<TODO: Initial stuff>

Abstract

In an era where software products dominate everyday life, mechanisms for their efficient management and availability are essential. Application Programming Interfaces (APIs), particularly REST APIs, provide a widely adopted mechanism through which software components expose their functionalities to other components, applications or users. REST APIs define a set of endpoints, each exposing a specific functionality of the underlying software. However, the efficient and correct utilization of an API requires a thorough understanding of both the functionality and structure of each endpoint individually, as well as the interactions among them.

Often, API calls must adhere to a specific order, as invoking an endpoint correctly may require data returned from a call to another endpoint. Detecting these dependencies guides optimal API utilization and reveals valuable workflows. Although current documentation - such as OpenAPI specifications or Postman Collections - offers structural details, descriptions and examples, it lacks sufficient information to clarify endpoint interactions and dependencies.

This thesis first introduces an existing static method for identifying dependencies by analyzing Postman Collections. While this method successfully reveals such dependencies, it is constrained by the need for a Postman Collection - which is often lacking for an API - and the limited accuracy and completeness of its results. This study presents a new dynamic approach to identify dependencies by capturing and analyzing real-world API calls through a mediator system, thereby incorporating dynamic business logic insights into the analysis. This method significantly enhances result quality and enables the generation of dependency documentation even for APIs that are entirely undocumented. Finally, the dynamic method is integrated into the RADAR system, which provides an environment for visualizing and analyzing API dependencies identified through both static and dynamic analysis.

Acknowledgments

Contents

1 Introduction

In the era of the indisputable prevalence of software solutions in various aspects of everyday life, it is crucial for them to be easily and efficiently available in order to serve their functionalities and meet the needs that led to their development. As needs grow complex and applications are asked to satisfy complicated requirements, it is anything but uncommon for them to need services provided by other applications in order to enrich and enhance their functionality. Instead of ending up re-inventing the wheel and duplicate existing software solutions, applications could integrate and utilize these solutions so as to exploit their benefits minimizing the required effort and increasing the productivity during development [1]. Regardless of the type of the entities requiring access to a software solution - whether they are the developers who created it, the developers of a new solution (or a different component of the same solution) or even some clients - the need of a mechanism which is going to efficiently provide the services of a software component to any interested entity is evident.

Application Programming Interfaces (APIs) have become the predominant mechanism with which any application or software component can expose its functionalities to other services or users. Nowadays, APIs are highly adopted by many enterprises and are the building blocks of the most modern applications [2]. Enterprises can develop their applications by utilizing APIs to ensure their software solutions are accessible and enable different components of a single product to communicate with one another. At the same time, they can integrate other existing API-based software components in order to enhance the user experience, satisfy complex customer needs and actually focus on fast deliveries of genuinely innovative and valuable products rather than re-implementing existing so-

lutions.

Specifically, REST (Representational State Transfer) APIs have prevailed over their counterparts, like SOAP (Simple Object Access Protocol) and RPC (Remote Procedure Call) [3], due to their flexible and lightweight way to connect diverse components [12].

1.1 Comprehending APIs

It is indeed a fact that the existence of APIs revealed new opportunities in software development, as they are the fundamental mechanisms through which applications can make their functionalities available and as a result different software components can communicate with one another and integrate with each other. However, at the same time, it is important to consider some essential prerequisites through which this process can be carried out efficiently. Any software solution is useful not only when it satisfies the desired requirements but also when it is designed and built in such a way so it can be maintained, extended, integrated and easily utilized by other components or its end users. Consequently, it can be easily perceived that any API has to be entirely comprehensive to ensure proper and productive utilization.

Usually, APIs come with some documentation that contains information about their endpoints. This documentation can be a Postman Collection that provides the schema of the request and response body of every endpoint, descriptions and possibly some examples of API calls with real parameter values. It can also be an OpenAPI specification which also consists of similar information about the structure of the API. Both of these documentation tools focus on the representation of an API's structure considering each endpoint as an independent component of the whole system. However, every API endpoint is only a small piece of the business logic that is implemented by an application and thus, it is especially valuable to be able to ascertain how each endpoint participates in interactions with other endpoints so that they can collaboratively carry out this logic.

Inter-endpoint dependencies

For example, let us assume that an Online Payment API is available and we want to integrate it in our application in order to allow the users pay for our products. For this purpose, three endpoints are provided :

1. List products - to list all the available products
2. Make Order - to place a new order for a specific product
3. Pay - to process the payment for the specific order

If we are about to integrate this API in our application, it is really important to be aware of the structure of each endpoint and especially of the request body and the input parameters that are required for this endpoint to operate as expected. However, this is not enough. In order to make an order for a product we need to know the **id** of the selected product and to process the payment we need to define the **id** of the order to which this payment is related. As a consequence, it is important to understand that our application initially needs to call the first endpoint in order to obtain the identifiers of all the products from the response body, then select one of these identifiers and include it in the request body of the second endpoint to place a new order, thereby retrieving the identifier of the order from the response body and finally call the third endpoint including this order identifier in the request body. This particular sequence of API calls unveils some dependencies between the endpoints (inter-endpoint dependencies) that impose a strict order in which they should be invoked. The second endpoint is dependent on the first one as it needs an input parameter (in the request body) emitted by the response body of the first endpoint. For the same reason, the third endpoint is dependent on the second one.

This type of behavioral information is very beneficial when utilizing APIs as it uncovers some underlying patterns and possible workflows. Unveiling these patterns is very useful as it avoids blind decision making by the developers and thus, it prevents errors, enhances productivity and improves developer experience when integrating third party APIs or trying to expand existing APIs. Furthermore, in addition to the fact that understanding dependency patterns can offer a clear representation of the API's functionalities and possible user stories, it can also provide information about endpoints operating as bottlenecks as many other endpoints are dependent on them.

At the same time, existing API documentations do not contain information about inter-endpoint dependencies and as a result the process of integrating or expanding an API can be time-consuming, error-prone and inefficient.

1.2 Related work

The necessity to explore methods for extending the information encompassed within current API documentations and enrich it with insights about inter-endpoint dependencies has already been addressed in previous work. Specifically, this thesis aims to extend a previous research presented in <TODO: reference to the paper> . This research utilizes Postman Collections of APIs and tries to detect possible dependencies between different endpoints. To be more precise, the outcome of the aforementioned work is the development of RADAR (**R**EST **A**PI **D**ependencies and **A**nalysis of

Relationships), which can be accessed here. RADAR is a tool designed for detecting, analyzing and visualizing the inter-endpoint dependencies of an API, following a thorough and configurable examination of its Postman Collection. The result of this analysis is the depiction of inter-endpoint dependencies by employing an interactive directed graph. By examining this graph, a developer has the opportunity to better comprehend the functionalities of an API, the necessary data flows between endpoints and the attributes on which they are dependent.

1.3 Motivation

While RADAR is very helpful for extending the knowledge that can be gained from the current API documentation and thus, facilitate the process of utilizing and integrating APIs, it does come with two major limitations. First of all, this analysis is intimately connected with Postman Collections. In order for the dependency analysis to be carried out successfully, the Postman Collection of the API has to be available and, ideally, contain sufficient number of examples, in order for the analysis to take real parameter values into consideration. Although comprehensive Postman Collections can be available for many public or private APIs - presuming that a significant time has been dedicated to the preparation of this documentation material, in many cases developers are asked to utilize APIs which are utterly unfamiliar to them or poorly documented. In this case, a developer has to invest considerable time and effort, frequently encountering mistakes and oversights, so as to correctly and efficiently utilize or integrate such an API. At the same time, even if comprehensive Postman Collections enriched with many examples are available, the capabilities of that kind of static documents are limited.

RADAR examines the names and/or values of output and input parameters of each endpoint and detects matches. However, this information is static and even when examples are available, there is no guarantee that they have derived from a real interaction with the API. As a consequence, RADAR would be unable to identify some valuable dependencies as they would not emerge from the current examples. At the same time, comparing every pair of endpoints so as to identify dependencies while overlooking the potential realistic workflows and interactions between them, may lead to the discovery of many invaluable dependencies among semantically unrelated endpoints. Although RADAR offers valuable insights into the functionalities of APIs, the uncertain availability of the necessary documentation and the noise that is detected in the generated graphs as a result of the static nature of the analysis, provides the incentive to explore additional ways of inter-endpoint dependency detection.

To avoid conducting an analysis based on static information, this thesis

presents a new way of detecting dependencies based on real-world interaction with the API. This method is going to refine the aforementioned business logic agnostic dependency analysis, shaping it to consider and incorporate insights about the real-world utilization of an API. The objective is to extend RADAR so as to generate inter-endpoint dependency graphs derived from real-time API calls eliminating the necessity for a Postman Collection and reducing the noise resulted from the static analysis of the previous research.

1.4 Thesis structure

TODO

2
REST API Fundamentals

This section aims to **briefly** define the context in which this research is placed, acclimating the reader to the fundamental related topics. The information below make reference to core concepts of the current thesis, such as REST APIs over HTTP, JSON format, API calls parameters and API documentations.

2.1 Why REST

REST APIs, defined in 2000, are the common way of connecting software components as they are flexible and lightweight. The stateless nature of REST, according to which each new client's request is independent of all others and server does not save information about previous requests, makes this mechanism scalable by reducing the server load. Furthermore, REST's capability to enforce a clear separation between the client and the server by transparently decoupling different components and architecture layers along with its independence from the technology used on either server or client side, makes this mechanism highly flexible. Given its numerous advantages and wide adoption, this study utilizes REST APIs [7].

2.2 REST API over HTTP

As noted earlier, REST APIs represent the main mechanism that defines the rules according to which software components developed in any technology can communicate with each other. Enterprise applications can ac-

cess functionalities provided by other internal or third-party services in order to perform complex operations and improve the customers' satisfaction. API Clients can communicate with an API server with HTTP messages as presented below. A REST API exposes a set of endpoints, the invocation of which by an API client results in the execution of a specific functionality inside the API server.

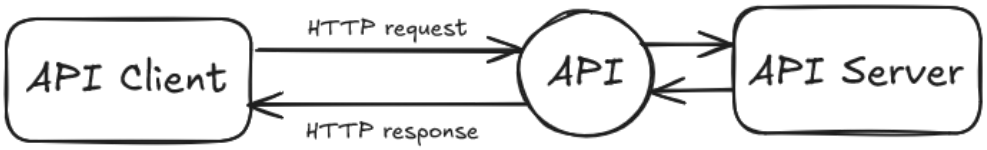


Figure 2.1: API Client-Server communication

Sending HTTP messages API Clients prepare and send an HTTP request to the server adhering to the rules set forth by any available API documentation. The request of the REST API Client asks for access to a specific resource designated by its unique URL (Uniform Resource Locator). HTTP messages use the generic message format according to which they consist of :

- a start line which is a request line with the request method, request URI and HTTP version for requests and status line with the HTTP version and a numeric and its corresponding textual status code for responses.
- zero or more header fields
- an empty line that identifies the end of the headers and
- possibly a message body [4]

The client defines the HTTP method which indicates the purpose for which the request was made. Among the various HTTP methods, the most frequently employed are the following.[5]

1. **GET**: This method returns the representation of the target resource to the client who can also define some filtering options in order to define some desired property values or other presentation preferences.
2. **POST**: This method is used so that the client can send data (the request payload) to the server for processing.
3. **PUT**: This method is used to totally replace the current representation of a resource on the server according to the request payload.

4. **PATCH** : These requests partially update a resource on the server by updating the value of some specific properties rather than replacing the entire resource. [6]

5. **DELETE**: This method is used to remove the target resource from the server.

The API Client also defines a set of request headers that are metadata determining various options exchanged between the client and the server [7]. For example, the client utilizes request header fields to provide information about the request format and the preferred format of the response, provide authentication credentials, make a request method conditional on the resource's modification date (If-Modified-Since header) as well as numerous others [5]. In response to the client's request, the server sends back an HTTP Response message which adheres to the generic message format described before and contains a numeric and textual status code. Status codes follow some conventions for the sake of consistency between various applications and the most frequently employed are the following.

- 200 OK: The request has succeeded
- 201 Created: The request was successful and resulted in a new resource being created.
- 204 No Content: The request was successful but did not need to return a body.
- 400 Bad Request: The request could not be understood due to malformed syntax.
- 401 Unauthorized: The request requires user authentication.
- 403 Forbidden: The server understood the request but is refusing to fulfill it.
- 404 Not Found: The required resource was not found
- 500 Internal Server Error: The server encountered an unexpected condition which prevented the fulfillment of the request.
- 502 Bad Gateway

as well as numerous others [4].

HTTP Request Parameters REST API Clients are able to encapsulate different types of parameters within their HTTP requests: path, query, body, cookie and header parameters [8]. Cookie and header parameters are not useful for the current research as they contain very specific information, but the initial three parameter types are very common and relevant to the current study.

1. **Path parameters**: they are variable parts included in the URL and they aim to identify a specific resource within a collection of similar resources (for instance, a user among multiple users):

   ```
   GET /users/{user_id}
   ```

2. **Query parameters**: They are included in the end of the URL after a question mark (?) and can be numbers, strings, arrays or objects. Different query parameters can be separated by ampersands (&).

   ```
   GET /users?joined_at=January&first_name=David
   ```

3. **Request and Reponse Body parameters** The most important and semantically rich parameters are those included into the body of the HTTP requests and responses. Request and response bodies contain the representation of the request payload or the required resource and they may be presented in either XML or JSON format. XML (eXtensible Markup Language) is a markup language that provides rules to define, store and exchange data by utilizing custom tags in order to represent the structure of the data [9]. JSON (JavaScript Object Notation) is a lightweight and language-independent data interchange format which defines rules for representing structured data [10]. At present, JSON format is the most widely used due to its simple and human readable syntax, fast and efficient parsing and flexibility [11] and as a result this study selects the utilization of this format. An

```xml
<user>
    <name>Dimitris</name>
    <address>
        <country>Greece</country>
        <city>Athens</city>
        <zipcode>12345</zipcode>
    </address>
</user>
```

(a) XML

```json
{
  "name": "Dimitris",
  "address": {
    "country": "Greece",
    "city": "Athens",
    "zipcode": "12345"
  }
}
```

(b) JSON

Figure 2.2: JSON vs XML formats

example demonstrating the differences in syntax between the two formats is presented in figure 2.2.

2.3 Documenting REST APIs

As mentioned before, proper and efficient utilization of APIs necessitates the availability of a comprehensive documentation. The two primary and widely adopted approaches for generating REST API documentations are the Postman Collections and the OpenAPI Specifications.

Postman Collections Postman is a platform for designing, building, utilizing and testing APIs and thereby streamlining the process of proper and efficient API management [13]. Postman Collections are sets of API requests that include information about each one of them, such as the schema of the request and response bodies, possible query and path parameters, headers, authorization details and examples demonstrating real API calls [14]. Postman Collections can be exported in JSON format and efficiently parsed so that they can be thoroughly analyzed. The resulting JSON file contains all relevant information for each API request, along with the available examples. Postman Collections of many public APIs are available through the Postman Public API Network.

OpenAPI Specification OpenAPI Specification (OAS) is an open standard for describing APIs using a simple and expressive language, available in both YAML and JSON format, which facilitates a shared understanding of the APIs among all stakeholders. OpenAPI specifications are beneficial in both Code-First and Design-First approaches, serving either to comprehend the design of the API to be implemented or to document an existing API. OpenAPI Specifications provide valuable support throughout all stages of the API lifecycle, including requirements gathering, design, development, deployment and testing [15].

API documentations serve as valuable static materials that aids in performing static API dependency analysis, as it will be addressed in the following section.

2.4 UML Sequence Diagrams

The current study on dynamic API analysis employs UML sequence diagrams to provide a comprehensive depiction of the real-world use cases for the APIs under examination. Therefore, a brief overview on how Visual Paradigm handles this diagrams is presented below. Use case diagrams

represent the sequence of messages and interactions between different components of the system in order to carry out a specific functionality. The role of an external entity (human user or external hardware) interacting with the system is represented by an actor. Every involved system's component has its own lifeline represented by a dashed vertical line and indicating the existence of the object throughout the use case execution. Call messages between different components lead to the invocation of an operation of the target lifeline and are represented by solid lines with an arrowhead, while return messages use a dashed line. The thin rectangles on a lifeline represent the period during which this component is active.

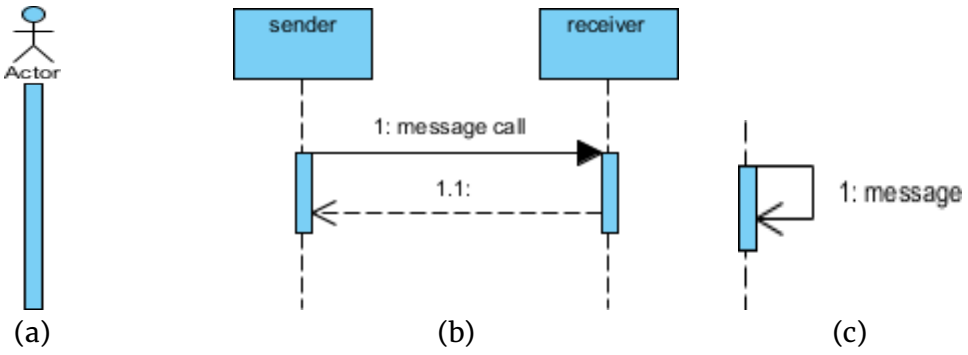


Figure 2.3: UML Sequence Diagram: Actors, lifelines and messages

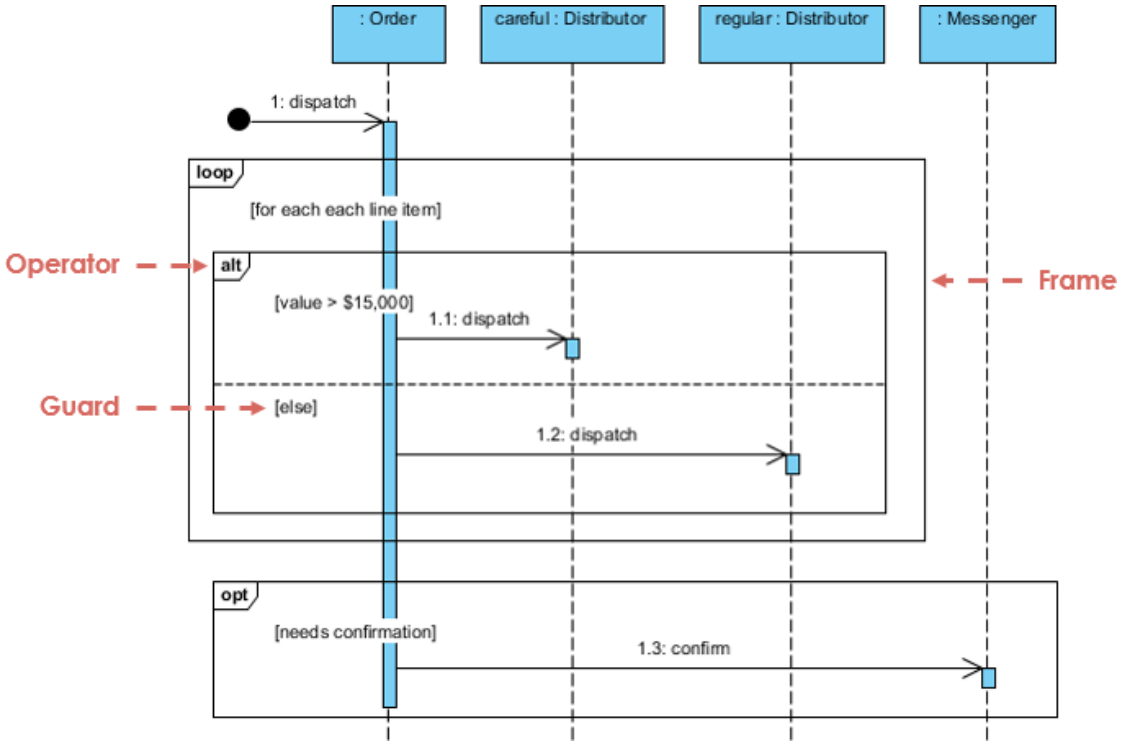


Figure 2.4: UML Sequence Diagram: Sequence Fragment

Self messages represent the invocation of operation on the same lifeline [Figure 2.3].

UML 2.0 also introduces sequence fragments that enclose a set of interactions characterized by specific behaviors, such as iterative (fragment operator **loop**) or conditional (fragment operator **alt**) execution, providing the capability to combine multiple operators [16, 17] [Figure 2.4].

3
Static API Dependency Analysis

As previously mentioned, the current work builds upon and expands an existing study that delved into the discovery of inter-endpoint dependencies in RESTful APIs so that to enrich the knowledge that a common documentation material can provide. In order to better understand why such information is necessary and why extensive research is needed to discover methods to enrich current documentations, it is essential to examine how APIs are utilized today.

3.1 Utilizing APIs

Today's complex needs necessitate more efficient and sophisticated architectures capable of supporting intricate applications that offer a wide range of diverse functionalities. In many cases, these architectures require the breakdown of a complex software solution into smaller interoperable components. These components have to efficiently interact with each other in order to achieve their purpose. This is where an API comes in to facilitate this process.

Enterprises utilize APIs so that they can build complex applications composed of multiple components. These components can be internal enterprise components as well as third-party external ones which are integrated into the enterprise's solution so as to enhance its functionality without the need to re-implement existing software components. Furthermore, enterprises can offer their products through APIs, allowing third parties to utilize them. In this way, their products become accessible, increasing popularity and profit.

However, to efficiently utilize, maintain and expand APIs, it is crucial to thoroughly comprehend all aspects of their functionalities, ranging from the structure and behavior of every single API endpoint to the interactions among all of them. While the structure of every endpoint is comprehensively presented in documentations such as Postman Collections and OpenAPI Specifications, insights regarding the interactions between them are not included. However, these insights are valuable when utilizing APIs as they contribute to a faster, more efficient, error-free and productive process. If such knowledge is available, developers can understand the order in which the endpoints should be invoked and be aware of the data to expect of previous calls in order to provide it in subsequent requests. This is precisely the motivation for the research upon which this thesis is based.

3.2 Inter-endpoint dependencies in-depth

An endpoint E_1 is dependent to an endpoint E_2 when the response of E_2 contain information that is needed in order to call successfully the endpoint E_1. When two endpoints are dependent, they have to be invoked in a specific order and the developer has to be aware of this order so as to integrate, maintain and build new functionalities upon the existing API. API documentations include valuable information about the two endpoints, such as the structure of their request and response bodies, other input parameters and headers, possible descriptions and many more. Nevertheless, there is no clear evidence indicating that the two endpoints are dependent and must be invoked in a specific order. To clarify the concept and importance of inter-endpoint dependencies, we examine this issue through an example of a hypothetical Flight Booking API that comprises three endpoints.

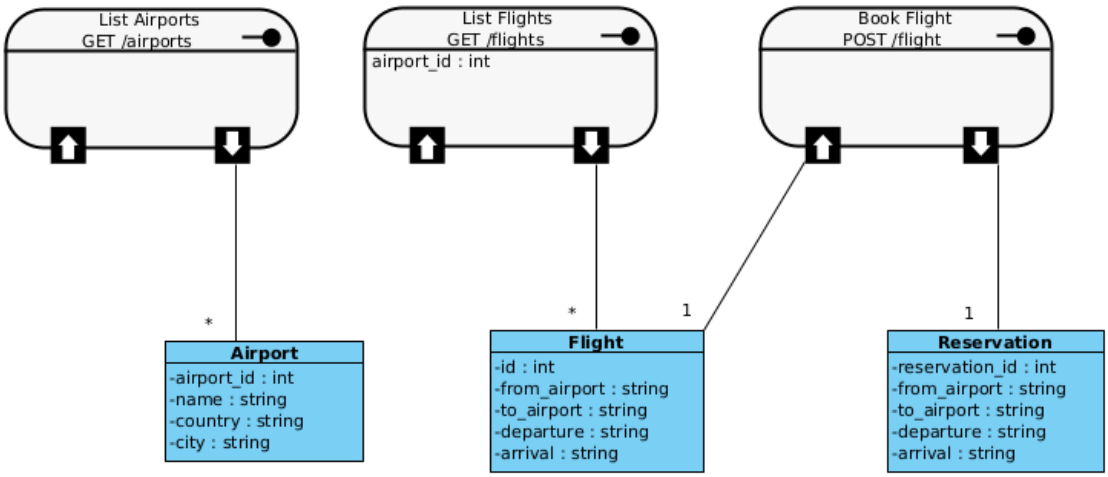


Figure 3.1: Flight Booking API

1. **List Airports**: This endpoint provides a list with the identifiers, names, country and city of all possible airports.
2. **List Flights**: This endpoints accepts the airport identifier as path parameter input and returns all the available flights for this airport.
3. **Book Flight**: This endpoint accepts flight information in its request body and returns the reservation identifier along with relevant flight information.

Examples of response bodies for each one of the three endpoints are presented in figure 3.2.


```
[
  {
    "airport_id": 1,
    "name": "Charles de Gaulle Airport",
    "country": "France",
    "city": "Paris"
  },
  {
    "airport_id": 2,
    "name": "Warsaw Chopin Airport",
    "country": "Poland",
    "city": "Warsaw"
  },
  {
    "airport_id": 3,
    "name": "Humberto Delgado Airport",
    "country": "Portugal",
    "city": "Lisbon"
  }
]
```


(a) List Airports Response


```
[
  {
    "id": 1,
    "from_airport": "Humberto Delgado Airport",
    "to_airport": "Warsaw Chopin Airport",
    "departure": "16 November 2024, 07:00 AM",
    "arrival": "16 November 2024, 12:00 PM"
  },
  {
    "id": 2,
    "from_airport": "Humberto Delgado Airport",
    "to_airport": "Charles de Gaulle Airport",
    "departure": "16 November 2024, 12:20 PM",
    "arrival": "16 November 2024, 15:55 PM"
  }
]
```


(b) List Flights Response


```
{
  "reservation_id": 1742,
  "from_airport": "Humberto Delgado Airport",
  "to_airport": "Warsaw Chopin Airport",
  "departure": "16 November 2024, 07:00 AM",
  "arrival": "16 November 2024, 12:00 PM"
}
```


(c) Book Flight Response

Figure 3.2: Flight Booking API

It becomes clear that successfully invoking the List Flights endpoint requires calling the List Airports endpoint to retrieve the identifiers of the available airports. Following this, one of these airport identifiers is selected and passed as a path parameter input to the List Flights endpoint. This relationship between the attribute airport_id of the List Airports response body and the path parameter of the List Flights endpoint is a repre-

sentative example of body-path inter-endpoint dependency and this observation indicates that the List Airports endpoint must be invoked prior to the List Flights endpoint. Furthermore, the successful invocation of Book Flight endpoint requires that the List Flights endpoint has been previously called and returned some flight identifiers. Thereafter, one of these flight identifiers is selected and utilized as a body parameter for the Book Flight endpoint. Similarly, this relationship indicates a body-body inter-endpoint dependency between List Flights and Book Flight endpoints which guides us that we firstly need to call List Flights endpoint and then call the Book Flight endpoint in order for the flight booking process to be successfully completed.

A third type of dependency that is also examined, yet not illustrated in the current example, is the body-query dependency according to which a value returned by the response body of an API call is utilized as a query parameter value in a different endpoint invocation. Both in this research and in the current thesis, all three cases are analyzed and taken into consideration, even though some are more susceptible to generating noise, which may result in the identification of non real dependencies.

It is also worth noting that two endpoints may exhibit dependencies on multiple attributes, thereby complicating both the inter-endpoint dependencies and our analysis. Nevertheless, for a thorough research, all of these dependencies are important, as they not only dictate the necessary order of endpoint invocations but also enhance our understanding of the data flow between various endpoints. Indeed, such an analysis can reveal not only dependencies between pairs of endpoints but also full dependency paths, showcasing a complete use case scenario for the API under examination.

3.3 RADAR design and implementation

The static analysis on which this thesis is based aimed precisely at enriching the understanding derived from the existing API documentations by identifying and analyzing dependencies such as those presented. The outcome was the development of RADAR (**R**EST **A**PI **D**ependencies and **A**nalysis of **R**elationships), which identifies, visualizes and facilitates the analysis of inter-endpoint dependencies by employing a directed graph where the nodes represent the endpoints and each edge is directed from E_1 to E_2 when E_2 depends on E_1, as the response body of E_1 contains information that is required by the input parameters of the endpoint E_2. In order to identify dependencies, RADAR accepts a Postman Collection as input and after a configurable thorough analysis, it generates the directed graph of inter-endpoint dependencies (body, path or query). The user of RADAR can isolate specific nodes for clearer results and view the attributes

on which two endpoints are dependent.

The Algorithm The algorithm employed by the RADAR serves as the foundation for the system to be implemented as part of this thesis. RADAR parses the Postman Collection and creates two sets, one for the request parameters and another for the response attributes. For every endpoint, all input parameters (body, path, query) are identified and for each parameter, its name, value and type are included in the set. The same process is applied to the attributes of the responses returned by all the endpoints. These two sets are then subjected to analysis, during which matches between input parameters and response attributes are identified. Specifically there are two ways of detecting dependencies. The first one identifies input parameters and output attributes that share the same name and type but belong to different endpoints while the second one is very similar but focuses on values rather than names and as a result it requires the availability of multiple Postman examples.

The system's architecture is illustrated in the following component diagram (figure 3.3) where DependencyGraph file conducts the analysis based on attribute values while DependencyGraphAttr file does so based on names. Once the Postman Collection is extracted, it undergoes dependency detection analysis and the results are forwarded and displayed by the Visualizer (RADAR UI). Furthermore, the analysis is configurable, allowing the user to select whether to detect or exclude dependencies that involve query/-path parameters and to filter only those dependencies that start from GET endpoints. Clearly, by selecting this option, some dependencies will be

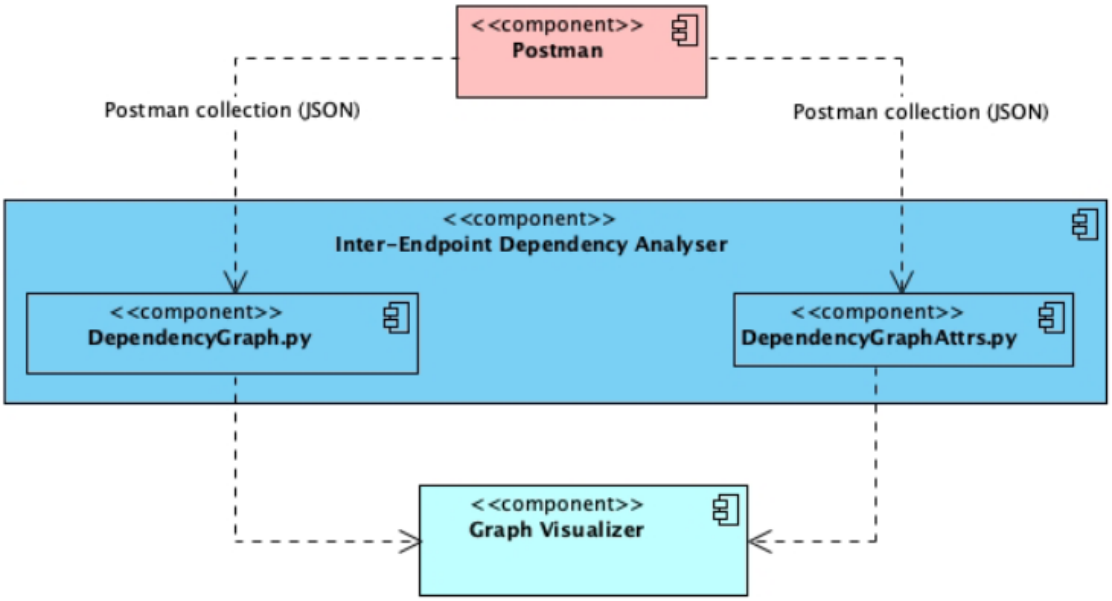


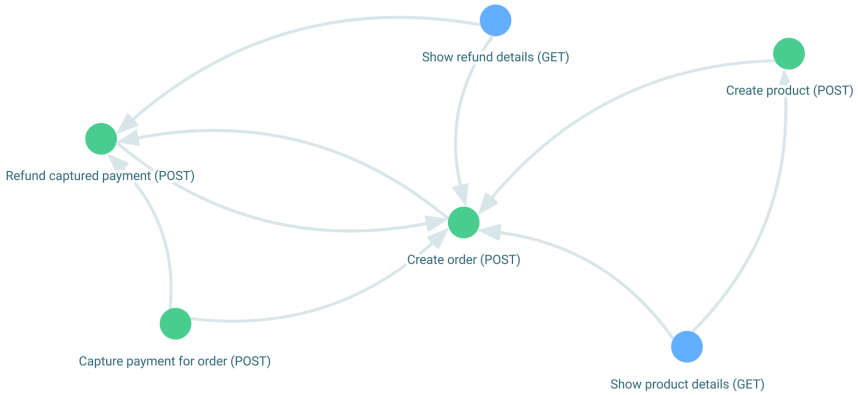
Figure 3.3: RADAR Component Diagram

excluded; however, this will also eliminate significant noise, as these dependencies are the most prevalent as previously shown (GET List Airports → GET List Flights → POST Book Flight).

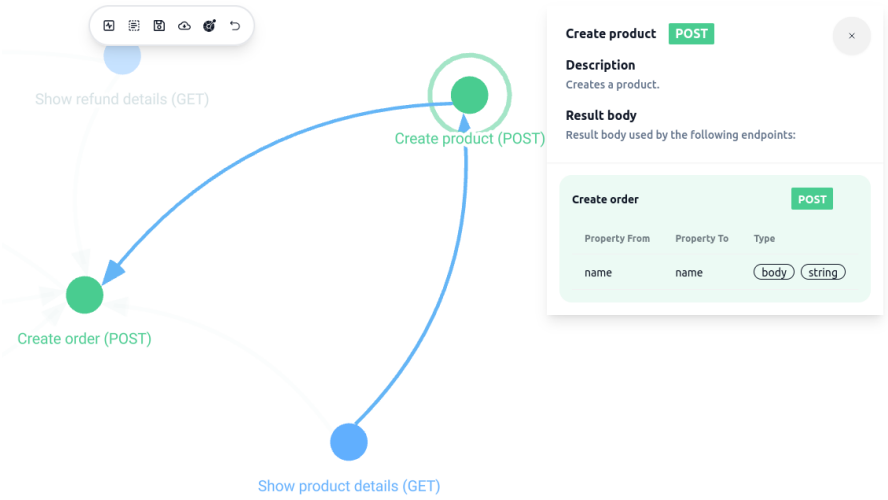
3.4 Examples

The following section presents and explains several examples of employing RADAR to detect and analyze inter-endpoint dependencies on available public APIs such as the PayPal API and the Notion API.

Paypal API consists of a rich API with many payment functions and features. A representative part of the dependency graph for a subset of PayPal API endpoints is illustrated in figure 3.4.



(a) Dependency Graph



(b) Attribute dependencies

Figure 3.4: RADAR: PayPal API - Products and Orders

Some notable dependencies identified in the graph are as follows:

- **Create Product** → **Create Order**: It is essential for a product to be created before adding a new order for that product. In figure 3.4b, it appears that the attribute on which these two endpoint are dependent is the "name". Specifically, the value of the "name" attribute (which is the name of the product) returned by the response body of Create Product is passed as an input body parameter ("type" field) to the request body of Create Order endpoint. This indeed, reflects a rational dependency. After creating a product, the name of that product has to be available in the request body of Create Order endpoint so as to generate a new order for the product having that name.
- **Capture Payment for order** → **Refund captured payment**: A refund is only possible after the payment has been captured successfully.
- **Create order** → **Refund captured payment**: A refund for an order payment is only possible after the order has been successfully created.

This graph also reveals a dependency path (Create product → Create order → Refund captured payment) that indicates a potential workflow. Understanding this workflow aids in both the integration of the API and comprehending a piece of business logic it implements.

Understanding the identified dependencies can significantly reduce the effort and time required for integrating the functionality presented in the previous graph. Nevertheless, that dependency graph illustrates several additional dependencies that are not semantically obvious and could potentially be misleading. This issue will be examined during the analysis of the limitations of this method. A second part of the dependency graph is presented in figure 3.5, where only the outgoing edges for the endpoint Show Invoices Details have been isolated. A valuable dependency detected in this graph is the one between Show Invoices Details and Record payment for invoice. It is important to view the details for an invoice so as to retrieve its identifier before processing a new payment for the invoice having that identifier. However, the significance of the remaining dependencies does not appear to be immediately apparent.

By employing the Notion API, users can create workspaces to perform CRUD operations on databases, manipulate and manage data and pages for organizing and managing information that needs to be shared among team members, as well as utilize it as a Content Management System for the maintenance and management of data. A part of the graph generated using the Notion API Postman Collection is presented in figure 3.6.
A useful workflow identified by RADAR is the following:
Create a Database → Create a page → Query a database → Update page properties.

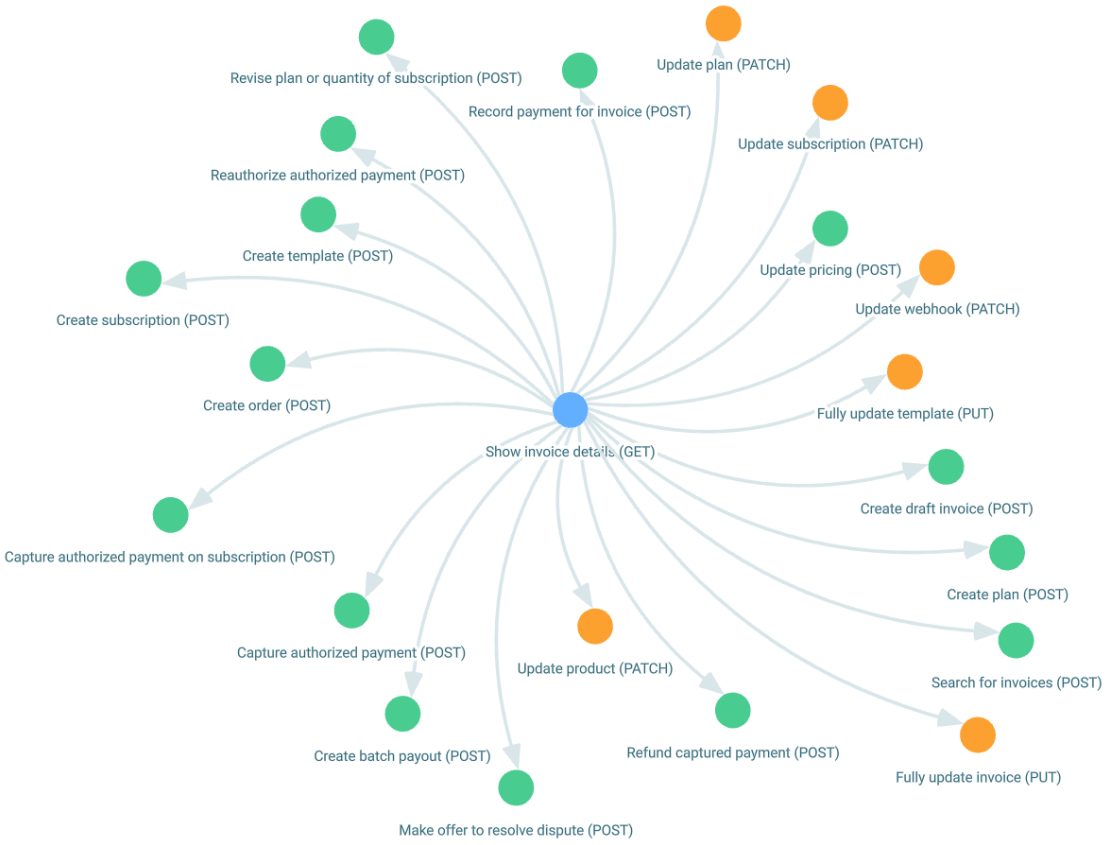


Figure 3.5: RADAR: PayPal API - Invoices

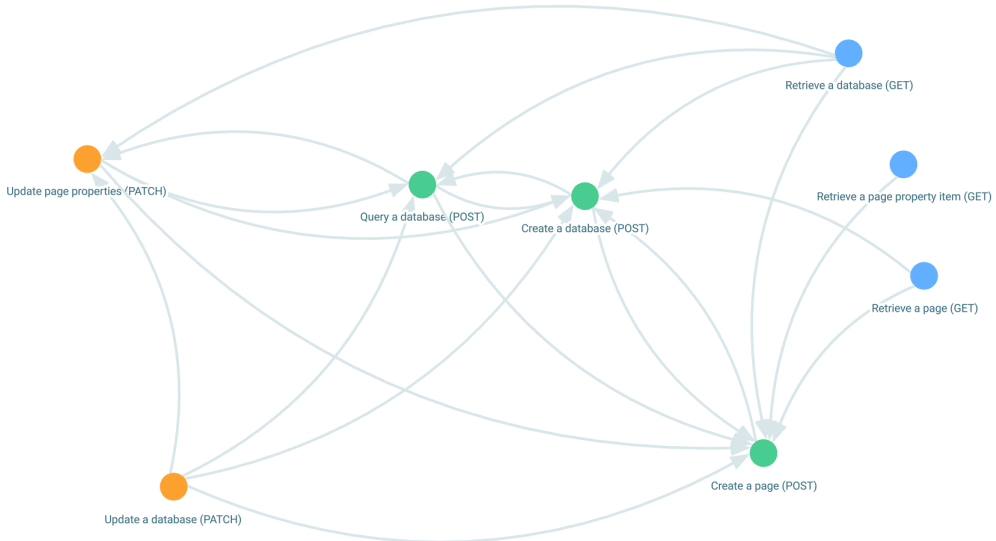


Figure 3.6: Notion API: Pages and Databases

Specifically, a database is initially created with a specific schema returning its identifier. Subsequently, a page is generated as a child of this

database, a query is executed on the database and finally, a property of the page is updated. Without any prior knowledge of the structure and, more importantly, the semantics of the API, a brief examination of the graph highlights valuable aspects of the interactions among different endpoints in order to collaboratively complete a part of the business logic implemented by the API!

3.5 Pain Points and Limitations

Despite the fact that RADAR can offer valuable insights for the effective utilization of APIs revealing useful semantic aspects, even when these APIs are entirely unfamiliar to the developers, it comes with certain pain points and limitations.

Prerequisites The primary limitation introduced by RADAR is the necessity for the existence of the Postman Collection for the API to be analyzed. While many public or even private enterprise APIs come with this type of documentation (requiring time and effort for their generation and maintenance), their availability and accessibility are not always assured. An additional important limitation for a complete analysis to take place is the necessity of multiple Postman examples to enhance the analysis with information about real attribute values. As more examples become available, the likelihood of identifying a correct dependency in one of them increases, resulting in another useful edge being added to the dependency graph. Lastly, an additional requirement is that the response bodies must be in JSON format, which, due to its widespread adoption, is not overly restrictive.

Noise A fundamental limitation of static, use case and business logic agnostic analysis is the generation of noise in the produced graphs, resulting in identifying inter-endpoint dependencies that are neither real nor beneficial. Thorough analysis of the Postman Collection detects all plausible dependencies. However, without additional knowledge beyond the static information, filtering out irrelevant dependencies is not possible.

As illustrated in figure 3.5, the resulting noise can become quite misleading for the proper comprehension and utilization of the API. Furthermore, the emphasis on identifying dependencies by attribute names (for instance, when not enough examples are available) may result in misleading dependencies. For instance, frequently used names such as “ID" (which can pertain to any entity) can lead to the generation of non-valuable edges.

Uncaught Dependencies Even when many Postman examples are available, these may not be derived from a real-world use of the API, and thus fail to reflect its actual and proper utilization. As a consequence, valuable dependencies and workflows encountered during some real utilization of the API may not be highlighted by the current static analysis.

For example, as shown in Figure 3.7, the endpoint Create product has only one detected dependent endpoint, the Create order endpoint. However, upon closer examination of the PayPal API, it becomes clear that some more valuable dependencies exist as well. For instance, the invocation of Create product endpoint can be followed by the creation of a new billing plan for this product, as well as the generation and update of an invoice associated with this product. As a result, the current static analysis not only leads to the detection of redundant, non-valuable dependencies but also fails to identify some important and useful ones.

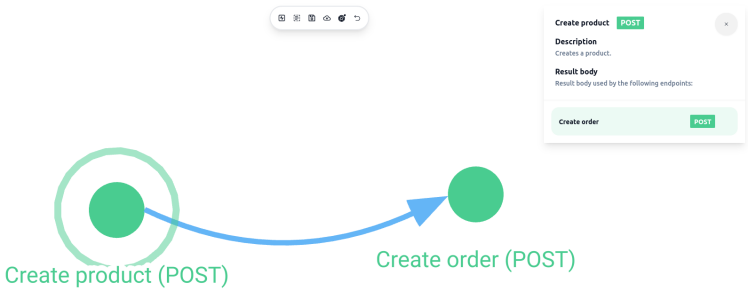


Figure 3.7: RADAR: PayPal API - Create Product dependencies

3.6 Towards Addressing the Limitations

In order to overcome the aforementioned issues, we need to eliminate the reliance of the dependency analysis on Postman Collections and improve the performance and quality of results in the current version of RADAR. Specifically, it is necessary to minimize the noise introduced into the results while ensuring that as many dependencies as possible -if not all- are identified. In other words, we need a system that potentially offers higher level both of accuracy and completeness in the generated results, without being dependent on a Postman Collection.

To eliminate the need for Postman Collections and any other form of documentation -whose availability is not guaranteed- it is reasonable to invest time in the exploration, design and development of a dynamic approach of dependency analysis. If we could gather records of real network traffic to and from an API server, we would have access to information similar to the previous data included in Postman Collections, without relying on such documentation material.

At the same time, this approach improves the dependency analysis by incorporating use case and business logic insights derived from the real-world utilization of an API. In this way, the input dataset would evolve from being static and potentially incomplete to being enriched with actual usage information. This could align the dependency analysis closer to the real, rather than theoretical, use of the API, potentially addressing both the accuracy and completeness issues.

This thesis focuses specifically on investigating this type of approach regarding the automated inter-endpoint dependency analysis in RESTful APIs.

4
Dynamic API Dependency Analysis

4.1 Problem definition

As has already been made clear, the correct, productive and efficient utilization, expansion and integration of APIs necessitates the availability of guiding information that facilitates this process. Simply understanding the structural characteristics of an API is insufficient, and treating it as a collection of independent endpoints is misleading. An API is actually a set of interacting endpoints that collectively implement pieces of business logic. Information regarding both the structure and usage of each endpoint, as well as their interoperability and interactions, is essential.

The static dependency analysis presented is a step towards enriching existing API documentations, which primarily consists of structural information, by highlighting the interactions between the endpoints. However, this was accompanied by a series of limitations and weaknesses, ranging from the necessity of having Postman Collections available to the frequent lack of accuracy in the results. When a Postman Collection is not available for an API, the current approach cannot be utilized. These cases are anything but uncommon, either due to the decision not to invest significant time in the generation of a Postman Collection or because the API is still under development and such documentation has not yet been produced, making it difficult for new developers to work on it. In such cases, the API remains entirely unknown. At the same time, even when the Postman Collection is available, the quality of the results may be inadequate.

The inability to apply this method to undocumented APIs along with the often misguided decisions resulting from the insufficient accuracy of the results -inherent to the static nature of the approach- underscores the

need to explore different methodologies.

4.2 The dynamic approach

The main difference in the strategy presented in this thesis lies in the source of data, which is no longer static. Instead, this method is going to utilize data that is dynamically generated during some real interaction with the API. The previous analysis relied on static information regarding the structure of the requests and responses of API calls, as well as potential examples which were not dynamically produced and therefore did not represent a real-world utilization of the API. The methodology presented in this thesis requires that the API is firstly subjected to a series of calls that represent all the possible realistic use case scenarios supported by this API.

For instance, if we are dealing with the Flight Booking API, which only implements the use case of making a flight reservation, the endpoints are invoked in the required order with the appropriate inputs so as to successfully register the reservation. The dynamic approach requires that the request inputs and response bodies of all these calls are recorded and stored in a database. For this purpose, a system acting as a mediator is required so as to monitor and record the network traffic to and from the API. Such a system has already been implemented as part of this research and will be presented subsequently. Once the API has been exhaustively utilized in real-world conditions covering all the functionalities and use cases it supports, the mediator system will have recorded the inputs and outputs of all the API calls. This dataset will serve as the input to be analyzed in the current dynamic approach of inter-endpoint dependency analysis, replacing the static Postman Collection of the previous method. Subsequently, this dataset undergoes a thorough examination - similar to the analysis of the static approach - which finally identifies and analyzes the inter-endpoint dependencies, leading to the generation of the dependency graph.

4.3 Man In the Middle (MIM)

As mentioned, the first step in performing the dynamic analysis presented in this thesis involves capturing and storing information about the requests and responses of a series of real API calls that cover all the possible use case scenarios supported by the API. For this purpose, a mediator system that acts as a Man In the Middle (MIM) has been developed as part of this research [18].

4.3.1 What MIM does

MIM is designed to capture and store API requests while also taking responsibility for forwarding them to the API and receive the response. In this way, MIM operates as a mediator between the API client and the API Server. The client sends API requests to MIM, which forwards them to the API, receives the response, stores information about the API call and finally returns the response to the client. In this way, the client can seamlessly interact with the API by sending requests and receiving responses, while a mediator system logs information regarding the network traffic. The only requirement is that the client has to send requests to the mediator instead of the API specifying the type and format of the request they wish to make. The rest is the responsibility of the MIM and is completely transparent to the client. This process is illustrated in the sequence diagram shown in figure 4.1.

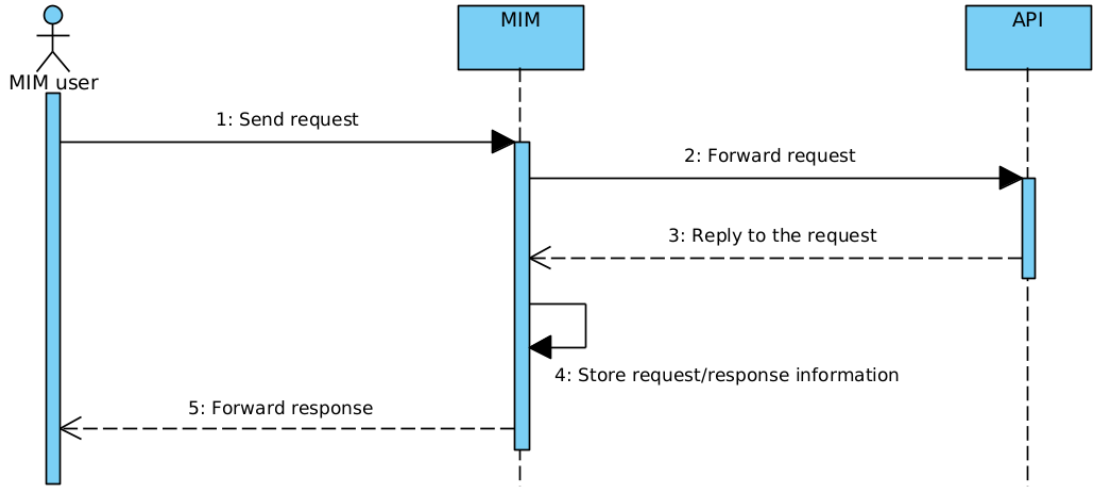


Figure 4.1: Utilization of Man In the Middle

4.3.2 Implementation insights

The functionality of MIM, as described above, was implemented using Node.js/Express and MongoDB. Upon receiving a new request, MIM extracts the sender's IP address, headers, request method, target domain, query parameters, URL segments (including path parameters), request body and any attached files. Subsequently, MIM processes the request before forwarding it to the API. For instance, if the request contains files, MIM has to save them locally before adding them into the request body of the forwarded request. In addition, certain header fields need to be excluded from the request, such as the "host" field and some other headers added by browsers. Specifically, the API client interacting with MIM

could be a frontend application running in a browser. For performance reasons, browsers add certain headers to prevent the re-sending of cached resources if the have not changed on the server. By including these headers in the forwarded request, there could be cases where the API server would return a "304 Not Modified" message instead of the actual response, which is problematic for the purpose of this analysis. Once the response is received, it stores information about both the request and the response and then sends the response back to the client.

4.3.3 Usage

The use of MIM is pretty simple and straightforward. The API of MIM provides the endpoint **/proxy/:domain/:tag**, where "domain" is the target domain, meaning the URL through which the target API is accessible, and the tag specifies a user-defined label that identifies the group of API calls to which the specific request belongs. For instance, if the API call is a part of the use case of booking a flight in the Flight Booking API, the three necessary calls to carry out this process would have the same tag, ideally a self-descriptive one, such as "flightbooking" or "usecase1" in the case of predefined numbering. Suppose that MIM is up and running locally at the address

http://localhost:3000

If the API client wants to make a POST API call to

https://flightbooking.com/api/flights

of the Flight Booking API to book a new flight, they will need to configure a request with the appropriate request body, headers, any necessary authorization tokens etc. Instead of using the previous URL that directly interacts with the Flight Booking API, they are supposed to replace it with the following:

POST http://localhost:3000/proxy/https_flightbooking_com/usecase1/api/flights

Notice how the domain is passed into the URL. The symbols :// should be replaced with an underscore along with any subsequent dots.

Once this process is carried out for all the calls related to every use case, MIM will have captured and stored all the required information which can be exported in JSON format. To achieve this, MIM provides the endpoint

/proxy_utils/export/:domain

enabling the export of the JSON file that contains all the call associated with the particular domain. This file serves as the input file for the dynamic inter-endpoint analysis.

4.3.4 Data and output format

MIM stores a document for each API call in the format shown in figure 4.2. The fields displayed represent the following information:

- **ip** : the IP address of the API client
- **method** : the HTTP request method
- **url** : the URL of the API
- **endpoint** : the API endpoint that is invoked
- **headers** : the list of request's headers and their values
- **body** : the request body
- **query** : the query parameters (if any)
- **params** : the segments of the URL (including the path parameters)
- **response** : the response body
- **tag** : the label associated with the call

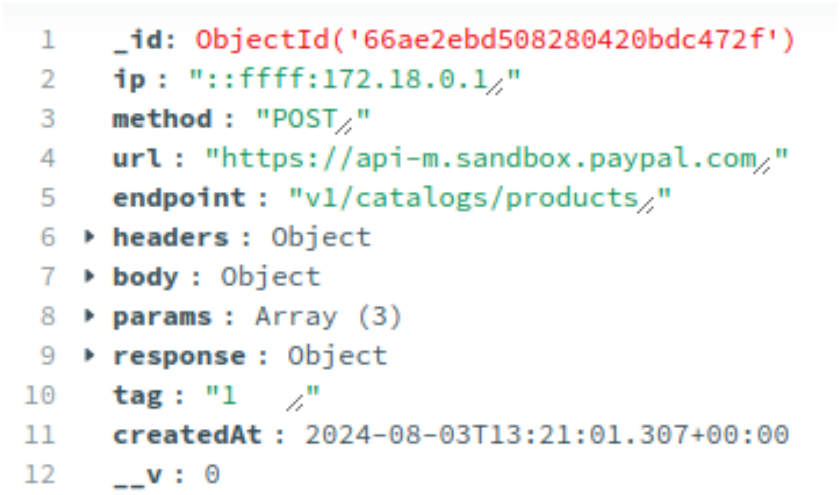


Figure 4.2: MIM - Documents format

The exported file is nothing more than a list of such documents, one for each API call.

4.3.5 Limitations

While the functionality and usage of MIM appear simple and clear, it is essential to note that there are certain limitation regarding the environment in which such a system can be utilized. MIM has to be placed between an API client and an API server accessible to the client. If the API is protected by a firewall and the client is not allowed to access it, MIM will not be able to forward and capture the network traffic. When working with public and widely accessible APIs, MIM can be installed and configured locally. In contrast, for a private enterprise API, MIM needs to be deployed within the relevant private network.

Furthermore, regarding the information recorded by such a dynamic system, it is important to highlight the inability to identify the path parameters within the URL. Due to the lack of available documentation specifying the structure of each endpoint, it is not possible to distinguish which items among the URL segments in the list of params (figure 4.2) are path parameters. This issue, as will be elaborated on later, limits the quality of the analysis that can be performed on inter-endpoint dependencies involving path parameters.

4.4 The modified algorithm

The algorithm for dynamic inter-endpoint dependency analysis is an extension and modification of the algorithm upon which the static analysis is based. Specifically, the algorithm utilizes the dynamically generated log file exported by MIM as input. After an in-depth analysis, it detects the inter-endpoint dependencies, the attributes on which these dependencies are based and the dependency type (body, path or query).

The core implementation is shown in the algorithms 1 and 2 presented in the next pages. First, the JSON file exported by MIM is read and sequence numbers are assigned to the calls of each use case according to the order in which they occured. Then, two dictionaries are created, one for the request values and another for the response values. The purpose is to record every attribute value encountered in a request or response. For each value, information such as the endpoint where it was found, the name of the corresponding attribute, its type and additional details are recorded.

The main part of the algorithm resides in the procedure of algorithm 2, which receives the two dictionaries as input and identifies inter-endpoint dependencies. Specifically, for each attribute value recorded in the response dictionary, it checks whether the same value was encountered in the request of any subsequent call (higher sequence number) of the use case. If so, a dependency is recorded.

The algorithm is also configurable. The following options are provided:

Algorithm 1: Dynamic inter-endpoint dependency analysis

```
Data: Read MIM log file as Input
/* Input = a list of calls as shown in figure 4.2          */
Result: REQUEST_VALUES/RESPONSE_VALUES = dictionaries
        with key=attribute value and value=info about the
        appearance of the attribute value in a request or response
Initialization: Give sequence numbers to the calls of every use case

foreach call in Input do
    if call.body then
        foreach parameter in call.body do
            REQUEST_VALUES[parameter.value].APPEND({
                url: call.url
                seq: call.sequence_number,
                tag: call.tag,
                paramName: parameter.name,
                paramType: parameter.type
            })
        end
    end
    if call.response then
        foreach attribute in call.response do
            RESPONSE_VALUES[attribute.value].APPEND({
                url: call.url
                seq: call.sequence_number,
                tag: call.tag,
                attrName: attribute.name,
                attrType: attribute.type
            })
        end
    end
end

COMPUTE_DEPENDENCY_GRAPH(
    REQUEST_VALUES,
    RESPONSE_VALUES
)
```

- **Include query parameters**: Whether to detect dependencies based on query parameters.
- **Include path parameters**: Whether to detect dependencies based on path parameters. As mentioned, it is impossible to distinguish which segments of the URL correspond to path parameters due to the lack of

Algorithm 2: COMPUTE_DEPENDENCY_GRAPH

Data: REQUEST_VALUES and RESPONSE_VALUES
Result: DEPENDENCY_GRAPH = dictionary with key=edge and value=edge info

```
procedure COMPUTE_DEPENDENCY_GRAPH
    foreach value in RESPONSE_VALUES do
        if value in REQUEST_VALUES then
            foreach appearanceRes in RESPONSE_VALUES[value] do
                foreach appearanceReq in REQUEST_VALUES[value] do
                    if appearanceRes.paramType==appearanceReq.paramType
                      AND appearanceRes.url!=appearanceReq.url
                      AND appearanceRes.seq < appearanceReq.seq
                      AND appearanceRes.tag == appearanceReq.tag then
                        DEPENDENCY_GRAPH[(
                            appearanceRes.url,
                            appearanceReq.url)].APPEND({
                                fromAttribute: appearanceRes.paramName,
                                toAttribute: appearanceReq.paramName
                             })
                    end
                end
            end
        end
    end
end procedure
```

documentation indicating this. By selecting this option, the dependency analysis will treat all the path segments as if they were path parameters. This may yield misleading results and detect dependencies that are not valuable.

- **GET dependencies only**: Whether to only detect dependencies that derive from GET API calls. These represent the most common cases of inter-endpoint dependencies, and although this option might exclude some valuable dependencies it is also supposed to reduce the noise.

- **Include boolean values**: Choose whether to include dependencies derived from boolean values. Boolean attributes may represent a misleading factor in the dependency analysis, as they allow only two possible values. As a result, it is possible for two boolean attributes to have the same value without having a dependency relationship.

- **With strict types**: Choose whether to include only dependencies derived from attributes with the same value and type or if it is acceptable for two dependent attributes to have the same values but different types (for instance, the string value "42" and the integer value 42 could be considered to contain the same information or not). The

option not to enforce strict types is useful when handling query parameters that are always parsed and recognized as strings and thus lose type information.

4.5 The new version of RADAR

Following the integration of dynamic analysis into RADAR, users can choose to perform this type of analysis by uploading the JSON file exported from MIM and completing the analysis configuration by choosing whether to enable the aforementioned options.

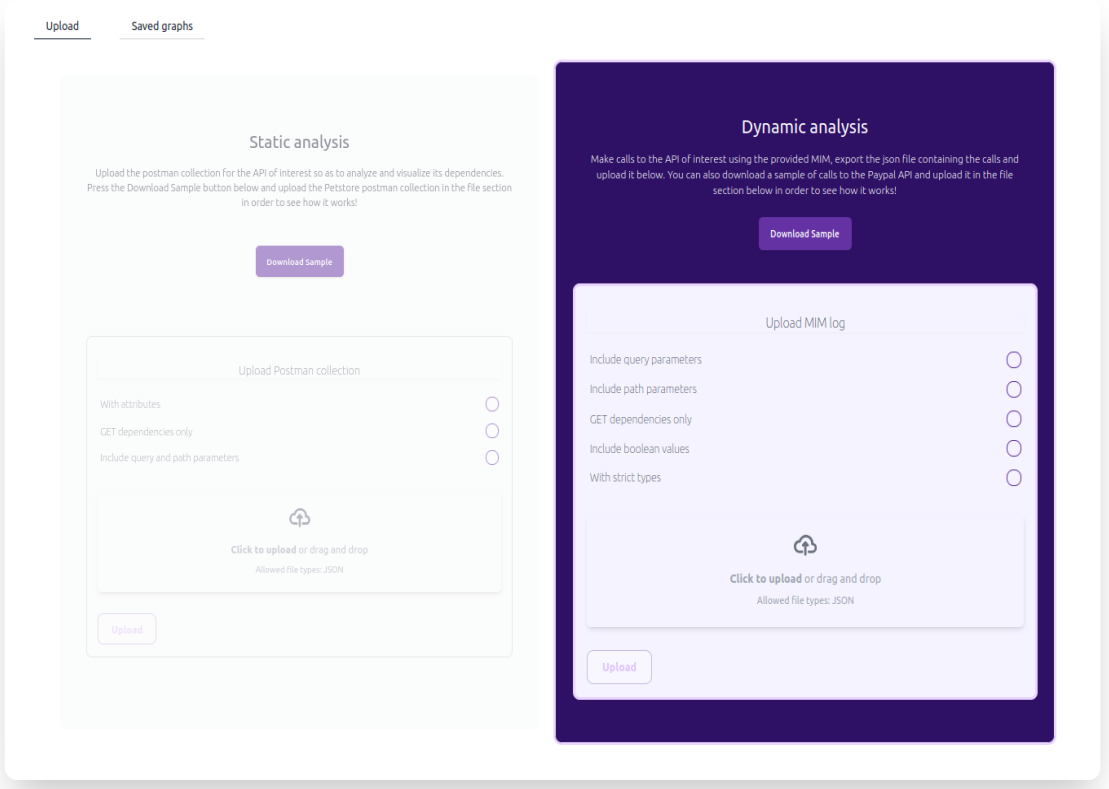


Figure 4.3: RADAR after integrating dynamic inter-endpoint dependency analysis

4.6 Activity diagram

Conducting dynamic analysis for an API requires following the steps outlined in the activity diagram shown in figure 4.4. The user of the system begins by setting up MIM in a controlled environment to ensure access to the target API (which could be a private enterprise API with restricted

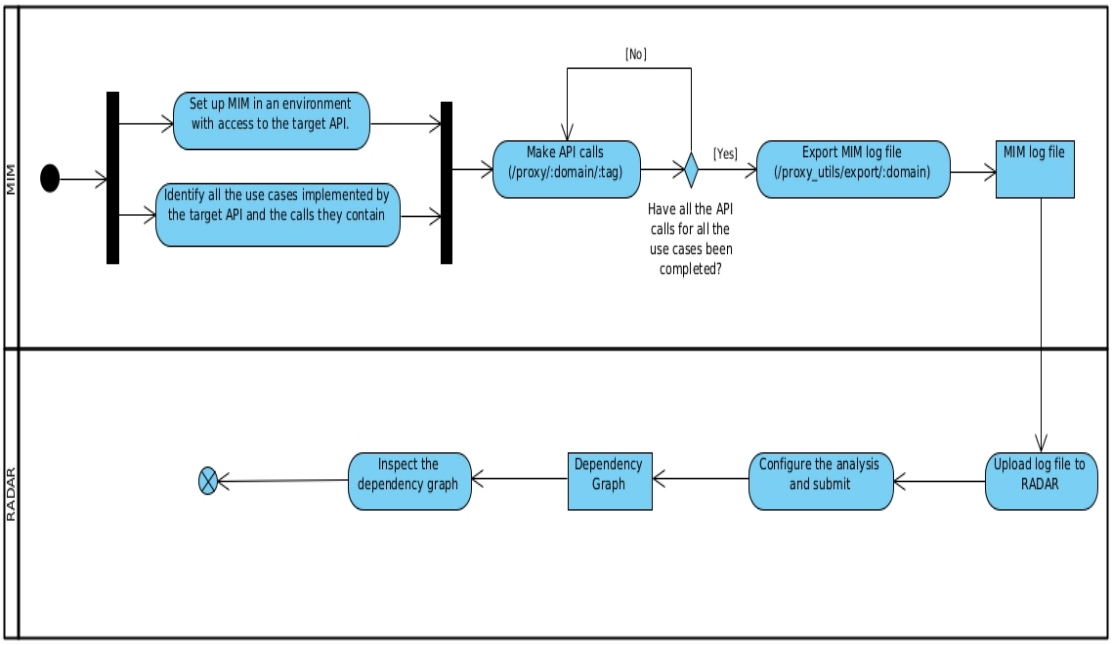


Figure 4.4: Conducting dynamic analysis

access). Meanwhile, it is important that the API under analysis has been thoroughly examined, identifying all possible use cases and the required API calls for their execution. After all these calls have been made through MIM, the resulting log file is exported and uploaded to RADAR. The user then configures the analysis parameters as outlined earlier. Afterwards, the user can access the generated dependency graph, allowing them to examine the inter-endpoint dependencies, the corresponding attributes and some statistics.

5 Utilizing the modified version of RADAR: Study cases

In this chapter, the system introduced in the previous chapter is utilized in order to study four APIs:

1. the public sandbox environment of PayPal API
2. the public OpenAI API
3. the public Notion API and
4. the private Echos API developed by Softlab which is available through a Frontend application

In each of these cases, the process illustrated in the activity diagram of figure 4.4 is followed. The examination of the results through the dependency graph is the subject of the next chapter. The current chapter outlines the initial design step required for conducting dynamic analysis, which involves identifying all the use cases implemented by the API along with the necessary API calls. The documentation for each API was carefully studied so as to determine these use cases. Notably, for the PayPal API, this initial design step was documented in Visual Paradigm and the relevant diagrams will be presented below.

5.1 PayPal API

The PayPal API provides a wide range of functionalities including the creation and management of products, orders, invoices, subscriptions and

payments [21] .

The PayPal API is organized into separate APIs, each of which contains a specific set of functionalities. This organization into separate APIs is shown in figure 5.1. After studying the documentation, the following use

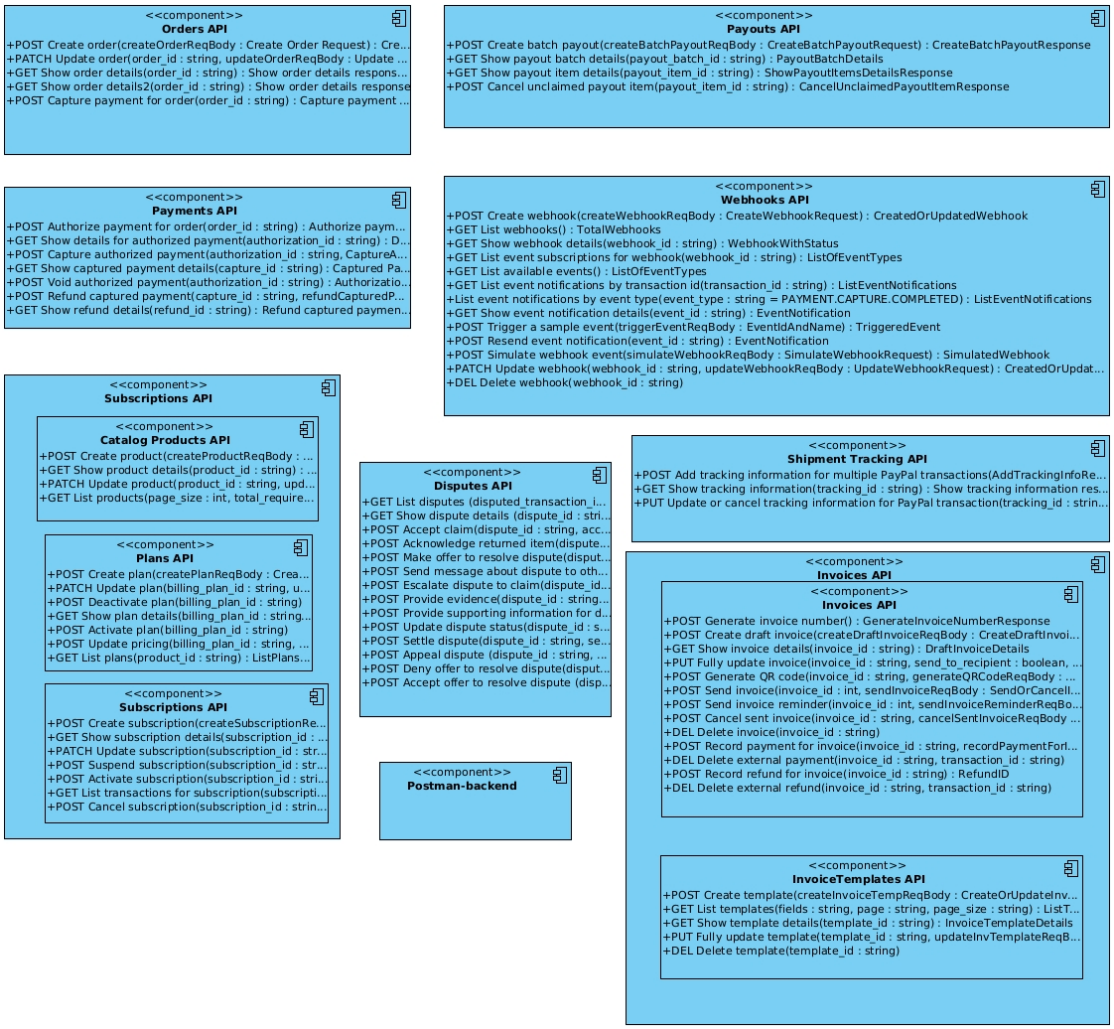


Figure 5.1: PayPal APIs

cases were identified. For each use case, a sequence diagram illustrating the API calls is provided, along with a description of the business logic they implement, allowing for an initial understanding of certain dependencies.

Use Case 1: New order - Payment - Tracking information [Fig. 5.2]

Initially, a product is created, information about it is retrieved and then its description is updated. Following this, and order is placed for the same product and after updating the shipping address, information about this order is retrieved. The payment of the order - once it is confirmed that it is feasible based on customer's balance (Authorize payment) - is captured by

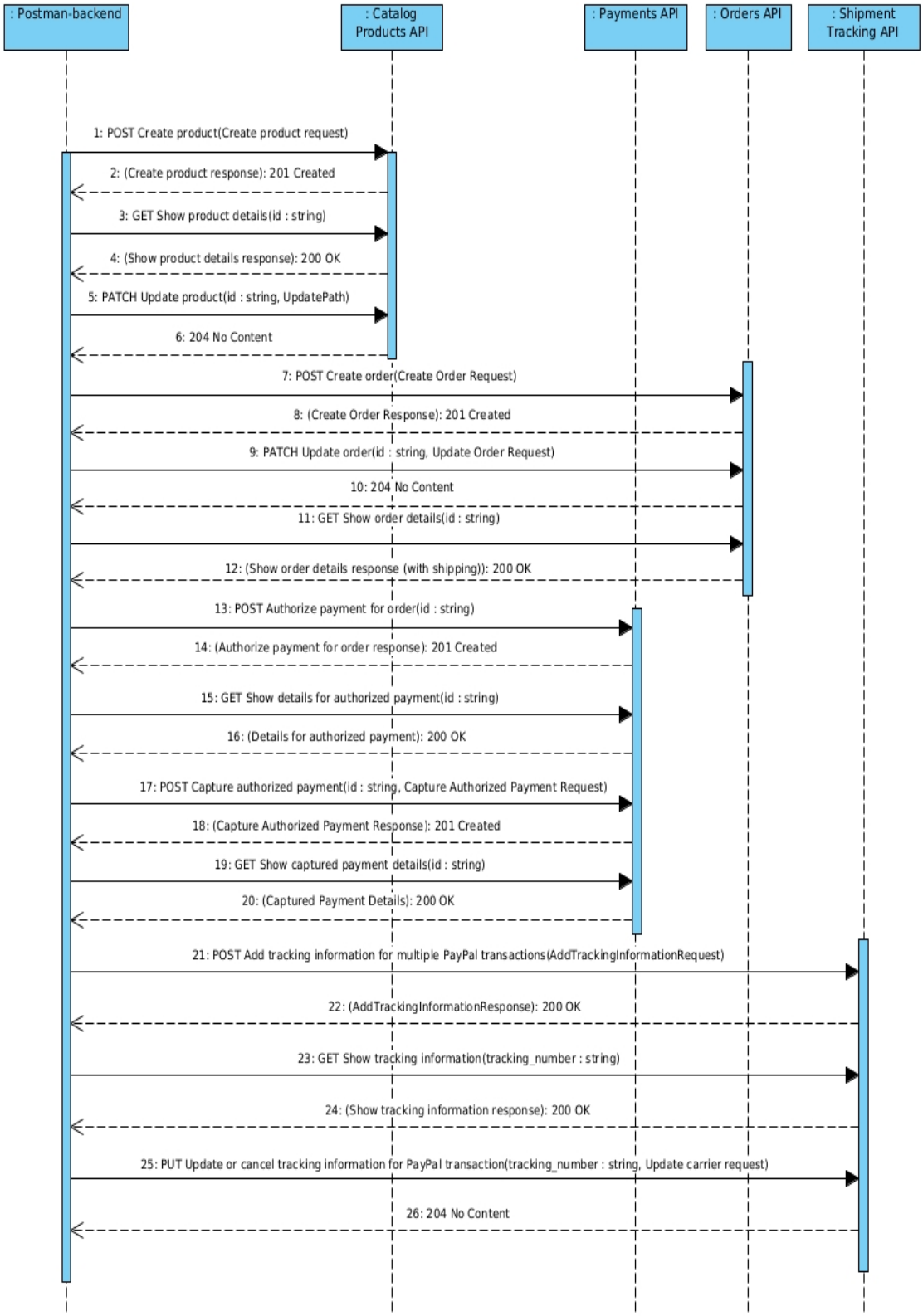


Figure 5.2: PayPal API - Use case 1 sequence diagram

the seller, and information about the specific payment is retrieved. Lastly, tracking information is added to the previous captured payment, which, after being retrieved, is updated (specifically the carrier is updated).

Use Case 1b: New order - Payment - Void Payment [Fig. 5.3]
This use case is a variation of Use Case 1, following a different path and leading to a different outcome. An order is created for a product, information is retrieved for it, the payment is authorized after confirming the customer's sufficient balance and finally the payment is canceled and information about it is retrieved.

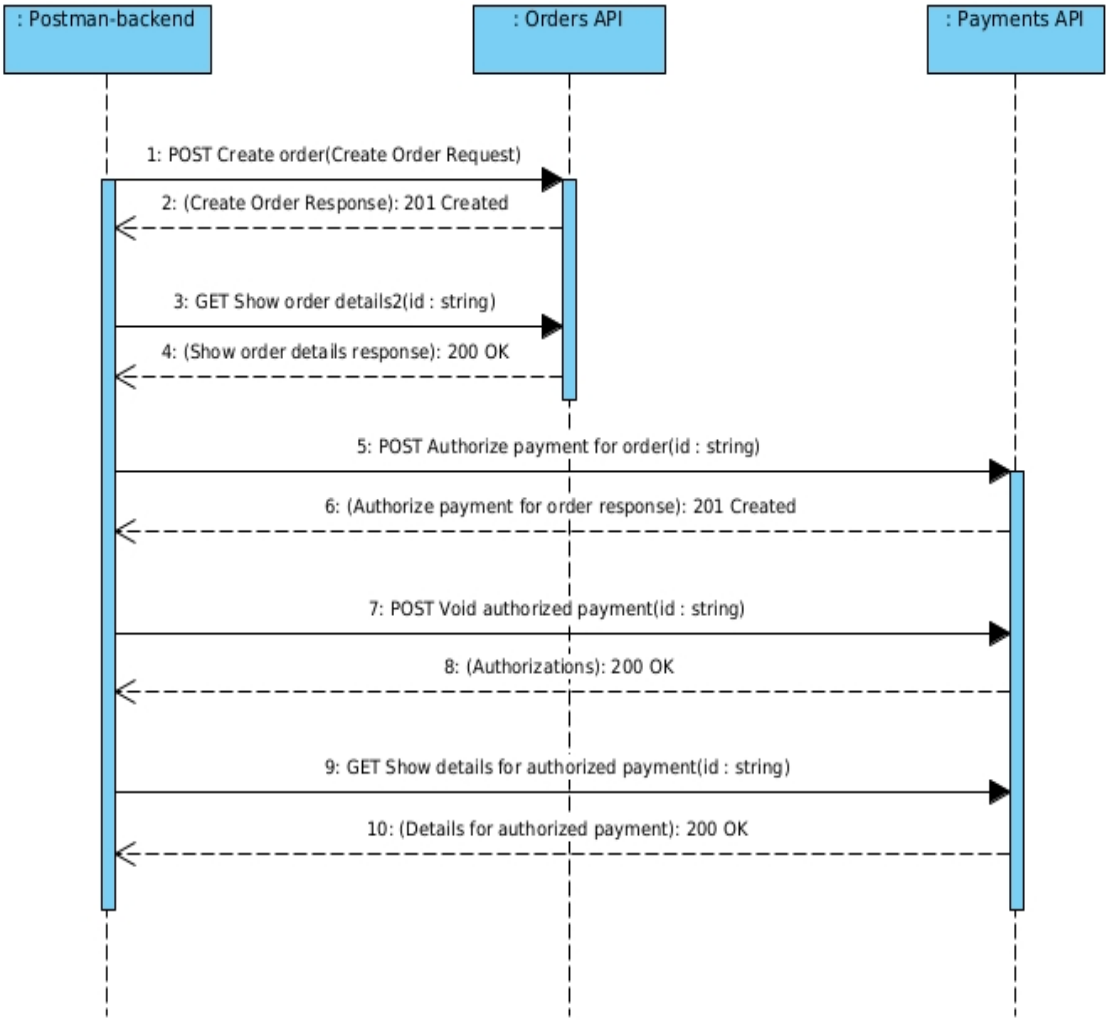


Figure 5.3: PayPal API - Use case 1b sequence diagram

Use Case 1c: New order - Capture Payment - Refund [Fig. 5.4]
This is yet another variation of Use Case 1 . A new product is created and information is retrieved for it. Then, an order is placed for the product, the

payment for this order is processed and captured by the seller, followed by a refund, and finally refund information is retrieved.

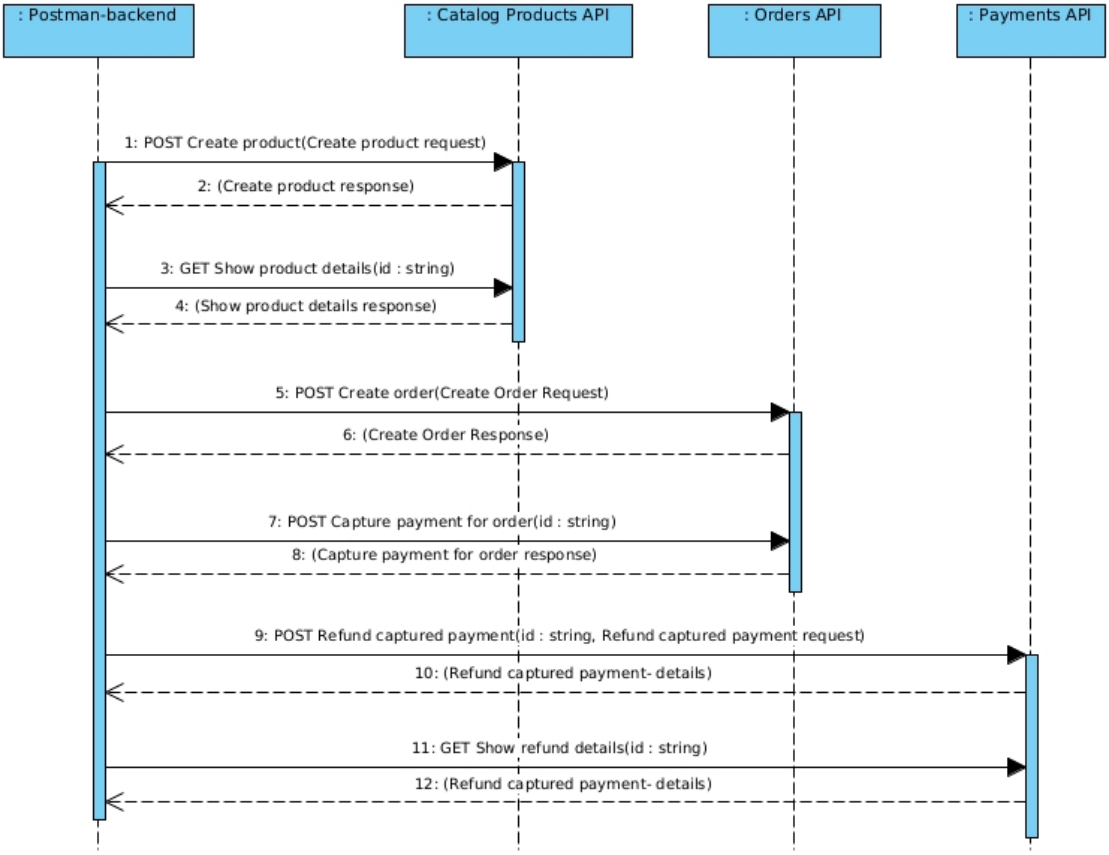


Figure 5.4: PayPal API - Use case 1c sequence diagram

Use Case 2: Create, update and send Invoice - Cancel & Delete [Fig. 5.5]
First, four distinct products are created and information is retrieved for them. Then, a new invoice number is generated, and using this number, a new invoice is created that includes the two initial products. Information regarding the invoice is retrieved before the invoice is updated by replacing the initial products with the other two created earlier. Following this, a QR code is generated and the invoice is sent to the customer followed by an invoice reminder. Finally, the invoice is canceled and then deleted.

Use Case 2b: Create Invoice - Pay and Refund [Fig. 5.6]
This use case is a variation of Use Case 2. Initially, all available products are retrieved and an invoice number is generated. Some of these products are selected and a new invoice, incorporating these products and referencing the previous invoice number, is then created and sent to the customer as before. Then, a new payment is recorded for this invoice, but is deleted

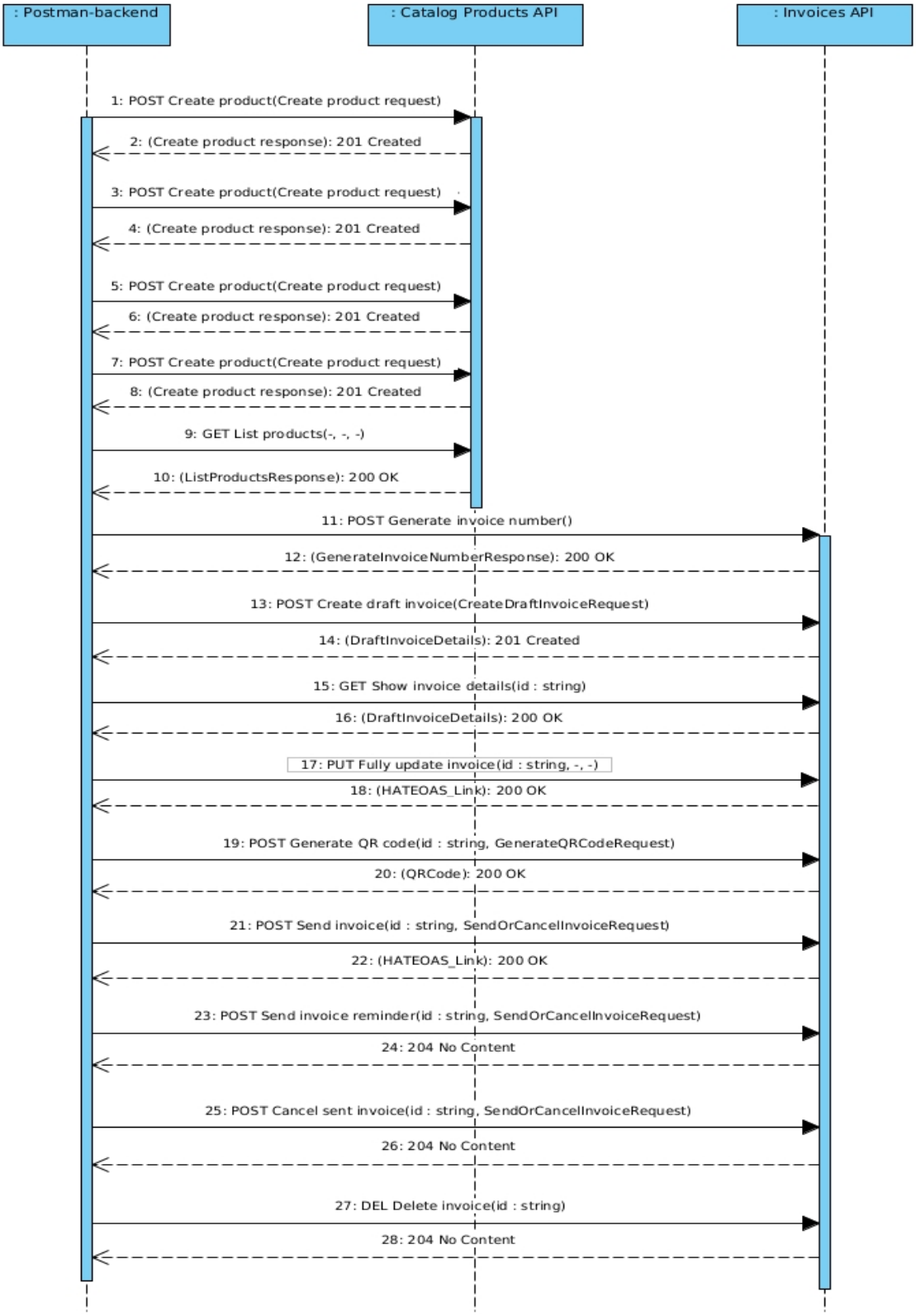


Figure 5.5: PayPal API - Use case 2 sequence diagram

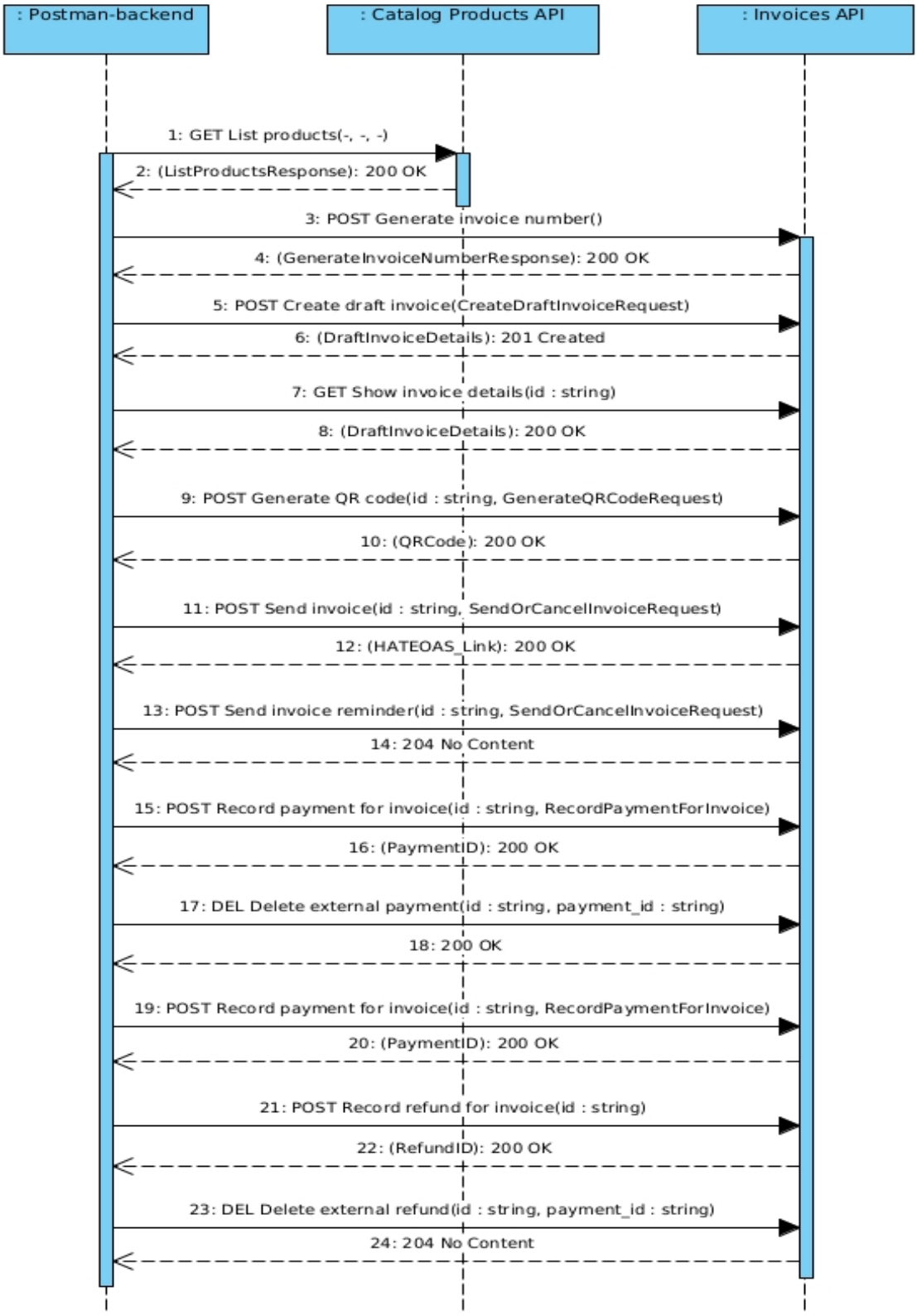


Figure 5.6: PayPal API - Use case 2b sequence diagram

afterward. Following this, a payment is recorded for the invoice again, a portion of which is refunded. Finally, this refund is canceled and deleted.

The sequence diagrams for use cases 2 and 2b are shown on the following two pages.

Use Case 3: New order - Payment - Tracking info - Dispute [Fig. 5.7]

This extensive use case simulates a complete scenario that leads to the submission and management of a dispute made by the customer due to receiving a defective product. The PayPal Disputes API offers a wide range of functionalities for handling such situations.

Initially, a new order is created for a product, and upon completion of the payment, tracking information is added to monitor its progress until delivery to the customer. This process is included in another use case but is also a prerequisite for utilizing the Disputes API. Also, a new dispute is submitted by the customer through the UI of the PayPal sandbox environment as this specific endpoint is not provided.

Then, all disputes are retrieved, and the previously submitted one is selected to get detailed information about it. The seller makes an offer to the customer regarding this dispute, specifically proposing a refund of 50$ out of the 80$ total value of the product. The customer rejects the offer and the seller sends a relevant message to the customer so as to apologize. Since the issue could not be resolved through communication between the two parties, the seller escalates the specific dispute to a PayPal claim for a PayPal agent to manage it. The seller provides evidence and extra information to support their position, and the status of the dispute is updated to WAITING_FOR_BUYER_RESPONSE, allowing the customer to also submit evidence. Following that, the customer submits evidence through the PayPal UI, as such functionality in not included in the available API. Subsequently, the PayPal agent settles the dispute in favor of the buyer based on the provided evidence. The seller appeals the decision by providing additional evidence and finally the PayPal agent settles the dispute in favor of the seller.

Use Case 3b: Dispute - Accept Offer - Return item [Fig. 5.8]

This use case is a variation of the previous one. The initial actions are the same, but now the customer accepts the seller's offer regarding the specific dispute (which refers to full refund and return) and chooses to return the item by filling out the corresponding form in the PayPal UI. Finally, the seller acknowledges the return of the item.

The sequence diagrams for both use cases (3, 3b) are shown on the following two pages, in figures 5.7, 5.8.

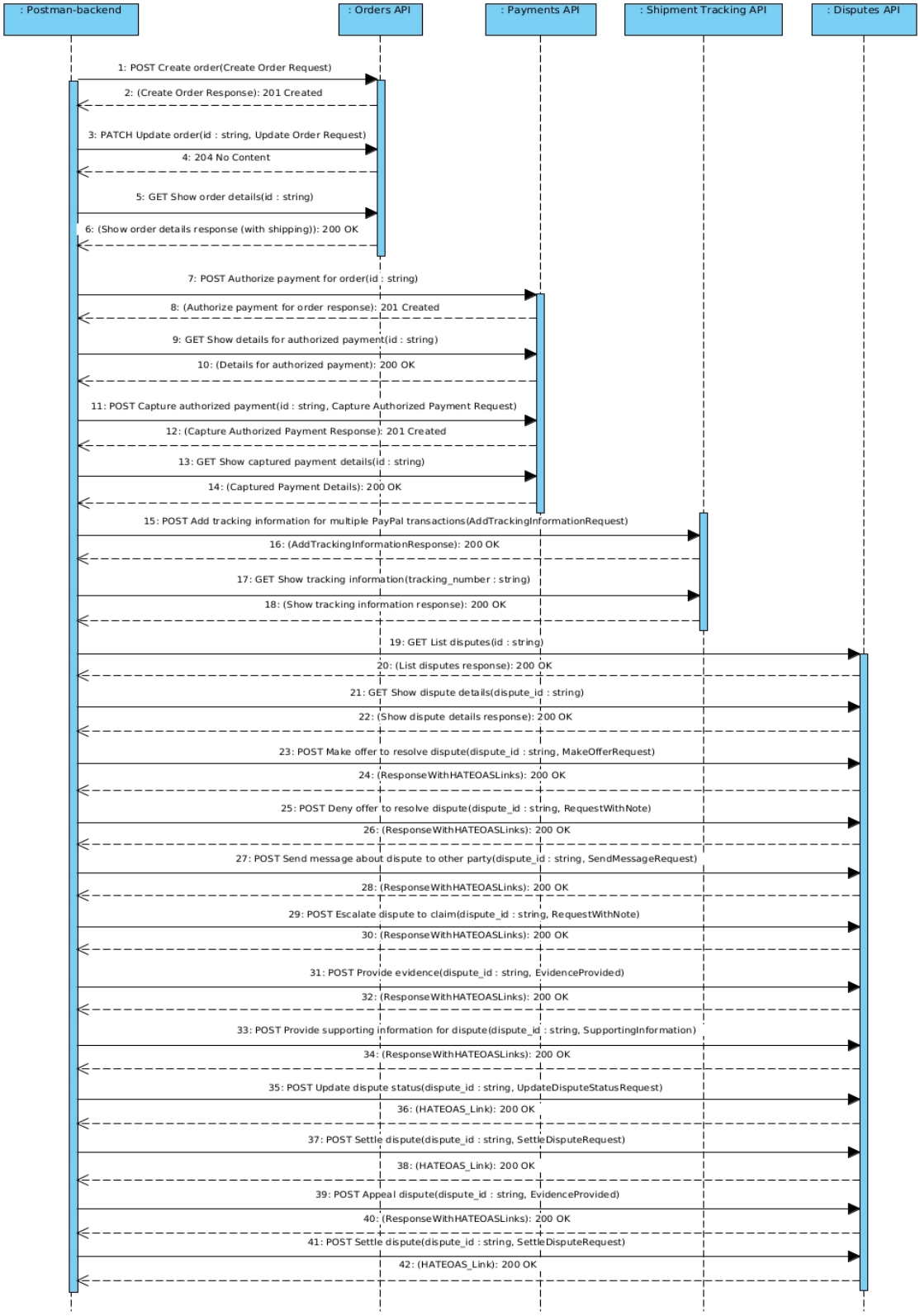


Figure 5.7: PayPal API - Use case 3 sequence diagram

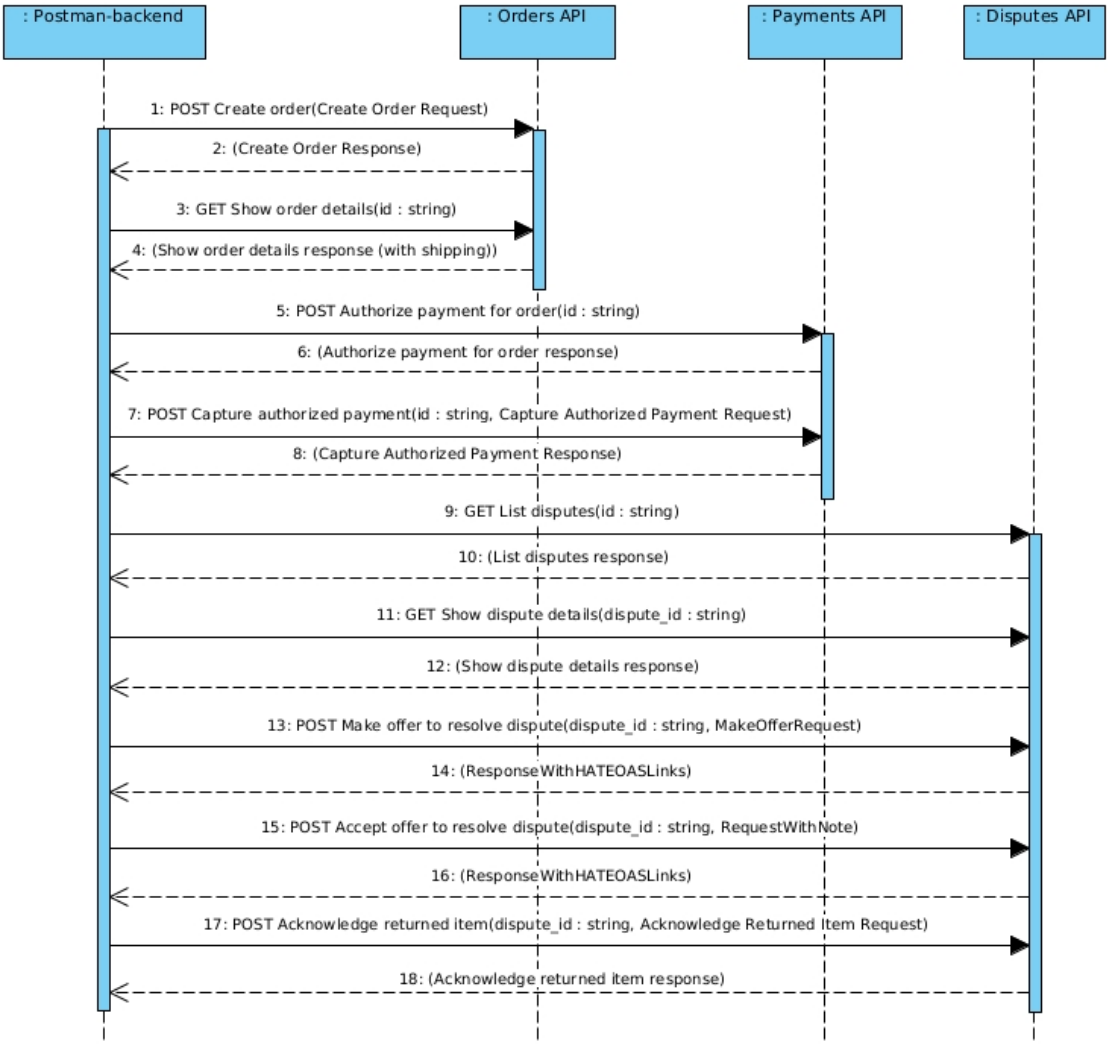


Figure 5.8: PayPal API - Use case 3b sequence diagram

Use Case 3c: Dispute - Accept Claim [Fig. 5.9]

This use case is the final variation of use case 3, in which the dispute is accepted by the seller immediately, without the need for any negotiation. Specifically, after an order is created and paid for, the customer submits a new dispute requesting a full refund and being willing to return the defective item back to the seller. The seller accepts the claim, the customer chooses to return the item using the PayPal sandbox UI and the seller acknowledges the return of the item. The sequence diagram is shown in figure 5.9.

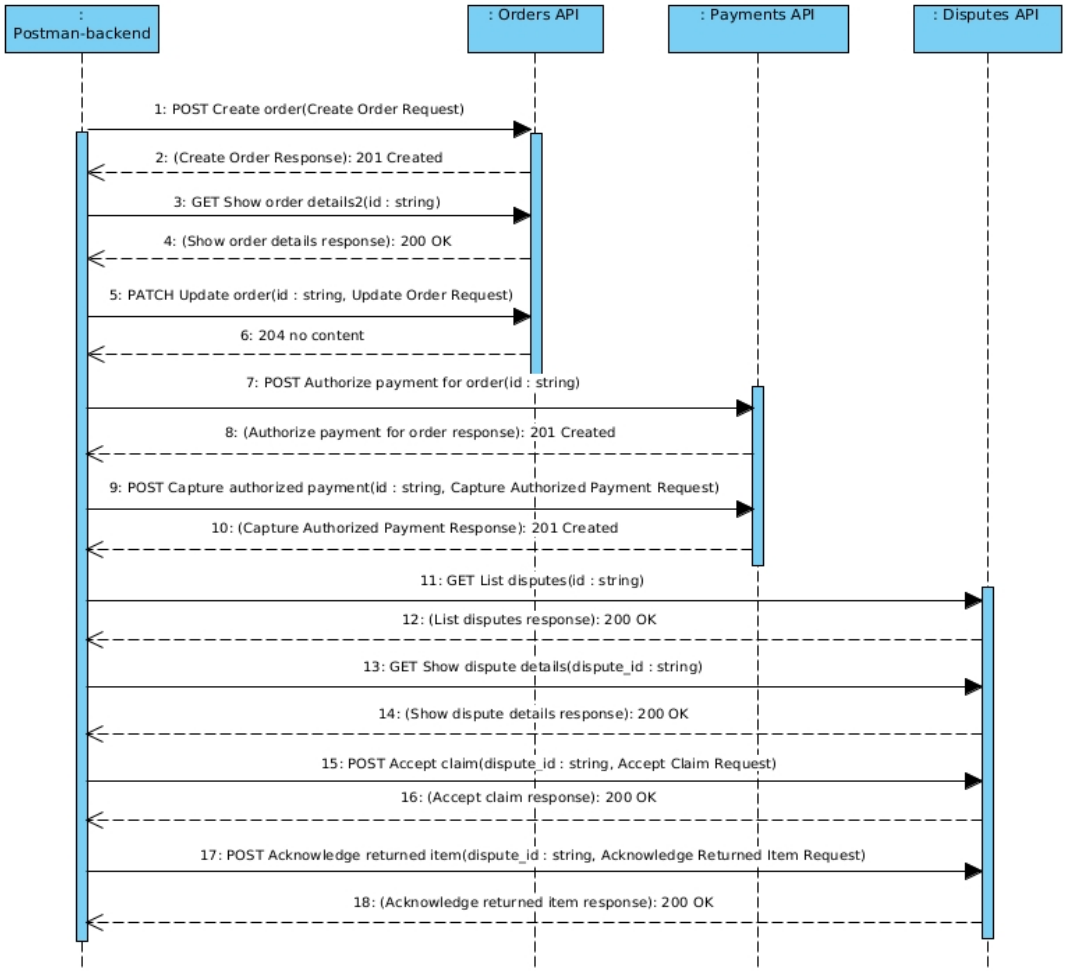


Figure 5.9: PayPal API - Use case 3c sequence diagram

Use Case 4: Billing plan and subscription [Fig. 5.10]

In this use case, a subscription is created based on a billing plan for a product. Specifically, a new product is created and information about it is retrieved. Then, a monthly billing plan is created for this product with a 1-month trial period and a 12-month regular period, which is later updated and deactivated. After retrieving information about the deactivated plan, it is reactivated and its pricing schema is updated by increasing the cost for both the trial and regular period. Following this, all available billing plans are retrieved and the previously generated plan is selected, for which a new subscription is created for a customer. The customer approves it through the PayPal UI (there is no endpoint avaialble for this), information about the new subscription is retrieved and certain subscription fields are updated. Subsequently, The subscription is suspended and then reactivated. Finally, all transactions related to this subscription are retrieved (any recorded payment) and the subscription is canceled.

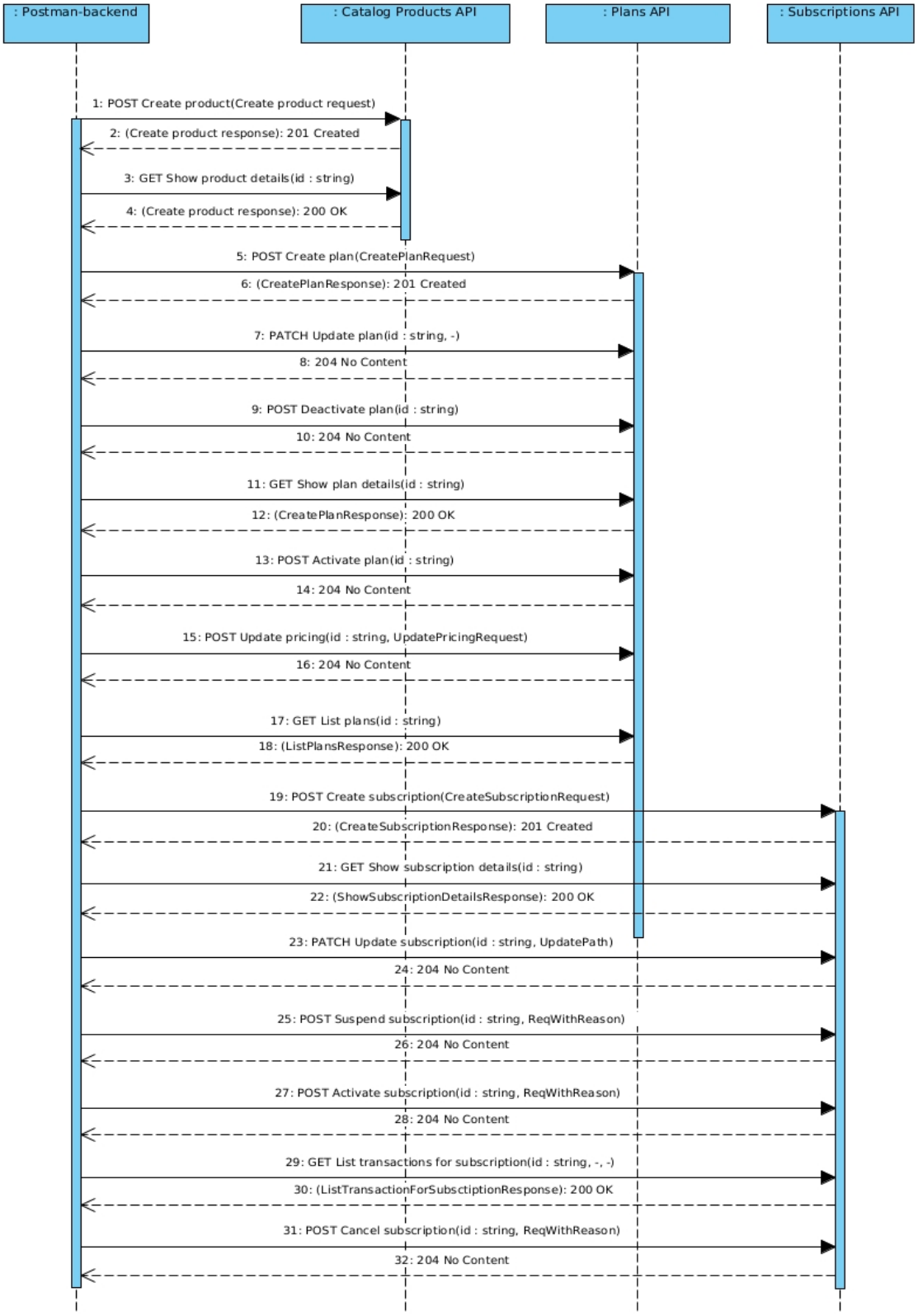


Figure 5.10: PayPal API - Use case 4 sequence diagram

Use Case 5: Payouts [Fig. 5.11]
The Payouts API is used to make payments (such as rebates or rewards) to multiple PayPal recipients [19].

A batch of payments to three recipients is created and information about the batch is retrieved. Then, one of the batch items (i.e. a specific payment) is selected and details regarding this payment are retrieved. Finally, any unclaimed payout items (for instance, those sent to an invalid account) are canceled, and the corresponding funds are returned to the sender.

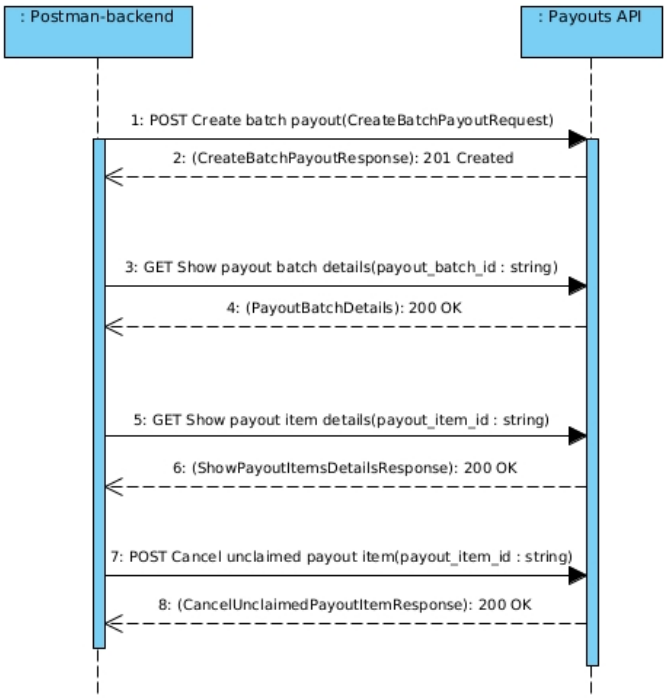


Figure 5.11: PayPal API - Use case 5 sequence diagram

Use Case 6: Webhooks [Fig. 5.12]
The Webhooks API introduces webhooks which are HTTP callbacks that receive notification messages for events. After configuring a webhook listener for an application, we can create a webhook, which subscribes the webhook listener to events [20]. In this use case, we create and utilize webhooks.

First, a webhook is created and subscribed to all possible events. We display all available webhooks and, after selecting the one just created, we retrieve detailed information and show its event subscriptions. Following this, the goal is to trigger events that will activate the created webhook. For this purpose, a new order for a product is created, and the necessary payment is completed (triggering the "PAYMENT CAPTURE COMPLETED" event). We then list all supported events and select "PAYMENT

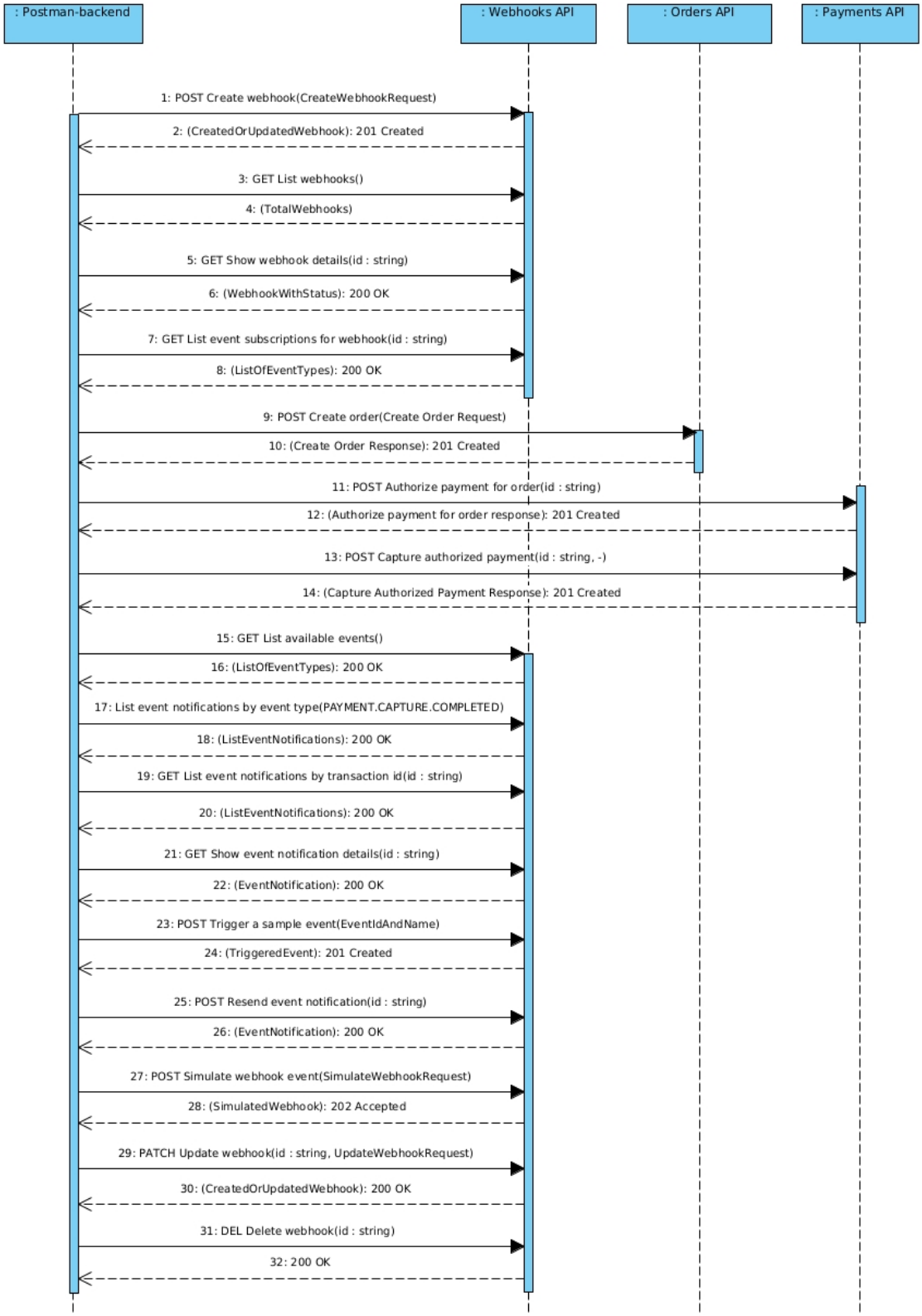


Figure 5.12: PayPal API - Use case 6 sequence diagram

CAPTURE COMPLETED" event. From the event notifications generated by the webhook, we filter those of type "PAYMENT CAPTURE COMPLETED". We then retrieve the event notifications related to the payment transaction of the order created earlier. For this specific payment event notification, we then retrieve further details. Afterward, we create a sample event and try to resend the corresponding event notification. Finally, we simulate the webhook behavior when a "CUSTOMER DISPUTE CREATED" event occurs, we update the webhook subscriptions so as to include only five specific events instead of all and we delete the webhook.

Use Case 7: Invoice Templates [Fig. 5.13]

The invoices created in use cases 2, 2b used a default template. The InvoicesTemplates API handles the creation of new invoice templates. To utilize this API, we create a new invoice template. Then, we list all available templates and select the one we just created, for which we display detailed information. Finally, we fully update this template and then delete it.

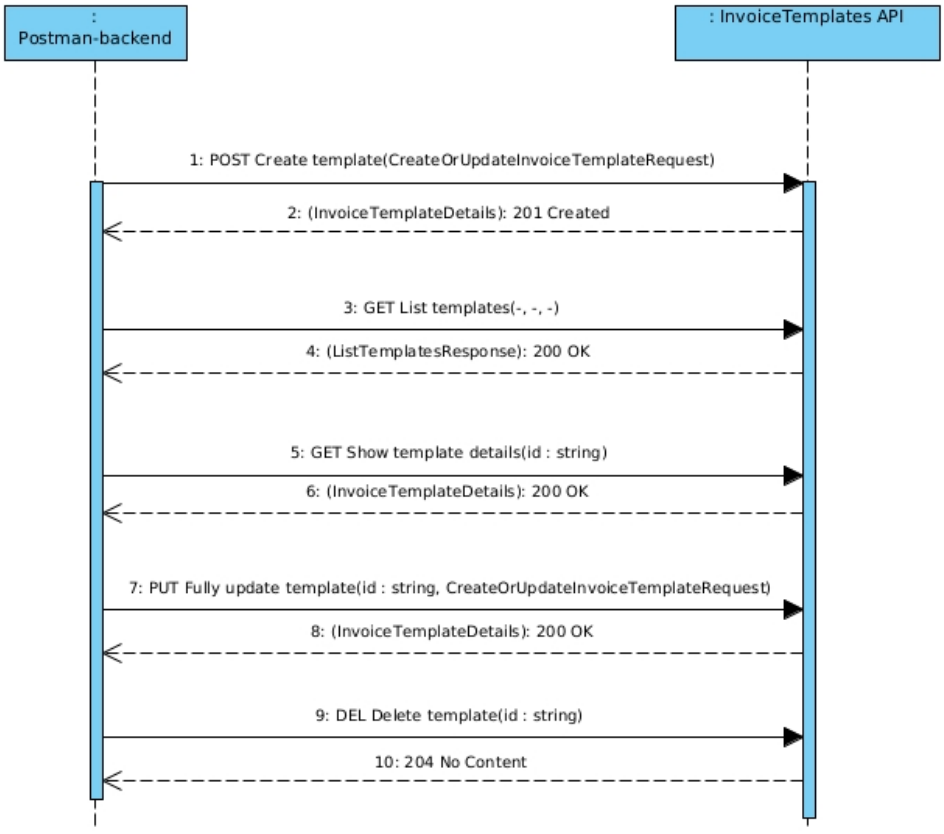


Figure 5.13: PayPal API - Use case 7 sequence diagram

5.2 OpenAI API

The OpenAI API offers a wide range of functionalities, including submitting various types of questions to AI models, creating batches of API requests for asynchronous processing and fine-tuning jobs, which allow an existing model to be trained on a provided dataset [22]. After reviewing the OpenAI API documentations, three use cases were identified. For each use case, the corresponding API calls are presented, along with a description of the business logic they implement.

Use Case 1 : Prompts [Fig. 5.14]
In this use case, an AI model is selected and various types of questions are submitted to it. As shown in figure 5.14, all available models are first listed. After selecting one, information about it is retrieved and from now on, questions will be submitted in this model.

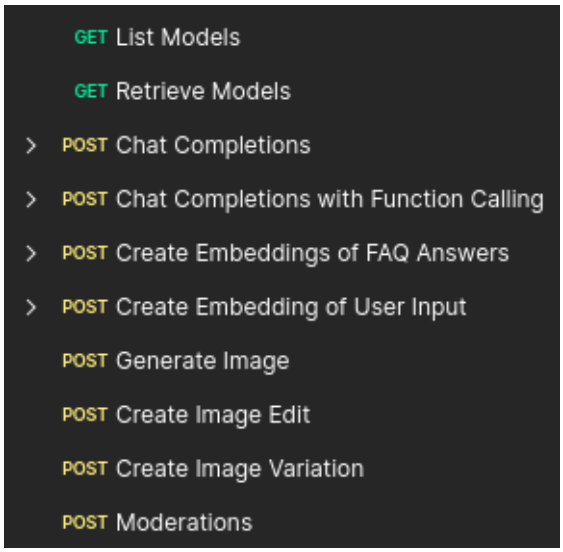


Figure 5.14: OpenAI API - Use case 1 API calls

The model is then subjected to 2 questions. In the second case, the function calling capability is employed, which requests the model not to return a message with the answer. Instead, a definition of an implemented function is provided and, based on the user's question, the model determines values for the input parameters of the function so that the client application can subsequently invoke it correctly [23].

Then, the creation of embeddings for two text inputs is requested: an FAQ answer and a user input. By converting the two texts into embeddings (which are vectors of floating-point numbers), the calculation of relatedness between them is facilitated, allowing, for example, the appropriate FAQ answer to be matched with a user's question.

Subsequently, an image is generated based on its description, an edited version of another image is created according to a description of the desired image and finally a variant of another image is generated. Ultimately, the model is requested to classify a given text input regarding whether it is harmful across several categories.

Use Case 2 : Request batches [Fig. 5.15]
In this use case, a file containing a set of prompts for the GPT-4 model is initially uploaded. Then, all files are listed and the previously uploaded file is selected to retrieve information about it and its content. Following this, a batch of requests is generated for the chat completion endpoint based on the uploaded file and details regarding the generated batch is retrieved. Subsequently, a second file containing prompts for the same model is uploaded and a second batch of requests is created. Once all created batches are listed, the last one is selected, canceled and finally deleted.

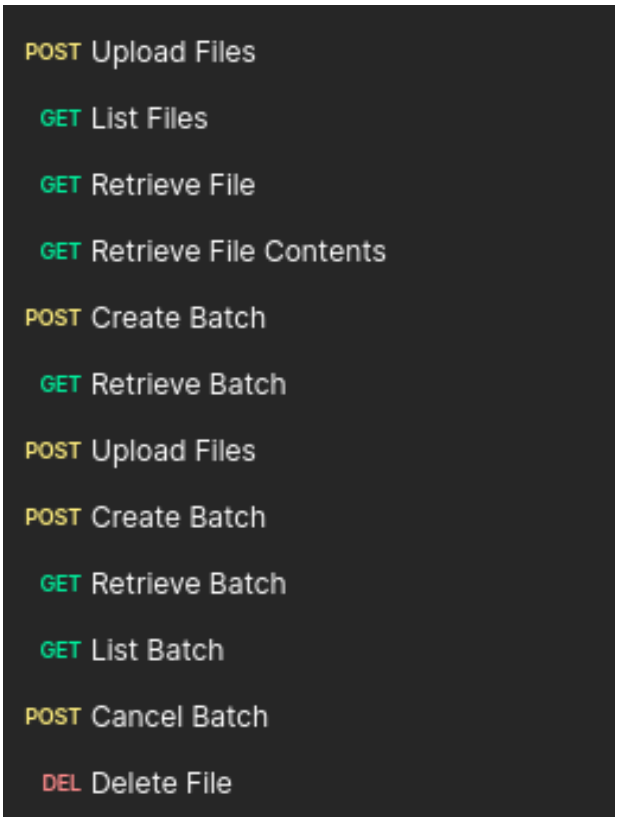


Figure 5.15: OpenAI API - Use case 2 API calls

A snippet of the uploaded file containing the batch of questions is shown in figure 5.16.

```
{
  "custom_id": "request-4",
  "method": "POST",
  "url": "/v1/chat/completions",
  "body": {
    "model": "gpt-4",
    "messages": [
      {
          "role": "system",
          "content": "You are a helpful assistant."
      },
      {
          "role": "user",
          "content": "How do I make a cup of coffee?"
      }
    ]
  }
}
```

Figure 5.16: A snippet of the file containing a batch of questions

Use Case 3 : Fine-tuning jobs [Fig. 5.17]
In this use case, existing AI models are trained on custom datasets, lead-

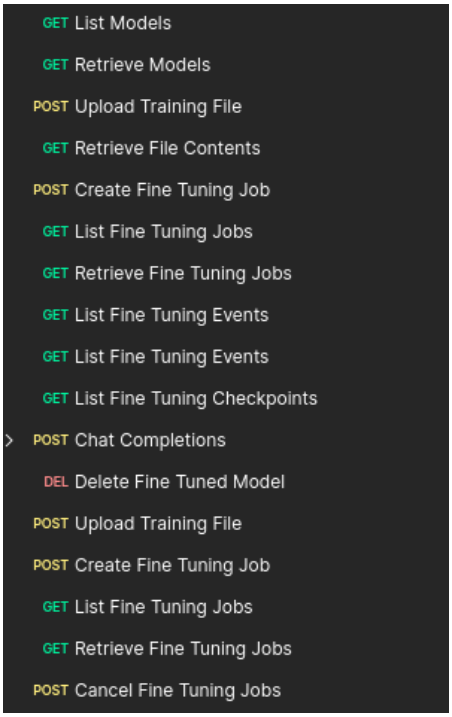


Figure 5.17: OpenAI API - Use case 3 API calls

ing to the creation of new versions of them. Initially, all models are listed, one of them is selected and information about it is retrieved. A file containing training data (regarding some operational procedures of a retail shop) is uploaded and details about its content are retrieved. Based on this training file, a new fine-tuning job is created so that the initially selected model can be trained using this file. All fine-tuning jobs are listed, and information about any events (status updates for the created job) is retrieved. When the job is completed, the training checkpoints are listed, with the final one being the trained version of the original model that resulted from the training data. In this new version of the model, a question is submitted, after which the model is deleted. Then, another training file is uploaded and a new fine-tuning job is created which is finally canceled.

A snippet of the training file is shown in figure 5.18

```
"messages": [
    {
      "role": "system",
      "content": "You are a helpful assistant
      for a retail shop, providing
      information about sales, returns,
      and shop hours."
    },
    {
        "role": "user",
        "content": "How can I return a product?"
    },
    {
      "role": "assistant",
      "content": "To return a product, please
      bring it back to the store within
      30 days of purchase along with the
      receipt. The item must be in its
      original condition and packaging."
    }
]
```

Figure 5.18: A snippet of the training file

5.3 Notion API

The Notion API [25] provides capabilities for managing and sharing content within a workspace allowing the creation of databases that can store

content in the form of pages, blocks, etc. In the use case illustrated in figure 5.19, the users of the workspace are listed, information about one of them is retrieved and details about the bot user associated with the API

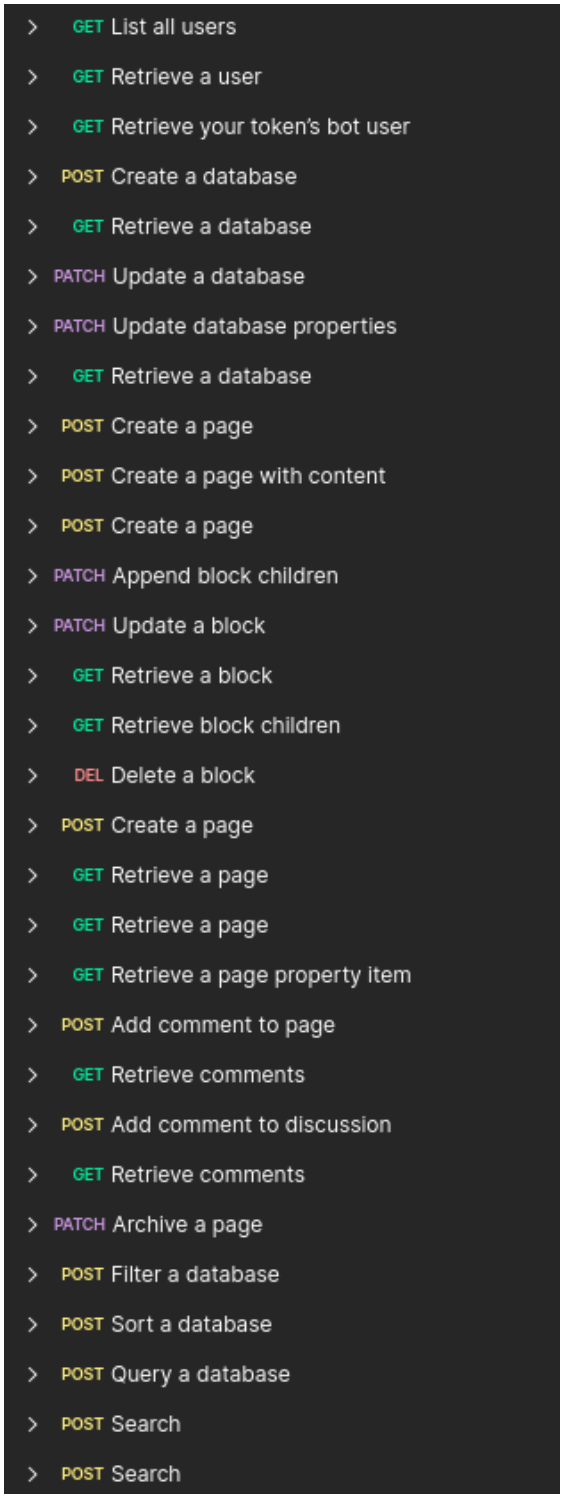


Figure 5.19: Notion API - Use case

token used for the requests are displayed (the Notion API facilitates the programmatic interaction with a workspace by linking API requests to a bot user, as each action must be associated with a Notion user).

Then, a database is created with a schema and content related to the sights a group can visit at a destination. Information about this database is retrieved, and then some fields are updated and a new field is added to the schema.

Subsequently, three new pages are created within this specific database, each corresponding to a record in the database (i.e. a recommended sight). Additional content is added to one of the pages in the form of a block, according to Notion's terminology. This content includes additional information and descriptions about a sight. Furthermore, in one of the pages that initially had no content, a child block with extra content is added, updated, retrieved and finally deleted.

Following this, another page is created, information about its properties is retrieved, and comments related to this page are inserted and retrieved. Furthermore, some queries are submitted to the database, specifically filtering based on the value of a property and sorting. Finally, a search is performed to locate pages, blocks or databases in the workspace that contain specific text.

5.4 Echos API

The dynamic analysis system was finally employed in an API accessible through a User Interface. In this case, rather than directly invoking the API, we can interpose the Man In the Middle between the frontend and backend. By interacting with the application via the frontend, MIM can log the necessary API calls. A prerequisite for conducting such an analysis is unrestricted access to the codebase of the application under examination so as to facilitate the interposition of MIM. Therefore, such analysis is beneficial for developers of an application who wish to generate the desired dependency graph for the API with which the frontend application interacts.

Echos is an application that provides a solver software (GAMS) as a service to address problems associated with energy data provided in the form of time series. After modifying the frontend code to route all its HTTP calls through the MIM to the backend, the following actions were performed by interacting with the UI:

1. **Login**: The user first logs in
2. **List scenarios and create a new scenario**: The user then views all available scenarios and creates a new scenario. The scenario includes

information about the type of problem that GAMS is tasked with solving and the kind of time series and parameters that will be provided as input.

3. **Edit scenario**: Subsequently, it is necessary to edit the generated scenario in order to add the input time series that are to be analyzed.
4. **Run scenario**: The scenario is then set for execution
5. **View results**: Once the analysis of the inputs and the solution of the specified problem are completed, we then view the results
6. **Download output files**: Finally the output files of the analysis are downloaded.

The results of the dependency analysis in this API are presented in the following chapter.

6 Utilizing the modified version of RADAR: Results

This chapter presents the results of conducting dynamic analysis on the previously studied APIs. For each API, representative parts of the dependency graph are shown for both static and dynamic analysis, highlighting differences and drawing conclusions regarding the benefits or limitations of the dynamic approach.

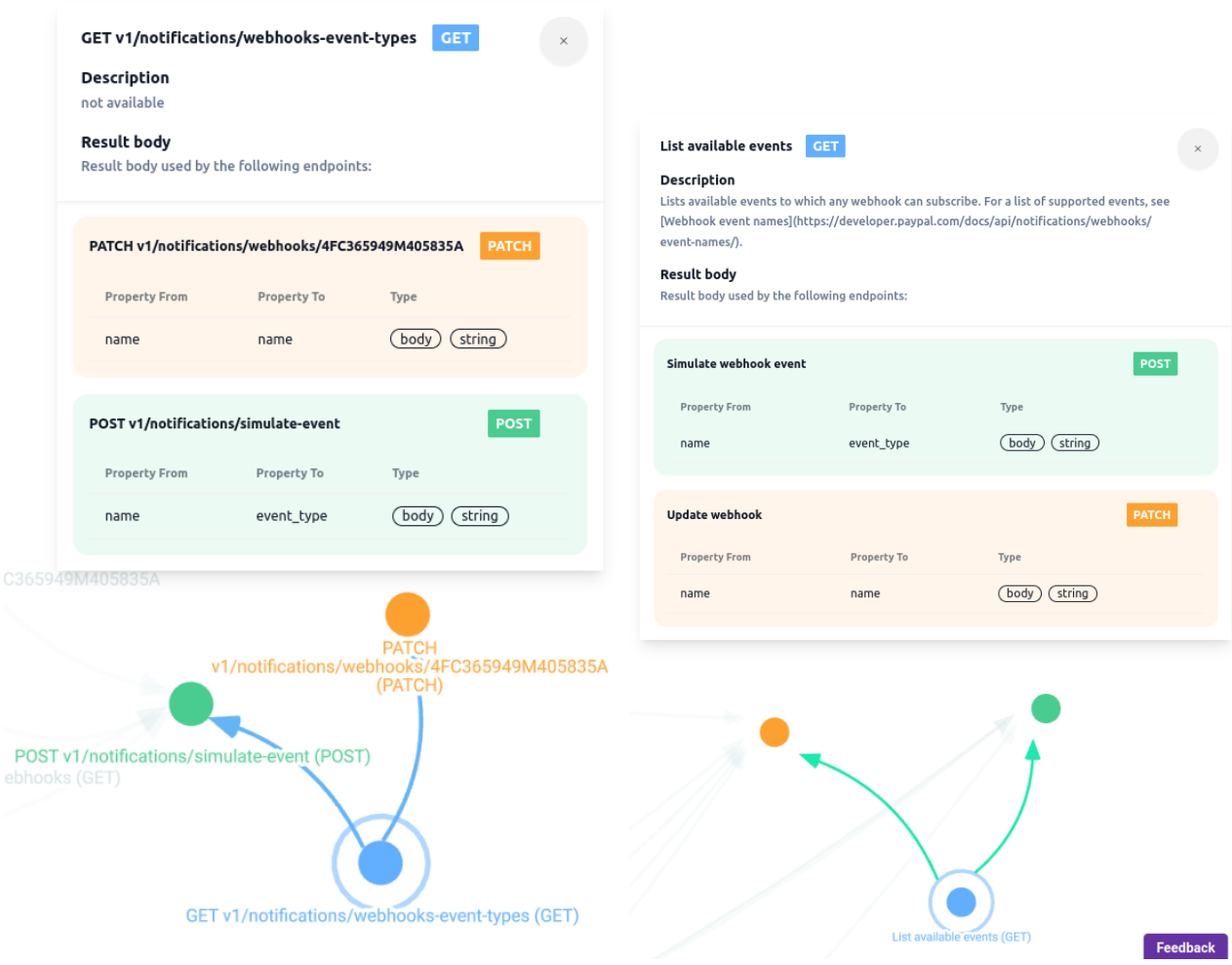
6.1 PayPal API

Case 1 : Same results from both dynamic and static analysis

First, a case is presented where both types of analysis identified the same inter-endpoint (body) dependencies. Specifically, as shown is figure 6.1, the API user is required to retrieve all possible supported events for a webhook and then select one of these events to either display notifications for that event or update a webhook to subscribe to it. The dependencies are based on the event name.

Case 2: Reduced noise in dynamic results

Subsequently, we analyze an example in which the dynamic analysis identified significantly fewer dependencies compared to the static analysis which seems to detect dependencies between endpoints that lack any meaningful relevance. By combining figures 6.2 and 6.3, we observe that the dynamic analysis identified that the API user must first list all available

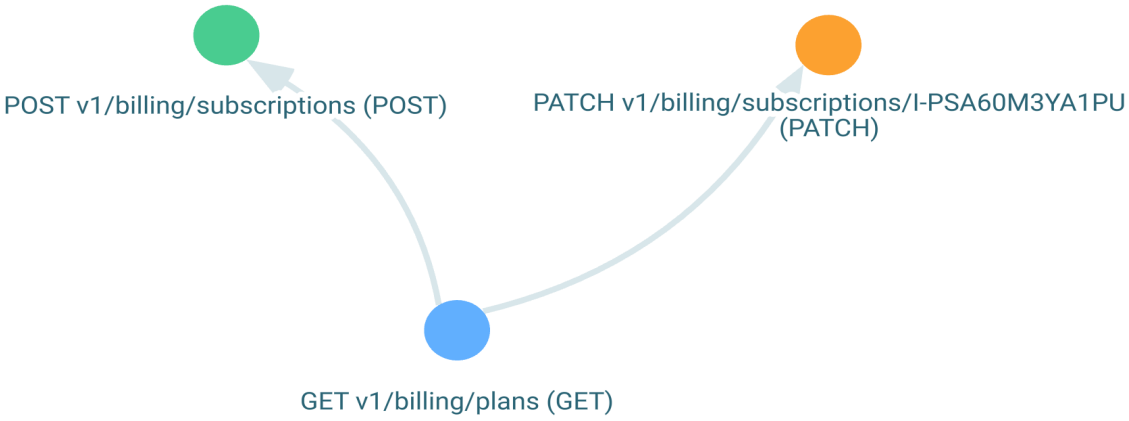


(a) Dynamic (b) Static

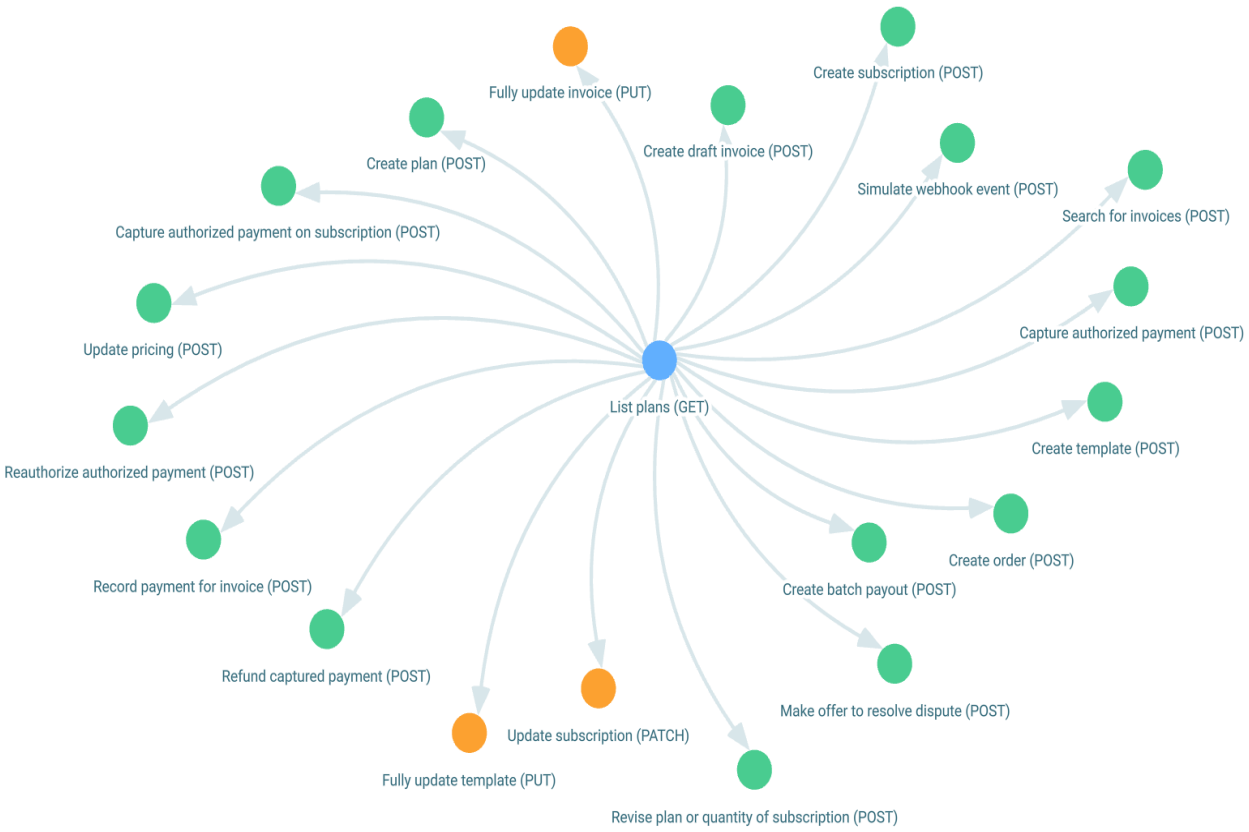
Figure 6.1: PayPal API - Same results

billing plans and then select one of them to create or update a subscription for that plan. The dependencies appear to be based on the attribute plan_id, as shown in figure 6.3, as well as on other, less obvious fields. In contrast, the static analysis has identified a multitude of dependencies that are neither apparent nor correct. For example, there is no reason for a dependency between retrieving all billing plans and refunding a captured payment and creating an invoice or an invoice template.

However, it should be noted that a dependency between the List billing plans endpoint and the Update pricing endpoint, which updates the pricing schema of a billing plan, could be considered logical. This dependency was not identified through dynamic analysis. This example is highlighted as it reveals an inherent characteristic of dynamic analysis, where the quality and completeness of the results largely depend on the initial design of the examined use cases. Referring to the use case depicted in figure 5.10,



(a) Dynamic



(b) Static

Figure 6.2: PayPal API - Reduced Noise

Figure 6.3: PayPal API - Case 2 - Dependent Attributes

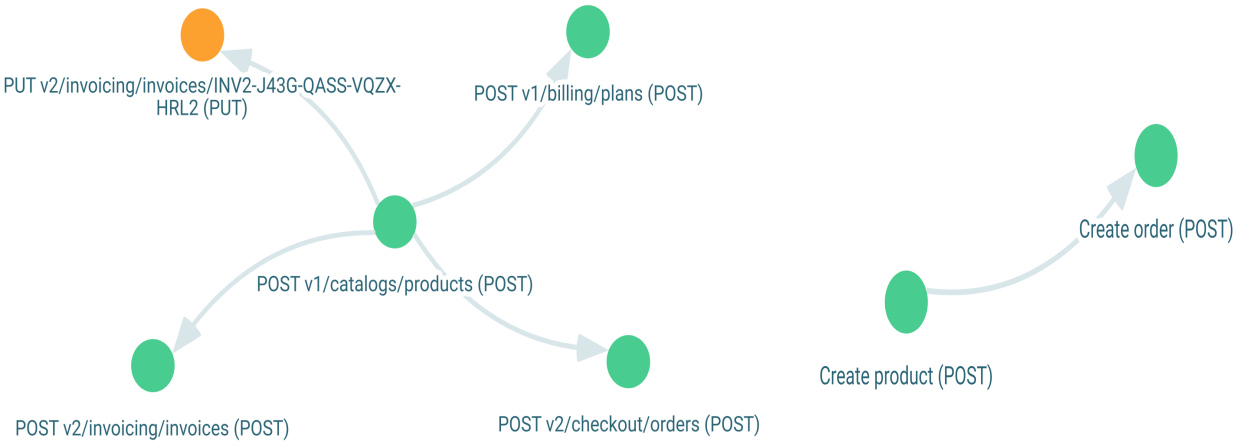
we find that the List plans endpoint follows the Update pricing endpoint, resulting in no dependency being detected during the analysis, even though it conceptually appears logical. This is not a drawback of the dynamic approach but rather a natural characteristic of it, underscoring the necessity of thorough and accurate design of use cases at the initial stage. In figure 6.4, detailed metrics are presented indicating that dynamic analysis identified 2 dependencies, while static analysis identified 19. The common dependencies found were 2 (specifically, the 2 identified by dynamic analysis were also present is static results), while 17 static dependencies were found that were not detected by dynamic analysis.

```
"GET v1/billing/plans": {
    "static_endpoint": "GET v1/billing/plans",
    "numberOfDynamicDeps": 2,
    "numberOfStaticDeps": 19,
    "intersectionOfDepsLength": 2,
    "dynamicAndNotStatic": 0,
    "staticAndNotDynamic": 17
},
```


Figure 6.4: PayPal API - Case 2 analytics

Case 3: Uncaught dependencies in static results

In addition to the fact that static analysis is likely to detect non-useful and misleading inter-endpoint dependencies, it may also prove inadequate in identifying some useful ones. In the example of figure 6.5, dynamic analysis indicates that the API user first lists all available products and then -after selecting one- creates an order or a billing plan for it as well as generates or updates an invoice to include that product. However, static analysis detects only one of these dependencies. As mentioned, the reliance of static analysis on Postman examples, which are non dynamically generated, results in incomplete information about the API's real-world utilization, thereby overlooking valuable dependencies.

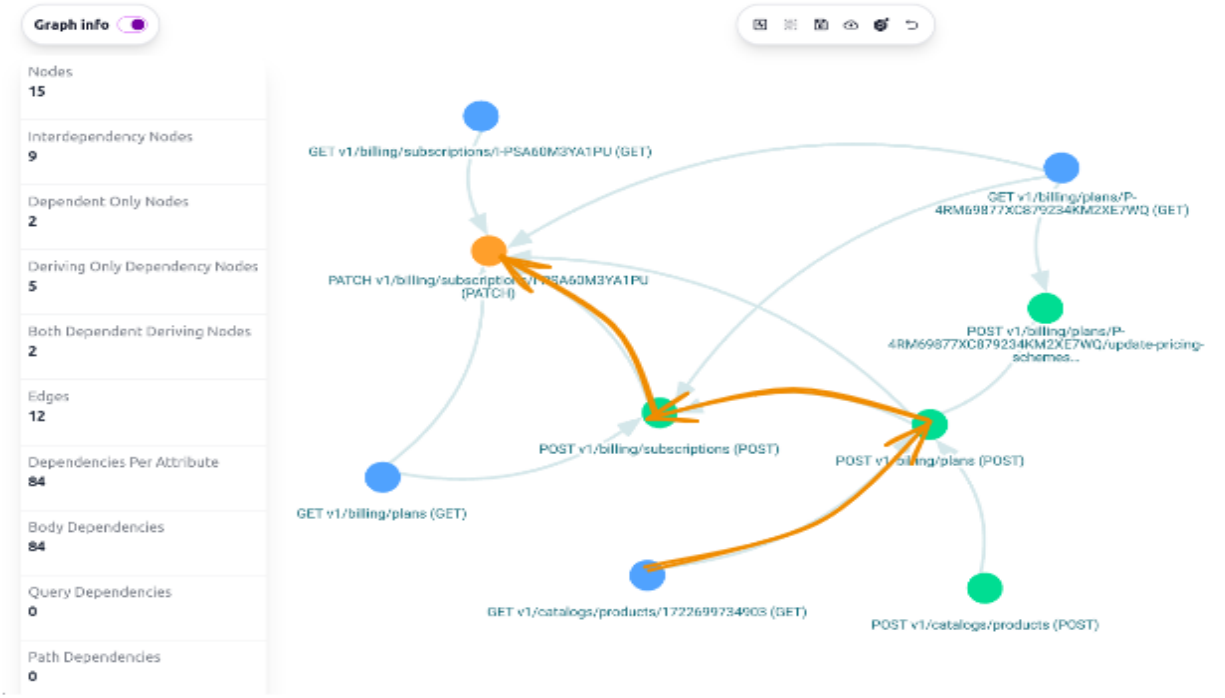


(a) Dynamic (b) Static

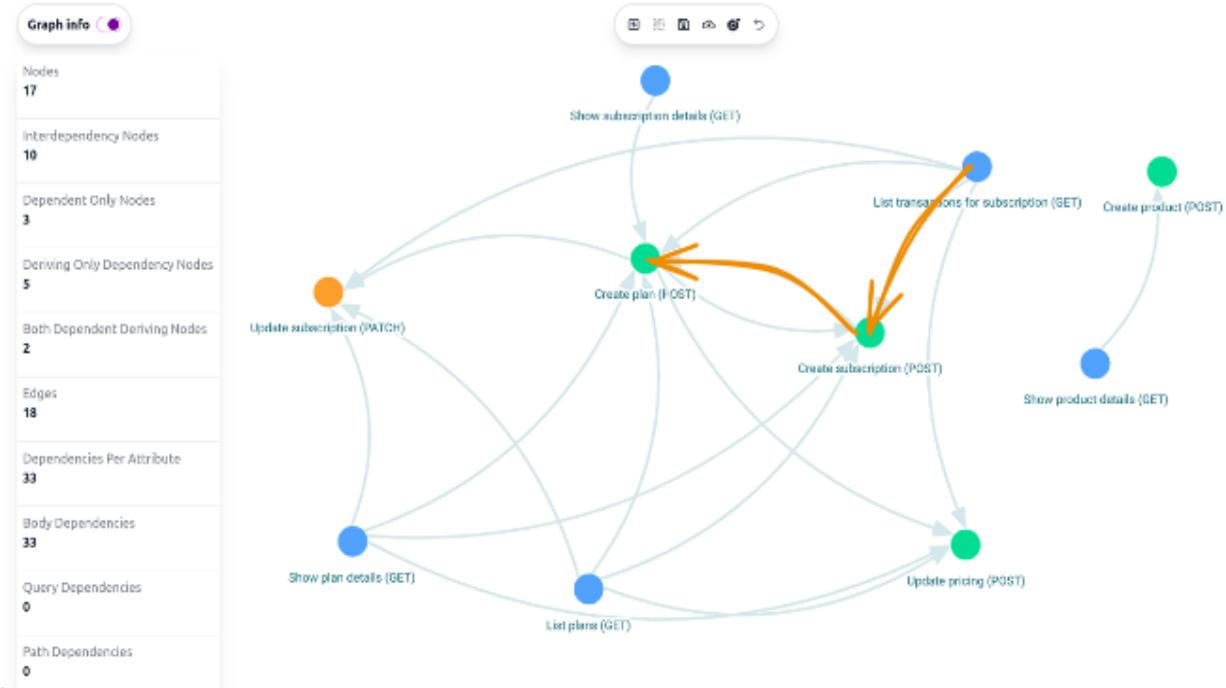
Figure 6.5: PayPal API - Uncaught dependencies

Resulting workflows

Subsequently, for a part of the dependency graph, we observe the workflows that have been uncovered, namely useful dependency paths that high-



(a) Dynamic



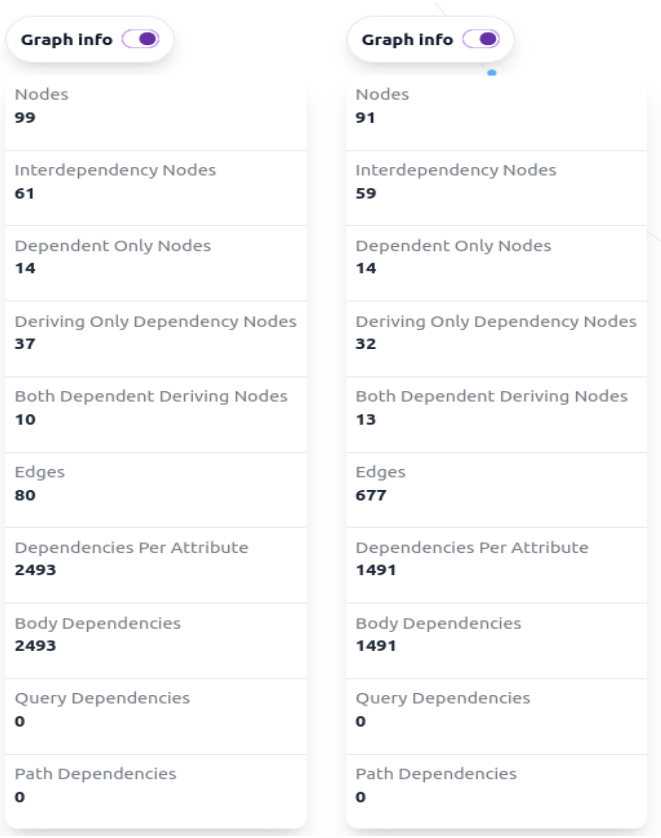
(b) Static

Figure 6.6: PayPal API - Resulting Workflows

light a sequence of API calls implementing specific business logic. By observing such dependency paths within the graph, several functionalities of the API are revealed, providing valuable insights for its proper utilization. Specifically, by examining figure 6.6, a useful workflow within the dynamic analysis graph is uncovered. First, information is retrieved for a specific product, a billing plan is generated for it and then a subscription is created for that plan. Finally, the subscription is updated. We observe that this useful workflow is not present in the static analysis graph, as some valuable dependencies are missing there. In contrast, due to the noise included in the static analysis graph, a workflow has been revealed that is likely not useful. Specifically, it appears that transactions related to a subscription are listed, followed by the creation of a subscription for a billing plan and then the creation of a billing plan. This workflow, however, does not seem reasonable.

Overall statistics and key observations

Finally, some overall statistics are presented regarding the inter-endpoint



(a) Dynamic (b) Static

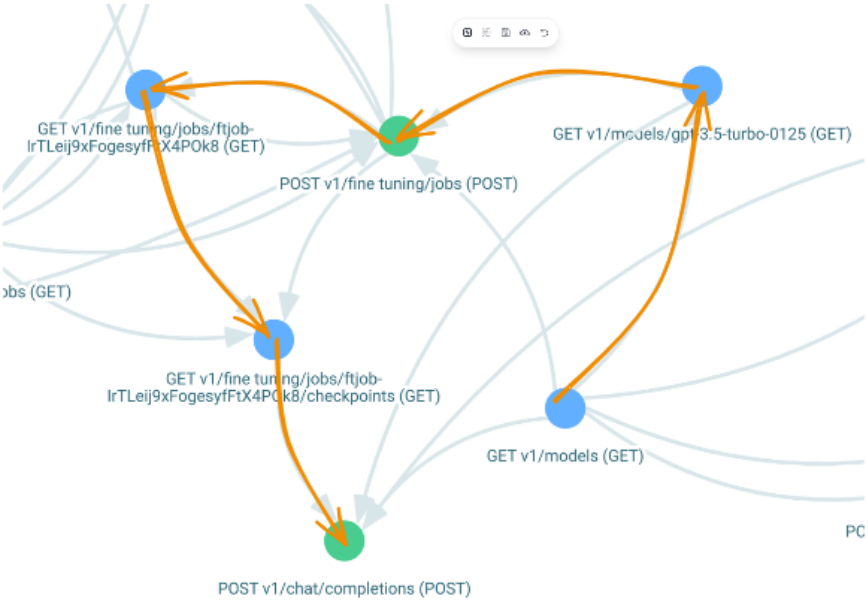
Figure 6.7: PayPal API - Statistics

dependencies based on body parameters. The first notable difference lies in the number of nodes, which were expected to be identical. It is important to highlight that dynamic analysis does not identify the specific operation an API call performs (e.g. Show order details) but treats endpoints as a combination of method and URL. As a result, when the same functionality (e.g. Show order details or the dynamic equivalent GET /v2/checkout /orders/:order_id) is executed multiple times with different path parameter values, as in the various use cases examined here (e.g. /v2/checkout/orders/1 and /v2/checkout/orders/2), these calls are considered distinct nodes in the graph by dynamic analysis, as it cannot detect that the number at the end is a path parameter (there is no available documentation to provide such structural information). This inherent characteristic of dynamic analysis sometimes requires the combination of information from different nodes, which may implement the same functionality but for different objects (e.g. orders).

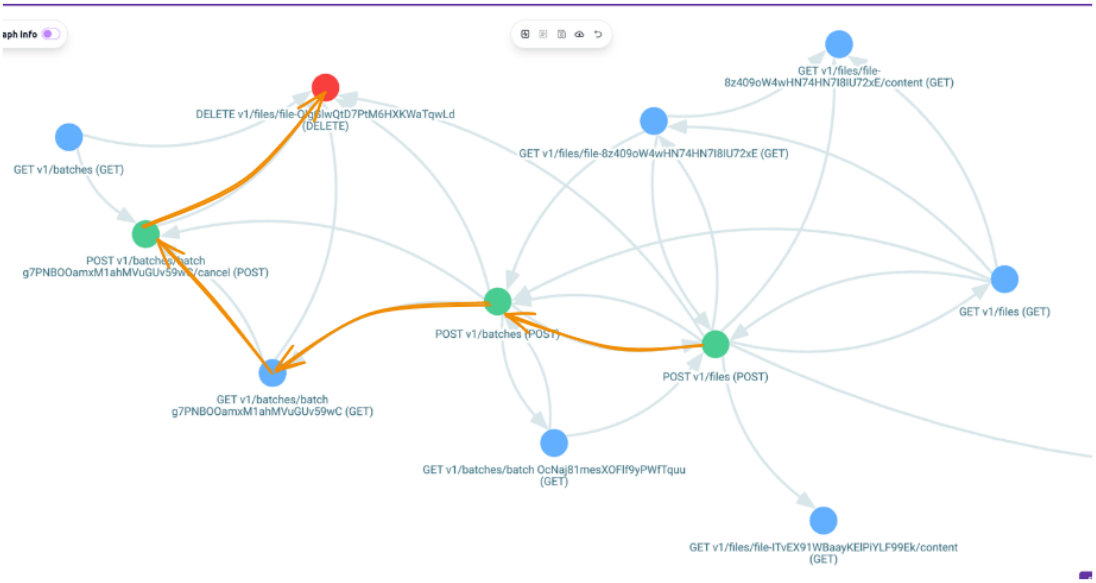
The most significant difference observed next is the number of edges, which is dramatically smaller in the case of dynamic approach, as it filters out much of the noise found in the static analysis. On the other hand, we notice that the dependencies per attribute in the dynamic analysis are considerably higher. Dependencies per attribute represent the cumulative number of dependencies identified between attributes of different endpoints. Since two dependent endpoints can be related through multiple attributes, the combination of fewer edges but a higher number of dependencies per attribute in dynamic analysis indicates that this analysis detects more dependent attributes between dependent endpoints, due to the dynamic nature of request and response bodies. In this way, dynamic analysis provides more detailed information about how two endpoints depend on each other.

6.2 OpenAI API

For the OpenAI API, we present two sections of the dependency graph that highlight potential workflows. The analysis is performed by taking body, path and query parameters into consideration. In the first example shown in figure 6.8, the following workflow is uncovered: first, all available models are listed, and after selecting one, information about the chosen model is retrieved. A new fine-tuning job is then created based on the selected model and information about the job is retrieved. Next, details about the checkpoints of the fine-tuning job are read and with the trained model available, a chat completion query is made. At the same time, by displaying the attributes through which the calls are dependent and assuming the nodes are numbered sequentially from 1 to 6, we observe that:



(a) Workflow 1



(b) Workflow 2

Figure 6.8: OpenAI API - Dynamic Workflows

- The API call 1 provides the id of one of the models as path parameter value in API call 2
- The API call 2 provides the id of the model to the "model" parameter in the request body of API call 3
- The API call 3 provides the id of the job as a path parameter value in API call 4

- The API call 4 supplies the id of the job as a path parameter value in API call 5
- Finally, the API call 5 provides the *fine_tuned_model_checkpoint* field from the response body to the "model" field of the request body in API call 6.

In the second workflow, a new file is first uploaded, after which a batch of requests is submitted based on this file. Information about the batch is then retrieved, and finally, the batch is canceled and the input file is deleted. Regarding the dependent attributes:

- The API call 1 provides the file id in the *input_file_id* field of the request body for API call 2
- The API call 2 supplies the id of the batch as a path parameter value in API call 3
- The API call 3 supplies the id of the batch as a path parameter value in API call 4
- The API call 4 provides the *input_file_id* field as a path parameter value in API call 5.

6.3 Notion API

Subsequently, we construct the dependency graph for the Notion API and examine two workflows that have been revealed. Both workflows begin

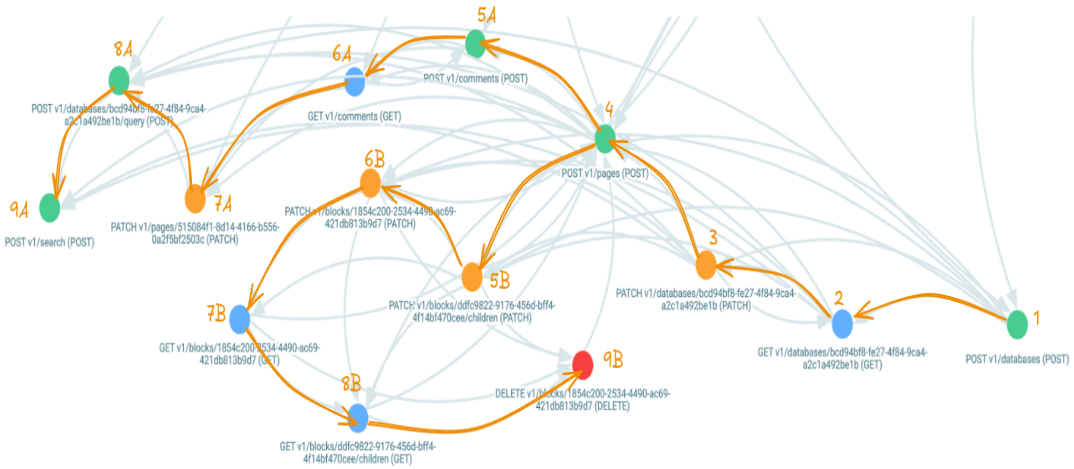


Figure 6.9: Notion API - Workflows

with the creation, retrieval and updating of a database, followed by the creation of a page within it. Then, in workflow A, a comment is created and retrieved on the specific page, the page is updated, the database is queried and finally a search is performed. In workflow B, a child block is added to the created page, which is then updated. The child block and its children are retrieved and finally the block is deleted.

When examining the corresponding graph generated by static analysis, these workflows are not present, as certain inter-endpoint dependencies are not identified. For instance, in the static graph, the path 1-2-3 does not appear. Additionally, the path 5A-9A from workflow A is also missing, as is the dependency path 6B-9B from workflow B. We can also examine the dependent attributes at each pair of successive endpoints of the dependency paths.

By combining the information obtained from the dependency paths and the dependent attributes, we can identify certain functionalities supported by the API and understand the way they are carried out.

6.4 Echos API

Finally, we present the results of dynamic inter-endpoint analysis in the Echos API. We observe that in order to execute a scenario using a solver (endpoint 4), download the results of a scenario execution (endpoint 3) and upload a time series for a scenario (endpoint 5), the scenario and solver

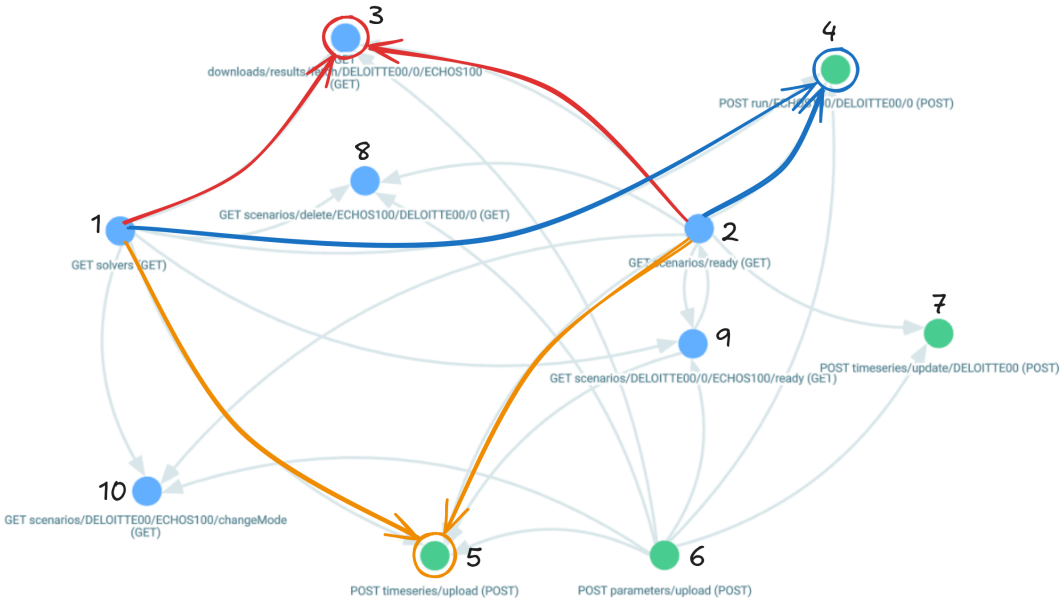


Figure 6.10: Echos API - Dependency graph

identifiers are required. This information is obtained from endpoints 1 and 2 in figure 6.10.

However, in this diagram, we also identify missing and misleading dependencies. This particular analysis was conducted considering path and query parameters. As mentioned before, due to the inability to determine path parameters in a dynamic URL without any documentation available, the analysis assumes that each URL segment is a path parameter. The dependency between endpoints 1 and 2 in figure 6.11 is based on the "ready" value returned by the API call to endpoint 1. Endpoint 2 includes "ready" as a URL segment, which is not actually a path parameter but is recognized as such. This dependency is misleading.

Additionally, we observe that useful dependencies exist between endpoints 3-5 and 4-5, but these were not detected by the analysis. The time series and parameters must first be uploaded and only then can the scenario be executed. However, these are logical dependencies, not data flow dependencies as studied so far and they are neither detected by our analysis nor by the previous static analysis. This inherent limitation is discussed in greater detail in the next chapter.

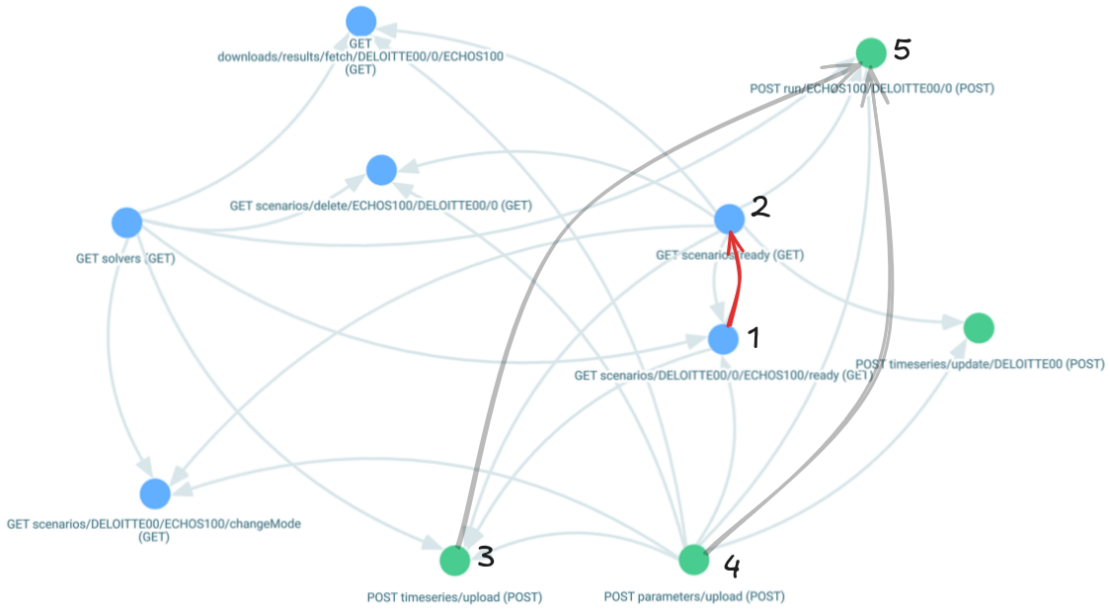


Figure 6.11: Echos API - Uncaught or misleading dependencies

7 Conclusion

This thesis introduced a new -dynamic- approach for identifying dependencies between the endpoints of an API, through real-world interaction with it. The objective was to overcome the limitations imposed by the prior static analysis, which was based on Postman Collections. The functionality, implementation and usage of the dynamic analysis system were presented in detail before the system was applied to four APIs. Following the identification of the API's use cases and their execution in real-world conditions, the resulting dependency graphs were presented and analyzed. The dynamic analysis system was implemented so as to improve the quality and completeness of the static analysis results, though it introduces certain limitations. Dynamic dependency analysis is not meant to replace static analysis but rather to extend it and provide better results when applicable.

This chapter provides an overview of the benefits and limitations of dynamic inter-endpoint dependency analysis, as identified during its application to real APIs in previous chapters. Finally, potential directions for extending this work are provided.

7.1 Benefits

The benefits of dynamic dependency analysis primarily lie in its ability to address the limitations of static analysis.

Specifically, we observed in all cases - particularly with the extensive PayPal API - that employing dynamic analysis significantly reduces the detection of misleading dependencies. This enhances the **accuracy** of the results generated by dynamic analysis.

Additionally, with a comprehensive and well-designed set of use cases for execution, all useful dependencies can be identified, thus eliminating undetected dependencies. This enhances the **completeness** of the results.

Moreover, the dynamic approach **eliminates the need for any Postman Collection** to conduct dependency analysis. Many APIs lack such documentation, making static analysis not applicable for them. Even if time and effort were invested in generating a Postman Collection, conducting dynamic analysis would still be significantly easier. In dynamic analysis, the primary challenges lie in the correct installation of MIM (which is not particularly difficult in controlled environments) and in the design of the use cases (which is not overly complex for those familiar with the API, such as its developers). Once these issues are addressed, utilizing the API through MIM is the only remaining requirement.

7.2 Limitations and challenges

The primary limitation of using the dynamic analysis system lies in the capability of employing the Man In the Middle. Such a mediator system cannot be utilized with any API, as the API itself may impose access restrictions. For public APIs, like the first three examined (PayPal sandbox, OpenAI, Notion), setting up the Man In the Middle is straightforward and is recommended to be done locally. However, for private enterprise APIs or APIs protected by firewalls, this process becomes more complex and MIM must be deployed within the appropriate internal network with the necessary access permissions. In any case MIM can only be utilized in a **controlled environment** with access to the target API.

Furthermore, we observed that the full discovery of dependencies and the elimination of undetected ones largely **depend on the initial design** and the accurate definition of the use cases to be executed. To uncover dependencies between API calls, the corresponding call sequences must be executed during the analysis. In cases where APIs are accessed through a frontend application, this process might be more efficient, as it only requires complete interaction with the UI, which is a simpler process than conducting a thorough study of the API to identify all the possible use cases.

Another issue identified during the analysis of the four APIs is the **inability to determine the path parameters** within the URL due to the lack of documentation indicating them. This results in two challenges. The first challenge concerns the noise generated from detecting inter-endpoint dependencies based on path parameters. The analysis assumes that each URL segment represents a path parameter. While this helps identify these dependencies, it also introduces misleading dependencies resulting from URL segments that are not actually path parameters. The second challenge pertains to interpreting the dependency graph, which can become

more complex. In dynamic analysis, each endpoint is uniquely identified by the method and URL combination. However, the inability to determine path parameters results in API calls being treated as distinct endpoints - separate nodes in the dependency graph - even though they refer to the same endpoint but with different path parameter values. As a result, two or more distinct nodes may actually represent the same endpoint and identifying the dependencies of that endpoint requires combining the information provided by all these nodes.

One final issue concerns the definition of inter-endpoint dependencies themselves. In the context of this thesis, inter-endpoint dependencies are defined as data flow dependencies. Specifically, a value returned as a response of an API call is subsequently used as input for another endpoint. Analyzing such dependencies allow us to identify which other endpoints need to be invoked first to provide the necessary data for subsequent API calls. Additionally, it also reveals potential workflows that uncover functionalities implemented by the API. This information is crucial but it is not entirely comprehensive. Beyond the dependencies examined so far, there is another type of valuable dependency that is not revealed by such a dependency graph. These are logical dependencies that do not involve data flow between different endpoints. As shown in figure 6.11, in the Echos API, two such dependencies exist between the upload of time series and parameters, and the execution of the scenario. The logic implemented by the API requires the scenario to be "ready", with the necessary time series and parameters uploaded, before it can be executed. However, there is no data flow between these endpoints. The dependency is purely based on the logic implemented. While such dependencies are important, they are not examined in this study and require a different approach. One way to detect these dependencies would be to test various sequences of API calls to identify configurations that lead to errors. If a sequence of API calls causes the API to return a message indicating that a specific action is not allowed at this stage along with an appropriate status code, we can infer that the sequence contains logical dependencies that have been violated.

7.3 Future expansions

Bibliography

[1] Key Benefits of API Integration for Developers (with Statistics) https://apitoolkit.io/blog/benefits-of-api-integration/

[2] BUILDING A CONNECTED BUSINESS IN THE APP ECONOMY https://docs.broadcom.com/doc/apis-building-a-connected-business-in-the-app-economy

[3] An Analysis of Public REST Web Service APIs Andy Neumann, Nuno Laranjeiro, Jorge Bernardino

[4] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, June 1999 https://datatracker.ietf.org/doc/html/rfc2616

[5] RFC 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, June 2014 https://datatracker.ietf.org/doc/html/rfc7231

[6] PATCH HTTP method https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/PATCH

[7] What is RESTful API https://aws.amazon.com/what-is/restful-api/

[8] Describing Parameters https://swagger.io/docs/specification/v3_0/describing-parameters/

[9] What is XML? https://aws.amazon.com/what-is/xml/

[10] RFC 4627: The application/json Media Type for JavaScript Object Notation (JSON), July 2006 https://datatracker.ietf.org/doc/html/rfc4627

[11] Which API Data Format Is the Best? JSON Vs. XML, Dan Suciu, May 24, 2021 https://medium.com/api-world/which-api-data-format-is-the-best-json-vs-xml-aaf9b1f94155/

[12] What is a REST API? https://www.ibm.com/topics/rest-apis

[13] About Postman
https://www.postman.com/company/about-postman/

[14] Organize and automate API requests in Postman Collections
https://learning.postman.com/docs/collections/collections-overview/

[15] What is OpenAPI?
https://www.openapis.org/what-is-openapi

[16] What is Sequence Diagram?
https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/

[17] Sequence Diagram
https://www.visual-paradigm.com/learning/handbooks/software-design-handbook/sequence-diagram.jsp

[18] Man In the Middle (MIM)
https://github.com/ntua/rest_api_live_gerokonstantis/tree/main/mim_to_publish

[19] PayPal Payouts API
https://developer.paypal.com/docs/api/payments.payouts-batch/v1/

[20] PayPal Webhooks API
https://developer.paypal.com/docs/api/webhooks/v1/

[21] PayPal REST API
https://developer.paypal.com/api/rest/

[22] OpenAI API
https://platform.openai.com/docs/api-reference/introduction

[23] OpenAI API: Function calling
https://platform.openai.com/docs/guides/function-calling

[24] OpenAI API: Embeddings
https://platform.openai.com/docs/guides/embeddings

[25] Notion API reference
https://developers.notion.com/reference/intro