
operationId: updatePetWithForm
parameters:
- name: petId
  in: path
  description: ID of pet that needs to be updated
  required: true
  type: string
requestBody:
  content:
    'application/x-www-form-urlencoded':
      schema:
       properties:
          name:
            description: Updated name of the pet
            type: string
          status:
            description: Updated status of the pet
            type: string
        required:
          - status
responses:
  '200':
    description: Pet updated.
    content:
      'application/json': {}
      'application/xml': {}
  '405':
    description: Invalid input
    content:
      'application/json': {}
      'application/xml': {}
security:
- petstore_auth:
  - write:pets
  - read:pets
```

Figure 5: Example operation element formated in YAML

3.2. Security mechanisms in OpenAPI

The security model envisaged in OpenAPI assumes (1) the abstract definition of security related capabilities of the exposed web interfaces on a level of a particular interface (document), and then (2) subsequent reuse of those definitions across the described API. These definitions can be applied at different levels, starting from broader API segments, such as whole endpoints and their sub-endpoints, to particular operations and calls registered under each of those entities, which may impose additional security requirements than their hierarchically related counterparts.

Such organization enables specification of different security requirements for parts of interface that are e.g. related to reading the data and the ones that are dedicated to insert or update data.

The first element relevant for the security functionality is the `Security Definitions Object`. It is contained in the `Components` section and serves as a container for supported and reusable building blocks that describe security functionalities referenced across the API. The establishment of these elements from the document root is shown in Figure 6. `Security Definitions` hence provide a description of each authentication mechanism that is or may be used in the API.

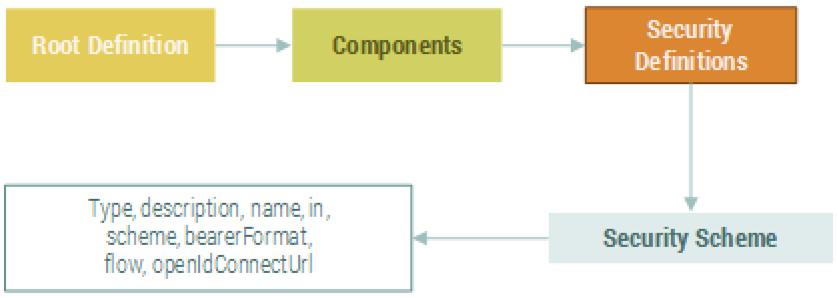
Root Definition
Components
Security Definitions
Security Scheme
Type, description, name, in, scheme, bearerFormat, flow, openIdConnectUrl

Figure 6: Establishing security scheme and definitions in OpenAPI 3.0

Each of those elements represents a particular `Security Scheme`. The first element in a scheme determines the `type` of the scheme[8], which then implies the fields in definition structure that have to be populated and integrated. The instances of this object are reused by other objects in the API, allowing a higher level of granularity and the application of different security (authentication) schemes in the sections of the exposed and described API.

Figure 8 shows the abstract application of this element by other structures present in OpenAPI document. The example description that illustrates how `Security Schemes` are defined is shown in Figure 7. Two supported methods are announced there: (1) traditional *API keys*, which are provided in the request header, and (2) *OAuth 2* with implicit flow, specifying authorization URL and two supported authorization scopes.

```
api_key:
  type: apiKey
  name: api_key
  in: header
petstore_auth:
  type: oauth2
  flow:
    implicit:
      authorizationUrl: http://swagger.io/api/oauth/dialog
      scopes:
        write:pets: modify pets in your account
        read:pets: read your pets
```

Figure 7: Example definition of two supported security mechanisms

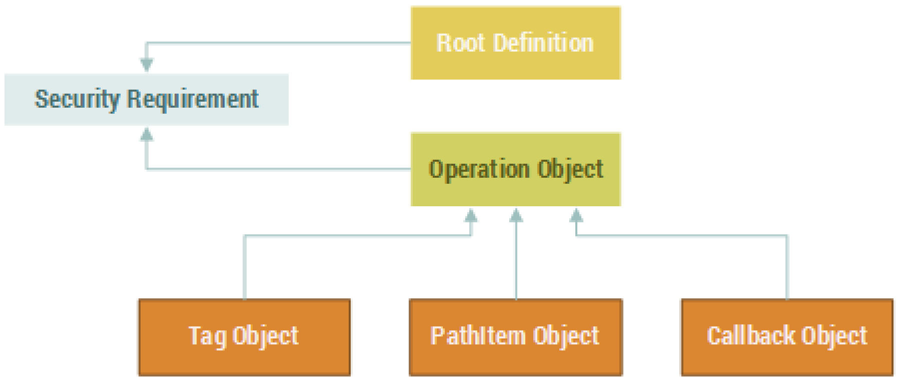

Figure 8: Expression of Security Requirement in OpenAPI 3.0

[8] One of the fields foreseen by the specification

3.3. RAML concepts

RAML (RESTful API Modelling Language) was first proposed in 2013 with the goal to support API design and the management of its lifecycle from scratch. Although it was proposed more like proprietary approach, RAML is today managed by the dedicated workgroup consisting of several important industry players and open-sourced in the form of an Apache license.

In its current version, 1.0 RAML represents a step towards additional maturity and expressiveness compared to previous RAML 0.8. In comparison to other approaches, RAML introduces the concept of *libraries*, which are sets of predefined data and resource types that can be shared and reused across the systems. This capability is complemented by *overlays*, a feature that allows the transparent extensions of API descriptions by including new descriptions and annotations in a transparent way that can be tracked back through different versions. In practical terms, this feature enables the implementation of inheritance and overriding of parameters defined in base documents. It also allows the separation of interfaces from implementations and management of API lifecycle changes considering both behavioral and implementation aspects of the API.

In comparison to OpenAPI, RAML allows a higher degree of modularization and customization. While modularization supports reuse of artifacts in the description document or across the documents, the supported levels of customization allow the expressive and granular definitions of resource or data types and fine-tuning of options at the level of each method or operation. The drawback of this feature is, however, a (currently) limited range of tooling and support tools for the newest version of this specification that makes the creation and integration of RAML documents easier for developers. The overall organization of RAML specification document includes four main blocks that enable the definition of *traits*, as a well as *data*, *resource* and *annotation* types. These are complemented with the mean to define security capabilities of the API and with imported libraries, overlays, and extensions.

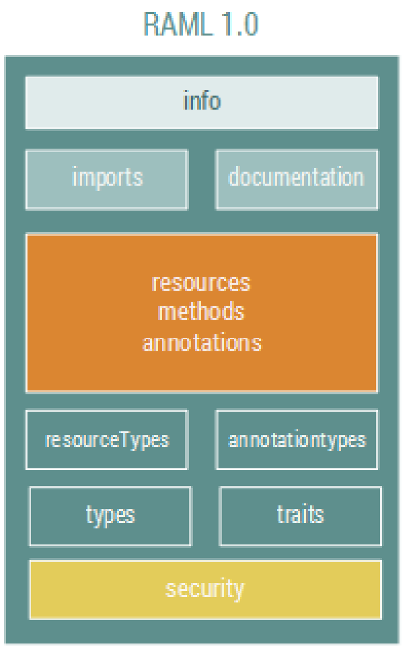

Figure 9: RAML 1.0 general document structure

In Figure 10 shown is the abstract approach employed to describe resources. Due to the dependencies between methods, resources, and their constitutive parts, the definition of these entities is coupled. The structural API descriptions in RAML start with top-level resources, which are then augmented with nested resources through the different levels. Key for each child node is its URI, which is relative to URI of its parent. Hence, starting from base URI, each relative URI in the chain is appended to form a key of each nested resource.

Each instantiation of a resource in RAML is characterized by its *type*, *URI parameters*, *traits*, and *method*. The latter is a primary object that allows further specialization of operations that apply to a resource. These operations correspond to different supported HTTP methods and additionally specify traits, query strings and query parameters specific to the method as well as the formats of headers, body, and responses. This allows specifying detailed information for each parameter expected in header or body, or the parameters that the operation provides back to the caller in a response. In the scope of these definitions, the rich expressivity features of RAML allow a fine-tuned specification and reuse of various default or additionally specified types in the root of the document. The general overview of these fields is provided in Figure 10.

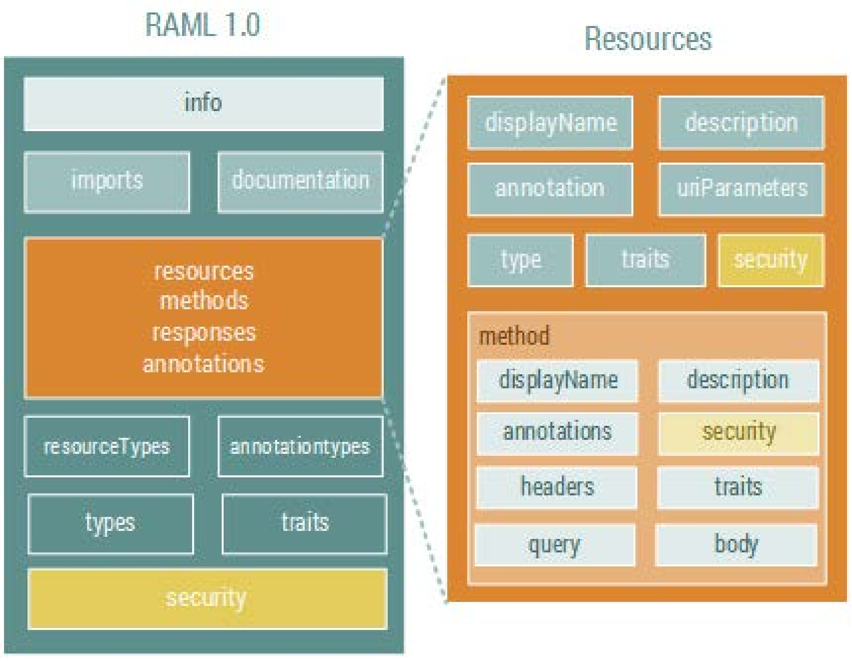

Figure 10: Abstract entities for definition of resources and methods in RAML

Figure 11 illustrates a specification of a resource type in RAML. This example shows a resource type called `companyResource` with optional `post` node that defines required header `X-Chargeback` and a custom parameter YY. The resource `/servers` inherits `companyResource` and defines a `post` method, which based on inheritance has to include the `X-Chargeback` header requirement. Similarly YY has to be defined as well. As `/queues` does not implement optional post method, it does not have to define `X-Chargeback` and YY elements. This example illustrates only a portion of customizable features of RAML, which enables detailed specification of APIs and servers and extensively supports reuse principles. As such, it represents a solid base for API lifecycle development cycles, including designing, testing, documenting and reusing of APIs.

```
title: Example of Optional Properties
resourceTypes:
  companyResource:
    post?:
      description: Info about <<YY>>.
      headers:
        X-Chargeback:
          required: true
/servers:
  type:
    companyResource:
      YY: post method # post defined, force definition of YY parameter
  get:
  post: # will require the X-Chargeback header
/queues:
  type: companyResource
  get:
```

Figure 11: Example definition of a resource type using inheritance

3.4. Security mechanisms in RAML

Figure 12 visually illustrates the approach followed in RAML to expose and declare security capabilities of description model. The root of API definition can contain different optional and required elements, of whom we represent `Resources` and `Security Schemes` as the ones most relevant in current context. `Resources` is an element that corresponds to path items present in typical RESTful URI. They reference `Methods`, which are the operations performed on a resource. Methods can be defined on an abstract level, referring to all possible methods executed on a resource, or defined separately for each of HTTP methods, as provided in RFC 2616 and RFC 5789.

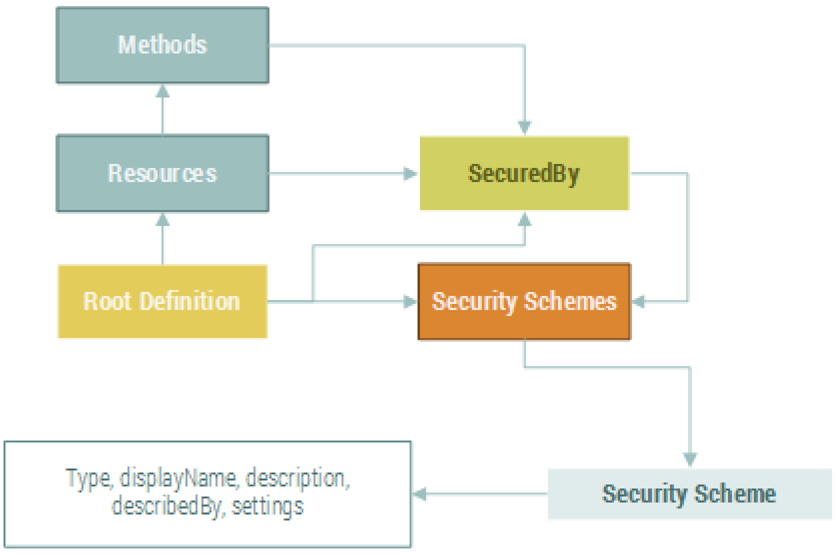

Figure 12: Defining security capabilities in RAML

Both of these elements contain the property `securedBy`, which refers to one of `Security Schemes` defined in the root of the document. `Security Schemes` is hence a set of records that either configure already defined and supported security (authentication) methods in RAML, or extend basic specification and establish a new mechanism. Each `Security Scheme` contained in this set, therefore, exposes basic data for its representative mechanism and configures that mechanism by populating *settings* property.

Following the hierarchy and semantics of `Resources` and `Methods`, each of these elements may point to particular `Security Scheme` or a set of `Security Schemes` by referring it using `SecuredBy` property. This approach allows denoting a default schema that is valid for the whole (sub-)API, while each of hierarchically subordinated elements can override parent schema and introduce its own configuration, as shown through the relations in the figure.

One particular difference in RAML compared to other approaches is the pass-through feature, which enables to specify passing through the headers or query parameters to API backend, with the purpose to delegate authentication/authorization. This allows a specification of customized authentication methods, such as key-based authentication, that allow clients and servers to exchange authentication tokens using specific parameters provided in the requests.

3.5. API BluePrint concepts

API BluePrint is a solution proposed by Apiary, an independent vendor of API management tools. In the beginning of 2017th Apiary has been acquired by Oracle.

In its current version, this specification does not establish any kind of controls that communicate security capabilities of the API or its underlying elements. However, the specification repository[9] has several active RFCs that deal with typical authentication mechanisms such as Basic and OAuth 2 based authentication. These proposals mostly focus on definition of interaction points that allow i.e. the integration of protocol flows with a client API.

Due to the incomplete and non-agreed approach on addressing the security aspects in API BluePrint, the current work will not analyze this solution in detail.

4. Issues in Existing Description Approaches

Based on the presented concepts, in the scope of the current work, the features of both frameworks related to security were analyzed. In this section, we review the issues identified in this analysis.

4.1. Establishing security descriptions in frameworks

Both OpenAPI and RAML address description of security characteristics in a similar way. They define fixed structures that need to be populated, depending on security mechanism that is being applied. However, the abstraction levels used across these two approaches differ in practice.

In OpenAPI, the primary `Security Scheme Object` contains several fields that describe the security mechanisms. Of them, the primary field is `type`, which has to contain one of the predefined values to refer to the mechanism actually defined in the object. Hence, one of values `apiKey`, `http`, `oauth2` or `openIdconnect` is to be used. Other fields are required and take different roles, depending on the specified type of security mechanisms. For instance, if API keys are used, then the fields for `name` and `in` have to be specified. Alternatively, in case of OAuth 2, the `flow` has to be further specified, by using one of the fields shown in Figure 13 to provide further configurations relevant for particular OAuth 2.0 flow.

RAML takes a bit more abstract approach. The field `type` similarly determines the mechanism that is specified. Depending on its value, other fields such as `describedBy` and `settings` are to be appropriately populated. The former field, optionally specified, is present mostly for descriptive purposes, to state which headers, query parameters or responses are used by security mechanism. This way the developers can be informed about the type of the header and its role in the processes. This kind of specification may help to prevent issues in the naming of header or query fields, but it also can be used to model requests and responses and apply them for i.e. firewalling purposes. The latter parameter, `settings`, contains a map with the parameters that further specify an authentication method. The choice and naming of these fields differ between the mechanisms that are employed. The example present on Figure 13 illustrates the case of using OAuth 1.0 mechanism.

In the cases of both frameworks the coupling of the descriptions can be observed as well as hard-wired specification of entities that provides information mostly on a syntactic correct level.

[9] https://github.com/apiaryio/api-blueprint-rfcs/tree/master/rfcs

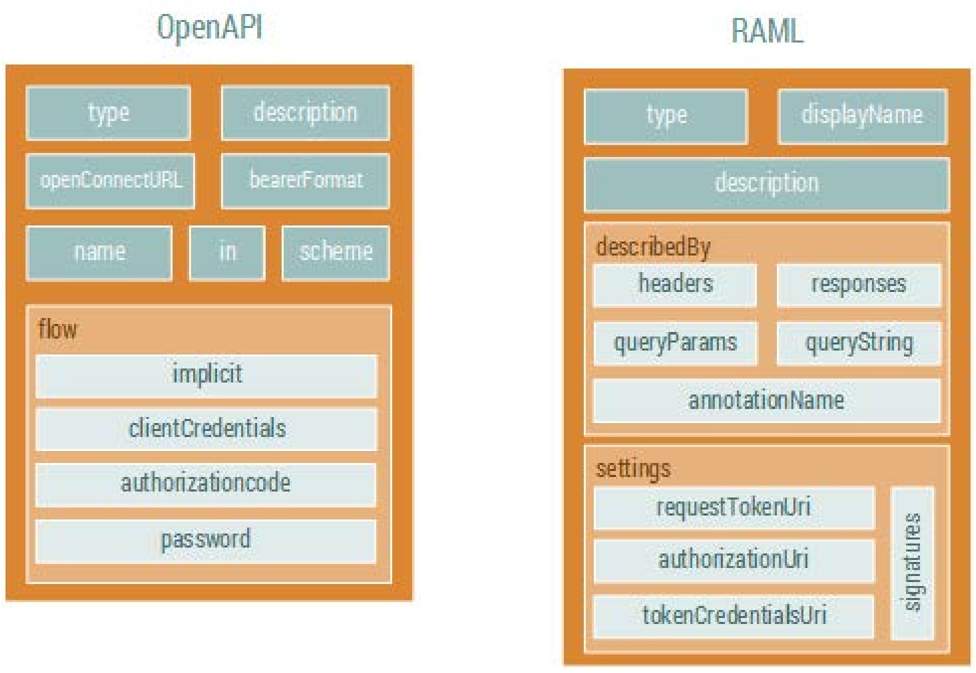

Figure 13: Comparing OpenAPI and RAML specification of security configurations

4.2. Communicating security requirements

The concepts present both in OpenAPI and RAML denote `Security Requirements`, as in the former, and `securedBy` reference, as in the latter case. Both of these approaches are applied in the various levels of descriptive documents and reference existing mechanisms, whose configuration parameters are already initialized in the related container structure.

Referencing security requirements in OpenAPI relies on a simple structure that contains fields for `name` and `value`. While the name refers to the security scheme declared prior in the Security Definitions, the value is by default empty and must be populated if OAuth 2 or OpenID Connect are used. In the latter case, the value needs to correspond to scope names required for the particular execution.

These cases are illustrated by examples provided in Figure 14, which presents an expression of security requirements for API keys method and OAuth 2 access, respectively. In the first case, we can notice that the execution of some operation may be protected by the requirement for the caller to provide an API key. This description hence provides the information to the caller that a particular information or a resource requires a possession and integration of API key in the call. In the later case, the same could be translated to the possession of an access token with two authorization scopes denoted in the structure.

```
{
  "api_key": []
}
```

```
{
  "petstore_auth": [
    "write:pets",
    "read:pets"
  ]
}
```

Figure 14: Security requirements in OpenAPI considering API keys and OAuth 2 methods

Following its reuse capabilities, RAML communicates security requirements similarly by applying `securedBy` reference to a particular object. It however extensively relies on reuse, enabling the less verbose specifications to refer to existing segments located outside of the document. This is demonstrated by the example from Figure 15. In this snippet the configuration parameters for OAuth 2.0 authentication are defined in the separate file. These are referenced by `securedBy` inside of a particular URL path.

```
title: GitHub API
version: v3
baseUri: https://api.github.com
securitySchemes:
  oauth_2_0:!include securitySchemes/oauth_2_0.raml
/users/{userid}/gists:
  get:
    securedBy: [null, oauth_2_0: { scopes: [ administrator ] } ]
```

Figure 15: Security requirements in RAML

The example above implies that the resource may be called without authentication (the existence of `null` parameter in the definition). In the case of authentication, this example requires predefined OAuth 2.0 flow to be applied and communicates *administrative* scope as required authorization to fully access the resource.

Both of these approaches integrate security mechanisms that are currently broadly in use. While OpenAPI partially includes OpenID Connect as well, the RAML does not assume this protocol as a part of its primary specification. Although relying on hard-wired definitions, both of these approaches allow automated agents to automatically derive related configuration data and take part in authentication flows required for particular action. These approaches, however, require manually based and hard-wired implementation of such functionalities. This hinders the adoption of potentially new security mechanisms and requires additional manual work on the side of client applications and developers to integrate all required mechanisms.

As they primarily focus on authentication mechanisms present among a broader range of available security-related controls, the communication of other security requirements and their practical integration in applications is still to be further considered. The security frameworks, such as RMIAS, deliver a range of other potential security controls that need to be further structurally presented and integrated into application flows.

4.3. Understanding authorization scopes

Authorization scopes define the extent of the authorization consent that resource owner agrees upon while delegating resources or access to them to other entities. They are a part of OAuth 2.0 framework, which is currently only cross-organizational authorization framework broadly used on the web.

Although syntactically different, both RAML and OpenAPI approaches illustrate hard-wiring aspect that characterizes the description of security capabilities and requirements. The meaning and role of the fields and structures can only be implicitly assumed; it is not consistent across different mechanisms and need to be integrated in out-of-the band process by developers.

In a practical situation referring to *authorization scopes*, this means that the accessing agent may retrieve the data structures that reference particular scopes, such as *write:pets* or *administrator* in above two cases, but cannot infer their real meaning. The accessing agent hence has no information on what actually a particular scope means and what is necessary to perform to satisfy its represented requirement. Such descriptions, although syntactically correctly structured, can still be fully understood only by humans (developers) who access the descriptions and correlate them with the documentation and implementation manuals that are separately available and aimed at humans. Related explanations of this issue on a general level have been provided in [8, 9], while more details dealing with security domain have been provided in [10, 11] and [12] concerning authorization scopes.

As the machines do not have access to relevant information and cannot derive the out-of-the band knowledge, they cannot be involved in deeper integration levels. This deficit can be partially compensated by applying coupling and hard-wired based approaches; they however impose high development and maintenance overheads and can potentially lead to issues in terms of interoperability and security.

4.4. Integrating authorization capabilities

The frameworks analyzed in this work consider a range of security mechanisms to protect APIs, including API keys, OAuth 2, OpenID Connect and diverse HTTP Authorization means. The API keys mechanism itself is traditionally applied to secure the access to the whole API. This means that a particular API key practically cannot be applied to secure different parts (functions) of an API or to provide context-sensitive security. OAuth 2 goes beyond that limitation by establishing authorization scopes, of whom each may correspond to some API part or an underlying functionality. The meaning of authorization scopes is, however, opaque for the application as it can be derived only by reading and interpreting the natural language based documentation that is primarily intended for humans.

The API description approaches, such as OpenAPI and RAML, do not provide facilities to describe the meaning and extent of authorization scopes beyond their simple declaration. In addition to that, these frameworks do not assume the mechanisms for correlation between other structures in the document, such as RESTful paths and structures, and authorization scopes.

After achieving the understanding of the role and extent of authorization scopes, the next step would be their practical integration into interaction flows. However, without the means for the machine to infer und understand their meaning, especially across the domains, the practical application of the scopes is limited to their out-of-the-band enumeration. This also implies the restricted application of scopes within the RESTful ecosystem, whose dependence on manual integration hinders capabilities of automated and inter-domain security management.

5. Beyond API Descriptions

This section presents the results of the work that aims to bring together potential synergistic effects of existing Web API description approaches, such as OpenAPI and RAML frameworks, and novel multilateral semantic data security framework that enables cooperative and transformative security management across domains.

5.1. Positioning description frameworks

API description frameworks, such as OpenAPI or RAML, primarily serve the purpose to provide the companies and developers with the tools to assist the operations within API lifecycle management. The lifecycle of APIs contains several milestones, which are described in the following paragraphs.

In the first phase, **API design**, the requirements are established and API is designed and structured to correspond to the needs of relevant parties. By relying on a structured code that describes the API in the background, the users can employ diverse tools and other helpers to visually design interfaces using as little coding as possible. Furthermore, the integration of visual tools in the whole lifecycle may accelerate and improve the process of redesigning and improving APIs by reusing existing API representations in structured form.

Building is a second phase during which an API gets programmatically generated, what usually corresponds to coding activities using a particular language and a framework. This activity is usually performed manually and may be error-prone task. The API descriptions in this phase are employed to derive automated code that can be easily integrated in different languages and frameworks with a minimal coding effort.

Testing represents an important step in the implementation and quality assurance of each API, supporting the production of bug-free code as well as harmonization of inputs and outputs of interfaces. Testing also facilitates the evolution of APIs by maintaining backward compatibility across the different versions. The execution of these tasks is significantly improved by reusing the structured descriptive code that can be applied to automatically generate tests, examine their coverage or check the conformance to overall requirements.

Documenting is an important task in every software development process. The reliance on structured code for API description enables automated maintenance of human-readable documentation in different formats that are suitable for various end-user devices or applications.

Reuse is a step that allows the application of existing API descriptions by several parties in a collaborative way. In this sense, APIs can be improved, extended, enhanced or simply reused in other processes or systems on a structured way that is independent on maintainer's infrastructure or processes.

It should be noticed that all these activities gravitate to the software development process of an API. From this point, the primary purpose of API description frameworks is to remove existing barriers that occur during the maintenance, coding, documenting and visualizing of APIs. Accordingly, the technology and approach selected to provide API descriptions correspond to these goals but impose some limitations to the potential application of descriptions in other areas. For instance, the reliance on pure syntactic-based descriptions restricts their reuse in the quickly advancing fields that recently gained a broad public attention, such as artificial intelligence, cognitive computing, smart systems and systems of systems. Similarly, the tree-based data structures utilized to format the documents impose predefined and less flexible constructs that hinder the establishment of rich relations between different nodes.

Furthermore, both of the approaches considered in this document restrict the application of API descriptions in the above-mentioned areas, including information security as well. This is due to the strict formats that do not structurally expose underlying semantics.

5.2. Application within a broader security-related ecosystem

DASP Framework (DAta Sharing and Processing) is a set of tools and vocabularies used to define and manage security policies across (organizational) domains and protocols, with the focus on transactions based on Web-API interfaces [11, 12]. This work aims to bridge existing API description frameworks with the DASP framework, enabling the reuse of available API descriptions and their integration with concepts and functionalities provided by DASP.

Table 1 presents the reduced set of elements from DASP-Core vocabulary that are relevant for the scope of this work. In a typical scenario, these elements are applied to describe underlying semantics of web services using the constructs such as ontologies (vocabularies), classes, axioms and relationships.

Class	***Description***
`Service`	*Abstract system that exposes resources and operations*
`Resource`	*Basic type that represents retrievable resource by the means of API*
`Element`	*Element is a consisting part of a resource*
`Action`	*Entity that represents the activity that can be executed on a resource*
`Operation`	*References the operation that can be executed over the particular element or action result*

Table 2: Basic classes used for API modelling in DASP-Core ontology

Due to the expressive modeling capabilities of underlying RDF and OWL languages [13, 14], the models based on DASP framework separate the description from the instantiation of models[10] and can support multiple relations and inheritance across the classes. The models established with this framework can be exposed side-by-side with standard API resources, by integrating them in API call response headers, or they can be retrieved using separate endpoints as well.

The description of API interfaces using classes in DASP-Core vocabulary starts with the `Service` class, which denotes a service compartment and provides its other properties. By applying service classes inherited from Service-based concept hierarchy, the different type of services can be specified. The service then exposes a `Resource`, which represents an entity that is provided by the API and as such can be retrieved or modified. An `Element`, on the other hand, is a constitutive part of a `Resource` that is provided through the interface dedicated to the Resource. I

n a typical case, the element can be isolated by applying an XPath expression, or it can be presented as a part of a JSON document. `Actions` and `Operations` share similar functionality. While *action* represents an activity that can be executed over a resource as a whole, *operation* denotes a transformation that can be executed over particular element or action result and is ephemeral in nature. In other words, the operation provides a temporary, context-sensitive representation of an element or result of an action.

Figure 16 shows a mapping between concepts in OpenAPI and RAML, on the one side, and reduced DASP-Core vocabulary. The purpose of this mapping is to enable the reuse of existing API descriptions in one of these two formats in DASP security framework. This way, the tools that are part of the framework can be employed to provide security functionality that is scalable and decoupled from original web services or their internal processes.

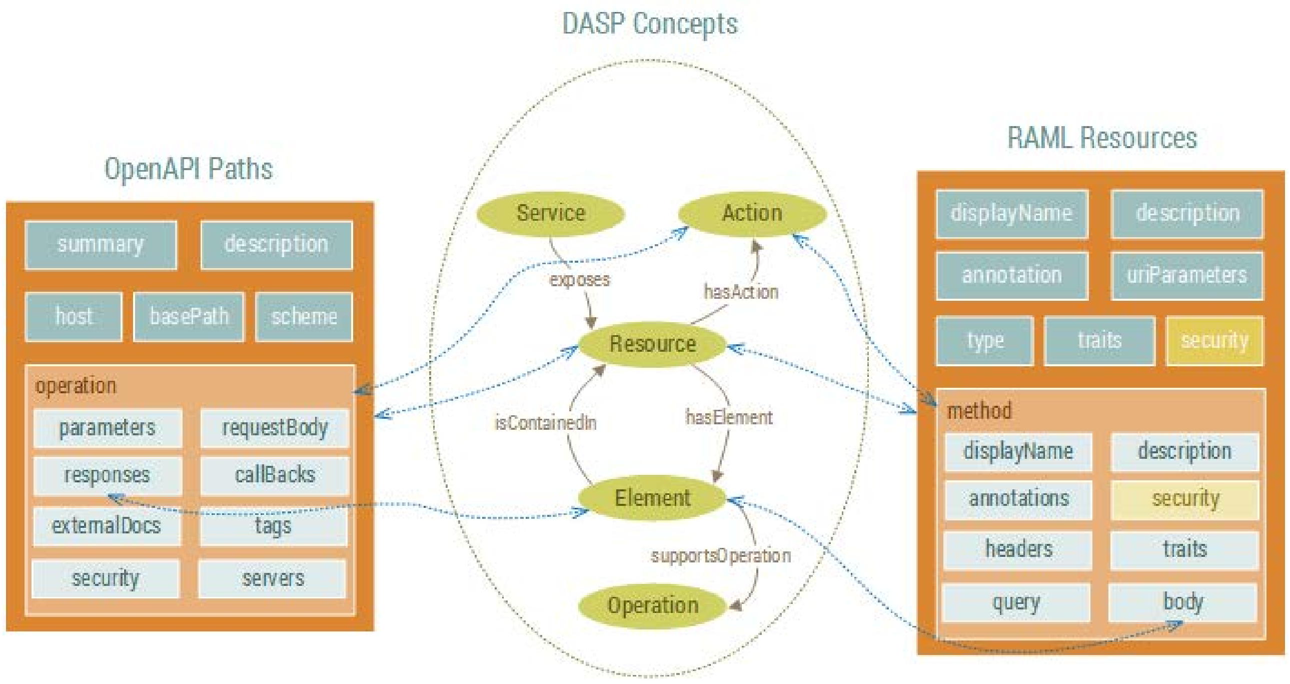

Figure 16: Mapping OpenAPI and RAML descriptions to the concepts of DASP ontology

[10] A-Box vs T-Box modelling

5.3. Benefits of integration of API descriptions

The integration of API description formats with other technologies and mechanisms allows a range of applications that extend the overall applicability beyond initially envisioned functionalities such as the generation of human-readable documentation or of implementation code. This section considers the benefits arising from the integration with semantic-based structures and tools for multilateral data security management[11] on a high-level.

Cross-system interoperability

API descriptions based on syntactic structures by default support the interoperability only on syntactical level [15]. This kind of interoperability requires that each party that consumes the structure has to correlate it with its internal information and knowledge representation models by manually deriving the meaning from specified syntax and human-readable descriptions and adopting it to conform to the local environment. This process is costly, time-consuming and error-prone, leading to the lower level of interoperability or reuse across the systems. From this perspective, different organizations or systems tend to implement own structures that are coupled with (own) proprietary systems and do not scale well to other parties. This is especially true for interoperability with other systems, as there are relatively high interoperability obstacles present, which require additional effort to be invested.

The integration with reference semantic framework enables to establish the interoperability between different systems on the semantic level [15]. This transfers the interoperability requirements from the lower syntactic to higher semantic layer, allowing the exploitation of the data and structures beyond organizational or system's boundaries in less costly manner. In this sense, the syntactic or other adjustments in the data representations on each side can be automatically resolved with less overhead by relying on the properties of semantic technologies, such as inference and automated alignment.

Semantic correlation for machine-based understanding

Typical API description structured in the framework such as RAML does not provide the information on the conceptual level, which is a crucial prerequisite to support machine-based understanding and reasoning. Such processes require the inferable information that explains underlying concepts and relationships, including what kind or resources is used, what kind of method is applied, what type of parameters it receives or provides, and which kind of relation exists with another resources or entities from the same or other descriptive documents. Even when the objects are reused in the same document, the software that reads descriptions cannot automatically infer the knowledge about the real type of that object or its role in a particular context. Instead, this information has to be hard-coded and manually correlated not only at the creation of the relationship, but this information also needs to be maintained throughout the process lifecycle.

The reliance on semantic mappings allows automated correlation of resources with existing models and derivation of their meanings, relationships, and roles based on the intrinsically or extrinsically provided knowledge in the model. Due to the expressive capabilities of underlying framework, these relationships can be re-applied to derive supplementary and extensive knowledge, which can be further employed to provide additional functionality or integration with other tools. Therefore, the reliance on semantic models enables the transition from data-based structures to knowledge-based relationships which enable a plethora of potential applications envisaged under cognitive and autonomous computing [16].

Application outside of API models

As shown in Section 4.1, the integration of existing description frameworks with other tools from their environments represents basically passive and unidirectional interaction. These frameworks primarily serve to support activities such as building the documentation or generating client libraries. The active and bidirectional integration, on the other hand, would mean that other tools could reference description structures and use them in a dynamic manner that allows adjustments and back-references on both sides. One example of such applications is the referencing of the elements provided by these frameworks for the purpose of security or administrative management

[11] Particularly considering DASP framework

of web and cloud services. The application related to security management would hence allow a model of a resource exposed by the API to be reused for authorization management within the system or, in a more complex scenario, across the systems.
Due to the static, hard-wired underlying structure of existing frameworks, such scenarios are not easily attainable in practice and impose an extensive re-engineering and maintenance overhead. By incorporating the API descriptions with DASP framework and applying the techniques such as *semantic uplift* and *semantic down-lift* [17] the static and passive data provided in description documents can be converted to conceptual models and reused in a new layer that supports knowledge management and derivation beyond syntactic descriptions.

Requesting authorization
In initial steps of typical interactions[12] the clients request a range of authorizations from resource owner. Existing frameworks structure these requests as access scopes, which represent an opaque structure without automatically inferable meaning[13]. This incurs a range of implications to security and automation, including the inability to correlate the scopes across and within the systems, to dynamically manage scopes, to relate scopes to resources and roles or to discover the extent of provided authorizations. The details on these and other issues are provided in [11, 12].
API descriptions translated to and expressed through the concepts from DASP framework, allow to overcome these issues by providing a means to express the degree of the requested authorization on a way that is identifiable across the systems and can be traced back to the resource, role or permission. This allows the agents to dynamically structure, derive and understand the extent of requested authorizations. Similarly, the application of structured requests enables the resource owners and servers to correlate and manage requested and provided permissions within a particular or across the different systems.

Implementing access control
The typical access control of Web API accesses often relies either on API-keys based access control or on OAuth-based authorization consents. Both of these cases do not allow for integration with expressive security policies. While API keys determine the access control to the whole API, OAuth scopes allow the separation of API parts or functionalities and issuance of different access tokens for requests to separate parts of functionalities of APIs. However, this solution does not scale well nor allow a full interoperability across different systems. The implementation of access control is furthermore pertained with additional overhead, as the scopes need to be related and maintained to accesses and resources in a non-transparent way.
The application of API descriptions brings the possibility to reuse existing API models, as well as request and response specifications, and to relate them to structured security policies. This way, the security policies can be related to concepts both on syntactic (system and platform specific) and semantic (cross-system) layers. By modeling requests, responses, security policies and resources using the same framework[14], the complete process of managing and enforcing security policies is more transparent and less error prone.

Checking and auditing permissions
API descriptions provide a model of Web API that gives a complete overview over available endpoints, as well as their supported methods, expected and provided resources. The establishment of models that describe other relevant activities, resources[15] and rules that govern them[16] allow analyzing a coverage of security policies or API endpoints and deriving additional knowledge about the reach and applicability of security policies. This way the vulnerable resources or missing security policies can be determined with lower overhead.

[12] Such as the ones from OAuth and UMA protocols

[13] The accessing agent cannot automatically or autonomously derive the meaning or purpose of the scope, nor relate it with an existing authorization, role or object.

[14] DASP structures and related tools that enable their integration

[15] Such as requests and responses

[16] Such as security policies

6. Conclusion

Web APIs play increasingly important role in data sharing over the Internet. By relying on technologically neutral and non-proprietary technologies, Web APIs represent a key building block in modern cross-organizational web-based interactions. The lack of standardized approaches, a diverse range of implementations and representations of RESTful interfaces, as well as a present diversity in applied models and their maturity, however, limit the practical adoption of RESTful Web APIs. A range of approaches exist that support the lifecycle management of REST APIs. They facilitate the execution of activities that include design, maintenance, documentation, testing or code generation from specifications, allowing a faster time-to-market, easier collaboration, sharing and implementation of Web API interfaces. The existing solutions, however, do not primarily focus on security, implementing only a subset of features potentially applicable to the security domain.

This work considered two dominant approaches, OpenAPI and RAML, performing the analysis of their practical applicability and features related to security. This work additionally investigated the joint application and possible synergies between industry-backed specifications such as OpenAPI and RAML, with emerging DASP framework for multilateral security management. In the scope of this work, we established the mappings between concepts in OpenAPI and RAML specifications and classes and relationships present in DASP framework. By providing integration points for these frameworks we enabled their reuse and joint application, extending the initial lifecycle management activities with the security management related capabilities.

References

[1] Nordic APIs, "Ten New Breeds of Businesses That Have Emerged out of the API Economy," 2016. [Online]. Available: http://bit.ly/2k7T4ie. [Accessed 2017].

[2] European Commision, "Building a European Data Economy," 2017.

[3] European Commission, "Report on the Implementation of the Communication 'Unleashing the Potential of Cloud Computing in Europe'," 2014.

[4] E. Parliament, "DIRECTIVE (EU) 2015/2366 OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL," 2015.

[5] M. Salmony, "The Concept of an Open Standard Interface for Controlled Access to Payment Services (CAPS)," 2014.

[6] F. Bülthoff and M. Maleshkova, "RESTful or RESTless - Current State of Today's Top Web APIs," in *European Semantic Web Conference*, 2014.

[7] Payments UK, "APIs – what do they mean for payments?," 2016.

[8] M. Lanthaler, "On Using JSON-LD to Create Evolvable RESTful Services," in *Proceedings of the 22nd International World Wide Web Conference (WWW2013*, 2013.

[9] M. Lanthaler, "Creating 3rd Generation Web APIs with Hydra," in *Proceedings of the 22nd International World Wide Web Conference (WWW2013)*, 2013.

[10] B. Suzic, "Securing integration of cloud services in cross-domain distributed environments," in *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, 2016.

[11] B. Suzic, "User-centered Security Management of API-based Data Integration Workflows," *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium,* pp. 1233-1238, 2016.

[12] B. Suzic, *Multidimensional Security Policies,* Graz: Zentrum für sichere Informationstechnologie - Austria (A-SIT), 2016.

[13] R. Cyganiak, D. Wood and M. Lanthaler, "RDF 1.1 concepts and abstract syntax," W3C, 2014.

[14] W3C Owl Working Group, *OWL 2 Web Ontology Language Document Overview,* W3C, 2009.

[15] H. der Veer and A. Wiles, "Achieving Technical Interoperability," ETSI, 2008.

[16] P. Baranyi and A. Csapo, "Definition and synergies of cognitive infocommunications," in *Acta Polytechnica Hungarica 9.1*, 2012.

[17] A. I. Rana and B. Jennings, "Semantic uplift of monitoring data to select policies to manage home area networks," in *Advanced Information Networking and Applications (AINA), 2012 IEEE 26th International Conference on*, 2012.

[18] D. Hardt, *The OAuth 2.0 authorization framework.,* 2012.

[19] I. Salvadori and F. Siqueira, *A Maturity Model for Semantic RESTful Web APIs,* IEEE, 2015.

[20] M. Sporny and L. Markus, *Json-ld 1.0-a json-based serialization for linked data,* W3C, 2014.