TÉCNICO LISBOA

API Management Platform - Based on OutSystems

André Filipe Martins de Matos Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Manuel da Costa Alves Marques

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. José Manuel da Costa Alves Marques
Member of the Committee: Prof. Luís Manuel Antunes Veiga

May 2019

Abstract

In the recent years, application integration has shifted a lot. Disruptive technology is being developed at great pace and Application Program Interface (API) are, usually, in the middle serving as pillars. Either used internally, by partners or by the outside world, APIs are trendy and have been for quite a while now. However, and even though their ubiquity is undeniable, OutSystems does not have a way to expose the web services developed through the platform to neither usage while, on the other side, an API Management Platform (AMP) allows users to create, expose, monitor and monetize APIs. The goal of the project is then to take advantage of the Rapid Application Development (RAD) technology OutSystems provides on their platform and endow the capability to expose web services as APIs to its Integrated Development Environment (IDE) while also allowing to generate a Software Development Kit (SDK) from any Swagger Specification. The report begins by setting the project goals. Then, the APIs' world is introduced, concretely by presenting the APIs themselves, the AMPs' most wanted characteristics and the existing tools in the market. After, the implementation process is explained, namely by detailing the system components and the possible flows of interaction with the solution. Subsequently, the project results are shown and the conclusions are taken. The report finishes by enumerating the work to be done and a few suggestions of my own on how to improve the solution.

Keywords

API; RAD; AMP; OutSystems; Swagger; Apigee; Proxy; Policy.

Resumo

Nos últimos anos, a integração das aplicações mudou imenso. Tecnologia disruptiva está a ser desenvolvida com grande ritmo e as Interfaces de Programação de Aplicações (*APIs*) estão, geralmente, no meio a servir de pilares. Sejam usadas internamente, por parceiros ou pelo mundo exterior, as *APIs* estão na moda e já o estão há alguns anos. No entanto, embora a sua omnipresença seja inegável, a OutSystems não tem maneira de expor os serviços web desenvolvidos através da plataforma a nenhuma das utilizações ao passo que, por outro lado, várias Plataformas de Gestão de *APIs* (*AMP*) permitem que ao utilizador criar, expor, monitorizar e monetizar as *APIs*. Assim, o objetivo do projeto é aproveitar a tecnologia de Desenvolvimento Rápido de Aplicações (RAD) que a OutSystems fornece na sua plataforma e dotar ao seu Ambiente de Desenvolvimento Integrado (IDE) a capacidade de expor como *APIs* os serviços web, além de também permitir a geração de Kits de Desenvolvimento de Software (*SDK*) a partir de qualquer Especificação Swagger. No relatório são inicialmente definidas as metas a atingir com o projeto. De seguida é introduzido o mundo das *APIs*, concretamente explicando as próprias *APIs*, as características mais desejadas das *AMPs* e as ferramentas existentes no mercado. Depois, é explicado o processo de implementação, nomeadamente detalhando os componentes do sistema e os possíveis fluxos de interação dos utilizadores com a solução. Posteriormente, os resultados do projeto são mostrados bem como as conclusões. O relatório termina mencionando o trabalho a ser feito no futuro e são enumeradas algumas sugestões sobre como melhorar a solução.

Palavras Chave

Interface de Programação de Aplicações (API); Desenvolvimento Rápido de Aplicações (RAD); Plataforma de Gestão de APIs (AMP); *OutSystems*; *Swagger*; *Apigee*; *Proxy*; Política.

Contents

List of Figures

Listings

Acronyms

API	Application Program Interface
IP	Internet Protocol
UI	User Interface
URL	Uniform Resource Locator
AMP	API Management Platform
CD	Continuous Deployment
ESB	Enterprise Service Bus
DoS	Denial of Service
GUID	Global Unique Identifier
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
IoT	Internet of Things
IP	Internet Protocol
JSON	Javascript Object Notation
OAI	OpenAPI Initiative
OAS	OpenAPI Specification
PaaS	Platform as a Service
RAD	Rapid Application Development
REST	Representational State Transfer

SDK Software Development Kit

SLA Service Level Agreement

SOAP Simple Object Access Protocol

TTP Trusted Third Party

XML Extensible Markup Language

1

Introduction

Contents

Every day, more and more, people depend on technology, either to be used in their professional or personal life. In a way, it means there is a need for connectivity to be ubiquitous and the tendency is to continue spreading [1, 2].

While not long ago, data transformation, messaging delivery and routing were the most wanted Enterprise Service Bus (ESB) functionalities in enterprises, at the moment, solutions for the previous problems are already developed (even open-source) and the integration context shifted from the communication between firms' distinct software to the connectivity needed to bring the best of the several end-user products. In short, this means that modern applications, either mobile or web, depend on information which other applications provide in order to get their job done properly. Due to these advances, Cloud Computing and Big Data are everywhere, the Internet of Things (IoT) is becoming a trend, wearable technology is already a fashion and the mobile devices usage numbers is almost countless. As dependency generates demand, on the other side of things, people who produce technology need to deliver fast and efficient, yet robust, products.

The majority of the technology at people's hands, from the apps on their phones to the intelligent gadgets they interact with, are built using APIs as a basis [3–6]. This, in a way, implies that on the consumers' side they should be easy to understand, test, use and evolve, while for API developers it is extremely important to display them in the most attractive and easy way possible in order to motivate their consumption.

Furthermore, as we are already in the API era, exposed services still need to be integrated within systems whose communications are made through different protocols and, for that matter, have the need to be secure, robust, monetizable and, due to the market needs for fast deliverance, they also should be easy to evolve. All this combined, in addition to the popularity growth of the microservices architecture, leads to recognize the importance of AMPs. It might not seem immediately related but their existence is nowadays crucial for companies that want to expose their services in a transparent way and, above all, meet all the requirements mentioned before [7].

1.1 Problem statement

APIs are a hot topic nowadays. The industry giants — the so-called service providers — are making use of them to supply their customers with the best experience possible but also the medium, and even some small, sized companies are adopting the tendency more and more. Being APIs a core part of the business's strategy, there must be a way to expose them to whichever potential interested consumer in a way that they can easily be secured, managed, monitored and monetized. And in fact, there are actually a few good options (see section 2.3.2).

OutSystems, the world leader of the low-code development platforms, provides an IDE which allows

developers to use drag-and-drop functionalities, enhancing the development speed. Optimized for RAD, developers can consume and expose Representational State Transfer (REST) and Simple Object Access Protocol (SOAP) services in a few clicks within their applications. Furthermore, for every exposed REST service through the platform it is possible to generate the service documentation, making use of the Swagger tools.

Overall, an API developer using the OutSystems' tools will be able to deliver the product faster but at this stage, an easy and quick way to publish those services into an externally visible place is missing. A simple, yet optimal, way to deal with this gap would be if either the OutSystems' IDE or the service documentation page had a mechanism (e.g. a button) to directly expose a specific service as an API.

1.2 Objectives and contribution

Taking into account the needs of the API world, and the suggestion in section 1.1, the core goal of this project is to contribute to boosting both the APIs' exposure and consumption by designing, building and testing an AMP that equips the OutSystems' platform with an intuitive and attractive way to quickly expose services as APIs. For that to happen, the following functional requirements are expected to be fulfilled by the platform: (i) APIs exposed as reverse proxies; (ii) document APIs in a portal; (iii) test APIs directly in the platform; (iv) implement, at least, one policy of each type (i.e. security, mediation, service interaction and traffic management); (v) ability to download an API SDK; (vi) generate API usage analytics reports.
Regarding non-functional requirements, the solution must be: (i) intuitive; (ii) responsive; (iii) performative; (iv) maintainable.
The platform, which from now on I will refer to as ≪AMP≫, was developed using the OutSystems technology and consists, basically, of a web portal to which API providers and consumers can access (i.e. users can be providers and consumers at the same time as there is no actual distinction). The portal should list the publicly available APIs and, for each one, their methods, attributes and policies that can be applied. Even though it was initially aimed at the OutSystems developers, everyone who possesses a valid API definition in the Javascript Object Notation (JSON) format is able to expose APIs at the AMP.

1.3 Document structure

The structural organization of this thesis is as follows:

- Chapter 2 introduces the definition of API, while also focusing on the differences to the services. Moreover, the concept of API management platform is explained by tackling its added value. Lastly there is a reference to the platform current market leaders.

- Chapter 3 refers to the existing work and inspirations behind the project, namely Swagger, Apigee and OutSystems. There, I examined and detailed every bit of what was important to the conclusion of AMP, why and how.
- Chapter 4 is related to the architectural decisions I made, concretely to the design of the platform taking into account the requirements and restrictions imposed.
- Chapter 5 is where I provide a final evaluation of the platform by doing a proof of concept in regards to the established requirements and the developed policies.
- Chapter 6 content relates to my conclusions on the work I developed but also serves to present what could not be accomplished and a few other ideas for future development on the platform.

2

State of the Art

Contents

Comprehending the concepts, methodologies and technologies approached in this dissertation is of great importance as they are the beginning to the fundamental understanding of the addressed thematic. I will then start by introducing the world of APIs as I demystify the concept against the one of ≪service≫, with emphasis on both differences and similarities. Moving into the importance of APIs in our modern world, I will also tackle the value of the platforms responsible for their managing and identify the trends.

2.1 Application Programming Interface

Given the generic designation of Interface in the informatics world, APIs can be defined as a shared contract between two entities with the intention of formalizing a common way for them to talk with each other and understand what is being said [8]. So, since APIs are meant to communicate from system to system, it is correct to treat them as a set of protocols and definitions used to build software that, eventually, feeds on others applications information. In this context, I will relate to Web APIs [9], which are therefore a subset of APIs defined by a standard Hypertext Transfer Protocol (HTTP) request-reply message exchange that is usually formatted as JSON. As the request-reply process may be a simple one to understand, it might not be so easy to implement if the consumer is unaware about what kind of information the provider offers, how to request it and what to expect as a response. In short, APIs exist to make the implementation of such feature a not so complicated task for developers. For that to happen, APIs always come with a description of their capabilities, divided into three complementary parts, as explained in the following sections.

2.1.1 API Documentation

API Documentation is exactly what the name implies — the documentation of an API. It contains examples on how the developers can use its functions while being aware of the existing constraints. Being the responsible part for the API comprehension, it is divided in three layers [10]:

- **Top-level documentation** is the tier responsible for making possible the basic understanding of the basic functionality. It may encompass a set of simple code examples, test environments and/or even screenshots.
- **Functional understanding** is the tier with the goal of removing the abstraction from the rest of the documentation. It exists to make sure the average user of the API understands it without relating to technical terminology.
- **Technical understanding** is, in opposition to the functional understanding, one that represents the references from which developers can draw their applications (e.g. formatting examples).

2.1.2 API Definition

The API Definition is very much linked with the ≪Documentation≫, mostly because the first can be generated from the latter. Yet, the ≪API Definition≫ is meant to be understood by machines whilst the ≪Documentation≫ is to be human-consumable. ≪API Definition≫ can then be understood as the backbone and function of an API at a base-machine readable level.

2.1.2.A API Definition formats

Regarding formats to specify APIs, there exist two main choices [11]: (i) OpenAPI and (ii) RAML. OpenAPI [12] is available through Swagger framework [13], which provides a built-in editor for developers to document the services, a code-generator tool used both to generate API server and client code, and a way for the developers/end-consumers to visualize and test the API. In respect to its documentation, Swagger uses annotations to define the content (i.e. endpoint, operations, parameters, responses, and so on) allowing nesting different options, for example, to enumerate the different possible responses.

RAML [14, 15], in comparison to Swagger, gives a better balance between the technical aspects and human readability, which is also an important factor. At its core, RAML is a REST orientated documentation language, yet due to the way it is designed, it also supports less common protocols such as SOAP. Much like Swagger, RAML also requires data related to the API endpoint but allows to include reusable libraries that can contain data types, resource types, schemas, examples and so on. Based on the ease of writing and interpreting the ≪Definition≫, my choice would go for RAML. On the other hand, Swagger is heavily adopted and has a large community of consumers and contributors. Also, OutSystems uses it to consume and expose the platform REST APIs [16], and so does Apigee (section 3.2) to expose the APIs.

2.1.3 API Specification

The ≪API Specification≫ concerns about the API's general behavior as well as how it connects with the other APIs. For this reason it is most of the time confused with its ≪Documentation≫, however, they are not the same thing. While ≪API Documentation≫ describes with examples how the API is supposed to work (e.g. how to call their methods), the ≪API Specification≫ can be thought as the total explanation of the API, by combining elements of both the ≪Documentation≫ and the ≪Definition≫. The ≪Specification≫ allows developers to check what functions are available, how to call them, what they will do, what kind of parameters are mandatory to invoke them, what is the structure of the resulting reply, the type of objects that are received and so much more. To summarize, the ≪API Specification≫ defines the supported data types which lead to the main design structure and consequently to the functional and expected behavior.

2.2 APIs and services

People think, and say, the most varied things about APIs: what are they for and what are they not, what do they really do, what are they composed of, and so on, and so forth. But people also share the most interesting concepts about services and seem to often correlate these words. In a certain way, they are correct, but at the same time it is not strictly necessary for them to be linked.

APIs are, as a matter of fact, a mean to be an end, if by end we consider a service. However, services can simply be isolated. With the following real-world example [17] I intend to clarify what exactly are both:

- **Short story:** A person goes on a cruise trip.
- **Pre-conditions:** (i) Every passenger wears a bracelet which can be used for a number of activities, including as a digital wallet. (ii) The bartenders of the cruise's bar are robots and the menu is digital and available at multiple touch-panels where the orders are made.
- **Use-case:** (i) The person approaches the bracelet near the touch-panel sensor and authenticates, in order to pay for whatever is requested. (ii) The person browses the bar menu, order two different drinks, the robots prepare them and deliver.

  Figure A.1, present in appendix A, illustrates the sequence of interactions that happen from the moment the person finalizes the order until they receive the drink (i.e. use-case number two).

  Even though it is not represented, the digital menu can be understood as the ≪API documentation≫ which is there to help end-users interacting with the system.

  The touch-screen is the way to interact with the server (i.e. where the API is stored) and will be responsible to transmit the client request details into the API which, in turn, will trigger the adequate service. In the example, we see that two services are triggered in sequence: first the billing system and then the bartender.

  As for the responses, in the normal world APIs do not deliver physical objects such as the receipt or the drink but instead a formatted response which can be understood and treated by the machine responsible to show the demanded content to the final user.

2.3 API Management Platform

AMPs are the linking channel between providers and consumers. They can be seen as a bundle of tools, features and services set together to allow managing and exposing APIs to the world. Making use of their compelling characteristics [18–21] they respond to the needs of both user types.

2.3.1 The added value

In this section the most common and important AMP's features are presented and described:

Discoverability: it is the first AMP benefit that tends to come up. As providers want to make their APIs public for use, consumers need a place to find what they are looking for. The solution, then, is a common portal that impacts both user types.

Documentation: this is, once again, a fundamental step for both user groups. For providers, exposing APIs with structured and appealing documentation (perhaps even generated automatically from the definition) is a must. On the consumer side, good documentation is how they can be sure if the API is what they are looking for or not.

Monetization: as explained in section 2.2, an API is a mean to get a service done; and services are not, usually, for free. So, in the eventuality a consumer actually subscribes to/invokes a company API, it means they will need to pay for it. This is a process that differs a lot on how it is done [22] but overall the AMP role is to be able to identify who is making the API calls and inform the provider so that the billing can be done.

Analytics: by monitoring the APIs behaviour, providers can better understand how much their APIs are used and, more importantly, how are the customers using them. Typically AMPs provide fairly interesting dashboards which are helpful in understanding what is really happening with the API (e.g. demographic stats, popularity rankings, performance measures, and more). From there, for instance, providers can better understand the customer needs, make business decisions and take the necessary actions.

Security: this is a key aspect and besides "mandatory" is also one of the most flashy characteristics of such platforms. It is important to notice that providers want to expose APIs, but they want it to be a safe thing to do. The APIs server should not be vulnerable to the exterior, so apart from exposing them through a reverse proxy (i.e. making APIs location agnostic), AMPs typically provide a set of other security measures from which the provider can choose in order to keep their content as safe as it can be. Consumer authentication, request limitations and validations or IP filtering are some of the common key cases platforms make available for customers.

2.3.2 The current market

When one decides to include an AMP in the business and starts studying the market, the results can be very overwhelming due to the amount of offer that is available. Some are open source and some are not but in general they all provide the same basic criteria which were mentioned in section 2.3.1. Thus, as there is a lot to choose from it might not be that easy to take a decision. According to a study conducted by Forrester [23] which analyzed the ≪15 platforms that matter the most≫, there are a lot of strong contenders but there are only five which fit into the ≪leader≫ category: IBM API Management by IBM, Apigee by Google, WSO2 API Management by WSO2, webMethods API Cloud by Software AG and Rogue Wave Software by Akana. Figure 2.1, from Forrester's study report, shows the full list and where they stand in the market quoted:

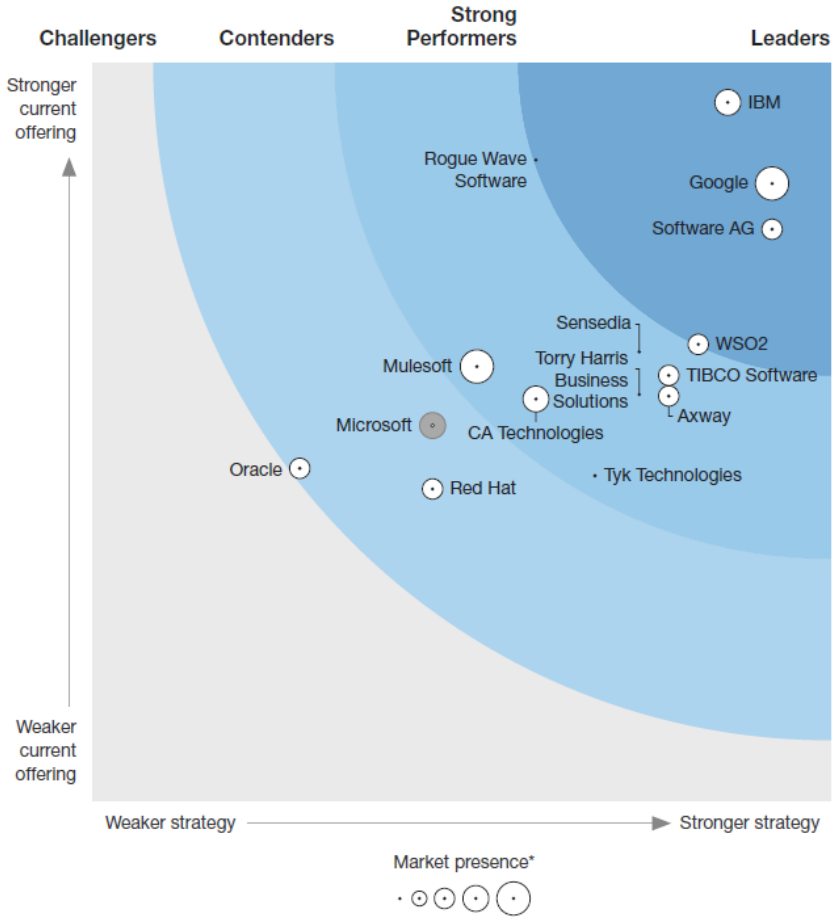


Figure 2.1: API management platforms value analysis

3

Related work

Contents

This chapter introduces tools, applications and contributions that have served as inspiration or helped in the design and development of the portal, namely (i) Swagger, (ii) Apigee and (iii) OutSystems.

3.1 Swagger

Swagger tools are one of the most important pieces of API development nowadays. In a way, it started in November 2015 when Swagger announced a partnership with OpenAPI Initiative (OAI) [24]. On the last day of the year, they donated Swagger Specification which changed its name and became known as OpenAPI Specification (OAS).

OAI is an agglomeration of industry expert companies [25] that envision the value of standardizing the way REST APIs were described, more specifically with a goal based on creating a standard for a language-agnostic interface directed to RESTfull APIs. This standard is intended to be a way for both computers and people to be able to analyze and understand a service without having to look neither into its source code nor to its documentation. Summing up, the core idea is that a consumer should be able to easily comprehend and interact with the service with low effort and only a tiny amount of implementation logic. Then, an OpenAPI definition document can be used to achieve different goals such as displaying the API by using documentation generation tools — interactive documentation —, generating code in several programming languages for both servers and clients, testing and more.

Despite the changing of name, the different existing tools remain known as Swagger due to a number of reasons, the most important being the recognition and contribution from the existent community at the time. Since the start of the partnership those tools evolved, got a huge boost, and are now the pillars beneath a great percentage of most used software in the world.

As cited by the OAI itself, ≪APIs form the connecting glue between modern applications. Nearly every application uses APIs to connect with corporate data sources, third party data services or other applications. Creating an open description format for API services that is vendor neutral, portable and open is critical to accelerating the vision of a truly connected world≫.

In the following sectins, I will introduce and briefly explain the different tools Swagger provides, referring to their purpose and importance.

3.1.1 Swagger Editor

This is the tool that allows writting APIs' specifications [26, 27]. Either designing from scratch or editing existing ones (it is possible to import from either a file or a Uniform Resource Locator (URL)), this editor provides a set of useful functionalities to the developer like automatically rendering the content of the specification and allowing to interact with it while still defining. It is available to download and install for any environment but can also be used in its cloud-based version.

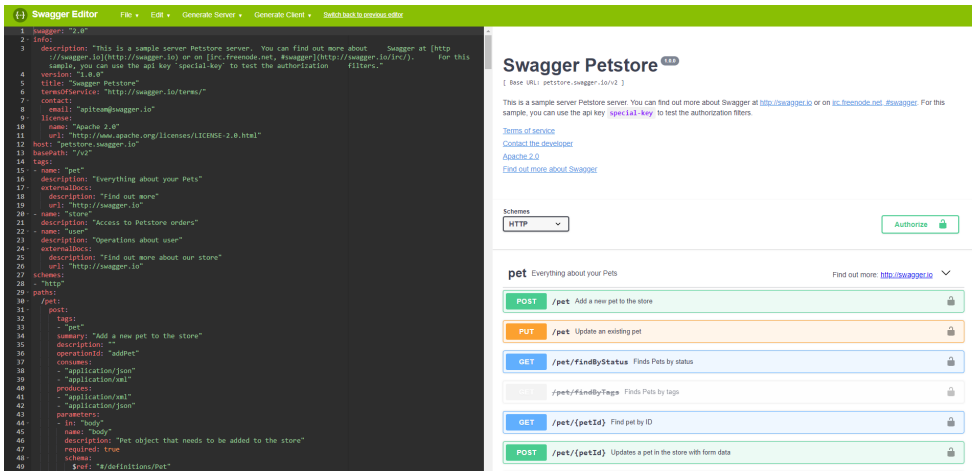


Figure 3.1: Swagger Editor screenshot - cloud version

3.1.2 Swagger UI

This tool, just like Swagger Editor, generates a visual representation of the API specification while also allowing to perform calls to said functions, verify inputs and outputs, documentation and more [28, 29]. It can be seen as part of the Swagger Editor functionality since the editing part is not available for this tool. However, this is intended for API consumers that might be interested in checking the API behavior without the need to have the logic implemented.

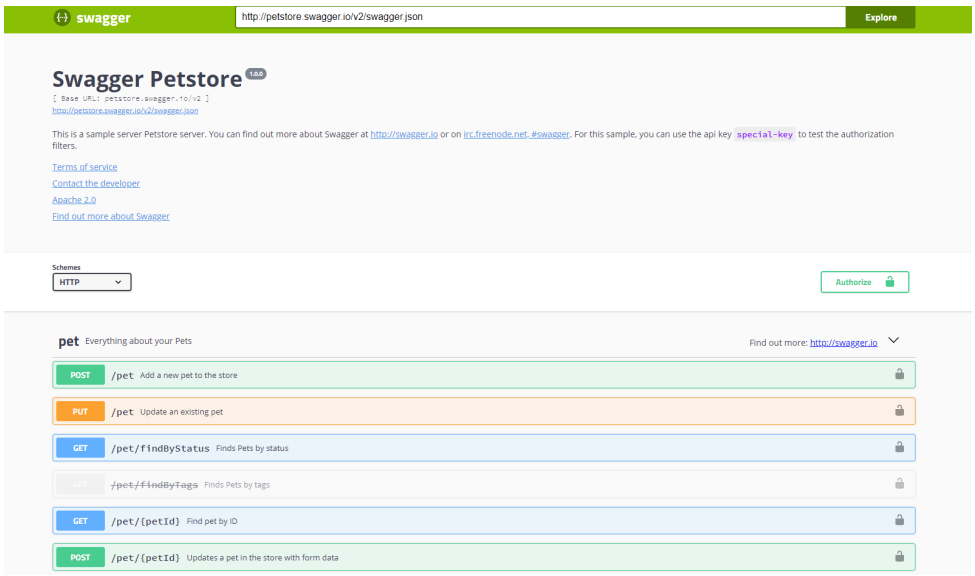


Figure 3.2: Swagger UI screenshot - cloud version

3.1.3 Swagger Codegen

This tool is able to generate server stubs and client libraries in most popular programming languages directly from an API specification [30–32]. As a result, Swagger Codegen allows to improve the API consumption numbers, and both ease and speed of developing an application on top of it.

As mentioned afterwards, in chapter 4, this tool is used within the AMP as the capability of generating code from the API specification was part of the goal of the project.

3.1.4 Swagger Inspector

This tool, once again cloud-based, allows to perform requests to web-services exposed using REST, GraphQL or SOAP [33, 34]. Despite allowing to invoke a specific endpoint of an API, Swagger Inspector offers the possibility of using its definition by making a call to the root URL. This will import all the available functions, making easy to select which one to test.

Swagger claims that this is the best way to create OAS from the existing APIs, and they actually facilitate it by storing the history of the functions' invocation. After, it is possible to select the ones desired from the list and generate new documentation with those chosen endpoints.

Swagger Inspector is the last released Swagger tool — it was not existing at the time the AMP development started — and it closed a big gap on API testing by making able to do it directly from any browser.

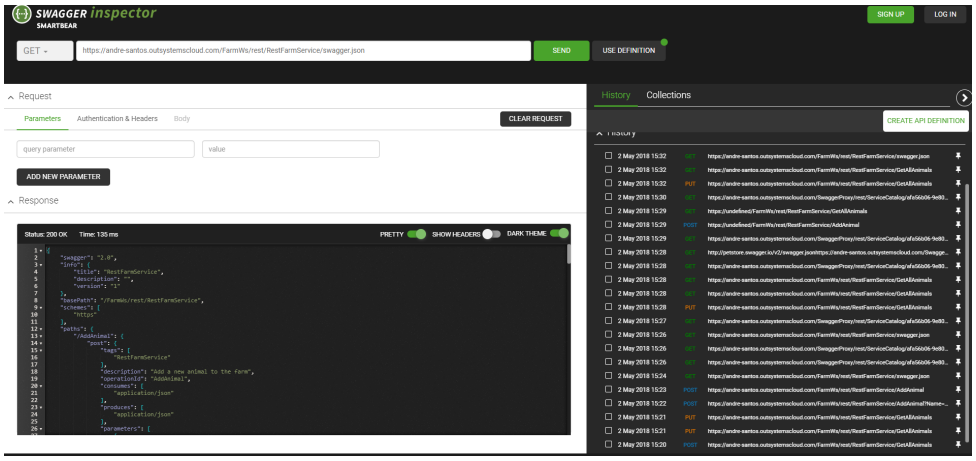


Figure 3.3: Swagger Inspector screenshot - cloud version

3.2 Apigee

Apigee, which is property of Google since 2016, provides the leading API platform (i.e. Apigee Edge) for enterprises and developers to design, secure, scale, analyze and monetize APIs. In short, when using the platform, users are able to manage the entire API life cycle and with Google being one of the co-founders of the Open API Initiative (see section 3.1), Apigee delivers a powerful platform completely integrated with Swagger tools.

The platform focus on two key aspects: people and technology. Their goal regarding people is to connect the application developer (consumer) with the API team (provider) and in order to do this, Apigee provides analytics and developer channel services. Regarding technology, their aim is to connect the APIs with the apps by providing gateway and app services.

With a focus mainly on app developers, the platform disposes of a developer portal that allows API providers to show their work to whomever is interested, in a clean and appealing way. On the other side, these APIs are exposed through gateway services responsible to ensure safety and mediation between the applications and the backend servers. In fact, the exposure of the API through the proxy allows it to be customized with different available policies [35] which, according to Apigee itself, enable to ”augment your API with sophisticated features to control traffic, enhance performance, enforce security, and increase the utility of your APIs, without requiring you to write any code or to modify any backend services”. These same proxies are also monitored allowing both API consumers and providers to evaluate the API behaviors depending on the circumstances — e.g. as it might be useful for an app developer to monitor both API and app behavior (developer channel services), for an API provider what matters is the business information that is extracted from the API usage (analytics services).

The strengths of the platform noted in the previous paragraph match, in fact, the functional requirements set in section 1.2, namely: (i) APIs exposed in gateways services (i.e. proxies); (ii) developer portal where providers show their work (i.e. documentation and testing); (iii) proxy customization (i.e. policies); (iv) proxy monitoring and API behavior evaluation (i.e. analytics reports). As a matter of fact, the only AMP functional requirement that Apigee does not fulfill is the possibility to generate an SDK for any given API.

Given Apigee’s success and being considered one of the industry leaders, it served as inspiration for the design and development of AMP, especially in regard to the proxy and policy usage.

3.3 OutSystems

OutSystems, a Portuguese company born in 2001, is the main contributor to the development of this project. As implied in the title, the AMP was developed using the OutSystems tools and technology. The company provides a cloud solution for application development in the form of Platform as a Service (PaaS) which is, at the moment, considered the number one low-code platform for RAD, in both web and mobile worlds [36–39].

3.3.1 Service Studio

Service Studio, their IDE, allows developers to build complex applications either to mobile or web platforms by leading their focus into four main categories, which are presented in different tabs:

- **Data** tab is dedicated to the creation and management of entities, structures, session variables and site properties. *Entity* is the keyword OutSystems uses to refer to databases, which can be created on the platform or imported from a .xls filetype. Entities store basic data types (e.g. boolean). Structures are similar to Entities but exist only in memory. They are normally used to store temporary data (e.g. the result of a Web Service invocation) and can store any combination of data types, including other structures. Session variables and site properties provide similar functionality with the difference being that the first is used to save data related to a user during the existing session (e.g. session ID) while the latest contain information that can be shared between every user (e.g. variable to store the minimum password size).

- **Logic** tab is dedicated to the creation and management of server actions, integrations and roles. Server actions are what its name states: actions that run logic in the server side (e.g. LogMessage and CommitTransaction). Integrations allow the user to consume and expose web services through SOAP and REST protocols, but also to consume remote functions of SAP systems. To note that for the case of exposing a REST web service, OutSystems can generate a visual representation — similar to what Swagger User Interface (UI) presents — but does not allow to test it directly nor to download code in any programming language. Lastly, roles are meant to set users access permissions to pages and/or functionalities in the application.

- **Interface** tab is dedicated to the UI flows, the images and the themes. Regarding the flows, it is possible to define which are the existing screens, their appearance and how they connect with each other. Themes can be loaded into the application, or applied to certain parts of it. Also, before each screen load, a special action (referred to as preparation) is executed in order to apply some logic (e.g. access permissions checks) and to fetch every needed information from the database.

- **Processes** tab is responsible to define the workflow behind an application. Typically this is used when a user needs to interact with other (or even itself), but depend on other tasks (i.e. asynchronism). In short, it is used to define processes and timers.

In the end, OutSystems uses a Continuous Deployment (CD) model, allowing the developer to use their ≪1-Click≫ button to immediately publish the application [40].

3.3.2 Integration Studio

Being Service Studio an IDE that allows developers to, easily and intuitively, build applications without requiring them to know how to code, OutSystems also developed the Integration Studio: an IDE meant for proficient software developers to write and deploy custom extensions, which can be actions or entities. An action is a function that a software developer can code for JAVA, .NET or both platforms. An entity extension is basically meant to be a connection to an external database. Such extensions can after be integrated into the application through the integration folder in the logic tab of Service Studio.

3.3.3 Forge

OutSystems Forge [41] is a community repository to which developers can contribute with open-source projects. These projects can be full applications but are often components, connectors, widgets or themes. The aim is to reuse these parts and speed up the development process.

To develop AMP I browsed Forge for some extensions and in the end I used seven of them:

Swagger Parser [42] is an extension that receives a URL as input (ideally a Swagger specification/ OAS v2.0) and fetches its content. Afterwards parses it and stores some of the information in the database. It also comes with a simple interface where the APIs documentation is shown. For AMP needs, I had to retrieve a few more important fields, so I extended the code using Service Studio 3.3.1 even though I did not use the provided interface.

JSON Pretty Format [43] is a module containing a web block (reusable screen) that takes a JSON text input and applies the correspondent syntax highlighting.

ardoHTTP [44] is an extension which comes with helper functions to perform the major HTTP request types (GET, POST, PUT, DELETE). It allows to define a few characteristics of the request, such as the header or the timeout.

ardoJSON [45] is another extension that comes with a set of helper functions, in this case regarding JSON serialization and deserialization.

Back to top [46] provides a small icon that shows up when the page's top part is no longer visible allowing the user to smoothly scroll back to the top with one simple click.

HTTPRequestHandler [47] comes with fairly important functionalities related to the web such as retrieving the Internet Protocol (IP) of the machine which performs a request; retrieving URLs, URL methods, URL request and more. For instance, it was this extension that made possible for me to create a session cookie for the user, as well as to read it.

PlatformPasswordUtils [48] was used to generate and validate passwords for the AMP users, specifically using the SHA512 hash algorithm.

4

Architecture of the solution

Contents

In the context of this chapter, I specify the design requirements and elaborate on the reasoning behind the choices I made, in regard to the possibilities I had and considered. Also, I refer to the system components and detail both providers, consumers and final users action flows of interaction with the system.

4.1 Architecture design requirements

Taking into account the lack of possibilities to do the same with web services exposed through the OutSystems platform as the ones described in sections 3.1 and 3.2, the main goal of the project is to equip the OutSystems platform of a way to, quickly and easily, expose web services as APIs. Furthermore, it was also intended that users can check on the APIs' behaviour without leaving the platform and that the APIs' definitions could be generated into different programming languages and downloaded to be used by the consumers.

4.1.1 Requirements

With the main requirement to deliver this project being to use the OutSystems platform and technologies, the architectural decisions (i.e. functional requirements) were then made considering the existing related work mentioned in chapter 3 and the different desired users flows:

1. Expose APIs as reverse proxies;
2. Document APIs in a portal;
3. Test APIs directly in the platform;
4. Implement, at least, one policy of the following types: (i) security, (ii) mediation, (iii) service interaction and (iv) traffic management
5. Ability to download an API SDK;
6. Generate API usage analytics report;

Moreover, in regards to the non-functional requirements, the solution must also be (i) intuitive; (ii) responsive; (iii) performative; (iv) maintainable.

4.1.2 Architecture

At Apigee, each exposed API has a dedicated reverse proxy, and requirement 1 follows that idea. The proxy itself immediately brings an important security feature: service agnosticism. By calling the proxy

address and not the API's, the providers can feel more safe as their servers cannot be targeted directly, at least from the "API exposure" point of view. Furthermore, the proxy endows a decoupling capability (i.e. life cycle management), making possible for providers to alter the service behaviour without needing to expose it again. Apart from that, and since proxies are a sort of man-in-the-middle, certain security, traffic management, service interaction and data mediation features can be applied, and differ from API to API.

As for requirement 2, due to the idea of allowing two types of user roles at AMP (i.e. provider and consumer), initially, the plan was to create two separate portals. This, in practice, allowed to create two isolated applications that would consume different resources and could be managed independently. However, since the roles have shared privileges, the final decision was to merge the functionalities and build only one. The portal itself is supposed to allow consumers to browse the APIs list and, for each one, to study the applied policies and the respective methods. For each method, consumers must be able to inspect the input and output parameters, their types and examples; the possible responses after a call; the request type to be done for the method; the request URL.

Taking into account requirement 3 and considering APIs might also be developed for internal usage and not only to the outside world, I considered that providers are also consumers — i.e. consumers share their access rights with providers, even though providers have a few exclusive ones. For this matter, both roles are able to test every existing API exposed through the platform. Testing an API on the portal is very similar to an actual API invocation. In short, the call to the proxy is done by the platform instead of the final user application. Optionally, consumers can trigger any available policies they choose.

For requirement 4, once again the inspiration came from Apigee. In order to endow the platform of the same basic features Apigee and other AMPs have, a set of policies must be implemented. With that in mind, the following, sorted by group, were chosen due to being core functionalities of their respective groups: API key and OAuth2 on the security group; JSON to Extensible Markup Language (XML) and XML to JSON on the data mediation group; message logging as part of the service interaction group. Even though traffic management policies were initially meant to be implemented, I decided to postpone the idea to future developments on the AMP due to its complexity. All of these are going to be detailed further ahead in section 4.1.2.B.

Requirement 5 is something that does not exist in Apigee, but is possible to be done with the help of Swagger. As mentioned in section 3.1.3, Swagger makes possible for users to download an SDK of any API definition they provide, and this should be the case with AMP. Users must be able to ask AMP (i.e. which in turn requests to Swagger Codegen) for the SDK of the API they are currently browsing and get a .zip file with the corresponding content.

Requirement 6 targets a tool which is as a must for any AMP. Typically both user types have access

to graphic analysis concerning the APIs, even if the content is different for each. In this case, as a proof of concept, it is only expected for providers to have usage information about the APIs they own.

In the end, given the requirements and the solution I came up with, the following enumeration describes each component that is part of the platform. Fig. B.1 present in appendix B supports the idea, giving a visual representation of the different user flows and interactions, later described in section 4.2.

4.1.2.A AMP components

Database: A more complex database was idealized in the beginning but only a small portion of the information ended up being used (i.e. the most relevant). For that matter, the database stores information related to the APIs themselves but also regarding the proxies, the users, the Trusted Third Party (TTP)s and the proxy logs. Fig. C.1 present in appendix C shows the used database schema.

Codegen API: This is, as a matter of fact, an API call to the Swagger Codegen tool. It allows any user to request server stubs or client SDKs in a bunch of programming languages as shown in Fig. 4.1.

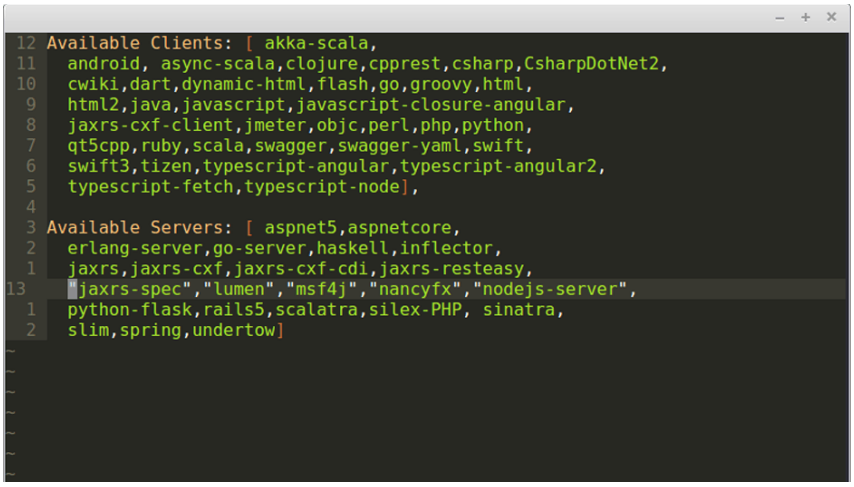


Figure 4.1: Available languages at Swagger Codegen tool

Proxy: The main purpose of a proxy is to act as an intermediary between two systems. An API proxy is one that belongs to a subset of proxy and is, consequently, intended to deal with the communication between an API and its consumer [8].

Its use brings several advantages to providers, especially in terms of security and life cycle management of their services. As a matter of fact, the proxy allows to integrate several modules that can provide security, mediation and traffic management features, yet regarding security, besides the capabilities for access control and authentication checking, one important aspect is that service location agnosticism is provided, meaning the API real address is masked. In respect to the

life-cycle management, this type of proxy contributes to the decoupling between applications and back-end services. This implies that despite changes that are made to said services, applications are shielded as they will continue to call the same API without any interruption.

This proxy solution, inspired on Apigee, has two different endpoints: one responsible to receive the initial requests from the applications and deal with them (i.e. by applying whichever policies needed (refer to section 4.1.2.B) and another responsible for the redirection of the request to the original API address.

API portal: The initial idea was to have two different portals: one meant to be used to publish APIs — provider role — and the other to search, test and consume APIs — consumer role. Ultimately, due to the decision to create only a shared one, there is no more role separation as any registered user can be both at the same time. This means that a user is not only able to publish a new API but also to subscribe to any of the existing and download their corresponding stubs.

The portal welcomes the users to the page where all the APIs are listed. From there, and even without registering, they are allowed to examine any public API specification and test it directly on the platform. Those are however the only rights of a ”guest” user. The other available functionalities are for registered users only and are comprised of uploading new API specifications; managing the owned APIs policies and visibility; subscribing to APIs and downloading their corresponding SDKs.

On its core, the platform is divided into two main parts: (i) the navigation menu, on the left-hand side and (ii) the main container where the information is displayed, on the right; and is composed of the following pages:

- **New API:** It is actually a popup. To expose a new API the user needs to introduce its name, a brief summary, the URL corresponding to the swagger specification and finally, from a set of policies that is displayed, select the ones, if any, to be applied. When hovering over the policy name, an information panel will crop out describing what it does.
- **All APIs:** As previously mentioned, this is the entry page. In here, a table is displayed with all the existing APIs and for each one the following information is visible: name, owner, description, applied policies and existing endpoints — with the last two being shown by hovering over an information icon.
- **My APIs:** Similar to the previous, this page also shows a table of APIs. However, only the ones published by the signed-in user (provider concept) are displayed. With a few extra tweaks, in here it is possible for the provider to manage the policies that are applied to the API, change its visibility status and delete it. Deletion is only allowed for APIs that are not being consumed/subscribed to.

- **API specification:** This page is dynamic and relative to each API. In the page, the API endpoints are listed along with the corresponding request types and, for each one, the input and output criteria (with data types and examples), the response messages and the request URL.

  Each method of the API can also be tested, and users who are subscribed or own the API have the option to download the API SDK. If a user is not subscribed yet, he can do it so by clicking the ≪subscribe≫ link on the bottom right of the page. Whenever a subscription occurs, a Global Unique Identifier (GUID) is generated and can be retrieved in the correspondent page. This is described as the API key, and must be passed in the header of a request in order to verify if that specific user is allowed to invoke that specific API.

  When testing, and depending on the active policies, the users might have the option to provide the request data in JSON, XML or plain text and the same happens for the reply.

- **Subscriptions:** This is a page used to track and show the keys correspondent to every API the user subscribed. It includes a shortcut to allow copying the value immediately.

- **Policies management:** This is a hidden page accessible only to the administrator of the portal. Here, the existing policies can be removed (as long as they are not being used by any API) and new ones can be added. An updated list of policies will be immediately available for new API publications and policy management. As for new policies, they should be coded and working, as well as integrated with the existing code before being created here.

4.1.2.B API policies

Policies are individual modules — as shown in Fig. 4.2 — which can, optionally, be integrated on each API proxy. These modules will have distinct functions/goals that will depend on the API's preferences and needs. Moreover, as referred in section 1.2, the policies are categorized within security, mediation or service interaction groups. For this project, I applied at least one of each group, attending the objective was to show how doable it is.

- **API key** is part of the security policies and it works as the identifier of a one-to-one relationship between a user and an API. It is, in fact, an aggregation of two unique keys which are generated from the combination of alphanumeric characters and separated by the dash sign. Each unique key is itself composed by one group of eight chars followed by three groups of four and one final group of twelve.

- **XML to JSON:** is part of the mediation group and allows to transform messages from the XML format to JSON. This policy might be useful, for instance, if the exposed API consumes a JSON

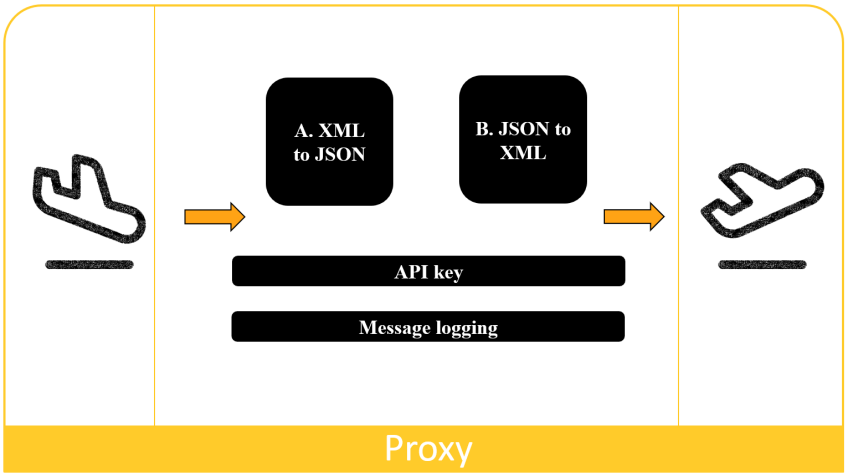


Figure 4.2: Policy modules inside of an API proxy

format but the provider trusts AMP to accept and transform it from XML into the expected/accepted format.

- **JSON to XML:** also belongs to the mediation group. It allows to transform messages from the JSON format to XML. In case the API response is of JSON type but the provider wants to allow consumers to receive responses in XML (always optional), he can activate this policy. This might be helpful for the consumer side if the application under development only understands XML.

- **Message logging** belongs to the service interaction group. It is a useful feature which is triggered every time (not optional) an action occurs inside of the proxy. In short, it allows to keep record of every request and response that goes through a proxy ensuring a way to track down possible problems during runtime. Furthermore, the proxy logs make possible for the owners to see a graphical and statistical analysis of their APIs traffic of the previous five days. With the logs, providers are able to download a detailed excel file whose content corresponds to the following information:

  1. ≪Date≫, with the format YYYY-MM-DD, following ISO 8601 [49];
  2. ≪Time of day≫, with the format hh:mm:ss, also following ISO 8601;
  3. ≪IsTest≫ is a boolean which indicates if the request came from an actual application or from someone testing the API from inside the portal;
  4. ≪Requester email≫ if the proxy was triggered from an actual application. In case of a test, the value will be empty;
  5. ≪Flow type≫, corresponding to the first 3 letters, in capitals, of request or response — i.e. REQ/RES;

6. ≪Method name≫ refers to the method that was triggered;
7. ≪Request type≫, identified by the first 3 letters, in capitals, of the action — i.e. GET/POS /PUT/DEL;
8. ≪Data≫, assuming the only data types will be plain text, JSON and XML, all the spaces are trimmed.

Moreover, apart from the proxy policies, the portal makes available the option to use **OAuth2** as an authentication mechanism. This policy fits in the security group and allows users to authorize an application to take action for them, without providing their sign-in credentials. In practice, it allows identity providers to issue tokens directly to third-party applications, with the approval of the user, which will then have rights to access the resource server on their behalf.

Initially thought to be a proxy policy with the goal of authenticating whoever is calling the gateway, I ended up enforcing the API testing and subscribing to registered users only. Thus, this policy can be triggered whenever a user registers or signs himself in the platform.

4.2 Users perspectives

This section describes, in detail, the different users' flows of interaction with, namely of the provider when exposing an API, the consumer when browsing, testing and downloading the different APIs and the final user when invoking them through an application.

4.2.1 Provider flow - Publish

This is the flow describing how a user exposes an API in AMP. The process starts when the provider accesses the portal and opens the ≪New API≫ popup. Publishing an API is only available for signed-in users, so they must be authenticated as a pre-condition. When the popup is shown, the provider should (i) name the API, (ii) write a brief description, (iii) paste the corresponding URL, (iv) choose, from a list, the available policies to be applied and finally (v) choose if the API will be immediately visible to the public. Then, AMP will fetch the API definition from the provided URL, parse it and store the API information in the database. Lastly, AMP builds a proxy for the API and redirects the user to its specification page. That API is, as from that moment, accessible to any consumer that is browsing the portal and can be used from that point onwards, as long as its visibility is not set to be private.

Fig. D.1, in appendix D, is a sequence diagram that backs up the flow described above.

4.2.2 Consumer flow - Search, test & download

This is the flow describing how a user searches for information about an API, tests it and eventually downloads its SDK. Fig. D.2, in appendix D, relates to the flow description.

At first, the consumer must access the portal and open the ≪All APIs≫ page. All the publicly available APIs are listed and, for each one, the user is able to see its name and summary, the provider's name and email, and its methods and available policies. Users also have a search tool which they can use to search by the name and summary of the API they wish to find.

By clicking in the API name, the user will be redirected to its ≪specification≫ page where all the API's information is presented. In there, users can look into the API definition (i.e. a URL on the top of the page will open a new tab with the content, if clicked) and for each API method the input and output parameters (with examples), the possible responses and the request URL.

Also, the page contains a ≪Try out≫ button which enable users to test the API methods right from the portal if they choose to do so. If they do, an HTTP request is sent to the API proxy which in turn will map to the correct API address, apply any policies that may be needed and forward the request to the API server. After receiving the response, more policies might need to be applied and then the final result is shown on the page.

Apart from testing, there is also the possibility to download the API SDK, yet, only signed-in and subscribed users are allowed to do this action — API owners are automatically subscribed. When the ≪Generate library≫ button is clicked, a popup is shown prompting the user to select the desired type (i.e. client or server) and the programming language. After choosing and confirming, AMP requests Swagger Codegen for the corresponding stub and then downloads it into the user's device.

4.2.3 Final user flow - Invoke

This is the flow describing how an API invocation is done on AMP. For the final user, this process is a black-box as they are triggering APIs by interacting with an application.
That said, when the proxy first receives the request from the app, it tries to find an API key in the request header. If it finds it and is a valid one, the request is mapped to the correct URL and forwarded to the API server. After receiving the response, the proxy sends it back to the app so the results are displayed to the user. Just like the "test API" situation, if any policies are required to be executed in the proxy, they will after the request is received or before the response is sent back. If the API key verification is unsuccessful (i.e. not found or not valid), the following response will be returned: "Invalid rights. Please contact the administrator". The flow can be related to Fig. D.3, in appendix D.

5

Validation of the implementation

Contents

This chapter describes an overall representation of the work I developed, more specifically the concept results, in images, and respective explanations in regards to the proposed goals in 1.2.

5.1 Requirements' proof of concept

- For requirement "APIs exposed as reverse proxies", as a matter of fact, I exposed an API which accepts GET, POST, DELETE and PUT requests. The requests receive a GUID as a parameter which is then used to identify the correct API. For GET and DELETE requests, it also expects an array if any values are needed to be passed along to the true API. When the request is received, the correct URL is fetched from the database and the attributes are mapped. It is at this time that the policies are applied. Then, the final URL is built and the API server is invoked. Fig. 5.1 mimics the functioning of the proxy.

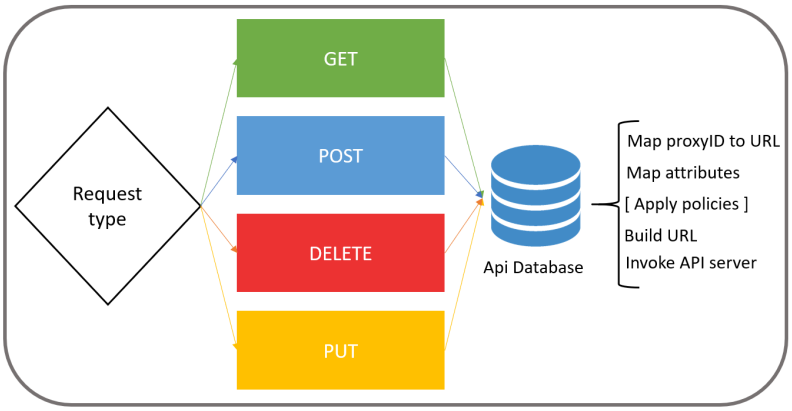


Figure 5.1: Inside of the proxy

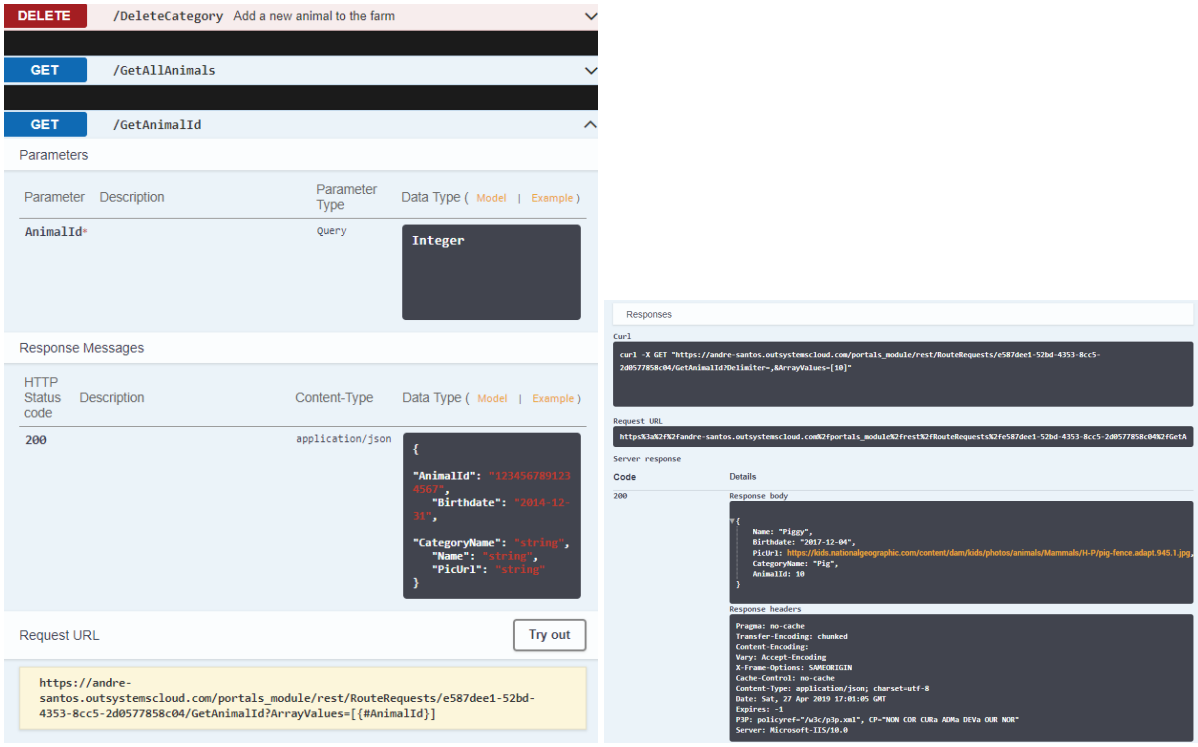
- For requirement "Document APIs in the portal" to be possible, all the APIs' content is parsed and stored in the database (Fig. C.1) at publication time. Then, the ≪API specification≫ page is made available to the users. A preview of the page can be seen in Fig. 5.2(a).

- The "Test APIs directly in the platform" requirement is possible due to the combination of the ≪API specification≫ page and the proxy. The test mode is activated from inside the page and when triggered, the request is sent to the proxy, processed and executed according to the flow which was previously introduced. Fig. 5.2(b) shows the result of a test result being shown to the user, in the platform.

- The "Ability to download API SDK" requirement is possible thanks to the Swagger Codegen tool. Fig. 5.3 shows the popup displayed to the user for him to choose to which framework the SDK is needed. When chosen, the correct SDK will be downloaded into the user's device.

- The last requirement "Generate API usage analytics report" is possible due to the "Message logging" policy. For every message, either request or response, that goes by the proxy, a new entry is made in the database with the content detailed in section 4.1.2.B. From there, AMP is able to render, and show

to the proxy owners, graphical information concerning their proxy usage. In addition to the charts, the provider can also request access to a file with the same information.



(a) API method examination. **(b)** Test results on the platform.

Figure 5.2: API method examination & testing.

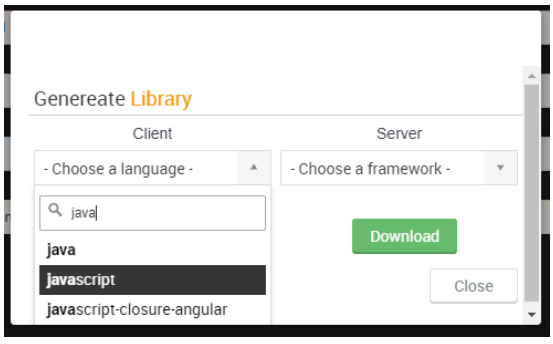
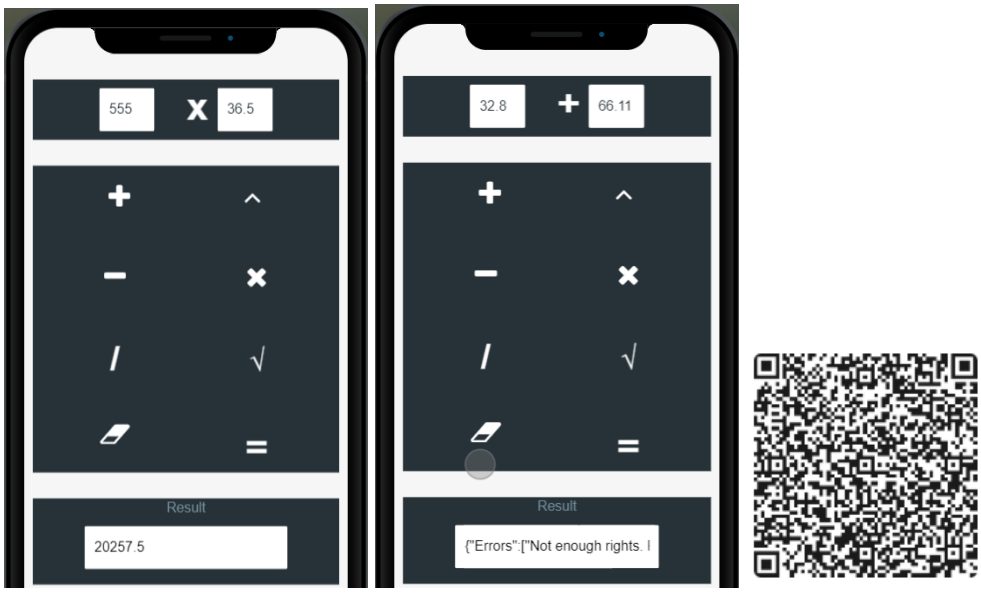


Figure 5.3: Popup showing possible API's SDKs.

As for the policies, I ended up not implementing any regarding "traffic management", yet I included two on the "mediation", two on the "security" and one of the "service interaction" groups.

5.2 Policies' proof of concept

API key: The API proxies work like locked doors. If the users have the right keys, they are able to unlock them and enter whilst if they do not, the access is denied. So, concerning the ≪API key≫ there are two takes to retain: (i) The platform has a built-in API key which is used to grant access to whomever is testing at platform level and (ii) each subscribed user will have its own key to use within their applications. As the first point is already proven in the previous section, in order to prove the second one I built a simple mobile application around one of the sample APIs I created: a calculator. Figures 5.4(a) and 5.4(b) show different interactions with the application. On the first, the provided API key is a good one so the obtained result matches the expected but on the second, as the API key is wrong, an error response is received as it can be seen in listing 5.1. The mobile application can be accessed and tested by using the QR code available in Fig. 5.4(c).



(a) API call with good API key.

(b) API call with bad API key.

(c) QR code allowing to access the calculator mobile app.

Figure 5.4: Policy proof of concept figures - API key

Listing 5.1: No rights response message.

```
1          {"Errors":["Not enough rights. Please contact the
               administrator"],"StatusCode":500}
```

Message logging: In the ≪My APIs≫ page there is the ≪Usage Report≫ button. When clicked the platform displays a popup showing two graphs: (i) column chart which shows how many API calls happened, per day, during the past five days and (ii) a circular chart detailing the frequency of the invoked methods. An example of this popup can be seen in Fig. 5.5.

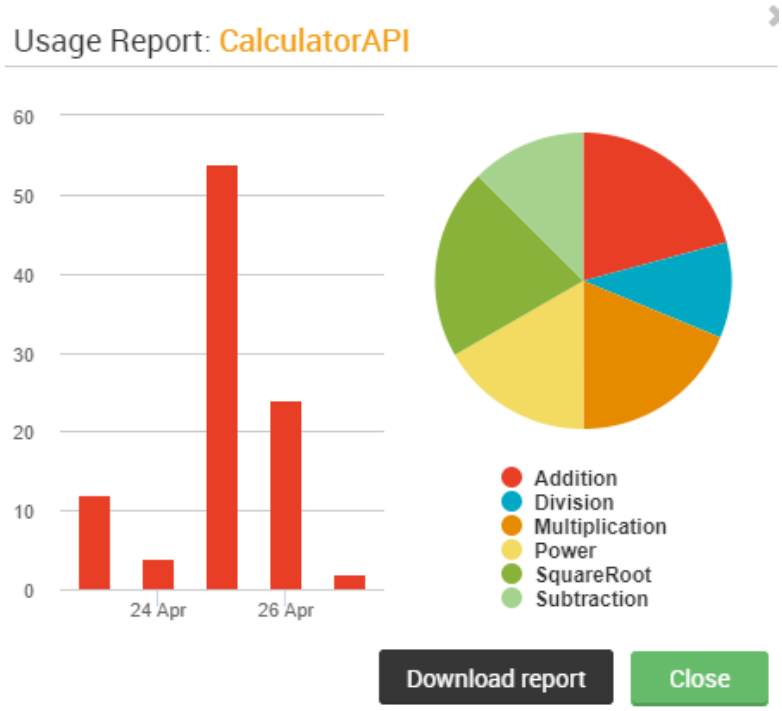


Figure 5.5: Graphs showing API proxy activity levels.

In the popup there is a ≪Download report≫ button which will download a .xlsx file containing the details of the proxy activity, as shown in Fig. 5.6.

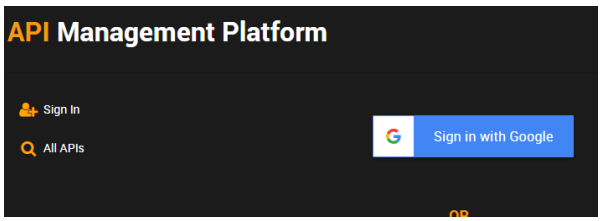
	A	B	C	D	E	F	G	H
1	Data	FlowTyp	MethodNar	RequestTy	TimeOfDa	Date	Email	IsTest
2	[66,21]	REQ	Addition	GET	10:32:19	2019-04-25		VERDADEIRO
3	87.0	RES	Addition	GET	10:32:20	2019-04-25		VERDADEIRO
4	[555,36.5	REQ	Multiplication	GET	13:19:50	2019-04-27	andrefmms23@	FALSO
5	20257.5	RES	Multiplication	GET	13:19:50	2019-04-27	andrefmms23@	FALSO
6	[3{char}:	REQ	Power	GET	15:09:16	2019-04-24		VERDADEIRO
7	{"Errors":	RES	Power	GET	15:09:17	2019-04-24		VERDADEIRO
8	[3051]	REQ	Power	GET	15:09:56	2019-04-24		VERDADEIRO
9	{"Operati	RES	Power	GET	15:09:56	2019-04-24		VERDADEIRO
10	[0.1,0.1]	REQ	Addition	GET	19:59:54	2019-04-25		VERDADEIRO
11	0.2	RES	Addition	GET	19:59:54	2019-04-25		VERDADEIRO
12	[2,44]	REQ	Subtraction	GET	20:01:29	2019-04-25	andrefmms23@	FALSO

Figure 5.6: .xlsx file containing the details of the proxy activity.

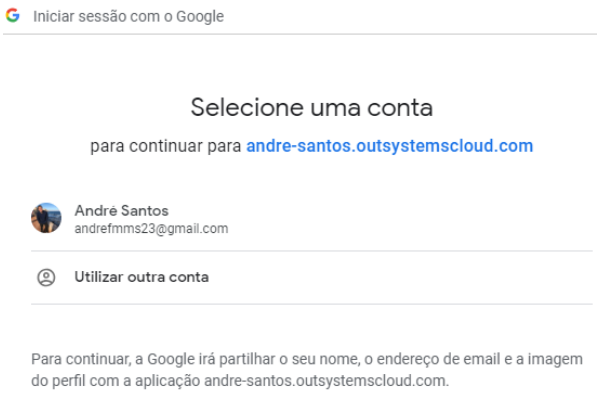
JSON to XML and **XML to JSON** : Concerning the mediation policies, and as explained in section 4.1.2.B, they are always optional on both provider and consumer sides. Fig. 5.7 show a consumer receiving a response in XML despite the original one being in JSON.

Figure 5.7: Response converted from JSON to XML.

OAuth2 : As for the authentication, the only TTP implemented at this point in time is Google. A ≪Sign in with Google≫ button is available at the ≪Sign in≫ page, as shown in Fig. 5.8(a). The button, when clicked, displays the TTP authentication popup from where the user can authenticate himself — Fig. 5.8(b). In order to experience this capability, users must possess Google accounts. In case they do not, there is still the possibility to register with an email and a password.



(a) Sign in with Google button.



(b) OAuth popup.

Figure 5.8: Policy proof of concept figures - OAuth2

6

Conclusion

Contents

Nowadays, mobile apps are made (not only, but almost) of services provided by APIs, and most of the new ones tend to go the same way. It is a market of reusability where developers create new content composed of the one others provide.

Taking that into account, companies seek to develop their services and expose them as fast as possible, as reliable as possible, to as many people as possible. Moreover, firms need some guarantees, for instance in traffic management or mediation, and from the information presented in the report, it can be concluded that the Apigee platform is, at the moment, the close-to-perfect solution as is way ahead of their competitors.

However, there is still enough space for a similar platform pointed to the OutSystems world. With the RAD possibility that is offered, users are able to produce software much faster and should be provided with a way to expose their services with the same easiness as they do everything else.

Lastly, in regard to the features proposed in section 4.1.1, it was proven they meet and emulate the basic functionalities of a proxy as well as the user flows, with one small exception which is approached in the following section.

6.1 System Limitations

With regard to impossibilities, there is only one caveat: OutSystems developers have no way to publish the REST APIs developed directly in OutSystems' Service Studio to a common accessible portal for them to be studied, tested, and used. This, in the first instance, was already known and is accepted since OutSystems does not have any APIs management platform, nor does it have a partnership with any.

However, OutSystems offers an option where, for each API, it is possible to dive into a web page showing the visual representation of its contents (e.g. functions, inputs, outputs, data types and more) — it makes use of Swagger UI — and a URL that redirects the developer to its JSON format specification.

Thus, it was possible to implement a workaround for the raised issue: in the portal that was constructed within the scope of this dissertation, the developer just has to place the URL of such page whenever a new API is published into the portal — in practise this behaviour mimics Apigee and, consequently, the Swagger tools. The backend will then retrieve the specification, parse its content, and redirect to its respective page on the portal. From there, as mentioned earlier, every user is able to test and download its corresponding SDK.

6.2 Future Work

This last section of the report entails features which were part of the initial scope but were not implemented and a few suggestions that could help to evolve the platform closer to an Apigee look-alike.

6.2.1 Not implemented features

Multiple TTPs: So far the platform only supports one authentication provider: Google (i.e OAuth). However, it is designed to allow the users to benefit from all the ones whose credentials are in the database, more concretely in the ≪TrustedThirdParty≫ table. Whenever the credentials of a new one are introduced in the database, the customer will be able to use it the next time an authentication needs to be performed.

OpenID: This policy is categorized under the security group. It allows users to authenticate towards the API manager without introducing their sign in credentials directly in there. In practice, the users obtain an identity confirmation token from an identity provider (e.g. Google), which is then transmitted to the relying party (i.e. AMP) to be checked. The policy will then authenticate the user if the token was provided by a trusted identity provider.

In the initial design of the AMP, the platform was supposed to be completely free of usage with the exception for the ≪subscribe≫ use-case — and consequently the ≪download SDK≫ because of its dependency on the subscription. At that point, the user was supposed to sign up/in using the OpenID authentication protocol with one of the available TTPs in the portal.

Due to the ≪message logging≫ policy, this design had to be altered into only allowing the non-registered users to browse and inspect the existing APIs. The goal of this change was to also track who is testing what, when and how. From that point onwards also the ≪test API≫ use-case was allowed only to registered users.

6.2.2 Suggestions

Application concept & app key: While in AMP the concept of ≪user≫ exists, the notion of ≪application≫ does not. With that in mind, as the platform already has the API key policy implemented — refer to section 4.1.2.B —, it lacks an ≪App key≫ policy.

Even though this is something that could only be implemented after the platform recognizes what an application is, an app key is a nice-to-have feature as it allows to identify and bill the actual consumer of an API.

As of now, AMP allows an API to be called from a various number of applications with a single key but with the combination ≪API key + app key≫ it would be able to recognize which applications are allowed to call which APIs while also identifying if the application owner is legit.

Monitorization: Key factor for any API management platform, monitorization tools grants API providers and AMP administrators a graphical overview, at different scales, on several kinds of statistics and results related to the API usage. As an example, providers can see which of their APIs is more popular, subscribed and/or downloaded, while administrators can understand the policy usage rate or, for instance, if any policy is causing a big increase on some response time.

The ≪message logging≫ policy is a starting point for the implementation of this feature, as the charts for API usage can be generated from the existing logs data and, as the first evolves the latest can too.

AMP Monetization: Crucial for every business, monetization is also a great deal with API management platforms, especially because it can be done in quite a few different ways.

Apigee, for instance, provides billing plans which consist of different bundles made of: (i) number of allowed users, (ii) number of authorized API calls per month, (iii) number of days included in the analytics reports, (iv) support type and (v) Service Level Agreement (SLA).

Considering the final result of AMP, a good commencement would be to also charge providers based on personalizable unit bundles composed of: (i) a number of exposed APIs, (ii) a number of allowed calls per API and (iii) a limited number of policies by API. In concrete, users would pay only for the number of APIs they expose, with the limit of calls they choose and the number of policies they want to benefit from.

API Monetization: Getting paid per finished transaction is the typical way API management platforms operate (e.g. that is Apigee's business model [22]).

As a new platform, AMP should differentiate itself and charge customers a fixed price based on their API popularity. In practice, this idea allows customers to pay an amount per transaction that fluctuates depending on the number of calls an API gets.

With the continuous evolution of the platform and the addition of new policies, the business model can be adapted to discern premium policies from not-so-special ones and possibly charge different values for them.

More policies: The more relevant policies, the better. Some can be quite challenging to develop, in particular if they can be configured with different values for different API proxies.

Being that AMP does not have any traffic management policy available, these are the two I find more important and would like to have implemented:

- **Quota policy** is especially interesting because it allows to define the number of request messages that an API proxy accepts over a period of time. Ideally, the user should be able to set a counter value to the number of allowed API calls and, a time measure (i.e. second, minute, hours or day) and a time amount. Then, the policy will let through the requests and decrease the counter by one while it is not equal to zero. After the predefined time, the counter will be reset to the initial value.
- **Spike arrest policy** enables the proxy to play defense in case of an attack. The user should simply set a number of authorized requests to per time unit (just like the quota policy). On this case, the policy should work by throttling the time passed from the last incoming message and not by a counter.

Even though these two policies may seem similar in the first place, they serve two purposes. Quota policy exists to limit the number of connections the consumers can make to the backend during a specified time interval. Spike arrest is idealized to protect the proxy again a spike in the requests or a Denial of Service (DoS) attack. As an example, a proxy can be configured to accept 1000 requests per minute with the ≪quota≫ policy but also to reject more than 5 per second with the ≪spike arrest≫ policy meaning they can work simultaneously.

Bibliography

[1] J. Miller, “Zuckerberg: Facebook’s mission is to ’connect the world’ - BBC News,” 2014. [Online]. Available: http://www.bbc.com/news/technology-26326844

[2] L. Grossman, “Mark Zuckerberg and Facebook’s Plan to Wire the World,” 2014. [Online]. Available: http://time.com/facebook-world-plan/

[3] “4 Apps that Rely on APIs for Survival - Nordic APIs -.” [Online]. Available: https://nordicapis.com/4-apps-rely-apis-survival/

[4] “Why Uber—and whatever is coming next—is really about the rise of APIs - Why Uber—and whatever is coming next—is really about the rise of APIs.” [Online]. Available: http://www.betaboston.com/news/2014/07/03/uber-mobile-app-cloud-service-api/

[5] “3 Ways APIs Create Value and 5 Acquisitions that Prove it.” [Online]. Available: https://nordicapis.com/3-ways-apis-create-value-and-5-acquisitions-that-prove-it/

[6] “How APIs Are Driving Smart Cities.” [Online]. Available: https://nordicapis.com/how-apis-are-driving-smart-cities/

[7] “Should Every Company Consider Providing an API? - Nordic APIs -.” [Online]. Available: https://nordicapis.com/should-every-company-consider-providing-an-api/

[8] “Top 10 Best API Management Tools with Feature Comparison.” [Online]. Available: Basicterminology-ApigeeDocs

[9] “Werner Vogels - Amazon and the Lean Cloud on Vimeo.” [Online]. Available: https://vimeo.com/29719577

[10] “What is the Difference Between API Documentation, Specification, and Definition.” [Online]. Available: http://nordicapis.com/difference-api-documentation-specification-definition/

[11] “Top Specification Formats for REST APIs.” [Online]. Available: http://nordicapis.com/top-specification-formats-for-rest-apis/

[12] “OpenAPI-Specification.” [Online]. Available: https://github.com/OAI/OpenAPI-Specification

[13] “The Best APIs are Built with Swagger Tools.” [Online]. Available: http://swagger.io/

[14] “Design your API - RAML.” [Online]. Available: http://raml.org/developers/design-your-api

[15] “RAML Version 1.0: RESTful API Modeling Language.” [Online]. Available: https://github.com/raml-org/raml-spec/blob/master/versions/raml-10/raml-10.md

[16] “REST - OutSystems.” [Online]. Available: https://success.outsystems.com/Documentation/11/Extensibility_and_Integration/REST

[17] “CNET Update - Robot bartenders shake it up on cruise ship.” [Online]. Available: https://www.youtube.com/watch?v=NINFJUlJlrI

[18] “API Management - Features.” [Online]. Available: https://wso2.com/api-management/features/

[19] “API Management Platforms Capabilities.” [Online]. Available: https://www.infoq.com/research/api-management

[20] “The Definitive Guide to API Management © CC BY-SA,” Tech. Rep. [Online]. Available: https://cloud.google.com/files/apigee/apigee-definite-guide-to-api-management-ebook.pdf

[21] “Top 10 Best API Management Tools with Feature Comparison.” [Online]. Available: https://www.softwaretestinghelp.com/api-management-tools/

[22] “Monetization overview - Apigee Docs.” [Online]. Available: https://docs.apigee.com/api-platform/monetization/basics-monetization

[23] R. Heffner, “The Forrester Wave™: API Management Solutions, Q4 2018,” Tech. Rep., 2018. [Online]. Available: https://reprints.forrester.com/#/assets/2/1501/RES141540/reports

[24] “OpenAPI-Specification.” [Online]. Available: https://swagger.io/blog/api-development/introducing-the-open-api-initiative

[25] “Current Members - OpenAPI Initiative.” [Online]. Available: https://www.openapis.org/membership/members

[26] “API Development Tools - Swagger Editor - Swagger.” [Online]. Available: https://swagger.io/tools/swagger-editor/

[27] “Swagger Documentation - Swagger.” [Online]. Available: https://swagger.io/docs/open-source-tools/swagger-editor/

[28] “API Development Tools - Swagger UI - Swagger.” [Online]. Available: https://swagger.io/tools/swagger-ui/

[29] “Swagger Documentation - Swagger.” [Online]. Available: https://swagger.io/docs/open-source-tools/swagger-ui/usage/

[30] “API Development Tools - Swagger Codegen - Swagger.” [Online]. Available: https://swagger.io/tools/swagger-codegen/

[31] “Swagger Codegen # Online generators.” [Online]. Available: https://github.com/swagger-api/swagger-codegen#online-generators

[32] “Swagger Documentation - Swagger.” [Online]. Available: https://swagger.io/docs/open-source-tools/swagger-codegen/

[33] “How to Use Swagger Inspector - Swagger.” [Online]. Available: https://swagger.io/docs/swagger-inspector/how-to-use-swagger-inspector/

[34] “API Testing & Documentation Tool - Swagger Inspector - Swagger.” [Online]. Available: https://swagger.io/tools/swagger-inspector/

[35] “Policy reference overview - Apigee Docs.” [Online]. Available: https://docs.apigee.com/api-platform/reference/policies/reference-overview-policy

[36] C. Richardson and J. R. Rymer, “The Forrester Wave™: Low-Code Development Platforms, Q2 2016,” Tech. Rep., 2016. [Online]. Available: http://agilepoint.com/wp-content/uploads/Q2-2016-Forrester-Low-Code.pdf

[37] J. S. Hammond, “The Forrester Wave™: Mobile Low-Code Development Platforms, Q1 2017,” Tech. Rep., 2017.

[38] G. P. Insights, “Gartner Magic Quadrant for High-Productivity Application Platform as a Service (hpaPaaS) 2017,” Tech. Rep., 2017.

[39] OutSystems, “The State of Application Development,” Tech. Rep., 2017.

[40] “OutSystems 2-Minute Overview,” 2016. [Online]. Available: https://www.outsystems.com/videos/platform-overview/

[41] “Forge FAQ.” [Online]. Available: https://www.outsystems.com/forge/FAQ.aspx

[42] “SwaggerParser - OutSystems.” [Online]. Available: https://www.outsystems.com/forge/component/2015/

[43] “JSON Pretty Format - OutSystems.” [Online]. Available: https://www.outsystems.com/forge/component/969/

[44] “ardoHTTP - OutSystems.” [Online]. Available: https://www.outsystems.com/forge/component/427/

[45] “ardoJSON - OutSystems.” [Online]. Available: https://www.outsystems.com/forge/413/

[46] “Back to top - OutSystems.” [Online]. Available: https://www.outsystems.com/forge/component/799/

[47] “HTTPRequestHandler - OutSystems.” [Online]. Available: https://success.outsystems.com/Documentation/11/Reference/OutSystems{_}APIs/HTTPRequestHandler{_}API

[48] “PlatformPasswordUtils - OutSystems.” [Online]. Available: https://success.outsystems.com/Documentation/11/Reference/OutSystems{_}APIs/PlatformPasswordUtils{_}API

[49] “ISO 8601 Date and time format.” [Online]. Available: https://www.iso.org/iso-8601-date-and-time-format.html

Api vs Service example

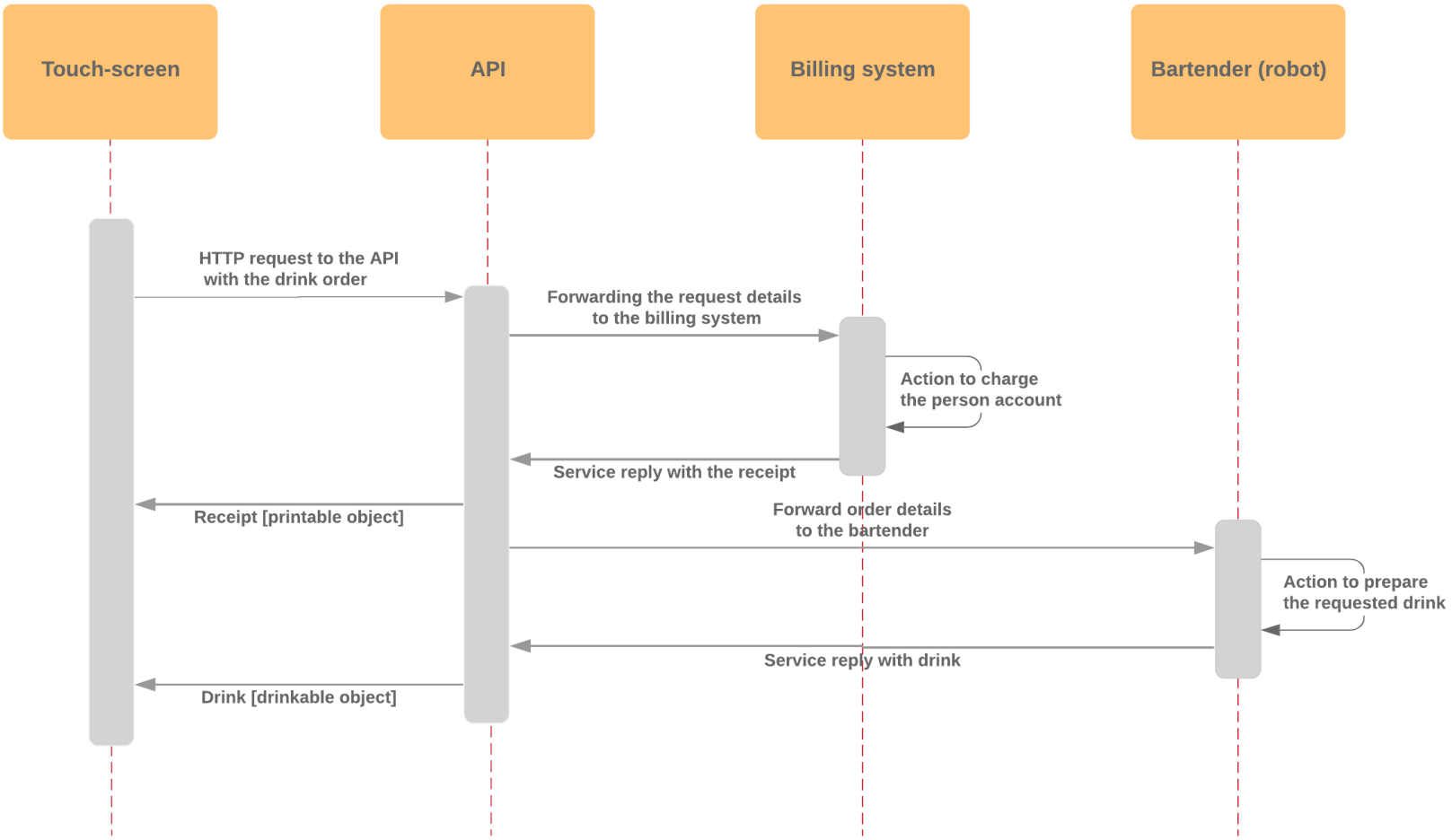


Figure A.1: Sequence diagram of the ≪order drink≫ use-case.

B

Architecture

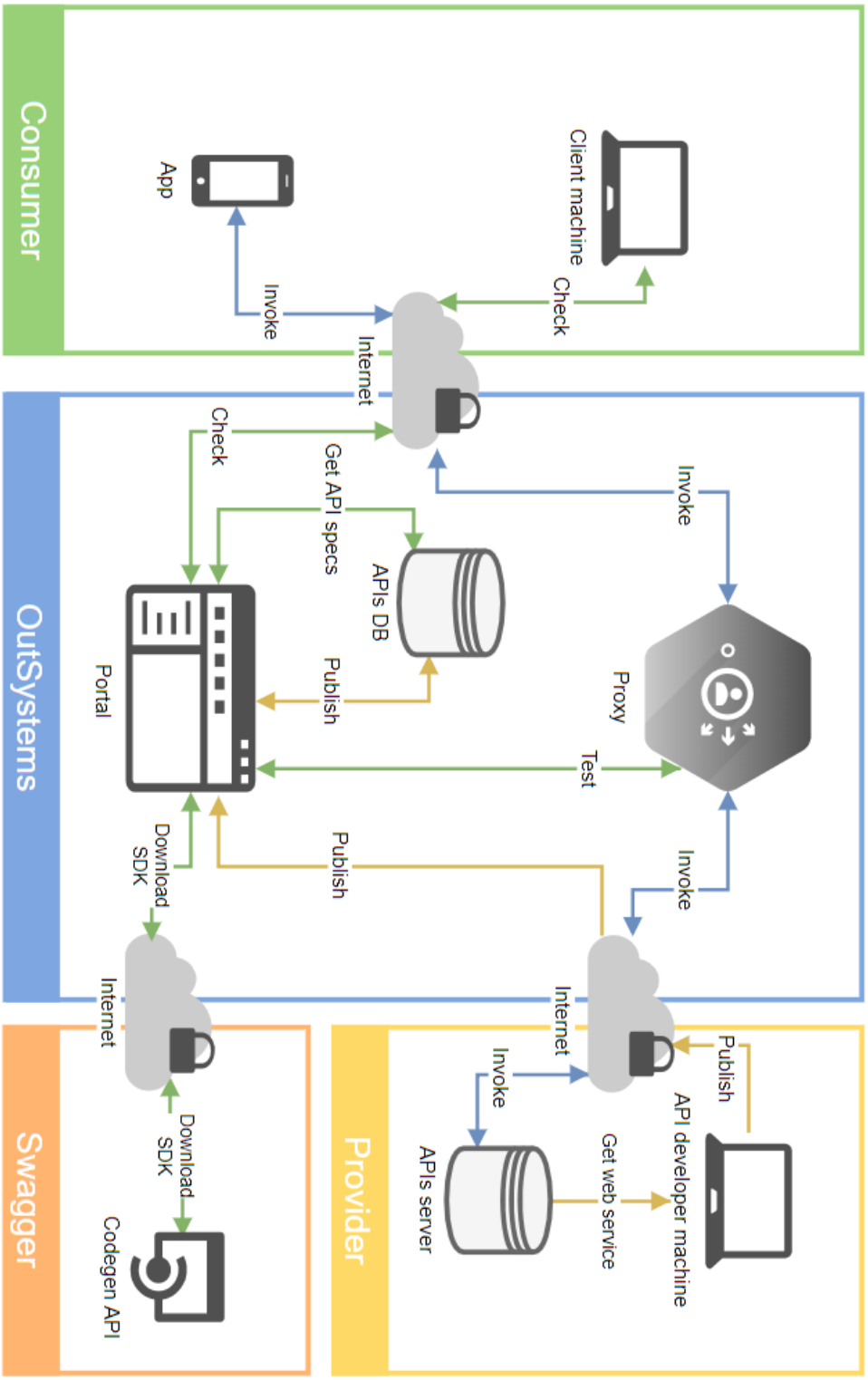


Figure B.1: Architecture of the system, with user flows and interactions

C

Platform database

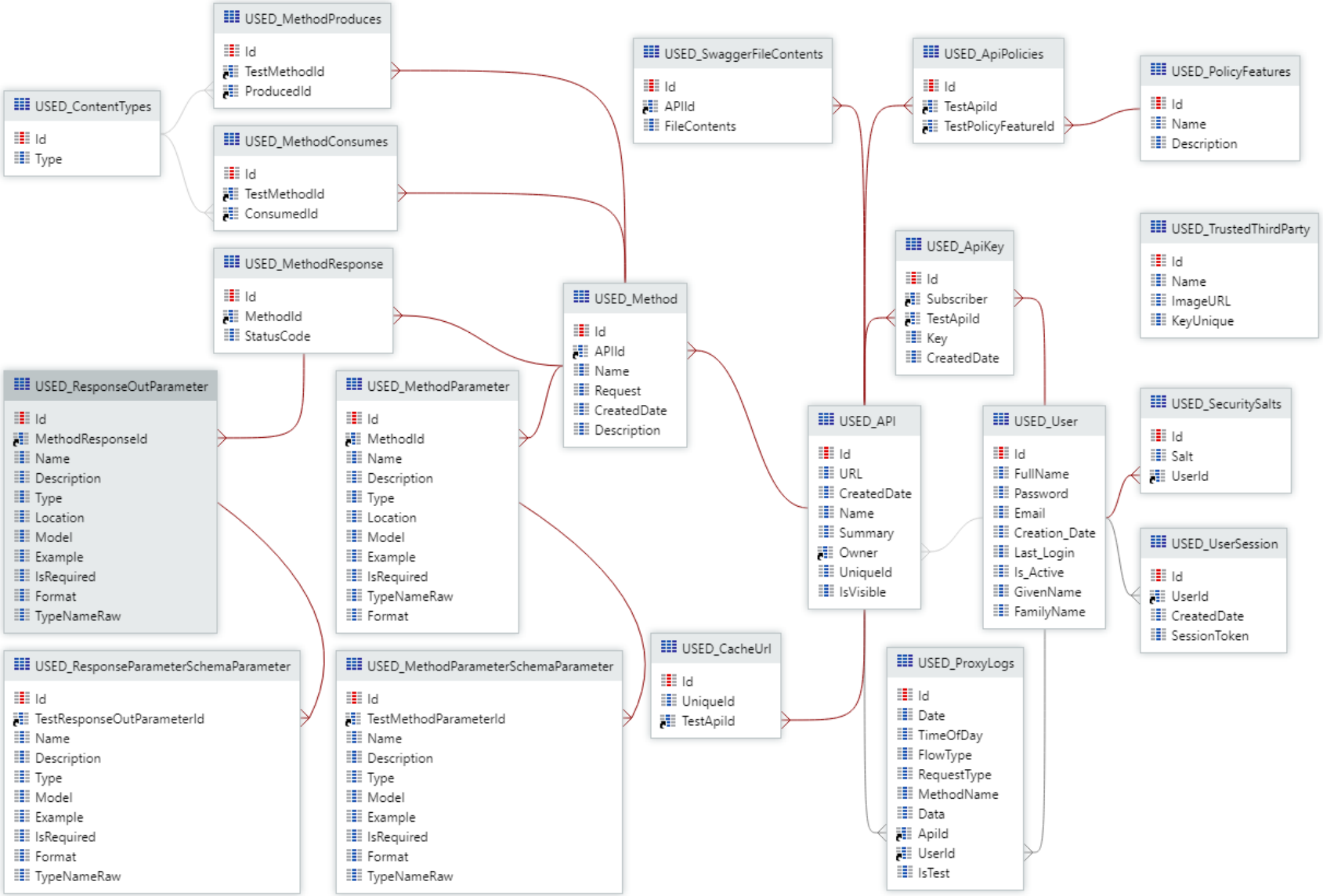


Figure C.1: Simplified version of the database supporting the platform

D

User perspectives

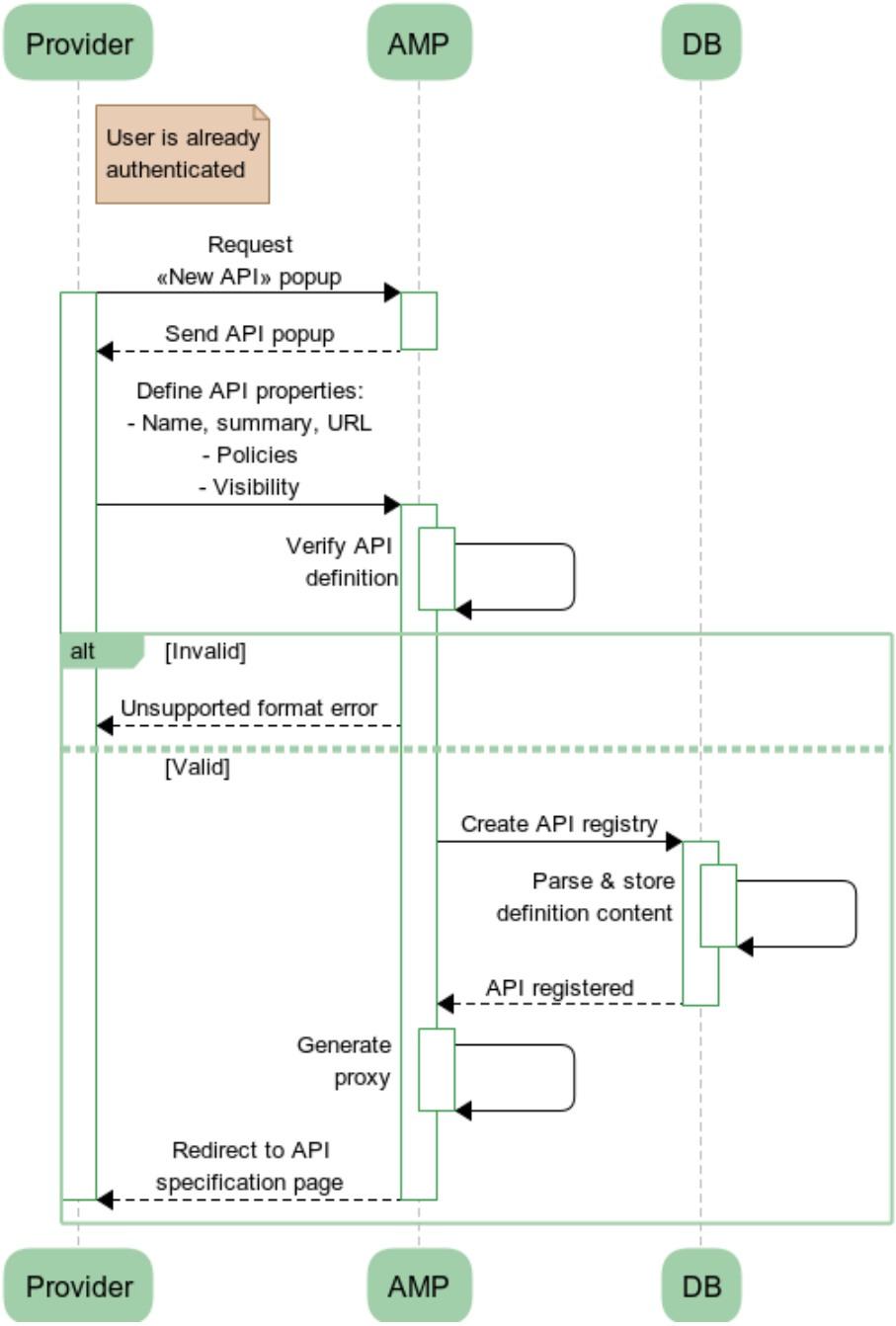


Figure D.1: Sequence diagram for publishing an API

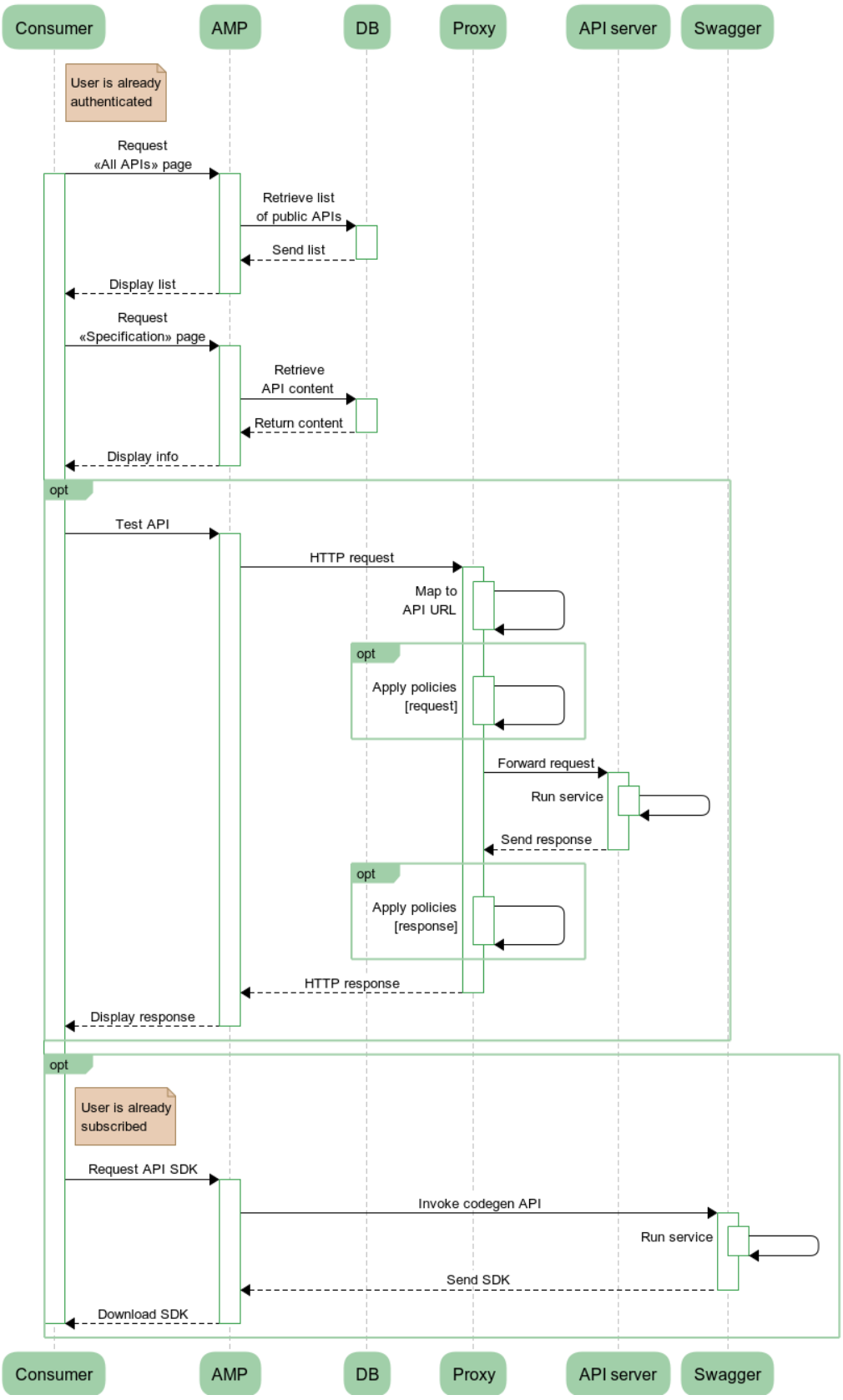


Figure D.2: Sequence diagram for testing an API

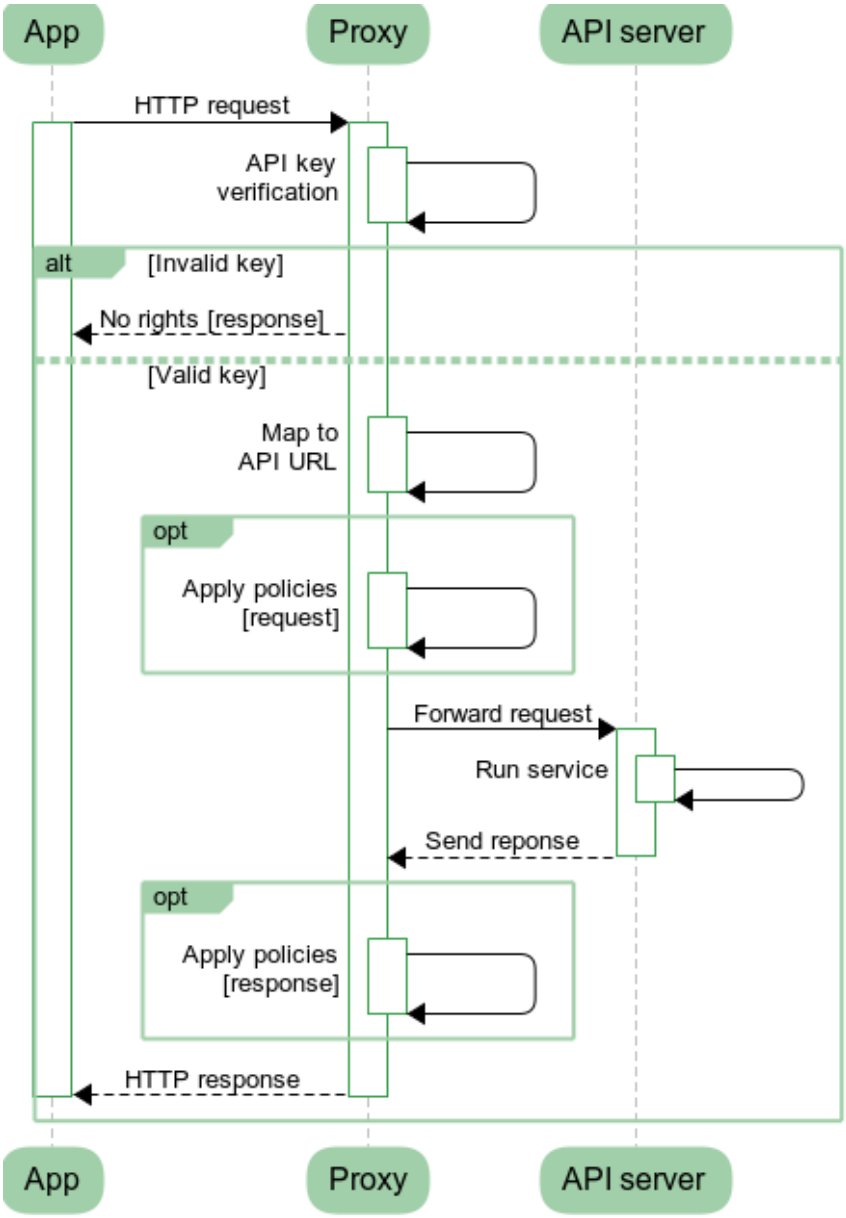


Figure D.3: Sequence diagram for SDK download