UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR
DE INGENIEROS DE TELECOMUNICACIÓN

GRADO EN INGENIERÍA DE TECNOLOGÍAS Y SERVICIOS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

DESIGN AND IMPLEMENTATION OF AN API GATEWAY BASED ON NODEJS AND EXPRESS

SERGIO GIL RODRÍGUEZ

JUNIO 2019

TRABAJO DE FIN DE GRADO

Título:	Diseño e implementación de un nexo de APIs basado en NodeJS y Express
Título (inglés):	Design and implementation of an API gateway based on NodeJS and Express
Autor:	Sergio Gil Rodríguez
Tutor:	Carlos A. Iglesias Fernández
Departamento:	Departamento de Ingeniería de Sistemas Telemáticos

MIEMBROS DEL TRIBUNAL CALIFICADOR

Presidente:	—
Vocal:	—
Secretario:	—
Suplente:	—

FECHA DE LECTURA:

CALIFICACIÓN:

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE
INGENIEROS DE TELECOMUNICACIÓN

Departamento de Ingeniería de Sistemas Telemáticos
Grupo de Sistemas Inteligentes

TRABAJO FIN DE GRADO

DESIGN AND IMPLEMENTATION OF AN API GATEWAY BASED ON NODEJS AND EXPRESS

Sergio Gil Rodríguez

Junio 2019

Resumen

Hace años que podemos decir que las tecnologías web están completamente integradas en nuestras vidas. Las aplicaciones web se han convertido en uno de los medios principales con los que interaccionamos con el mundo y desarrollamos nuestras actividades diarias. Es difícil encontrar un ámbito en el que estos sistemas de la información no jueguen algún papel o nos ofrezcan algún servicio.

Esto implica un almacenamiento, gestión y disponibilización de una gran cantidad de información de diverso origen y naturaleza. Es cada vez más frecuente que para ofrecer un servicio complejo mediante una aplicación web se deba obtener y procesar información de múltiples sistemas independientes. Lo que obliga al equipo de desarrollo a interactuar con diferentes interfaces, documentación y criterios de diseño y seguridad.

En este trabajo se propone una plataforma que sirve como gestor de publicación de estas interfaces HTTP (APIs), siendo el responsable único de la interacción con los sistemas de fondo que las exponen. De esta forma es posible ofrecer una interfaz uniforme y una abstracción de las peculiaridades de cada sistema a las distintas aplicaciones de cliente que necesiten su consumo.

Para ello este nexo de APIs debe permitir que sus usuarios especifiquen cómo debe consumirse la información de un sistema de fondo, cómo se va a exponer dicha información a los consumidores, y qué funciones va a tener que realizar como intermediario para convertir lo que el sistema de fondo expone en lo que se quiere ofrecer a los consumidores.

A lo largo del trabajo, se diseña y se desarrolla una arquitectura que permite realizar esta tarea teniendo en cuenta la escalabilidad y las diferentes situaciones o necesidades que podrán darse en el futuro en un sistema que pretende poder dar cabida a cualquier API HTTP. A continuación se presenta un caso de estudio que justifica la necesidad de este nexo para una situación ficticia, y se resuelve mediante su uso.

Por último se exponen las conclusiones extraídas de la realización de este trabajo y posibles líneas de trabajo que amplíen su funcionalidad.

Palabras clave: Web technologies, API, Gateway, Nodejs.

Abstract

For years now we can say that web technologies are completely integrated into our lives. Web applications have become one of the main tools through which we interact with the world and perform our daily activities. It is hard to find an area in which these information systems do not play a role or offer us a service.

This implies the storage, management and provision of a large amount of information from diverse sources and nature. It is becoming more common that in order to offer a complex service through a web application, information from multiple independent systems must be obtained and processed. This requires the development team to interact with different interfaces, documentation and design as well as security criteria.

This paper proposes a platform that serves as a publication manager for these HTTP interfaces (APIs), being the only responsible for the interaction with the backend systems that expose them. As a result, it is possible to offer a uniform interface and an abstraction of the particularities of each system to the different client applications that need their consumption.

To this end, this API gateway must enable its users to specify how the information in a backend system should be consumed, how this information will be exposed to consumers, and what role the gateway will have to play as an intermediary, to turn what the backend system exposes into what it is intended to offer to consumers.

Throughout the study, it is designed and developed an architecture that allows this task, taking into account scalability and the different situations and requirements that may occur in the future in a system that aims to accommodate any possible HTTP API. The following is a case study that justifies the need for this gateway for a fictional scenario, and is resolved through its use.

Finally, it presents the conclusions drawn from this work and possible lines of work to extend its functionality.

Keywords: Web technologies, API, Gateway, Nodejs.

Agradecimientos

A mi tutor Carlos, por darme todas las facilidades y ayuda posibles para que pudiera realizar este trabajo, y terminar una etapa.

A mi familia, por todo el apoyo y las oportunidades que me han dado.

Contents

List of Figures

CHAPTER 1

Introduction

Digital transformation in companies has many implications in all its areas. The integration of digital technologies causes fundamental changes in how they operate and deliver value to consumers. It also needs a cultural evolution that requires organizations to periodically change their paradigm, and to accept in their philosophy that it is necessary to be comfortable with experimentation and failure.

This adaptive behaviour is necessary because the transformation never ends, it does not end with the digitalization of business processes. Technology itself evolves, opening up new opportunities, but also offering new solutions to classic problems. The ability of a company to take advantage of technological development is a competitive advantage.

Incorporating new technology into the business structure is not trivial, and its difficulty tends to be proportional to the age of the technological resources that are already productive, and the business volume that the company handles with them. Hence the criticality of taking into account extensibility and scalability when making design decisions in technical solutions.

Our case study arises within the framework of a long-distance company in Spain in the retail sector. Although the identity of the company and the details of its architecture and technical implementations cannot be shared, it serves to place the requirements and

solutions in a tangible context.

The retail sector is a peculiar scenario in the digital transformation [19]. While the ultimate goal of the companies in the sector (the transfer of goods to customers) cannot be digitized in itself, all consumer-business interaction and its internal organization have undergone very aggressive changes in the last 20 years. Thus, a sector based on physical presence and intimately related to the real estate market, is becoming radically transformed to be based on online presence and to be intimately related to the logistics sector.

During this process, the company has been incorporating technology and digitizing processes incrementally until completely replacing the core business and basing its entire operation on digital technologies. However, these developments were addressed to meet specific needs of digitization, without taking into account the sudden growth of online sales in the sector. With the appearance of new sales and service channels, interaction between all subsystems is required and they start to be put under a load they were not designed for.

Once again, these needs are solved with local solutions and scaling vertically the systems involved. But the situation is not sustainable in the long term, the amount of accumulated technical debt and the limitations of the technologies used make it very difficult to implement new functionalities and scale the performance of these systems to deliver the customer experience demanded by the market.

At this point, the development of applications that provide new services implies the interaction with a multitude of systems responsible for different functional areas of the organization, which have been developed in different technologies and with different design criteria. This makes it very difficult to follow the methodologies and design patterns on which modern software development is based.

In this situation, it is proposed to develop an architectural piece that interfaces with these subsystems and alleviates some of their problems, but at the same time allows sufficient decoupling so that the teams responsible for implementing new client applications can do so without dealing with all this complexity.

1.1 Context

The case study takes place specifically in the web and mobile application development division of the company. At the moment the solution is proposed, a sales web portal is in operation and three new projects are started: a food-exclusive web application and native mobile applications in Android and iOS.

It is at this point in which it becomes evident that the parallel development of three applications becomes unnecessarily complex due to the fact that a good part of the workload comes from the data consumption of the different APIs of the backend systems and from implementing in triplicate the logic of interaction with these systems.

The technological background of the company described previously has resulted in the backend system of web applications being composed of many isolated subsystems, each with a different interface, which must be taken into account when making use of them.

So that a web application in this ecosystem will have to implement, for example, logic to interact with both SOAP [9] and REST [18] services, and within the latter, a variety of questionable implementations of REST. In addition, the documentation of APIs is the responsibility of the team behind each subsystem, so they are located in multiple sites and are documented at different levels of detail.

Each API is also under its own security layer and different credentials and authentication methods have to be dealt with. As if that weren't enough, some systems have seen their workload increase drastically, and suffer performance problems that are not going to improve with the arrival of new applications.

There are also issues with the topology of the corporate network because some systems can only be reached from within the organization, and exposing them directly presents a security risk that needs to be carefully evaluated first. It is essential for mobile applications to be able to access this information from a public network.

Ideally, all these problems should be solved in their original systems, in the right time and form. However, the difficulty that the development has in many of them and the time required for its re-engineering (which for some subsystems is already planned), is incompatible with the roadmap of the company for the division of client web applications, which is critical to its business strategy.

That's why it was decided that, even if it didn't solve the real problems, at least it would be worth taking a common factor from the efforts that were being made separately in each application to adapt to the situation, and unify them into a single piece of software that performs that work transparently for current applications and those that may arrive in the future.

1.2 Project goals

The objective of the project is the design and implementation of an API gateway, which allows to decouple the background systems from the web applications enough to make them available with an interface that follows more modern API design and security standards [20].

It must be able to support the publication of any API, offering enough flexibility to make any adaptation in the communication between consumer and resource, but so the development of this software continues to be sustainable in the long term and does not grow out of control trying to accommodate all the particular needs of each system involved.

In order to do so, the following tasks have had to be carried out:

- Design of the general architecture of the API gateway.
- Implementation of the functional core of the API gateway.
- Elaboration of a case study that covers the typical needs that this software fulfills.
- Design and implementation of the gateway's plugins that solve this case study.

1.3 Structure of this document

In this section we provide a brief overview of the chapters included in this document. The structure is as follows:

Chapter 1 An introduction where the motivation, the main goals, and the structure are described.

Chapter 2 The enabling technologies are explained, giving specific details of their most useful features for this particular work.

Chapter 3 The system's architecture is presented, as a possible approach to efficiently fulfill the requirements.

Chapter 4 A case study is proposed and solved, illustrating the use of this software.

Chapter 5 Final conclusions are drawed from the entire work, a few lines of work to improve functionalities are suggested.

CHAPTER 2

Enabling Technologies

2.1 Language and libraries

2.1.1 Javascript

JavaScript [10] has been the language chosen for the development of the project. It is an interpreted language based on the ECMAScript [26] specification, with dynamic typing and multi-paradigm, therefore it has some interesting features for our use case.

The fact that it is compatible with the most important features of imperative and functional languages simultaneously makes it convenient to model the configuration of the APIs in a manageable way as objects and yet implement the business logic under a functional paradigm.

Although the origin of JavaScript is found in web technologies, and its use is predominant in web browsers, its growing popularity has made it reach many areas. We can find it in background systems, mobile applications, or even in "Internet of Things" projects.

This versatility has led to a multitude of libraries, tools and frameworks being developed in order to make JavaScript a flexible working environment in so many areas. The choice

of these tools can be truly tedious due to the large number of alternatives that arise each year and that would have to be analyzed.

2.1.2 Ramda.js

Ramda.js [1] is a library of utility functions like many others, but with a purely functional approach. It provides a large number of functions not natively available in JavaScript that are indispensable to achieve a productive and fluid development workflow under the functional paradigm.

It is based on the Fantasy Land specification for JavaScript, with immutability and the avoidance of side effects as its basic design pillars. All the functions it provides are curried by default and their parameters arranged in a convenient way to take advantage of this feature.

This is possible because functions in JavaScript are first-class citizens, this means that they support the same properties as all other entities. In JavaScript functions are objects, they inherit from the Object prototype and they can be assigned key-value pairs. The value of JavaScript functions enjoying first-class citizenship is the flexibility they allow. Functions as first-class objects opens the door to all kinds of paradigms and programmatic techniques that wouldn't be possible otherwise. Functional programming is one of the paradigms that first-class functions enable.

Functional programming [12] encourages building complex logic by composing pure functions. This results in an elegant code, whose behavior is predictable and independent of context, and makes testing much easier.

2.1.3 Mocha.js, Chai.js y Sinon.js

The development of tests is a basic task to ensure a minimum level of quality in software engineering. During the development phases of this project it has been followed as widespread practice the development guided by tests or TDD [5] (Test-driven development), consisting in that the requirements of the software are first described as test cases that the software must meet successfully. Thus, all the functionalities that are incorporated into the stable versions of the application must have at least automated tests of the requirements they satisfy.

Mocha.js [25] offers a framework in which synchronous or asynchronous test cases can be defined, that run in series and generate a wide variety of reports.

Chai.js [25] is a library of assertions that offers a chain syntax, very comfortable when building the expressions in the form of expectations that define the test cases.

Sinon.js [3] is a library of fake test entities, such as spies, stubs and mocks. They serve to cover the cases of black box test in which we have dependencies with other modules from the own module under test. The dependencies of the tested code are simulated by providing these false modules instead, over whom we have absolute control of their behavior at test execution time.

2.2 Framework and database

2.2.1 Node.js

Node.js [21] is a cross-platform, open source JavaScript execution environment, based on the JavaScript v8 interpreter engine. It allows the creation of web servers using JavaScript and includes a good collection of modules that provide basic functionalities such as interaction with the file system, network protocols, cryptography, etc.

It processes the requests through a single execution thread, arranging the calls following a loop of non-blocking events. This strategy of attention and processing of requests makes it especially efficient for scenarios in which there are many requests of low computational requirements, as is our case.

This vastly simplifies writing web apps since there is no need to deal with concurrency problems, and it address the needs of multithread performance by booting multiple instances. This could seem like a step backwards in computation, but its has to be though as just putting the complexity of multi-thread programming hidden into the operative system scheduler.

In Node.js module system each file is a separate module. With a really simple syntax any file can define the interface it exposes, which make a neat way of defining self-containing modules in plain JavaScript. In the same way, modules can be imported from any file by referring to the path of its file. To make route management easier, the environment provides a directory dedicated exclusively to saving modules, which represents the default root for any relative path.

Node.js is currently in its v12 version, which includes substantial improvements in stability and performance. Its maturity, the great reception by major companies in the world of software, and the possibilities offered by being able to develop full-stack applications

knowing only JavaScript, have made it very popular in recent years.

Currently some of the most popular areas in which Node.js is being used are serverless solutions, micro service architectures developed on Node.js and real-time applications.

Its package manager, npm (node package manager) automates dependency management, downloading and installing from the official repository (npmjs.org) the packages configured in the project.

2.2.2 Express.js

Express.js [6] defines itself as a fast, unopinionated, minimalist web framework for Node.js, and is a fairly accurate description. It provides a diverse set of features to easily prototype a basic web application server scaffold.

Express.js philosophy is to provide a minimum layer between you and the server, trying not to get in your way. Unlike many frameworks that give you everything and are inflated with unnecessary functionality, Express.js provides you with a very minimal framework, and you can add different parts as needed, replacing anything that doesn't meet your needs.

Express.js is widely used in single-page web applications. This means that the server provides the static files needed to represent the interface, and that work is delegated to the client, while the backend system is relegated to serving data.

Its URL parsing utilities, routing methods and middleware-oriented design has been very useful when developing over the whole project.

2.2.3 MongoDB

MongoDB [7] is a popular non-relational open-source database, oriented towards the storage of flexibly structured documents. Each database created in MongoDB is organized in collections that index documents, encoded in JavaScript Object Note in Binary form (BSON being Binary JSON). This versatility will allow us to create a live configuration format, adaptable to future APIs that are published without having to twist an unnecessarily complex data model each time a new subsystem must be accommodated.

By allowing the storage of full JavaScript compliant documents,it makes possible to represent complex hierarchical relationships within a single record. This makes interacting with the database much closer to the development time experience, because it is much more intuitive to exchange information with the database that has the same format in which the

developer usually conceives the data.

MongoDB also provides a great variety of features. On indexing, it supports generic secondary indexes that allow to perform fast queries on compound or full-text indexing for example. On data types, it supports time-to-live collections and documents, fixed-sized collections, and provides a really handy protocol for large file storage.

This database is designed to be distributed, which makes it adaptable to the horizontal scalability needs of each project, either for reasons of performance or availability. It uses a "master-slave" configuration between replicas, and allows intelligent fragmentation of collections to balance storage and workload.

2.2.3.1 Why a non-relational database?

This type of database has become very popular in recent years, but its use in software projects is not always properly justified. In this case, enough options have been assessed before making the decision, and the strongest arguments that tipped the balance towards a non-relational database are the following:

- The database will only be used to save HTTP responses and configuration files, both in JSON format.
- The structure of a configuration file must be versatile enough to accommodate any API.
- The structure of an HTTP response is unpredictable and it is not the responsibility of this system to validate it in any way.
- Since no queries will be made on the configurations, there is no benefit for the query in formally modeling the configuration.

2.3 Tools

2.3.1 Docker

Docker [24] is an open-source containerization engine that combines an application deployment engine over an execution environment for virtualized containers. It is a light and fast solution that allows you to isolate your software project from the particular conditions of the host machine where it is hosted, but without the great overhead of classic virtualization.

It consists of a minimum binary client that is available for most operating systems. It uses a differential copy system to build compositional images used to launch containers that start quickly without the need for a hypervisor.

Docker hub is a collaborative image repository used by the community where you can find and fetch images for the vast majority of popular open-source tools and frameworks, that official main supporters publish and maintain.

It is a very powerful tool to reduce problems related to provisioning and deployment in the software development cycle, and to simplify its distribution and scalability. In this project it has been very useful to implement a continuous integration cycle and to deploy a testing setup.

2.3.2 Postman

Postman [13] is a popular API development environment that provides plenty of useful functionality when working with APIs. Postman allows you to:

- Organize APIs in collections in a very user-friendly way.
- Create mocks and use them for quick prototype in early phase development.
- Debug APIs in real time performing calls and inspecting their answers.
- Perform automatic testing on APIs and use these tests to monitor their status.
- Create automatically generated API web documentation.

In this project it has been essential to implement the tests of the APIs exposed by the gateway in the use case.

CHAPTER 3

Architecture

3.1 Introduction

In this chapter we describe the main concepts that have been kept in mind when designing the gateway, and the architecture that has finally taken shape. Despite having a simple data model, the greatest complexity of the gateway resides in the level of abstraction that the architecture must be implemented with, which tries to accommodate any specific need that an API published in the future may have.

Also, a good amount of the implementation work of the gateway reside in the plugins, which we will introduce later, they are what gives the power to this software. With a good work of design and refactoring of these, the difficulty of publishing a new API can be decreased with the number of APIs published, because their reutilization allows to take advantage of the work already done.

3.2 The API model

The architecture of the API gateway revolves primarily around the concept of APIs. In order to offer users of the gateway a consistent but versatile syntax for publishing APIs, we first have to define what an API means in the logical domain of this system.

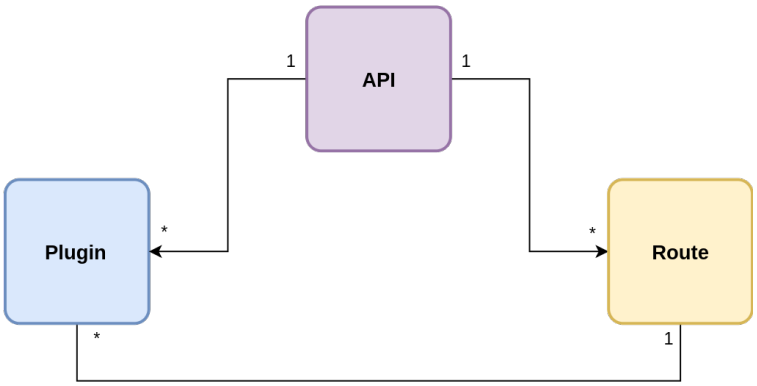


Figure 3.1: Entities composing an API

When we think of describing and adapting an API it immediately arises the need to define entities to cover the concepts of resource and adaptation, which in this system have been modeled as routes and plugins respectively.

An API symbolizes a set of resources in a background system with a high semantic cohesion that will be exposed by the gateway under the same root domain. The description of an API contains the information about which is that domain, security aspects, and plugins that will act on all the resources of that API.

The routes symbolize the multiple resources of an API that will be exposed by gateway for consumption. The description of a route contains information on what is its exact path in the backend system, security aspects and specific plugins of that route.

The plugins symbolize the multiple adaptation operations that can be performed to transform the interface offered by the backend system into the interface offered to the API consumer. The description of a plugin contains information about which adaptation is to be performed, and what configuration will be used in that adaptation.

Later on, we will see specific examples of an API description file and of each of the entities mentioned.

3.3 The configuration file

The configuration file is where all the necessary aspects to publish an API in the gateway are collected. Below is an example of a configuration file for a dummy but syntactically valid API. The different sections and their purpose will be commented in the file itself.

```
{
  "id": "api_v1",  // API id
  "name": "example",  // API name
  "version": "1",  // API version
  "host": "http://backenddomain.com",  // Root domain of the API in the backend system
  "security": {  // Needed scopes for accessing the API
    "scopes": [
    "example",
    "example:read",
    "example:write"
    ]
  },
  "plugins": [  // API level plugins list
  {
    "name": "generalplugin", // Plugin name
    "config": {}  // Plugin runtime config
  }
  ],
  "routes": [   // API routes list
  {
    "path": "/route/:param",  // Route's exposed path, with param notation
    "method": "GET",  // Route method
    "cached": 86400000,  // Route using cache, deactivated by default. Value is
        expiration in milliseconds.
    "security": {  // Needed scopes for accessing the route
      "scopes": [
      "example",
      "example:read"
      ]
    },
    "plugins": [  // Route level plugins list, same sintax as API level ones.
    {
      "name": "routeplugin",  // Plugin name
      "config": {} // Plugin runtime config
    }
    ],
    "upstream_path": "/route?param=:value"  // Specific path in the backend system for
         this resource
  }
  ]
}
```

3.4 The plugin design

As already explained, the plugins symbolize the logical operations that must be performed to adjust the API of the background system to the exposed API. To do this, you may need to add or transform headers before sending the request to the backend system, or even transform the response to match the exposed data model.

To this end, the plug-ins have been designed as a encapsulated logic in a single independent file. All these files must expose the same interface, which consists of a pair of generic functions that symbolize respectively the adaptions to be performed before and after sending the request to the backend system. As such, it is said that plugins have a request phase and a response phase. All plugins can receive a configuration object to condition their behavior at runtime if necessary.

Plugins do not have information on what level or in what order they should be used, they are mere operations agnostic to each other. It is the responsibility of the configuration file to compose them properly so the result is as expected.

The composition of plugins is done following an "onion" model, in which the plugins are nested with a sense of depth. When a plugin B follows a plugin A in a configuration file, the request phase of A will be executed first and then the request phase of B and, after receiving the response from the backend system, the response phase of B will be executed and then the response phase of A .

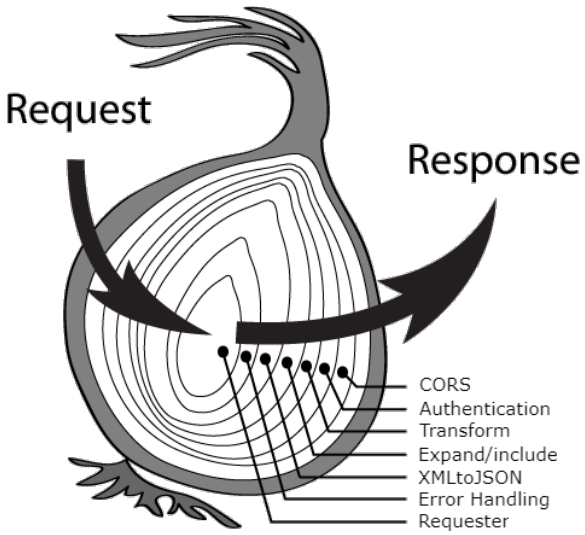


Figure 3.2: "Onion" model of the plugins [2]

There are two levels within the configuration file where plugins can be composed. The

first level "plugins" property, which refers to the plugins that will affect all API routes equally, and the "plugins" property of each route object, which comprises the plugins that will run only for that particular route. When a request is made by a consumer, the stack of plugins to be executed will be the one formed by the API plugins and then the plugins of the requested route.

3.5 The logic architecture

Next, it is specified how the entities mentioned in the previous sections are integrated into the gateway, and what role they play in the consumption of APIs. The architecture will be described following step by step the processing of an HTTP request and describing the modules that take part in its treatment and response.

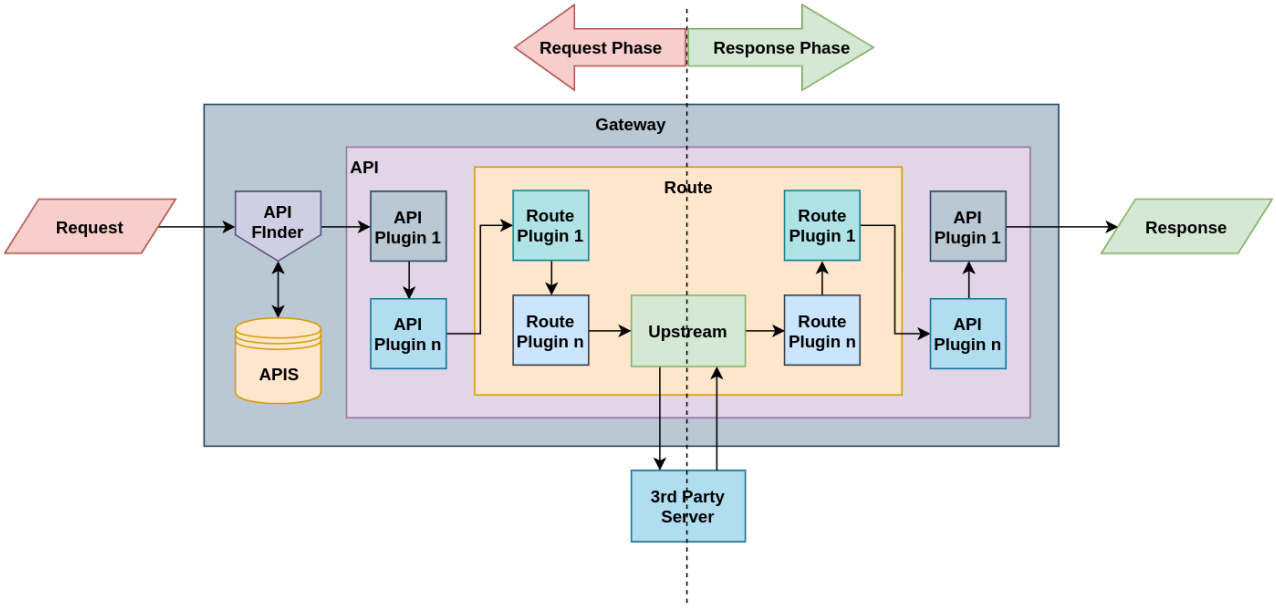


Figure 3.3: API gateway flowchart

The sequence from receiving a request to sending a response is conceptually divided into two phases. The request phase, from the reception until dispatch of the corresponding request to the backend system, and the response phase, from the reception from the backend

system until dispatch to the gateway consumer.

The request phase begins when the request is received from the consumer, then the gateway attempts to identify which API is intended to be consumed. To do this, a memory database is consulted in which every API configuration file has been indexed by root domain.

Once the API configuration has been located, the gateway already have all the information about which adaptations will be made to the request and the response, and which resource of the backend system the user wants to access. At this point the gateway starts to execute the stack of adaptations needed for this resource, starting with the request phase of the API-level plugins, followed by the request phase of the route-specific plugins.

With the request duly adapted, it is sent to the backend system and, once its reply is received, the response phase of the route-specific plugins is executed, followed by the response phase of the API-level plugins, both in reverse order. Now that the response is properly adapted, it is sent to the consumer.

3.6 The security

One of the virtues of the gateway is the abstraction that offers to customers from the specific security layers of each background system. However, a layer of security must be offered to protect the consumption of APIs and, following the gateway philosophy, it must be uniform for all consumers. Even so, we must offer a way to distinguish scopes, to restrict access to a subset of the exposed APIs.

The authentication flow is based on Oauth 2.0 [11]. As usual, applications authorized to access the gateway must authenticate using valid credentials existing in the database. Then, the gateway grants an access token with the same associated access permissions than authentication credentials. The configuration file of each API specify the permissions needed to gain access.

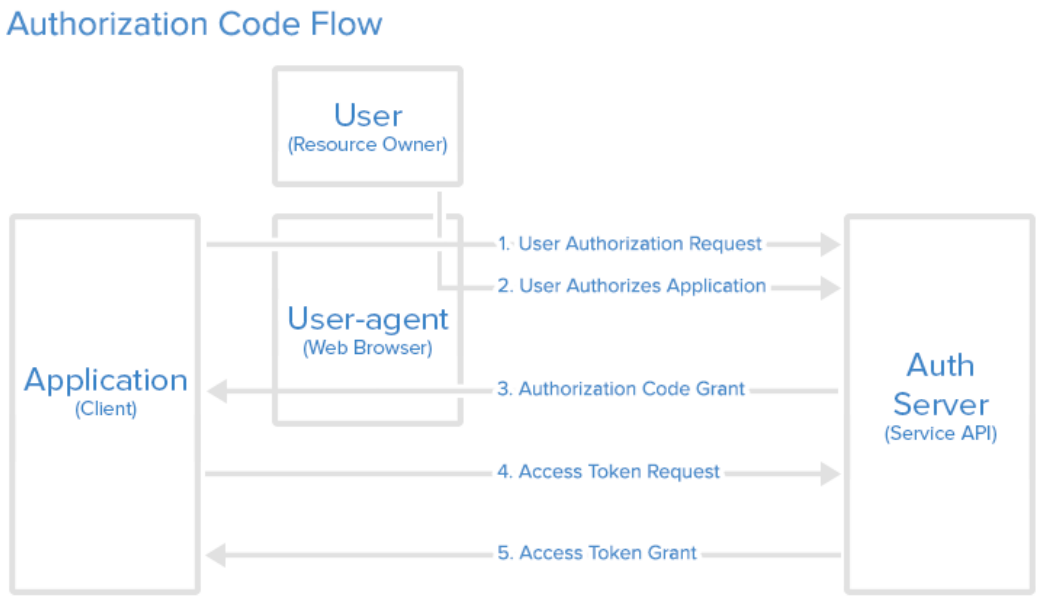


Figure 3.4: Oauth 2.0 authentication flow [4]

3.7 The cache

A response caching system has also been implemented and offers several advantages. On the one hand, it relieves some workload on back-end systems that may have performance problems, but at the same time it speeds up the response time and can help saving API calls on backends with payment plans that invoice by volume of queries (the vast majority).

For this purpose, a collection has been enabled in the database dedicated to this functionality. The system considers that a response already exists in the cache when there is an entry in the database with the same **domain, path and parameters** as the request made to the back-end system, and this entry **hasn't expired**. In this case, the module responsible for making the request serves the response stored in the cache in its place.

In order to configure the cache in an API there is a section of the configuration file for each route in which we can specify if this route is cached, and what is the maximum time to live for the cache value.

3.8 The infrastructure and initialization

Since its inception, the architecture of the gateway has been conceived to be able of being scaled horizontally. Since its behavior is only determined by the API configuration files in

use, it is perfectly possible to balance the load among multiple instances of the gateway that are using the same configuration, since they share the same code base.

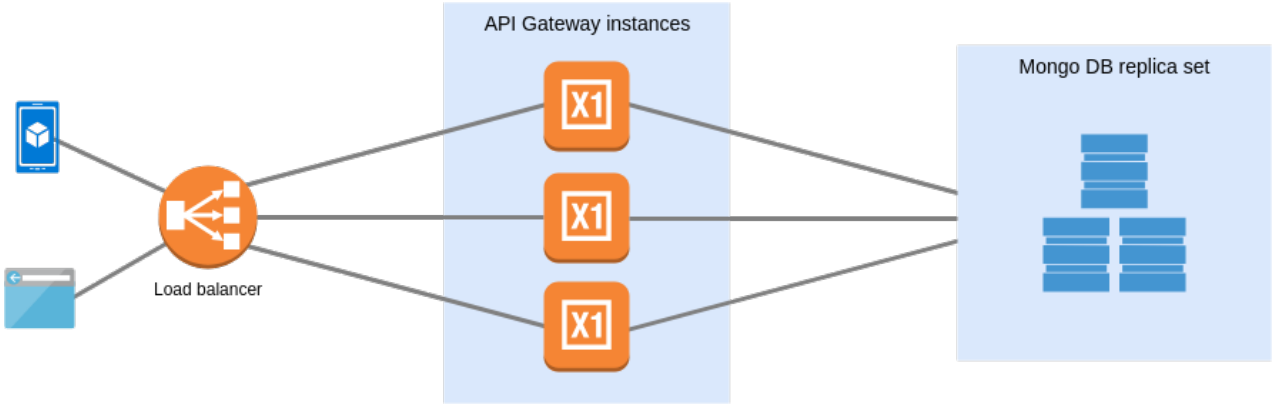


Figure 3.5: Deployment Infrastructure

In the boot process, the application reads the collection of configuration files stored in the database, and makes a copy in memory. It is in this copy where the incoming requests to the instance are consulted, and the one used as a reference to process the request. This reduces the latency that the constant query to the database would cause, but adds an additional synchronization problem. It must be guaranteed that all instances are using the same configuration, and that publishing a new API or modifying an already existing one implies that all instances are updated.

This is achieved using what MongoDB defines as a "tailable" cursor. Cursors are objects that represent the result of a query, and they are closed and destroyed when all of the documents in that query have been read by the database client. However, these tailable cursors symbolize a "live" query, in which new documents can arrive at any time. With these cursors, a permanent connection to the database is maintained, over a collection where the identifiers of the configuration files that have been added/modified/deleted are published. Thus, all instances keep this cursor alive, and react to the publication of updates by querying the modified file in the database and replacing it in their in-memory database.

CHAPTER 4

Case study

4.1 Introduction

In this chapter, a case study for the system is proposed, and the solutions that have been adopted to solve it will be presented.

The chapter will be developed by presenting a made-up case study in which some APIs should be published in the gateway, and their adaptation requirements will be analyzed and solved case by case, with the development of generic or specific plugins.

To propose a case study, its necessary to have functional APIs with enough access permission. Although there are many open and public APIs available, its not easy to find exactly those that comply with an arbitrary fictional scenario and ,at the same time, they cover a good example of gateway's use case. So as the domain of the problem is not important in this case, the reverse process has been carried out.

Investigating collections of public and free to use APIs, we made up an scenario that serves as a simple case study to illustrate the basic functionality of the gateway.

4.2 Scenario

The case study scenario consists of a web application, with browser and mobile version. This application has the purpose of offering leisure activities close to the user's location, such as places to eat, events and outdoor activities.

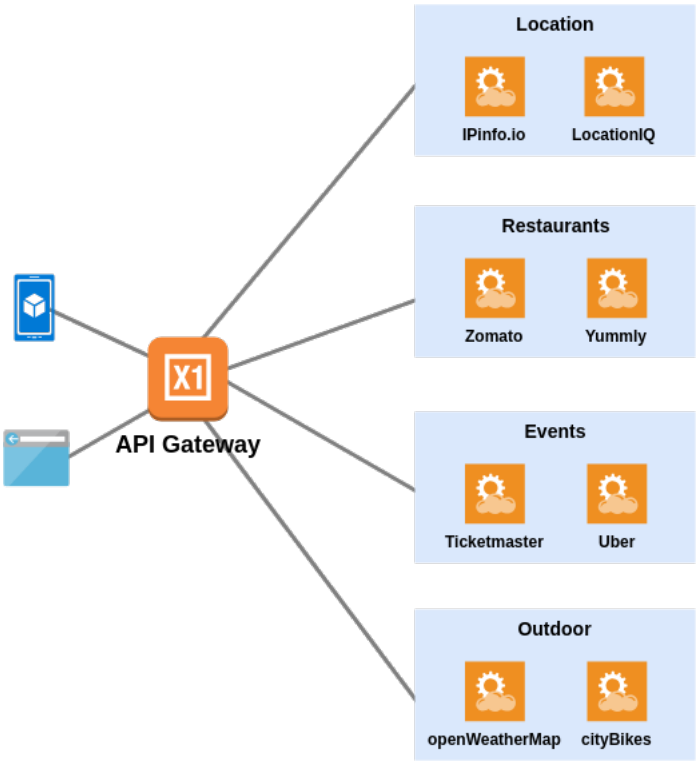


Figure 4.1: Scenario

For this purpose, several public APIs are used:

- IPinfo.io [15] and LocationIQ [16] to locate the user in a city by IP and geographic information.
- Zomato [17] to find restaurants and their menus in that city and Yummly [27] to provide the nutritional information of each dish.
- Ticketmaster [23] to look for events in that city and Uber [14] to simulate the cost of getting there from user's location.
- DarkSky [22] as a source of meteorological information and cityBikes [8] to look for bicycle rental service providers in the city.

4.3 API analysis

In this section, the information providers will be analyzed one by one and drawing conclusions about the adaptions needed to expose those APIs to consumers.

4.3.1 IPinfo.io

This API is going to be used to locate the user by IP address.

API data

Host	https://ipinfo.io
Auth	Header API-Key
Name	iplocation

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/:ip/geo		/cities/ip/:ip	

We want to abstract the user from the API authentication so we will need to adapt the request to **add headers** for authentication purposes.

API response

```
{
  "ip": "83.51.33.167",
  "city": "Madrid",
  "region": "Madrid",
  "country": "ES",
  "loc": "40.4143,-3.7016",
  "postal": "28039"
}
```

We are only interested in the city, so the adaptation would be **filtering** the "city" field from the original answer.

4.3.2 LocationIQ

This API is going to be used to locate the user by geographic coordinates.

API data

Host	https://eu1.locationiq.com
Auth	Query token
Name	geolocation

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/v1/reverse.php	key: token lat: latitude lon: longitude format: format	/cities/geo	lat: latitude lon: longitude

We want to abstract the user from the API authentication and response format so we will need to adapt the request to **add parameters** for authentication and formatting purposes.

API response

```
// Response to: https://eu1.locationiq.com/v1/reverse.php?lat=40.4143&lon=-3.7016&
    format=json
{
  "place_id": "333837290613",
  "lat": "40.414099",
  "lon": "-3.70154",
  "display_name": "Fonda de San Sebastian, Cortes, Madrid, Madrid, Spain",
  ... //More attributes omitted for simplicity
  "address": {
    "name": "Fonda de San Sebastian",
    "neighbourhood": "Cortes",
    "city": "Madrid",
    "state": "Madrid",
    "country": "Spain",
    "country_code": "es"
  }
```

```
}
```

We are only interested in the city, so the adaptation would be **extracting** the "address.city" field from the original answer.

4.3.3 Zomato

This API will be used to find restaurants and daily menus for a given city.

API data

Host	https://developers.zomato.com/api/
Auth	Header API-Key
Name	zomato

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/v2.1/cities	q: query	/cities	q: query
/v2.1/search	entity_id: city id entity_type="city"	/restaurants	entity_id: city id
/v2.1/dailymenu	res_id: restaurant id	/dailymenu	res_id: restaurant id

The restaurant query will be only by city, so we will **add parameters** for entity type and **add headers** for authentication.

API responses

/v2.1/cities

```
// Response to: https://developers.zomato.com/api/v2.1/cities?q=New%20york
{
  "location_suggestions": [
```

```
  {
    "id": 280,
    "name": "New York City, NY",
    "country_id": 216,
    "country_name": "United States",
    "country_flag_url": "https://b.zmtcdn.com/images/countries/flags/country_216.png",
    "should_experiment_with": 0,
    "discovery_enabled": 1,
    "has_new_ad_format": 0,
    "is_state": 0,
    "state_id": 103,
    "state_name": "New York State",
    "state_code": "NY"
  },
  // More cities with same format, omitted for simplicity
  ]
}
```

We are only interested in the id and city name, so the adaptation would be **filtering** the search results in order to avoid transmitting unnecesary information.

/v2.1/search

```
// Response to: https://developers.zomato.com/api/v2.1/search?entity_id=280&
    entity_type=city&count=20
{
  "results_found": 42626,
  "results_start": 0,
  "results_shown": 20,
  "restaurants": [
  {
    "apikey": "***",
    "id": "16769546",
    "name": "Katz's Delicatessen",
    "url": // Long URL, omitted for simplicity,
    "location": // Location object, not interesting,
    "switch_to_order_menu": 0,
    "cuisines": "Sandwich",
    "average_cost_for_two": 30,
    "price_range": 2,
    "currency": "$",
    "offers": [],
    "opentable_support": 0,
    "is_zomato_book_res": 0,
    "mezzo_provider": "OTHER",
    "is_book_form_web_view": 0,
    "book_form_web_view_url": "",
    "book_again_url": "",
    "thumb":// Long URL, omitted for simplicity,
    "user_rating": // Rating object, not interesting,
```

```
    "photos_url": // Long URL, omitted for simplicity
    "menu_url": // Long URL, omitted for simplicity,
    "featured_image": // Long URL, omitted for simplicity,
    "has_online_delivery": 0,
    "is_delivering_now": 0,
    "has_fake_reviews": 0,
    "include_bogo_offers": true,
    "deeplink": "zomato://restaurant/16769546",
    "is_table_reservation_supported": 0,
    "has_table_booking": 0,
    "events_url":// Long URL, omitted for simplicity,
    "establishment_types": []
  },
  // More restaurants with same format, omitted for simplicity
  ]
}
```


We are only interested in the id name and cuisines of the restaurant, so the adaptation would be **filtering** the search results in order to avoid transmitting unnecesary information.

/v2.1/dailymenu


```
// Response to: https://developers.zomato.com/api/v2.1/dailymenu?res_id=66699
{
  "daily_menu": [
  {
    "daily_menu_id": "16507624",
    "name": "Vinohradsky pivovar",
    "start_date": "2016-03-08 11:00",
    "end_date": "2016-03-08 15:00",
    "dishes": [
    {
      "dish_id": "104089345",
      "name": "Tatarak ze sumce s toustem",
      "price": "149 Kc"
    }
    ]
  }
  ]
}
```


There is no need to adapt this response in any way, all the information and its structure is useful as it is.

4.3.4 Yummly

This API will be used to search for nutritional info for a given recipe.

API data

Host	http://api.yummly.com
Auth	Header User Password
Name	nutrition

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/v1/api/recipes	q: query	/recipes	q: query
/v1/api/recipe/:id		/recipes/:id	

We want to abstract the user from the API authentication so we will need to adapt the request to **add headers** for authentication purposes.

API responses

/v1/api/recipes

```
// Response to: http://api.yummly.com/v1/api/recipes?q=onion
{
  "attribution": {}, //Omitted
  "totalMatchCount": 39,
  "facetCounts": {},
  "matches": [
  {
    "attributes": {}, //Omitted
    "flavors": {},
    "rating": 4.6,
    "id": "Vegetarian-Cabbage-Soup-Recipezaar",
    "smallImageUrls": [],
    "sourceDisplayName": "Food.com",
    "totalTimeInSeconds": 4500,
    "ingredients": [] //Ingredients omitted list,
```

```
    "recipeName": "Vegetarian Cabbage Soup"
  },
  ... // More recipes with same format
  ],
  "criteria": {} // A lot of search criteria infor omitted.
}
```

In this response we are only interested in the recipes and within them, only the id and name. So there is a need for **extracting** and **filtering** the response.

/v1/api/recipe/:id

```
// Response to: http://api.yummly.com/v1/api/recipe/Hot-Turkey-Salad-Sandwiches-
    Allrecipes
{
  "attribution": {},  // omitted
  "ingredientLines": [
  "2 cups diced cooked turkey",
  ... // More ingredients
  ],
  "flavors": {},// omitted
  "nutritionEstimates": [
  {
    "attribute": "ENERC_KCAL",
    "description": "Energy",
    "value": 317.4,
    "unit": {
      "name": "calorie",
      "abbreviation": "kcal",
      "plural": "calories",
      "pluralAbbreviation": "kcal"
    }
  },
  ... // More nutrition attributes with the same format
  ],
  "images": [],// omitted
  "name": "Hot Turkey Salad Sandwiches",
  "yield": "6 servings",
  "totalTime": "30 Min",
  "attributes": {},
  "totalTimeInSeconds": 1800,
  "rating": 4.44,
  "numberOfServings": 6,
  "source": {},// omitted
  "id": "Hot-Turkey-Salad-Sandwiches-Allrecipes"
}
```

In this response we are only interested in the name and the nutrition estimates. So there

is a need for **filtering** the response.

4.3.5 Ticketmaster

This API is going to be used to find events for a given city.

API Data

Host	https://app.ticketmaster.com/discovery/
Auth	Query API-Key
Name	ticketmaster

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/v2/events.json	size: page size page: page number city: city name apikey: API Key	/events	size: page size page: page number city: city name

We want to abstract the user from the API authentication so we will need to adapt the request to **add parameters** for authentication purposes.

API responses

/v2/events.json

```
// Response to: https://app.ticketmaster.com/discovery/v2/events.json?size=1&apikey
    =***&city=Madrid&locale=es
{
  "_embedded": {
    "events": [
    {
      "name": "Backstreet Boys",
      "type": "event",
      "url": "https://www.ticketmaster.es/event/backstreet-boys-entradas/14031",
      "dates": {
```

```
        "start": {
          "localDate": "2019-05-13",
          "localTime": "21:00:00",
        },
        "timezone": "Europe/Madrid",
      },
      "priceRanges": [
      {
        "type": "standard including fees",
        "currency": "EUR",
        "min": 61.5,
        "max": 117.5
      }
      ],
      "_embedded": {
        "venues": [
        {
          "name": "WiZink Center",
          "postalCode": "28009",
          "city": {
            "name": "Madrid"
          },
          "state": {
            "name": "Madrid"
          },
          "country": {
            "name": "Espana",
            "countryCode": "ES"
          },
          "address": {
            "line1": "Av. Felipe II, s/n"
          },
          "location": {
            "longitude": "-3.6758",
            "latitude": "40.42394"
          }
        }
        ],
      }
    }
    ]
  },
  "page": { //Pagination info
    "size": 1,
    "totalElements": 166,
    "totalPages": 166,
    "number": 0
  }
}
```


A good amount of the irrelevant data have been omitted from the response. Basically

there is an intense work of **filtering** to do.

4.3.6 Uber

This API is going to be used to estimate how much it would cost, in time and money, get to the events consulted on Ticketmaster's API

API data

Host	https://api.uber.com
Auth	Header API-Key
Name	uber

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/v1.2/estimates/price	start_latitude start_longitude end_latitude end_longitude	/estimate	start_latitude start_longitude end_latitude end_longitude

We want to abstract the user from the API authentication so we will need to adapt the request to **add headers** for authentication purposes.

API responses

/v1.2/estimates/price

```
// Response to: https://api.uber.com/v1.2/estimates/price?start_latitude=40.452527&
    start_longitude=-3.726773&end_latitude=40.420647&end_longitude=-3.685160
{
  "prices": [
  {
    "localized_display_name": "UberX",
    "distance": 4.47,
    "display_name": "UberX",
```

```
    "product_id": "19f4a090-ce5b-44c5-8b43-35d35b7eb3bc",
    "high_estimate": 14,
    "low_estimate": 11,
    "duration": 1140,
    "currency_code": "EUR"
  },
    //More products with same format
  ]
}
```

There is no much need for adaption apart from **filtering** some useless information.

4.3.7 DarkSky

This API is going to be used for consulting a weather forecast for a given city.

API data

Host	https://api.darksky.net
Auth	URI API-Key
Name	weather

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/forecast/:key/:latlong		/forecast/:latlong	

This API has a peculiar authentication scheme, in which the key is informed as substring of the resource's URI. As we have seen in previous sections, the configuration syntax allows to define an upstream route, in which we have hardcoded the key.

API responses

/forecast/:key/:latlong

```
// Response to: https://api.darksky.net/forecast/**apikey**/40.463946,%20-3.714540
{
```

```
  "latitude": 40.463946,
  "longitude": -3.71454,
  "timezone": "Europe/Madrid",
  "currently": {}, // Current weather
  "hourly": {}, // Hourly info
  "daily": {
    "summary": "Light rain today through Wednesday, with high temperatures rising to 6
        8F on Monday.",
    "icon": "rain",
    "data": [
    {   //Omitted some of the not interesting fields
      "time": 1555538400,
      "summary": "Rain in the morning and evening.",
      "icon": "rain",
      "sunriseTime": 1555565601,
      "sunsetTime": 1555613836,
      "precipType": "rain",
      "temperatureHigh": 52.92,
      "temperatureHighTime": 1555606800,
      "temperatureLow": 47.67,
      "temperatureLowTime": 1555653600,
      "temperatureMin": 48.8,
      "temperatureMinTime": 1555560000,
      "temperatureMax": 59.26,
      "temperatureMaxTime": 1555538400,

    },
    // More days with same format
    ]
  },
  "flags": {}, // Sources info
  "offset": 2
}
```


In this response we are only interested in the daily forecast information and within it, only a subset of the attributes. So there is a need for **extracting** and **filtering** the response.

4.3.8 CityBikes

This API is going to be used for consulting services of bicycle renting and their stations location for a given city.

API data

Host	http://api.citybik.es
Auth	None
Name	bikes

Resource Mapping			
Upstream		**Exposed Route**	
Resource	**Params**	**Resource**	**Params**
/v2/networks		/networks	city: city name
/v2/networks/:networkid		/stations/:networkid	

This API does not give any filtering or searching option, so there is a need for implementing one as a gateway plugin.

API responses

/v2/networks

```
// Response to: http://api.citybik.es/v2/networks
{
  "networks": [
  ...,
    {
    "company": "Empresa Municipal de Transportes de Madrid, S.A.",
    "href": "/v2/networks/bicimad",
    "id": "bicimad",
    "location": {
      "city": "Madrid",
      "country": "ES",
      "latitude": 40.4168,
      "longitude": -3.7038
    },
    "name": "BiciMAD"
    },
  ... //More networks with same format
  ]
}
```

This response consists of a list of companies that provide bicycle rental services. It is

not parameterized in any way to be able to only consult those in the target city, so a **search plugin** should be implemented. In addition, due to the response being basically a heavy query of an absolute list, it is very likely that this route would benefit from the use of the **cache** of the gateway.

/v2/networks/:networkid

```
// Response to: http://api.citybik.es/v2/networks/bicimad
{
  "network": {
    "company": "Empresa Municipal de Transportes de Madrid, S.A.",
    "href": "/v2/networks/bicimad",
    "id": "bicimad",
    "location": {
      "city": "Madrid",
      "country": "ES",
      "latitude": 40.4168,
      "longitude": -3.7038
    },
    "name": "BiciMAD",
    "stations": [
      ...,
      {
      "empty_slots": 9,
      "extra": {
        "address": "Puerta del Sol n 1",
        "light": "yellow",
        "number": "1a",
        "online": true,
        "slots": 24,
        "uid": 1
      },
      "free_bikes": 15,
      "id": "8bc9ec4cbd6b8f27714636c87065e393",
      "latitude": 40.4168961,
      "longitude": -3.7024255,
      "name": "Puerta del Sol A",
      "timestamp": "2019-04-18T16:01:00.173000Z"
      },
      ... //More networks with same format
    ]
  }
}
```

This response, despite being large, does not seem to have much irrelevant information to omit. Also it contains bicycle availability information, so its not suitable for a the caching system, due to the risk of offering outdated information.

4.4 Summary and planning

Below is a summary of the routes that will be available in the gateway and what adaptation needs they have, in order to draw a common factor from the necessary work and design the plugins. The routes listed are the part of the URL we would add to the domain of the gateway to access the resource, and have the format **/:api/:version/:resource**.

API	**API level adaption**
Ipinfo.io	Add auth headers
LocationIQ	Add auth query
Zomato	Add auth headers
Yummly	Add auth headers
Ticketmaster	Add auth query
Uber	Add auth headers
DarkSky	Add auth URI param

Route	**Route level adaption**
/iplocation/v1/cities/ip/:ip	Pick fields
/geolocation/v1/cities/geo	Pick fields
/zomato/v1/*	Pick fields
/zomato/v1/dailymenu	Omit fields
/nutrition/v1/recipes	Extract field, Pick fields
/nutrition/v1/recipes/:id	Pick fields
/ticketmaster/v1/events	Pick fields
/uber/v1/estimate	Omit fields
/weather/v1/forecast/:latlong	Extract field, Pick fields
/bikes/v1/networks	Search result, cache

Studying the needs of adaptation it can be clearly seen that there are common requirements to a good part of the APIs or resources, and other specific requirements for a particular use case. The following plugins are planned trying to address the transversal needs in a generic way with the purpose of reusing that effort.

Generic plugins

- **Transformer**. Responsible for the mutation of data in the request or response.
- **Filter**. Responsible for selecting a subset of attributes in the response.
- **Extractor**. Responsible for repositioning the root of the response by navigating in depth.

Specific plugins

- **Bicycle service finder**. Responsible for finding companies operating in a city.

4.5 Plugin development

This section will detail the design decisions that have been made for the development of the necessary plugins to carry out the case study.

4.5.1 Transformer

The transformer is the most versatile plugin of those that are going to be developed. The most common needs when publishing APIs in the gateway are usually the addition, replacement or removal of some attribute in the request or response, either in the headers (e.g. for authentication), in the parameters (e.g. to force part of the query) or in the body (e.g. to transform the data model).

The approach to this plugin will be made in such a way that it is easy to perform a simple transformation, giving sufficient power in the configuration of the plugin itself to reflect these changes without leaving the API configuration file, but also allowing the possibility of implementing arbitrarily complex transformations through a reference to a file containing its logic.

Configuration syntax

```
{
  phase#destination:{
    operation:{
      key:value
    }
    adaption_path: "/path/to/adaption/file"
  }
}
```

- **phase#destination**. Specifies where the transformation will be applied, the phase values can be 'request' and 'response' and the destination values 'header', 'query' and 'body'.

- **operation**. Specifies which simple operation to perform. Its values can be 'add', 'replace' or 'remove'.

- **adaption_path**. You can specify a transformation described in a file, the file must exist inside a dedicated directory of the project and it must export a function that

will be executed by passing as argument the object that is going to suffer the transformation. Its return value will replace the initial object.

Once the transformer is implemented, in theory all needs could be solved as transformations by file. However, the purpose of generic plugins is to give the power to fully describe an API in a file in the most declarative way possible.

4.5.2 Filter

The filter addresses the need to select a subset of the attributes of a response, to eliminate irrelevant information and reduce the size of the response. Depending on the proportion of attributes to be selected to the total number of them in the response, we conveniently choose to talk about choices or omissions, and therefore this plugin must support both functionalities.

In addition, although this has not been the case, it is also possible that the information we want to include/exclude is distributed in different depths of the response, so we need a syntax that allows us to navigate through an object and make unequivocal reference to the attributes.

For this, JSONPath has been chosen, a standard that proposes a notation with which we can refer to a navigation through a JSON object in serialized form.

Configuration syntax

```
{
  operation: "type",
  paths:[
    "paths.to.some[*].attributes"
  ]
}
```

- **type**. Specifies whether the filtering will be positive or negative mask, the values can be 'pick' or 'omit'.

- **paths**. Collection of references to response attributes in JSONPath format

4.5.3 Extractor

The extractor may seem like a very concrete need looking at the case study, but it is a requirement that is often given quite a lot when trying to publish APIs. Not only are we interested in a subset of the answer, but it is a single attribute and it is within an unnecessary nesting for our purposes although it makes sense in the original answer. So it is not enough to filter this attribute, but we must also raise it to first level so that its value becomes the complete answer.

In the same way as before, we need a unique and unambiguous reference to this attribute, so we use JSONPath again.

Configuration syntax

```
{
  path: "attribute.to.extract"
}
```

- **path**. The path within the response to the attribute whose value will become the response.

4.5.4 Bicycle service finder

Sometimes a specific need arises that only makes sense in the context of the API that is being published, and is hardly usable in future publications. In this case we need a plugin that operates when a consumer wants to consult the companies that rent bicycles in a city, and yet the background system does not provide any mechanism for filtering, it only returns a list of all those it knows.

So a plugin is developed that in the request phase does nothing, but in the response phase uses the information from the request to filter only the results that match the name of the queried city. In this manner, we have provided additional functionality to an API that does not bring it by default. It is not necessary to provide anything in the configuration since it is not a generic component and all the necessary information is contained in the query itself, which all plugins have access to at runtime.

4.6 API config files

Once the APIs adaptation requirements have been analyzed and the plugins to satisfy them have been developed, the APIs must be published in the gateway. To do this, we will write a configuration file for each API following the scheme described in previous sections, in which we will define the general characteristics of the API and the use of the transversal plugins, as well as the routes that will be exposed under that domain and the use of the dedicated plugins.

4.7 System validation

Once the API configuration files have been written, we only need to use them to publish the APIs in the gateway and prove that everything works correctly. We will describe the process of setting up a test scenario in order to validate the work done so far.

4.7.1 Validation setup

In order to test the analysis and development performed in the case study, it' s necessary to set up a proper test scenario, as similar as possible to a production one.To this end, it has been relied on the containerization provided by Docker, using public official images to launch containers that execute the necessary tools to deploy a fully functional instance of the gateway.

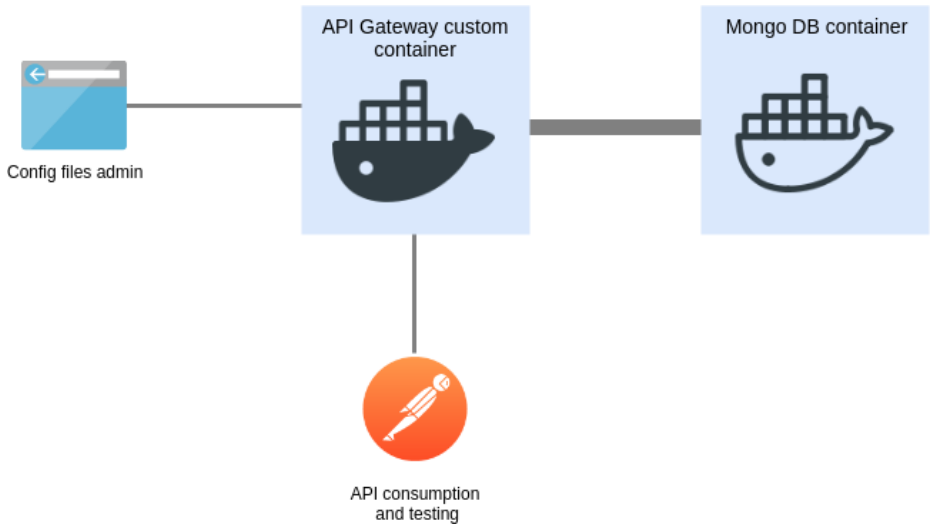


Figure 4.2: Validation setup scheme

On one side, the public image of MongoDB has been used, which has been very useful to boot up a database instance with good reliability practically effortless. Also, a docker image has been created containing the source code of the application, and built on top of a public image that provides a functional environment of Nodejs in the required version.

In this scenario, the role of the gateway user (publisher) will be performed using a browser that executes the backoffice MVP web application served by the gateway's instance container. On the other hand, the APIs will be tested with a Postman instance that will act as the consumer of the APIs exposed.

4.7.2 API publication

The first thing to do once the test scenario is operational is to publish the case study configuration files to the gateway. Although traditionally this was done by operating the database manually, in order to exemplify its use a slightly better, a quick prototype of a web interface has been developed, as a minimum viable product of the backoffice application that could be developed to manage the gateway. We leave this as a possible task to be done in the future and will be discussed further in the next chapter.

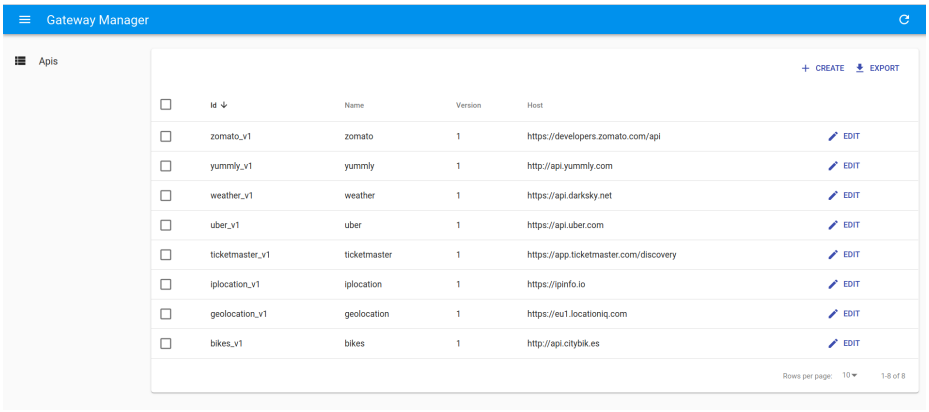


Figure 4.3: Publication web interface

This web interface consists of an administration panel that manages the configuration files as generic resources, and that allows the usual operations of any storage system: create, read, update and delete. Because configuration files are stored as documents in a MongoDB collection, these operations are nothing more than an adaptation of an HTTP API to the usual interface of a database.

4.7.3 API testing

Once the APIs have been published in the gateway by loading the configuration files into the database, it is necessary to check if they are available for consumption, and if they are being adapted properly. To do this, Postman has been used as an API testing environment, creating a collection in which a request is stored for each route exposed, in which a known call is made.

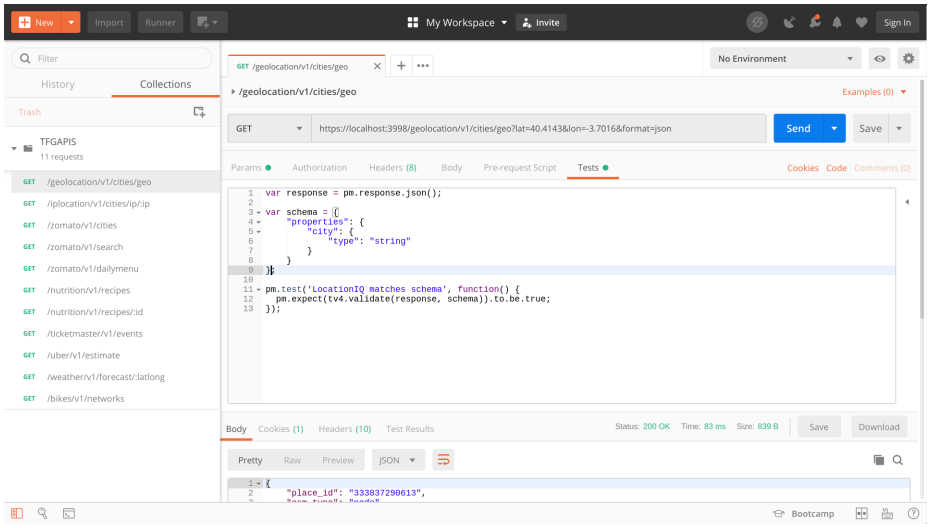


Figure 4.4: Testing suite in Postman

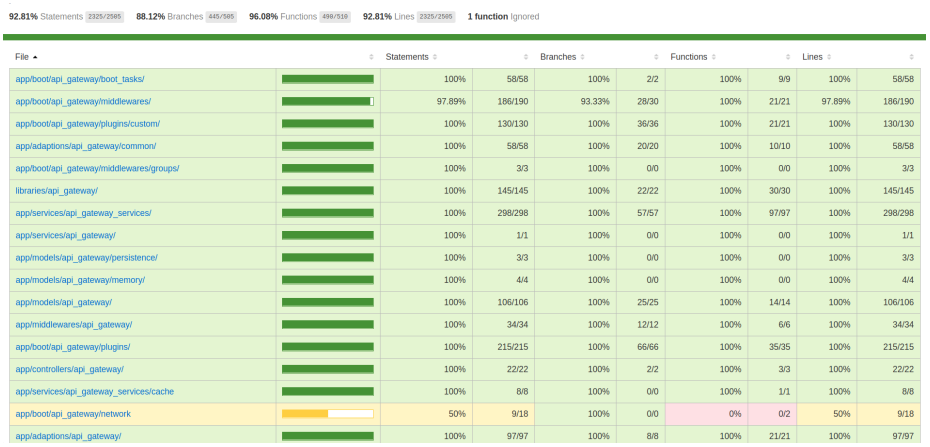
Postman allows us to include tests in the actual request, in order to automatically check the API status at any time. To do this, it offers the user a JavaScript editor, including a simple yet powerful testing framework for analyzing specific aspects of the response or validating its format. In this case the tests have been developed making use of the schema validation functions, therefore we not only check that the API is available, but also that the response conforms to a data structure and that the plugins have done their function.

When we already have a complete collection of tests that helps to validate the resources exposed by the gateway in our case study, we can export this collection in a JSON file, which can be used to include these tests as an automatic process in our development cycle. Postman offers a headless client called “newman”, which with a CLI command can run the entire test suite and generate a report that can be used as a validation phase in a continuous integration pipeline.

4.7.4 JavaScript unit testing

Although the unit testing process has been performed during the development of the project, it seems appropriate to talk about it in a validation section, although at this point in time the tests have been finished by far. This testing effort has been very useful to ensure a certain level of correctness of the software during development, offering a comfortable level of confidence that we were not impacting negatively the rest of the system while developing a new feature.

To do this, we have used the tools mentioned at the beginning of the document. Using Mocha.js, Chai.js and Sinon.js we have described a test suite that verifies the components of the system with a high enough level of reliability not only to justify its elaboration, but also to credit them an appreciable contribution to the development speed.



92.81% Statements 2325/2505 **88.12%** Branches 445/505 **96.08%** Functions 490/510 **92.81%** Lines 2325/2505 **1 function** Ignored

File		Statements		Branches		Functions		Lines	
app/boot/api_gateway/boot_tasks/		100%	58/58	100%	2/2	100%	9/9	100%	58/58
app/boot/api_gateway/middlewares/		97.89%	186/190	93.33%	28/30	100%	21/21	97.89%	186/190
app/boot/api_gateway/plugins/custom/		100%	130/130	100%	36/36	100%	21/21	100%	130/130
app/adaptions/api_gateway/common/		100%	58/58	100%	20/20	100%	10/10	100%	58/58
app/boot/api_gateway/middlewares/groups/		100%	3/3	100%	0/0	100%	0/0	100%	3/3
libraries/api_gateway/		100%	145/145	100%	22/22	100%	30/30	100%	145/145
app/services/api_gateway_services/		100%	298/298	100%	57/57	100%	97/97	100%	298/298
app/services/api_gateway/		100%	1/1	100%	0/0	100%	0/0	100%	1/1
app/models/api_gateway/persistence/		100%	3/3	100%	0/0	100%	0/0	100%	3/3
app/models/api_gateway/memory/		100%	4/4	100%	0/0	100%	0/0	100%	4/4
app/models/api_gateway/		100%	106/106	100%	25/25	100%	14/14	100%	106/106
app/middlewares/api_gateway/		100%	34/34	100%	12/12	100%	6/6	100%	34/34
app/boot/api_gateway/plugins/		100%	215/215	100%	66/66	100%	35/35	100%	215/215
app/controllers/api_gateway/		100%	22/22	100%	2/2	100%	3/3	100%	22/22
app/services/api_gateway_services/cache		100%	8/8	100%	0/0	100%	1/1	100%	8/8
app/boot/api_gateway/network		50%	9/18	100%	0/0	0%	0/2	50%	9/18
app/adaptions/api_gateway/		100%	97/97	100%	8/8	100%	21/21	100%	97/97

Figure 4.5: Coverage report

With the tools provided by these libraries, it is possible to keep track of the amount of code executed in the tests, and the percentage of this to the total. Also, the coverage reports that are generated are very useful to review which lines of code are not tested yet and which branches have not been executed at all.

4.8 Use example

As we already introduced in the presentation of the case study, the idea is to propose a hypothetical situation in which it is intended to cover the needs of a web application that needs to use the discussed APIs to provide a leisure information service located in the user's city, to show how the gateway would be used to adapt and serve those APIs to

the application. The APIs have already been analyzed and the gateway has been used to publish and adapt them.

In order to put into use the work carried out in the case study, an application prototype has been developed, consumer of the published APIs, which will make use of the information they provide to offer a simple dashboard of miscellaneous information about the user's location.

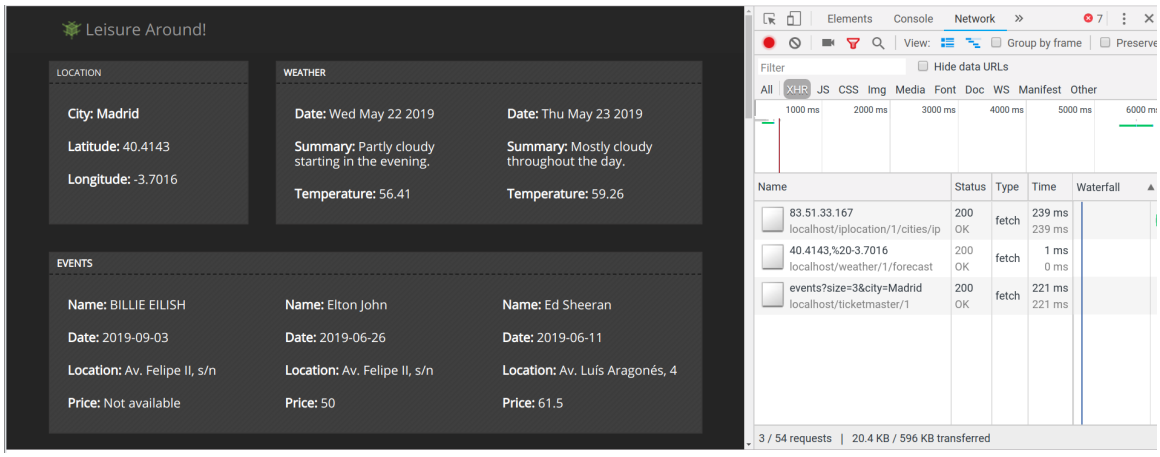


Figure 4.6: Gateway client application

Thanks to the use of the gateway, the application can access the information published by APIs of several origins from a single known domain, without dealing with individual authentication, and with a more appropriate data model for the use that will be given to that information. This is especially useful when there are several different client applications making use of the system, because it makes more noticeable the benefits of taking common factor of the access to third party network services into the gateway.

CHAPTER 5

Conclusions and future work

5.1 Conclusions

This paper details the process of designing and implementing a software solution that arises from a corporate problem that occurs more than we'd like. However, its approach has not been that of a patch to an uncomfortable situation, but to offer a stand-alone and multi-purpose solution to a need for decoupling between suppliers and consumers of data in web applications. While it is something that should not happen in theory, when requirements unrelated to the development environment dictate objectives and rhythms above all else, it is not difficult to see this asynchrony sooner or later.

Approaching it as a stand-alone application capable of solving a specific problem through only by configuration has been a success. It has made possible to adapt to a changing situation in both requirements and involved systems with a minimum workload in terms of software development. This enables other types of technological profiles to use the gateway without explicit knowledge of the project at the development level.

It has also been able to separate the responsibilities within the architecture of the gateway well enough so that the functional core has not undergone substantial changes since its first implementation, and so that an intelligent design of the necessary plugins

considerably reduces the difficulty curve when publishing a new API in the gateway, since at a certain point there will be enough developed plugins to solve most of the necessary adaptations with a composition of those.

5.2 Achieved goals

- **Multi-instance architecture:** A basic architecture has been planned and developed that has been constrained from scratch to be easily horizontally scaled. If enough resources are provided to the system, the performance impact of increased traffic should not become a problem, since the architecture is designed so that independent instances can be started on demand.
- **Design and implementation of the functional core of an API gateway:** All the functionality necessary to consume and adapt any HTTP API has been designed with a high level of abstraction. Using a system of generic plugins it is possible to provide a way to encapsulate the required logic to adapt APIs in a clean way but nevertheless allowing reuse.
- **APIs and plugin configuration syntax:** A data structure has been described which symbolizes the configuration of an API, containing all the necessary details for its publication in the gateway. This implies the logic necessary to validate and parse these configuration files.
- **Design and implementation of general purpose functional plugins:** The gateway has been provisioned with a handy set of generic plugins that cover the most common adaptation needs that have appeared during the use of this software. This greatly reduces the difficulty of publishing new APIs and minimizes the development load.

5.3 Future work

- **API Composition:** While the gateway currently allows the complete adaptation of an API, from its paths and parameters to completely changing the data model of the resources it provides, sometimes we are not convinced of how the resources are partitioned within the API or would like to offer another resource schema. In order to do this, it would be interesting to be able to compose APIs among them, to be able to publish new resources that require requests to several routes of the same API, or

even different APIs.

- **Secret management:** As we have seen in the case study, one of the gateway's use cases has been dealing with the multiple security implementations of each background system. In these cases it is the gateway which is authenticated against these systems using the API configuration as the data source, which include the authentication credentials. In order to allow users with different security profiles to use the gateway, it would be useful to implement some kind of secret management in order to refer to credentials without needing an explicit knowledge of them.

- **API management portal:** The gateway has made it possible to publish APIs with a reduced development work using only the configuration, to do this you just need to know how to write JSON and the syntax of the configuration file. Although this greatly reduces the technical knowledge barrier needed to publish to the gateway, it is still too high for it to be considered a friendly software. Ideally, we should have a tool that allows the development of this configuration in a graphical way and without technical knowledge other than the data model of the API to be adapted.

Bibliography

[1] Ramda docs. https://ramdajs.com/docs.

[2] Koa github repository. https://github.com/eggjs/egg/blob/master/docs/source/en/intro/egg-and-koa.md, 2019.

[3] Enrique Amodeo. *Learning behavior-driven development with JavaScript.* Packt Publishing Ltd, 2015.

[4] Mitchell Anicas. An introduction to oauth 2. https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2, 2018.

[5] Kent Beck. *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[6] Ethan Brown. *Web development with node and express: leveraging the JavaScript stack.* O'Reilly Media, Inc., 2014.

[7] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage.* O'Reilly Media, Inc., 2013.

[8] CityBikes. Citybikes api. https://api.citybik.es/v2/, 2019.

[9] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet computing*, 6(2):86–93, 2002.

[10] David Flanagan. *JavaScript: the definitive guide.* O'Reilly Media, Inc., 2006.

[11] Dick Hardt. The oauth 2.0 authorization framework. Technical report, 2012.

[12] AJ Field PG Harrison. *Functional programming.* Addison-Wesley Publishing Company, 1988.

[13] Postman Inc. Postman learning center. https://learning.getpostman.com/, 2019.

[14] Uber Technologies Inc. Uber api. https://developer.uber.com/, 2019.

[15] IPinfo. Ipinfo api. https://ipinfo.io/developers, 2019.

[16] Unwired Labs. Locationiq api docs. https://locationiq.com/docs, 2019.

[17] Zomato Media Pvt Ltd. Zomato api. https://developers.zomato.com/api, 2016.

[18] Mark Masse. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces.* O'Reilly Media, Inc., 2011.

[19] Keyur Patel and Mary Pat McCarthy. *Digital transformation: the essentials of e-business leadership*. McGraw-Hill Professional, 2000.

[20] Prabath Siriwardena. *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. Apress, 2014.

[21] Pedro Teixeira. *Professional Node. js: Building Javascript based scalable software*. John Wiley & Sons, 2012.

[22] LLC The Dark Sky Company. Darksky api. `https://darksky.net/dev`, 2019.

[23] Ticketmaster. Ticketmaster api. `https://developer.ticketmaster.com/`, 2019.

[24] James Turnbull. *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[25] Jim Wilson. *Node.js 8 the Right Way*. Pragmatic Bookshelf, 2018.

[26] Allen Wirfs-Brock. Ecmascript 2015 language specification. *Ecma International*, 2015.

[27] Yummly. Yummly api. `https://developer.yummly.com/`, 2018.