universität wien

MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„APIs for subscription services for a fact checking system based on Azure Cloud Services“

verfasst von / submitted by

Adrian Thöndel BSc

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2023 / Vienna, 2023

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 066 935
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Medieninformatik
Betreut von / Supervisor:	Univ.-Prof. Dipl.-Ing. Dr.Wolfgang Klas

Acknowledgements

First of all, I want to thank my supervisor Univ.-Prof. Dipl.-Ing. Dr.Wolfgang Klas for his patient support during my work. I also want to say that I appreciate the trust given to me when providing me access to an Azure account.

Furthermore, I want to thank Dipl.-Ing. Dr.Peter Kalchgruber for introducing me to his field of expertise and providing valuable guidance whenever I encountered obstacles in my studies.

Thank you!

Abstract

In today's modern world, a vast amount of data is shared on the internet. A lot of this data can be categorized into an essential entity: a fact. People encounter them online and sometimes it is not sure whether they are correct or carry forth misinformation. This problem was tackled in a past project done by the MIS research group of the University of Vienna. The developed tool, called FactCheck, gathers fact data over time and allows users to compare fact data while browsing websites. The browser extension IdaFix was developed with the purpose to provide an accessible interface for end-users, which both takes care of the communication with backend services and shows processed information in the browser. The starting point of this thesis is based on the results of this research project. In this work, we examine the possibilities of how to monetize the previous project as online APIs on the web. We investigate Azure Cloud Services to see how they may benefit the FactCheck system in terms of scalability and accessibility. In the process, we set up an Azure API Management instance and refactor the existing code to match the restrictions of Azure services. To serve the implementation as distinct API features, existing functionality is refactored into Azure Function Apps, which are exposed through the API Management instance. Furthermore, database logic is realized as a Cosmos database. Azure API Management provides user management by itself and also a subscription system, which allows registered users to access functionalities by the use of API keys. To monetize these APIs, we integrate PayPal as a payment solution. This platform allows money transactions between buyers and sellers and we use it to verify the purchase of API keys. The API Management's user account management feature is extended by Azure B2C, so that the registration process can be adapted to our needs. Research shows that API Management puts restrictions on the developed resources, as Function Apps can only be assigned as a whole to a purchasable product and not their individual methods. While the resource imposes more thoughts to be done for the integration, it eases the publishing process with the offered email service integration and developer portal.

Kurzfassung

Heutzutage werden große Mengen an Daten im Internet über Websites und Soial Media zugänglich gemacht. Dabei kann es problematisch sein, den Überblick zu behalten. Außerdem kann nicht immer garantiert werden, dass diese Informationen korrekt sind beziehungsweise es sich dabei um absichtlich verbreitete Falschinformationen handelt. Um diesem Umstand entgegen zu wirken und Benutzern zu erleichtern, widersprüchliche Informationen im Web zu erkennen, wurde das Framework "FactCheck" in einem vorangegangenen Forschungsprojekt der Forschungsgruppe MIS entwickelt. Mit einer Browser extension für Chrome, "IdaFix" genannt, können strukturierte Daten auf Webseiten hinsichtlich Konflikten überprüft werden, indem sie mit gespeicherten Daten im backend verglichen werden. Die Ergebnisse werden dem Benutzer als Score im Browser angezeigt und geben Auskunft darüber, wie viele Konflikte auf der Seite gefunden wurden.

Aufbauend auf den Ergebnissen aus FactCheck, ist das Ziel dieser Arbeit Möglichkeiten zu finden, das bestehende System im Web mit API-Subscriptions monetarisierungsfähig zu machen. Dafür werden Resourcen des Cloud Anbieters Azure erforscht und auf ihre Eignung für ein API Modell überprüft. Es hat sich herausgestellt, dass der Service Azure API-Management (APIM) sehr gut für diesen Anwendungsfall geeignet ist, weil von er von vornherein eine anpassbare Plattform anbietet, die Account Management, Subscriptions und Freigabe von API Services bereitstellt. Für diese Arbeit wird APIM verwendet, um den Zugriff auf Azure Functions mit Hilfe von Subscription Keys zu ermöglichen. Um das bestehende FactCheck System als Cloud Service bereitstellen zu können, wurden Teile dessen als Azure Functions realisiert. IdaFix wurde dahingehend angepasst, dass Anfragen an APIM gesendet werden. Zum Zwecke der Monetarisierung wird der Sandbox Service von PayPal genutzt, welcher es erlaubt, Zahlungsprozesse zu simulieren und den darauf basierenden Programmcode zu testen. Somit wird PayPal integriert, um Subscriptions mit Einmalzahlungen oder monatlichem Abbonnements zu verkaufen.

Des Weiteren wird das entstandene verteilte Cloud System evaluiert und es werden Herausforderungen und Probleme beschrieben, die während der Implementierung entstanden sind.

Contents

List of Figures

Listings

1 Introduction

To combat misinformation and the spread of misleading claims, fact-checking is a solution to identify incorrect statements and may help to reduce these. Approaches specialize in exploring a specific field, like explicitly verifying Tweets on the social media platform Twitter[16, 14], or checking sentences of newspaper texts for their truthfulness [15]. Especially with the use of social media, misinformation, sometimes in the form of fake news, can quickly spread among the internet [10]. Although, fact-checking services deal with misinformation, there is still the question, if such services can be trusted, or even if people accept the result of a performed fact check [1, 10].

1.1 Motivation

The work of [13] examines the problem that structured data found on different Web pages about the same entity are in conflict. As a solution, they propose a system that helps users to identify conflicting data on the Web. While the architecture allows access by a multitude of concurrent users, performance may decrease if the number of active users gets too high. The scalable computing resources of Cloud providers can be leveraged to supply high user access demands as well as ensuring availability [21]. Other fact-checking projects like Fact or Fiction [16] or the system proposed in [14] make use of cloud providers to host their system or integrate specific cloud resources in the architecture. The goal of this work is to explore the capabilities of certain services of the Azure cloud to make use of the appropriate cloud services for creating APIs for the existing fact-checking system and develop a subscription-based monetization model for these APIs.

1.1.1 FactCheck

This work is based on the proposed fact-checking system of [13, 12, 11] which is an approach to address conflicting structured data on the Web. The distributed system compares facts from websites with a local database that is populated with an enormous number of facts which are gathered over time from various websites. To allow users to interact with the backend of the system, the browser extension IdaFix was implemented. The add-on presents fact and comparison data of the fact-check and focuses on the reduction of conflicting data through crowdsourcing. More details of the project will be described in chapter 3.

1.1.2 Marketing on the Web

The importance of the internet as a marketplace grew over the last decades. Business is made online and digital goods are sold in virtual stores. Existing infrastructure, like PayPal provides it, simplifies the process to monetize software and services. Especially subscription models increased in popularity, and it is a recognizable trend, that it will not change in the near future [4]. On the Web, APIs are made available for developers, either free or access can be redeemed by purchase, which allows companies to sell parts of their program as a service or make access to data possible through them. While it may not be possible to apply an API approach to every system, in this work, we investigate the potential of generating APIs for FactCheck.

1.1.3 Benefits of using Cloud Services

Organizations and companies use cloud services in different fields. The most commonly used cloud providers are Microsoft with Azure[1] and Amazon with AWS[2], despite there are many other providers in the field [25].
In general, they are chosen due to the pricing for computation. They offer hardware on a pay-per-use model and allow fast deployments in different environments and this for a multitude of services. They take care of scaling resources depending on the current traffic and usage without requiring developers to do this manually. Additionally, they offer storage solutions for a variety of data types. The platforms are responsible for the consistency of the data and make sure that no data loss occurs. Furthermore, cloud systems allow creating Web-based applications that are connected to the internet [25].

The offerings and their respective pricing, however, differ between providers, which suggests choosing one provider over another depending on the needs of the project.

1.2 Contribution and Research Questions

This thesis contributes an approach for providing FactCheck APIs through a subscription service for monetization. Products of the cloud service provider Azure, that seem promising for the realization of a subscription service, are investigated to determine whether they are suitable for the project. The desired result is a prototypical system implementation, that allows to sell API functionality. Therefore, this thesis also elaborates an approach to derive APIs from FactCheck and examines how a payment processor can be integrated.

[1] https://azure.microsoft.com/ (Last accessed: 2022.06)
[2] https://aws.amazon.com/ (Last accessed: 2022.06)

This work is trying to answer the following research questions, which also were driving factors in structuring this research process:

1. How can users be served by means of subscription services to use FactCheck? Aspects considered include which user groups of FactCheck, end users and content providers, are the target and how subscriptions can be structured for them. This entails the categorization of subscriptions into tiers, that offer a different set of functionality. Subscription services also need to maintain a flexible configuration of their individual functionality.

2. Which APIs may serve as the building blocks for such subscription services? The challenge is to find a proper bundling of the interfaces of the existing system and provide them as dedicated APIs. It must be investigated, which parts of the system are suitable and which do not fit into an API approach. Subscriptions may also apply restrictions to access and functionality.

3. What is the proper bundling of functionality to form dedicated APIs? It must be determined here, which functionality should be grouped together to forma an API. The decision about the coupling can be done based on system components or to meet the realization of a subscription.

4. How to implement the APIs based on Azure cloud services? The question deals with the problem of choosing adequate Azure technology to build the necessary system structure and APIs. Besides serving bundled functionality as API, the remaining system must be built with resources for user management, data storage and scalability for performance.

5. How can the usage of subscription APIs be accounted for the purpose of billing users? The system must be able to distinguish between free users and paying customers, which in the end may lead to an accounting system. A subscription model for the monetization of APIs must be elaborated, that allows to charge users.

6. How to technically set up a subscription model for the APIs on the basis of Azure cloud services so that users can make use of the subscribed services. In the end, the challenge arises to combine all these considerations and implement a prototype. The maintenance of flexibility in the service configuration is of interest.

The design and the implementation of the desired prototype sets the following goals regariding the functional requirements:

- Realizing the prototype's functionality based on Azure cloud services
- Apply cloud functionalities for scalability.

- Finding a solution for a subscription system, including a free and a paid version, that is applicable for the user groups of content providers and end-users in the context of FactCheck.
- Subscription plans can be redeemed by users to access certain APIs.
- Implementing an API-layer which serves as middleware between end-users and the internal system.
- Elaborate a design approach so that APIs can be assigned to paid or free subscription plans adaptively.

1.3 Methodology

The goal of this research emerged from the findings of previous research topics: Finding suitable means to apply a subscription model to the FactCheck system.

To address concerns regarding scalability and maintenance, it was soon identified, that a cloud-based approach would be the way to go. Azure was the cloud provider of choice due to the knowledge gained from research projects where we used this cloud service. Before implementing a cloud-oriented solution, literature was gathered, which does revolve around the same topics of cloud computing, Azure services and API management as well as subscription models. These papers about related work had then been studied to learn about the state-of-the-art of such systems.

In parallel, we followed an iterative and incremental approach to examine the Azure catalogue and learn more about available products and determine which of the offered services are beneficial for this work and which of them could become a potential part of the prototype. If an Azure product seemed promising, we created an instance of the service to become familiar with the functionality that comes with the product and learn more about how we could integrate this into a prototype. However, we at first chose the cheapest billing plan whenever possible, to minimize costs during the development phase. When we realized that a chosen plan does not include the functionality we needed, we upgraded the instance's billing plan to the tier which fulfilled our requirements.

In this work, we aimed at moving the existing FactCheck prototype from its existing architecture into a cloud environment. Since the original code implemented the Python Web-frameworks Tornado [3] and Django [4], which were responsible for handling HTTP requests, it was not feasible to simply port the application over. We needed to substantially refactor the code to adapt it to the new environment. As we decided to use Azure Functions as the execution environment for FactCheck tasks, we broke down the migration process

[3] https://www.tornadoweb.org/ (Last accessed: 2022.06)
[4] https://www.djangoproject.com/ (Last accessed: 2022.06)

into several phases. First, we identified the most crucial processes that are already implemented in the Python frameworks. We concluded, that the process, which handles requests from the IdaFix extension and performs the fact check, was the most significant. The next phase focused on isolating the affiliated code segments and refactoring them into their specific Azure Function. With the resulting Azure Function, IdaFix could communicate with the HTTP endpoint instead and the initial Python Web-frameworks were no longer required. After we have learned from the first successful migration, we applied the same phases when creating the Function app for providing precision metrics.

During the migration process, whenever databases were required, we mocked data to test the implementation. To find a successor for these two databases, we selected the cloud-native CosmosDB as our database and experimented with the SQL and Mongo core APIs to get familiar with this type of database and its capabilities.

Over time, Function apps and the quantity of collections in CosmosDB were updated incrementally as development of functions progressed, or database access was established.

1.4 Outline

The contribution of this thesis is to develop subscription services. The work is structured in the following chapters:

Chapter 1 serves as the introduction to the field of research and the background for the research question. It introduces to the previous project FactCheck, which is the starting point for this work and spans the arc from the motivation of using cloud services to the pursued methodological approach.

Chapter 2 examines projects which are related to the fields of fact-checking, cloud services and monetization through subscription services. It also features previous research topics of FactCheck, which led to the starting point of this thesis.

Chapter 3 is dedicated to FactCheck and explains the architecture of the prototype and the functionality of its components. This chapter conveys knowledge about the framework, which will be important to have in mind when reading the following chapters.

Chapter 4 summarizes the challenges encountered while elaborating the design for the final implementation of the system architecture. Furthermore, the thought process behind decisions to tackle encountered problems is explained.

Chapter 5 describes the final implementation of the system and explains the cloud resources used. It gives insights about the underlying processes of the working prototype in textual explanation, code examples and also provides a variety of diagrams.

Chapter 6 shows the results of qualitative analysis and evaluates the state of the research. It recapitulates the development costs and gives an outlook on a production scenario.

Chapter 7 concludes the thesis and discusses the results and insights of this project. The

limitations that restrict the outcome of the theses are outlined. In the end, possibilities for future work are discussed.

2 Related Work

This chapter presents related work and organizes content into subchapters, according to the related topic. Before diving into the area of fact-checking, a set of reoccurring terms in this field shall be mentioned, which were elaborated and summarized within the work of [9].

Fact-checking deals with false information, where false information can be considered as subdivided into misinformation, which refers to misleading information that was published unintended and disinformation, which categorizes the malicious spread of wrong information. Another term is fake news, used to describe the spread of purposefully manipulated news to pursue a malevolent goal.

2.1 Websites for fact-checking

One specific approach to verify claims tackle misinformation on the internet, is by providing fact-checking services on websites. Organizations provide their tools online and therefore maintain their own websites, which serve the purpose of determining the veracity of fake news and communicating facts [19]. While these websites share the goal of fact-checking, their implementation differs depending on their organization's philosophy and main emphasis:

Driven by a non-profit organization, the website FactCheck.org[5] aims at reducing the deception and confusion of in U.S. politics for voters. They make use of journalistic practices while monitoring various resources, like speeches, TV shows and interviews and determine the accuracy of political claims. This is achieved by manual research done on the gathered material. If misleading statements are found, they search for evidence in primary information resources, for example libraries.

Different to that, the website Snopes.com[23] focuses on a variety of topics and material for the fact check. Resources can be submitted by users. The staff maintaining the website advances their work with journalistic practices and conducts research for possible misinformation. As their trusted resources, they use agency statistics, journals and examine articles, books, and statistics.

Additional to political claims FullFact.org[6] also includes online content that went viral and aims at reducing the spread of false information by providing a corrected suggestion with the link to the original resource. As an independent group, they are funded by charity.

On their website, users can submit a possible wrong source for checking. Additionally, they developed a software to automate the fact-checking process. This means, that data is gathered from news sites and media platforms and the information is transformed into textual representations and broken down into sentences. From these sentences, claims are classified to help filter the amount of information for human fact-checkers. While the information gathering process happens online, they perform the actual fact check offline and make the final result public on their website.

2.2 Automated fact-checking

While these organizations put in a significant effort in examining potential fake news or misinformation, the sheer amount of data in social media and on the web, in general, exceeds the resources of those companies and makes fake-news detection cumbersome. In the past, several projects designed and implemented prototypes, that address this problem by automating the process of classifying facts and checking them. Depending on the project, different external tools are used or a special architecture is implemented.

One prototype that performs fact-checking automatically is ClaimBuster [8, 7]. The system addresses the problem that human fact-checkers cannot keep up with the amount of data. The complex system consists of multiple collaborating components that constantly track social media channels and seeks potential claims in sentences. The *Claim Monitor* component is responsible to retrieve texts from multiple sources. It is capable of reading captions from media broadcasts, filtering political Tweets from preselected accounts utilizing the Twitter streaming API and is as well capable of gathering information about political sentences from texts on websites. *Claim Spotter* is responsible for assigning a score to a given sentence. The value denotes whether the sentence contains facts that should be checked. The classification model learned from a set of pre-labelled sentences from elections. In a following step, the *Claim Matcher* queries fact-checked data from a repository that match the claim in the sentence. The similarity of claims is either determined by matching tokens or semantic characteristics. The claim is further processed by the *Claim Checker*, which gathers evidence for the claim by calling external knowledge bases of Google and Wolfram Alpha and structures the result based on the context. At last, *Fact-check Reporter* creates a report that is visualizable on the project website.

The architecture of fact-checking systems differs between projects and use cases and often leads to complex distributed systems. Moreover, the source of facts differs. The system proposed in [13, 12] features five major components, which encapsulate data management and computation tasks. The purpose of the project is to identify conflicting structured data on the Web, which is used as the data for fact-checking. It contains two databases for storing acquired fact data and tags from user feedback, respectively. A management server to maintain precision metrics and a server for performing the

comparison of facts. At last, a service is used to present information and notifications of the system. The approach for identifying a fact, determining if there is a conflict and presenting a result is tackled differently by projects. It remains the question of how results are evaluated. The evaluation can be subject to users, as [11] states, "the user or use-case determines the quality of the data". To measure the similarity of facts, they first transform given data into a unified format and compares the resulting data set to an internally maintained reference set. In this step, a deviation function is computed using normalized Levenshtein distance.

The work presented in [18] proposes a Semantic Web platform for automated fact-checking. The project aims at increasing the accuracy of automatically populated fact-checking knowledge bases. They use a multistep approach to evaluate the veracity of facts, where Resource Description Framework (RDF) is used to represent facts as a composition of subject, object and, predicate. The first step determines the eligibility of a fact by searching for synonyms of the fact's predicate and generating a predicate set. By comparing the set to data in the knowledge base, a possible match of predicates may be found. The result of this step is a list of facts that also match in the object range and subject domain. In step two, the system seeks supporting evidence and therefore retrieves objects from DBPedia by sending the subjects and predicates of facts as input. The retrieved data is the input for a machine learning algorithm. This regression algorithm is the next step of the procedure and evaluates the veracity of a fact where similarity measure is applied.

Another challenge of the process is the final output and its presentation. Some prototypes return a true or false classification for claims [8, 7], while others use similarity measures to describe the degree of veracity of a given statement [13, 16]. Besides the classification, there are also approaches, that utilize artificial intelligence to train a model for fake-news identification or machine learning for fact-checking [18].

2.3 Using Azure

The work presented in [16] features the implementation of an application for sentence classification to help users fact-check on opinionated news. The algorithm checks if sentences have an objective meaning or not, in contrast to classifying claims as true or false.To analyse the objectiveness of sentences, they use the service Azure Machine learning. The classification algorithm learns from a labelled dataset of five thousand sentences retrieved from various news sources. Although developing for a cloud system, they stick to the software engineering pattern Model-View-Controller, to organize the structure of the Web application. The Model section encapsulates user data and submitted text entries. The Controller part of the pattern handles updating models and rendering views. A website implements the user interface of the application by using React which

was hosted with Azure Web services which allows them to scale their website depending on user sessions. For sentence classification, The Web interface enables users to submit their text, as well as to explore the results and vote for the truthfulness of the calculated objectiveness of sentences.

The Azure Cloud platform has many services to offer, that can be integrated in a fact-checking service. Azure Cognitive Service API is used in the work of [24], where the service is used to recognize facial features in images.

2.4 Using Cloud Services in General

Storage management is an important factor to consider when there are high quantities of fact data. As suggested by [2], a possible solution is the usage of a cloud-based storage solution. With the use of cloud-managed storage, data can be backed up and protected against data loss. Furthermore, it increases the ability to reuse data among applications. In addition, different database solutions are provided, to store different kinds of data. Another cloud-based approach is presented in [14], where a cloud-based tool for validating Tweets posted on the social media platform Twitter is developed. Therefore, they deploy a RESTful Web application on Amazon AWS. Like in [16], a website is used to interact with users. There, users can submit the URL to a tweet, which should be scheduled for validation. After validation in the backend, results are presented in the interface.
To deploy their code, they utilize an S3 Bucket, which stores the Web application. The upload and deployment of code updates follow a continuous integration and continuous deployment model. This is achieved by deployment pipelines, which apply validation tests before storing the code in the cloud. Additionally, the final deployment process is automated by Amazon CodeDeploy, which takes care of pushing the code to an EC2 instance and deploys the resources in the environment. There, a Tomcat server is hosted, which provides the REST website on the internet. Computations are performed in the EC2 environment due to their scalability mechanics.

As presented in [16], [8] also introduces a web-interface for user communication. However, they extend the possibilities further by introducing a Slackbot which allows users to communicate with the system. After adding the bot to a group in the messaging tool Slack[1], statements can be submitted for verification by sending a message. This survey explores techniques on how the architecture of fact-checking applications can be built and deployed in a cloud environment. It was found, that many use cloud platforms to leverage their computational opportunities as well as for their variety of offered services. Although, while they aim at automating the procedure for fact-checking, there are still sections, which rely on manual user intervention.

[1] https://slack.com/ (Last accessed: 2022.06)

2.5 Monetization Models in Subscription Services

Lately, the selling of subscriptions became the standard as a confident business model. A subscription revolves around the concept of a recurring payment plan with a certain price that is settled in regular intervals of time. Over the years, it can be seen as a trend that companies transition to subscription-based models or adapt their business model accordingly [20, 4].

In [20] research is conducted on how subscription models can be beneficial for conventional manufacturing businesses.

Furthermore, subscription services are predominant in the multimedia sector. Streaming platforms like Netflix[2], Spotify[3] and YouTube Premium[4] provide service access or additional content, based on the purchased plan. Customers can select from one or more pricing tiers. Even Microsoft sells their cloud services as a subscription-based product[5].

Cloud service platforms provide a variety of products relating to different aspects. They are commonly categorized in IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service). Moreover, those platforms sell their products by their pricing models, which is investigated by [3]. Platforms charge resource usage based on a pay-per-use model, or a subscription. They find that customers generally prefer the former, however, customers who use the system extensively may prefer a subscription.

2.6 Developing APIs

An Application programming interface (API) is a general term for a software part, that enables a connection between two separate systems and allows the exchange of data. As mentioned by [26], this is a way of exposing information on the web. In their research, they examine the additional management costs that are imposed by maintaining API systems. One concern that arises is the security of the endpoints, which is investigated in [22]. To protect access to underlying cloud infrastructure, they develop their custom access control mechanism based on two layers in the API itself, which build on each other. The first is an access control policy, where user credentials are gathered, and it is verified if access for this user is legitimate. From this data, a user can be assigned to a role, which acts as the second level of access control and determines, which parts of the underlying infrastructure can be accessed.
[17] focuses on the security threats that have to be dealt with when granting access to APIs with access keys. The concern boils down to the problem that the API key may be exposed in plain text, either during the initial download, while interacting with the cloud

[2]https://www.netflix.com/ (Last accessed: 2022.06)
[3]https://www.spotify.com/ (Last accessed: 2022.06)
[4]https://www.youtube.com/premium (Last accessed: 2022.06)
[5]https://azure.microsoft.com/en-us/pricing/purchase-options/pay-as-you-go/ (Last accessed: 2022.06)

service or from its storage. To tackle this, they suggest incorporating a piece of hardware, a secure element (like banking cards), in the transaction process for API keys between the service provider and the client. The secure element takes care of the key as well as signs API requests by performing cryptography operations based on the stored key.

3 Background

This chapter is dedicated to FactCheck, the work proposed in [13, 12] and currently is a project under active development. Today's research projects run under the name **FactCheck++**. However, in this thesis, we refer to the project with FactCheck, how it was initially proposed. The initial implementation resulted in a distributed system, with the goal to detect conflicts in structured data on the Web. This approach compares described information about real-world entities and determines conflicts based on similarity measures.

The project settles in between the actors of data providers and data consumers. The former are responsible for publishing structured data. It is possible that available information deviates across the domains of different data providers. Data consumers are dependent on the data offered and would benefit from improved quality of structured data on the Web.

The FactCheck framework forms the starting ground for this thesis. For this reason, it is important to be familiar with the components of FactCheck because it helps to understand design decision and challenges encountered in this thesis.

3.1 Problem Formalization

The FactCheck framework developed in [13, 12] identifies conflicting data on the Web and proposes a possible solution to help reduce these. It determines conflicts by comparing information about fictional or real-world entities from data sources. Websites like IMDb[1] and RottenTomatoes[2] are examples for possible data sources. Data, encoded in semantic markup formats like Microdata or JSON-LD, is retrieved from a data source about an entity, called an instance in the context of FactCheck. An instance is described by a set of facts, which form a value pair out of the fact information and a defined data type. Values of facts are the elements used for the comparison of instances, whereby values of equal facts in all instances are eligible for this step. The representation of values on various data sources can occur in variations. To achieve better comparability, predicates provide a way to classify facts and measure the similarity of two values and in the end determine if the values are conflicting or non-conflicting. To handle varieties of predicates that derive

[1] https://www.imdb.com/ (Last accessed: 2022.06)
[2] https://www.rottentomatoes.com/ (Last accessed: 2022.06)

from a high number of facts, precision metrics combine a set of related predicates by logical operators. Metrics apply to facts in the form of precision metric classes based on a similarity predicate, which in turn defines the allowed degree of deviation of a comparison result before it is determined as a conflict. Classes dynamically constitute from similarity predicates, that are built over time from crowdsourced user feedback.

3.2 Basic Terminology

In the context of FactCheck, several terms reoccur frequently. A list of basic terms, which is relevant for this thesis and which partially already has been agreed on in the broader context of FactCheck, can be derived [9]:

Entity An entity is a self-contained thing that can be described with facts. They can be fictional or non-fictional concepts or phenomena.

Fact A fact is a piece of information about an entity. It can be thought of as a key-value pair.

Fact Set A fact set is a collection of unique facts about an entity.

Instance An instance is an occurrence of an entity and its affiliated fact set on a certain data source.

Comparison Result When comparing fact sets from two different instances, there are four possible outcomes for a comparison of fact values:

1. equal: fact values are determined as equal based on a certain precision metric
2. conflicting: fact values are considered conflicting with a given precision metric
3. local missing: a fact is missing in the instance from the local page but is present in the remote instance
4. remote missing: facts which are only found on the local page and are missing in the remote instance

Approval When comparing fact sets about an entity from different domains, it is possible, that a fact is conflicting with the information on one site and non-conflicting with a different source. Approval is a percentage value, which informs with what percentage a fact corresponds with the values on other instances.

Unique Facts Refers to the special case where a fact is only present on the currently visited website, but is not present in the remote database.

Predicates Predicates are used to automatically detect if two values of two different facts are in conflict or not, by measuring the similarity of the given values.

Precision Metrics Used for the comparison of facts by using a combination of similarity predicates. It helps to manage the complexity of a potentially high variety of predicates that come with a magnitude of facts.

3.3 Backend Architecture of FactCheck

The architecture of FactCheck is depicted in figure 3.1, showing the existing components, their individual responsibilities inside the system and the interaction flow of each component.

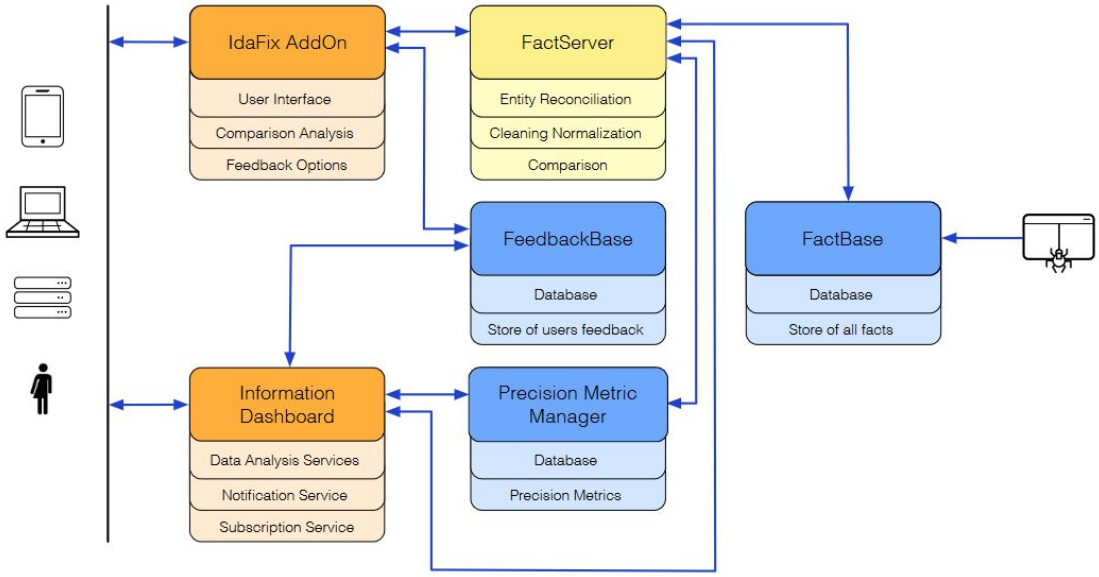

Figure 3.1: Overview of the System Architecture for FactCheck proposed in [13]

In the following, the components of FactCheck are described:

1. FactServer
   This backend component can be seen as the central component of the system, which is responsible for comparing facts and communicating with the internal database. With the Entity Reconciliation Sub-Component, a set of related objects can be determined for a given object, according to the probability confidence value. FactServer also provides a set of APIs to access fact data about a given object, as well as data sources related to an object. In the Comparison Sub-Component, FactServer performs the comparison of facts between two or more objects. The data for reconciliation and comparison are retrieved from the FactBase. FactServer is also involved in populating the database with new fact data.

2. FactBase
   FactBase resembles the internal database which stores objects and the facts and values associated with them. There are three major methods that extend information inside the database. Firstly, the database can be set up with structured datasets which already exist. Another option is to insert data acquired by Web crawlers. Lastly, applications, such as a browser extension, can send new data to the FactServer component, which passes information to FactBase.

3. FeedbackBase
   This backend component handles user feedback according to the idea of a crowd-sourced solution. It provides APIs to accept new user feedback as well as endpoints which allow retrieving stored feedback data about facts, which may be conflicting. The FeedbackBase is composed out of a database, which is used to store feedback data and a component that builds and enhances precision metrics based on the acquired feedback.

4. Precision Metric Manager
   This is a backend component which provides precision metrics for facts and also provides interfaces to manage precision metrics. Based on the feedback acquired from the FeedbackBase, the manager component also deals with the improvement of metrics by using services of (un-)supervised learning.

5. Information Dashboard
   This is another component located in the backend of the system and is used as an interface to manage data sources and facts. It is oriented towards expert users who are interested in using fact data for analysis purposes, but can also provide interfaces for third-party tools.

3.4 IdaFix Browser Extension

The name of the extension constitutes from the vision to "identify and fix structured data on the Web". It is implemented as a browser add-on and serves the purpose to visualize and compare structured data. It gathers fact data from the currently visited website, makes a request to the FactServer for fact comparison, and after receiving the result, notifies the user about potential conflicting facts on the site. To display the fact data, it realizes the user interface that allows data consumers to interact with FactCheck.

The resent implementation updates the visual interface and also adds new features. Target users, addressed in this implementation, are reflected in the categories casual and expert. The former category defines users who have the extension installed to receive notifications about potential conflicts while browsing. They have no profound knowledge about procedures in the backend, nor are experts in the domain. The category for expert

users includes persons like analysts, data scientists and researchers who are professionals in the field and thus want to use the extension to receive elaborated statistics. Based on these categories, the interface changes the functionality and amount of visualized details, to either reduce complexity for casual users or let expert users leverage IdaFix' complete set of features.

IdaFix presents received comparison and fact data in its user interface. This includes the page statistics of the currently visited website in the form of an internally calculated page rating, and also showing the number of facts that were taken into consideration. Additionally, a bar chart depicts the ratio of conflicting and equal facts that were determined for the visited website. Furthermore, data about particular facts can be viewed To support the exploration of result data, a variety of filters can be applied to shift the view towards the used precision metric, the comparison result type such as conflicting, non-conflicting, local-missing and remote-missing or one of the remote websites that were used in the comparison. The interface adapts to the user type, offering expert users more detailed statistics.

Another responsibility of the add-on is providing a feedback interface, where users can send their subjective assessment about a given result. With this feature, it is possible to report incorrect comparison results, precision metrics as well as incorrect facts on the local website. Users submit feedback from a form, which IdaFix transmits as a JSON object to the feedback server in the backend. The data is stored and processed to generate data reports.

4 Design

Since the beginning of this project, we were certain that the implementation of the existing FactCheck framework, cannot be directly deployed on a cloud platform without any changes to its structure, and we may reconsider how components are separated and communicate with each other. We chose Azure as the target cloud platform and hence exclusively used products from this cloud provider. For this reason, we had to deal with possible restrictions when using cloud products, and we were aware, that this decision would heavily affect how we will refactor or redesign the original system.

Another part of this project was to explore and learn about the product catalogue of Azure. This implies that we were not familiar with all the cloud products beforehand and were not sure, which of them would be the ideal solution for this migration purpose. We also figured out during development which pricing plans we should consider. Thus, we constantly learned while experimenting with Azure products, which also meant that we sometimes had to reconsider the choice of a resource we wanted to use or the idea of how we could migrate the existing code. This section contains records of these mentioned cases and describes what challenges occurred, how they affected our design decision, and how we developed a solution for them. Furthermore, we introduce the Azure resources we examined.

4.1 Refactoring into Azure Functions

The code of the original FactCheck framework is realized in the programming language Python. For this project, we aimed to also program our components with Python, to avoid administrative overhead if we translated the existing code into a different language.

The requirements for the integration into the cloud environment and the selection of a suitable service were threefold. Foremost, the service must support our programming language of choice, to keep the refactoring process manageable and potentially allow reusing existing components. Second, the Web frameworks used in the original prototype defined interfaces accessible via HTTP requests. Internal components require connections to backend databases and external services. The target platform should allow the creation of similar interfaces, and it should also be possible for components to reach external endpoints. Lastly, it should be possible to bundle certain functionality we implemented and expose them as an API service.

Due to the listed requirements, we chose to use Azure Functions[1] to implement backend logic, fact comparison and APIs. In the Azure environment, Azure Functions are hosted as an application, that is a container for a potential multitude of functions.

A trigger[2] invokes a function and defines what type of call must be made to execute a function. While there is a great variety of triggers to choose from, we describe the ones that we considered as potentially relevant in our work in more detail. An HTTP trigger starts a function upon receiving a request to its Web endpoint. A timer trigger is useful to run a function after a certain amount of time or recurring interval. To execute a function event if a file changes in an Azure Blob storage, a Blob trigger can be used. A CosmoDB trigger can be set to intercept updates to documents in a database and invoke a function to handle the event.

Additionally, Azure Functions can embed bindings as a means of connecting to an external resource. Input and output bindings are available and are defined in the settings of the app, specifying the credentials. An example usage of bindings in a Function app would be to connect to an external CosmosDB with an input binding and query documents before executing the logic of the function. In an output binding, results can be written to the database, when the function finished.

Users of Azure Function services can choose from one of three plans to define the way resource usage is priced. The consumption-based pricing plan provides dynamic scaling for Function apps and charges costs based on computing time and memory usage. A premium plan incorporates pre-warmed instances alongside automatic scaling, however charges extra costs for this service. The Azure App Service plan is intended for functions that run tasks with long computing time.

Functions of an application can be written in a variety of supported languages, including common languages like C#, JavaScript, Java, and Python.

Before deployment, Azure Functions can be developed and tested on a local machine. For this work's development, we used the IDE Visual Studio Code (VS Code)[3]. In addition, Azure Functions Core Tools[4] must be installed on the local machine, providing a local environment to run and test Function apps. Furthermore, we used Core Tools to log in to our Azure account to deploy our local Function projects to a Function app in Azure.

4.1.1 The Environment

Azure Functions are capable of hosting Python application. The respective environment has to be selected when initially creating the application. Required Python modules must

[1] https://azure.microsoft.com/en-us/services/functions/ (Last accessed: 2022.06)
[2] https://docs.microsoft.com/en-us/azure/azure-functions/functions-triggers-bindings (Last accessed: 2022.06)
[3] https://code.visualstudio.com/ (Last accessed: 2022.06)
[4] https://docs.microsoft.com/en-us/azure/azure-functions/functions-run-local (Last accessed: 2022.06)

be specified in the requirements.txt file of the app and are downloaded automatically upon deployment. Furthermore, we needed a way to preserve the ability of the Python application, to send HTTP requests at certain points in the code. In the past, the Python module Requests[5] was used to perform HTTP calls. We found that this library was still feasible in an Azure Function environment. Hence, we decided to use this specific library in our Function apps to perform HTTP requests.

4.1.2 Finding the suitable Function Trigger

Developers may choose from a variety of trigger options, to execute code in a function. The trigger we used for our Function apps was the HTTP trigger. This was because it allows handling HTTP requests comparable to the Web servers used in FactCheck, and would provide a solution to handle requests from the IdaFix extension. We also found that this specific trigger type suits the goal of modular APIs because the endpoints can be potentially reached by any component that is able of sending API requests.

From the list of previously described triggers, Blob triggers did not fit in our concept, as we did not plan to integrate any Blob storage accounts. We could imagine to use Timer triggers for possible background tasks in the system implementation. However, it did not blend into the subscription design and fact comparison. Lastly, a CosmosDB trigger could be of use, as we planned to use such a database. Similar to the Timer trigger, this would rather serve for invoking background tasks than realizing subscription APIs.

4.1.3 Deriving Code Segments suitable for Azure Functions

Since the original structure included multiple servers and databases, it is not possible to place all of them into a single Function app. Function apps would not be the correct target environment for databases, but would be the solution to host the functionality of the middleware and backend servers. Subsequently, functions replace the Tornado framework implementation for the FactServer and the Django framework used in the backend server.

Besides grouping coherent logic, the goal of subscription integration also dictated the partitioning across functions and also which functionalities were important refactoring targets.

We started by isolating the fact comparison and implementing a dedicated Function app. We deemed this as the most important feature to implement because on the one hand it provides the interface for the IdaFix extension. On the other hand, the comparison requires a connection to the external service Google Knowledge Graph[6] and queries fact

[5] https://docs.python-requests.org/en/latest/ (Last accessed: 2022.06)
[6] https://developers.google.com/knowledge-graph (Last accessed: 2022.06)

and precision metric data from the database, during comparison. Due to the central position of the component, we advanced this implementation first.

It followed the extraction of logic for accessing the fact database. Previously achieved with the Python CouchDB module[7], we needed to refactor the way the component interacts with the database because CosmosDB was used instead.

The connection to the precision metric database, originally hosted in an SQL schema, also required refactoring. An Azure Function emerged, that queries documents from the CosmosDB and serves precision metric and predicate data in a processed JSON file, like it was the responsibility of the Django backend.

4.1.4 Handling original Config Files

In the original project, configuration parameters had been placed into config files to separate them from code. This way, parameters can be changed without looking for them in scripts. However, pursuing this approach in Azure Functions, this would require the re-deployment of the app's code to Azure, whenever data is changed in the local config file. To also adapt this approach to a cloud-based solution, we used the application settings of the app to store parameters as key-value pairs. In Python code, those are accessible at runtime via **os.environ**, for example os.environ["ManagementURL"], using the OS Python module, used as interface to the operating system. After an app is deployed successfully, parameters can be changed in the Azure Web portal and no IDE or re-deployment is required.

For local development, we defined configuration settings in the local settings JSON file. After deployment to Azure, we manually defined the parameters in the "Configuration" section in the Azure portal and named them equally to the local settings. Otherwise, function execution would fail because deployment does not include the local settings file.

4.2 Using Azure API Management to implement the Subscription Service

Azure API Management[8] (APIM) is a solution to host a platform to manage APIs from different environments. It acts as a gateway between requests and backend APIs, as it exposes its self-managed endpoints and routes traffic to the backend nodes. To secure those, APIM takes care of the management for access keys. Keys must be included in every request header for authorization. Otherwise, the request is blocked and not passed through to the API.

[7] https://couchdb-python.readthedocs.io/en/latest/ (Last accessed: 2022.06)
[8] https://azure.microsoft.com/en-us/services/api-management/(Last accessed: 2022.06)

Furthermore, access restrictions can be set for certain keys or subscriptions if needed. Therefore, APIM introduces policies[9] for APIs. Policies can apply the inspection of requests, logging, authentication and more, as well as can be used to define quotas for subscriptions.

We considered APIM as the core solution for this thesis because it allows to manage APIs and host them on the Web. Additionally, the product provides an integrated solution for subscriptions.

4.2.1 APIM Pricing Tiers

Azure offers tiers that are priced on different characteristics. Besides the difference in monthly expenses, the selected tier also dictates the availability of certain features. The payment plan must be selected during the setup of the resource. Available options ranges from serverless, which is the cheapest, followed by developer-tier up to premium plans, which also are the most expensive ones. One tier we used was the “Consumption” pricing. For our purposes, this was by far the cheapest solution. Billing is based on consumed CPU time in a serverless environment, and the serverless solution would impose almost no costs during development. However, we found out, that important features to create our prototype, like the developer portal and the REST Management API, are missing in this version. Consequently, we switched to the next higher tier: The “Developer” tier. This service plan includes a developer portal, REST API and even account management, which the Consumption-Tier does not. In contrast to this, billing is set to a fixed monthly fee. Since this Tier provided us with the tools we needed, we decided to run our APIM on such a plan. The more sophisticated plans would support these and more features. However, they are too expensive to be justified for this work.

4.2.2 Creating APIs

Additional endpoints can be added to APIM in the "APIs" section of the Azure portal. APIM sopports multiple options to include new APIs. Thus, they can be created by manually defining an HTTP endpoint, or using definitions for REST service implementations, but most importantly, Azure resources can be directly imported, like Azure Function apps.

By using APIM as a middleware, the original APIs are hidden from end-users, who will only receive information from APIM instead, which also protects the backend from unauthorized access. As we wanted to implement our APIs as Azure Function apps, we selected the existing applications in the Azure portal. After the instructions guided us

[9] https://docs.microsoft.com/en-us/azure/api-management/api-management-policies (Last accessed: 2022.06)

through the process, APIM created a native HTTP interface for the endpoint. After the import, the APIs can be further modified or added to products.

An issue we faced was that all functions associated with a Function app would be imported and made accessible separately in APIM with their URL. However, we later noticed, that including an API in a product, APIM always includes all sub-functions of the app. By following this architectural approach, we would not be able to provide individual parts of an API as purchasable functionality. As a solution, we went down one step in terms of layers and created a Function app for each of the sub-functions of the apps we generated previously. This is why, for example, the logic for fact comparison and querying precision metrics is split into seprate Function apps, although they could technically reside inside the same app.

4.2.3 Developer Portal

The developer portal is the frontend user interface for developers, users and customers. It is realized as a website and can be accessed with a regular Web browser. In our system architecture, the portal serves a crucial role in providing users with information about the APIs. We leverage this web service to list the available APIs for users and show them their subscriptions and associated API keys. The developer portal does provide the functionality to handle a subscription process for APIs. We figured, that this could be the fitting location to incorporate payments. On the developer portal, the four main pillars of APIM we leverage to enable API usage are the following:

User Accounts

User accounts are the tool for APIM to handle the registration and login of developers. It uses accounts to assign a subscription relationship with products. While the login system can be extended by external authority providers, for example, Azure Active Directory, it still needs and creates a user account internally. Users of our prototypical system create an account in Active Directory B2C, which they use to log in to the APIM developer portal. This way, some custom information and data can be gathered during signup, which APIM does not provide. Both systems can coexist, which means users created in APIM can use the services alongside users which use an account created in an Active Directory.

APIs

APIs are available in the management instance after importing an endpoint. They are exposed at a new HTTP endpoint inside APIM which acts as a middleware between the request and the actual backend resource. Available APIs are listed in the developer portal

and if users are authorized, they can look up further information on where and how to call them.

Products

Products are created by APIM administrators in the Azure portal and define items customers can subscribe to. They can have multiple APIs assigned, to which developers gain access after they acquired a subscription. Multiple products can exist at the same time and can have APIs in common. Available Products are listed in the developer portal and can be browsed by users. There are additional options to customize request limits and quotas. We use this feature to maintain trial versions that have a restricted number of requests per hour.

Subscriptions

In APIM, a subscription defines a relation between a user and a product. Once established, a key is generated that enables users to use all APIs contained by this product. If they have a subscription active, users can see their API key for the respective product in the "Developer" section on the developer portal.

4.2.4 Delegating Subscriptions

API Management allows delegating the behaviour for authenticating users and acquiring subscriptions. It is possible to either delegate the subscribing and the authentication process separately, or both together. Therefore, a delegation endpoint has to be created, and its URL must be placed in the APIM "Delegation" section. Instead of processing the action itself, APIM forwards the browser to this URL and sends parameters for the action alongside.

We chose to only delegate product subscriptions. User authentication is still handled by APIM and extended by the B2C. As delegation endpoint, we created an Azure Function app, taking over the responsibility for subscription-related requests. At this point, checkout functionalities of payment processors can be integrated and subscriptions can be added to the user's account after a successful purchase. The disadvantage that emerges from this approach is, that the link between the subscription and the account has to be made manually, or in our case, programmatically, by using the REST Management endpoint of APIM.

4.2.5 API Management's REST API

API Management provides a management API based on a REST architecture[10]. Therefore, the HTTP methods GET, POST, PUT and DELETE can be used for reading, creating, updating and deleting internal resources. This is crucial to manipulate and manage entities of users, subscriptions and products outside of the Azure portal. As an architectural style, REST APIs provide access to resources based on HTTP, which makes this approach feasible for endpoints on the Web. By default, the REST Management API is disabled and has to be enabled manually in the Azure portal. Thereafter, any endpoint, browser, tool, or program can use this API by sending the correct HTTP request methods to the respective URL. To authorize requests, a Management SAS (Shared Access Signature) must be retrieved from the Azure portal and included in any message, that is sent to the REST API. The signature token is valid for one month, and incoming requests would be blocked after the token expired. In that case, a new one must be generated in the portal and the old signature should be replaced in all locations. We identified the REST API as a powerful tool to edit subscriptions managed by the APIM instance. Firstly, subscriptions can be created and updated, without a person using the Azure portal. Secondly, HTTP calls can be implemented in a program to automate the procedure and lastly, the implementation can be located outside the system, for example in an Azure Function.

4.3 API Usage for End-Users

Right from the beginning, we needed to determine, how APIs we develop, might be used by end-users. The type of user interface could potentially influence how other parts of the system advance.

4.3.1 IdaFix as the User Interface

In the previously developed prototype, the browser extension IdaFix was the tool that allowed users to interact with the system and had its own graphical user interface to display data. Despite the change of the system architecture, the basic interaction between a human user and the servers would remain the same in the process. That meant, that IdaFix would remain as the interaction tool, that sends requests to the component responsible for the fact comparison. This led to the conclusion, that IdaFix will be a part of this work. The code, which is responsible to send requests, had to be updated so that it can communicate with the new API system instead.

[10] https://docs.microsoft.com/en-us/rest/api/apimanagement/apimanagementrest/api-management-rest (Last accessed: 2022.06)

4.3.2 Updating the Communication Endpoints

To initiate the fact comparison, the initial implementation of IdaFix sent an HTTP POST request to the server, including a JSON object of the parsed structured data from the currently visited website. When transferring the backend functionality to Azure Functions, the functionality of the receiving end would remain similar due to the use of HTTP triggers. However, Azure API Management, as the middleware between the request and the functions, imposes the use of subscription keys to access APIs.

This implied, that related parts in the add-on had to change to adapt to the new conditions. The question was how users could enter and save the key in the extension. Therefore, we used the existing implementation for storing options in the browser storage and added a field in the code, intended to contain the key. The HTML user interface of the extension's option panel received an additional field, that allows users to insert a string copied from the developer portal. In a following step, we adapted the code, so that the API key is retrieved from the storage and included in the header of the POST request made with jQuery AJAX.

4.4 Account and User Management

With providing subscription services, the question arose, where the assignment of subscriptions should be stored, to verify if users of our system are eligible to use APIs. It was likely, to us, that this would require a solution to include some form of user account management. This carried forth the concern about the type of data, that should be stored in accounts. Are they the traditional combination of a user identifier and a password, or additional parameters? As a last challenge, we needed to consider where the management should be located in the system and which resource type should be used. With the cloud resources introduced in this section, we had two possible locations to manage user accounts.

4.4.1 User Management with APIM

Azure API Management provides its built-in user management and account handling system, which it uses internally to assign access to APIs and products. However, we found, that this service is limited in features and also data that can be gathered during sign-up. The information required to create an account are an e-mail address and a password.

4.4.2 User Management with Azure Active Directory B2C

We tested Active Directory B2C[11] (Business to Consumer) to manage user accounts. It is a self-contained management service that provides a centralized organization of users and their data, which live in a subspace called "Tenant", that represents an organization. Active Directory can serve as authority provider in external applications. This is also possible for APIM, where the user verification process can be forwarded to an Active Directory. We identified this as a powerful technique because, as we found, B2C allows gathering custom parameters upon account creation. This means, that besides the usual fields for an e-mail address, password and username, custom parameters can be gathered like a nickname.

Connecting Active Directory to APIM

To integrate Azure B2C in APIM, we had to follow Azure's guide[12] to connect both instances. This required creating a user flow for registering new accounts and signing in users. Then, the B2C instance had to be set as an identity provider in APIM. For external identity providers, APIM provides a redirect URL, to which the provider forwards after the authentication process. On the B2C end, APIM is registered as an app and the explicit redirect URL must be manually copied from the APIM site and saved in the respective field on the B2C side. The app is identified by an application ID, which needs to be copied to the site in APIM, to finalize the creation of the identity provider. As a final step, a secret must be generated in the B2C instance and inserted into the APIM resource. After successful installation, B2C is shown as an additional option on the login page of the APIM developer portal.

Creating User Flows

To manage user activities, Active Directory provides user flows, which can be enabled and implemented on demand. Those are pre-built user interfaces, that guide users through specific workflows, such as log in, create a new account, reset passwords and more. The graphical appearance of flows is customizable, which allows adapting the design to the visual necessities of one's brand.

[11] https://azure.microsoft.com/en-us/services/active-directory/external-identities/b2c/ (Last accessed: 2022.06)
[12] https://docs.microsoft.com/en-us/azure/api-management/api-management-howto-aad-b2c (Last accessed: 2022.06)

4.5 Migrating to Azure Cosmos Database

The implicit consequence for us of moving the system logic and access points into a cloud environment, was to also host database instances in this area. As a database solution within Azure, CosmosDB[13] is available. CosmosDB is designed as a serverless database solution and is capable of scaling elastically, to meet variable loads of traffic. The pricing mode can be selected from a provisioned plan, guaranteeing a throughput threshold and a consumption-based model, that charges customers based on the consumed resource units and therefore making it an attractive solution for development and testing projects.

CosmosDB follows a NoSQL (Not only SQL) database architecture. Furthermore, CosmosDB provides APIs, making it possible to retrieve stored data with queries like utilized for Cassandra and MongoDB, however allows using conventional SQL queries as well. A database resource can contain multiple databases, that further organize stored documents in collections.

CosmosDB appeared to be a promising database solution to integrate in our implementation. We chose it because of its automatic resource scaling as well as the consumption-based pricing, making it suitable for our development. Also, following a NoSQL structure makes CosmosDB similar to the previously used databases in FactCheck.

For local development, we installed CosmosDB Emulator[14]. While it can be used to test interactions with the database, in terms of API support its feature set deviates from the service provided in the cloud. Moreover, local hardware may affect performance.

4.5.1 Migrating CouchDB

Cosmos provides a feature, which allows inserting a new JSON document into a container from the Web interface of the data explorer. There, the text can be submitted as a new item, for example: {"document1":"data"}.

It is also possible to upload a text file that contains the data. It is even possible to submit a file, which contains multiple JSON objects, which will be added as a distinct document. However, it is important to structure the JSON data as a JSON array, for example: [{"document1":"data1"}, {"document2":"data2"}]

The problem here is that uploads must not exceed two megabytes per upload request. Hence, this approach is not feasible for multiple gigabytes of fact data. Furthermore, even though CouchDB provides the functionality to export databases as a JSON dump and formatted as a text file, the data would need manual modification, since the exported data has not the required JSON array structure.

To transfer documents, we implemented a Python script, which utilizes the respective Python modules for CosmosDB and CouchDB to connect to the database services as

[13] https://azure.microsoft.com/en-us/services/cosmos-db/ (Last accessed: 2022.06)
[14] https://docs.microsoft.com/en-us/azure/cosmos-db/local-emulator (Last accessed: 2022.06)

well as reading documents from CouchDB and writing documents to CosmosDB instance. The advantage here is, that this process can be extended and customized. Furthermore, documents can be edited before they are inserted.

4.5.2 Migrating from MySQL Database

In the original prototype, a SQL database stores precision metric data in tables. To store the same data in the CosmosDB, the table structure had to be converted into a JSON document format. Therefore, we created a database in a Cosmos instance, which essentially replaces the original MySQL component. Then, we created a collection for each table and converted the attributes of each entry of a table into a JSON format. This data was then inserted as an item in the respective collection. This approach also meant that any SQL-based query would not work any more and would imply the adaption to the new Cosmos API in the refactored prototype.

4.5.3 CosmosDB's Core APIs

The underlying structure of CosmosDB supports different database APIs[15] to access and manipulate stored data. Therefore, CosmosDB allows executing queries from existing languages like SQL and Mongo that define the core APIs. The desired core API must be selected when creating the resource.

In a first approach, we used a CosmosDB based on a SQL core. It seemed to us, that this type has the biggest support regarding features. This allowed us to directly perform database queries by using Azure Function bindings and establish database connections without additional programming effort. Using bindings made the code in Function apps cleaner and more structured. As a downside, the queries defined for the CouchDB in the original system had to be rewritten in a SQL query that returns the same document result.

In a second attempt, we decided to use a CosmosDB instance providing the Mongo API. MongoDB was not a database technology that was used in the original system, but we chose it for its document-based approach. Like CouchDB, MongoDB is a document-based database, which provides its query language to retrieve documents and filter them.

4.6 A Payment Solution based on PayPal

Since one goal of this thesis is to find ways of monetization of FactCheck, it would not be complete without a working payment component. We identified PayPal[16] as a promising solution. This payment provider orients its financial services towards a digital platform

[15] https://docs.microsoft.com/en-us/azure/cosmos-db/choose-api (Last accessed: 2022.06)
[16] https://www.paypal.com/ (Last accessed: 2022.06)

to serve customers as well as companies globally. PayPal provides solutions for online purchases, transactions for marketplaces and organization of donations. Payments can either be settled with the credit of a user account, or fees can be debited from a credit card. Important to our work, online payments can be realized by PayPal Checkout, which can be integrated into a website interface. Furthermore, there is a solution for recurring payments, offering the creation of subscription plans with variable billing periods and also incorporating test phases.

For this work, actually setting up a business including accounts and credit cards would be out of scope. This could also be remedied by using PayPal in this thesis. To test the environment and implement payment processes, developers can use a sandbox, that behaves similar to the main website.

4.6.1 PayPal Sandbox

PayPal maintains a sandbox[17] that reflects the production site and allows testing financial services as they would behave in the live environment, however, the sandbox website resides on a different domain. Accounts to simulate customers are created and managed on the developer portal[18]. Each account has fake data assigned and money can be set to any value to test required payment workflows.

4.6.2 Managing Payments

A predominant use case scenario in this work was that users must be able to purchase a subscription. Since we planned to use PayPal Checkout, which is designed for embedding into websites, it was logically to implement a Web application serving an HTML document.

In a first implementation attempt, we created an Azure Function that serves an HTML page, displaying the transaction information and a PayPal button for issuing a transaction request. This would log users into their PayPal account, where they confirm the purchase. Then, the browser would be redirected to another Azure Function after the payment function returned. This function added the subscription to the user account by sending a PUT request to the API Management REST API and forwarded the user back to the developer portal. However, this approach would be problematic if the user cancelled the redirect, which would exchange the money but would not add the subscription. Additionally, the function could be invoked manually with a request, to add a subscription without ever paying for it.

In a second approach, we integrated PayPal webhooks in our implementation. Webhooks allow to couple resources in a distributed system and to asynchronously notify predefined HTTP endpoints about events. Therefore, the receiving system, that should execute an

[17] https://www.sandbox.paypal.com/ (Last accessed: 2022.06)
[18] https://developer.paypal.com/ (Last accessed: 2022.06)

operation on such an event occurrence, exposes an HTTP endpoint and provides an URL to external resources. In general, the system maintaining the webhook, sends an HTTP call with the event specific data to this URL. In the case of PayPal, webhooks receive messages about transactions associated with a certain merchant application.

4.6.3 PayPal Webhook

In PayPal, a webhook can be set up for any registered client app. To consume messages, an HTTP endpoint has to be specified for the merchant application in the PayPal account. To intercept the events, we implemented an Azure Function and set it as endpoint for the webhook. Furthermore, the required events have to be selected from a list, which should issue a call to the webhook. For testing purposes, we selected the option to send notifications for every event[19] so that we could learn when messages are sent and also inspect the content of the requests to see the type. With that option selected, we noticed that for actions usually more than one request is sent, each covering another event type based on the stage of an internal process. For performing a payment with the checkout system, we received a message with the type "CHECKOUT.ORDER.APPROVED" and "PAYMENT.CAPTURE.COMPLETED". When using the checkout integration to start a subscription, the notifications "BILLING.SUBSCRIPTION.ACTIVATED", "BILLING.SUBSCRIPTION.CREATED" and "PAYMENT.SALE.COMPLETED were emitted. As multiple notifications are emitted for one action, we had to determine after which of these events a customer is eligible to receive and also use subscriptions in our environment.

Having the handling of notifications implemented, we needed to find a way to verify that they are sent by PayPal. Calling PayPal's REST API for authenticating PayPal messages was a promising solution. If the request was verified successfully, the API Management REST API is called, to add the subscription to the user account, for the respective product. This action is triggered asynchronously to the actual purchase, whenever PayPal sends an event notification.

Another problem to solve was, to retain the information about the buying user and the purchased product. API Management includes the product- and user ID when redirecting to the purchase website where the PayPal button resides. But the subscription is added by calling APIM's REST API from the webhook function, which is in a different location than the delegation function, where users purchase the product. To overcome this, pending purchases could potentially be stored inside a database, including the necessary product information as well as an ID, identifying a specific purchase. Later, when the message denoting a completed transaction is received from PayPal, the webhook function could retrieve the information from the database, based on the purchase ID, and add the subscription in APIM. However, this would add too much complexity in both the code

[19] https://developer.paypal.com/api/rest/webhooks/event-names/ (Last accessed: 2022.06)

and the system, just to store information about two values. Thus, the **custom_id** of the JavaScript PayPal button is used to directly tie the product ID and user ID value to a transaction. This can be seen in both of the notification examples 4.1 and 4.2, where the string concatenation can be found under **resource.custom_id**.


```
{
    [...],
    "resource_type": "capture",
    "resource_version": "2.0",
    "event_type": "PAYMENT.CAPTURE.COMPLETED",
    "summary": "Payment completed for EUR 4.99 EUR",
    "resource": {
        "amount": {
            "value": "4.99",
            "currency_code": "EUR"
        },
        "seller_protection": {
            "dispute_categories": [
                "ITEM_NOT_RECEIVED",
                "UNAUTHORIZED_TRANSACTION"
            ],
            "status": "ELIGIBLE"
        },
        [...],
        "final_capture": true,
        "seller_receivable_breakdown": {
            "paypal_fee": {
                "value": "0.52",
                "currency_code": "EUR"
            },
            "gross_amount": {
                "value": "4.99",
                "currency_code": "EUR"
            },
            "net_amount": {
                "value": "4.47",
                "currency_code": "EUR"
            }
        },
        "custom_id": "<user ID of customer>&idafixbasic",
        "links": [
            {
                "method": "GET",
                "rel": "self",
                "href": "https://api.sandbox.paypal.com/v2/payments/
    captures/XXXXXXXXXXXXXXXXX"
            },
            {
```

```
                "method": "POST",
                "rel": "refund",
                "href": "https://api.sandbox.paypal.com/v2/payments/
    captures/XXXXXXXXXXXXXXXXX/refund"
            },
            {
                "method": "GET",
                "rel": "up",
                "href": "https://api.sandbox.paypal.com/v2/checkout/
    orders/XXXXXXXXXXXXXXXXX"
            }
        ],
        "id": "XXXXXXXXXXXXXXXXX",
        "status": "COMPLETED"
    },
    "links": [
        {
            "href": "https://api.sandbox.paypal.com/v1/notifications/
    webhooks-events/WH-XXXXXXXXXXXXXXXX-XXXXXXXXXXXXXXXXX",
            "rel": "self",
            "method": "GET"
        },
        {
            "href": "https://api.sandbox.paypal.com/v1/notifications/
    webhooks-events/WH-XXXXXXXXXXXXXXXX-XXXXXXXXXXXXXXXXX/resend",
            "rel": "resend",
            "method": "POST"
        }
    ]
}
```

Listing 4.1: Payment Completed Notification from PayPal for a Purchase of the Product IdaFixBasic

```
{
        [...],
        "resource_type": "subscription",
        "resource_version": "2.0",
        "event_type": "BILLING.SUBSCRIPTION.CREATED",
        "summary": "Subscription created",
        "resource": {
            [...],
            "custom_id": "<user ID of customer>&idafixpremium",
            "links": [
                {
                    "href": "https://www.sandbox.paypal.com/webapps/
    billing/subscriptions?ba_token=BA-XXXXXXXXXXXXXXXXX",
                    "rel": "approve",
                    "method": "GET"
```

```
15                },
16                {
17                    "href": "https://api.sandbox.paypal.com/v1/billing/
    subscriptions/X-XXXXXXXXXXX",
18                    "rel": "edit",
19                    "method": "PATCH"
20                },
21                {
22                    "href": "https://api.sandbox.paypal.com/v1/billing/
    subscriptions/X-XXXXXXXXXXX",
23                    "rel": "self",
24                    "method": "GET"
25                }
26            ],
27            "id": "X-XXXXXXXXXXX",
28            "plan_overridden": false,
29            "plan_id": "P-XXXXXXXXXXXXXXXXXXXXXXX",
30            "status": "APPROVAL_PENDING"
31        },
32        [...]
33    }
```

Listing 4.2: Subscription Created Notification from PayPal for the Product IdaFixPremium

The field custom_id is intended to store the invoice ID that represents the purchase in the local system. The value is added when the Checkout button initiates the payment process, and then appears in further notifications for this transaction. We do not use invoice IDs in our system, for which reason, we could leverage this functionality to pass product- and user ID from the delegation resource to the webhook.

4.6.4 Resulting Architecture

After examining the possibilities and services available, we built the final architecture for the API subscription system for FactCheck based on the knowledge gathered during the design phase. This led to a distributed system utilizing PayPal as well as Azure services introduced in this chapter. While this chapter examined various problems regarding design decisions, the following chapter explains the state of the final prototype and describes important workflows of subscription purchases and interactions between system components.

5 Implementation

This section describes the final state of the implementation, which emerged from the background and design process, described in the previous chapters. The created prototype uses Azure API Management as the central component, to manage user access to APIs and acts as the gateway to underlying backend services. Those services refer to the refactored functionality from FactCheck, realized as Azure Functions.

5.1 Azure API Management

Azure API Management is an Azure Product designed to manage access to APIs. It builds the core of our cloud-based FactCheck system as it manages access to FactCheck APIs, is involved in user interactions, and we also use it to maintain subscriptions.

5.1.1 Usage in the Prototype

We instantiated one APIM resource based on a developer pricing plan, to keep costs manageable. Its developer portal establishes the central interaction point for users, where they can register for an account, sign-in, browse available products and included APIs as well as redeeming subscriptions for products and managing assigned subscription keys. Having an account in the management resource is mandatory to view all details on the developer portal and is required to subscribe to products. Accounts can be created on the sign-up page of the portal. Additionally, we integrated an Azure B2C tenant as identity provider to provide an alternative way to manage user accounts. Although B2C saves user accounts and handles authorization, APIM internally maintains an account to map subscriptions to it.

We do not delegate user authentication and keep the built-in account management functionality. However, we enabled delegation for product subscription so that we can include a step for payments into the process of redeeming subscriptions. With this settings active, APIM forwards to an external endpoint, including parameters in the URL. The most important one regarding subscriptions are the user ID, which identifies the account purchasing and the product ID, which defines the desired product the user wants to buy a subscription for. Alongside the IDs, a parameter defines if it is about a subscribe or unsubscribe operation. A potential delegation endpoint must adapt to this request schema and take action according to the passed parameters. Therefore, we implemented a

Function app operating with an HTTP trigger and added the Function URL as delegation endpoint URL in APIM.

5.1.2 APIM Delegation Function

We implemented an Azure Function based on an HTTP trigger, to which users will be forwarded when they click on the subscribe button in the developer portal. In code, product ID, user ID and the operation type are retrieved from the request parameter list. Based on the action, the function renders a different HTML response:

- Subscribe: Renders an HTML page with price and information based on the provided product ID and shows a PayPal button to purchase the product.
- Unsubscribe: Renders an HTML page that explains the cancellation process and provides links to PayPal, where the subscription can be cancelled.
- Success: Renders an HTML page that informs about a successful purchase and provides a link to return to the APIM developer portal.

The actions "Subscribe" and "Unsubscribe" are defined by API Management. We introduced the "Success" operation type because the Function already reacted to the APIM parameters, and this way we could reuse the flexible loading of HTML content. Basically, the function extends the subscription process of APIM by adding frontend functionality in the form of a webpage and opens up a connection to PayPal. A screenshot of the rendered webpage can be seen in figure 5.1, which features the state when purchasing a subscription.

The connection to PayPal is initiated by a PayPal button component, which is embedded as JavaScript on the website. To make this approach feasible for a variety of buttons, we created a separate static HTML file for each button. We named the files after the product name in APIM. Hence, we ended up with three files for IdaFixFree, IdaFixBasic and IdaFixPremium respectively. The reason for creating separate files was that each button type functions differently. The button for the basic subscription tier implements a payment button in PayPal, which requires the price to be set in the setup function. The button for IdaFixPremium, however, implements a PayPal subscription button, meaning it requires a plan ID in the setup function. The code in the file for IdaFixFree does not contain a PayPal button, but rather a conventional HTML button, that initiates a request to the webhook to bypass PayPal. It is the result of the special case, that a subscription should be initialized, but must not involve payment. Based on the product ID in the request to the delegation function, the button can be loaded dynamically. The Python module AdvancedHTMLParser[1] is used to load the snippets and insert its content into an intended element of the main HTML document.

[1] https://pypi.org/project/AdvancedHTMLParser/ (Last accessed: 2022.06)

The parser also loads the files that contain the HTML content for specific actions. The files of the application include a page, which is used during the subscription process and serves the injected PayPal button. Another page, that is shown when a subscription is about to be cancelled, and shows information to users, how subscriptions can be stopped on PayPal. Furthermore, the app includes a page that is shown after a successful purchase and one fallback page if an unexpected error occurs. It should be noted, that the delegation Function app itself is not responsible for modifying any data in APIM.

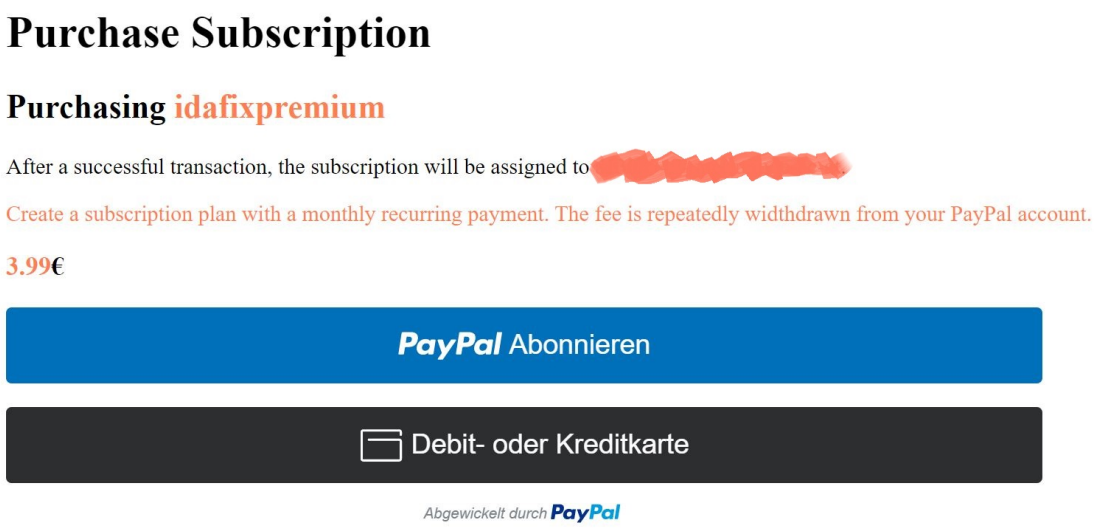

Figure 5.1: Rendered Web Page for purchasing a Subscription for IdaFixPremium provided by the APIMDelegation Function

5.1.3 Integrate Payment Processing for Subscriptions

An important goal of this work is to find ways to monetize FactCheck. We decided to use PayPal as a payment processor and needed to change APIM's behaviour when users subscribe to a product. By default, APIM creates a subscription and provides users with an API key as soon as they subscribe to a product. This would not work for our desired solution because we only want to allow users to use APIs after they successfully purchased a subscription.

Enabling delegation for subscriptions was our solution to this issue. This prompts APIM to forward to the delegation endpoint instead of adding subscriptions to an account instantly, when users click the corresponding button on the developer portal. However, this breaks the automatic assignment of subscriptions and requires implementing a custom solution. We identified APIM's REST Management API as a solution to create subscriptions on demand, however, the question was from where the API should be called.

Since we chose PayPal as payment processor, we implemented a webhook that receives and processes messages about transactions from the payment service. We identified the Azure Function app of the webhook as the ideal location because on the one hand the implementation is close to the information of successful payments and hence the request for subscriptions can be immediately validated. Furthermore, we only want to give away subscriptions after successful payments. On the other hand, the application provides the basis to be passively triggered and send HTTP requests to the REST API. Therefore, we implemented the functionality to create and cancel subscriptions by utilizing the management API in the same Function app that is also used for the realization of the webhook.

5.2 Azure Active Directory B2C

Azure provides alternative versions of Active Directory that are intended to fulfil a different purpose. We chose the B2C service because it incorporates a solution between producers and end-users, hence fitting into our use-case scenario. We use Active Directory B2C to extend user account management in APIM. We connected the B2C tenant to the APIM instance, allowing user authentication with Active Directory. B2C summarizes sign-in and sign-up activities into the same user flow. Therefore, we created one user flow entry in the tenant, that handles both of these activities, and named it "B2C_1_Register_Login" to specify the flow in the sign-in and sign-up policy for the identity provider in APIM. APIM calls this user flow when it forwards a user to B2C.

In the final state of the prototype, the B2C tenant can be seen as an extension to the authorization options to access the developer portal in APIM. However, the portal can be accessed by accounts native to API Management, making the B2C tenant redundant in that sense.

5.3 Refactored Parts of FactCheck as Azure Functions

When deploying an existing system architecture, it is likely that adjustments have to be made to the system, or the code has to be refactored. This is due to the new environment which, depending on the cloud provider, may not support all the required resources.

As a target area for deployment, we identified Azure Functions as a promising candidate. They provide flexible scalability and support the programming language Python. We selected the consumption-based pricing plan for the Functions in our system, because of the adaptive pricing model. Furthermore, Functions can be invoked via HTTP requests, which would replace the need to implement server frameworks. Additionally, Functions can be imported and exposed as API endpoint in API Management, the core service we wanted to use. For this reason, we saw a potential benefit in using a combination of these

technologies.

In this thesis, we refactored the existing code to fit into the environment of Azure Functions. In the case of FactCheck, the Web server functionalities of the implemented server frameworks Tornado and Django were removed in the process and replaced by Azure Function technology. The essential parts of the logic had to be isolated and adapted to the new environment. The context of the code determined the separation of functionality and location for deployment across multiple Function instances. This led to the development of the following Azure Function applications:

5.3.1 FactCheckCheck

This Function app encapsulates the functionality to handle requests received from the IdaFix extension and to compare included structured data with stored facts from the local database. We refactored the code, so that the comparison process can be initiated by an HTTP function trigger instead of the Tornado server framework. This is now handled by the entry script of the function, by default _init_.py, which is called to resolve the request, instantiates objects from the required classes for comparison and returns an HTTP response with the results. Furthermore, calls to resources outside the Function App must be performed. On the one hand, a connection to Google Knowledge Graph must be established, to resolve instances of entities. On the other, requests must be established to the MongoDB core API of the CosmosDB instance, which replaces the CouchDB of the original system. To send data to the Google service, we use the Python library Requests.

To query documents from the database, we had to address connections to multiple containers for each request. We account for loading facts from the fact container, required for the comparison, and a request for inserting a potentially new fact. Furthermore, we consider maintaining fact statistics, after comparison. Lastly, FactCheckCheck also loads data from the cache container, that stores resolved data from Knowledge Graph, or populates it with new items after the resolving step. To interact with the database, we created a subcomponent CosmosConnector, which uses the Python library PyMongo and contains the code related to database activities. New instances of CosmoConnector are created at runtime when requests need to be made. A flag defines if this object connects to the fact, stats, or cache container. CosmosConnector is required in instances of FactManager, DatabaseStats and Resolver. The class diagram in figure 5.2 depicts the classes that are responsible for the fact comparison process and are deployed in this Azure Function. The diagram also shows the internal dependencies of the classes.

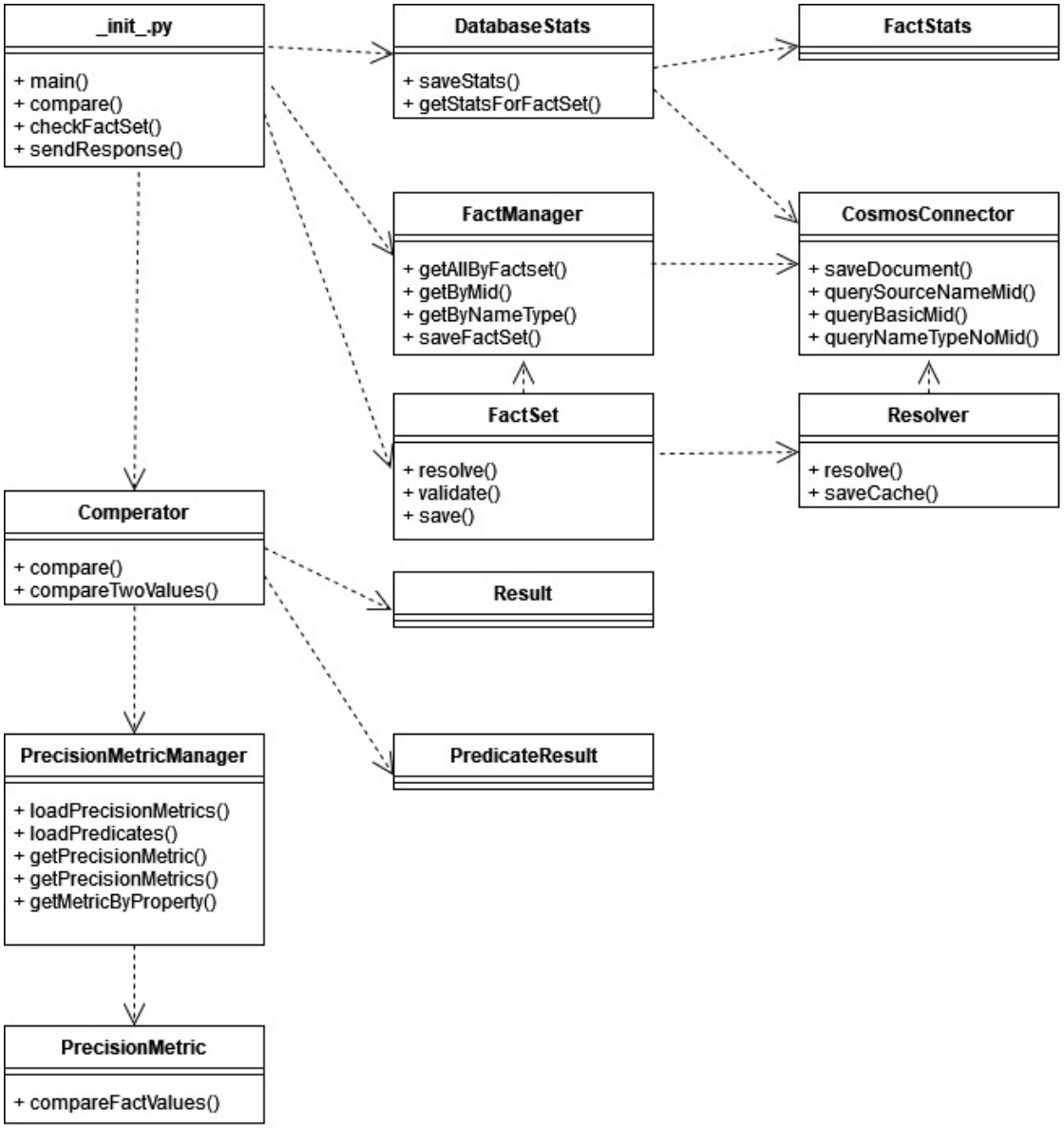

Figure 5.2: Conceptual Class Diagram of the Azure Function FactCheckCheck

5.3.2 BackendPrecisionmetricData

Based on an HTTP trigger as well, this Function app provides the functionality to query precision metrics and predicate data from a CosmosDB. For the database interaction, we integrated the PyMongo library, which establishes a connection to the database instance and executes the queries. BackendPrecisionmetricData is intended as a backend API for

other parts of the system that require these documents, and thus represents the interface that originally was handled by the Django backend server of FactCheck.

Since the requirement for precision metrics and predicates is relevant in the comparison procedure performed in the respective Azure Function, it would be a logical decision to implement the database requests in this location. However, we decided to implement the database access in a separate function for several reasons. First, it helps to maintain a clear organization of the code for the FactCheck system because the FactCheckCheck application is already responsible for fact comparison. Furthermore, by logically decoupling retrieving documents about predicates and metrics, a dedicate Function app, BackendPrecisionMetricData, could be implemented as a separate component. This also corresponds to our approach, where we try to isolate thematically related functionality into Azure Functions, to create modular components that can be exposed as self-contained APIs.

5.3.3 FactCheckFactView

The goal of FactCheckFactView was to extend the FactCheck endpoints by new API functionality. Therefore, we created another Function app that is accessible by HTTP and allows querying for all stored facts which are related to a given username. This should give users a way to see a list of all the facts they have contributed to FactCheck, while they browsed the Web with the IdaFix extension. The intention of the API is the integration in a user interface context, so it can serve as an endpoint in any user related systems of FactCheck. In the prototype of this work, IdaFix was adjusted to consume this API and render the list of documents in the extension window.

At the path **"/FactCheckFactView/userView/{username}"**, the function accepts GET requests where the segment "{username}" is a placeholder that is substituted by the desired username. The URL pattern is a result of the usage of the route settings in the function configuration. This way, the username can be retrieved as a route parameter at runtime and is used as input for the query. The PyMongo module establishes the connection to CosmosDB and queries fact documents sharing the same user ID. The Function app then produces a list of the retrieved fact documents. The JSON list is returned in the payload of the response.

5.3.4 FactCheckIO

FactCheckIO is an artefact, that emerged from the refactoring step of the backend servers, originally implemented with the FactCheck framework. They provided interfaces to query documents from the database by a REST style API. This Azure Function app encapsulates functionality to query data from the database by providing access points by the means of an HTTP trigger. We organized each database access into a separate function, so that

data can be retrieved by calling a descriptive URL path, for example **"/pm/id/11"** to retrieve the document for the precision metric with the ID 11. We achieved this by using the route binding in the settings for each individual function in the App. This allows to append additional segments to the URL path including route parameters, so we ended up with settings like **"route": "pm/id/{docId}"**, that we specified in the bindings option. URL parameters can be retrieved as route parameter from the request in the code. Thus, we use them as input for the Mongo query. The finished app provides a set of functions to retrieve lists of documents for fact data, based on mid and source, precision metrics, based on mid and name, property data, based on id and predicate-family and cache, based on id, mid and source.

5.4 Realization of Subscriptions

One of the driving goals of this thesis was to make APIs accessible through subscriptions. We achieved this by implementing Azure Functions and importing them into API Management. We use products to bundle functionalities together and provide them in a way, that different pricing tiers can be applied. The products we created are designed to monetize IdaFix' capability to identify conflicting data on websites. Therefore, we propose three tiers of products to which IdaFix users can subscribe to. Each of them follows a different pricing model and provides unlimited or restricted access to functionalities.

5.4.1 Subscription Tiers

Starting with the cheapest product, we called it IdaFixFree, we want to provide a trial version of the functionality to compare facts. As the name suggests, this product does not impose fees nor does it require a PayPal payment. Only an activation is necessary, which unlocks the subscription for users. However, IdaFixFree limits the usage of the API for fact comparison through a usage quota, which is credited when creating an affiliated subscription. We implement this by adding a quota element to the inbound policy of the product in APIM, which regulates allowed requests for each individual subscription: <**quota calls="50" renewal-period="0"** />. With this setting, the subscription has a credit of 50 calls, which is decremented at each API request, until it is depleted. With the renewal period set to zero, the contingent never resets. This should give users the opportunity to test the functionality, but should also make sure that the trial version is not abused and users are incentivized to keep paying for the comparison service.

Moving to the next tier, we designed the product IdaFixBasic. Similar to the trial version, it is used to monetize the fact comparison API and is restricted by a quota. Although in that case, we utilize the quota differently: <**quota calls="100" renewal-period="3600"** />. This product has its call limit increased to 100 and additionally, with a renewal interval of 3600 seconds, the quota resets every hour. Opposing to the first

product introduced, redeeming a subscription key for IdaFixBasic requires a payment. Specifically, we implemented PayPal's payment solution.

Finally, we propose IdaFixPremium, defining a subscription type which provides more functionality than the other products. We see it appropriate to combine this approach with PayPal's recurring payment solution. On the one hand, this should extend the monetization capabilities by integrate another payment option. On the other, we provide a subscription for expert users, or dedicated FactCheck enthusiasts, who require a greater set of functionality and are also willing to spend more to access these services. Contrary to the previously introduced products, IdaFixPremium does not limit calls to the APIs it offers. Additionally, users not only get access to the check functionality of the FactCheckCheck Azure Function, but also to the FactCheckFactView Function, which allows requesting a list of facts for a provided username and FactCheckIO, which allows requesting documents from the database by defined queries.

An overview of the implemented subscriptions can be seen in 5.3, which shows the relational imports of Azure Functions and dependencies of the products, which eventually define the subscriptions in our system. The packages also specify the names of function implemented in Function apps and the APIs in Azure API Management.

5.4.2 Manage Products

Another goal of this thesis was to come up with a solution that allows subscription management, including the adjustment of the functionality a subscription provides. It should also be possible to update prices of the subscriptions. For the subscription management, the proposed solution uses APIM to provide products and create bundles of API functionality. It is the task of Azure developers, which in this thesis was us, to maintain the API Management instance. In the Azure developer portal for APIM, developers can import external resources as APIs or remove them. They can create products to provide subscription variations.

Owed to the integrated PayPal options, payment and subscription, the price management is decoupled from APIM and needs to be maintained by developers in two locations. Payments are initiated by a PayPal button, which is served from the delegation function FactCheckAPIMDelegation. When instantiated by JavaScript code, the button receives the price from a price list maintained as a Python dictionary object. For each subscription in APIM, an entry is deposited consisting of the price value and a discreption text. In order to change the price, the value must be updated in the list and the payment button will receive the new value. This also updates the values displayed on the website generated by the delegation function. For subscriptions, PayPal internally maintains a plan from where the price information is retrieved for billing purposes. In addition to changing the value in the Function app, an additional step needs to be done in the PayPal portal, to adapt the pricing.

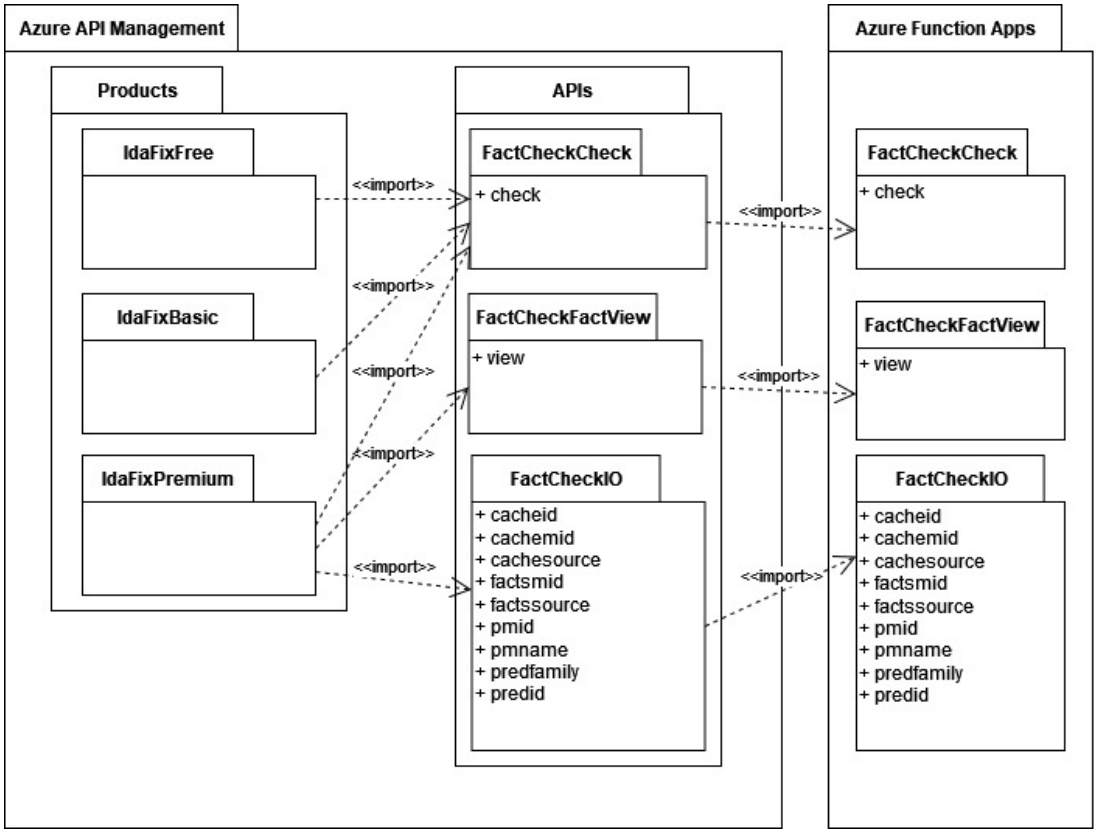

Figure 5.3: Package Diagram, giving an Overview of the Distribution of Functionality regarding Subscriptions

5.5 Processing Payments with PayPal

This project aims at implementing a feasible solution to monetize FactCheck functionality and decided to implement this idea by using PayPal, a payment platform for online purchases.

In the prototype, there are two places where an interaction occurs between our cloud system and PayPal. We use PayPal buttons on the delegation webpage served by the delegation Function. There, sandbox accounts are used to simulate payments for future customers. The second interaction interface manifests in the form of a PayPal webhook, that calls an Azure Function we specifically implemented for this purpose, and thus notifies our cloud system about events occurring on the payment platform.

Besides its popularity, we also reason our decision with its free sandbox environment, that allows to test the integration of this service in the desired system. To use it, it is required to create a personal account on the main website, *paypal.com*. It is required to share an e-mail address and a telephone number, to receive SMS for authentication purposes. This account is then eligible to access the domain *developer.paypal.com*, to access developer features. There, sandbox accounts can be created with a preset amount of money and fake credit cards to simulate payment processes.

The figure 5.4 depicts the resources we created on the live PayPal platform and the sandbox website, shows the localization of components and their relations.

5.5.1 Sandbox Accounts

For a minimal setup, two different account types are needed. These can be created with the main PayPal account at *developer.paypal.com/developer/accounts*. We created one business account, which acts as the merchant and one personal account, which impersonates a customer. Those accounts are bound to the main PayPal account and can be used like regular PayPal accounts, but are restricted to the domain *sandbox.paypal.com*. The default values for the account balance and credit cards were sufficient for our purposes, we only changed the currency code to Euros. To perform a payment, we used the credentials of the personal account to sign in and authorize the transaction. The business account specifies the shop owner in the merchant application.

5.5.2 PayPal Merchant Application

PayPal uses applications to group payments together and associates them with a certain shop. An application can be seen as the accounting section for a shop on PayPal. They reside on the main account as an entry and are identified by a unique ID. New apps can be created for the main account at *developer.paypal.com/developer/applications*, where we created our app named "FactCheck". We chose the "Merchant" type for the app since it is dedicated to sellers, while the opposing option "Platform" would be better suited for

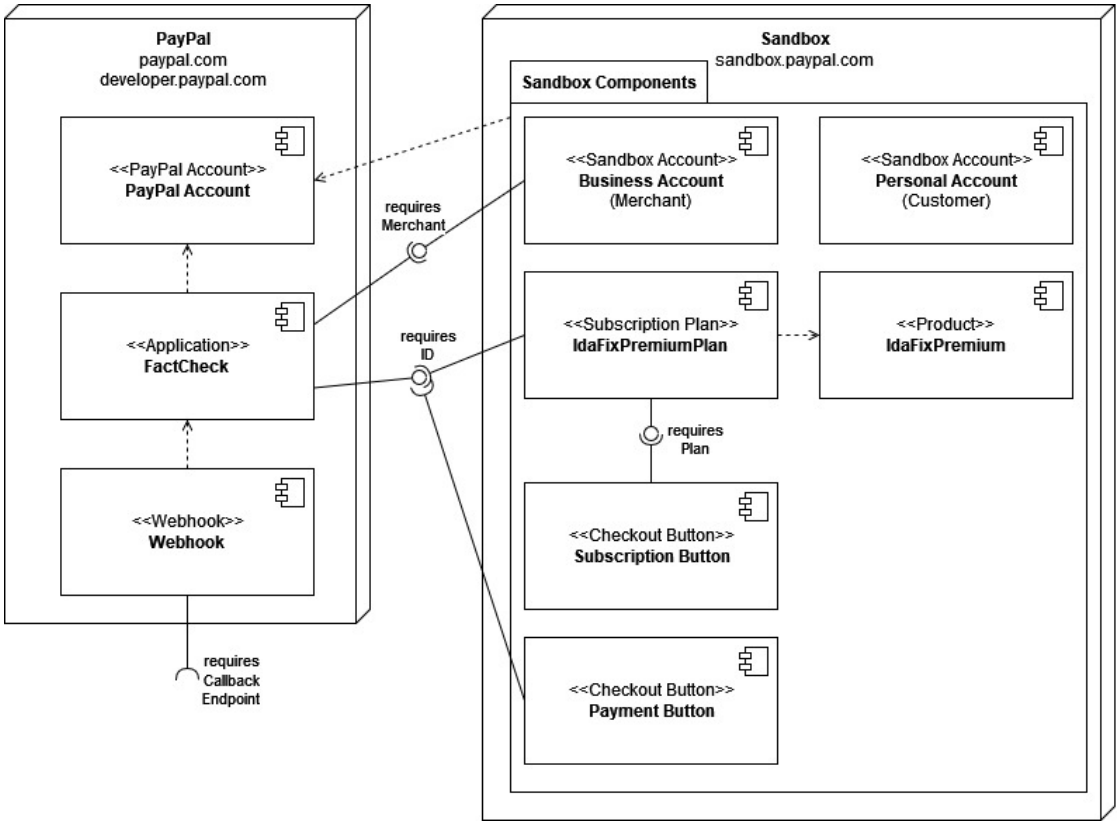

Figure 5.4: Structure and Distribution of the Generated Components used in PayPal

larger e-commerce companies. At this stage, it was crucial to select the e-mail address of the business account created earlier as the respective business account. We use the ID of the resulting app to test the integration of the payment functionality by passing it in every request to associate them purchases with our FactCheck business.

5.5.3 PayPal Webhook Function

We implemented the Azure Function app FactCheckWebhook that listens on an HTTP trigger and serves two purposes in the final system. For one, it is designed to receive and handle requests from PayPal because it serves as the callback for the PayPal webhook. On the other hand, it takes care of creating subscriptions in the API Management instance by calling its REST API.

As a means to notify the cloud system of subscription purchases and cause the distribution of access keys, we enabled a webhook for the FactCheck merchant app in PayPal. This required a URL for an HTTP endpoint, that must be specified in the webhook

configuration. A Function app was a practical choice because we could directly deposit the function URL as endpoint for the webhook. The Function is able to consume requests from the PayPal service, which are sent whenever an event occurs that is related to the merchant app. Since webhooks receive HTTP requests for a variety of notifications, we needed to only handle the important cases and drop the other ones. From our tests, we identified three event types we needed to account for.

- **PAYMENT.CAPTURE.COMPLETED** This event type identifies the notification for a successful payment and transaction by the payment processor.
- **PAYMENT.CAPTURE.COMPLETED** The Function expects this event type, to determine the successful purchase of a subscription. The notification about the creation is important because at this point customers have established an active billing plan in PayPal, imposing monthly costs, and therefore are eligible to use subscriptions in FactCheck.
- **BILLING.SUBSCRIPTION.CANCELLED** This marks the termination of a subscription and informs that a customer has stopped monthly payments for the subscription in their PayPal account. Hence, they are no longer allowed to use the FactCheck subscription, and access to subscription keys shall be cancelled.

Besides the webhook functionality, this Function application communicates with the REST Management API of API Management, to create and cancel subscriptions. This happens asynchronously to the preceding purchase, as soon as the event notification arrives and one of the above-mentioned types is specified in the message.

To call the REST service of an APIM instance, requests are sent to the exposed management URL. This URL is a concatenation of the domain for the API Management resource and the path *"/subscriptions/{}/resourceGroups/{}/providers/Microsoft.ApiManagement/service/{}"*, where the brackets are a placeholder for the Azure subscription ID, the resource group the resource resides in and the name of the APIM service.

Additional segments that are appended to this path, subdivide the access to specific objects in the APIM service. For subscriptions, this requires to further extend the management URL by appending the segment *"/subscriptions/"* followed by the desired name for the subscription to be created. We create new subscriptions by sending an HTTP PUT request, to the resulting URL, by utilizing the Python module Requests.

The solution we implemented is shown in the code excerpt 5.1, which shows at the example for creating a new subscription, how we retrieve parameters for the REST URL from the environment settings and how we dynamically build the path at runtime. It also exemplifies at which location product ID and user ID are transmitted in the request body. The parameter "ownerId" contains the user ID, "scope" defines the product for

which the subscription should be created and "displayName" describes the name for the subscription entity that is displayed in the developer portal. This also shows, that the Shared Access Signature has to be included in the header to authorize the request.

```
def addSubscriptionAfterPurchase(user, product):
    restPath = os.environ['RestURL'].format(
        os.environ['AzureSubscriptionId'],
        os.environ['ResourceGroup'],
        os.environ['APIMServiceName']
    )
    idOfCreatedSubscrition = product + "_" + user
    url = os.environ['ManagementURL'] + restPath + "/subscriptions/" +
    idOfCreatedSubscrition + "?api-version=" + os.environ['APIVersion']

    headers = {
        "Accept": "application/json",
        "Content-Type": "application/json",
        "Authorization": os.environ['ManagementSAS']
    }

    body = {
        "properties": {
            "ownerId": restPath + "/users/" + user,
            "scope": restPath + "/products/" + product,
            "displayName": "paypalsub_" + user + "_" + product
        }
    }

    response = requests.put(url, headers=headers, json=body)
    return response
```

Listing 5.1: Excerpt of the Webhook for adding a new Subscription

Removing a subscription by using the REST Management API works similar to creating one. However, instead of using the PUT method, we utilize the Request library to use HTTP's DELETE method. Furthermore, the path of the called URL links to the ID of the subscription. The ID of the subscription is a concatenation of product ID and user ID, resulting in a pattern like *"/subscriptions/idafixpremium_ < User ID of customer >"*. The SAS token must be included in this request as well.

Before actually sending a request to the management API, the function checks if the received request was sent by PayPal. The verification process is described in section 5.5.5.

To handle the special case of handing out subscription access for a free product, we had to implement a workaround to bypass PayPal payments and immediately assigning a subscription to a user. Our approach resulted in a dedicated function in the app, and thus, the webhook Function app contains the function **paypalWebhook**, handling PayPal requests as described in this section and the **activationWebhook** function, which is

intended to assign selected subscriptions without requiring a purchase. By sharing the same application, the configuration for the REST Management API can be reused. The expected POST request must contain the IDs for the product and the user. To make sure, only selected products can be redeemed, the product ID must match one of the IDs defined in the function, in our scenario IdaFixFree. The respective call is initiated when the user redeems a free product on the delegation function APIMDelefationFunction.

At this point, we want to clarify, that this application is not affiliated with the Function apps used for fact-checking and is not exposed as API through API Management.

5.5.4 Retaining User Information

Azure API Management needs the user ID and product ID to generate a subscription for a user. To provide users with an API key only after they successfully made the purchase on PayPal, we needed to find a way to retain this information from when a user clicks on the "Subscribe" button on the delegation page until the event arrives at the webhook after a successful transaction. We leverage the custom parameter "custom_id" in PayPal transactions, which can be specifically defined in PayPal buttons. This parameter can be used to append invoice information for requests, and this data will be persisted on the PayPal servers. It allows storing a string when the PayPal button is created in the user's client. This string is sent with the transaction request and is retained for this transaction on PayPal. As we require remembering the user ID and product ID, we decided to concatenate both into a string and separate it by a special character. This leads to a form of: "{custom_id: "user_id&product_id"}". If a value was set for the custom ID field, the string will be available in any subsequent webhook event, that PayPal might emit and the information about the user and purchased product can be retrieved from the request body of the event.

5.5.5 Verifying Messages from PayPal

Although it is not necessary for processing webhook events in the app, it could be possible, that one could find the Function app endpoint and send a faked request with the same structure as a webhook request. If the correct user ID and product ID were to be included, a subscription could be unlocked in API Management, without ever performing an action on PayPal. To tackle this, we implemented PayPal's verification API, which allows verifying if a certain message was actually issued by PayPal itself.

Messages from PayPal carry forth additional metadata in the header. That are parameters like transmission_id, transmission_time, Webhook_id, which are essential for the payment platform for being able to verify the authenticity of webhook notifications. Our webhook Function app is responsible for copying the parameters from the message header and include them in a new JSON object for the verification request.

To use PayPal API services, a Bearer token must be requested beforehand. This requires a post request to the service *https://apim.sandbox.paypal.com/v1/oauth2/token* which includes the webhook ID and secret to authenticate the application endpoint. We implemented a dedicated function in the Python script, as it can be seen in 5.2, that requests a valid access token from the PayPal service.

```
def getPaypalToken():
    url = os.environ['PaypalOauth']
    #id and secret from the webhook app
    authData = (
      os.environ['PaypalWebhookClientId'],
        os.environ['PaypalWebhookSecret']
    )
    postData = {
        "Accept": "application/json",
        "Accept-Language": "en_US",
        "grant_type": "client_credentials"
    }

    response = requests.post(url, auth=authData, data = postData)
    return response.json()['access_token']
```

Listing 5.2: Excerpt of how the Bearer Token is requested from PayPal

With the webhook Function capable of requesting a token and extracting required parameters from a message header, the actual request for verification can be sent. The target URL of the POST request is *https://api-m.sandbox.paypal.com/v1/notifications/verify-webhook-signature*, which provides the service to verify webhook notifications. In 5.3 we show our implementation of this process. It is shown, which parameters the function retrieves from the header data, that PayPal needs from the original message. Furthermore, the inclusion of the bearer token in the header of the validation request can be seen. Finally, the Requests module initializes the POST request.

```
def validateWebhook(req):
    header = req.headers

    verificationEndpoint = os.environ['PaypalWebhookVerification']
    authAlgo = header['paypal-auth-algo']
    certURL = header['paypal-cert-url']
    transmissionId = header['paypal-transmission-id']
    transmissionSig = header['paypal-transmission-sig']
    transmissionTime = header['paypal-transmission-time']
    webhookId = os.environ['PaypalWebhookId']
    webhookEvent = req.get_json()
    bearerToken = getPaypalToken()

    headers = {
        'Content-type': 'application/json',
```

```
16          'Authorization': "Bearer " + bearerToken
17          }
18
19      payload = {
20          "transmission_id":transmissionId,
21          "transmission_time":transmissionTime,
22          "cert_url":certURL,
23          "auth_algo":authAlgo,
24          "transmission_sig":transmissionSig,
25          "webhook_id":webhookId,
26          "webhook_event":webhookEvent
27      }
28
29      response = requests.post(verificationEndpoint, headers=headers, json=
        payload)
30
31      if(response.json()['verification_status'] == successVerification):
32          return True
33      return False
```

Listing 5.3: Excerpt of how the Request for Validation is implemented

A successful response contains a JSON document, where the field "verification_status" needs to be equal to "SUCCESS". This means that PayPal verified, that the original message was indeed sent by our webhook.

5.5.6 PayPal Buttons

PayPal buttons are a checkout service to embed purchase options in applications, for example HTML websites. It can be selected from a variety of button types, which are available at *sandbox.paypal.com/buttons/*. In an online editor, the appearance of the button can be customized and the PayPal app can be assigned, to which transactions it should be related. When the design of the button is finished, a text snippet can be copied from the editor. The snippet consists of an HTML element, so that it can be easily embedded into a website, as well as the JavaScript implementation of the button logic. To enable monetization for our API system, we implemented the PayPal button solutions for payments and subscriptions. All buttons are maintained in the delegation Function app FactCheckAPIMDelegation.

Payment Button

We use PayPal's Smart Button option, to create a checkout interface for a single-time purchase. After creating the button in the Web interface, the code could be pasted into the HTML content of the delegation function. It is important to note, that we selected the FactCheck merchant app as target shop for the button, which automatically added the

respective app ID to the button snippet. Additionally, we had to extend the JavaScript code so that the product ID and user ID, that come from the Azure API Management service, are passed to the custom_id parameter. The implementation of the PayPal button code snippet for the payment for the product IdaFixBasic can be viewed in the listing 5.4.

```
function initPayPalButton() {
    paypal.Buttons({
      style: {
        shape: 'rect',
        color: 'silver',
        layout: 'horizontal',
        label: 'paypal',

      },

      createOrder: function(data, actions) {
        var productId = document.getElementById("subscriptionproductId").
    value;
        var userId = document.getElementById("subscriptionuserId").value;
        var customdata = userId+"&"+productId;
        var price = document.getElementById("price").value;
        return actions.order.create({
          purchase_units: [
            {
              "description":"IdaFixBasic",
              "amount":{
                  "currency_code":"EUR",
                  "value":price
              },
              "custom_id":customdata
            }
        ]
        });
      },

      onApprove: function(data, actions) {
        return actions.order.capture().then(function(details) {
          redirect();
        });
      },

      onError: function(err) {
        console.log(err);
      }
    }).render('#paypal-button-container');
```

Listing 5.4: Code Snippet of the PayPal Button for the IdaFixBasic Product

Subscription Button

To achieve a payment option for monthly recurring payments, we chose PayPal's Smart-Subscribe button. We designed the button in the PayPal portal, similar to the payment option, however, a subscription button implies the creation of a product, that stores information about the item sold and a payment plan, which automatically handles recurring money transactions. Both were created on the PayPal sandbox website with the merchant account of the FactCheck shop.

As the subscription-product we created the item "IdaFixPremium". Since we wanted to use the subscription billing for the IdaFixPremium product in APIM, we chose a correlating name to avoid confusion. While there are many options to provide information about a product, which can be seen by users during the subscribe process, for this work, we only defined its category as digital goods and the business field as belonging to academic software. As the subscription plan, we installed "FactCheckPremiumPlan", intended to sell the assigned product IdaFixPremium. For the pricing, we set a fixed cost for monthly payments but waived activation fees. After the creation finished, the plan automatically received a plan ID, which we use to reference the plan in the Smart-Subscribe button.

The button type used to create subscriptions embeds in the HTML document the same way as buttons for payments, however, the implementation of the JavaScript code is different. The listing 5.5 features the button configuration which shall create a subscription for the APIM product IdaFixPremium. Compared to payments, creating a subscription button in code requires including the plan ID as an argument, referring to the previously created subscription plan on PayPal.

```
paypal.Buttons({
     style: {
         shape: 'rect',
         color: 'blue',
         layout: 'vertical',
         label: 'subscribe'
     },
     createSubscription: function(data, actions) {
       var productId = document.getElementById("subscriptionproductId").
   value;
       var userId = document.getElementById("subscriptionuserId").value;
       var customdata = userId+"&"+productId;
       return actions.subscription.create({
         plan_id: 'P-XXXXXXXXXXXXXXXXXXXXXXXX',
         custom_id: customdata
       });
     },
     onApprove: function(data, actions) {
       redirect();
     }
```

```
20  }).render('#paypal-button-container-P-XXXXXXXXXXXXXXXXXXXXXXXX');
```

Listing 5.5: Code Snippet of the PayPal Button for the IdaFixPremium Product

5.6 Azure CosmosDB

The Azure cloud provides a self-managed, cloud-based database solution: CosmosDB. During the development of the database, we chose a serverless capacity mode, applying adaptive pricing. Although Cosmos is a database product native to the Azure cloud environment, it supports a variety of APIs of popular databases, like SQL, MongoDB, and Cassandra. The desired core backend must be selected carefully during the creation of the Cosmos instance. Those APIs build the interfaces to query stored documents.

5.6.1 Utilizing the Mongo API provided by Azure Cosmos

To replace the originally used databases, we chose CosmosDB based on a MongoDB core as the cloud solution. The main reason to choose this over the other core technologies is that Mongo is a document-based storage system, similar to the previously used CouchDB. CouchDB has no native Azure solution and is not supported in Cosmos as a core API. Therefore, we switched to Mongo to make transitions and future use more streamlined. Furthermore, CouchDB provides a technology called views to structure and maintain documents. Similar to this, MongoDB also incorporates a view system to call queries on. Hence, choosing a Mongo core can build on the elaborated view schema.

5.6.2 Database Organization

The original system has its data distributed among two distinct databases. Data related to facts is stored in the document database CouchDB and tables concerning precision metrics are maintained in a MySQL database.

In the implementation of this work, a single CosmosDB centralizes the data in one instance. The two internal databases "Factbase", for storing fact related documents and "Precisionmetrics", covering precision metric data, logically separate related documents. Inside these databases, collections further organize documents. The collections and their distribution are depicted in the deployment diagram in figure 5.5. Pursuing this organizational approach, there is only one CosmosDB instance to maintain, instead of two distinct databases.

5.7 Adaption of the IdaFix Extension

Originally, IdaFix is an extension for the Web-browsers Chrome. When loading a website, the extension scans the site for structured data, processes the data into a combined JSON

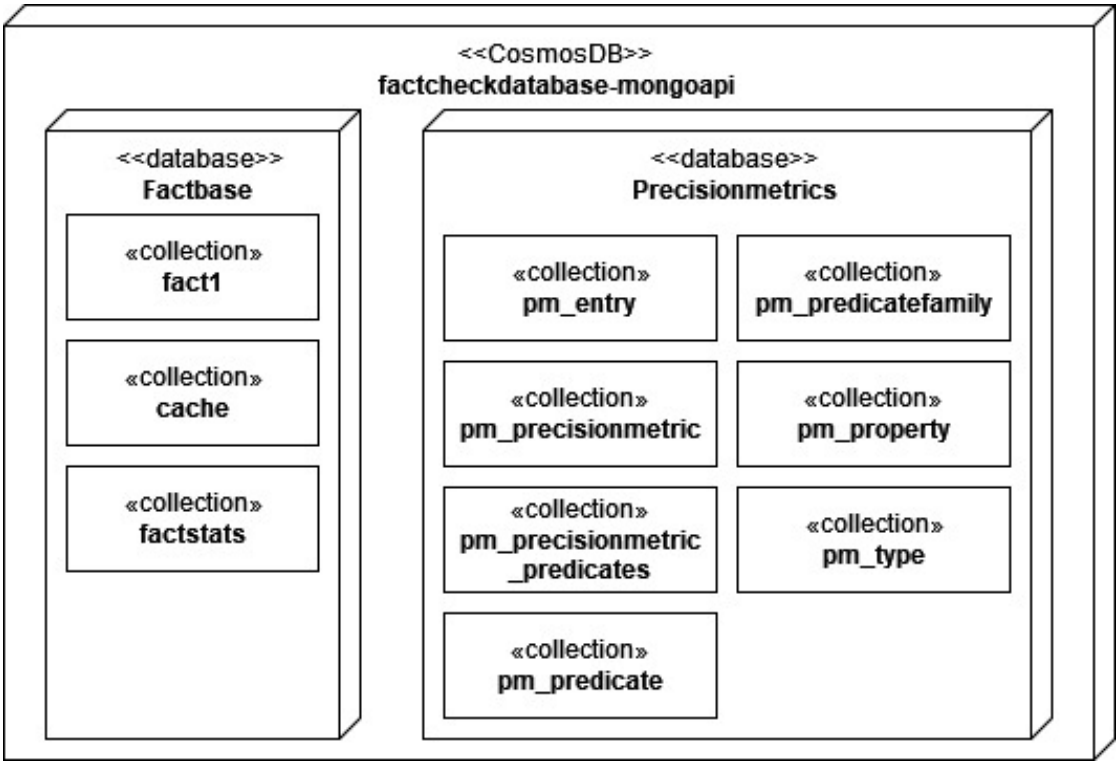

Figure 5.5: Deployment Structure of the Databases and Collections inside the CosmosDB

object, sends it to the server and displays the result in its own extension window. In this project, the target server moved from the previous location into a cloud environment. Fortunately, the extension already had the logic implemented, that the server URL can be changed in the extension user interface. To adapt to the new endpoint provided by the API Management, the URL had to be added in the options form. However, API Management asks for a valid key in every request, which IdaFix did not support by then.

5.7.1 Including API Keys

Azure API Management lists available subscription keys in the developer portal. Users can copy them and place them in the header of requests they send to the API Management. However, IdaFix add-on users should not be obligated to modify the code of the extension by themselves, to add a subscription key. Thus, we continued the end-user centred approach IdaFix was developed in, and therefore implemented another form in the extension's option panel, where the current API key can be inserted. Users can use the already implemented extension storage to persist the value of the key. We modified the original POST request, which initiates the fact comparison so that it also includes the

key's value as a header parameter. The default name for this parameter is "Ocp-Apim-Subscription-Key", which we used, but it could be configured in API Management.

5.7.2 Accessing new API Functionality

To further monetize IdaFix by adding new cloud-based API functionality, we experimented with an API request, that allows users to see a list of all the facts they have contributed to FactCheck, while browsing the Web with IdaFix. The API call as well as the interface for viewing submitted structured data inside the extension window was previously not present in the extension.

We decided to implement this feature in the add-on because in this thesis, IdaFix was already the tool to interact with the cloud system and therefore provided the basis for accessing other APIs. To realize this feature, IdaFix sends a request including the username to the API in APIM. The Azure Function app FactCheckFactView then queries the documents from the CosmosDB and returns a JSON list. A separate JavaScript in the extension manages this task, and the jQuery library is responsible for handling the requests. Realizing the ability to render the fact list in IdaFix also required to change the user interface.

Similar to updating the HTML of the extension's option panel for API keys, a field to allow users to insert their username was added as well as a field for the API URL of Function app FactCheckFactView. A clickable button initiates the HTTP request to the API Management. After successfully receiving the list of documents, it presents the data in a dynamically rendered HTML list. This can be seen in figure 5.6, which shows the fields implemented to add the new API endpoints, as well as subscription keys. It also displays an example of the contributed list of fact data related to a specific user.

5.8 Final System Overview

The figure 5.7 depicts a high-level overview of the final prototypes architecture. It shows the components present in the distributed system and groups them graphically based on their context in Azure. Furthermore, it gives information on how they are connected and where the respective services perform interactions with each other.

5.8.1 Overview

Starting at the top-left, the user's Web browser is depicted as the client. While this is a generic exemplification of how a browser can interact with the system, the quantity of browsers sending requests simultaneously is not limited to one. Browsers interact with the developer portal service of Azure API Management, which acts as the frontend of the final system. Browsers are not meant to interact directly with any other part of the

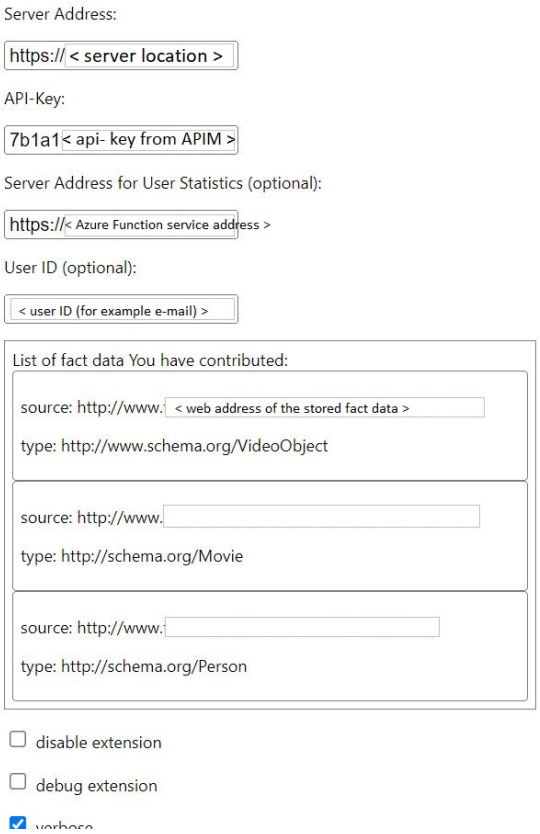

Figure 5.6: Screenshot of the adapted Option Panel in IdaFix

system. The browser extension IdaFix must be installed on the client's browser, which is eligible to call API endpoints exposed by APIM.

Azure API Management is the core of the final system, as it serves the interface for users of our system, acts as gateway for API requests to the Azure Function backend and is used to manage user accounts, products and, subscriptions. There is only a single instance present in the final system, where we can maintain all of these component in the backend of the Azure portal. The portal is the place where we create products, which contain a selection of APIs. For creating user accounts or subscriptions, this approach is not feasible because this would require a developer with access to the Azure account who manually adds them. Users can create accounts by signing up on the APIM developer portal, or they can use the Azure B2C service, which gives users an alternative way to register and authenticate. For B2C users, APIM internally creates a user account with a unique ID. The single B2C instance was connected to APIM in the Azure portal and provides the user flow for sign-in and register activities. At last, APIM's REST API takes care of creating and deleting subscriptions, after an external payment or cancellation step

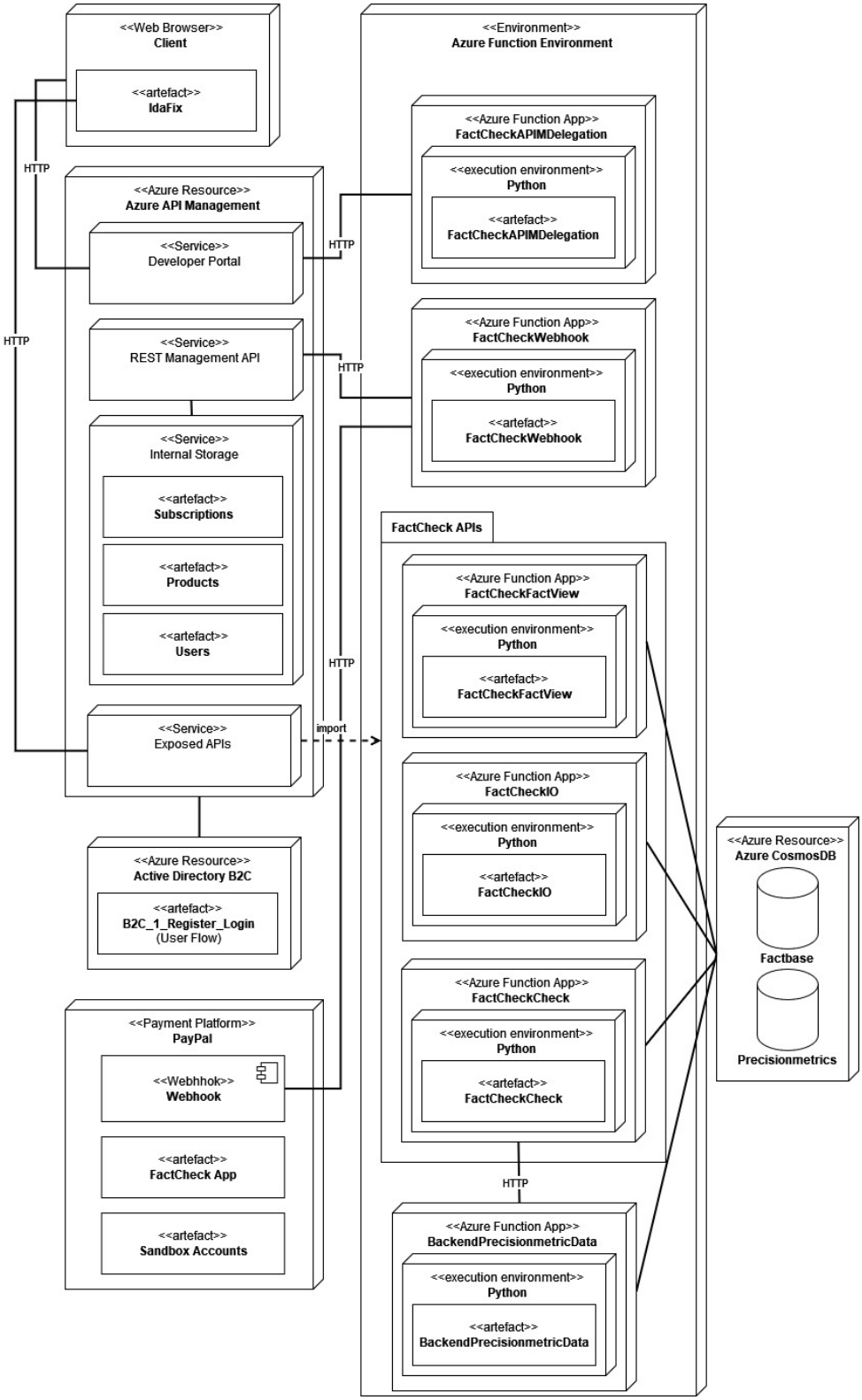

Figure 5.7: System Overview of the developed cloud-based FactCheck Subscription System.

completed.

In the center, the diagram depicts the Azure Function apps. These follow a serverless pricing plan, which means the number of concurrent executions scales with the load of requests. The locally produced functions deploy in the Python environment of the respectiveFunction apps.

FactCheckCheck is the Function app responsible for processing requests from IdaFix and perform the comparison of facts with stored fact data from the internal database. It is one of three Function apps, that are exposed in APIM. The check function is called by APIM and receives the forwarded request data from IdaFix and returns the comparison result to APIM. The Function app queries required fact documents from the Factbase in CosmosDB. and also calls the BackendPrecisionmetricData during the comparison process. This Function application is intended as a backend API to serve a processed document containing precision metric data and comparison predicates. Thus, its functions are not exposed in API Management. FactCheckIO provides various APIs to execute predefined database queries for specific attributes and returns a list of documents that match the query. The last app used as an API in APIM s FactCheckFactView, which returns a list of contributed facts that are related to a username.

FactCheckAPIMDelegation is used as an extension for the APIM Developer Portal. It is there to handle redirected browsers that were forwarded when the user initiated the purchase of a product or cancellation of a subscription. The Function app serves an HTTP page based on the action. For subscription purchases, it loads a PayPal button snippet based on the product ID. The PayPal user interface guides users through the payment flow, and the transaction triggers a notification for the webhook. If it is a cancellation attempt for a subscription, the delegation function instead shows a link, which takes users to their PayPal account, where they issue the actual cancellation. Terminating a subscription also sends a notification to the webhook. The last Azure Function app, FactCheckWebhook, is used as a middleware between PayPal and APIM. Its HTTP endpoint address is registered in the PayPal App, so that whenever a message occurs, PayPal sends a notification request to the function. The webhook verifies if it is a legitimate request and then calls the REST Management API in APIM to create or delete a subscription for a user.

We created one account on the PayPal platform, which encapsulates the app which unites all transactions assigned to our project and the sandbox accounts, which are used to simulate transactions. These accounts are created from the overlying PayPal account, but can be used on the sandbox website. We created the FactCheck shop app and the webhook with the main PayPal account.

Finally, on the right, the Azure CosmosDB is shown. Only one instance of CosmoDB exists in the final system. It contains two databases which are further structured in collections. This component replaces the two distinct databases in the original system.

Factbase stores data regarding facts. Precisionmetrics is the storage for precision metric related documents.

5.8.2 Use Case Scenarios

To graphically represent the logical flow of interactions in the final system, the most important ones are made explicit in the diagrams in this section.

To use APIs provided by the final prototype, a subscription is required. This applies to the usage of IdaFix too. Figure 5.8 shows the steps users need to follow, if they want to use the browser extension to connect to the system. In the diagram, lanes depict the location where activities are performed. For the starting point, it is assumed that users already have an active PayPal account.

Figure 5.9 describes the logical flow of messages sent in the process when using the fact comparison API with the browser extension IdaFix. In this diagram, a specific use case is exemplified, where facts about RoboCop, gathered from *metacritic.com* are compared to fact data loaded from the Factbase. It starts with the Web browser requesting a website. After the page was loaded, IdaFix starts gathering structured data and invokes the API management endpoint. APIM forwards the request to the check function in FactCheckCheck, which can be clearly identified as the main component in this process. The function performs the comparison computation and the diagram also shows, at which points in the process, the function requests fact data and precision metric data from the Cosmos database.

The sequence diagram presented in figure 5.10 shows the interaction flow of the components involved in the workflow of purchasing a subscription. This diagram assumes, that a user has already created a user account in the APIM resource and successfully logged into the developer dashboard. In this example, the IdaFixPremium product is purchased. As the first step, the user's Web browser needs to load the Web page of the product, where they can subscribe to the product by pressing the subscribe button. Clicking the button forwards to the Azure Function FactCheckAPIMDelegation which handles the delegation processes. This function dynamically generates an HTML page and inserts, depending on the product ID, the correct PayPal button, which initiates the money transaction when pressed. This specific interaction opens up a dedicated browser window, where the user at first has to log in and secondly has to submit the purchase. After a successful purchase, the window is closed and the browser is redirected back to the delegation function, however, to a different URL, where a success message is displayed.

The interaction flow for payments is analogue to purchasing a monthly billed subscription. The delegation function determines the monetization method and renders a different PayPal button, based on the decision. Besides that, PayPal internally adds a record for the subscription to the customer sandbox account.

After buying a subscription and starting a payment plan, PayPal internally takes

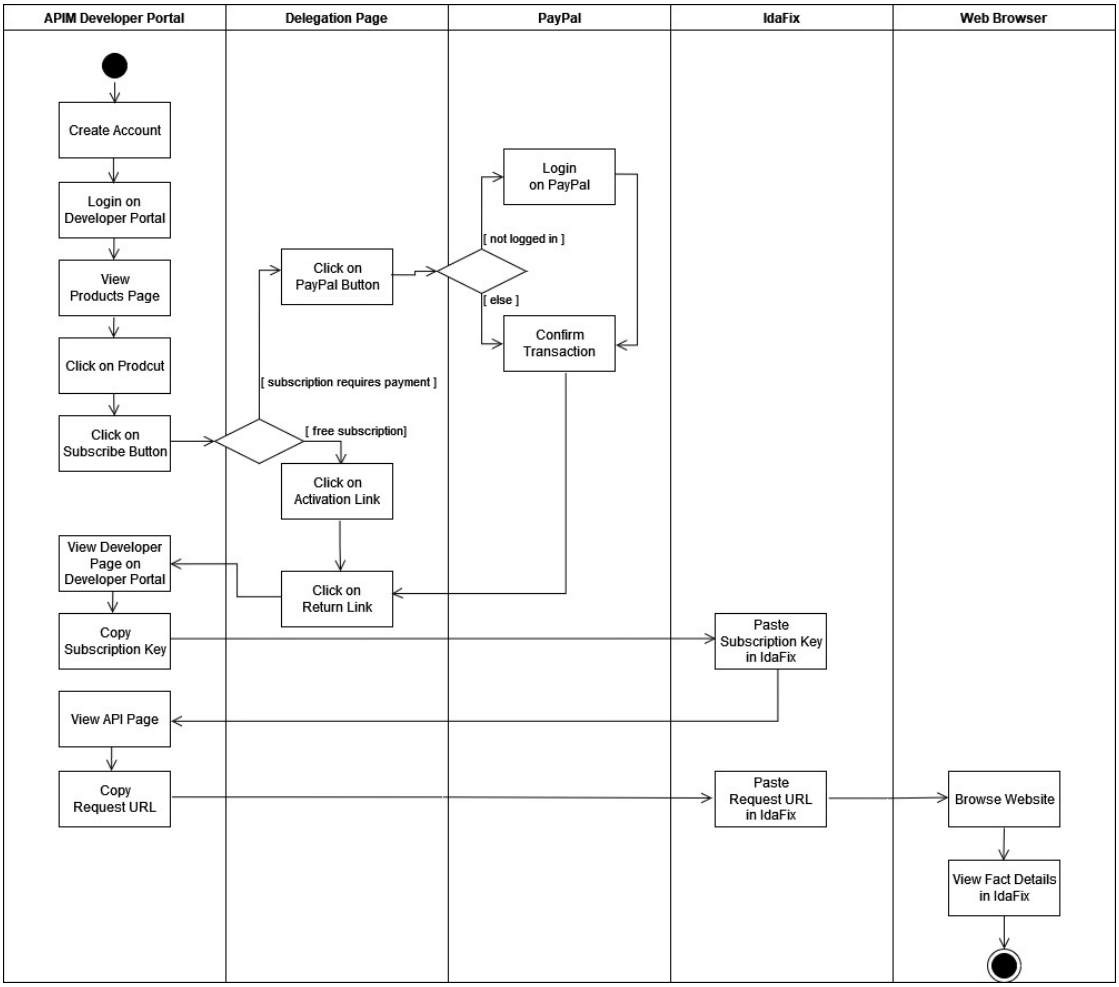

Figure 5.8: Required Actions for IdaFix Users to use a Subscription

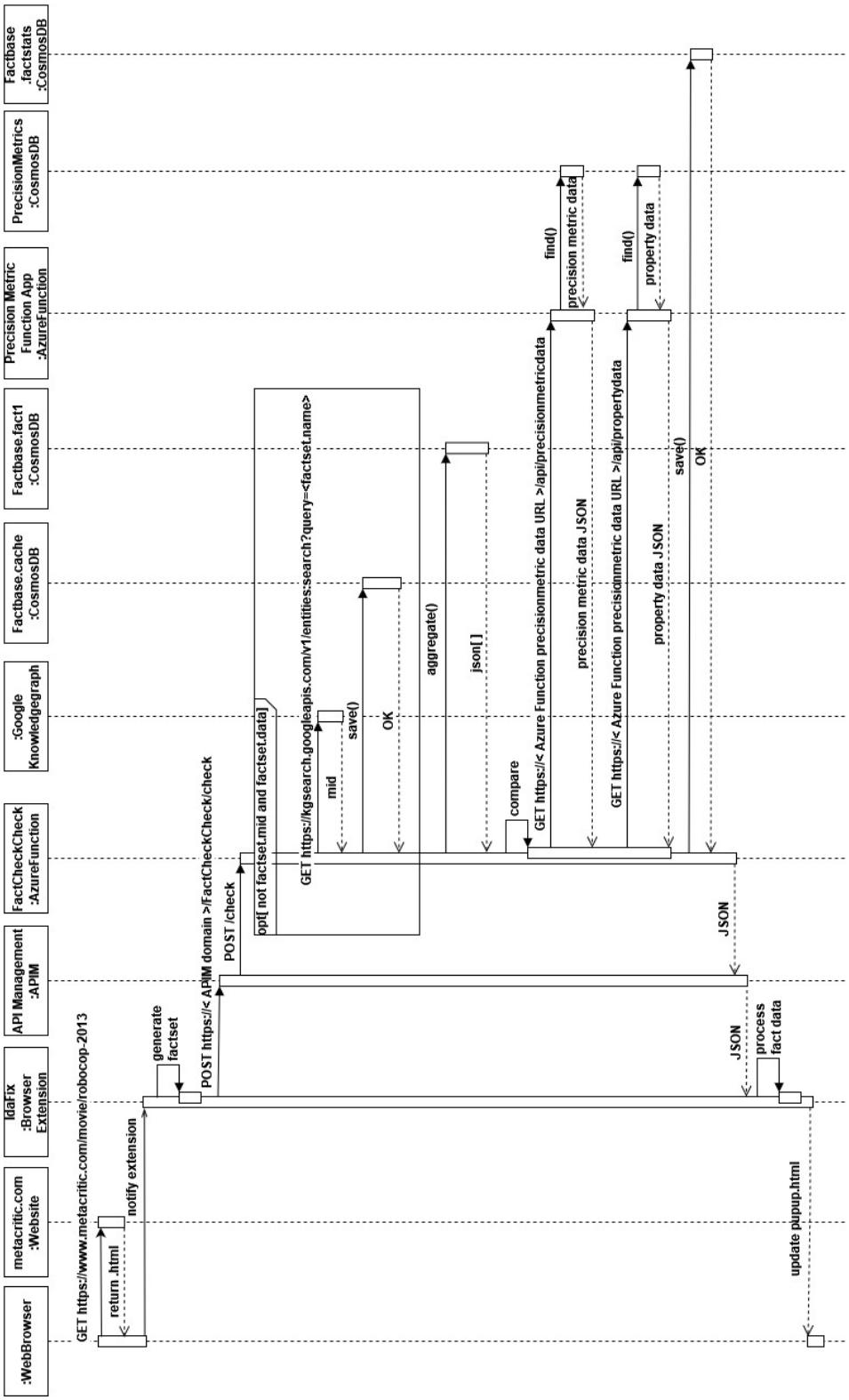

Figure 5.9: Sequence Diagram depicting the logical Flow when using the Fact Comparison API with IdaFix

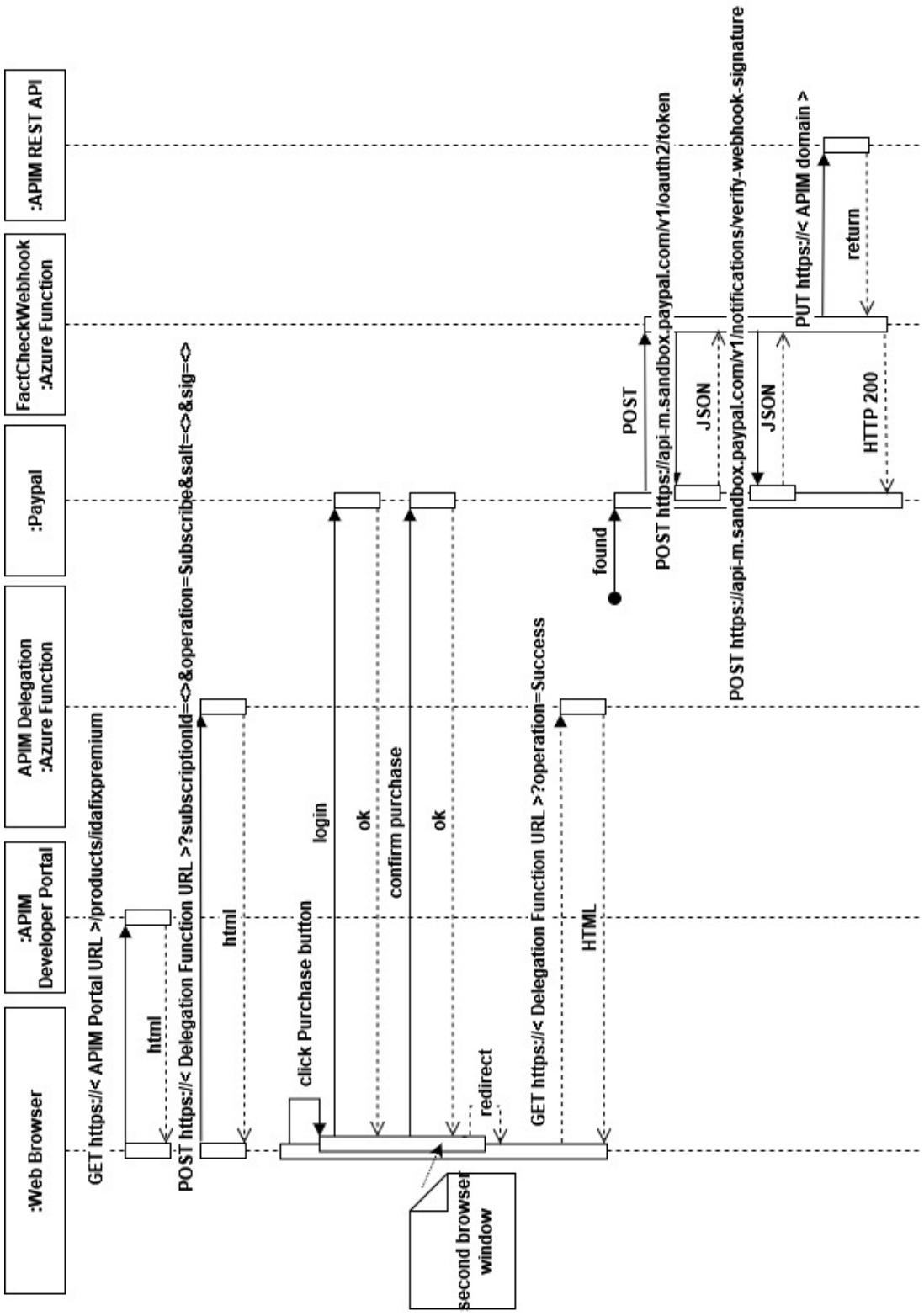

Figure 5.10: Sequence Diagram featuring the Information Flow when purchasing a Subscription.

care of the money transfer between the accounts and there is no interaction with our infrastructure. To stop recurring fees of a subscription, users must actively cancel the subscription. The required procedure is exemplified in figure 5.11, where the subscription IdaFixPremium is cancelled and required interactions of PayPal and the Azure instances are shown. The first step for a user is to open the site on the developer portal, where active subscriptions are listed. There, they need to click on the cancel button for the IdaFixPremium subscription. This will redirect the browser to the delegation Function app. Since PayPal solely handles money transactions, users must manually cancel the subscription in the PayPal portal. For this reason, FactCheckAPIMDelegation renders an HTML page, displaying information about the cancellation process and includes a link to the location for subscription cancellation on PayPal. On the PayPal portal, users must select the active subscription associated with the IdaFixPremium and use the available cancel option. After a successful cancellation step, PayPal removes the payment plan and notifies the webhook Azure Function about the termination. FactCheckWebhook in turn will verify the request and delete the IdaFixPremium subscription for the respective user from the API Management environment by sending an HTTP Delete request to the REST API.

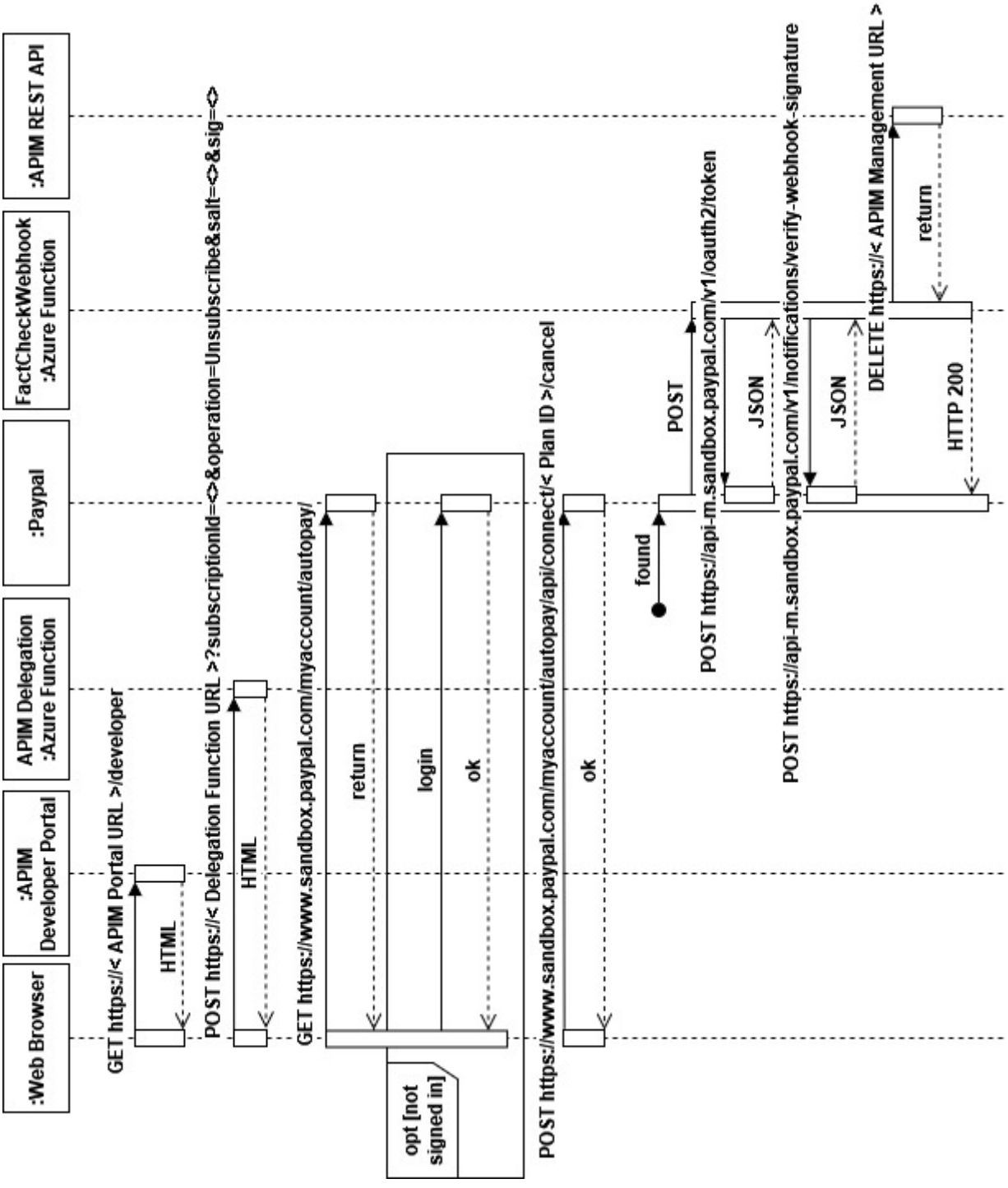

Figure 5.11: Sequence Diagram featuring the Information Flow when cancelling a Subscription.

6 Evaluation of the Prototypical Implementation

This chapter outlines the progress and achievements of this project and to what degree the implementation meets the goals that were set at the beginning of this thesis. The first section evaluates the final result based on the original goals presented in the introduction. In the following section, the costs of the used cloud resources will be discussed and an estimated example for production usage is given. The chapter concludes with summarizing the limitations of the prototype and this thesis.

6.1 Qualitative Analysis

In this section, we describe how well the objectives that were set for the design and the implementation of the desired prototype, could be achieved with the chosen approach. Therefore, we evaluate the system based on the goals specified in the introduction:

Realizing the Prototype's Functionality based on Azure cloud Services

The system developed in this thesis should use Azure. The functionality of the prototype is realized with the use of a variety of Azure products. API Management fulfils the requirement of subscriptions and exposes APIs for users. Active Directory B2C takes over the account management for users. The main part of logic and code was programmed as Azure Functions. With Function apps, we were able to program the logic for fact comparison, database access, creating and deleting subscriptions and integrating PayPal services. By using Azure Functions, we could produce modular APIs that can be invoked via HTTP. As database solution, we use Azure CosmosDB to store and retrieve document data in the cloud system. The database adopts the responsibilities of the databases used in the original system. Only the payment processing is outsourced to PayPal.

We see the criteria for basing the functionality on Azure as met because the logic of the FactCheck system could be deployed in Azure. PayPal is the only non-Azure service in our system. However, to some degree, the connections to the external platform required integration into the system logic as well, which we achieved with Azure Functions.

Apply Cloud Functionalities for Scalability

The goal here was to have a solution, that is scalable to a potential high load of traffic. Azure products are intentionally applicable for use-cases where scalability is a concerning factor. By using Azure services, any application may be further scaled. The main code is implemented as Azure Functions, which are highly scalable too. However, there are restrictions, which come with the respective payment plan. This affects APIM, Functions and CosmosDB, where we used the lowest pricing tier, that still supported our requirements. Hence, selecting a more sophisticated plan may increase performance on high load.

Finding a Solution for a Subscription System, including a free and a paid version, that is applicable for the User Groups of Content Providers and End-Users

This and the following questions are mostly addressed by APIM, which provides a subscription management system. It allows importing and publish API endpoints and group them together into bundles as products. With those, APIs can be dynamically grouped and be made available for purchase, so that users can redeem them at any time. While users can browse available products on the developer portal, administrators can modify APIs and products in the Azure portal.

Free and paid subscriptions are distinguished in the delegation function for APIM. For free products, the Azure Function creates a subscription immediately. We realize paid subscriptions by the integration of the payment platform PayPal. The final system provides subscriptions in three tiers: one free, a basic and a premium subscription.

Although the prototype could potentially be used by every user group. During its development, the prototype tended to focus more on the areas of end-users than content providers. With the integration of subscriptions, implementation of the fact comparison and IdaFix as the main tool to access our services, the supported functionality in the result orients towards data consumers.

Subscription Plans can be redeemed by Users to Access certain APIs

We realize subscription plans with Azure API Management, where we use products to define the APIs that should be included in a subscription. In APIM the subscription itself determines the right for a specific user to use the functionality of a certain product.

In the final prototype, users can redeem subscriptions by selecting the desired product on the APIM developer portal. Free subscriptions can be activated on the page of the delegation function. Paid subscriptions additionally require users to perform a payment with PayPal.

Implementing an API Layer that serves as Middleware between End-Users and the Internal System

From an architectural perspective, APIM is the middleware of the system, which separates incoming user requests from internal parts of the system. It generates distinct access points for APIs, thus users have no information about the location of the exposed Function apps. Additionally, subscription keys are required to use APIs of the APIM instance. This means, users must purchase a subscription, before APIM passes requests to the internal system. In the case of free products, at least an account must be created in API Management to use the API. As a further note, Azure Functions, that interact with CosmosDB, but are also used as an endpoint in APIM, help to further decouple external requests from sensitive database resources.

Elaborate a Design Approach so that APIs can be assigned to paid or free Subscription Plans adaptively

We use Azure API Management to maintain subscriptions and dynamically assign them API functionality. In its Azure portal, the list of APIs of a product can be modified by developers. Changes propagate to new and existing subscriptions at runtime. Thus, users can use their existing keys of a product, but access the new APIs that are included in the bundle.

The distinction between paid and free subscriptions is not handles by APIM and happens in the Azure Function used for subscription delegation. There, a predefined price list in code determines the mapping of a value to a product ID. From a user perspective, changing the price for a product would not affect the API bundle in it and vice versa.

To maintain the price management for products in this location is a practical solution because the Function directly receives the product information from API Management and is responsible to render the shop interface. A downside to this approach is that the management of the price list may become tedious, the more and more products are included.

6.2 Cost Analysis

Cloud services provide infrastructure, services and software solutions that can be used at the expense of money. As a customer, it is no longer required to take care of physical computing components or even set up operating systems. An outstanding advantage is however the seemingly unlimited amount of computing power. As long as the budget allows it, more and more computing resources can be added to the own system architecture, to satisfy the current demand. It is also possible to scale down and remove resources dynamically to reduce costs. We also considered this while developing a cloud-based

subscription solution for FactCheck.

6.2.1 Azure RU System

While it can be accurately determined, how many bytes a file occupies in a storage, it is not that trivial to measure computational expenses. To charge customers with the accumulated costs, they created a metric system, which is designed to provide countable units for variable consumption of computing power and time. Request Units (RU) are commonly used for Azure resources that follow a serverless pricing plan.

6.2.2 Serverless versus Provisioned

When initializing a new Azure resource, it can often be chosen between a serverless and a provisioned pricing plan for the resource. Serverless resources do not require users to maintain an explicit machine and are executed and also scaled up or down depending on demand. Provisioned, on the other hand, sets a minimum guaranteed throughput of requests, is suitable for a constant and high load of traffic.

6.2.3 Monthly Costs during Development

The costs in our Azure account were charged at a monthly rate by combining the expenses of all used resources As soon as a resource is created for the Azure subscription, it is included in the accumulation of the monthly bill. Our monthly costs during development were definitely lower compared to a production environment under a high load of user requests. The prototype of this thesis is a distributed system that is a combination of one API Management resource, one CosmosDB, and six distinct Function Apps.

The pie chart in figure 6.1 shows an example of a monthly bill we got during development. It can be easily seen, that APIM takes the biggest part of the generated costs. Functions have a monthly credit of free calls, or computation time, whichever runs out first. During development, we consumed a fraction of the free credit. The free quota also replenishes the next month, which is why there are no records for function execution costs. However, slight deviations occurred between months, as APIM is billed per hour, and months have a different number of days. Also, costs in CosmosDB are understandably higher during months, where we tested the usage of the database and generated more traffic in the process. The components that create the highest cost are *apim-factcheck-dev*, the API Management and *factcheckdatabase-mongoapi* as the CosmosDB resource. The chart does not include costs for Azure Functions because we never depleted the monthly free quota.

During development, we found that serverless resources are the way to go to experiment with resources and test implementation. Since computation time is only billed on actual execution time, the small number of test requests accumulate almost no costs. We also found that generally working with data and computation is more expensive than statically

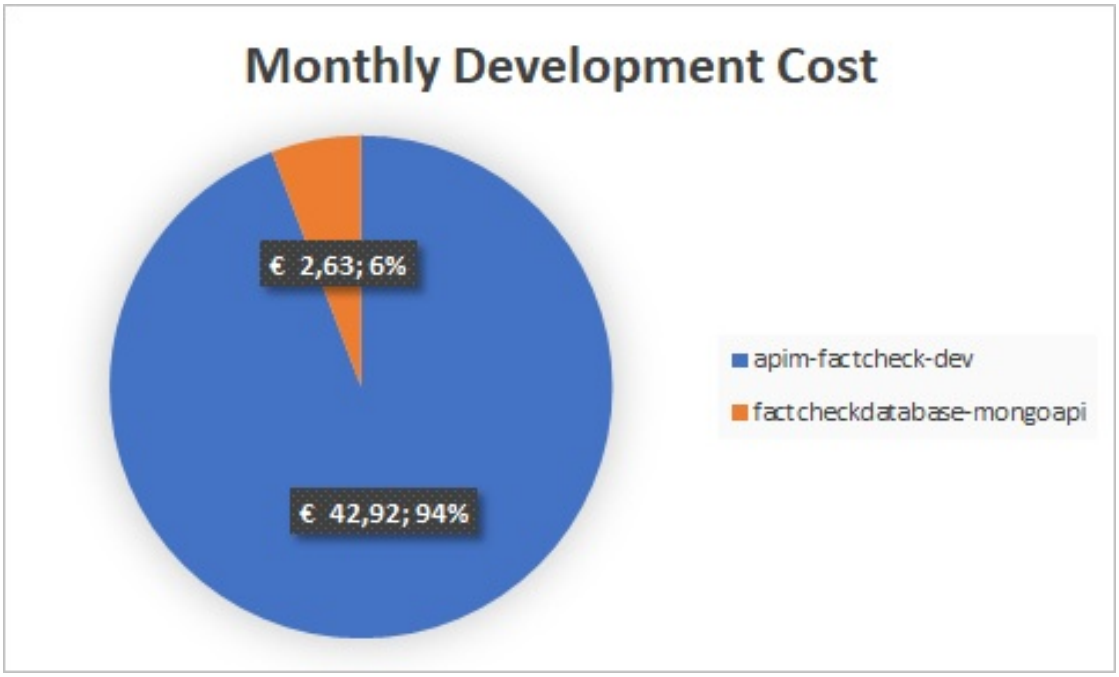

Figure 6.1: Example of the monthly Expenses during Development

storing data. A resource based on a provisioned plan would not be fully occupied anyway. For a production implementation, however, we may switch to a higher tiered plan, which allows a greater request throughput.

6.2.4 Expected monthly Costs in Production

To get an idea of the accumulated costs of the system resources in a production scenario, we use Azure's pricing calculator[1].For this example, we assumed a hypothetical scenario to estimate potential costs for individual components. We defined for all resources that they reside in the geolocation EU West.

For Function Apps, we set ten million function executions per month, which includes API requests from users and requests that are started internally from other functions, for example retrieving precision metric data in the check function. We assumed an execution time of five seconds, mostly because the check function takes more time due to additional requests to external resources, although this reflects a worse case scenario. We chose to stick to the consumption pricing for serverless Function Apps.

Azure API Management forms the core of the system, and hence it should be as performant as possible. Therefore, we assumed a standard pricing tier, which is, after

[1]https://azure.microsoft.com/en-us/pricing/calculator/ (Last accessed: 2022.06)

basic, two tiers higher than the developer pricing, but also more expensive.

Lastly, the CosmosDB imposes monthly fees based on the consumed units by requests and the stored amount of data. By the time of this work, the amount of stored data was around seven Gigabytes. For the calculation example, we increased the number to a potential storage need of hundred Gigabytes. As it can be estimated that the database must handle a great deal of queries, we chose the standard provisioned plan to guarantee a fixed throughput and assumed a consumption of 500 RUs per second. The calculator splits these costs into 23.34€ for storage and 27.26€ for RU consumption from queries.

Finally, with the assumed configuration and values, the production scenario would impose a monthly cost of 780.88€. Figure 6.2 breaks down the costs for each resource type. Looking at the chart, it is clear that Azure API Management would still be the most expensive resource. Although the number being high in relation, the execution of Function Apps is comparatively cheap.

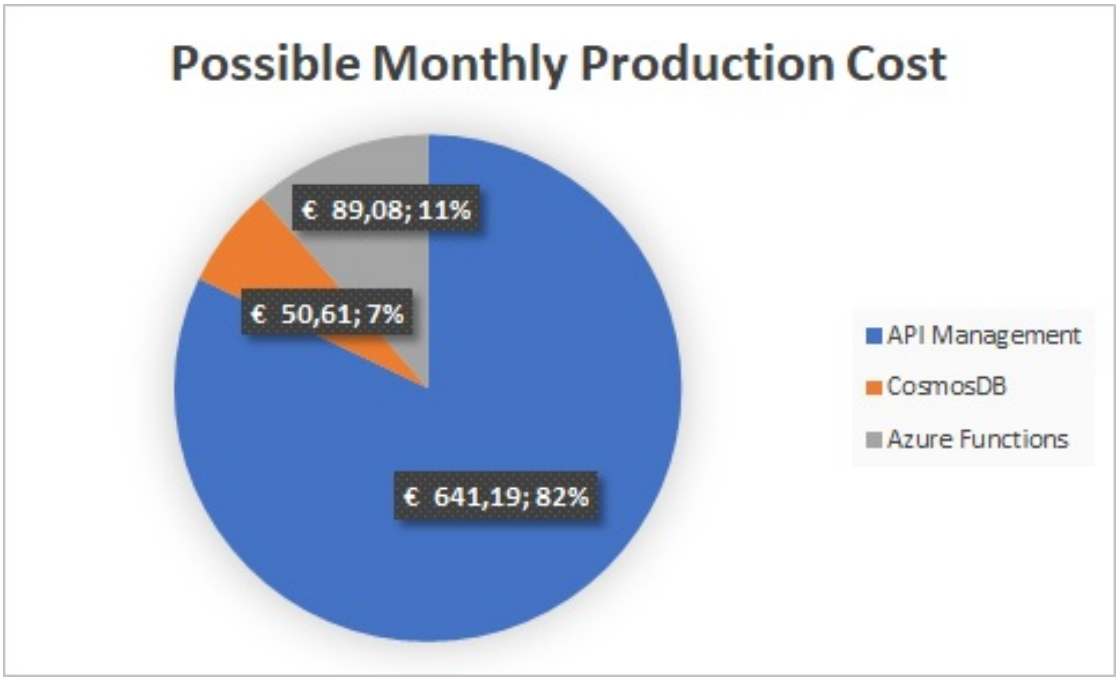

Figure 6.2: Estimated Costs for the Prototype in a potential Production Scenario

6.3 Limitations

The previous chapters explained how we implemented the system components to produce an API service in Azure. In multiple sections, it was outlined, that numerous steps were

necessary to overcome the restrictions of the cloud service. For this reason, adapting to the cloud service also restricts this specific implementation to be run in Azure exclusively. This causes a vendor lock for the system. Switching to other cloud providers may be difficult and could involve refactoring and redesigning the components so that they can be deployed in a new environment.

A similar scenario applies to the integration of PayPal. The implemented functionality to perform transactions is built around PayPal's APIs, and other payment providers will have their API endpoints structured differently. Additionally, PayPal is not the only payment provider, that is used to buy goods online. Thus, the selling of API keys is restricted to buyers who are willing to use PayPal and maintain an account there. Furthermore, the payment processing in this prototype is handled on PayPal's sandbox website. While the implementation of this service allows testing the integration of PayPal in the context of FactCheck, this specific prototype does not come in contact with actual user accounts on the main website.

Another issue is, that the version this prototype is utilizing is probably outdated or even not supported in the future. This regards the deprecated developer portal of API Management, which is scheduled for retirement in 2023[2]. Then, only the new version will be accessible, and this would break the forwarding to the delegation page when purchasing subscription keys. Furthermore, Azure keeps upgrading their services, which means, that as long as the service uses Azure products, it is crucial to adapt the program to new versions and updates.

[2] https://docs.microsoft.com/en-us/azure/api-management/developer-portal-deprecated-migration (Last accessed: 2022.06)

7 Conclusion

This chapter concludes the thesis and recapitulates on the research question. The insights gained during development will be summarized, and the thesis ends with an outlook on possible future work.

7.1 Summary

The goal of this research was to realize APIs for FactCheck based on Azure cloud services and make them accessible through subscription-based monetization methods. This thesis proposes a prototypical system structure implemented in Azure.

The evaluation of the created system shows, that the proposed architecture is capable of providing API endpoints and give users the option to purchase subscription keys by using PayPal as payment processor. The findings of this research show that Azure API Management is suitable for exposing parts of FactCheck as API services. This service was beneficial in particular because it already provides API management, user handling and a Web interface, which was crucial to meet the research interests of this work. We use Active Directory B2C to extend the account management of the API management component. Azure Functions contain the APIs for the subscription service and run parts of the refactored code of the original FactCheck system. From this work, Function apps emerged to provide the functionality for the fact comparison API, to execute specific database queries and to query fact contributions of a user. To handle payment processes, we introduced PayPal as a payment processor. We implemented a specific Azure Functions that interacts with PayPal's APIs and allows making purchases by providing PayPal Checkout buttons. Another Function acts as the callback for a PayPal webhook, receives events after successful transactions and creates subscriptions in the API Management. To store fact data and other related documents in the cloud, we instantiated a CosmosDB instance, based on the MongoDB core, which maintains the data in different collections.

7.2 Future Work

From the work of this thesis, a fundamental basis for a distributed cloud system emerged, which allows exposing APIs and has an integrated payment solution. This working basis can be extended incrementally in future projects, without the need to refactor the

architecture.

For the prototype, we refactored parts of the original FactCheck functionality and used Azure Functions to encapsulate them for APIs. Further APIs could be implemented as Function apps over time and exposed through APIM.

Furthermore, these suggested APIs could be integrated in the monetization model. This thesis presents a base set of products which prove the concept of redeeming subscription-keys through either a one-time purchase or a subscription. The number of offered products could be extended, as more APIs are developed and exposed through the API management.

In the final state of this thesis, payment processes are carried out on PayPal's sandbox service to test the integration of PayPal. In the future, if the proposed prototype reaches a state where it is ready for production, the connection should be altered to PayPal's main site, making transactions with real currency possible.

Azure API Management provides insights into the usage data of the APIs and visualizes them as statistics. As a long-term study goal, it would be interesting to see, which APIs users call the most. This might yield information, that can improve the future development of system components. If users tend to extensively use the check functionality compared to personal score features, like listing the contributed facts, the former mentioned functionality could be further improved. User statistics may also identify the most popular products, based on the number of purchases. If statistics show, that a certain product is rarely purchased, while others, that impose a lower fee, get subscribed to on a regular basis, it may be useful to adapt the price for this product accordingly.

Bibliography

[1] Petter Bae Brandtzaeg and Asbjørn Følstad. Trust and distrust in online fact-checking services. *Commun. ACM*, 60(9):65–71, August 2017. ISSN 0001-0782. doi: 10.1145/3122803. URL https://doi.org/10.1145/3122803.

[2] Sylvie Cazalens, Philippe Lamarre, Julien Leblay, Ioana Manolescu, and Xavier Tannier. A content management perspective on fact-checking. In *Companion Proceedings of the The Web Conference 2018*, WWW '18, page 565–574, Republic and Canton of Geneva, CHE, 2018. International World Wide Web Conferences Steering Committee. ISBN 9781450356404. doi: 10.1145/3184558.3188727. URL https://doi.org/10.1145/3184558.3188727.

[3] Se-Hak Chun and Byong-Sam Choi. Service models and pricing schemes for cloud computing. *Cluster Computing*, 17(2):529–535, Jun 2014. ISSN 1573-7543. doi: 10.1007/s10586-013-0296-1. URL https://doi.org/10.1007/s10586-013-0296-1.

[4] J. A. Cordón-García, R. Gómez-Díaz, J. Alonso-Arévalo, and A. García Rodríguez. Subscription models like innovative digital reading system. In *Proceedings of the Second International Conference on Technological Ecosystems for Enhancing Multiculturality*, TEEM '14, page 523–530, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450328968. doi: 10.1145/2669711.2669950. URL https://doi-org.uaccess.univie.ac.at/10.1145/2669711.2669950.

[5] FactCheck. Factcheck, 2022. URL https://www.factcheck.org.

[6] FullFact. Fullfact, 2022. URL https://fullfact.org.

[7] Naeemul Hassan, Fatma Arslan, Chengkai Li, and Mark Tremayne. Toward automated fact-checking: Detecting check-worthy factual claims by claimbuster. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1803–1812, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450348874. doi: 10.1145/3097983.3098131. URL https://doi.org/10.1145/3097983.3098131.

[8] Naeemul Hassan, Gensheng Zhang, Fatma Arslan, Josue Caraballo, Damian Jimenez, Siddhant Gawsane, Shohedul Hasan, Minumol Joseph, Aaditya Kulkarni, Anil Kumar Nayak, Vikas Sable, Chengkai Li, and Mark Tremayne. Claimbuster: The first-ever

end-to-end fact-checking system. *Proc. VLDB Endow.*, 10(12):1945–1948, August 2017. ISSN 2150-8097. doi: 10.14778/3137765.3137815. URL https://doi.org/10.14778/3137765.3137815.

[9] Jakob Hirschl. Factchecking - towards conflict analytics (dashboard) for content providers. Master's thesis, Universität Wien, 2022.

[10] Christian Humborg and Thuy Anh Nguyen. *Fake News – Fact Checking*, pages 29–31. Springer Fachmedien Wiesbaden, Wiesbaden, 2018. ISBN 978-3-658-20959-9. doi: 10.1007/978-3-658-20959-9_9. URL https://doi.org/10.1007/978-3-658-20959-9_9.

[11] N. Jnoub, W. Klas, P. Kalchgruber, and E. Momeni. A flexible algorithmic approach for identifying conflicting/deviating data on the web. In *2018 International Conference on Computer, Information and Telecommunication Systems (CITS)*, pages 1–5, 2018.

[12] Peter Kalchgruber, Wolfgang Klas, Nour Jnoub, and Elaheh Momeni. Factcheck - identify and fix conflicting data on the web. In Tommi Mikkonen, Ralf Klamma, and Juan Hernández, editors, *Web Engineering*, pages 312–320, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91662-0.

[13] Wolfang Klas and Peter Kalchgruber. Factcheck - identify and fix conflicting data on the web [vision], internal communication, 2017.

[14] S. Krishnan and M. Chen. Cloud-based system for fake tweet identification. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 720–721, 2019.

[15] Wee Yong Lim, Mong Li Lee, and Wynne Hsu. Ifact: An interactive framework to assess claims from tweets. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management*, CIKM '17, page 787–796, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349185. doi: 10.1145/3132847.3132995. URL https://doi.org/10.1145/3132847.3132995.

[16] Charles Lovering, Anqi Lu, Cuong Nguyen, Huyen Nguyen, David Hurley, and Emmanuel Agu. Fact or fiction. *Proc. ACM Hum.-Comput. Interact.*, 2(CSCW), November 2018. doi: 10.1145/3274380. URL https://doi.org/10.1145/3274380.

[17] Hongqian Karen Lu. Keeping your api keys in a safe. In *2014 IEEE 7th International Conference on Cloud Computing*, pages 962–965, 2014. doi: 10.1109/CLOUD.2014.143.

[18] Alex Carmine Olivieri. Improving automated fact-checking through the semantic web. In Alessandro Bozzon, Philippe Cudre-Maroux, and Cesare Pautasso, editors,

Web Engineering, pages 519–524, Cham, 2016. Springer International Publishing. ISBN 978-3-319-38791-8.

[19] Anjan Pal and Cliff Loke. Communicating fact to combat fake: Analysis of fact-checking websites. In *Proceedings of the 2019 International Conference on Information Technology and Computer Communications*, ITCC 2019, page 66–73, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450372282. doi: 10.1145/3355402.3355415. URL https://doi.org/10.1145/3355402.3355415.

[20] Günther Schuh, Jana Frank, Philipp Jussen, Calvin Rix, and Tobias Harland. Monetizing industry 4.0: Design principles for subscription business in the manufacturing industry. In *2019 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1–9, 2019. doi: 10.1109/ICE.2019.8792607.

[21] Jafar Shayan, Ahmad Azarnik, Suriayati Chuprat, Sasan Karamizadeh, and Mojtaba Alizadeh. Identifying benefits and risks associated with utilizing cloud computing. *CoRR*, abs/1401.5155, 2014. URL http://arxiv.org/abs/1401.5155.

[22] Avvari Sirisha and G. Geetha Kumari. Api access control in cloud using the role based access control model. In *Trendz in Information Sciences Computing(TISC2010)*, pages 135–137, 2010. doi: 10.1109/TISC.2010.5714624.

[23] Snopes. Snopes, 2022. URL https://www.snopes.com/.

[24] Francesca Spezzano, Anu Shrestha, Jerry Alan Fails, and Brian W. Stone. That's fake news! reliability of news when provided title, image, source bias & full article. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW1), apr 2021. doi: 10.1145/3449183. URL https://doi.org/10.1145/3449183.

[25] Radu Tudoran, Alexandru Costan, Gabriel Antoniu, and Luc Bougé. A performance evaluation of azure and nimbus clouds for scientific applications. In *Proceedings of the 2nd International Workshop on Cloud Computing Platforms*, CloudCP '12, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311618. doi: 10.1145/2168697.2168701. URL https://doi.org/10.1145/2168697.2168701.

[26] Erik Wilde and Mike Amundsen. The challenge of api management: Api strategies for decentralized api landscapes. In *Companion Proceedings of The 2019 World Wide Web Conference*, WWW '19, page 1327–1328, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450366755. doi: 10.1145/3308560.3320089. URL https://doi.org/10.1145/3308560.3320089.

Acronyms

API Application Programming Interface.

APIM API-Management.

CRUD Create Read Update Delete.

DB Database.

HTTP Hyper Text Transfer Protocol.

IaaS Infrastructure as a Service.

IDE Integrated Development Environment.

JSON Javascript Object Notation.

NoSQL Not only SQL.

PaaS Platform as a Service.

RDF Resource Description Framework.

REST Representational State Transfer.

SaaS Software as a Service.

SAS Shared Access Signature.

SQL Structured Query Language.