LUT University
School of Engineering Science
Software Engineering
Master’s Programme in Software Engineering and Digital Transformation

Konsta Ala-Ilomäki

APPLICATION PROGRAMMING INTERFACE MANAGEMENT FOR CLOUD ENTITIES OF AN ENTERPRISE RESOURCE PLANNING SOFTWARE

Examiners:	Professor Ajantha Dahanayake
	M. Sc. (Tech.) Pasi Tapio
Supervisors:	Professor Ajantha Dahanayake
	M. Sc. (Tech.) Pasi Tapio

TIIVISTELMÄ

LUT-yliopisto
School of Engineering Science
Tietotekniikan koulutusohjelma
Master's Programme in Software Engineering and Digital Transformation

Konsta Ala-Ilomäki

Toiminnanohjausjärjestelmään kuuluvien pilvipalveluiden ohjelmointirajapintojen hallinta

Diplomityö

2019

69 sivua, 5 kuvaa, 12 taulukkoa, 2 liitettä

Työn tarkastajat: Professori Ajantha Dahanayake
DI Pasi Tapio

Hakusanat: ohjelmointirajapinta, toiminnanohjausjärjestelmä, versionhallinta, dokumentointi, testaus, suunnittelu
Keywords: API, application programming interface, ERP, enterprise resource planning software, version management, documentation, testing, design

Tieto Oyj kehittää ja ylläpitää toiminnanohjausjärjestelmää suurelle metsäteollisuusyritykselle. Järjestelmän oheen kehitetään mobiilisovellusta, joka kutsuu järjestelmän palveluita pilvipalveluun sijoitettujen ohjelmointirajapintojen avulla. Diplomityön tavoitteena on selvittää kirjallisuuteen perustuen, miten ohjelmointirajapintoja kannattaa suunnitella ja hallita sekä toteuttaa alustava suunnitelma yhdelle ohjelmointirajapinnalle ja sen hallinnalle. Työn perusteella voidaan todeta, että ohjelmointirajapintojen suunnittelua on tutkittu aikaisemmin melko paljon, mutta ohjelmointirajapintojen hallinta on hieman vähemmän tutkittu aihe. Työn tuloksena luodaan yleiskatsaus ohjelmointirajapintojen hallintaan sekä viitekehys, jota on mahdollista hyödyntää ohjelmointirajapinnan ja sen hallinnan toteuttamisen suunnittelussa. Työn tulosten perusteella ei voida kuitenkaan arvioida ohjelmointirajapintojen tai hallinnan toteutuksen laatua. Laadun arviointi ja hallinnan toteutus jätetään myöhemmin tutkittavaksi.

ABSTRACT

LUT University
School of Engineering Science
Software Engineering
Master’s Programme in Software Engineering and Digital Transformation

Konsta Ala-Ilomäki

Application programming interface management for cloud entities of an enterprise resource planning software

Master’s Thesis
2019

69 pages, 5 figures, 12 tables, 2 appendices

Examiners: Professor Ajantha Dahanayake
M. Sc. (Tech.) Pasi Tapio

Keywords: API, application programming interface, ERP, enterprise resource planning software, version management, documentation, testing, design

Tieto Corporation develops and maintains an enterprise resource planning system (ERP) for a large forest industry corporation. In addition to the ERP system, a mobile application is developed. The mobile application calls the services of the ERP system via application programming interfaces (API) that are deployed in cloud. The primary goal of this thesis is to find out how APIs should be designed and maintained. Secondary goal is to come up with a design and management scheme for an API. The study suggests that the design of APIs is quite well studied topic in literature. On the contrary, API management has been studied slightly less. Based on literature, an API management framework is introduced, and API management is designed for a single API. The result of the study also includes an overview of API management and design for one API. Based on the study, it can be said that the introduced framework can be utilized in the design of an API management framework. However, the quality of the introduced designs of API and its management scheme are difficult to evaluate. The evaluation and implementation are left for a future study.

TABLE OF CONTENTS

LIST OF SYMBOLS AND ABBREVIATONS

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
DNS	Domain Name System
ERP	Enterprise Resource Planning
HATEOAS	Hypermedia As The Engine Of Application State
HTTP	Hypertext Transfer Protocol
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
MIME	Multipurpose Internet Mail Extensions
PDF	Portable Document Format
RAML	RESTful API Modeling Language
REST	Representational State Transfer
SAML	Security Assertion Markup Language
SLA	Service-Level Agreement
SOAP	Simple Object Access Protocol
SSL/TLS	Secure Sockets Layer/Transport Layer Security
URI	Uniform Resource Identifier
XML	eXtensible Markup Language

1 INTRODUCTION

Application programming interfaces (API) are one of the essential components in developing complex mobile applications. This is partly due to APIs providing the means for sophisticated communication between wireless mobile devices and back-end servers. The inclusion of APIs in an application may prolong the lifecycle of a software product and allows interaction with other applications which is often essential (Orenstein, 2000). In recent years, the explosive growth in the number of wireless devices, like smart phones, has led to similar growth in mobile application development. Consequently, the number of application programming interfaces published in the web has also grown as evidenced by API tracking sites like ProgrammableWeb. As of September 2019, the site tracks over 22 000 APIs when this number was close to 6 000 in 2012 (Garber, 2013). Another reason for the growing popularity of APIs is that many applications today utilize the services provided by others to build their own core functionality (Garber, 2013). For example, over a million web sites use Google Maps API to provide map functionality for visitors (De, 2017). One of the challenges regarding APIs is the management. That is, how to design, manage changes, automate testing and document the APIs among other activities. Improper management could reduce the benefits gained from utilizing APIs. One of the largest information and communication technology corporations in Finland, Tieto Corporation, is aiming to develop a mobile application for a long-standing enterprise resource planning system which has been in production use for over two decades. To establish communication between the mobile app and the ERP system, APIs should be developed and deployed in cloud. In this thesis, we will look at how to design APIs and API management in Tieto's case.

1.1 Background

The software system that is covered in this study (later referred to as "the ERP system" or "the system") is an enterprise resource planning (ERP) software managed and developed by Tieto Corporation for a global company operating in the forest industry. The system is specifically tailored for the needs of the customer and encloses many business functions such as order handling, production planning, logistics, invoicing and reporting. The first version was released already in 1996 and it has since then been in production use. During over two decades of operation the system has constantly received improvements and changes of differing size as releases and patches. The development effort is done in a distributed manner across continents by dozens of developers, testers and other staff members.

In 2018, the need for a mobile application on top of the system was identified. The mobile application will start off simple, with few functionalities, and grow larger and more diverse as the business needs show up. Couple use cases for the mobile application have already been proposed and identified as of December 2018. One of these is a relabeling application which allows storage workers working at paper reel warehouses to print out new labels for paper reels by scanning the barcode of existing labels using a smart phone. In order to work, the mobile application needs to exchange data with the ERP system and its services that contain business logic. Thus, there is a need to develop application programming interfaces for existing services and publish them online. The plan is to host a series of APIs on Microsoft Azure cloud platform to allow standardized access. The process flow would then be as follows: First, a barcode is scanned using barcode reader in a smartphone. Second, the mobile application calls Azure cloud API with the details from the barcode to request new set of data, for example, new mill order number and shipping date. Third, API processes the call and forwards it to the ERP service. Last, the ERP service returns data to the API which then returns it to the mobile application.

The objective is to move from complex service content to more simplified and more available format. Currently, services are called directly from the core of the system or via JavaScript object notation (JSON) interface. These service calls require non-standard calls when calls through Azure API with standardized technique and format would be preferred. As an example, one functionality of the mobile application requires three pieces of data from unit level ('unit' meaning paper reels) from the ERP system, but the unit level contains over one hundred fields. Is it better to build an API which returns three fields, or all of them? What happens when some other functionality requires five fields instead of three? The digital transformation to make ERP system services available as APIs will follow the basic flow introduced in "Cloud Customer Architecture for API Management" by Cloud Standards Customer Council (2017). As illustrated in figure 1, the transformation starts from an existing enterprise IT investment which in this case is the ERP system. Some of the services of the ERP system will be exposed as APIs in cloud after which developers can consume the APIs to develop the mobile app and eventually fulfill the use cases.

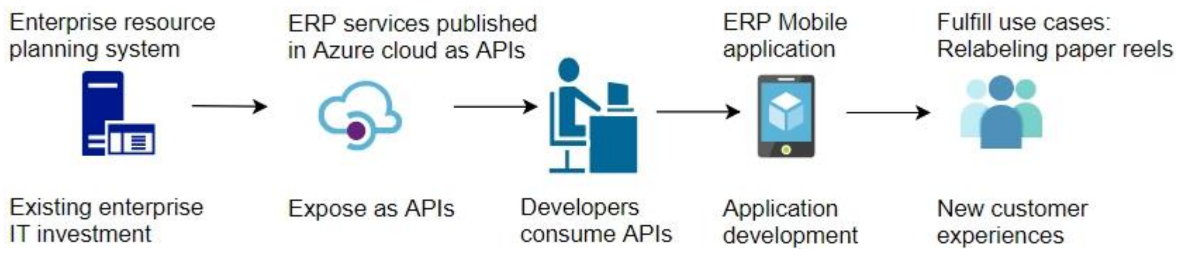

Enterprise resource planning system
ERP services published in Azure cloud as APIs
ERP Mobile application
Fulfill use cases: Relabeling paper reels
Existing enterprise IT investment
Expose as APIs
Developers consume APIs
Application development
New customer experiences

Figure 1. Enterprise digital transformation. Adapted from the white paper by Cloud Standards Customer Council (2017).

The ERP system contains over 5000 different business services many of which need to be published in cloud eventually. The problem is how to design and manage the abundance of APIs. Without proper upfront planning the APIs will be needlessly difficult and expensive to manage. The APIs need to be managed, tested and documented consistently. Also, the cost versus complexity viewpoint should be considered. Developing multiple APIs has the potential to be simpler than having one API that includes the interface to 5000 services. However, a single API has lower cost attached to it, but possibly with more complex management as a flip side. Some of the largest cloud providers offer their own API management solutions such as Microsoft Azure API Management and Amazon API Gateway. These 3rd party offerings have the drawback of their price and, in some cases, poor customizability. Thus, the development of an in-house API management solution should also be considered. In addition to the case specific challenges, some general web API related challenges have to be managed. Such challenges include service availability, latency and the quality of service (Wittern et al., 2017).

1.2 Goals and delimitations

The main goal of this study is to find out what literature says about API management and design, then build a framework to support the API management of ERP cloud entities. The framework should strike a balance between cost and complexity while also being grounded in literature. A literature review on cloud API offerings, API design and API management is conducted to support the framework. The theory part of the study is limited to APIs in the context of the ERP system. API security is briefly touched upon, but the focus is on API design and management including topics such as change management, automated testing and documentation. The study is limited to literature review and design. Implementation is out of the scope. The research questions that are answered in the context of the ERP system are:

1. How to design APIs?

2. How to manage APIs in practice so they stay up to date?
3. How to manage changes and new requests to APIs?

The outcome of the study includes three main artifacts: literature review, API management framework and design of a cloud API with API management. API management framework is introduced in the context of the ERP system and the mobile application which is built on top of the system. The design of cloud API and API management is an attempt at putting the framework to use by applying the guidelines and principles to application programming interfaces. The goal with the API design is to test the suitability of the management framework and set an example of a design. The development and implementation of the APIs and the API management platform are not done in this study, but rather conducted later as regular development activity within the company.

1.3 Structure of the thesis

This thesis is divided into theory and design sections. In chapter 2 the literature surrounding the topic of the study is reviewed. In chapter 3, the API management framework is introduced. In chapter 4, the design for an example API, the UNIT API, is discussed. Additionally, the design for Microsoft Azure API management for the UNIT API is introduced. The results of the study are presented in chapter 5. Lastly, discussion about future research and conclusions are in chapter 6.

2 LITERATURE

Literature review is conducted to get information on API design and management. Information on these topics is needed to establish a basis on which to ground any design decisions and conclusions that are made in the second part of this thesis. Thus, many of the arguments made for or against some solution concerning API management framework or the design of API management can be justified using the findings in literature review. In the first part of the literature review, the focus is on API design best practices. Second part of the literature review is centered on the management of APIs. In the context of this thesis, API management includes change management, testing, documentation and general API lifecycle management. Basic knowledge on the topic of APIs is assumed from the reader.

Material for the literature review is primarily retrieved from Lappeenranta Academic Library collection and the international e-materials collection (including databases of ACM, IEEE and others) that the library provides for university staff and students. Some material is also retrieved from other research databases such as Google Scholar. The literature review process starts by coming up with relevant search terms for the topics. Search terms are made accurate enough so that search results would not return hundreds or thousands of records. For example, instead of simply searching for “API” the search is done using for example “API automated testing” to limit the results. Search queries are first run on the Lappeenranta Academic Library collection which is the smaller of the two previously introduced collections and primarily contains books, some of which are interesting from the thesis’ point of view. The search results are browsed through, and any interesting titles are opened for the abstracts to be skimmed in more detail. Also, the references of the interesting search results are examined to find further material. The search for material continues like this in a tree-like structure, skimming through the documents and its references until the topic changes or same references start to appear time and again. Same process is repeated on the international collection. The literature review material gathering process can be halted when each of the topics is covered to a satisfactory extent, same references start to pop up and the rate of new material emerging begins to noticeably slow down.

Based on the literature search process it can be said that both topics have a well-established collection of material. It seems that API design has been studied more extensively than API management. Design guides and principles for the topic of application programming interfaces can be found from both independent and dependent sources. Best practices and tools for management are readily available, although software and material published by cloud vendors often has a price tag attached to them.

2.1 Application programming interfaces

In this section, APIs are examined further. The focus is on design and management of APIs, especially APIs in the cloud. Additionally, some existing solutions for design and management problems are discussed. The largest cloud providers have published their own tools and methodologies for implementing API management and design. Also, many kinds of third-party software are available for API management purposes. APIs can be private or public. They can also include often used programming frameworks such as Windows .NET (Stylos & Myers, 2016). This section will focus on private APIs which are the type of APIs that will be developed in Tieto in this case.

According to Garber (2013), APIs are on the rise and this is due to API technology being one of the enablers for cloud computing and mobile technologies. Garber states that APIs play an important role in integration between different applications and devices. For example, the sharing of photos between platforms is enabled by APIs. In particular, Garber emphasizes the fact that APIs are the easiest way to enable connection between mobile applications and back-end applications. The typical features of mobile devices should be considered when designing APIs. Limited computing power, limited storage and limited battery mean that achieving better performance requires minimizing the number of API calls and optimizing the data content of the calls (Garber, 2013).

In literature the definitions for application programming interface have some variance. For example, Stylos & Myers in their article "Mapping the Space of API Design Decisions" (2007, p. 51) define an API as "a collection of existing code that other programmers can call". Furthermore, they suggest that API calls are often used to achieve programming goals, such as retrieving data from other systems. Finally, Stylos & Myers (2007) note that often the definitions between seemingly similar entities such as libraries, frameworks, development kits, toolkits and APIs are used interchangeably, and they can all be seen as variations of API as the differences between the four are somewhat small. Orenstein has perhaps a more metalevel approach when he defines API in his article "Application Programming Interface" (2000, p. 66) as "a description of the way one piece of software asks another program to perform a service". Orenstein emphasizes the fact that APIs are needed everywhere, in operating systems as well as in banking. In their book "API-talous 101" (2018), Moilanen et al. have a manifold definition of API. They see APIs mainly from the business point of view as services that are offered to customers. According to them, an API can be a part of a physical product, a product in itself, or a part of a digital product. API can also be an interface to a resource, to an underlying physical device or to a software system, such as an ERP. All in all, the definition of application programming interface is quite well established in the literature. Different authors have published definitions that are slightly different from each other and have varying emphasis but share the same core idea of APIs.

The interaction process with web APIs is often the same. Developers create applications which make calls to web APIs utilizing HTTP, TCP and IP in application, transport and network layers respectively. The API exposes a set of resources which are identified by their URLs. These resources describe the interaction scheme. The semantics, on the other hand, are depending on HTTP methods. In other words, the result of the call depends on the HTTP verb that was used. The connection to the web API server is established and afterwards the data transfer can begin. (Bermbach & Wittern, 2019)

According to Stylos & Myers (2007), there are four main reasons as to why APIs should be implemented and utilized. Firstly, APIs save development time by allowing the reuse of code. Secondly, an API is very good at hiding information which enables the possibility to modify the implementation of an application while still preserving functional API calls. Thirdly, due to the ability to reuse code, same APIs can be used in multiple applications providing end users with similar user experience across platforms and avoiding the need for re-learning. For example, text editor software can have similar appearance and functionality in a web-based forum and intranet support ticket application. Thus, user can use both applications efficiently by learning only one of them. Lastly, APIs can establish otherwise difficult to create connections between applications. In this thesis' case, establishing a connection between mobile application and 1990's ERP software system is not necessarily the easiest of tasks without the leverage provided by API technology. Security is established by using well-known security approaches such as SSL/TLS, OAuth and OpenID Connect.

Garber (2013) continues to talk about the advantages of APIs by stating that utilization of APIs in cloud can be used to publish distributed resources online. Additionally, APIs provide developers with flexibility and security. Flexibility can be seen in the fact that new functionality can be made available by simply developing new API or updating an old interface. (Garber, 2013) Furthermore, a stack of APIs is a prime target for generating analytics because all communications between the mobile application and the ERP system go through the APIs.

API design solutions and recommendations have been published by many authors. The most relevant recommendations for this thesis are examined next in chapter 2.2 API design. Some of the recommendations are more platform specific and focus on implementation like Bob Familiar's book "Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions" (2015), while others have put out more general and platform independent recommendations on API design, for example Joshua Bloch's presentation "How to Design a Good API and Why it Matters" (2006).

2.2 API design

The design of APIs is discussed next from the best practices point of view. The level of discussion is intentionally kept general. Pedantic code examples and fine-grained details are omitted to keep the design chapter compact. Also, as noted by Stylos & Myers (2007) the space of API design decisions is huge. Considering the purpose of this thesis, listing everything is neither feasible nor reasonable. To limit the size of the chapter, the goal is to give the reader some general guidelines and best practices on how to design APIs.

As noted by Maleshkova et al. (2010), the current web APIs are quite heterogenous and standards for API development are not yet matured meaning that it is difficult to define the properties of a typical web API. Thus, only guidelines for API design can be given. A list of best practices collected by Petrillo et al. (2016) can be found in appendix 1. For step by step implementation of APIs, more thorough guides such as "Practical API Design: Confessions of a Java Framework Architect" by Tulach (2008) or "ASP.NET Web API 2: Building a REST Service from Start to Finish" by Kurtz & Wortman (2014) should be read. API usability, in particular, has been studied quite extensively. Also, the major cloud providers, Microsoft, Google and Amazon, have published design guides for APIs.

To start designing an API it is important to consider the stakeholders and their needs. Stylos & Myers (2007) identify API designers, API users and consumers of products as the most important stakeholders. API designers are mostly concerned with the adoption of the APIs and minimizing support and development costs. API users, on the other hand, develop new software products that consume APIs. Their main goal is to develop software fast and efficiently which requires certain attributes from the APIs that are being consumed. Finally, consumers of the products want the software to include needed features without bugs. The design of APIs should consider the needs of all stakeholders. Higher impact on stakeholders means more important design decision. Different decisions can be important in different ways. Some decisions are made frequently while others are only made once in the lifecycle of an API. Also, some decisions are hard to make and often end up far from optimal while others are easier to make and have existing recommendations to help. High-level quality attributes that are important for APIs are usability and power. Going into detail, usability consists of learnability, productivity, error prevention, simplicity, consistency and the matching between APIs and users' mental models. Power consists of expressiveness, extensibility, evolvability, performance and robustness. These are all quality attributes that affect the stakeholders, and thus should be considered in the design of APIs. Ultimately, these attributes together create a good API. Some points of controversy in API design are the handling of exceptions and naming of different API versions. (Stylos & Myers, 2007)

Patni (2017) describes four strategies for designing APIs: Bolt-on, Greenfield, agile and façade. Bolt-on strategy assumes an existing application such as the ERP system in this thesis' context. The API will be exploiting existing back-end and its services. In this strategy, API and mobile application are separate from the main product. Greenfield strategy follows the mobile-first ideology where application is built from scratch, i.e. there is no pre-existing application as in the bolt-on strategy. Agile strategy starts building the API without full specifications. Façade strategy utilizes existing systems but also shapes them according to needs. (Patni, 2017) Of these strategies, the most relevant is the bolt-on strategy. Tieto has an old software product with robust set of back-end services. The mobile API project is not meant to touch the existing product, but rather create something new on top of it.

According to Tulach (2008), some of the important quality attributes of APIs are consistency and discoverability. Application developers that utilize APIs in their applications should only learn one API to understand the concept that is used in the others. Consistency should also be applied in the development of APIs. The style of development should not change with time to keep the APIs consistent. APIs should be developed considering the use cases but keeping the APIs simple and clean with only the minimum required functionality. Exposing too much functionality is risky as removing parts of API is harder than adding new functionality in the future. All functionality that is added should be justified with use cases to determine if they are needed. Any unnecessary features always contribute to the effort of keeping the API backwards compatible. It is also advised to trim out any unnecessary features prior to a release. (Tulach, 2008) Consistency of APIs is underlined by Patni (2017), as he argues that consistent data structures, data representations, resource identifiers and error messages are very helpful for developers. Especially, in cases where API orchestration is implemented, in other words, the output of one API is the input of another API.

In their research, Stylos & Myers (2008; 2016) demonstrate the impact of design on discoverability, learnability, efficiency and correctness. They argue that calling the function of the "main" class is better than calling the function of helper class. For example, email sending function is more discoverable when the call is mailMessage.send(mailServer) instead of mailServer.send(mailMessage). The threefold conclusion is that all programmers tend to find the same classes when looking for APIs, programmers search for classes by looking at the classes that the methods of another already discovered class refers to and finding needed classes is faster when the classes that they first find refer to other useful classes (Stylos & Myers, 2008). Some consideration needs to be paid to potential drawbacks of placing methods to the main class as sometimes it's the case that this practice inhibits, for example, information hiding and reuse (Stylos & Myers, 2008).

API design should consider Nielsen's usability heuristics (Stylos & Myers, 2016; Molich & Nielsen, 1990). Applying Nielsen's heuristics improves the overall usability of the API. Thus, application developers are more likely to generate higher quality code. Stylos & Myers (2016) map the heuristics to APIs as depicted in table 1.

Table 1. Some of Nielsen's heuristics mapped to API context (Stylos & Myers, 2016)

Heuristic	**Mapping to API context**
Match between system and real world	Naming conventions of APIs should match with users' expectations. I.e. "natural programming" should be applied.
Consistency	API should be thoroughly consistent. This includes but is not limited to naming, design and documentation.
Error prevention	API should provide guidance on the usage of the API to the user.
Minimize the user's memory load	API should offer autocomplete popups for users when they start typing a method name in integrated development environment (IDE)
Provide good error messages	API should provide meaningful errors that the user can interpret and apply in practice to make corrections.

Doglio (2015) lists developer friendliness, extensibility, real-time documentation, well implemented error handling, multiple libraries, security and scalability as some of the features of a good API. User friendliness can be achieved by simplifying the API, so it is easy to use. However, one should be wary of oversimplification which may lead into other problems. Other factors that improve developer experience are the choice of communication protocol, transport language and access points that are easy to remember. A safe bet for the choice of communication protocol is HTTP as it is very well known and used everywhere. JSON should be selected as the transport language due to it being lightweight, human readable and supporting many data types. Extensibility can be understood as the ability to add something new to the existing API. When an API is easy to extend the changes are backward-compatible and new endpoints can be added easily. Extensibility is also concerned with the clients' ability to use the new version of the API. (Doglio, 2015)

Bloch (2006) agrees with Tulach in the fact that API design should begin with gathering requirements using use cases. The specification doesn't need to be longer than one page and it should be circulated for feedback. The API definition should specify what functionality will be offered, where the API can be accessed, what parameters should be used and additionally any SLAs and other legal and performance related agreements (De, 2017).

According to Henning (2007), the documentation of the API should be written before actual implementation to avoid situations where the implementer only writes down what he has done and not the actual documentation. Writing of the API should begin early in the development process to avoid throwing away multiple implementations and specs. The design of the API should follow the general principles found in table 2. Technical requirements are a part of API design. Some thought should be given to, for example, API specification (Swagger etc.) and API versioning strategy (De, 2017).

Table 2. API design main principles (Bloch, 2006; Cwalina & Abrams, 2005; Stylos & Myers, 2008; Moilanen et al., 2018)

Principle	**Description**
Focus on one thing	API should have one clearly defined function which it does well. The function should be easy to explain and have a descriptive name.
Size	API should be big enough to satisfy the requirements but have no extra functionality. Adding functionality later is easier than removing existing.
Implementation	Implementation details should be hidden from product users and API consumers. The details can confuse users.
Accessibility	Maximize information hiding to facilitate modules to be implemented and tested independently.
Naming	Try to create self-explanatory names and avoid hard-to-understand abbreviations. Be consistent with naming across all APIs. Naming can be considered very important for usability of APIs. The best names should be reserved for the most important APIs. Also, names that start early in the alphabet should be preferred.
Documentation	Document everything to facilitate reuse and maintenance. Keep documentation up-to-date with code samples of typical use cases. Documentation should include machine-readable specification such as Open API.
Performance	Be wary of design decisions that will lower performance. For example, mutable data types will cause excessive object allocations which will add up.

In his book “Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions” (2015), Familiar recommends implementing microservices application on Azure platform. One part of the application implementation is the APIs that are called. Familiar suggests utilizing the out-of-the-box API management feature of Microsoft Azure cloud platform for managing APIs. He also gives more elaborate recommendations such as clients should never call APIs directly, but rather an API gateway, that handles all API calls, should be used. When it comes to API protocol and communication, using REST over HTTP with JSON is the way to go as this is the most used combination today (Familiar, 2015; Garber, 2013; De, 2017). JSON should be preferred over XML due to being more lightweight, easier to read and easier to parse (De, 2017). According to Garber (2013), RESTful APIs are simpler than legacy SOAP APIs which makes developing APIs accessible for more developers. The qualities of REST as an architectural style include minimal latency, minimal network communication, maximal independence and scalability of components (Fielding, 2000; Fielding & Taylor, 2002) all of which are important for the operation of the ERP system.

Moilanen et al. (2018) agree on the fact that simplicity is the key to success. According to them, application developers don’t usually have time to familiarize themselves with a single API as they may be working with dozens of other APIs simultaneously. APIs should therefore be easy to understand and deploy even without reading through the manual. Key quality attributes in API design are intuitiveness, trustworthiness and ease of deployment. Suggested methodology for the design and development of APIs is agile. In the beginning of API lifecycle, it’s important to define the need: Why is this API needed and what customer need does it fulfill? User stories are a great way to go about this. When creating user stories, one should give answers to following questions: How does the API create value for customer? Who are the customers and how do they discover the API? How will the API be used? API definitions can be created, for example, using the Open API specification which has the added benefit of automatic conversion from specification to code. Second phase of API lifecycle is development and testing. Minimum viable product should be created to gather feedback and ensure that the product fulfills customer needs. The structure of the API should be solidified as changing the structure will easily break any existing applications that use the API. This means that the Open API specification should be ready and published.

Third, and last, phase of API lifecycle is maintenance. Maintenance is cheaper and easier if the implemented APIs are standardized and uniform. (Moilanen et al., 2018)

According to De (2017), the design of a RESTful API should consider the REST principles. Careful consideration should be paid to the modelling and hierarchy of resources. Resources should not be too fine-grained but not too coarse-grained either. The granularity should strike a balance to avoid degraded performance or difficulties in maintenance. (De, 2017) The resources could be for example customers and reservations which have a clear hierarchy where customers make the reservations. Thus, reservations made by a customer can be found via following uniform resource identifier (URI): /customers/{customer ID}/reservations. De (2017) also advices to avoid bringing business logic to APIs. In other words, the client should not have to worry about following the correct flow of business logic. Mixing business logic in API will make all subsequent logic changes more difficult due to the requirement of having to change the logic both in API and in back-end system (De, 2017). This might not be as big of a problem in cases where the client (mobile application) and back-end system are maintained by the same company (e.g. Tieto Corporation). De (2017) goes on to add that also business processes can be modelled as resources which can be a useful way of avoiding the inclusion of business logic in the APIs. For example, the reservation process, i.e. making a reservation, can be modelled as a resource such as /reservationSetup. In addition, resource URI paths should be predictable and have maximum depth of three, if at all feasible (De, 2017). Useful naming conventions for URIs can be found in table 3.

Table 3. URI path naming best practices (De, 2017)

Convention	**Example**
Plural nouns should be used to name collections.	/api/**customers**
Singular nouns should be used to name single resources	/api/customers/**customer**
Verbs should be used to name controller resources	/api/customers/customer/**reserve**
CRUD operation names should not be used	Avoid: /api/**getreservations**
Lower case letters should be used in names	Avoid: /api/**hotelCustomers**
Hyphens should be used in place of spaces or underscores	/api/**hotel-customers**

Characters that require URL encoding should not be used (e.g. space character is encoded as %20)	Avoid: /api/**hotel customers**

Instead of traditional XML content and single API entry point, APIs should be designed to include qualities mentioned in the Richardson maturity model. The model includes three core concepts: Resources, HTTP verbs and hypermedia. The base level simply entails a single URI through which all methods of the API are called. The first level adds unique URIs for each method. HTTP verbs are not fully utilized in the first level. Next step in the maturity model includes the utilization of HTTP verbs. HTTP verbs should be used based on their attributes, such as safety and idempotency. Traditionally, only POST is used, but the maturity model calls for the utilization of other verbs. For example, GET is a safe verb which can always be used to retrieve a representation of a resource. Finally, hypermedia controls (also known as HATEOAS), such as links and forms, are the peak of the maturity model. They enable the client to decide on the next action by providing a list of links on available actions. Thus, hypermedia controls erase the need for guessing the next action for the client. If the user has no idea what to do he can use the safe verb GET to retrieve the representation of the base URL. GET does no harm so user can be assured that he will not break or disrupt the back-end system. If the API is on level three, the user client can consequently scan the representation for hypermedia which can be used to find the next resources. The purpose of hypermedia controls is to have the client only remember the root address of the website and be able to navigate the API from thereon. (De, 2017; Richardson et al., 2013; Kurtz & Wortman, 2014)

In the context of the ERP system, resources can be interpreted as enabling interaction with each individual unit. For example, getting the information of a single unit instead of a generic "getUnits" which would return all units. Different HTTP verbs should be used in suitable scenarios. GET could be used to get the units details instead of POST as GET is safe and read-only. Hypermedia controls could be used in the API to offer client the next steps after fetching unit data. For example, links to removing unit or changing some of its details. Also, the concept of idempotency should be considered when choosing which verbs to use. Idempotent verbs have the same effect regardless of whether they are used once or multiple times (Richardson et al., 2013). For example, making a call to a specific resource using

DELETE will not have any extra effects even if the call is made 100 times. Verb safety and idempotency help prevent errors caused by connection timeouts.

Palma et al. (2014) describe design patterns and antipatterns regarding RESTful APIs. Design patterns are good practices in API design whereas antipatterns are the opposite. The abandonment of hypermedia controls is one such antipattern. It implies the absence of hypermedia controls in resource representations. This practice makes it difficult for clients to navigate the API and transfer between states. Other antipatterns are breaking self-descriptiveness, ignoring caching, ignoring MIME types, ignoring status codes, misusing cookies, tunneling through GET and tunneling through POST. Design patterns include content negotiation, end-point redirection, entity linking, entity endpoint and response caching. Antipatterns and design patterns are further explained in table 4.

Table 4. REST API antipatterns and design patterns (Palma et al., 2014)

Pattern (D=Design pattern, A=Antipattern)	**Description**
Content negotiation (D)	Clients have the ability to request resources in different standard content types that can include JSON, XML, PDF and others.
End-point redirection (D)	When redirection status code is returned the server redirects the client to a new location. Redirection keeps the API service functional even after a bad call by client.
Entity linking (D)	The use of hypermedia controls in resource representations.
Entity endpoint (D)	APIs should have endpoints that are defined with at least to identifiers. The first identifier is a global name for the API and the second identifier identifies a resource inside the API.
Response caching (D)	API responses are stored in client's local cache which prevents sending same API calls multiple times.
Breaking self-descriptiveness (A)	API uses custom-made headers, formats or protocols which disables the self-descriptiveness of the API response resulting in reduced reusability and adaptability.
Forgetting hypermedia (A)	API doesn't utilize hypermedia controls in resource representations.

Ignoring caching (A)	API responses are not stored in client's local cache. This forces the client to always call the API instead of reading from the cache.
Ignoring MIME types (A)	Only one of the standardized resource representation formats are supported or even a custom format is used.
Ignoring status code (A)	API only uses very limited set of HTTP status codes (typically 200, 404 and 500). Using all the available codes would benefit the client.
Misusing cookies (A)	REST calls for statelessness. Thus, using cookies to retain session state is not a good practice and goes against the REST principles.
Tunneling through GET (A)	GET should be used only for its intended purpose which is retrieving a representation of a resource. Sometimes API calls use GET for everything including deletion and updating of resources.
Tunneling through POST (A)	POST should be used for creating a resource. Sometimes API calls use POST for everything including deletion and updating of resources.

Changes to an existing API can lead to situations where the backward compatibility is lost, and connected applications stop working. Versioning can be a solution to this problem. Radical changes to APIs are handled so that current version of the API stays untouched and a new version is created for the changes. The circumstances in which a new version should be created should be documented, for example, in the style guide. (Moilanen et al., 2018) APIs are bound to need changing at some point of their lifecycle and sometimes compatibility breaks no matter what. Early warnings of depreciation should be issued whenever possible. (Tulach, 2008) The issue is underlined by Macvean et al. (2016) when they argue that even small changes to API can cause interruption in operation of a software as the software is dependent on a specific version of the API. Patni (2017) suggests maintaining at least one historical version of the API.

Open API is an API specification which describes interface that is implemented based on REST architecture. In the recent years, it has become the industry standard. It supports describing, producing, consuming and visualizing the structure of an interface while also being machine-readable. The organization behind Open API specification includes large tech companies like IBM, Google, Microsoft and Oracle among others. Open API has the benefit of open source tool pack which can be used to automatically produce code and minimum viable products. Open source tools include Swagger Editor, Swagger Codegen, Swagger UI and Swagger Inspector. The tools can be used to design, describe, document, generate code, visualize and test. The organization behind Swagger and Open API also produces learning material such as the best practices in API design. Open API specification is only a framework and allows quite a lot of freedom for developers. Therefore, it is suggested to create a style guide which defines how to name parameters and API versions, how to implement error handling and how to enforce information security. API style guide has multiple benefits including easier maintenance of code and lesser workload for application developers. (Moilanen et al., 2018; Pinkham, 2017) Neglecting the documentation using a machine-readable specification, such as Open API or RAML, has its consequences. For example, refactoring and automatic testing can be hindered. Without machine-readable specification, application developers are solely depending on the human-readable documentation which results in more manual labor. API management solutions are also often relying on the machine-readable specification. (Wittern et al., 2017)

Many times, API calls return much more data than what is needed. For example, a call to get customer data will likely return all the available data, such as address, name, occupation, account balance, age and so on. Creating own API call for each piece of data might also not be a reasonable solution due to the extra overhead that is generated. Patni (2017) introduces the idea of partial responses which is already implemented by some large companies such as Facebook or Google. Partial response allows the API caller to define exactly what data is needed. This can be done, for example, via an optional parameter: "?fields=age,name,occupation". Doglio (2015) recommends optional parameters such as 'page', 'perpage' and 'sort' which can be used to define page number, number of items per page and sorting order of items in calls which will potentially return a large result set.

Varga (2016) calls for information hiding in the API design. Often, the callers of the API are either clueless or careless, sometimes even both. In these cases, it is important to shield the API and the back-end against calls that lead to adverse changes in the system. For example, exposing some field directly to API users can have negative consequences when a clueless user accidently changes the value without noticing. The internals of the back-end system and the API should be hidden to prevent unintended changes. The interface segregation principle can also be applied in API design. The principle states that you shouldn't create too generalized interfaces (or APIs in this case) but rather group similar methods in one interface (Dooley, 2011). API callers should not depend on APIs that they do not need. This entails creating multiple APIs even for single use case to improve maintainability, usability and reusability. (Varga, 2016) From the ERP system's point of view, this means that API methods should be somehow grouped together into separate APIs. Grouping could be done based on ERP areas such as logistics and invoicing. If a use case needs functionality from both then it calls both APIs. When some other use case needs functionality only from logistics side, it is not dependent on methods or APIs that it doesn't need.

In his paper "API design matters", Henning (2007) lists a set of guidelines to help improve API design. API should use as few types, functions and parameters as possible to facilitate learnability and correct usage. For example, C++ standard string class has over 100 member functions which makes learnability difficult. Same goes for Unix which includes five differently named variants for wait function. User must read the manual to understand the difference and use the correct function. API design should consider the context where the API will be used. Context knowledge is needed to determine what values are expected to be returned. Depending on the context an API call which returns nothing should sometimes return an exception and other times null or an empty string. It is the API designer's responsibility to recognize the needs necessitated by the context. Based on the context, the API designer should also impose policies. For example, the context of the API can require a policy of returning an exception on API call which contains invalid order number. The API user has to then expect exception and interpret it correctly. If the user is expecting null or an empty string, the application will malfunction. Policies are often a must as without them the parameter lists become needlessly long and reduce usability. API design should also consider the API caller's perspective. The caller often sees the API as a black box having no clue what's inside. Thus, it is often helpful for the callers when methods' parameter lists are descriptive. Instead of a method with three booleans as parameters the parameter list should have descriptive names defined using e.g. enum definitions. It is a good idea to let the users define the API call and afterwards let the developers implement the internals based on the call. A good API has a clear definition of what can be achieved using it and what cannot be done. (Henning, 2007)

Information security is an important aspect of API design. Deficient security may lead to data breaches through SQL injection or cloud log files that collect data on URL parameters. API information security checklist is provided by The Open Web Application Security Project and it can be used to check a design for any information security issues. For example, the REST security cheat sheet available in GitHub includes multiple check items which can be used to validate the security of an API. Moilanen et al. (2018) recommend not to include sensitive data in URL parameters, validating inputs and sanitizing outputs to prevent injection attacks. They also recommend avoiding one-to-one connection between URL parameters and database identifiers. (Moilanen et al., 2018; Righetto et al., 2019) Information security aspects of API design are not discussed more in this thesis, as the subject contains so much more information and could have its own thesis. The goal is to create a design and the security aspects are not yet relevant and should be a subject of later research that improves on the existing API.

Finally, many large API providers, such as Microsoft and Google, offer their API design guideline documents for public. They can be found from http://apistylebook.com/design/guidelines/.

2.3 API management

API management is discussed next from a general point of view. The structure of API management is examined: What does API management consist of? Also, we will summarize the solutions provided by the largest cloud vendors that are Microsoft Azure, Amazon AWS, and Google Cloud according to “State of the could report” by Flexera (2019). The goal is to get a general understanding of API management content and practices.

API management is a challenge for API developers. The APIs connect mobile application and back-end services together and often the back-end services have not been designed to handle the traffic introduced by the mobile application. As a solution, some kind of layer between the API caller and the back-end service should be established to control the traffic. (Garber, 2013) Such layer could be the API gateway component. De (2017) says that API gateway can be argued to be the single most important part of API management. According to him, its task is very diverse: It facilitates API security through authentication, authorization, identity mediation, data privacy, certificate management, denial-of-service attack protection and threat detection. API gateway also deals with traffic management through consumption limiting, spike arrests, usage throttling and traffic prioritization. Finally, API gateway includes interface translation (format, protocol and service translation), caching, service routing and service orchestration (De, 2017). Additionally, API portal or developer portal is a key feature to successful API management (APIacademy, 2015). The tasks of API gateway are further examined in table 5.

Table 5. API gateway features (De, 2017)

Main feature	**Sub-feature**	**Description**
Security	Authentication	Identifying and validating a client based on unique identifiers such as application keys.
	Authorization	Controls the access of specific applications identified by authentication to specific APIs.
	Identity mediation	Provide integration between different identity protocols. Commonly there is a need to integrate between OAuth used by API and SAML used by back-end services.
	Data privacy	Encryption of data in API calls.
	Key and certificate management	Management of keys and certificates that are used to encrypt data in API calls.
	Denial-of-service attack protection	Detect denial-of-service attacks and prevent their impact on the system.
	Threat detection	Detect malicious content inside API calls.
Traffic management	Consumption limiting	Limit the allowed API calls of a client. E.g. single client cannot bombard the API gateway with thousands of calls in a minute.
	Spike arrest	Related to denial-of-service attack protection. Detect spikes in incoming API calls and act accordingly.

	Usage throttling	Add some delay between subsequent calls. Can be useful if there are clear peeks in the calls during different times of day or week.
	Traffic prioritization	Specific types of API calls or clients can be given priority over others.
Interface translation	Format translation	Provide translation between data formats. Some commonly used formats are XML, JSON and CSV and sometimes there is a need to translate one into another.
	Protocol translation	Provide translation between protocols. Some commonly used protocols are SOAP and REST.
	Service and data mapping	Provide representation of back-end systems and how they map to APIs.
Caching	Caching	Store static replies in memory to increase performance.
Service routing	URL mapping	Provide mapping between URL in the API call and actual URL in the back-end system.
	Service dispatching	Provides ability to call correct back-end service based on incoming API call.
	Connection pooling	Manage the connections to back-end system.
	Load balancing	Balance the load imposed by incoming API calls.
Service orchestration	Service orchestration	Provide ability to call multiple APIs in the correct order to return an aggregated result.

According to Garber (2013), API frameworks are an emerging trend. They help developers in writing APIs but limit the flexibility. The advantage is that developers can be convinced that the solution will work as it has been proven beforehand. Therefore, the developers of APIs do not need to make architectural decisions, but they can focus on developing the APIs using predefined framework. Garber notes that today’s applications often require multiple APIs for different functions. In the ERP system’s case this could mean separate APIs for emailing, reporting and message services inside invoicing application. Garber points out the difference between maintaining a single individual API versus maintaining a whole cluster of APIs. The maintenance of an individual API can be easy, but in many cases one API is not enough. Garber talks about a new trend in which multiple APIs are aggregated into a stack of APIs. The stack of APIs provides developers with one interface, thus facilitating the maintenance efforts.

According to the white paper "Cloud Customer Architecture for API Management" by Cloud Standards Customer Council (2017), a good API management platform covers the whole lifecycle of an API. The lifecycle includes for example creation, deployment and management. Next, a general view on API management platform components and architecture is discussed, and finally a view on how MS Azure API management, AWS API Gateway and Google Cloud Endpoints implement discussed features and architecture. The API management platform should include a manifold of features and tools listed in more detail in table 6. The main functionality of an API management platform can be divided into four: creating, running, managing and securing.

Table 6. The main features of an API management platform (Cloud Standards Customer Council, 2017; De, 2017).

Category	**Details**
Automated tools for creating APIs	Include automated tools for designing, modeling, development, testing and deployment of APIs.
API management tools	Include tools for monitoring and managing performance, stability, scalability, load balancing, bandwidth priority control and failovers.
Language support	Different use cases may require using different languages. At least support for Node.js and Java are recommended.
API governance	Include tools for version management, packaging, cataloging and lifecycle tracking.
Access control tools	Include tools to manage access to the APIs, runtime analytics to detect intrusion, traffic management and data privacy.

Figure 2 shows the components of an API management platform. Edge services between public network and cloud include components, such as DNS servers and firewalls, that enable connection and the transportation of content between the two networks. API developer tools contain features which allow API developers to model, create, debug, test, build, deploy and publish APIs. API developer tools also facilitate naming and versioning of APIs. API gateway acts as a gate between API calls and the back-end services or the API runtime component. API gateway contains a set of policies that will be put into effect when API calls arrive. As a result, API gateway controls the traffic and priority of incoming API calls (for more API gateway features, see table 5). A secondary function for the API gateway is that it collects analytics data and monitors the traffic. Analytics data is sent to API management and finally to API analytics visualization where different stakeholders can see the visualization. (Cloud Standards Customer Council, 2017)

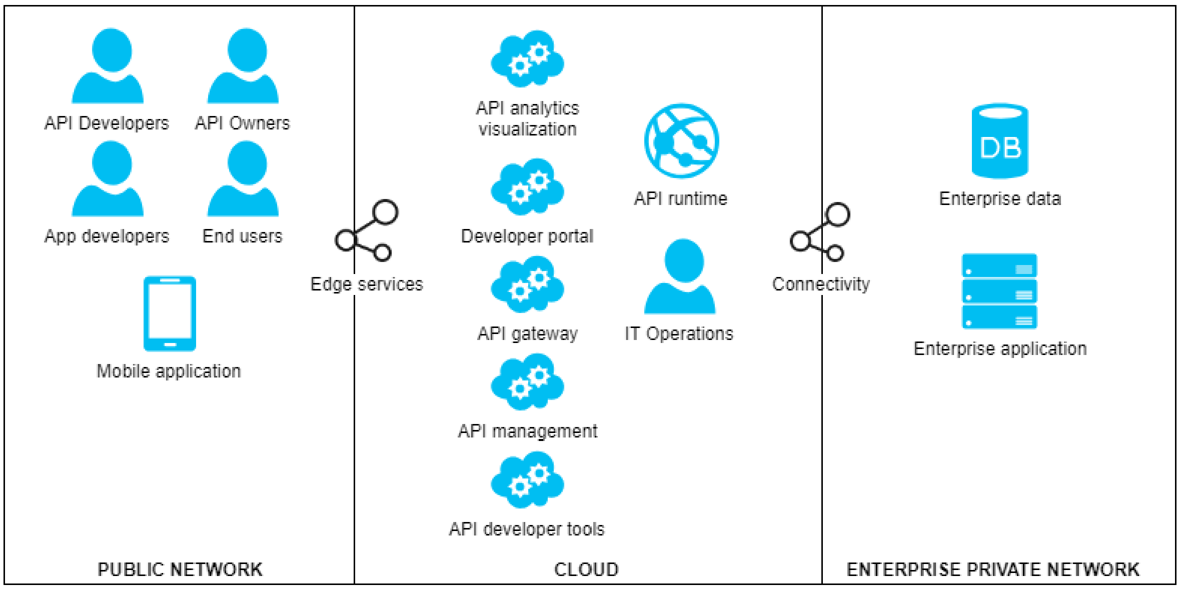


Figure 2. Components of the API management platform visualized. Figure is adapted from Cloud Standards Customer Council (2017).

API runtime is used to run the actual business logic based on incoming API calls. API runtime is not always necessary as the API calls can be directed straight to enterprise application. However, the API runtime has some benefits, mainly the ability to scale based on workload and support for multiple languages. API management component's core function is to catalog, package, version and publish APIs. It can, for example, create groups or stacks of related APIs which application developers can then subscribe to. This is especially useful when there are a lot of APIs and maintaining them is difficult. API management also takes in API metrics, analytics and monitoring data and forwards them to API visualization component. Developer portal shows all the published APIs to the developers and they can subscribe to the necessary ones. The developer portal helps developers to discover APIs and provides self-service which can expedite development. Most important use cases from the thesis' point of view are listed in table 7. (Cloud Standards Customer Council, 2017) According to Moilanen et al. (2018), developer portal typically offers access to APIs for application developers in only couple steps. These steps can, for example, be registering to the portal, choosing an API for testing and finally connecting the API to actual production data. API developer portal can be built in-house but doing so naturally entails a huge development overhead (APIacademy, 2015a). The benefit of in-house solution is the customizability. Another option is to use third-party solutions some of which are highly customizable. Often organizations have very different API initiatives with highly differing requirements. Thus, some organizations choose to develop an in-house API developer portal which suits their needs exactly. (APIacademy, 2015b) Hosting API management solution in cloud instead of on premises can have the benefit of better scalability, reduced management, easier and faster updates, and finally, reduced costs (CITO Research, 2015). Generally, deploying APIs in cloud has the following benefits over on-premises deployment: Reliability, availability, time to market, lower capital and operational costs, reduced management, scalability and agility. Cloud also has some negatives such as higher network latency and less control over data. (De, 2017)

Table 7. Some of the use cases in API management platform (Cloud Standards Council, 2017).

Title	**Actors**	**Description**
Develop a new API	API developer	API developer signs in to the API management platform. He accesses API developer tools to create a new API. He tests the API and finally deploys it to production environment. Finally, he publishes the API using the API management component, so others can subscribe to the API.

Discover and subscribe	App developer	Application developer signs in to the API management platform. He accesses the developer portal and searches for specific API. He finds the correct API and subscribes to it.
Use API in an application	App developer, end users	Application developer has subscribed to an API which he calls in a new application that he has developed. He deploys the application for end user devices. End users start using the new application.
Use an application	End users	End user opens an application on his device. The application makes an API call which is handled by the API gateway. The API gateway applies all the predefined policies to the API call and invokes the API. The API gateway finally forwards the returning API call to the end user device.

Often it is useful to have different environments for different phases of API lifecycle. For example, testing environment and production environment. Depending on the level of isolation required between these environments they can be implemented using completely separate instances of API management platforms or simply separate API catalogs inside a single management platform.

The goal of versioning is to allow changes in APIs with minimal impact to API consumers. Changes need to be controlled and predictable which can often be achieved by using some API version management scheme. API versioning is somewhat different from software versioning. New API version should generally be created in cases where the service interface is being changed. Creating new versions should be avoided in other cases, e.g. adding a small change to the API. An API is a contract between API developer and API consumer. API developers promise to offer services using operations such as read and update with certain parameters. Changes in the parameters, i.e. the interface, also change the contract. Some changes are backwards compatible and require no change in the contract. A new version of an API should be created only when the changes are not backwards compatible. A new version of an API is a prompt to the developers to let them know that they should adapt and change their code. Making backwards incompatible changes should be avoided resulting in low frequency of major API versions which is also desirable. These principles go hand-in-hand: Avoid making backwards incompatible changes to avoid breaking the client software to avoid creating new major versions of the API. Finally, API version and software version should be kept separate as their release cycles probably differ considerably. (De, 2017)

API version management can be done using URLs. For example, http://www.forestapi.com/v3/. Another way of versioning is to use HTTP headers. The call to API includes HTTP header which includes the API version that should be called. This approach has one major drawback: How should it be decided which version to use if client call does not include the header? Using a default version to solve the problem has its own ramifications. Third way of going around versioning is to include the version in the call parameters, such as http://www.forestapi.com/api/units?version=2. The parameter doesn’t have to be mandatory. The API call receiver can opt to use default version instead. Finally, there should exist a plan on how to handle deprecated versions. Whether it is returning 404 error or returning a link to newer version. (De, 2017; Kurtz & Wortman, 2014; Varga, 2016)

Small, backward compatible changes to APIs can be published without much hassle. They only require a documentation update and small notification to application developers to let them know of the change. Management of backward incompatible changes is much more difficult. It requires announcement of new version, migration plan and warnings to old versions. On release, old versions should still work for some time simultaneously with the new version to give time for developers to migrate over. (De, 2017) Sometimes it is necessary to rollback to an earlier version when making changes to APIs. In these situations, it is important to manage the previous versions using automated processes (Krintz et al., 2014).

API testing is used to verify the API and its logic. Typically, the testing is done using a software which calls the API and reads the response. APIs are often directly calling the back-end business logic which makes testing them important. Testing software generates API calls based on the API definition. The software then calls the API and attempts to analyze the returning message. It is important that the software is able to automatically generate API calls of differing parameters for good test coverage. Automation is a challenge due to some business logic cases require API calls in a specific order. The order can sometimes be dynamic and also the input parameters can be dynamic. Any policies set in API management platform should be considered in the testing. An example of such policy is the orchestration of API calls in API gateway. The calls have to be done in a specific order to achieve the correct aggregated result. Some tools for API unit and integration testing include Mocha, Chai, JUnit and JMeter. Performance testing tools include JMeter, LoadUI and Wrk among others. (De, 2017) Difficulty in testing web APIs is increased even further as the results are not easy to verify due to the test oracle problem (Segura et al., 2018; Barr et al., 2015), where a test oracle should be available to tell whether a test output is correct or incorrect (Howden, 1978).

Orenstein (2000) argues that APIs are not necessarily hard to develop, instead the difficulties emerge when one tries to learn the APIs developed by others. He notes that API documentation is one of the most important management practices as without documentation the developed APIs will not easily be understood by other developers. Consequently, when the API goes out of date it can be near impossible to update without documentation. API documentation is important especially for the consumers of APIs. Often, the developers of APIs are very familiar with the product, but the consumers are not so. The goal is to enable easy adoption of API. API documentation is a part of the API management scheme and should contain steps to get started quickly, sample code, list of features, list of REST endpoints, message payload documentation and the HTTP status codes that are being used. There should be documentation created both in human- and machine-readable formats. Manually updating documentation and keeping it synced with the implementation is difficult. Tools for automatic API documentation should be used, for example Swagger. Swagger includes tools for code generating, API exploration, editing and more. Some alternatives to Swagger are RAML and API Blueprint. (De, 2017) Swagger can be recommended as it is one of, if not the most used description languages for REST APIs (Haupt et al., 2018).

To facilitate the adoption of APIs, Robillard and DeLine (2011) suggest taking into account five key factors in API documentation: documentation of intent, example code snippets, mapping API scenarios, penetrability of API, and documentation format and presentation. All of the five factors impact developers’ ability to learn APIs. The documentation of intent means that the documentation should tell developers why some API related decision was made and how the API in question is intended to be used. Example code should be provided in the documentation and it should contain best practices and examples of more than single API call. This means that a single example should demonstrate more than one call resulting in a usage pattern. Mapping API scenarios means that the documentation should show developers how certain scenarios can be implemented using the API calls. E.g. the scenario of making an order should be somehow linked to the correct API call. This can be thought as providing usage patterns that are linked to a scenario. Penetrability of API is concerned with how exposed the internals of the API are. The documentation should reveal enough details of the API internals so that developers can understand the impact (e.g. performance impact) of using certain methods. (Robillard & DeLine, 2011)

Based on research on the documentation of SAP, Jeong et al. (2009) provide recommendations regarding API documentation. The appearance of the whole documentation should be consistent and unique. Developers trying to learn API should be presented with high-level image of the documentation which helps them to understand the structure and the scale of the documentation. If the documentation contains action paths, i.e. descriptions of series of tasks to accomplish a bigger goal such as calling an API, the descriptions of the paths should include metadata. Metadata should tell the developers who are the intended target audience for this specific path and how it differs from another similar paths. For example, calling some API can be done via web browser and via application code. There could be two paths targeted at application developers and business users. Bread crumbs can help in navigating the documentation and understanding the structure. In addition, users should be provided a list of always-visible links to selected checkpoints. Checkpoints are the most common starting points for searching the documentation. (Jeong et al., 2009) In the context of this thesis, the documentation of the mobile API could include checkpoint links that point to different ERP branches, such as logistics or invoicing. This enables users to quickly navigate to some area. Jeong et al. (2009) go on to add that the basic features that the API documentation should include are integrated ‘How-to-use’ information and search functionality. ‘How-to-use’ information should be integrated into the documentation, for example, as popups.

In addition to documentation, another aspect of API management that helps application developers is developer portal. Its main purpose is to aid in the adoption of APIs. Developer portal is a platform where developers can get acquainted with the API program. Developer portal acts as the platform where APIs are published for developer use. The portal includes details about published APIs which help developers in adopting new APIs. Developer portal also includes a social aspect where developers can interact with each other to understand the APIs and their purpose. Some of the key features that a developer portal should include are API documentation, test console and application registration. (De, 2017)

Microsoft, Google and Amazon are all offering their API management solutions to go along with their cloud offerings. Microsoft's product is called API Management (APIM) and it includes basic API management features such as gateways and developer portal (Microsoft, 2017). Microsoft's API management solution runs on Azure which is Microsoft's cloud platform. APIM has support for importing APIs using Swagger and automating API management related processes. (CITO Research, 2015) The features differ slightly depending on the subscription. There are five different types of subscriptions available and prices range from tens of euros to couple thousand euros per month. The entry level subscription only contains the gateway component and is a pay-per-use based subscription. The highest tier of subscriptions is intended for high-volume production use. (Microsoft, 2019)

Amazon's API management solution is called Amazon API Gateway. It includes tools for the API lifecycle, i.e. creating, publishing, maintaining, monitoring and securing APIs. (Amazon, 2019a) Swagger generated code can be used to import APIs to Amazon API Gateway (Amazon, 2019b). Amazon API Gateway is priced using a pay-per-use based subscription. There is a limited free subscription available for 12 months. The price of the paid subscription varies based on the amount of API calls and cached data ($1.64-3.5 per million calls in addition to $0.02-3.8 per hour of cached data). (Amazon, 2019c)

Google offers two API management solutions: Cloud Endpoints and Apigee Edge. Apigee Edge is the more comprehensive solution of the two. It includes tools for analysis and monetization among others while Cloud Endpoints is a more basic solution offering features like security, deployment, integration, logging and monitoring. Cloud Endpoints pricing is based on the number of API calls ranging from $0.00 to $3.00 per million API calls. (Google, 2019a; Google, 2019b) There is also a rate limit of 10 million calls per 100 seconds which can be increased if necessary (Google, 2019c). Apigee Edge has four subscription tiers first of which is a free evaluation subscription. The prices of the three paid tiers have to be separately requested and are not available on their website. (Google, 2019d)

Overall, the largest third-party API management solutions seem to include all necessary functionality to manage APIs. Some have free and trial subscriptions which can be used to test out the product to see if it fits for company's needs. The pricing models are divided into pay-per-use and subscription based. Microsoft Azure has been chosen as the cloud platform in the mobile application project at Tieto. Microsoft's API management solution should thus be considered as an alternative to in-house solution.

3 API MANAGEMENT FRAMEWORK

In this chapter, the different aspects of API management that were introduced in chapter 2 will be put into a framework context. According to one definition, a framework can be defined as "broad overview, outline, or skeleton of interlinked items which supports a particular approach to a specific objective, and servers as a guide" (BusinessDictionary, 2019). Thus, an API management framework can be defined as a set of guidelines to API managerial activities. The goal of the framework is to guide users in the right direction regarding API management so that APIs are kept updated and well documented. As a result, users will have some idea on how to manage certain aspects of APIs. The framework is largely based on the literature presented in chapter 2. The main API management related activities covered by the framework are version management, documentation, testing, runtime management (API gateway) and release management (developer portal). Also, small considerations are made regarding naming and other lesser aspects of API management that do not fit into other categories. In practice, many of the API management practices are bundled together in an API management platform such as Microsoft Azure API Management.

In the mobile API project, a bolt-on strategy (Patni, 2017) should be assumed. The strategy is based on the assumption of some existing backend system. Mobile app and APIs will be completely separate from the ERP system which offers the backend services. New functionality should be built on top of existing systems without affecting its internals.

API management for the ERP system should be implemented so that it contains all the main features that are described by Cloud Standards Council (2017). These are automated tools for creating APIs, API management tools, language support, API governance and access control tools. Automated tools are already quite readily available. It is recommended to try different tools and see which suits the project best. For example, Swagger can be a good alternative as it is mentioned many times in literature (e.g. Haupt et al., 2018; De, 2017; Moilanen et al., 2018). Tools should include support for the whole development process from design to deployment. API management tools include tools for monitoring and managing different runtime aspects of APIs such as performance and load balancing. These are typically included in the API gateway part of any API management platform. Their existence should be confirmed. Language support differs slightly between platforms. The support for required programming and specification languages should be verified before selecting an API management platform. API governance is concerned with version management and release management, i.e. how to make APIs available to developers. API governance should be taken care of by documenting a version management process and deploying a developer portal. Finally, access control tools are designed to perform runtime analytics and manage traffic.

3.1 Design

The design of the APIs should start with sketching out use cases. The content of the APIs is then decided based on the identified use cases. Expanding on existing APIs will likely be easier than reducing them. For example, if some use case requires three fields of an order, then only those three should be returned by the API. In future, more fields could be returned if there is a need indicated by a use case. The structure of APIs should be so that main APIs refer to helper APIs to facilitate discoverability. For example, if UNIT is the class to which most programmers first stumble into, i.e. the main class, then UNIT class should refer to other useful ("helper") classes, so programmers can find them easier.

It should be decided whether to create APIs that comply with the RESTful practices. REST offers many desirable features such as low latencies, low communication overhead, high independence and scalability of components. These features would be highly sought after in the ERP system which works in real-time and requires scalable components to serve customers. Thus, the use of REST can quite easily be justified. Best practices for REST APIs can be found from Appendix 1.

In the ERP system, there is the problem of how to implement APIs so that they can adapt to future needs. For example, if a use case has a requirement to get two fields: unit ID and mill order number, what happens when in the future there is a need to get two more fields, such as unit weight and core diameter. This could be solved by using partial responses. By default, the unit API will return all available data on unit, but when needed the application developers can define which fields they want to receive by changing optional parameters in the API call.

3.2 Version management

Version management can be regarded as more important in the context of APIs when compared to traditional software engineering. Particularly in the context of APIs, it is often the case that new versions have indirect effect on all users of the API. Whereas in the context of software engineering, new versions of, for example, mobile apps are often installed automatically and go unnoticed by the user. On the contrary, new API version can cause incompatibility issues when the client software no longer matches the changed API interface rendering the client software unusable in some cases. Such loss of incompatibility is often referred to as a backwards incompatible change, and it is the least preferred scenario. Versioning can be used to handle more radical changes and prevent the loss of compatibility consequently minimizing the negative impact on users.

Changes, sometimes even backwards incompatible ones, are inevitable in APIs. Therefore, it is important to have a predefined process for handling them. Company's style guide or some other documentation should include the rules for when a new version of API is introduced (Moilanen et al., 2018). The rule can be, for example, that a backwards incompatible change is always introduced with a new version. Along with new version, deprecation warnings for the users of old versions should be issued (Tulach, 2008). This gives application developers time to plan necessary changes to their application interfaces. It would also be beneficial to retain support for at least one older version of the API (Patni, 2017), meaning that there are always at least two versions supported in the production environment. Creating new versions should be avoided where possible (De, 2017) to keep the interface between users and API stable. Generally, it is advisable to avoid creating new versions of APIs and try to make incremental, backwards compatible changes instead. API version should not be tied to underlying backend software version (De, 2017). This can be achieved easily by, for example, having separate projects for both applications. Of course, there needs to be clear communication between developers, so the API doesn't brake due to changes in the backend.

Let's say we have an API that returns all available data on units. It is called with mandatory parameter MILLORDERNUMBER and then the API returns all units linked to that specific mill order. Later, there is a need to add another mandatory parameter because mill order number is no longer specific enough. Adding another mandatory parameter in API call is a typical way of creating a backwards incompatible change. In this case, making the parameter optional might not be feasible so we have to create new version of the API. Users of the earlier version are issued warnings of deprecation but can continue to use the API alongside the new one. One alternative is to create a completely new API with different parameters and allow users to use both. Small, backwards compatible changes should naturally be preferred in API development. In most cases, they can be published fast and easy by issuing a notification to application developers and updating the documentation.

In practice, API version can be indicated in quite many ways. Different approaches have their own advantages and drawbacks and the approach should be selected on a case by case basis. In case of the ERP system, the APIs are meant for private use between two enterprises and the application developers are in close contact with the API developers. Thus, it is quite easy to come to an agreement of the API call syntax. HTTP header can be used to indicate the API version. In large public APIs using HTTP header can have the drawback of some application developers omitting the header altogether, but in relatively small and private APIs it can be a viable option.

3.3 Documentation

Documentation is undoubtedly one of the most important managerial practices in API management. It typically consists of machine-readable and human-readable parts which both have their purpose. Machine-readable API documentation facilitates automatic code generation, and thus creates more opportunities for automation. On the other hand, human-readable documentation is vital for application developers who need to get acquainted with the APIs functionality in order to utilize APIs in applications. The two parts of documentation are often very distinct. Machine-readable documentation is implemented using one of many readily available specification languages which have strict syntactical rules and structure. Human-readable documentation can take many forms, but recommendations on how to implement it do exist. Generally, some thought should be given to the presentation of human-readable documentation. Producing documentation can be automated to some extent using software such as Swagger.

The API specification language should be selected based on what is supported by the selected API management platform. In this project, Microsoft Azure is selected as the platform for deploying the APIs. OpenAPI Specification is one very popular (Haupt et al., 2018) API specification language which is supported by Azure. Creating machine-readable documentation has multiple advantages: It allows for automatic code generation, automatic testing and refactoring of code (Wittern et al., 2017). As such, machine-readable documentation reduces manual labor and supports automation. The successful utilization of the features of the underlying API platform, e.g. Azure API Management, can be hindered if machine-readable documentation is neglected.

Human-readable API documentation helps developer familiarize themselves with new APIs. API documentation should contain steps to get started, sample code, list of features, list of endpoints, message payload documentation, HTTP status codes, documentation of intent, scenarios and API internals (Robillard & DeLine, 2011; De, 2017). In addition, the documentation should be consistent and unique (Jeong et al., 2009). This is to help developers recognize and separate the documentation from other documents, and also help memorize the documentation. It should be clear to the reader how the document is structured and what it contains. The documentation should have usability increasing features such as breadcrumbs and links to checkpoints (i.e. common starting points). The documentation should also include instructions on how to use the documentation.

3.4 Testing

The purpose of testing is to verify the correctness of an API. Testing is somewhat difficult aspect of API management. Especially, setting up automatic testing can be a challenge as the results are often not easy to verify. This is called the test oracle problem (Howden, 1978). API testing should be done using software that was made for the purpose. The software relies on the definition of the API, i.e. the machine-readable specification. The difficulties lie in analyzing the returning values of a call and generating diverse test cases. Test coverage can be achieved when testing software generates different parameters for the calls.

Possible API policies set using the API management platform have to be considered in testing. An example of such policy is API call orchestration where number of different API calls are set to be executed in specific order. The correct aggregated result can thus be achieved only using the defined order. Of course, the different APIs can be also tested one by one to verify their individual output. In these cases, one should pay attention to the input parameters which might be the output parameters of some other API call. Some recommended tools for API testing are Mocha, Chai, Junit and JMeter (De, 2017). It can be a good idea to evaluate each of these programs and see which one suits the needs best.

3.5 Runtime and release management

API gateway is a layer between API calls and backend system. It is one of the most important parts of API management platforms (De, 2017). The responsibilities of API gateway are largely related to API runtime management. Runtime management takes care of security related tasks, traffic management, interface translation, caching, routing and orchestration (more in table 5). API management platform should be selected so that needed API gateway features are found in the product.

Release management in this context refers to the publication of APIs to be used by application developers. Taking care of release management is often the purpose of developer portal. The portal shows application developers all the available APIs and they can use the portal to subscribe to needed ones. API developers can release APIs in the developer portal and application developers can browse them. The portal is especially useful for discovering and adopting new APIs. Developer portal expedites application development by offering self-service for application developers. The portal should include details on the APIs (e.g. documentation) to help application developers familiarize themselves with the APIs. Moreover, developer portal should include an interactive platform where developers can interact. This can help developers better understand the APIs. It’s also useful to include a test console which can be used to test the API before subscribing.

3.6 Resources, methods and naming

There are multiple best practices to be followed in regard to resources in APIs. Resources should follow specific naming practices, but also the methods used to modify, or request resources are important. Finally, the structure of the URIs that are used to identify resources should confront to best practices.

In literature, there exists a quite strong consensus on the naming of resources. Best naming practices should be considered to make APIs easier to read and understand for API developers and application developers alike. The company that implements the API should decide on naming conventions that will be used in the API implementation. The conventions should be written down in a style guide or similar document. For example, plural, singular nouns and verbs should be assigned to collections, single resources and controller resources respectively (De, 2017). Same kind of decision should be made on the case of letters and how to convert spaces in API calls. Lower case is recommended throughout the API (De, 2017). Hyphens can be used as the replacement for space characters. When it comes to URI structure, forward slashes should be used to indicate hierarchical relationships and any file extensions should be left out (Petrillo et al., 2016). URI should provide ways to filter and paginate the returning resource representations.

Hypertext transfer protocol contains many methods in addition to GET and POST. It is quite common to find APIs where only one or two of all the available methods are utilized. It is, however, recommended to use HTTP methods to their full extent and for their designated purposes. Using GET and POST for everything including deleting resources is referred to as "tunneling", and it is an antipattern which should be avoided (Palma et al., 2014). GET should be used for retrieving a representation of a resource. Similarly, POST should be used only for creating a resource. Other useful methods are HEAD, PUT, DELETE and PATCH. Most importantly, PUT is used to insert or update resources and DELETE is used to delete resources.

4 FRAMEWORK IN PRACTICE

In this chapter, the framework introduced in chapter 3 and literature in chapter 2 is put into practice. This is done by applying the framework guidelines and designing the UNIT API which includes functionality to retrieve details on a unit. As a background information, a unit refers to a paper reel or pulp bale which typically weighs several hundred kilograms and is physically located in a warehouse. The UNIT API thus retrieves information on a single paper reel or a set of reels. The API also includes methods to insert new units in the system. Units are often linked to some order and using the API they can be moved from one order to another. In addition to the UNIT API, design and considerations for Azure API management are introduced. The use case of relabeling a unit potentially requires dozens of API calls resulting in similar number of endpoints and methods. One of the needed API calls is designed in this chapter. The chapter starts by examining the design of the unit API. Then in the second part, API management practices are proposed for the API.

The use case that is covered with the API is about relabeling a unit, i.e. paper reel. The approximate flow of actions starts with a storage worker whose task is to move a paper reel to another mill order and print a new label for the reel. The worker scans current barcode with a mobile device and requests moving to another mill order. Some of the services of the ERP system are called via API to obtain data on the paper reel, related mill order and any allocations. The movement of the paper reel between mill orders is processed, and new label is printed for the storage worker. The process includes a check to verify the user and sufficient user rights. The full definition of the use case can be found in appendix 2.

4.1 API design

The design of the relabeling API is centered around the use case description and stakeholders’ needs. Additionally, the design considers quality attributes that are perceived as important in the literature. According to Stylos & Myers (2007), these are usability and power. The design should thus contain elements which promote learnability, consistency, extensibility, performance, and prevent errors. Aspects of the design which promote the quality attributes are listed in table 8. The design follows mostly the bolt-on strategy as described by Patni (2017). In this strategy, a backend system on top of which the APIs will be built is assumed. The backend remains mostly untouched and separated from the APIs and mobile application. The design also attempts to reflect the needs of the stakeholders which are listed in table 9.

Table 8. The connection between design solutions and quality attributes

Quality attribute	**Design solutions**
Learnability	- Machine- and human-readable documentation - Consistent naming of API calls - Consistent design
Consistency	- Consistent naming of API calls - Consistent design
Extensibility	- Consideration given to API structure: Which fields are defined mandatory? Can the API be extended to include more fields later?
Performance	- Minimizing service calls to performance-heavy backend services - Caching
Error prevention	- Error checking and recovery - Implement proper error codes (see table 10)

Table 9. Relabeling API main stakeholders

Stakeholder	**Main needs**
API developer/designer	- Ease of development
Application developer	- Learnability and discoverability of APIs - Fast and efficient application development
Storage worker	- Working software without bugs
Customer IT personnel	- Minimize development cost

Based on the use case definition (see appendix 2), some endpoints can be identified as mandatory for the successful execution of the use case. First, the crucial pieces of data that are verified or retrieved are unit details, mill order details and allocation details. Second, the unit is moved to another mill order. Last, new label is printed for the moved unit. User access checks are omitted here as they do not belong to the core flow of the use case. In other words, the use case can be fulfilled even without checking for user rights, although it poses a security risk. Core resources, including their properties and methods are shown in table 10. Query parameters support filtering, pagination and utilization of optional search terms. Optional parameters can be, for example, used to define what data should be returned.

In addition to resources mentioned in table 10, it's worth considering if print jobs and allocations would deserve their own resources and not be included inside unit. All resources use GET to retrieve a representation of the resource. Unit also supports PUT and POST which allow to modify and insert resource. This is needed to be able to move units between orders. The naming of endpoints and URIs should be implemented as instructed in the framework and in appendix 1. Mainly, verbs, singular and plural nouns should be used correctly. CRUD names are to be avoided, hyphens are used to replace space characters and lower-case characters are used throughout the naming scheme.

Table 10. Relabeling a unit: Resources and properties

Resource {Methods}	**Properties**	**Endpoints and query parameters**
Mill order {GET}	- Mill order number - Order line number - Machine chain number	/orders/ /orders/order/ ?q Optional search term ?{column name} Filtering ?p Pagination
Unit {GET, POST, PUT}	- UIB - Allocating mill order - Allocating order line number - Allocating machine chain number - Mill order number - Order line number - Machine chain number - New mill order number - New order line number - New machine chain number	/units/ /units/unit/ ?q Optional search term ?{column name} Filtering ?p Pagination

User {GET}	- User ID - Home group - Active directory user - Password	/users/ /users/user/ ?q Optional search term ?{column name} Filtering ?p Pagination

The relabeling API will use JavaScript Object Notation (JSON) in return sets. Support for content negotiation and additional markup languages can be added in the future if deemed necessary but is omitted for now. The HTTP status codes that are used, and their meaning, are covered in table 11.

Table 11. Unit API: HTTP status codes

Status code	**Explanation**
200	Used to indicate success
201	Resource was created successfully
204	No content to return
400	General return code for unspecified error
401	Problems in authentication
403	Access is denied
404	Requested resource doesn’t exist
405	Used method was not supported
406	Requested media type is not supported
500	Server-side error, i.e. a malfunction in the API

In the ERP system, the UNIT table contains 130 columns so there is the potential of having to make dozens of changes to the API every time some new column value needs to be returned. This may happen in a scenario where initially an API which supports 10 fields is created. Later on, it’s noticed that actually two more fields need to be returned and a change to the API is implemented. Clearly, this is not desired, but rather all of the columns should be supported from the beginning. In the implementation of the UNIT details API, an optional parameter should be included and used to define which fields are to be returned. This optional parameter can be used for retrieving data of single or multiple units, i.e. collection of units. There could also be defined a set of default columns which are seen as necessary to return on every call. To get data on single unit, one must be able to single out a unit based on some fields. The fields that single out a unit should be defined as mandatory in the API call. They are naturally derived from the primary key of the UNIT table. In this case, mandatory fields would be mill order number (MILLORDNO), machine chain number (MACHCHAINNO), order line number (ORDERLINENO) and unit identifier (UNITID).

The UNIT table holds over 28 million rows as of July 2019. Returning such huge number of rows is not reasonable nor useful. Thus, we'll need to implement filtering and pagination in the API. They are used when unit collection is retrieved using the UNIT API. In such cases, caller can define how to filter the collection and whether to paginate it or not. In addition to pagination, a default result set size should be defined to avoid situation where huge result sets are returned by accident. Filtering of results can be implemented for all columns or some predefined set of columns that are seen as the most useful. For example, filtering would allow user to retrieve a collection of all the units that belong to a specific mill order. Even more specific filtering can be supported by, for example, specifying mill order number and order line number. Example UNIT API implementation details are listed in table 12. Pagination and filtering are not necessary in API call which returns the representation of a single unit as there is nothing to filter and the result set is always of fixed size.

*Table 12.*UNIT API details

Method	**Endpoint**	**Parameters (m=mandatory, o=optional), Possible parameter values**	**Return values**
GET	/units/unit	millordno (m), varchar(16) machchainno (m), char(1) orderlineno (m), char(2) unitid (m), varchar(16) q (o), optional columns: any of the columns in UNIT table including support for multiple columns	Return representation of a single unit with following columns as default: - millordno - machchainno - orderlineno - unitid - uib HTTP status code:

			- 200
GET	/units	filtering (o), any of the columns that are supported for filtering p (o), pagination: limit the number of pages q (o), optional columns: any of the columns in UNIT table including support for multiple columns	Representation of a unit collection HTTP status code: - 200
PUT	/units/unit	millordno (m), varchar(16) machchainno (m), char(1) orderlineno (m), char(2) unitid (m), varchar(16) columns to update (o), which columns should be updated?	HTTP status codes: - 200 or 204 on successful update - 201 on successful creation
POST	/units/unit	millordno (m), varchar(16) machchainno (m), char(1) orderlineno (m), char(2) unitid (m), varchar(16) optional column values (o), any values that should be assigned to optional columns	HTTP status code: - 201 on successful creation

Based on table 12, example API calls and JSON response body are illustrated below. For clarity's sake, the parameters are separated with a new line. The naming scheme should follow the guidelines introduced in the literature and the framework (see chapters 2 and 3). According to the framework, lowercase letters should be used. However, some data in the backend is always uppercase such as database column names and letters in mill order numbers. It should be decided on whether to consistently use the same case as in the backend or convert everything to lower case. In below examples, lowercase is used throughout. API call to get a representation of a single unit with additional columns UDATE and UBY would look like this:

```
GET /units/unit?
millordno=abcd-123456&
machchainno=0&
```

```
orderlineno=01&
unitid=1234567890123456&
q=udate&
q=uby
```

Example response body from previous call, assuming that specified unit exists, would return JSON similar to this:

```
{
        “units”: [
                {
                        “millordno”: “ABCD-123456”,
                        “machchainno”: “0”,
                        “orderlineno”: “01”,
                        “unitid”: “1234567890123456”,
                        “uib”: “0987654321654321”,
                        “udate”:“2017-06-22 10:35:48.000”,
                        “uby”: “alailko”
                }
        ]
}
```

Similarly, a call to UNIT API to retrieve a collection of units filtered by mill order and order line, limited to five pages and including optional columns UBY and UDATE would look something like this:

```
GET /units?
        millordno=abcd-123456&
        orderlineno=01&
        p=5&
        q=uby&
        q=udate
```

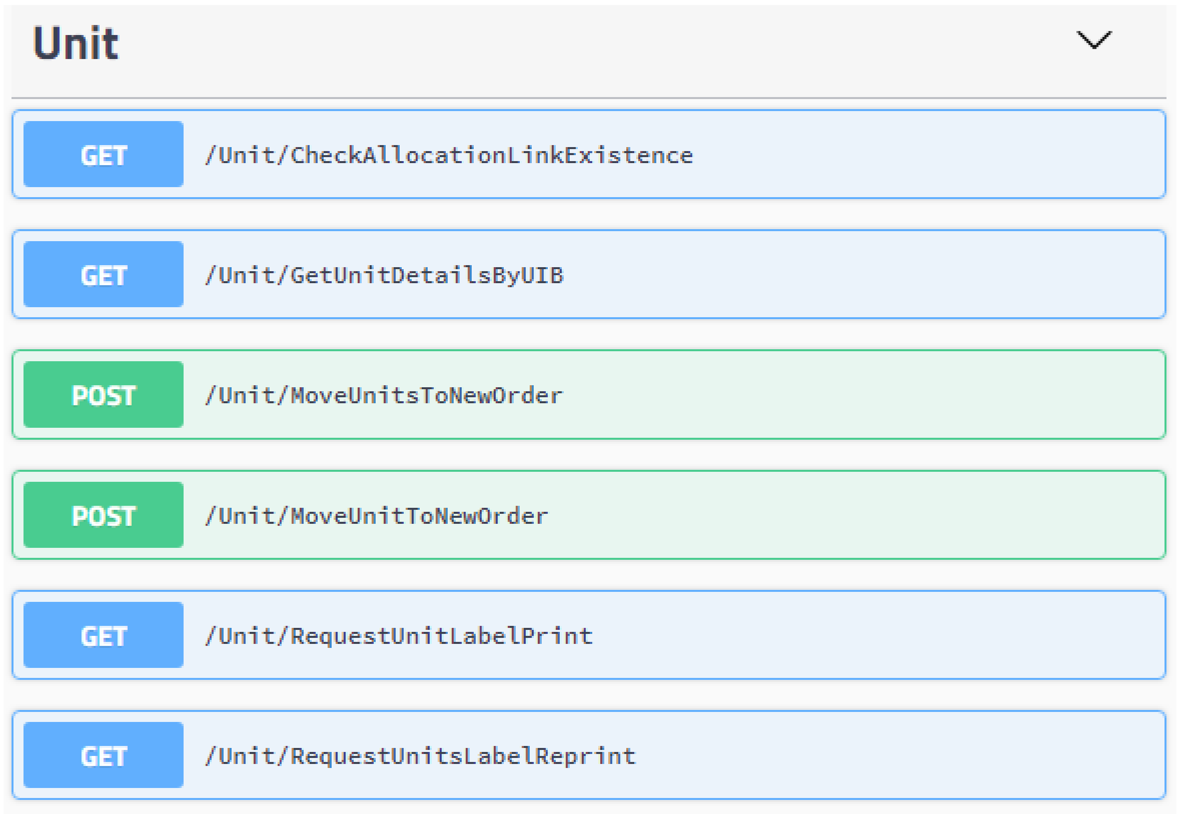
Response body would include all the units that meet the filtering criteria. The response would be similar to the representation of a single unit, but the “units” array would contain many units. In cases where the result set contains hundreds of records, the response could be paginated. This means that a certain number of records is sent in a single response along with a link to the next page. In the above example, the “p” parameter limits the number of pages to five.

UNIT API calls using PUT method would either update existing resource or create a new resource. The benefit of PUT is idempotency meaning that same call can be safely made many times in succession. On the contrary, many POST calls would create similar amount of new resources or return errors. In PUT calls, the caller can define the columns that should be updated using query parameters. These are optional, so leaving them out would result in no action.

4.2 API management design

The Azure cloud platform currently hosts plenty of APIs which were developed in Tieto to test the feasibility of the mobile application. These include APIs dealing with unit (see figure 3), user and order details. The APIs can currently be used for basic tasks such as retrieving details of users, orders and units, but also for more complex ones such as moving units from order to order. The pre-existing APIs are not part of this thesis as such, but they are used as examples to help design the management. The APIs are documented using Swagger which allows users to experiment with the APIs and provides the request endpoint, parameters, result and response codes. There is, however, no way for API developers to properly manage or monitor the APIs. The basic components of an API management platform are included in the design (see chapter 2.3 API management). These include the runtime, gateway, management, developer portal and developer tools components which allow users to monitor traffic, set security policies, and discover and deploy new APIs among other activities.

In Tieto, the API management process starts with defining the API strategy. Definition of API management setup and platform setup follow along with defining API management policies and security related issues. As Moilanen et al. (2018) put it, the needs of the customer should be defined first: What customer needs does the API fulfill? Similarly, it should be defined what customer needs does API management fulfill. In regard to API management, it is a platform which allows customers to discover, manage and monitor APIs. Without API management, customers can face difficulties in everyday API activities such as generating usage data or setting security policies.


Unit

GET /Unit/CheckAllocationLinkExistence

GET /Unit/GetUnitDetailsByUIB

POST /Unit/MoveUnitsToNewOrder

POST /Unit/MoveUnitToNewOrder

GET /Unit/RequestUnitLabelPrint

GET /Unit/RequestUnitsLabelReprint

Figure 3. Sample unit APIs

API strategy can consist of practical matters such as naming conventions and versioning but also the planned setup for the management platform and the goals that are set to be achieved using the setup. API strategy definition is done by considering stakeholder needs and desired business outcomes (MuleSoft, 2019). For example, it is decided which components of API management platform to deploy and what benefits they bring to stakeholders and customers. The business needs are not further considered in this study but rather the focus is on stakeholder needs. Naming conventions affect application and API developers. Uniform naming eases learnability and discoverability of APIs. Literature chapter (chapter 2) identified many naming conventions which should be followed in the API management platform design. The conventions concern the capitalization, logical structure and the usage of special characters and words in method, endpoint, API and version names.

The high-level API structure is defined in a way that makes it easy to understand for all related stakeholders. In Tieto's case, the company maintains many separate software systems which all may require back-end services to be published as APIs. These include separate

ERP systems and other predictive software systems. Some of the systems are fully managed by Tieto (internal) and others are only partly managed inside Tieto premises (external). It is likely that the services of these systems need to be called using mobile devices. Mixing all of these under one API is to be avoided as the structure can get very messy and difficult to comprehend when more and more systems are being added. The structure should follow the division of main functionalities in the ERP system. For a structure proposition, see figure 4. The source code is similarly divided into main and sub functionalities in the Azure repository.

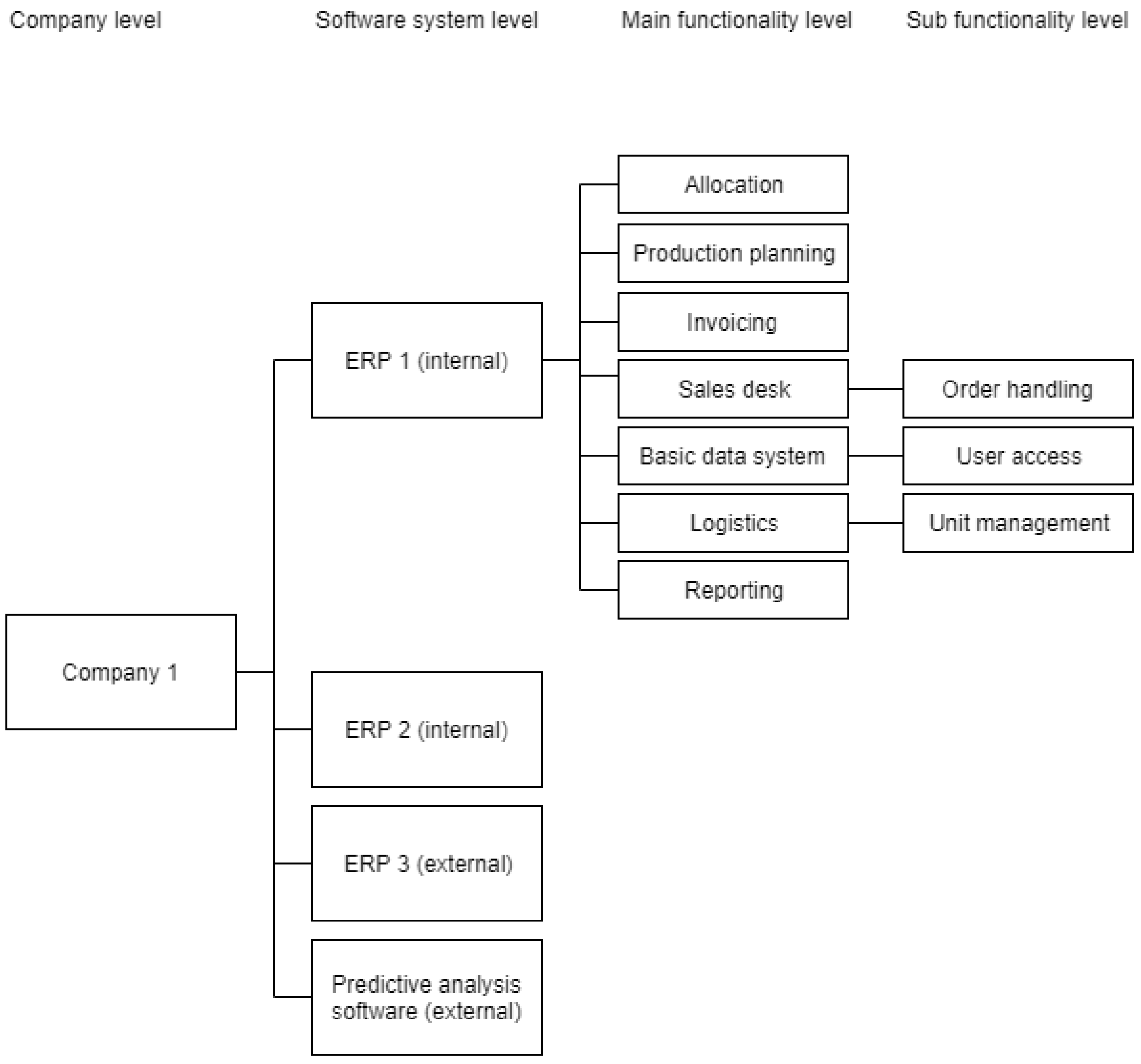


Figure 4. API structure proposition

The company level is the whole Microsoft Azure platform which contains several projects and their source codes which all belong to the same customer company. The projects in Azure form the software system level where each software system has its dedicated project.

The software systems are further divided into main and sub functionalities. The ERP system in this study ("ERP 1" in fig. 4) is given as an example. The repository folder structure supports the structure of the ERP. The structure of APIs and API management is similarly divided into different levels. Azure API management is deployed in the company level in Microsoft Azure platform. Instances of API management platform components can be deployed for each software system separately. In other words, one component, e.g. developer portal, for each software system. Finally, the API management platform components should be divided based on system functionalities where feasible. For example, the developer portal should have a structure where all main functionalities have their own section, so the structure is familiar to developers. API source code folder structure should be similar to the ERP repository structure.

API management platform setup consists of the different API management platform components. The parts that are deployed are to be selected based on their implications. For example, developer portal supports the discoverability and learnability of the APIs. Developer portal is selected as part of the setup if these quality attributes are considered as important. Omitting developer portal may consequently limit developers' ability to discover needed APIs, thus prolonging development time. Similarly, gateway component has many important roles in runtime management and should be included in the management setup. API developer tools can facilitate development by providing tools for automating some of the developer tasks such as code generation, testing and documenting. API management component supports version management, change management, cataloging and publishing APIs. Additionally, it can be used to create the API structure by grouping related APIs to packages to which developers can subscribe. Analytics component is not necessary in the setup at first but can be added later if needed. In conclusion, the API management platform setup for the ERP system would consist of developer portal, gateway, management component and developer tools. This set of components supports the activities listed in the API management framework: Design, version management, documentation, testing, runtime management, release management and naming. An example of the whole API management structure for the ERP system can be found in figure 5. The structure contains some of the currently implemented sample APIs to demonstrate the distribution of APIs inside the API management platform.

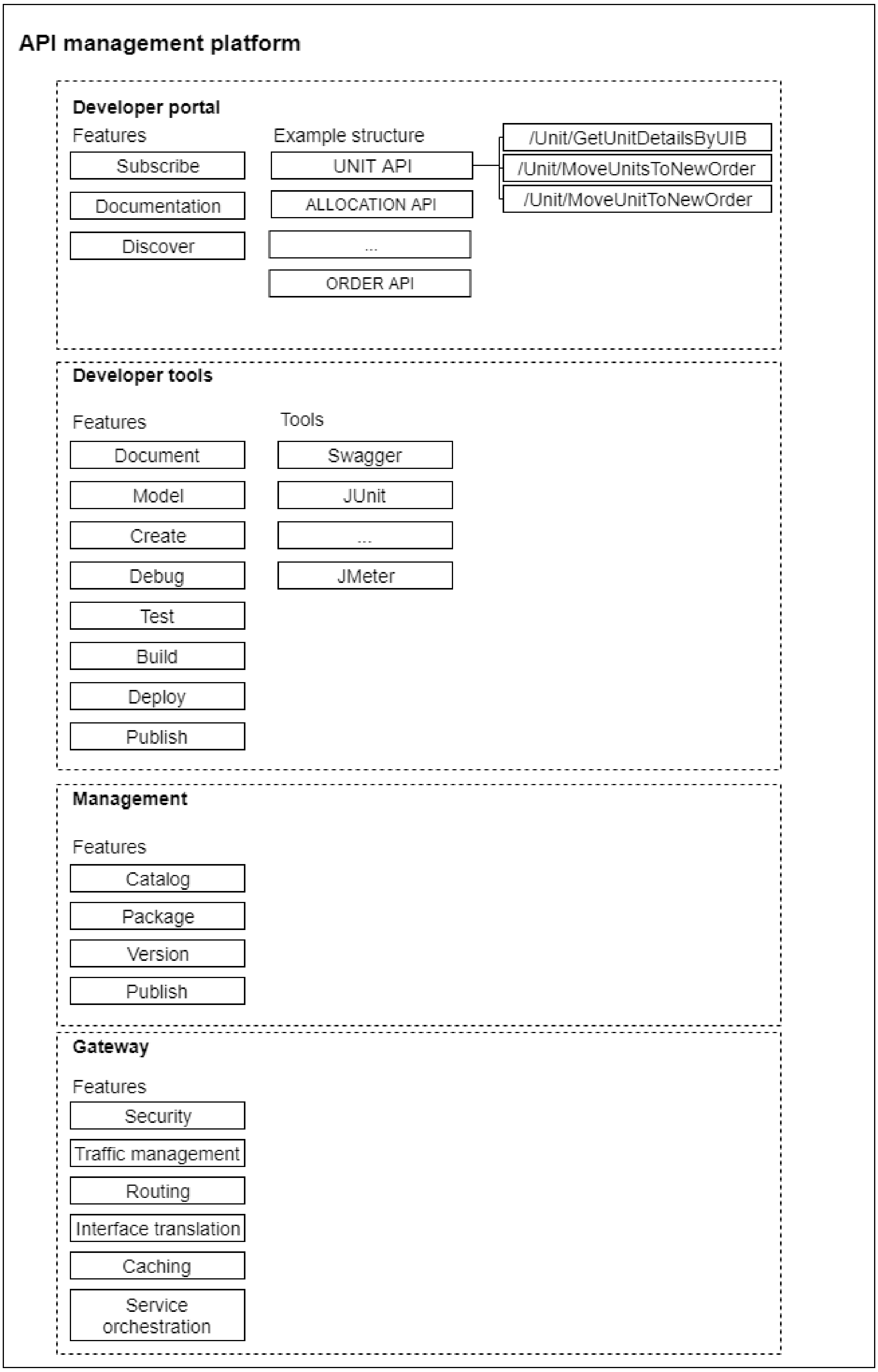


Figure 5. Example API management structure for the ERP system

5 RESULTS

Literature review on API management included three main topics: APIs in general, API design and API management. First part of the review focused on APIs in general finding out the definition of APIs and why they are so prevalent today. It also shed some light on the purpose and benefits of APIs. API design chapter discussed the main quality attributes that are often linked to good API design. Identified quality attributes were usability, power, consistency, discoverability, learnability, simplicity and efficiency. Some ways to achieve the attributes through design were also proposed in the form of best practices, design patterns and antipatterns.

The process of designing APIs was explained as starting from the stakeholders and their needs and creating a strategy and use cases based on the needs. The definition of the API was important to get done early in the process. The proposed communication protocol was HTTP and transport language JSON. It was suggested to use the REST principles when designing APIs. These were described in the Richardson maturity model as resources, HTTP verbs and hypermedia. The design chapter also introduced some API specification languages along with open source tools to automate code generation, design, document, visualize and test. It was suggested to create a design guide inside a company to agree on design practices such as naming and versioning. The implementation of machine-readable API specification was strongly suggested to facilitate API management solutions. API management chapter explained the structure of API management platforms which consist of different components such as analytics, gateway, developer portal and developer tools. The purpose of each component was discussed in more detail. The qualities of a good API management platform were mentioned. Finally, versioning, testing, naming and documentation were explained.

The literature review was used to create a framework for API management. The framework introduced and clarified different processes of API management. The purpose was to provide guidelines on how to execute each process of API management. The framework was put into practice by applying its guidelines to design proposals for an API and API management. These designs attempted to reflect the best practices while also considering the context of the thesis, i.e. a large enterprise resource planning system.

In conclusion, APIs should be designed based on stakeholder needs. The needs can be refined into a set of quality attributes with different priorities. The quality attributes can then be realized in the design by utilizing design patterns and best practices. APIs should be managed using the components of an API platform. Each component has different set of features which contribute to the success of the management as a whole. Maintaining up to date APIs requires careful upfront design but also well-defined version, runtime and release management. In addition, the documentation, both human- and machine-readable, should be created and updated regularly. Style and design guides should be introduced within the company to define versioning strategy and the ways in which new endpoints, resources and methods are created and named. The design guide should also include details on which HTTP verbs are used and for what purpose. Managing changes to APIs should be done using a change management process. The process should define how backwards incompatible changes are handled. For example, how version deprecation warnings are distributed and how far back old versions are supported.

6 DISCUSSION AND CONCLUSIONS

The goal of this study was to find information on API management and design and to use that information to create a proposal on how to manage APIs in the context of large ERP system. The study was divided into literature review, framework implementation and design parts. A literature review was conducted to find best practices on how to design and manage APIs. Some best practices by different authors were found and introduced. The literature review provides quite good high-level understanding of the topic. The API design process and the structure of an API management platform are especially interesting from the thesis' point of view. Some of the best practices that were identified are very helpful in the design of APIs and API management. The findings from the literature review were used to create a small framework for API management. The framework contains guidelines on how to implement different aspects of API management. With the help of the framework, a high-level API and API management designs were proposed.

The study evidences that the literature along with the implemented framework could be used to produce tentative design for API management. Material from the study can be later used in the company to implement APIs and API management. It can be useful, especially, in providing the high-level guidelines, but also some very practical advice. The study reached its goal to find information on API management, and thus can be considered a success. The framework and design part of the study could've been more detailed. However, both are quite large topics, and further studies can be done on them individually. These studies are left for future research.

Based on the framework, designs for an example API and API management were proposed. The quality of these designs is not assessed in the study. The quality is based on the quality of the literature which was reviewed in the study. Future research can be done on assessing the quality of the framework, designs, and possible implementation of APIs and API management based on the guidelines that were introduced in this study. API security and related security protocols were not included in the study and are a topic of future research.

REFERENCES

Amazon. (2019a). *API Management*. Retrieved from https://aws.amazon.com/api-gateway/api-management/

Amazon. (2019b). *Amazon API Gateway FAQs.* Retrieved from https://aws.amazon.com/api-gateway/faqs/

Amazon. (2019c). *Amazon API Gateway pricing*. Retrieved from https://aws.amazon.com/api-gateway/pricing/

APIacademy. (2015a). *API management architectural components.* Retrieved from https://www.apiacademy.co/lessons/2015/04/api-management-architectural-components

APIacademy. (2015b). *API Management 103: Choosing a Solution.* Retrieved from https://www.apiacademy.co/lessons/2015/04/api-management-103-choosing-a-solution

Barr, T., Harman, M., McMinn, P., Shahbaz, M. & Shin, Y. (2015). *The Oracle Problem in Software Testing: A Survey.* IEEE Transactions on Software Engineering, 41(5), 507–525.

Bermbach, D. & Wittern, E. (2019). *Benchmarking Web API Quality – Revisited.*

Bloch, J. (2006). *How to Design a Good API & Why it Matters* [Online presentation]. Retrieved from https://www.infoq.com/presentations/effective-api-design

BusinessDictionary. (2019). *framework.* Retrieved from http://www.businessdictionary.com/definition/framework.html

CITO Research. (2015). *Cloud-based API Management: Harnessing the Power of APIs.* Retrieved from https://j.mp/ms-apim-whitepaper

Cloud Standards Customer Council. (2017). *Cloud Customer Architecture for API Management.* Retrieved from https://www.omg.org/cloud/deliverables/CSCC-Cloud-Customer-Architecture-for-API-Management.pdf

De, B. (2017). *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization.* Berkeley, CA: Apress.

Doglio, F. (2015). *Pro REST API Development with Node.js*. Berkeley, CA: Apress.

Dooley, J. (2011). *Software Development and Professional Practice*. New York: Apress.

Familiar, B. (2015). *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*. Berkeley, CA: Apress.

Fielding, R. & Taylor, R. (2002). *Principled design of the modern Web architecture*. ACM Transactions on Internet Technology (TOIT), 2(2), 115-150.

Fielding, R. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* Ph.D. dissertation, U. of California, Irvine.

Flexera. (2019). *State of the cloud report*. Retrieved from https://resources.flexera.com/web/pdf/RightScale-2019-State-of-the-Cloud-Report-from-Flexera.pdf

Garber, L. (2013). *The lowly API is ready to step front and center*. Computer, 46(8), 14-17.

Google. (2019a). *Cloud Endpoints*. Retrieved from https://cloud.google.com/endpoints/

Google. (2019b). *API Management Platform*. Retrieved from https://cloud.google.com/apigee/api-management/

Google. (2019c). *Pricing and quotas.* Retrieved from https://cloud.google.com/endpoints/pricing-and-quotas

Google. (2019d). *Apigee Edge*. Retrieved from https://cloud.google.com/apigee/pricing/

Haupt, F., Leymann, F. & Vukojevic-Haupt, K. (2018). *API governance support through the structural analysis of REST APIs*. Computer Science Research and Development, 33(3-4), 291-303.

Henning, M. (2007). *API design matters*. ACM Queue 5, 4, 24–36.

Howden, W. E. (1978). *Theoretical and Empirical Studies of Program Testing*. IEEE Transactions on Software Engineering, 4(4), 293–298.

IBM. (2019). *Use-case template.* Retrieved from https://www.ibm.com/support/knowledgecenter/en/SSWSR9_11.6.0/com.ibm.pim.dev.doc/pim_ref_usecasetemp.html

Jeong, S. Y., Xie, Y., Beaton, J., Myers, B. A., Stylos, J., Ehret, R., Karstens, J., Efeoglu, A. & Busse, D. K. (2009). *Improving documentation for eSOA APIs through user studies*. In: Proc. 2nd int'l symp. on enduser development. LNCS, vol 5435. Springer, 86–105.

Krintz, C., Jayathilaka, H., Dimopoulos, S., Pucher, A., Wolski, R. & Bultan, T. (2014). *Cloud Platform Support for API Governance*. IEEE International Conference on Cloud Engineering, Boston, MA, 615-618.

Kurtz, J. & Wortman, B. (2014). *ASP.NET Web API 2: Building a REST Service from Start to Finish.* Berkeley, CA: Apress.

Macvean, A., Church, L., Daughtry, J. & Citro, C. (2016). *API Usability at Scale*. In 27th Annual Workshop of the Psychology of Programming Interest Group-PPIG 2016. 177–187.

Maleshkova, M., Pedrinaci, C. & Domingue, J. (2010). *Investigating Web APIs on the World Wide Web*. Web Services (ECOWS), 2010 IEEE 8th European Conference on, 107-114.

Masse, M. (2011). *REST API Design Rulebook.* Sebastopol, CA: O'Reilly Media.

Microsoft. (2017). *About API Management.* Retrieved from https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts

Microsoft. (2019). *API Management pricing*. Retrieved from https://azure.microsoft.com/en-us/pricing/details/api-management/

Moilanen, J., Niinioja, M., Seppänen, M. & Honkanen M. (2018). *API-talous 101*. Helsinki: Alma Talent.

Molich, R. & Nielsen, J. (1990). *Improving a human-computer dialogue*. Communications of the ACM, 33(3), 338-348.

MuleSoft. (2019). *API strategy essentials: A practical guide for winning in the API economy.* Retrieved from https://www.mulesoft.com/lp/whitepaper/api/api-strategy-essentials

Orenstein, D. (2000). *Application programming interface*. Computerworld, 34(2), 66.

Palma, F., Dubois, J., Moha, N. & Guhneuc, Y. G. (2014). *Detection of REST patterns and antipatterns: a heuristics-based approach*. ICSOC 2014. Berlin: Springer.

Palma, F., Gonzalez-Huerta, J., Moha, N., Guéhéneuc, Y.G. & Tremblay, G. (2015). *Are RESTful APIs Well-Designed? Detection of their Linguistic (Anti)Patterns*. In: Service-Oriented Computing, Lecture Notes in Computer Science, 8954, 171–187. Springer International Publishing, Cham

Patni, S. (2017). *Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS.* Berkeley, CA: Apress.

Petrillo, F., Philippe, M., Naouel, M. & Yann-Gaël, G. (2016). *Are REST APIs for Cloud Computing Well-Designed? An Exploratory Study*. 14th International Conference on Service Oriented Computing (ICSOC), Oct 2016, Banff, Canada. 157–170

Pinkham, R. (2017). *What Is the Difference Between Swagger and OpenAPI?* Retrieved from https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi/

Richardson, L. & Ruby, S. (2007). *RESTful Web Services.* Sebastopol, CA: O'Reilly Media, Inc.

Richardson, L., Ruby, S. & Amundsen, M. (2013). *RESTful Web APIs.* Sebastopol, CA: O'Reilly Media.

Righetto, D., Pham, M. & Prasad, G. (2019). *REST Security Cheat Sheet*. Retrieved from https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/REST_Security_Cheat_Sheet.md

Robillard, M.P. & DeLine, R. (2011). *A field study of API learning obstacles*. Empirical Software Engineering, 16(6), 703-732.

Rodríguez, C., Baez, M., Daniel, F., Casati, F., Carlos, J., Canali, L. & Percannella, G. (2016). *REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices*. In: 16th International Conference on Web Engineering (ICWE2016). Lugano

Segura, S., Parejo, J., Troya, J. & Ruiz-Cortes, A. (2018). *Metamorphic Testing of RESTful Web APIs.* Software Engineering, IEEE Transactions on, 44(11), 1083-1099.

Stowe, M. (2015). *Undisturbed REST: A guide to designing the perfect API.* MuleSoft

Stylos, J. & Myers, B. (2007). *Mapping the space of API design decisions.* In IEEE symposium on visual languages and human-centric computing. IEEE, 50–60.

Stylos, J. & Myers, B. (2008). *The implications of method placement on API learnability*. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering. ACM, 105–112.

Stylos, J. & Myers, B. (2016). *Improving API Usability*. Communications of the ACM. 59(6).

Tulach, J. (2008). *Practical API Design: Confessions of a Java Framework Architect*. Berkeley, CA: Apress.

Varga, E. (2016). *Creating Maintainable APIs: A Practical, Case-Study Approach*. Berkeley, CA: Apress.

Vinoski, S. (2008). *RESTful Web Services Development Checklist.* IEEE Internet Computing 12(6), 96–95.

Wittern, E., Ying, A., Zheng, Y., Laredo, J., Dolby, J., Young, C. & Slominski, A. (2017). *Opportunities in Software Engineering Research for Web API Consumption*. IEEE/ACM 1st International Workshop on API Usage and Evolution (WAPI), Buenos Aires, 7-10.

APPENDIX 1. REST API design best practices

List of REST API design best practices collected and divided into five categories by Petrillo et al. (2016). Best practices are originally created by Masse (2011), Richardson & Ruby (2007), Palma et al. (2015), Rodríguez et al. (2016), Stowe (2015) and Vinoski (2008).

Category	**Best practice**
URI	Forward slash indicates hierarchical relationship in URIs
	Trailing forward slashes are not used in URIs
	Hyphens improve readability of URIs
	Underscores are not used in URIs
	Lowercase letters should be used in URIs instead of upper-case letters
	File extensions are omitted from URIs
	Subdomain names in URIs should be kept consistent throughout the API. For example, http://api.forest.restapi.org for the API and http://developer.forest.restapi.org for developer portal.
	Singular nouns are used for document names
	Plural nouns are used for collection names, e.g. collection of documents
	Plural nouns are used for store names
	Verbs are used for controller names
	CRUD names are not used in URIs
	Path variables (/**store**/**book**) should be used to establish hierarchical relationships and not to pass query parameters
	Don't include API version number in URI path
	URI can be used to filter resources
	URI can be used to paginate resources
	URI should contain most of the information, not the request metadata
	Version number should not be included in query parameters
Request methods	GET and POST should only be used for their real purpose and not to replace other functions
	GET is used to get a representation of a resource
	HEAD is used to get response headers
	PUT is used to insert or update a resource
	POST is used to create a new resource

	DELETE is used to remove a resource
Error handling	200 (OK) is used to indicate success
	200 is not used to indicate errors
	201 (Created) is used to indicate that resource was created successfully
	202 (Accepted) is used to indicate the beginning of an asynchronous action. The request seems valid, but information of its result is not yet available.
	204 (No content) is used when there is nothing to return and the body of response is left empty on purpose
	302 (Found) is not used
	304 (Not modified) is used to save bandwidth
	400 (Bad request) is used to indicate error that cannot be specified more accurately
	401 (Unauthorized) is used to indicate problem in authentication
	403 (Forbidden) is used to prevent access
	404 (Not found) is used when the resource requested by the client does not correspond to anything
	405 (Method not allowed) is used when client uses a method that is not supported
	406 (Not acceptable) is used when client requests a media type which the server cannot return
	409 (Conflict) is used to indicate that the state of a resource is being violated
	500 (Internal server error) is used in cases where the API is not working normally, i.e. malfunctions
	JSON is used for error messages
HTTP header	Use ‘Content-Type’
	Use ‘Content-Length’
	Use ‘Last-Modified’
	Use ‘ETag’
	Support conditional PUT method
	When a new resource is created its location is indicated using Location
	‘Cache-Control’, ‘Expires’ and ‘Date’ headers are used to enable caching

	Encourage caching
	Do not create custom HTTP headers
Others	Use content negotiation
	Support media type selection in query parameters
	Support JSON as the media type for resource representations

APPENDIX 2. Description of use case: Unit relabeling

Use case template is from IBM (2019).

Use case name	Relabel a paper reel
Subject area	Logistics
Business event	Unit moved from order to order
Actors	Storage workers
Use case overview	Use mobile application to scan a barcode of a paper reel, assign new label and print the label
Preconditions	The barcode of a paper reel has to be successfully scanned
Termination outcome	Successful outcomes: - Unit is moved to a new mill order - New label is printed for the unit that was moved Unsuccessful outcomes: - Barcode scan fails - Unable to move unit - Unable to print new label
Condition affecting termination outcome	- Unclear barcode - Business rules - Hardware failure (e.g. mobile device, printer) - Software failure - Human error - User has limited or no rights to access the functionality - Backend system or services are down
Use case description	Paper reel should be moved to another mill order and new label printed: - Storage worker opens the application using a mobile device such as a tablet. - User scans the barcode that is attached to the reel. - The UIB of the reel is extracted from the barcode and multiple API calls are made with the UIB as a mandatory parameter. The ERP system responds with details about the reel and related mill order and allocations.

	- User inputs the number of a new mill order to which the reel should be moved. System responds with HTTP status code 200 (success). - User requests printing of new label. System responds with HTTP status code 200 (success).
Use case associations	- Get unit details by scanning a barcode
Traceability to	This use case is related to following entities: - Mill order - Unit - Allocation - User - Location - Home group
Input summary	Storage worker inputs following data: - Barcode of a paper reel - Desired action (e.g. move unit to new order, print new label)
Output summary	Following data is output by the backend system based on the scanned barcode: - Unit data - Mill order data - Allocation data
Use case notes	