Interface Quality Patterns — Communicating and Improving the Quality of Microservices APIs

Mirko Stocker
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland

Olaf Zimmermann
University of Applied Sciences of Eastern Switzerland, Rapperswil, Switzerland

Uwe Zdun
University of Vienna, Faculty of Computer Science, Software Architecture Research Group, Vienna, Austria

Daniel Lübke
iQuest GmbH, Hanover, Germany

Cesare Pautasso
Software Institute, Faculty of Informatics, USI Lugano, Switzerland

ABSTRACT

The design and evolution of Application Programming Interfaces (APIs) in microservices architectures is challenging. General design issues in integration and programming have been covered in great detail in many pattern languages since the beginnings of the patterns movement, and service-oriented infrastructure design patterns have also been published in the last decade. However, the interface representations (i.e., the content of message payloads) have received less attention. We presented five structural representation patterns in our previous work; in this paper we continue our coverage of the API design space and propose five interface quality patterns that deal with the observable aspects of quality-attribute-driven interface design for efficiency, security, and manageability: An *API Key* allows API providers to identify clients. Providers may offer rich data contracts in their responses, which not all consumers might need. A *Wish List* allows the client to request only the attributes in a response data set that it is interested in. If a client makes many API calls, the provider can employ a *Rate Limit* and bill clients according to a specified *Rate Plan*. A provider has to provide a high-quality service while at the same time having to use its available resources economically. The resulting compromise is expressed in a provider's *Service Level Agreement*.

CCS CONCEPTS

• **Software and its engineering → Patterns**; *Designing software*;

ACM Reference Format:
Mirko Stocker, Olaf Zimmermann, Uwe Zdun, Daniel Lübke, and Cesare Pautasso. 2018. Interface Quality Patterns — Communicating and Improving the Quality of Microservices APIs. In *23rd European Conference on Pattern Languages of Programs (EuroPLoP '18), July 4–8, 2018, Irsee, Germany*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3282308.3282319

1 INTRODUCTION

A recent trend in the software landscape is the ubiquity of message-based remote Application Programming Interfaces (APIs): software-as-a-service providers, for example Salesforce, increasingly provide not just end-user facing websites but also offer the same services as APIs to third parties, making it possible to integrate services into other applications or to combine APIs to enable new use cases[1]. This trend has been called the *API economy* [21].

APIs have also become more important in intra-company use cases: starting with Service-Oriented Architecture (SOA) [22] and culminating in the microservices trend [24], software systems have become more and more distributed. On the client side, single page applications (i.e., client programs written in JavaScript communicating with one or more backend APIs over the Internet) have also greatly contributed to the proliferation of APIs. This trend will likely continue into the near future with masses of internet-enabled things arriving in our homes.

Quality aspects of APIs play an important role in the API economy and in intra-company use cases; they have to be communicated (from providers to clients) and to be achieved/improved. We have identified several patterns that provide answers on how to achieve a certain level of quality of the offered services in terms of API design and usage at the interface level. When applying these patterns, API providers often need to identify the clients making API calls, for example by assigning each client a unique *API Key*. Providers serve a potentially large and diverse group of clients and may offer rather rich response data sets; not all clients might need all of this information all the time. A *Wish List* allows clients to request only the attributes in a response data set that they are interested in. Providers need to economise their resources – by slowing down heavy users or by billing them more. A *Rate Limit* can be used to block clients that have exceeded a predefined limit of API usage. Providers can define a *Rate Plan* for the API usage to bill clients or other stakeholders. On the other hand clients need to know that a provider can deliver an acceptable service quality. Providers can use a *Service Level Agreement* to specify measurable aspects of performance, scalability, and availabiliy and define penalties and compensation credits.

[1]An example of such a service is https://ifttt.com, who allows non-programmers to connect APIs of different providers.

We have collected these patterns by studying 31 Web APIs and API-related specifications. We also reflected on our own professional experience and interactions with practicing architects and developers.

API providers need to balance different, conflicting concerns to guarantee high service quality while ensuring cost-effectiveness. Hence, all five patterns presented in this paper address or contribute to the following overarching design issue:

> *How to achieve a certain level of quality in an offered API, while at the same time using the available resources in a cost-effective way?*

The paper is structured in the following way. Section 2 discusses relations to patterns in other languages. Section 3 outlines the terminology and basic abstractions that are used throughout the paper. Section 4 then introduces five interface quality patterns. Section 5 summarizes and gives a brief outlook on future work.

2 RELATIONS TO OTHER PATTERNS AND PATTERN LANGUAGES

The patterns presented here are a continuation of our work on Interface Representation Patterns [25] that introduced five basic patterns for structuring messages in remote APIs: *Atomic Parameter*, *Atomic Parameter List*, *Parameter Tree*, *Parameter Forest*, and *Pagination*.

These structural interface representation patterns deal with the following design issue:

> *What is an adequate number of API message parameters and how should these parameters be structured?*

For instance, in an HTTP context, this design issue can be interpreted as how the parameters transported with the message are structured and how many parameters are transported. In an HTTP resource API usually the request body is used for data sent to or received from the server (e.g., in JSON, XML, or another MIME type), and query parameters of the URL can also be used to further specify the requested data. In a WSDL/SOAP context, we can interpret this design issue as how should the SOAP message parts be organized and which data types are used to define the corresponding elements in XML Schema (XSD). Similar considerations apply to other technologies such as gRPC or Avro.

A summary of the problem-solution pairs of our interface representation patterns that are referenced by the patterns presented in this paper can be found in Table 1.

The relationships between the patterns presented in this paper and the previously published ones are shown in Figure 1.

Both Release It! [14] and R. Hanmer's pattern language Patterns for Fault Tolerant Software [11] introduce many patterns for improving the stability, resilience and reliability of software systems. These high-level concerns are important quality characteristics for most (if not all) API provider implementations. In contrast to our patterns, these patterns focus on the internal architecture and implementation of the software system, while our patterns describe characteristics of the API description.

3 BASIC ABSTRACTIONS AND CONCEPTS

This paper uses a number of basic abstractions and concepts which form the domain model of our pattern language. At the most abstract level, there are two kinds of *communication participants* (or participants for short) that communicate via an *API*: the *API provider* and the *API client*. An *API provider* exposes any number of *APIs*; an *API client* uses any number of *APIs*. One participant can also play both roles (for instance, in an *API Gateway* [17] in which the *communication participant* offers services as the provider of the gateway and is client to the services shielded by the gateway).

In the client role, a *communication participant* uses *API endpoints* to access the *API*. An *API endpoint* is a provider-side end of a communication channel and a specification of where the *API* resources are located so that *APIs* can be accessed by *API clients*. Each endpoint thus needs to have a unique address such as a Uniform Resource Locator (URL), as commonly used on the World-Wide Web, as well as in HTTP-based SOAP or RESTful HTTP. Each *API endpoint* belongs to an *API*; one *API* can have different endpoints.

The *API* exposes *operations*. In addition to the endpoint address, an operation identifier is needed to identify the operation. For instance, in SOAP this is the top-level XML tag in the body of the message (if WSDL is used to describe the service, named after a WSDL operation element); in RESTful HTTP this is the name of the HTTP method (or verb) such as GET.

The *operations* of an *API* can be invoked by a *conversation*. A *conversation* is any kind of exchange of *messages* (i.e., the conversation uses *messages*). For instance, a *conversation* can be a *call* conversation which usually uses a *request message* and a *response message* (unless the call is a one-way call which omits the response message).

Finally, all *operations* are part of the technical *API contract* (which usually details all possible *conversations* and *messages* down to the technical *parameter* representations and *addresses*). Thus, the contract describes the *API endpoint*. *API contracts* are necessary for realizing any interoperable and testable technical communication; that is, in order to be able to communicate, *API clients* must comply with the *API providers contract* for those parts of the *API* that are used. This can be done explicitly at design time (with the help of static contracts) and/or at runtime (to achieve a more dynamic contract nature).

These classes and relationships of the domain model form the basic vocabulary for all sections of the following pattern texts.

4 PATTERNS FOR COMMUNICATING AND IMPROVING INTERFACE QUALITY

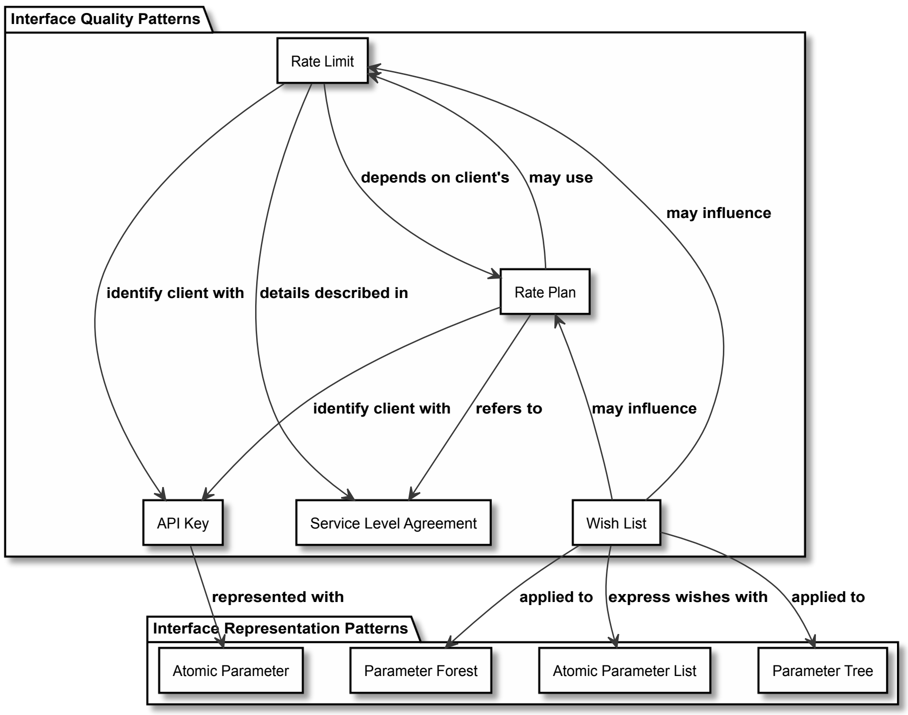
The quality of an API has many dimensions, starting with the accuracy of the functionality described in the API contract, but also including many other qualities such as reliability, performance, security, and scalability [2]. These operational technical qualities are often referred to as Quality-of-Service (QoS) properties. QoS qualities might be conflicting among each other, and almost always need to be balanced with development concerns such as changeability [15] and economic forces such as costs.

The following patterns presented in this paper can be used to communicate the quality attributes of an API and also to improve them:

Table 1: Problem-solution pairs of previously published Interface Representation Patterns.

Pattern	Problem	Solution
Atomic Parameter	How can an API provider define a single, primitive data element as parameter in a request message or a response message?	To exchange a simple, unstructured data element (such as a number, a string, or a boolean value), define only a single scalar parameter for a message.
Atomic Parameter List	How can the API provider define multiple primitive data elements as parameters in a request message or a response message?	To transmit two or more simple, unstructured data elements, define the message's parameters as multiple *Atomic Parameters* (such as numbers, strings, or boolean values) arranged in an ordered list.
Parameter Tree	How can the API provider define tree data structures in the parameters of a message? How can the API provider define repetitive or nested data elements in the parameters of a message?	Define the parameter representation of a message based on a single root data element that contains one or more subordinate composite data structures such as tuples, arrays, or trees.
Parameter Forest	How can the API provider define repetitive or nested data between elements in the parameters of a message that cannot or should not be represented well in a single tree structure?	Define the parameter representation of a message as multiple simple and composite data structure representations, including scalars, lists, and complex types like trees, arranged in an ordered list of those structures.

Figure 1: Relationships between the patterns presented in this paper and selected Interface Representation Patterns [25].



API Key: An API provider needs to identify the communication participant it receives a message from to decide if that message actually originates from a registered, valid customer or some unknown client. A unique, provider-allocated *API Key* per client to be included in each request allows the provider to identify and authenticate its clients. This pattern is concerned with the quality attribute *security*.

Wish List: Performance requirements and bandwidth limitations might dictate a parsimonious conversation between the provider and the client. Providers may offer rather rich data sets in their response messages, but not all clients might need all of this information all the time. A *Wish List* allows the client to request only the attributes in a response data set that it is interested in. This pattern addresses qualities such as *accuracy* of the information needed by the consumer, *response time*, and *performance*, i.e., processing power required to answer a request.

Rate Limit: Having identified its clients, an authenticated client could use excessively many resources, thus negatively impacting the service for other clients. To limit such abuse, a *Rate Limit* can be employed to restrain certain clients. The client can stick to its *Rate Limit* by avoiding unnecessary calls to the API. This pattern is concerned with the quality attributes of *reliability*, *performance*, and *economic viability*.

Rate Plan: If the service is paid for or follows a freemium model, the provider needs to come up with one or more pricing schemes. The most common variations are a simple flat-rate subscription or a more elaborate consumption-based pricing scheme [10], explored in the *Rate Plan* pattern. This pattern addresses the *commercialization* aspect of an API.

Service Level Agreement: API providers want to deliver high-quality services while at the same time using their available resources economically. The resulting compromise is expressed in a provider's *Service Level Agreement* (SLA) by the targeted service level objectives and associated penalties (including reporting procedures). This pattern is concerned with the communication of any quality attribute between API providers and clients. *Availability* is an example of a quality that is often expressed in such an SLA.

The primary target audience for the first four patterns are API architects and developers. The last two patterns concern business aspects of APIs and are thus more relevant for API product owners.

4.1 Pattern: *API Key*

a.k.a. Access Token, Provider-Allocated Client Identifier

Context. An API provider offers services to subscribed participants only. One or more clients have signed up and want to use the services. These clients have to be identified.

Problem. How can an API provider identify and authenticate different clients (that make requests)?

Forces. When identifying and authenticating clients on the API provider side, the following forces come into play:

- How can client programs *identify* themselves at an API endpoint without having to store and transmit user account credentials?
- How can a client calling an API be *decoupled* from the client's organization?
- How can varying levels of API *authentication*, depending on security criticality, be implemented?

When resolving these forces, conflicts between security requirements and other qualities make trade offs necessary:

- How can security, in particular identification and authentication of clients at an endpoint, be established while still making the API easy to use for clients?
- How can endpoints be secured while minimizing performance impacts?

Non-solution. A rich portfolio of application-level security solutions adressing Confidentiality, Integrity, and Availability (CIA) requirements is available. However, for a free and public API the management overhead and performance impact might not be economically feasible. For a solution-internal or community API, security could be implemented at the network level with a Virtual Private Network (VPN) or two-way SSL. However, this complicates application-level usage scenarios such as enforcing *Rate Limits*.

Solution. As an API provider, assign each client a unique token – the *API Key* – that the client can present to the API endpoint for identification purposes.

How it works. Encode the *API Key* as an *Atomic Parameter*, i.e., a single string parameter. This interoperable representation makes it easy to send the key in the request header, as part of a URL query string , or in the request body (a.k.a. payload). Because of its small size, including it in every request causes only minimal overhead.

As the API provider, make sure that the *API Keys* you generate are unique and hard to guess. This can be achieved by using a serial number (to guarantee uniqueness) padded by random data and signed and/or encrypted with a private key (to prevent guessing). Alternatively, base the key on a *Universally Unique Identifier* UUID[2]. UUIDs are easier to use in a distributed setting because there is no serial number that needs to be synchronized across systems. However, UUIDs are not necessarily random-generated[3]; hence, they also require further obfuscation just like in the serial number scheme.

Example. In the following call to the Cloud Convert API[4] a new process to convert a DOCX file to PDF format is started. The client creates a new conversion process by informing the provider of the desired in- and output format, passed as two *Atomic Parameters* in the body of the request (the input file has to be provided by a second call to the API). For billing purposes, the client identifies itself by passing the *API Key* `gqmbwwB74tToo4YOPEsev5` in the `Authorization` header of the request, according to the HTTP/1.1

[2]https://tools.ietf.org/html/rfc4122.html

[3]Version 1 UUIDs are a combination of timestamp and hardware addresses: https://en.wikipedia.org/wiki/Universally_unique_identifier#Versions. The *Security Considerations* section in RFC 4122 warns not to "assume that UUIDs are hard to guess; they should not be used as security capabilities (identifiers whose mere possession grants access), for example."

[4]https://cloudconvert.com

Authentication RFC 7235[5] specification. HTTP supports various types of authentication, here the RFC 6750[6] `Bearer` type is used:

```
POST https://api.cloudconvert.com/process
Authorization: Bearer gqmbwwB74tToo4YOPEsev5
Content-Type: application/json

{
    "inputformat": "docx",
    "outputformat": "pdf"
}
```

The API provider can thus identify the client and charge their account.

Implementation hints. When securing an API with an *API Key*, the following advice should be taken into consideration:

- To further protect against brute force attackers trying to guess an *API Key*, an IP range-specific *Rate Limit* can be used to block an attacker's requests.
- Use a well-tested library and refrain from writing your own cryptography code. For example, Java has a Java Cryptography Extension that provides various widely used key generation algorithms.
- Follow the OWASP REST Security Cheat Sheet[7] when securing your HTTP resource API. The cheat sheet contains a section on *API Keys* and contains other valuable information on security as well.
- *API Keys* should not be presented as HTML/HTTP cookies because cookies may be used to identify a user session and might be stored for a long period of time. *API Keys* should also be usable by non-browser clients such as Software Development Kits (SDKs) or command line tools that interact with the API [7].
- Client developers should not accidentally store their shared secrets (e.g., keys) in version control systems[8] or other places where others can get easy access to them (e.g., code repositories).

Consequences. An *API Key* is a lightweight alternative to a full-fledged authentication protocol and balances basic security requirements with the desire to minimize management and communication overhead.

Resolution of forces.

+ Having the *API Key* as a shared secret between the API endpoint and the client, the endpoint can identify the client making the call and use this information to further authenticate and authorize the client.
+ Using a separate *API Key* instead of the customer's account credentials decouples different customer roles, such as administration, business management, and API usage, from each other. This makes it possible to let the customer create and manage multiple *API Keys*, for example to be used in different client implementations or locations, with varying permissions associated to them. In the case of a security break or leak, they can also be revoked independently of the client account. A provider might also give clients the option to use multiple *API Keys* with different permissions or provide analytics (e.g., number of API calls performed) and *Rate Limits* per *API Key*.
+ Because the *API Key* is small, it can be included in each request without impacting performance much.

– The *API Key* is a shared secret, and because it is transported with each request, it should only be used over a secure connection such as HTTPS. If this is not possible, additional security measures (VPN, public-key cryptography) must be used. Configuring and using secure protocols and other security measures has a certain configuration management and performance overhead.
– An *API Key* is just a simple identifier and thus cannot be used to transport additional payload, such as an expiration time or authorizations.

Further discussion. An *API Key* can also be combined with an additional secret key to ensure the integrity of requests. The secret key is shared between the client and the server but never transmitted in API requests. The client uses this key to create a signature hash of the request and sends the hash along with the *API Key*. The provider can identify the client with the provided *API Key*, calculate the same signature hash using the shared secret key and compare the two. This ensures that the request was not tampered with. Amazon uses such asymmetric cryptography to secure access to its Elastic Compute Cloud[9].

Alternatives. Even if combined with a secret key, *API Keys* might be insufficient or impractical as the sole means of authentication and authorization. Consider the case where three parties are involved in an interaction: the user, the service provider and a third party that wants to interact with the service provider on behalf of the user. For example, consider a user who wants to allow a mobile app to store its data on the user's Dropbox account. In this case *API Keys* cannot be used if the user does not want to share them with the third party. For such use cases, consider using OAuth 2.0 instead. Another security technology that could possibly be leveraged is the Security Assertion Markup Language (SAML)[10], which can, for instance, be used in backend integration to secure the communication between backend APIs. These alternatives offer better security but also come with a much higher implementation and runtime complexity.

Another popular alternative is the JSON Web Tokens (JWT) standard RFC 7519[11] (see this JWT Introduction[12]). JWT defines a simple message format for access tokens with a payload. The access tokens are created and cryptographically signed by the API provider. Providers can verify the authenticity of such a token and use it to identify clients. Because JWT tokens, in contrast to *API Keys*, have a payload, the provider can securely store additional information there.

[5]https://tools.ietf.org/html/rfc7235.html
[6]https://tools.ietf.org/html/rfc6750
[7]https://www.owasp.org/index.php/REST_Security_Cheat_Sheet
[8]This has actually happened: https://www.theregister.co.uk/2015/01/06/dev_blunder_shows_github_crawling_with_keyslurping_bots/.
[9]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-key-pairs.html
[10]https://wiki.oasis-open.org/security/FrontPage
[11]https://tools.ietf.org/html/rfc7519
[12]https://jwt.io/introduction

Known Uses. Many public Web APIs use the *API Key* concept, sometimes under different names (such as access token). A few examples are:

- The YouTube Data API[13] supports both OAuth 2.0 as well as *API Keys*. Clients have to generate different *API Keys*, depending on where these keys are used, e.g., server keys, browser keys, iOS and Android keys. These keys can then only be used by the configured apps (in case of the iOS and Android apps), IP addresses or domain names. This additional layer of protection makes a specific key unusable outside of the specified app. This is done to avoid that an attacker could, for example, simply extract the key from an installed Android application and use it to make requests on behalf of that application.
- The GitHub API's primary means of authentication and authorization is OAuth, but basic authentication[14] with a username and token – the *API Key* – is also supported. Here, the *API Key* is not sent via HTTP header but through the password parameter of the basic authentication. For example, a request to access the user resource using the cURL command line tool looks as follows: `curl -u username:token https://api.github.com/user`.
- The API of online payment provider Stripe[15] uses a *publishable* key and a *secret* key. The secret key takes the role of an *API Key* and is transmitted in the `Authorization` header of each request, whereas the publishable key is just the account identifier. This naming scheme might be surprising for clients expecting the secret key to be kept private, like Amazon's secret key, which is never transmitted and only used to sign requests.

Related Patterns. Many web servers use *Session Identifiers* [9] to maintain and track user sessions across multiple requests; this is a similar concept. In contrast to *API Keys*, *Session Identifiers* are only used for single sessions and then discarded.

The Security Patterns in [18] provide solutions satisfying security requirements such as Confidentiality, Integrity, and Authentication/Authorization, and discusses their strengths and weaknesses in detail. Access control mechanisms, such as *Role-based Access Control* (RBAC) or *Attribute-based Access Control* (ABAC), can complement *API Keys* and other approaches to authentication; these access control practices require one of the described authentication mechanisms to be in place.

Other Sources. Chapter 12 of the RESTful Web Services Cookbook [1] is dedicated to security and presents six related recipes. [16] covers two related patterns of alternative authentication mechanism in a RESTful context, *Basic Resource Authentication* and *Form-Based Resource Authentication*.

[19] provides a comprehensive discussion on securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE. Chapter 9 of [20] has a discussion of conceptual and technology alternatives and instructions on how to implement an OAuth 2.0 server. The OpenID Connect[16] specification deals with user identification on top of the OAuth 2.0 protocol.

[13] https://developers.google.com/youtube/registering_an_application

[14] https://developer.github.com/v3/auth/#basic-authentication

[15] https://stripe.com/docs/api

4.2 Pattern: *Wish List*

a.k.a. Data Wish Enumeration, Partial Response Representation Request, Data Selection Profile

Context. API providers need to serve multiple different clients that invoke the same operations. Not all clients have the same information needs: some might just need a subset of the data offered by the endpoint, other clients might need rich data sets.

Problem. How can an API client inform the API provider at runtime about the data it is interested in?

Forces. Multiple clients with different information needs might use an API. Hence, the primary design issues that drive selection and adoption of this pattern are:

- How can a provider satisfy the possibly conflicting information needs of individual clients without having to implement per-client endpoints but still avoiding under- and over-fetching?
- How can a client specify and learn about provider-side selection filters that reduce message verbosity by allowing clients to select the level of detail of the retrieved information?
- How can a provider cope with the increased complexity of its endpoint design with regards to security, testing and maintenance that results from having to create responses tailored to individual client's needs?

Furthermore, API providers also need to balance general quality forces such as response time, throughput and processing time.

Non-solution. These forces could be resolved by introducing infrastructure components such as network- and application-level gateways and caches to reduce the load on the server, but such components add to the complexity of the deployment model and network topology of the API ecosystem and increase related infrastructure testing, operations management, and maintenance efforts.

Solution. As an API client, provide a *Wish List* in the request that enumerates all desired data elements of the requested resource. As an API provider, deliver only those data elements in the response message that are enumerated in the *Wish List*.

How it works. Specify the *Wish List* as an *Atomic Parameter List*, which in some cases can have a special form of being scalar (i.e, a simple *Atomic Parameter*) that indicates a verbosity level such as `minimal`, `medium`, and `full`.

A common variant is providing options for expansion in responses. That is, the response to the first request only provides a terse result with a list of parameters that can be expanded in subsequent requests. The client can select one or more of these parameters in a *Wish List* to expand the request results.

As another variant, define and support a wild card mechanism as known from SQL and other query languages, e.g., a star `*` to request

[16] http://openid.net/connect/

all data elements of a particular resource (which could then be the default, if no wishes are specified). Even more complex schemes are possible like cascaded specifications (like `customer.*` fetching all data about the customer).

As yet another variant, regular expression syntax or query languages such as XPath (for XML payloads) can be used. Likewise, GraphQL[17] or RestSQL[18] offer declarative query languages to describe the representation to be retrieved against an agreed upon schema found in the API documentation.

Example. In the Lakeside Mutual Customer-Care application[19], a request for a customer returns all of its available attributes.

```
curl http://localhost:8080/customers/gktlipwhjr
```

For customer ID `gktlipwhjr`, this would return:

```
{
  "customerId" : "gktlipwhjr",
  "firstname" : "Max",
  "lastname" : "Mustermann",
  "birthday" : "1989-12-31T23:00:00.000+0000",
  "streetAddress" : "Oberseestrasse 10",
  "postalCode" : "8640",
  "city" : "Rapperswil",
  "email" : "admin@example.com",
  "phoneNumber" : "055 222 4111",
  "moveHistory" : [ ],
  "customerInteractionLog" : {
    "contactHistory" : [ ],
    "classification" : {
      "priority" : "gold"
    }
  }
}
```

Alternatively, the client can also provide a *Wish List* of fields in the query string to restrict the result to just those fields. For example, a client might only be interested in the `customerId`, `birthday` and `postalCode` fields:

```
curl http://localhost:8080/customers/gktlipwhjr?\
fields=customerId,birthday,postalCode
```

The returned response now contains only the requested fields:

```
{
  "customerId" : "gktlipwhjr",
  "birthday" : "1989-12-31T23:00:00.000+0000",
  "postalCode" : "8640"
}
```

This response is much smaller; only the information required by the client was transmitted.

Implementation hints. When introducing a *Wish List* in your API, consider the following advice:

- Do not blindly apply *Wish List* to all API operations. Typical result data structures on which *Wish Lists* are applied are *Parameter Trees* and *Parameter Forests*. That is, it usually makes most sense to wish for a subset of the possible data elements as a client, if a complex data set needs to be queried, and the result again is a complex data structure.
- The complexity of endpoint design and programming effort increases to achieve a higher degree of flexibility and optimize qualities like performance. To check whether introducing the *Wish List* leads to better performance, during API development and maintenance you should keep track of the size of messages, number of messages, and measure end-to-end processing times. Also, given the variety of possible client requests, introducing logging helps to detect and fix problematic requests.
- When implementing a *Wish List* that may extend to sub-resources in RESTful HTTP, include *Atomic Parameters* that lists the paths to available sub-resources. Have a look at the practices recommended by Atlassian for the Confluence REST API[20] to further understand this *expansion* concept.
- One option to implement a variant of this pattern in RESTful HTTP is to define multiple representations, one for each type of wish (e.g., verbosity level), and let the client articulate its preferred representation during content negotiation.

Consequences. Wish List helps to manage the different information needs of API clients. Consider applying this pattern if the network has limited capacity and you have a certain amount of confidence that clients only need a subset of the available data; but be aware of the potential negative consequences such as additional security threats, additional complexity, as well as test and maintenance efforts.

Resolution of forces.

+ By adding or not adding attribute values in the *Wish List* instance, the API client expresses its wishes to the provider; hence, the desire for “Datensparsamkeit” (i.e., data parsimony) is met.
+ The provider does not need to provide specialized and optimized versions for operations or to guess required data for clients’ use cases.
+ Clients can specify data they require thereby enhancing performance by creating less database and network load.
– Providers have to implement more logic in their service layers [9] , possibly affecting other layers down to data access as well.
– Providers risk to expose their data model to clients thereby increasing coupling.
– Clients have to create the *Wish List*, the network has to transport this metadata and the provider has to process it.

Further discussion. A comma-separated list of attribute names can lead to problems when mapped to programming language elements. For instance, misspelling an attribute name might lead to an error (if the API client is lucky) or the expressed wish might simply be ignored (which might lead the API client to the impression that the attribute does not exist). Furthermore, API changes might have unexpected consequences; for instances, a renamed attribute might no longer be found.

[17]https://graphql.org
[18]http://restsql.org
[19]Lakeside Mutual is a fictitious insurance company invented by the authors comprising of several microservices to demonstrate the patterns in action. It can be found at https://github.com/Microservice-API-Patterns.
[20]https://developer.atlassian.com/confdev/confluence-server-rest-api/expansions-in-the-rest-api

Solutions using the more complex schemes introduced above (like cascaded specifications, wild cards, or expansion) might be harder to understand and build than simpler alternatives. Sometimes existing provider-internal search-and-filter capabilities like wild cards or regular expressions can be reused.

Before introducing a *Wish List* mechanism, possible negative consequences regarding security, reliability, additional test and maintenance effort, and so on need to be carefully considered. Those aspects are often treated as an afterthought and can get complex to fix once the API is in production.

Known Uses. Wish Lists can be found in many public Web APIs under different names and in many variations:

- The Google Calendar API[21] has the notion of partial requests, partial responses and patches. These can be seen as a variant of the *Wish List* pattern: the client sends a `field` parameter that lists desired values. The API also supports an expression syntax including wild cards.
- The Facebook Graph API[22] supports a variation of this pattern under "making nested requests": Field expansion in the Graph API allows client developers to nest multiple graph queries into a single call.
- Sparse fieldsets in the JSON API[23] specification realize the *Wish List* pattern as well: A client can request that an endpoint returns only parts of a data element in the response message.
- The PayPal API Guidelines[24] use the HTTP header `Prefer: return=minimal` to instruct the API endpoint to return only a minimal representation of the resource. *Wish List* behaviour is also supported via the `fields` query parameter.
- Atlassian JIRA supports *Wish Lists* under the name *expansion*[25] comprehensively, but in a slightly different way. Initial responses are terse, but contain elements that can be "expanded" in further requests and are marked as such. The client can then provide a *Wish List* of parameters in the query string to be expanded, which will then be present in the subsequent response. The same concept is supported by the Confluence REST API[26].
- The Microsoft Graph API[27] has an `expand` parameter with expansion semantics.
- The TMForum, an association of telecommunication providers and suppliers, offers the TMForum REST API[28] which allows query operations to select the resource's attributes that should be returned. Furthermore, it also allows filtering of attributes with various filter expressions.
- A Swiss software vendor specializing on the insurance industry uses partial representations and expanded GETs in its internal REST API Design Guidelines. These guidelines recommend defining a request variable called `fields` that lists the requested fields. A wildcard operator exists; if the `fields` variable is not present, all fields must be returned.
- Another enterprise information system usage of the pattern is outlined in [3].

Related Patterns. Wish List deals with instances of *Parameter Tree* and *Parameter Forest* as "origin" and result data structures, as the patterns are usually applied to more complex data structures.

Using a *Wish List* has a positive influence on a *Rate Limit*, as less data is transferred when the pattern is used.

4.3 Pattern: *Rate Limit*

a.k.a. Quota, Usage Limitation

Context. An API endpoint and the API contract defining operations, messages, and data representations have been established. Clients of the API might have signed up with the provider and, if required, have agreed to the terms and conditions that govern the usage of the endpoint and operations. Alternatively, the offering might not require any contractual relation, e.g., when offered as an open government data service or during a trial period.

Problem. How can the API provider prevent API clients from excessive API usage?[29]

Forces. When preventing excessive API usage that may harm provider operations or other clients, the following more detailed design concerns have to be balanced:

- How can the provider maintain a high performance for all clients while properly economizing its resources?
- How can a provider prevent a client from abusing the API, or minimize the impact of excessive and unwanted usage (fairness)?
- How can a provider offer a reliable, cost-efficient service without overly restricting individual clients' ability to use the service?
- How can a client control its API consumption (if it wants or has to save computing and communication capacity)?

Non-solution. To prevent clients that exhibit an excessive usage from harming other API clients, one could simply add more processing power, storage space and network bandwidth. Often this is not economically viable.

Solution. Introduce and enforce a *Rate Limit* to safeguard against API clients that overuse the API.

How it works. Formulate this limit as a certain number of requests that are allowed per period of time. If the client exceeds this limit, further requests can either be declined, be processed in a later period or be serviced by allocating a smaller amount of resources or by providing only best-effort guarantees.

[21] https://developers.google.com/google-apps/calendar/performance
[22] https://developers.facebook.com/docs/graph-api/using-graph-api
[23] http://jsonapi.org/format/#fetching-sparse-fieldsets
[24] https://github.com/paypal/api-standards/blob/master/patterns.md#projected-response
[25] https://docs.atlassian.com/jira/REST/cloud/#expansion
[26] https://developer.atlassian.com/confdev/confluence-server-rest-api/expansions-in-the-rest-api
[27] https://developer.microsoft.com/en-us/graph/docs/concepts/query_parameters
[28] https://projects.tmforum.org/wiki/display/API/Query+Resources+Patterns

[29] What exactly is deemed excessive needs to be defined by the API provider. A flat rate subscription typically imposes different limitations than a free billing plan. See the *Rate Plan* pattern for a detailed discussion of the trade-offs of different subscription models.

Set the scope of the *Rate Limit*, this can be the entire API, a single endpoint, a group of operations or an individual operation: Requests do not need to be treated uniformly, endpoints can have varying operational costs and token usage can thus differ. For example, retrieving a simple ID costs a single token (unit) in the Youtube API[30], whereas a video upload consumes approximately 1600 units.

Define an appropriate time period, for example, daily or monthly, per API operation or group of API operations after which the *Rate Limit* is reset; this interval may be rolling. Keep track of client calls in the defined time period through monitoring and logging.

A *Rate Limit* can also restrict the amount of concurrency allowed, i.e., the number of concurrent requests a client is allowed to make. For example, under a free billing plan clients could be limited to just a single concurrent request (see for instance the Quandl API[31]).

Example. GitHub uses this pattern to control access to its RESTful HTTP API: Once a *Rate Limit* is exceeded, subsequent requests are answered with HTTP status code `429 Too Many Requests`. To inform clients about the current state of each *Rate Limits* and to help clients manage their allowance of tokens, custom HTTP headers are sent with each rate-limited response.

The following code listing shows an excerpt of such a rate-limited response from the GitHub API. The API has a limit of 60 requests per hour, of which 59 remain:

```
GET https://api.github.com/users/misto
HTTP/1.1 200 OK
...
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 59
X-RateLimit-Reset: 1498811560
```

The `X-RateLimit-Reset` indicates the time when the limit will be reset with a Unix timestamp[32].

Implementation hints. Architects and developers applying *Rate Limits* should:

- Be transparent and polite when communicating *Rate Limits*, current status, and the consequences of having reached them to clients. This can be done by introducing a grace period or interval, e.g., if you are 10% over one day, nothing happens, and then more limiting happens gradually. The current status can be shown in a management dashboard or as part of the API response, as in the GitHub example shown above.
- As an API client, be parsimonious in your API consumption (e.g., avoid unnecessary calls) and keep track of consumed resources (and/or retrieve provider-side usage statistics regularly).
- As an API provider, be aware that consumers might try to work around the *Rate Limit* that you want to enforce, for instance, by sending periodic "mini batch" requests shortly after the time window re-opens. This might lead to undesired and unexpected peak workloads[33].
- Store the current state of the *Rate Limits* in a client database and not just in a stateful session. Otherwise the client could just open a new session to overcome the *Rate Limit.*
- Make sure that the implementation of the *Rate Limit* algorithms and its supporting metering infrastructure do not consume more server-side resources than the throttled or declined client calls would. *Rate Limit* also will not protect you from denial-of-service attacks [13] on a protocol level like a TCP SYN flooding[34].
- A *Leaky Bucket Counter* [11] is a natural realization of a rolling *Rate Limit* interval. Each client is assigned a bucket of tokens that is replenished periodically. For example, if the limit is 1000 requests per minute, the bucket has a size of 1000 tokens and each 1/1000 minute a token is added to the bucket, up to a maximum of 1000 tokens. For each request, a token is removed from the bucket. Once the bucket is empty, further requests will fail. Alternatively, a simpler solution would be to limit only the number of calls per billing period.
- When also implementing the *Wish List* pattern, a *Rate Limit* based on the number of API calls will not treat clients fairly, because a single call might suddenly return vast amounts of data. This is also a problem for complex request schemes (e.g. GraphQL[35] as used in the Facebook Social Graph[36]) where the client can send arbitrary queries to the endpoint and receives potentially huge responses. Such APIs require a more sophisticated accounting method than simply counting the number of requests, possibly including data volumes into the definition of the limit(s). For an inspiration on how to implement this, see the GitHub v4 API[37].

Consequences. A *Rate Limit* gives the provider control over the client's API consumption, but deciding on the right limits is not easy.

Resolution of forces.

+ By implementing a *Rate Limit*, an API provider can protect its offering from malicious clients, such as unwelcome bots, and maintain the quality of its service.
+ The provider can better provision resources due to capped maximal usage thereby improving performance and availability for all clients.

– If the *Rate Limit* is set too high, it will not have the desired effect. Having it set too low will annoy API users. Finding the right levels will need some experimentation and tuning. For example, a provider's *Rate Plan* might allow for 30'000 requests per month. With no additional restrictions, a client could consume all these requests in a short burst of time, probably overwhelming the provider. To mitigate this particular problem, the provider could additionally restrict clients to just one request per second.

[30]https://developers.google.com/youtube/v3/getting-started#quota
[31]https://docs.quandl.com/docs/getting-started#section-rate-limits
[32]Unix timestamps count the number of seconds since January 1st, 1970.
[33]This has been reported as a problem for the free GitHub service, consumed by software engineering research teams.
[34]TCP SYN Flooding and IP Spoofing Attacks can be used in denial-of-service attacks on a protocol level by creating many half-open connections that are not properly closed and can overwhelm the server: https://www.cert.org/historical/advisories/CA-1996-21.cfm
[35]GraphQL is a query language optimized for data organized in graphs.
[36]https://developers.facebook.com/docs/graph-api
[37]https://developer.github.com/v4/guides/resource-limitations/

– Clients need to control their usage and manage the case of hitting the rate limit, e.g., by tracing their API usage and/or by queuing requests. This can be achieved by caching and prioritizing API calls. Systems management patterns [12] can implement continuous monitoring of resource usage over time.

Further discussion. Paid offerings are in a better position to manage *Rate Limits* with multiple subscription levels and accordingly different limits; excessive API usage can even be seen as something positive (because it leads to increased revenue). But a free service does not have to give all its clients the same *Rate Limit* either. It can instead take into account other metrics to accommodate clients of various sizes and stages. For example, Facebook[38] grants API calls proportional to the number of users that have the client's app installed:

> "Your app can make 200 calls per hour per user in aggregate. As an example, if your app has 100 users, this means that your app can make 20,000 calls."

When a client has exceeded its *Rate Limit*, the provider can stop serving the client altogether or just slow it down (or, for commercial offerings, offer to upgrade to a higher-paid plan). This latter case is sometimes also described as *throttling*. Note that the exact terminology differs by providers, and often *Rate Limit* and *throttling* are used interchangeably.

If a client is hitting the *Rate Limit* too often, the account or corresponding *API Key* can even be suspended (Twitter[39] calls this blacklisted[40]).

In order to measure and to enforce the rate limit metrics, the provider needs to identify the client or user. For identification purposes, the API client has obtained a means to identify itself at the endpoint (more precisely, at the security *Policy Enforcement Point*[41] within the API), for instance with an *API Key* or an authentication protocol. If no sign-up is required, for example in a free service, the endpoint has established another way to identify the client, e.g., by IP address.

Known Uses. Rate Limits are implemented in many public Web APIs:

- The GitHub API v3[42] has a 5000 requests per hour per user limit for authenticated requests. Clients can also make unauthenticated requests but these are limited to just 60 requests per hour (as can be seen in the example above). In the new GraphQL-based GitHub v4 API[43], the *Rate Limit* has become more sophisticated and takes into account the number of queried nodes.
- Open Weather Map[44] calls its rate limits *access limitation* and restricts clients to a certain amount of calls per minute, depending on the subscription.
- *Rate Limits* in Quandl[45] depend on the subscription level and also have a limit on the number of concurrent requests.
- The Twitter REST API[46] only allows authenticated clients and has *Rate Limits* divided into 15 minute intervals.
- LinkedIn[47] calls their *Rate Limits* "request throttling". The limits are defined per API call and are defined on an application, user, and developer level.
- The Swiss Federal Administration's registry of companies ("UID-Register") has a public webservice API[48]. The API is free to use but is limited to 20 requests per minute. If the limit is exceeded, a `Request_limit_exceeded` error is returned.
- Many *API Gateways*, such as MuleSoft API Manager[49], allow developers to introduce *Rate Limits*. API gateways often also support *throttling* to further protect the exposed APIs.
- The open Certificate Authority (CA) Let's Encrypt[50] limits the weekly number of certificates issued per registered domain, but also provides a renewal exemption. Its Automatic Certificate Management Environment (ACME) API also limits the number of accounts that can be registered by a given IP address every hour.

Some Web frameworks provide *Rate Limit* as an optional feature. For example, the Play-Guard[51] library for the Java/Scala Play Framework provides a basic implementation.

Related Patterns. The details of a *Rate Limit* can be part of a *Service Level Agreement*. A *Rate Limit* can be dependent on the client's subscription level, which is further described in the *Rate Plan* pattern. In such cases the *Rate Limit* is used to enforce different billing levels of the *Rate Plan*.

To observe individual clients and manage their allowances, the service provider needs to identify the client making a request. Therefore, clients need to present some form of identification (e.g. an *API Key*, an IP address or another authentication practice) so that the API provider can do the bookkeeping.

The systems management patterns published by [12] can help to implement metering and can thus also be used as enforcement points. For example, a *Control Bus* can be used to increase or decrease certain limits dynamically at runtime.

As discussed above, *Leaky Bucket Counter* [11] offers a possible implementation variant for *Rate Limit*.

[38]https://developers.facebook.com/docs/graph-api/advanced/rate-limiting
[39]Rate Limiting in the Twitter API: https://dev.twitter.com/rest/public/rate-limiting.
[40]A blacklist is an access control mechanism that bars certain "blacklisted" elements but lets all others pass. This is in contrast to a whitelist where only elements that are on a list can pass.
[41]In the eXtensible Access Control Markup Language, the *Policy Enforcement Point* protects a resource from unauthorized access: https://www.oasis-open.org/committees/download.php/2713/Brief_Introduction_to_XACML.html.
[42]https://developer.github.com/v3/#rate-limiting
[43]https://developer.github.com/v4/guides/resource-limitations/
[44]http://openweathermap.org/appid
[45]https://docs.quandl.com/docs#section-authentication
[46]https://dev.twitter.com/rest/public/rate-limiting
[47]https://developer.linkedin.com/docs/rest-api
[48]https://www.bfs.admin.ch/bfs/de/home/register/unternehmensregister/unternehmens-identifikationsnummer/uid-register/uid-schnittstellen.assetdetail.1760903.html
[49]https://docs.mulesoft.com/api-manager/rate-limiting-and-throttling-sla-based-policies
[50]https://letsencrypt.org/docs/rate-limits/
[51]https://github.com/sief/play-guard/blob/master/module/app/com/digitaltangible/tokenbucket/TokenBucketGroup.scala

4.4 Pattern: *Rate Plan*

a.k.a. Pricing Plan, Metering and Billing, Accounting

Context. An API is an asset of the organizations or individuals that have built it. From the viewpoint of commercial organizations, this means that it has both monetary and immaterial value. The development and operations of this asset has to be funded somehow. The API clients can be charged for API usage, but the API provider can also sell advertisements or find other means of raising funds.

Problem. How can the API provider meter API service consumption and charge for it?

Forces. When metering and billing, the following concerns are hard to resolve in a way that is acceptable both for API clients and providers:

- How can an API provider select a pricing model that balances its own economic interests with those of its customers and the competition?
- How accurately and fine-grained should API consumptions be metered to satisfy client's information needs without incurring in unnecessary performance penalties or availability issues?
- How can the security of the metering information be guaranteed but billing accuracy and auditability still be assured?

Non-solution. One could just invoice the client a flat sign-up fee, but this would treat hobbyists and high-volume corporate users equally; that is, it would be too cheap for one user segment and too expensive for the other one.

Solution. Assign a *Rate Plan* for the API usage to the API description that is used to bill API clients, advertisers, or other stakeholders accordingly. Define and monitor metrics for measuring API usage, such as API usage statistics on a per-operation level.

How it works. Several variants of *Rate Plans* exist. The most common ones are: *flat-rate subscription* and *usage-based pricing*. A *market-based allocation* is less often seen (also known as *auction-style allocation* of resources). All these plans can be combined with a *freemium model* where a certain low or hobbyist usage level is free, and payment only comes into effect for higher usages or once an initial trial period expires. Combinations of different plans are also possible, for example a monthly base *flat-rate subscription* fee for a base package and an extra *usage-based pricing* for additionally consumed services.

In *subscription-based* or *flat-rate pricing* the client is billed a recurring (e.g., monthly or yearly) fee that is independent of the actual usage of the service, sometimes in combination with a *Rate Limit* to ensure fair use. Within these boundaries, the subscription typically allows customers for near unlimited usage and requires less bookkeeping than *usage-based pricing*. Alternatively, a provider can offer different billing levels, from which a user can choose the one that best matches its expected usage. If a client exceeds its allowance, it can be given the option to either upgrade to a more expensive billing level or to have further calls blocked.

Table 2: Example of a *usage-based Rate Plan* of a fictitious storage provider with different billing levels. The files can be stored and accessed by an API.

Storage (up to)	Pricing per Month
First 5GB	Free
Next 95GB	$0.15 per GB
Over 100GB	$0.14 per GB

A *usage-based pricing* policy only bills the client for actual usage (e.g., API calls or amount of data transferred) of the service resources. The pricing can be varied for different API calls; for instance, a simple reading of a resource might cost less than creating a resource. This usage can then be billed periodically or be offered as prepaid packages (as sometimes done in mobile telephony contracts) with credits that are then spent (e.g. when using CloudConvert, a document-conversion SaaS, clients can purchase packages of conversion minutes that can then be spent over time).

Elastic *market-based pricing* is a third variant. For a market to emerge, the price of a resource may have to move in line with the demand for the service. A client then places a bid to use the service at a certain maximum price, and when the market price falls to or below the bid price, the client is allocated the service until the price rises above the bidding price again.

These variants of *Rate Plans* differ in the effort required to define and update the prices; they have an impact on attracting and retaining customers. They also differ in their ambitions to making sustainable profits. Finally, they may also differ in their scope: entire API endpoint vs. individual operations, API access vs. actual computing/retrieval (back-end service) are two such scoping dimensions. Client developers and application owners are advised to always read the fine print and run some trials to familiarize themselves with the billing granularity and operational procedures before committing to using a particular offering. Architectural and/or code-level refactoring [23] might be necessary to find an API usage profile that is both technically and financially satisfying.

Example. Imagine a fictitious provider offering binary large object storage as a cloud service, as explained in the *Blob Storage* pattern by [8]. The blob files can be stored and accessed via an API, but clients need to sign up with the provider first and obtain an *API Key*.

The provider decided to implement a *usage-based Rate Plan*, with a *freemium* level for low usage and different pricing levels depending on how much storage is used, as can be seen in Table 2.

A competitor of the provider, trying to differentiate itself and wanting to keep the monitoring at a minimum, might instead decide to go with a *flat-rate subscription* fee of $20 per month that offers unlimited storage space to clients.

Implementation hints. Product owners who decide to specify a *Rate Plan* can be advised to:

- Consider procuring a commercially-off-the-shelf billing system instead of building one yourself. Buying or renting rather than building a billing system is an executive-level make-or-buy decision due to its strategic and monetary impact.

- Have the metering and billing implementation of the *Rate Plan* audited.
- Adjust the prices regularly as required to stay competitive in the market. Do not forget to adjust the metering and performance monitoring accordingly, and update the API description and the corresponding *Service Level Agreement*.
- When the *Rate Plan* depends on the number of API calls made by the client, think about how to account for *Wish List* calls to implement integration scenarios that would otherwise require multiple single API calls.
- In many cases it is sufficient to guarantee that metering data becomes consistent *eventually*, as described in [8]; therefore, using a scalable NoSQL database that does support eventual consisency (rather than strict consistency) can reduce the performance penalty. A map-reduce job can be used to aggregate metering data periodically.
- For a community API, billing could be done per client organization, not on the level of individual API clients in the organizations.
- When implementing *usage-based pricing*, each API call needs to be logged in a non-repudiable manner. If an *API Gateway* [17] is used, it is a good place to implement metering. This could be as simple as counting the number of API calls. Alternatively, event sourcing and storing the whole request is also an option, which allows API providers to analyze the API usage per client and establish an audit trail. Clients might also want to track their usage on the client side to be able to validate the API provider's metering.
- If your application is run by a cloud provider, existing API Gateway offerings might provide this functionality already. For example, the Amazon API Gateway[52] "meters traffic to your APIs and lets you extract utilization data for each API key".

Consequences. A *Rate Plan* resolves most of the above forces. Security guarantees, however, have to be satisfied by the underlying implementation.

Resolution of forces.

+ By using a *Rate Plan* customers and the provider have a clear agreement of incurring costs and obligations.
– Writing and publishing sensible *Rate Plans* is hard and requires much knowledge about client's interests and business models on client and provider side.
– API clients need to be identified by an *API Key* or some other means of authentication mechanism.
– *Usage-based pricing* requires a detailed monitoring and measurement of a client's actions. To avoid disputes, the client will want detailed reporting to track and monitor their usage. This requires more effort on the provider's side. Limits can be put in place that trigger a notification when exceeded.

Further discussion. Another consideration is how to deal with outages of the metering functions of the *Rate Plan* implementation: if metering cannot be performed, it is impossible to later bill the client for its consumption. Consequently, the API has to be shut down until the metering system is available again or the service has to be provided for free during the outage.

Known Uses. *Rate Plans* are widely used by commercial offerings of all kinds:

- *Business Support Systems (BSS)* have been used in telecommunications for a long time; many mature billing and metering solutions exist in this industry. The TM Forum Applications Framework 3.0[53] covers these and other related business capabilities on a platform-independent level.
- A dynamic interface to a core banking backend [3] allows implementing *usage-based pricing* per operation type; a banking customer portfolio lookup may have a different price than an account balance check.
- AWS Lambda[54] implements *usage-based pricing* with a free-tier of one million requests per month and $0.20 for each further million requests.
- Amazon S3[55] is an example of *usage-based pricing* of storage capacity and operations performed on the storage. It also comes with several pricing tiers and volume discounts.
- CloudConvert[56] offers both prepaid packages of conversion-minutes as well as different monthly subscriptions.
- Amazon EC2 Spot Instances[57] use *market-based pricing*.

Related Patterns. A *Rate Plan* can use *Rate Limits* to enforce different billing levels. If used, the *Rate Plan* should refer to the *Service Level Agreement*.

To identify the client making a request, an *API Key* can be used (or another authentication practice).

API Gateways [17] and the systems management patterns in [12], especially *Wire Tap*, can be used to implement metering and can thus also be used as enforcement points. A *Wire Tap* can be inserted between the source and destination of a message to copy incoming messages to a secondary channel or a *Message Store* that is used to count the requests per client without having to implement this at the API endpoint.

4.5 Pattern: *Service Level Agreement*

a.k.a. Quality-of-Service Policies, Explicit and Structured Quality Goals

Context. An API contract has been defined for the API, including the functional interface specification (i.e., request and response messages with parameters) of the operations. The dynamic behavior of the API's operations when being invoked has not been articulated precisely yet in terms of its qualitative and quantitative Quality-of-Service (QoS) characteristics. Furthermore, the support of the service along its lifecycle has not been precisely articulated either (e.g., guaranteed lifetime and mean time to repair).

[52]https://aws.amazon.com/api-gateway/faqs/?nc1=f_ls

[53]https://fenix.tecnico.ulisboa.pt/downloadFile/3779580051402/TM_Forum_Applications_Framework_3-2.pdf

[54]https://aws.amazon.com/lambda/pricing/

[55]https://aws.amazon.com/s3/pricing/

[56]https://cloudconvert.com/pricing

[57]https://aws.amazon.com/ec2/spot/

Problem. How can an API client learn about the specific quality-of-service characteristics of an API and its operations? How can these characteristics, and the consequences of not meeting them, be defined and communicated in a measurable way?

Forces. Partially conflicting concerns make it hard to specify QoS characteristics in a way that is acceptable both for clients and providers. Specifically, the following questions have to be answered:

- How can a client decide whether a provider's offerings match the client's business needs from a business agility and vitality point of view?
- How can a client learn about a provider's compliance with government regulations, security and privacy measures and other legal obligations?
- How can an API provider communicate the attractiveness, availability and performance goals of its services to clients (assuming that more than one provider offers a certain functionality) without making unrealistic promises that may cause client dissatisfaction or even financial losses?
- How can a provider strike a balance between economizing its available resources and making a profit (or keep costs at a minimum – e.g., for open government offerings)?
- What is the right level of detail for QoS specifications, avoiding underspecification (which may lead to tension between clients and providers) and overspecification (which may cause a lot of effort in development, operations, and maintenance)?

Non-solution. The client could simply trust the provider to make commercially and technically reasonable efforts to provide a satisfying API usage experience, and in many public APIs as well as solution-internal APIs this is the only or the preferred option. However, if API usage is business critical for the client , the resulting risk might not be tolerable. One might rely on unstructured, free-form text that states the commercial and technical terms and conditions of API usage, and many public APIs provide such documents. However, such natural language documents are ambiguous and leave room for interpretations which might lead to misunderstandings and, in turn, to critical project situations. They might no longer be sufficient when competitive pressure increases. When not having any alternative or room for negotiating a customized agreement, deciding on using an API simply comes down to trusting the provider and/or predicting its future QoS characteristics based on historical data and previous experiences.

Solution. As an API product owner, define a structured *Service Level Agreement* (SLA) and write it up in an assertive, unambiguous way: the SLA must identify the specific API operation(s) that it pertains to and must contain at least one measurable *Service Level Objective* (SLO). An SLO must specify a measurable aspect of the API, such as performance, scalability, or availability.

How it works. In any *Service Level Agreement*, define SLOs as well as penalties, compensation credits or actions, and reporting procedures for violations of the SLA. As API client developer, study the SLA and its SLOs carefully before committing to use a given API. The SLA structure should be recognizable, ideally even standardized across offerings.

Derive the SLOs for each controlled service from specific and measurable Quality Attributes (QAs) that are relevant for the API and ideally have been specified during service analysis and design activities [4]. SLOs can also arise from regulatory guidelines; for example, personal data protection laws might mandate that data is erased once it is no longer needed. SLOs can be broadly grouped into different categories. For example, the European Commission's SLA guidelines [5] categorizes SLOs into those for *Performance*, *Security*, *Data Management* and *Personal Data Protection*.

In each SLO that coresponds to a particular quality attribute, specify a threshold value and the unit of measurement. Give a guarantee (minimum percentage) for how much of the time it will be met and a penalty in case it is not achieved. For example, an SLO might be "met for 99% of the requests" and have a "discount credit of 10%" as a penalty. It is important to clearly state how the measurement will be performed and interpreted to avoid confusion and unrealistic expectations.

When defining the SLA, involve all relevant internal and external stakeholders early (e.g., C-level executives, legal department, security officer). API providers should let the SLA specification be reviewed and approved by a well-defined set of these stakeholders (e.g., the legal department). Plan ahead as several iterations are typically required, which might be rather time consuming due to busy schedules; agreeing on SLA content and wording is a negotiation process.[58]

See the implementation hints section below for some advice regarding good vs. bad SLAs.

Example. Imagine a fictitious SaaS provider, offering a salary administration software including an API for a payroll service. The provider states that:

> "The payroll service has a response time of maximally 0.93 seconds."

The response time might need some clarification:

> "The response time is measured from the time the request arrives at the API endpoint until the response has been fully processed."

Note that this does not include the time it takes for the request and response to travel across the network from the provider's API endpoint to the client's endpoint. Furthermore, the provider assures:

> "The Payroll SLO will be met for 99% of the requests, otherwise the customer will receive a discount credit of 10% on the current billing period. To receive a credit the customer must submit a claim to our customer support center including the dates and times of the incident."

Implementation hints. As the API product owner or architect, consider the following when writing a *Service Level Agreement*:

- Find a balance between being attractive (or at least competitive) and, at the same time, being trustworthy and accountable in your SLA and SLO commitments. You may want to define (and design/test against) an *internal* SLA that is more aggressive than the communicated and agreed external one.

[58] In some enterprises, supporting functions have a reputation of being slow to respond; if responding at all eventually, some of them come across as "Dr. No".

- From the provider side, consider cloud deployment elasticity and scale out tactics if SLO targets are endangered [8]; from the consumer side, consider switching to a backup provider. On both provider and client sides, make sure to permanently monitor all relevant qualities. For the provider this is important to keep track of the state of your SLOs at any time (for improvement and accountability) and to detect where you might have problems; for the clients this is important in order to judge whether the provider has met the specified objectives or.
- Define a common or at least API-wide SLA template that serves as a guidance for providers and can easily be recognized by clients. SLOs are runtime or operational quality goals and, as such, should be specified in a SMART way (i.e., specific, measurable, agreed upon, realistic and timed[59]).
- Consider to complement the human-readable SLOs with machine-readable representations, e.g., by offering endpoints where the system's health can be observed, so that automated approaches such as elastic autoscaling or instant redeployment can be supported if there is a risk of an SLA violation.
- Consider following the European Commission's SLA guidelines [5] if your API will be used by European clients.

If an SLA is over-achieved over a longer period of time, consumers might come to expect the higher qualities and take them for granted. This might suggest to manage services/systems in such a way that the SLAs are met exactly, without exceeding them much. For availability SLAs, deliberately making the service/system unavailable temporarily is one less obvious option when following a "principle that availability shouldn't be much better than the SLO"[60].

According to a Dimension Data [6], an IT services company and IaaS cloud provider, an SLA should answer questions such as:

1. When does uptime and availability calculation start?
2. Do the API clients have to make any provisions to be covered by the SLA?
3. Does a severe performance degradation, e.g. very high latency or low throughput, count differently than a hard downtime?
4. Are penalties for SLA violations defined? Does the client get a refund or just a credit for future usage?
5. What steps does the client need to take to request such a credit? Who needs to prove that the outage happened?

As a provider, make sure these questions can be answered unambiguously.

Consequences. The main target audience for this pattern is the API product owner on the provider side rather than the developers of API operations. An SLA often is part of the API provider's terms and conditions (of services) or a master service agreement, along with other policies such as an "acceptable use policy" or a "privacy policy". The SLA can resolve all the previously introduced forces:

[59]See Service Level Agreement – Best Practices & Crucial Elements: https://www.userlike.com/en/blog/service-level-agreement-best-practices for a discussion of the SMART models and SLAs.

[60]https://cloudplatform.googleblog.com/2018/07/sre-fundamentals-slis-slas-and-slos.html

Resolution of forces.

+ Clients establish a shared understanding with the provider concerning service levels and quality levels that can be expected.
+ An SLA can target all services of a provider or just specific operations exposed at a particular endpoint. For example, SLOs relating to a personal data protection regulation will likely be handled in an overall SLA, but data management objectives – e.g. data backup frequency – might differ per endpoint and service operation.
+ Well-crafted SLAs with measurable SLOs are an indicator of service maturity and transparency and are able to resolve the outlined forces when implemented and monitored properly. Rather surprisingly, many public APIs and cloud offerings did not expose any, or only rather weak, SLAs (which can be attributed to market dynamics and lack of regulation) at the time of writing.

– The provider can be held accountable for failing to provide the service. Sometimes organizations do not want to be held accountable for their failures. Establishing clearly defined obligations like a *Service Level Agreement* might therefore hit internal organizational resistance.

Further discussion. It only makes sense to define SLAs and precise SLOs, and publish them to clients, if this is required (and paid for) by clients or is seen as beneficial from a business point of view. Otherwise SLAs are an unnecessary *business risk* as they are often legally binding and it might actually be hard to fulfill them at all times; if it is not needed, why offer such a strong guarantee? SLAs require substantial effort to be designed, implemented, and monitored; mitigating SLA violations also causes work. Maintaining operations personnel to quickly deal with SLA violations is also expensive. Business risks can be mitigated by limiting the liability of the API provider, e.g., to offer service credits as the only remedy for SLA violations.

Alternatives to an SLA with formally specified SLOs, as suggested in this pattern, are to define *no SLAs* or to set quality goals in more vague terms (i.e., *SLA with informally specified SLOs*). For instance, some security aspects are often defined in more vague terms, as they are hard to capture formally. Please note that formal and informal SLOs can also be combined in one SLA.

SLAs can be beneficial to the API provider even in the form of an *internal SLA* – yielding a variant of the pattern in which the API provider uses the SLA to specify and measure its own performance on relevant qualities, but does not share this information with clients external to the organization.

Known Uses. Many public APIs on the Web do not expose explicit SLAs, but ask their users to agree with their terms and conditions, which may cover related topics. Usually, no hard guarantees are given; the SLOs are only outlined and not specified formally. However, many public cloud providers have explicit SLAs, for instance Amazon Web Services (AWS) and Microsoft Azure. At the time of writing, SLAs are provided by these public cloud providers and offerings:

- Amazon EC2[61] commits to an SLO of a "Monthly Uptime Percentage" that is specified in terms of "minutes during the month in which Amazon EC2 [..] was in the state of Region Unavailable", which is further specified in the agreement.
- Microsoft Azure SLA for Functions[62] also defines a "Monthly Uptime Percentage" that is calculated as "Monthly Uptime % = (Maximum Available Minutes-Downtime)/(Maximum Available Minutes) x 100". The SLA goes on to further specify downtime and "Maximum Available Minutes". It limits the liability of Microsoft for downtimes to service credits as the only remedy for SLA violations.
- The combination of a precise uptime guarantee with service credits as the only compensation is a commonly used SLA variant. Two other examples using it are Singlewire[63] and Microsoft Dynamics CRM[64].
- Google Compute Engine[65] gives a similar uptime guarantee but makes it conditional on the client having its instances "hosted across two or more zones in the same region combined with the inability to launch replacement Instances in any zone in that region". Downtime is measured as "a period of one or more consecutive minutes of Downtime. Partial minutes or Intermittent Downtime for a period of less than one minute will not be counted towards any Downtime Periods."
- Optimizely defines in its Service Agreement[66] that it "agrees to maintain commercially reasonable technical and organizational measures designed to secure its systems from unauthorized disclosure and modification" and lists a few such measures explicitly like "storing Customer Data on servers located in a physically secured location" and "using firewalls, access controls, and similar security technology". SLA parts about security often stay at this level of an *SLA with informally specified SLOs*, as security is a quality that is hard to quantify.

Related Patterns. The details of *Rate Limits* and *Rate Plans* can be included in a *Service Level Agreement.*

5 SUMMARY AND OUTLOOK

In this paper, we described and presented the second set of patterns – consisting of five patterns – from a more complete pattern language envisioned to offer help and guidance for API designers and API product owners. The presented patterns focus on quality characteristics of an API and their negotiation and agreement: By applying relevant patterns, designers and product owners can a) strengthen desired quality attributes and b) communicate the quality properties to other stakeholders – first and foremost, their API clients, but also other stakeholders such as operations roles, risk managers, and auditors.

In the future, we consider to extend our pattern collection with further patterns that belong to other categories: for instance, additional structural representation patterns as well as patterns concerning the architectural roles and responsibilities of operations within an API are currently being mined, captured, and validated.

ACKNOWLEDGMENTS

We want to thank our shepherds, students and members of our professional networks who helped to investigate public Web APIs, donated candidate forces and patterns, and reviewed early drafts of pattern candidates and language structure, including Joseph Corneli, Nicolas Dipner, Antonio Gámez Díaz, Reto Fankhauser, Andrei Furda, Alex Gfeller, Gregor Hohpe, Hans-Peter Hoidn, Stefan Holtel, Ana Ivanchikj, Sebnem Kaslack, Michael Krisper, Padmalata Nistala, Philipp Oser, Andreas Sahlbach, Niels Seidel, Peter Sommerlad, Toni Suter, Dominic Ullmann and Robert Weiser.

REFERENCES

[1] Subbu Allamaraju. 2010. *RESTful Web Services Cookbook.* O'Reilly.
[2] Mario R Barbacci, Robert J Ellison, Anthony Lattanze, Judith Stafford, Charles B Weinstock, and William Wood. 2002. Quality attribute workshops. (2002).
[3] Michael Brandner, Michael Craes, Frank Oellermann, and Olaf Zimmermann. 2004. Web services-oriented architecture in production in the finance industry. *Informatik-Spektrum* 27, 2 (2004), 136–145. https://doi.org/10.1007/s00287-004-0380-2
[4] Humberto Cervantes and Rick Kazman. 2016. *Designing Software Architectures: A Practical Approach* (1st ed.). Addison-Wesley Professional.
[5] Service Level Agreements Subgroup Cloud Select Industry Group (C-SIG). 2014. Cloud Service Level Agreement Standardisation Guidelines. https://ec.europa.eu/digital-agenda/news-redirect/16934. (2014).
[6] Dimension Data. 2013. Comparing Public Cloud Service Level Agreements. https://www.dimensiondata.com/Global/Downloadable%20Documents/Comparing%20Public%20Cloud%20Service%20Level%20Agreements%20White%20Paper.pdf. (2013).
[7] Stephen Farrell. 2009. API Keys to the Kingdom. *IEEE Internet Computing* 5 (2009), 91–93.
[8] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications.* Springer.
[9] Martin Fowler. 2002. *Patterns of Enterprise Application Architecture.* Addison-Wesley.
[10] Antonio Gamez-Diaz, Pablo Fernandez, and Antonio Ruiz-Cortes. 2017. An Analysis of RESTful APIs Offerings in the Industry. In *Proc. of the 15th International Conference on Service-Oriented Computing (ICSOC 2017).* Springer, 589–604.
[11] Robert Hanmer. 2007. *Patterns for Fault Tolerant Software.* Wiley.
[12] Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Addison-Wesley.
[13] Jelena Mirkovic and Peter Reiher. 2004. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review* 34, 2 (2004), 39–53.
[14] Michael Nygard. 2018. *Release It! Design and Deploy Production-Ready Software* (2nd ed.). Pragmatic Bookshelf.
[15] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623
[16] Cesare Pautasso, Ana Ivanchikj, and Silvia Schreier. 2016. A Pattern Language for RESTful Conversations. In *Proceedings of the 21st European Conference on Pattern Languages of Programs (EuroPLoP).* Irsee, Germany.
[17] Chris Richardson. 2016. Microservice Architecture. http://microservices.io. (2016).
[18] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. 2006. *Security Patterns: Integrating security and systems engineering.* John Wiley & Sons.
[19] Prabath Siriwardena. 2014. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE.* Apress.
[20] Phil Sturgeon. 2016. *Build APIs you won't hate.* LeanPub, https://leanpub.com/build-apis-you-wont-hate.
[21] Chris Wood, Art Anthony, Arnaud Lauret, and Kristopher Sandoval. 2016. *The API Economy: Disruption and the Business of APIs.* Nordic APIs AB, Stockholm, Sweden. https://nordicapis.com/api-ebooks/the-api-economy/

[61] https://aws.amazon.com/ec2/sla/
[62] https://azure.microsoft.com/en-us/support/legal/sla/functions/v1_0/
[63] https://www.singlewire.com/SLA
[64] https://port.crm.dynamics.com/portal/static/1033/sla.htm
[65] https://cloud.google.com/compute/sla
[66] https://www.optimizely.com/terms/

[22] Olaf Zimmermann. 2009. *An architectural decision modeling framework for service-oriented architecture design.* Ph.D. Dissertation. University of Stuttgart, Germany. http://elib.uni-stuttgart.de/opus/volltexte/2010/5228/

[23] O. Zimmermann. 2015. Architectural Refactoring: A Task-Centric View on Software Evolution. *IEEE Software* 32, 2 (Mar.-Apr. 2015), 26–29. https://doi.org/10.1109/MS.2015.37

[24] Olaf Zimmermann. 2017. Microservices Tenets. *Comput. Sci.* 32, 3-4 (July 2017), 301–310. https://doi.org/10.1007/s00450-016-0337-0

[25] Olaf Zimmermann, Mirko Stocker, Daniel Lübke, and Uwe Zdun. 2017. Interface Representation Patterns: Crafting and Consuming Message-Based Remote APIs. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs (EuroPLoP '17)*. ACM, Article 27, 36 pages. https://doi.org/10.1145/3147704.3147734