jamk.fi

API-First Design with Modern Tools

Oona Hämäläinen

Bachelor’s thesis
May 2019
School of Business
Degree Programme in Business Information Technology

jamk.fi

Description

Author(s) Hämäläinen, Oona	Type of publication Bachelor’s thesis	Date May 2019
		Language of publication: English
	Number of pages 57	Permission for web publi-cation: x

Title of publication
API-First Design with Modern Tools

Degree programme
Business Information Technology

Supervisor(s)
Kiviaho, Niko

Assigned by
Tieto Finland Oy

Abstract

Business strategies that aim to benefit from APIs are rapidly becoming more common in the IT industry. Well-designed and implemented APIs are an important step towards successful API economy, and for this reason the assigner of the thesis was interested in an API design strategy called API-first design.

The objective of the thesis was to research how to apply API-first design in practice and what kind of benefits does it offer, and also to test this method to find out an effective way of using it in Microsoft Azure environment. Different tools and standards dedicated to API design were expected to make API-first design more efficient. The research method used was design research, where the API-first process was tested iteratively to evaluate how different tools and standards were able to support this process with Azure API Management service.

The result of the thesis was a working API-first process in Azure environment that uses OpenAPI 2.0 standard and the tools Swagger Editor and Swagger Codegen. The research concluded that API-first design is a beneficial method to be used in Azure environment, and that tools and standards can support this process very well by making the development faster and more precise and making it easier to start applying this design method.

Keywords/tags (subjects)
API, REST API, API-first design, API economy, Azure API Management

Miscellaneous (Confidential information)

jamk.fi

Kuvailulehti

Tekijä(t) Hämäläinen, Oona	Julkaisun laji Opinnäytetyö, AMK	Päivämäärä Toukokuu 2019
		Julkaisun kieli Englanti
	Sivumäärä 57	Verkkojulkaisulupa myön-netty: x
Työn nimi **Rajapintalähtöinen sovelluskehitys nykyaikaisilla työkaluilla** Rajapinnan ja pääteohjelman kehittäminen rajapintalähtöisesti		
Tutkinto-ohjelma Tietojenkäsittelyn tutkinto-ohjelma		
Työn ohjaaja(t) Niko Kiviaho		
Toimeksiantaja(t) Tieto Finland Oy		

Tiivistelmä

Ohjelmistorajapintoja hyödyntävät API-talouteen kuuluvat liiketoimintastrategiat tulevat jatkuvasti yleisemmiksi ohjelmistoalalla. Hyvin suunnitellut ja toteutetut rajapinnat ovat tärkeä askel kohti onnistunutta API-taloutta, minkä vuoksi opinnäytetyön toimeksiantaja oli kiinnostunut rajapintojen suunnittelutekniikka rajapintalähtöisestä sovelluskehityksestä.

Tutkimuksen tavoitteena oli selvittää, kuinka rajapintalähtöistä sovelluskehitystä voidaan toteuttaa käytännössä ja mitä hyötyjä siitä on, sekä löytää tehokas tapa käyttää sitä Microsoft Azure ympäristössä. Erilaisten rajapintojen kehittämiseen tarkoitettujen työkalujen ja standardien odotettiin tehostavan rajapintalähtöistä kehitystä. Tutkimus toteutettiin kehittämistutkimuksena, jossa rajapintalähtöistä sovelluskehitysprosessia testattiin iteratiivisesti arvioiden, kuinka eri työkalut ja standardit pystyvät tukemaan tätä prosessia Azure API Management palvelun kanssa.

Opinnäytetyön tulos oli toimiva rajapintalähtöinen sovelluskehitysprosessi, joka käyttää OpenAPI 2.0 standardia sekä Swagger Editor ja Swagger Codegen työkaluja. Tutkimus osoitti, että rajapintalähtöinen sovelluskehitys on hyödyllinen suunnittelutekniikka käyttää Azure ympäristössä ja että työkalujen käyttö rajapintalähtöisessä kehitysprosessissa tukee sitä erinomaisesti tehden siitä nopeamman ja tarkemman, sekä helpottaen tekniikan käyttöönottoa.

Avainsanat (asiasanat)
rajapinta, REST API, API-lähtöinen sovelluskehitys, API ekonomia, Azure API Management

Muut tiedot (salassa pidettävät liitteet)

Contents

Figures

Tables

Acronyms and terminology

API (Application Programming

Interface): Interface which enables communication between applications

APIM: API Management

CRUD operations: Create, read, update and delete

DX: Developer experience

HTTP protocol: Protocol used by browsers and web servers for transferring information

JSON: JavaScript Object Notation file format

REST: Architectural style for designing systems

REST API: APIs that use REST architecture

SDK: Software Development Kit

YAML: YAML Ain’t Markup Language data serialization language

1 Introduction

As API economy is a growing business nowadays, it is important to consider what kind of actions to take as an organization to partake in it. Finding a suitable method to building APIs as products and a right set of tools and platforms is the starting point of this goal. Starting to apply an API designing method called API-first design could be a step towards organization's own API ecosystem.

The subject of this research is API-first design and tools that can be used to support this design method. API-first design is a relatively new method for designing APIs and it has not been the subject of many researches yet. This research aims to open the meaning of API-first design and show its process in practice by building a demo API and application according to the method. It can act as an introduction material to API-first design as well as it evaluates tools regarding this design approach.

There are multiple tools that can help designing and developing APIs at every stage of their development. This thesis introduces an API defining standard and a pair of tools that are used to support API development in Azure cloud environment. They were chosen to fit API-first design in this environment.

2 Research

2.1 Purpose and assigner of thesis

The main objective of this research paper is to find out what is API-first design and what benefits does it have, and how to apply API-first design method in Azure API Management environment with third-party tools that make the process efficient. This research aims to find out how much of the API building process can be accelerated with the usage of appropriate tools.

To achieve this goal, this thesis consists of research about API-first design to understand how to apply the method, introducing suitable tools and standards to support API-first design and building a demo API and small client application using the approach. The demo is built for testing and introductory purposes only and is done by one person. Its documentation can be used as a guide to API-first design. The tools are evaluated regarding how well they support API-first design approach and if they can make its process efficient in Azure environment.

The research problem is rooted to the issue of combining manual API development with expanding documentation, which hinders the ability to quickly deploy new APIs. The thesis assigner wants to look at a possibility of developing APIs by using predefined tools and thereby minimizing the need for manual input while getting well-made APIs as a result. API-first design could be one solution to this problem, which is why it is the subject of this thesis.

The subject of this research was proposed by Tieto Finland Oy. Tieto's branch of industry, Tieto Education builds digital solutions for daycare and education. Tieto Education is interested in API economy and finding new agile methods for building and managing APIs. The demo API's subject is related to Tieto Educations' industry.

2.2 Research questions

This research aims to answer the following questions to solve the research problem:

1. What is API-first design?
2. What kind of tools and standards are there to support API-first design?
3. How can API-first design benefit API development in Azure environment efficiently?

2.3 Research methodology

The research methodology used in this thesis is design research (applied action research). Design research aims to solve a research problem or improve a current situation by developing practical solutions. What makes design research different compared to any development work that aims to improve or solve an issue is documenting the work as well as publishing and presenting it. (Kananen 2012, 42-44.)

Design research was chosen for this thesis because its goal is to improve the current situation. The thesis assigner wants to make their current way of building APIs faster and more efficient. This thesis follows an iterative pattern of design research, which means that when a solution to the research problem is tested, it is altered according to the results and retested (Kananen 2012, 52-53).

The iterations follow a structure that has four parts: start, design, implementation and evaluation, as demonstrated in Figure 1. These four steps are completed in each iteration with necessary changes in the tools and practices used. The starting point in the first iteration is to use a set of tools and standards the assigner of the thesis is interested in.

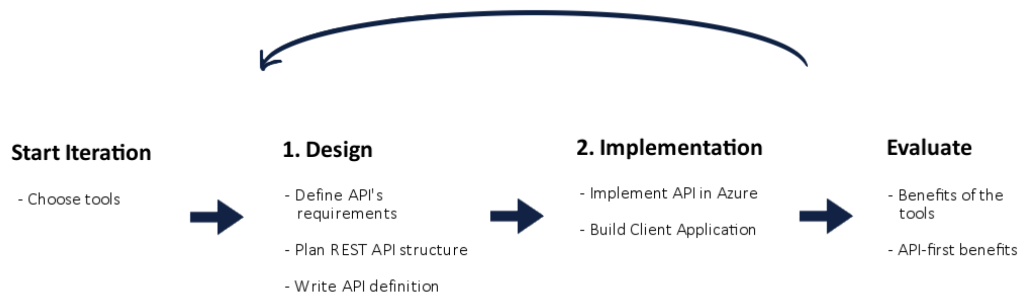


Figure 1. Structure of iterations

There are two main matters analyzed in each iteration: the benefits of the tools used in the iteration and overall benefits of API-first design.

The results of each iteration are evaluated by considering how well the used tools benefit API-first design process in Azure environment and if the tools can be used more efficiently compared to past iterations. The tools' benefits are evaluated by how they fit to Azure working environment and if they bring enough value to the process compared to the resources (time, money etc.) they take. Iterations are repeated until a satisfactory API-first process can be documented and presented as a result of this thesis.

This research also evaluates the benefits of API-first design. The results of iterations and the final process are compared to a list of benefits from the frame of reference and the benefits found inside this research. It is expected that not all benefits from the source material can be tested due to the nature of this research. Such benefits would be anything concerning teamwork, production environment and working with multiple development projects, which are not in the scope of this thesis.

3 APIs in Modern Development

An API (Application Programming Interface) is a connecting piece between applications or components that allows them to communicate with each other. APIs enable data to be retrieved, sent, altered and deleted through defined function calls from one application to another. Calling these functions makes it possible to use a program or its data in another application. Since APIs are interfaces called through applications and used by programmers, the end user of an application is never directly in touch with APIs. It is code the purpose of which is to be used by other web applications. Even though APIs are functionally always part of another program, nowadays APIs have become the focal point and a source of business on their own for many software companies. (Cater 2013; Rouse 2017.)

Traditional take on APIs is that they are only needed to support applications that have already been built and answer a need to use that application's data. More recently, APIs have become a notable part of business for two reasons: they offer more effective scalability of applications for different devices inside the company and make it possible to offer APIs as a product externally for other companies. Companies can have APIs as their main or even only product. (Robles 2014.)

3.1 API Economy

API economy is a business strategy where the goal is to financially benefit from APIs. API economy is a broad concept that includes building and using company's own internal APIs, building external APIs for others to use and using external APIs from another company. Each of these approaches benefits companies, either directly financially by bringing in a new source of revenue or new customers, or indirectly by improving the efficiency of development. API Economy is an evergrowing area of business that keeps gaining popularity. (Glickenhouse 2018; Urpilainen 2017.)

API economy is close to another concept called API monetization (Glickenhouse 2018). There are many business models that can be used in order to start profiting from APIs. It is important to choose the right model for one's API specifically based on the API's purpose and usage. (Glickenhouse 2017.)

APIs can be divided into three categories: Internal or private APIs, partner APIs and external APIs. All categories have their own differing means to bring benefit. APIs from each category can co-exist inside a company or even overlap to some extent, since external and partner APIs can be used internally as well. Another way of categorizing APIs is that they are divided into protected and open APIs. Protected APIs require user authorization, while open APIs are available for everybody. (Ewerlöf 2018.)

There are some differing opinions on how many API categories exist, as some experts on the subject define internal and private APIs as different concepts while others do not. According to Moilanen, Niinioja, Seppänen & Honkanen (2018, 73-74.), the difference between the two is that private APIs are used in public internet and internal APIs in an organization's internal network. (Moilanen et al. 2018, 73-75.)

Internal API

Internal and private APIs are used inside a company to develop its own services. They are usually used by developers or people working close to them. Internal APIs often handle data that is different from the type of data that is used externally, such as data that is not allowed to be shown to customers or competitors. (Moilanen et al. 2018, 73-77, 81-82.)

Partner API

Partner APIs are used by other organizations as well; however, only by the ones that have an agreement with the API provider. They are protected from users outside the agreement and are there to benefit all related parties. Partner APIs are protected for example by API or subscription keys.

External API

On the contrary to partner APIs, external APIs are open to use for everyone and often handle open data. External APIs can open new business prospects from two points of view: a company can either offer APIs externally for public use or use external APIs from outside source. External APIs offered for public use can be used inside the organization as well. It is even recommended to do so, because this way the API is updated more regularly, and issues are noticed and fixed faster. (Moilanen et al. 2018, 73-77, 93-95.)

Even though external APIs are available for everyone, it is possible to charge for their use depending on the level of usage and prioritize certain user groups to give them better service level. There can be different types of subscriptions and pricing levels. External APIs often use *Freemium* business model, where the basic level of usage is free; however, a priced premium subscription is required for professional use. (Moilanen et al. 2018, 94, 96.)

Sometimes the best option is to use an external API made by someone else. It is time efficient and prevents having to do the same work another API provider has already done. Using external APIs made by other companies comes with a risk of deprecation of that API. (Moilanen et al 2018, 73-77, 93-95.; Vasudevan 2017b.)

The number of public APIs is constantly growing. There are API directories such as *Programmableweb.com*, which provide a platform to publish and find external APIs. (Moilanen et al 2018, 94, 102.)

3.2 API management platforms

API management platforms contain features for monitoring and managing APIs. Usually they include tools for deploying and publishing APIs, and restricting, documenting and versioning them. In addition to these most common management features, API management platforms can have other tools such as management of users, pricing and policies. (Moilanen et al. 2018, 211-212.)

An API management platform can also have API gateway for managing traffic and a developer portal that has the API’s documentation (Moilanen et al. 2018, 212).

4 API-First Design

This chapter explains the importance of investing effort into designing APIs and introduces the API-first design method.

4.1 API Design

Designing an API means making deliberate decisions about different aspects of an API's structure and its development process. API design aims to improve the quality of APIs, which makes them easier and more pleasant for developers to use and reduces mistakes. Designing an API affects directly its structure, documentation and syntax, and through these physical features many other aspects of API development and usage indirectly. An API's design can be described as the blueprint of one's API. It is the foundation that helps to create successful APIs. (API Design – API-University N.d.; Vasudevan 2016b.) Designing APIs takes effort to learn and put into practice; nevertheless, it is worthwhile because well-designed APIs and people with the skills to create them can help to grow a functioning API ecosystem of internal, partner and external APIs. (Patni 2017.)

Good API design aims to give APIs certain characteristics. According to Vasudevan (2016a), three desirable traits for APIs are being *easy to read and work with*, *hard to misuse* and *complete and concise*. Developer experience, also called DX for short, is an important concept behind gaining popularity for APIs; APIs that have a good developer experience will be used and liked amongst developers. Designing one's APIs well has a significant affect on their developer experience. (Vasudevan 2016b.)

According to Patni (2017), there are four overall strategies for designing APIs:

- **Bolt-on strategy:** This method is used when an application and a back-end system exist before an API's development has started. A new API is added in between the systems afterwards. With this design approach, existing systems can be used to help design and build an API. It is considered the fastest way to get a working solution, however, it does not have the best requisites to give the API a good design (Taman 2019).

- **Greenfield strategy:** When there are no existing applications or systems, designing the API starts from scratch without any underlying technologies. This strategy makes using completely new technologies possible due to starting from the very beginning.

- **Agile strategy:** Iterative agile development approach to API development that allows changes on a fast pace. It follows the idea that development can start before having all specifications ready. Agile strategy is recommended for use only until the API is published.

- **Façade strategy:** It has elements from both Bolt-on and Greenfield strategies. This strategy is usually used in companies where systems exist already; however, Greenfield approach is used in API development.

When starting the design process of an API, it is important to figure out the data that should be exposed, the best way to expose it and how the API can be improved. (Patni 2017.)

Different organizations can have their own API design guidelines that help giving all of their APIs a consistent design in terms of structure and quality throughout the company. (Moilanen et al. 2018, 145.)

Best practices for REST API design

There are best practices for designing REST APIs that are worth taking into account when writing new APIs. These practices focus on making REST APIs clean and consistent, while using REST API's stucture to its benefit.

- Resource names should be in plural.
- Resource names should be nouns and there should not be any verbs.
- HTTP methods should be used accordingly and used to describe action.
- Sub-resources should be used when resources are connected to eachother.
- HTTP status codes should return proper feedback.
- APIs should be versioned. (Jauker 2014.)

4.2 What is API-First Design?

API-first design is one method for designing and developing APIs. When following this method, an API's development starts from the design phase. In practice, it means that the API's functionality is first planned and described in a standardized format and the code to implement it is built according to that plan. An API's lifecycle always starts from a business need that an API can answer. Following the design-first approach, the next step is to make an API definition in a standardized machine and human readable format that answers that need. This method for developing APIs is relatively new but it is rapidly becoming more common to use. (Moilanen et al. 2018, 148-149; Vasudevan 2017a.) Riggins (2015) declares that “API-first design is about a series of best practices […] that prioritize a better developer experience”.

API-first design is also referred to as API-driven-development, API design-first or schema-first development (Rosenstock 2018).

API-first design can either be defined as an approach to only developing APIs or as an approach to developing applications as well. The difference between the two ways of looking at API-first design is that the latter one underlines that the API should be implemented before building an application. It is the desirable situation for using API-first design, because the method has the most benefits if a new application is built on top of the API. The other outlook focuses only on API development and does not include application development into API-first design’s definition. It is possible to use API-first approach even if there is an existing application or functionality, since making an API definition before implementation is still a beneficial way of developing APIs. (Levin 2016; Riggins 2015; Vasudevan 2017a.)

API-first design falls under the Greenfield design strategy, where the design process begins without underlying technologies. This strategy is based on simulating the APIs. Developers can start working on a new application that uses an API by developing against a mock of that API. This means the API is usable before a finished API exists. (Patni 2017.)

The core ideas behind API-first design are that APIs are built for their users who are developers and that APIs should be seen as an independent entity (Trieloff 2017). API-first puts emphasis on the idea that an APIs design require a lot of attention, because creating a good developer experience is a priority. In the beginning of development, creating a good design starts with ignoring existing systems if there are some, and focusing on what is needed from the API. (Riggins 2015.)

According to Trieloff (2017), there are three API-first principles:

- “Your API is the first user interface of your application.”

- “Your API comes first, then the implementation.”

- “Your API is described (and maybe even self-descriptive).”

API-first design is an effective way to end up with APIs that are pleasant to work with. Going through the process drives one to design and test one’s API properly, which makes it easier and more understandable for developers, and it is more likely to be as suitable as possible for the purpose it is needed for. Building developer-friendly APIs is not only done for the sake of developer experience but it also has great benefits for the company that is taking on this approach. (Levin 2016.)

4.3 API-First process

There is no standardized API-first process as it can be adjusted to fit a company’s own procedures, as the only important common factor is making API specification before implementation. However, the lack of detailed guides makes it much more difficult to understand how API-first design process works in practice and start using it in one’s own work. This chapter presents the process described by Riggins (2015) in six API-first steps, which is supplemented with information about said steps from other sources.

Step 1: Planning

API-first design starts with planning. Planning phase should provide answers to questions *why* and *who*. Why is this API needed and who are its stakeholders and consumers? (Santos 2016.)

According to Moilanen et al. (2018, 137-138) creating user stories is a good method to use when starting to design any APIs. User stories help the people with creating APIs to figure out API’s business requirements and get a clear idea of what is being developed. It also lays out the means to communicate the plan to other stakeholders through them.

Step 2: Design and validate

This step covers designing the API according to the plan from Step 1. One should take a few hours to design the API. Riggins (2015) recommends that already at this point there should be a tool that enables testing API requests before any implementations.

Step 3: Lock down the API specification

At this point, an API specification is made according to the design and should be locked down, so it can be implemented and shown as the final API for everyone involved in the development, including stakeholders and developers. It is recommended that the specification should stay the same for at least a few months. (Riggins 2015.)

Step 4: Test

The API is tested at this step. Its functionalities, consistency and developer experience should be issues considered when committing tests. (Riggins 2015.)

Step 5: Implementation

Implement the API to make it accessible for stakeholders to try it. (Riggins 2015.)

Step 6: Operate and engage

At this step the API is deployed, and the clients should be able to give feedback about it. (Riggins 2015.)

4.4 Benefits of API-First approach

Many experts on the subject recommend API-first method, because there are various benefits in using it (Rosenstock 2018). The method's benefits affect different aspects of development and business.

Reusable APIs

API-first design approach aims to cover every needed functionality in an application with an API call. When this principle is followed, there are two positive outcomes: being able to use these same API calls in web, mobile and tablet development for one application, and also reuse them in another applications. Reusable APIs can lessen the developers' workload. (Jaswal 2017; Santos 2016.)

Jaswal (2017) explains the power of reusing APIs in different applications in his article *Why the phrase 'API-first' should be at the heart of every digital experience* in Digital Pulse online publication by describing a hypothetic situation, where a bank wants to make an application that allows people to send money to their relatives in another country.

> *The APIs that make this [application] work are plentiful. One would be the address-book API, linking the user to the people they transact with. Another could be an exchange-rate API to convert one currency to another. An API could be needed to authenticate the customer, say by checking their PIN number matches the account. Then finally, one more API could move the actual money. – Jaswal 2017*

After describing what kind of APIs this said bank application would need, Jaswal (2017) explains how the bank could use two of these old APIs in a new application that people could use to take money from an ATM in a foreign country.

API Documentation

API-first design starts with writing the API's definition in a standardized format. An important part of API-first is that making a definition of the API also creates documentation for it in the process. An API definition can be the documentation in itself; however, there are also many tools for the purpose of generating more understandable documentation automatically according to API definitions. (Vasudevan 2016b; Viswanathan 2017.)

Consistency of the APIs

API-first helps to create consistent APIs throughout the company. The consistency of APIs means that their structure and documentation follow the same guidelines. All of a company's APIs can be created by making an API definition first and following the same standard and practices. Reusing, planning, documentation and writing API definitions all affect consistency. Consistency is important because the usage of APIs is easier to learn if they're similar to each other. (Vasudevan 2017a.)

Creating a better developer experience

Developer experience is one of the key aspects that API-first design affects. It is a sum of many of the benefits that API-first design has that together create a good DX. In short, developer experience in the context of APIs means how easy it is to develop applications against an API.

Due to APIs being designed and tested with vast care throughout API-first design, it affects APIs' developer experience. Developing an application against an API with good DX can save plenty of time during development, and such an API is more likely to become widely used, get good feedback from developers and get them to use it for a longer time. (Jarman 2017; Patni 2017; Vasudevan 2016b.)

Reduce deferring dependencies between teams

Developer teams in charge of developing different parts of the project do not have to wait for each other to finish, to start or continue working. API-first approach makes it possible for front-end, back-end and test teams to work simultaniously after an API definition and a mock API have been made as visualized in Figure 2. Front-end developers can build their application against a mock API while back-end team(s) are building their implementation. This also applies to working with different clients, such as desktop and mobile implementation that use the same API. (Levin 2016.)

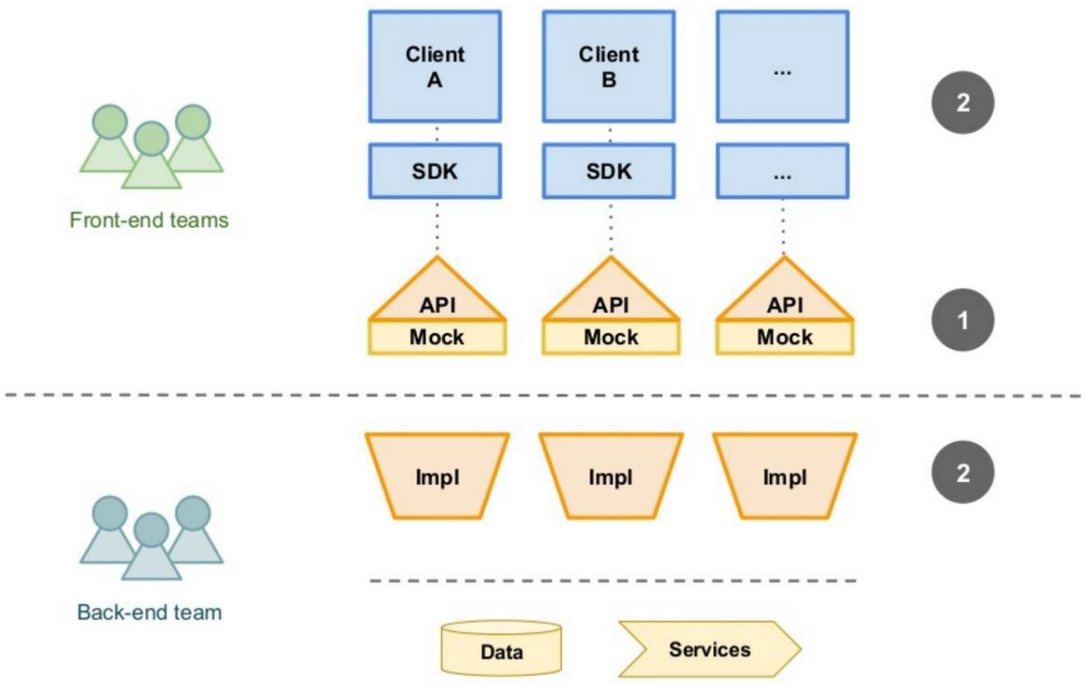


Figure 2. Different developer teams working simultaniously against mocked APIs

Make development and delivery faster

Development process is faster due to the previously mentioned paraller development and reusability (Levin 2016). The beginning of starting to use API-first design method, development might be slower than development without careful designing. It takes time to learn to design APIs, create API definitions and use API-first method but it saves time in a long run.

Collaboration

API-first approach enables getting feedback and comments of the API under work at a very early state (Moilanen et al. 2018, 136). An API's definition can and should be shared with stakeholders from early on as a part of making the design as good as possible. This makes it possible to get valuable feedback before any code has been written and changes to the API are easier to commit and cost less resources to make. (Levin 2016; Santos 2016.)

Since the API definition is human readable and can be presented in a very simple, visual way by using a tool for it, understanding the API definition does not require technical skills such as programming knowledge. This makes the API definition accessible to all stakeholders. (Pinkham 2017.)

Encourages to think past legacy systems

The goal in API-first design is to start designing without concidering legacy systems in the beginning. This will result in APIs designed for the actual need they are supposed to fulfill and not for what seems to be possible when taking legacy systems into account. Legacy systems should become a factor in design decisions only after the initial design is made, and it is to be decided whether the old system or the new design can be compromised. (Riggins 2015.)

API-First fits API economy

As a benefit itself and also a cause of the other benefits listed above, API-first is a good way to develop APIs that become a part of API economy. To succeed in API economy, a company needs well made, fastly implemented and documented APIs, which is exactly what API-first design aims for. (Viswanathan 2017.)

5 Tools and standards to support API-First approach

One of the main objectives of this thesis is to present, try out and evaluate suitable tools and standards that help building APIs using the API-first approach efficiently. The following standard and tools presented in this chapter are used in the demonstration of API-first design in Chapter 6. They are chosen to fit the scope of the work and are tools that the thesis assigner is interested in.

5.1 Role of tools and standards in API-First Design

API-first design emphasizes that designing APIs takes time and effort, and ultimately the end result is worth those resources used. However, API-first process can be made more efficient with the usage of the right tools, which saves time without sacrificing quality.

There are tools and standards available dedicated to all phases of API development: designing, building, testing and deploying APIs. Using tools will help the process of designing and make using the API-first method more accessible as well as make errors less likely to occur during the process (Rosenstock 2018).

Using the right tools, it is possible to design and test the new API's functionality without any programming and dependencies on certain programming languages, platforms or applications (Moilanen et al. 2018, 136). The tools automate the process and make it easy to iterate, which produces the best results.

5.2 OpenAPI Specification

OpenAPI Specification (OAS) is a standard for defining RESTful APIs in a manner that makes them understandable for both humans and machines. The files made in this format can be read and used by multiple tools that help to design, build and manage APIs and they also provide documentation for developers. OpenAPI has two versions, OpenAPI 2.0 and the latest OpenAPI 3.0 released in 2017. The specification is owned by The OpenAPI Initiative. (Open API Specification N.d.; Mackory 2018.) OpenAPI specification can be written in YAML or JSON format (Basic Structure | Swagger N.d.)

This research uses OpenAPI 2.0 in development of the demonstrative API, because Azure API Management does not support version 3.0 at the time of this research. Microsoft is in the process of adding support for OpenAPI 3.0 to Azure. (OpenAPI v3 support in Azure API Management 2018.)

5.3 Swagger

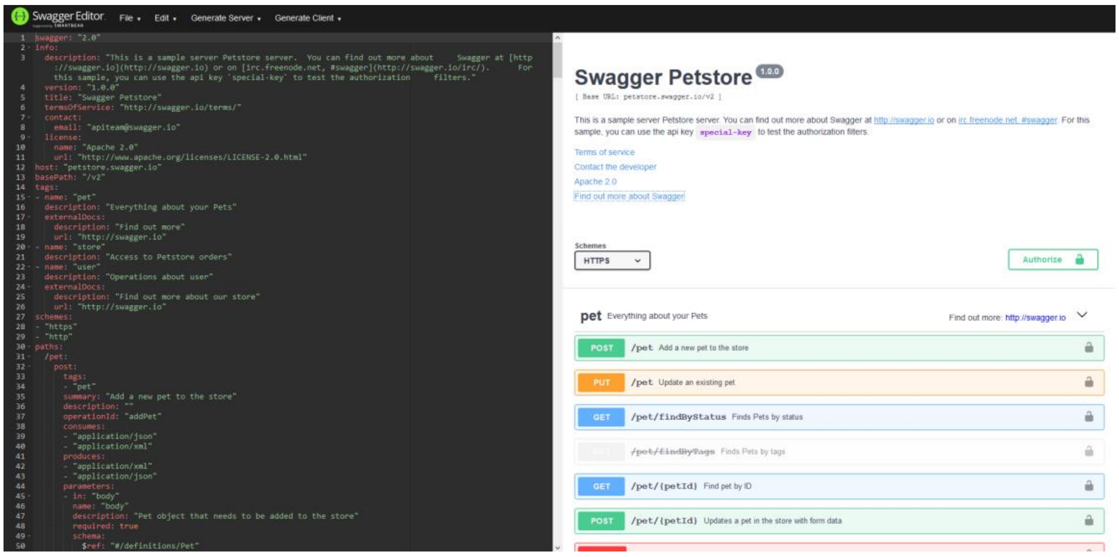
Swagger is a collection of API developer tools for every phase of API development from design to deployment. Swagger is a part of SmartBear Software that specializes in development tools and has Open source, free and commercial tools available. Swagger is the original creator of the OpenAPI specifications, and all of Swagger's products are built to support OpenAPI. (About | Swagger N.d.)

There are two Swagger products used in this research, *Swagger Editor* and *Swagger Codegen*.

Swagger Editor

Swagger Editor is an open source editor tool for OpenAPI specifications. With it, it is possible to write Open API specifications for APIs, while getting instant feedback and visual description of the API. Swagger Editor supports writing OpenAPI definitions in both YAML and JSON format. (Swagger Editor N.d.)

Swagger Editor is available for all development environments and can be used both locally and online. It is an editor tool showing a visual documentation in the side for current specification dynamically as shown in Figure 3. It also gives an error message if there is a syntax error with the OpenAPI specification that is being written as seen in Figure 4. It also offers auto-completion for faster writing and allows quick import and saving of API descriptions, as well as server stub and client generation in different programming languages. (Swagger Editor N.d.)



```yaml
swagger: "2.0"
info:
  description: "This is a sample server Petstore server.  You can find out more about     Swagger at [http
    ://swagger.io](http://swagger.io) or on [irc.freenode.net, #swagger](http://swagger.io/irc/).      For
    this sample, you can use the api key `special-key` to test the authorization     filters."
  version: "1.0.0"
  title: "Swagger Petstore"
  termsOfService: "http://swagger.io/terms/"
  contact:
    email: "apiteam@swagger.io"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2.0.html"
host: "petstore.swagger.io"
basePath: "/v2"
tags:
- name: "pet"
  description: "Everything about your Pets"
  externalDocs:
    description: "Find out more"
    url: "http://swagger.io"
- name: "store"
  description: "Access to Petstore orders"
- name: "user"
  description: "Operations about user"
  externalDocs:
    description: "Find out more about our store"
    url: "http://swagger.io"
schemes:
- "https"
- "http"
paths:
  /pet:
    post:
      tags:
      - "pet"
      summary: "Add a new pet to the store"
      description: ""
      operationId: "addPet"
      consumes:
      - "application/json"
      - "application/xml"
      produces:
      - "application/xml"
      - "application/json"
      parameters:
      - in: "body"
        name: "body"
        description: "Pet object that needs to be added to the store"
        required: true
        schema:
          $ref: "#/definitions/Pet"
```

Swagger Petstore 1.0.0

[Base URL: petstore.swagger.io/v2]

This is a sample server Petstore server. You can find out more about Swagger at http://swagger.io or on irc.freenode.net, #swagger. For this sample, you can use the api key `special-key` to test the authorization filters.

Terms of service
Contact the developer
Apache 2.0
Find out more about Swagger

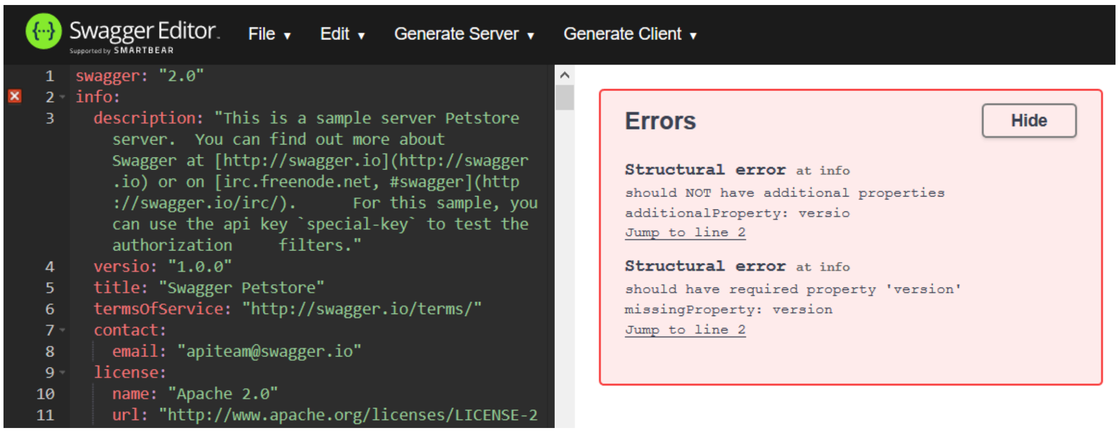
Schemes: HTTPS

Authorize

pet Everything about your Pets — Find out more: http://swagger.io

POST /pet Add a new pet to the store
PUT /pet Update an existing pet
GET /pet/findByStatus Finds Pets by status
~~/pet/findByTags~~ Finds Pets by tags
GET /pet/{petId} Find pet by ID
POST /pet/{petId} Updates a pet in the store with form data

Figure 3. Swagger Editor's online user interface with an example API



```yaml
swagger: "2.0"
info:
  description: "This is a sample server Petstore
    server.  You can find out more about
    Swagger at [http://swagger.io](http://swagger
    .io) or on [irc.freenode.net, #swagger](http
    ://swagger.io/irc/).      For this sample, you
    can use the api key `special-key` to test the
    authorization     filters."
  versio: "1.0.0"
  title: "Swagger Petstore"
  termsOfService: "http://swagger.io/terms/"
  contact:
    email: "apiteam@swagger.io"
  license:
    name: "Apache 2.0"
    url: "http://www.apache.org/licenses/LICENSE-2
```

Errors

```
Structural error at info
should NOT have additional properties
additionalProperty: versio
Jump to line 2

Structural error at info
should have required property 'version'
missingProperty: version
Jump to line 2
```

Figure 4. Example of an error message in Swagger Editor

Swagger Codegen

Swagger Codegen is an open source tool that uses OpenAPI specifications to generate client SDKs and server code stubs. It is available for download from its GitHub repository. Swagger Codegen can generate code in over 40 programming languages. (Download Swagger Codegen. N.d.; Swagger Codegen N.d.)

Swagger Codegen is used in a local environment and its operations are accessed through command line.

5.4 Azure API Management

Azure API Management is Microsoft's cloud-based API management platform. It is part of Microsoft Azure, which has many cloud services from different categories of software development. Azure API Management is a place where APIs can be deployed. (API Management: Establish API Gateways N.d.; What is Azure – Microsoft Cloud Services N.d.)

Azure API Management consists of three products: API gateway, Azure portal and Developer portal (Vinogradsky 2018).

API gateway is the endpoint that handles the traffic related to the API. (Vinogradsky 2018.)

Azure portal is an administrative platform that can be used to manage all kinds of applications in one place. In the context of this thesis, Azure is used to set up the API. In the portal, it is possible to set up an instance of API management service. Using that service, API schemas can be defined and managed. (Microsoft Azure Portal N.d.)

The developer portal is a separate page for an API management service with useful functionalities for developers. It has documentation for APIs deployed in said API management service, a console for testing APIs, a system for account management and information about API's analytics. (Vinogradsky 2018.)

6 Building API and Client App with API-First Design method

This chapter goes through the process of designing and implementing a new API and building a small client application that uses the API by using API-first design method. It is based on API-first process steps presented in Chapter 4.3, with some changes due to getting more valuable result with Azure API management. This process is the final result after multiple iterations of improving it.

The new API's subject is *students' absences from school*, which are added to individual students when they are absent from school for the day. The API will be referred to as the Absences API.

6.1 API’s subject and requirements

The requirements for Absences API are as simple as possible because it is a demo API that needs to be understandable for example’s sake. The goal is to have a simple API that does not have more than a few API endpoints and its requirements are chosen with that in mind.

The API’s subject is adding absences to students. Authorized users should be able to commit operations related to *absence entries*, which are information about a student’s daily participation at school. These operations are viewing absences of different students, adding new absence entries and deleting them.

The basic requirements for the Absences API from a REST API point of view are that it should be possible to *read*, *add* and *delete* absences for students. Additionally, information about students is editable and it should be possible to add and remove individual students. Absence entries are added for one whole day and they are the only information needed about a student’s attendance. Attending school does not require any entries.

Absences API’s users are *teachers*, *parents* and *administrators*, namely the people in charge of maintaining absences and information about students.

Information that the API needs to get to is *students* and their *absences*. All absences entries must belong to a student. Students must include general information such as their name, and an individual ID. The absences must have a date and a reason for being absent.

6.2 Designing the new API

Planning

The first thing to do in the planning stage is to define what the API needs to achieve and how it can meet the business requirements. In the beginning, the plan should be as simple as possible, as it can be expanded later (Moilanen et al. 2018, 136).

A recommended approach is to create user stories from the requirements. When creating user stories for APIs, it is important to focus on the users of the API, not the

API itself. For example, “as an API, I want to be able to fetch a student’s absence information” is not the right way of making user stories; they should start with “as a certain type of user”. User stories help developers to understand how the users are going to use the product and make better design decisions based on the user stories. (Jarrel 2018.)

User stories for this API are:

- As a teacher, I want to be able to add an absence to a student for a certain date, so I can keep track of every students’ absences
- As a teacher, I want to be able to delete individual absences from a student, so I can remove faulty absences
- As a parent, I want to be able to add an absence for my child, so the teacher can know my child is not coming to school on certain day
- As an administrator, I want to be able to add and delete students and change their information, so the information about current students is up to date

Starting from the simplest possibility, there needs to be a REST API for *students* that can have *absences*. The requirement is to have only one absence for one day, so absences are distinguished by date.

Absences API should follow REST API best practices. The resource names (e.g. student) should be in plural, because that resource can have multiple instances of the same resource. Individual resources inside a path are identifiable by a unique value they have (for example *studentId* in this case). The final structure of the REST API paths are written down (Table 1). This structure is what the OpenAPI definition is based on. Writing an API definition is easier and more straight-forward when there is a written plan to follow, even if the final definition can change during the test phase.

Table 1. REST API paths for Absences API

Path
/students
/students/{studentId}
/students/{studentId}/absences
/students/{studentId}/absences/{date}

After deciding the REST API structure, the next step is to choose CRUD operations for each path. There should be operations to do all necessary calls needed in an application; however, no redundant operations. A redundant operation in this API would be for example a *delete* operation to every absence from one student, because deleting every absence of a student should not be a possibility according to the API's business requirements. The CRUD operations chosen for Absences API are shown in Table 2.

Table 2. Operations for each path

Path	Operations
students	post, get
students/{studentId}	get, put, delete
students/{studentId}/absences	post, get
students/{studentId}/absences/{date}	get, put, delete

The chosen REST structure and operations should make previously made user stories possible to commit. Testing user stories:

- As a teacher, I want to be able to add an absence to a student, so I can keep track of every students' absences
  - ➔ Possible with *post:/students/{studentId}/absences*
- As a teacher, I want to be able to delete individual absences from a student, so I can remove faulty absences
  - ➔ Possible with *delete:/students/{studentId}/absences/{date}*

- As a parent, I want to be able to add an absence for my child, so the teacher can know my child is not coming to school on certain day
  - ➔ Possible with *post:/students/{studentId}/absences*
- As an administrator, I want to be able to add and delete students and change their information, so the information about current students is up to date
  - ➔ Possible with *post:/students*, *delete:/students/{studentId}* and *put:/students/{studentId}*

Already at this point of the design process, people making the API can communicate this API's design plan to other stakeholders to get feedback.

Making API Definition

When the requirements needed at this point, structure and CRUD operations, are defined, the next step in the API-first process is to make an OpenAPI Specification based on them. In this case it is done in the online version of Swagger Editor. The online version was chosen because it is able to fulfill the requirements for writing an API definition, and it does not require the process of installing a new software.

The OpenAPI definition follows a structure that covers a whole REST API. In this example, the definition is written in YAML format, and an older version of the definition, OpenAPI 2.0, is used due to compatibility with Azure APIM. The next part shows the steps to create an OpenAPI definition for Absences API and details about each part.

1. Metadata

Metadata needed for OpenAPI specification is OpenAPI's version that is used in the definition and information about the API that covers its title, version and description.

```
swagger: '2.0'
info:
  title: Absences
  version: '1.0'
  description: Absences for students API
```

2. Base URL

Define host, paths and schemes. API’s host URL is now defined with a placeholder text because it does not exist yet, and it will be updated to the definition once Azure API management service with the actual URL is running.

```
host: placeholder
basePath: /api/1.0
schemes:
  - http
  - https
```

3. Paths

Each individual API endpoint is defined separately under a section called *paths*. Each endpoint has its HTTP methods specified. The HTTP methods have some additional information about them that helps to describe and differentiate them. This information is an important piece in creating documentations based on OpenAPI definitions. In this example, */students* path and its HTTP methods, *post* and *get* are defined in a following way:

```
paths:
  /students:
    post:
      summary: Add a new student
      description: Add a new student
      operationId: addStudent
      consumes:
        - application/json
      produces:
        - application/json
    get:
      summary: Lists all students
      description: Lists all students
      operationId: listStudents
      produces:
        - application/json
```

4. Parameters

HTTP methods can have parameters, which is data that can passed in four different ways: via URL path, query string, header or body. The *post:/students* operation’s parameter is passed in request body and it refers to a definition called Student, which is common for the whole API.

```
parameters:
    - name: "body"
      in: body
      schema:
          $ref: '#/definitions/Student'
```

```
                required: true
                description: Student that will be added
```

5. Responses

Responses to API calls are added under each HTTP method’s definition. These are the responses for */students* path’s *post* function:

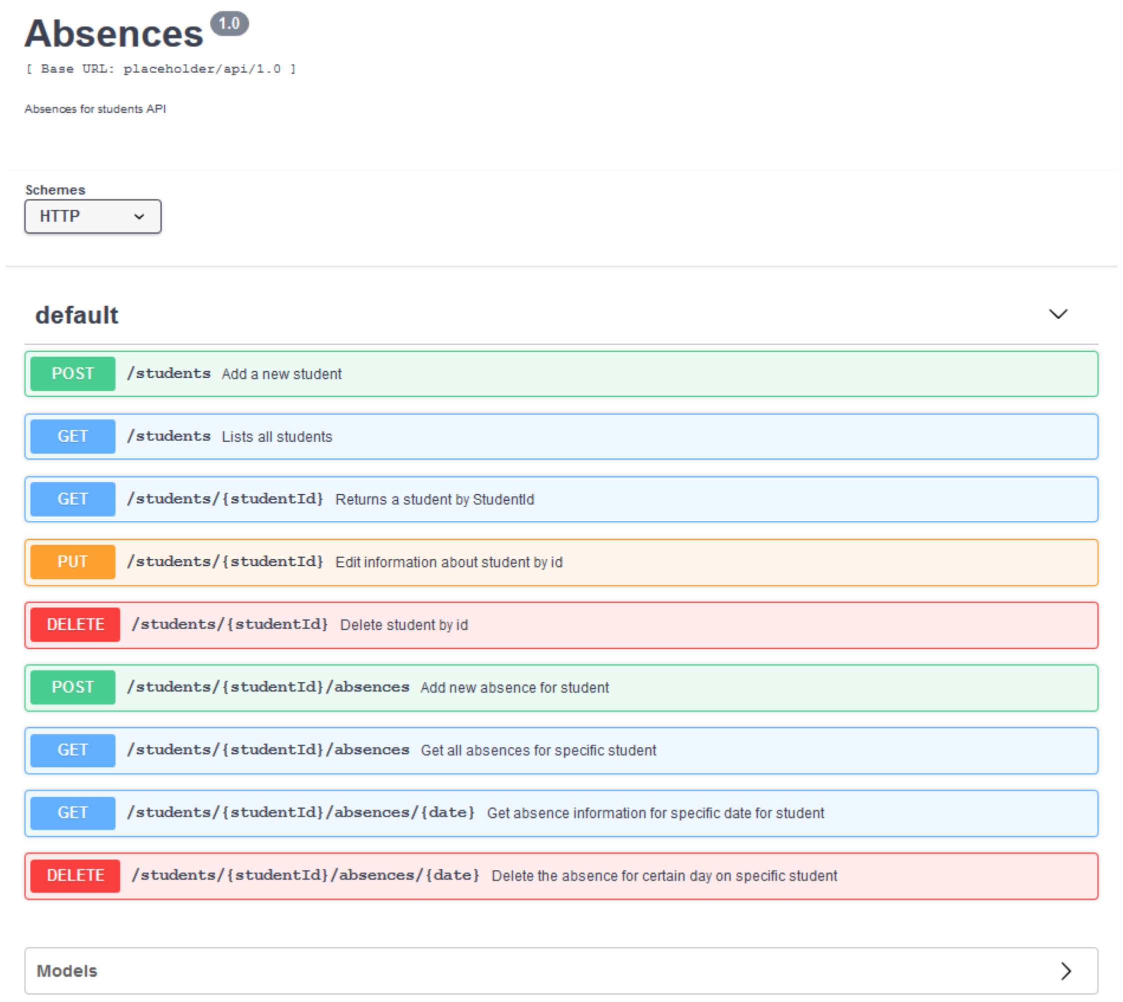
```
            responses:
                  '200':
                      description: OK - student added
                      schema:
                        $ref: "#/definitions/Student"
                  '500':
                      description: Unexpected error
                      schema:
                        $ref: '#/definitions/Error'
```

6. Models

Models are defined data structures that are common for the API.

```
            definitions:
              Student:
                type: object
                properties:
                  studentId:
                    type: integer
                    format: int64
                    example: '1234567890'
                  firstName:
                    type: string
                    example: Bob
                  lastName:
                    type: string
                    example: Smith
                required:
                  - studentId
```

Swagger Editor visually shows the structure all the time and updates it dynamically as illustrated in Figure 5.


Absences 1.0

[Base URL: placeholder/api/1.0]

Absences for students API

Schemes: HTTP

default

- POST `/students` Add a new student
- GET `/students` Lists all students
- GET `/students/{studentId}` Returns a student by StudentId
- PUT `/students/{studentId}` Edit information about student by id
- DELETE `/students/{studentId}` Delete student by id
- POST `/students/{studentId}/absences` Add new absence for student
- GET `/students/{studentId}/absences` Get all absences for specific student
- GET `/students/{studentId}/absences/{date}` Get absence information for specific date for student
- DELETE `/students/{studentId}/absences/{date}` Delete the absence for certain day on specific student

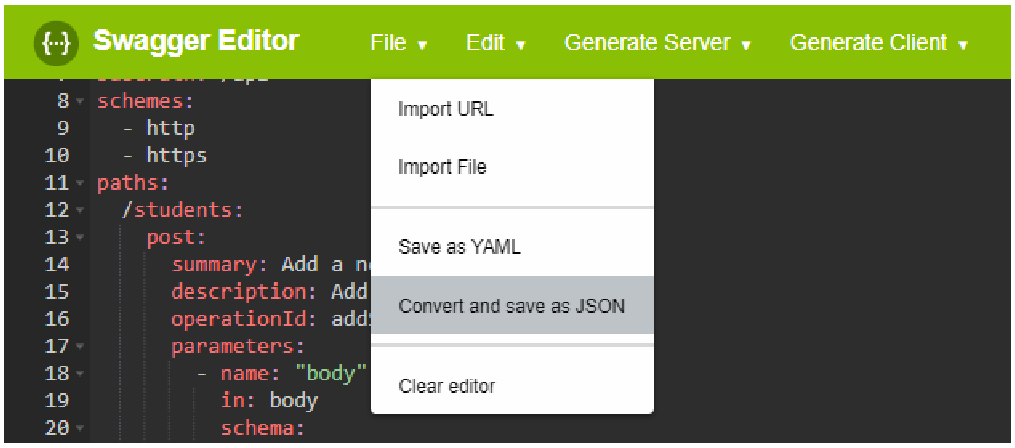
Models

Figure 5. Dynamic visual API documentation of Absences API in Swagger Editor

The finished API definition (Appendix 1) has all the needed parts for an OpenAPI definition, and it is done according to the previously chosen REST structure as seen in Table 2.

Download OpenAPI specification for Azure API Management

When the OpenAPI specification is tested and ready to be locked down, it can be downloaded through Swagger Editor. Azure API Management accepts only JSON format, so the definition file can be downloaded in that format by pressing “convert and save as JSON” as illustrated in Figure 6.



```
schemes:
  - http
  - https
paths:
  /students:
    post:
      summary: Add a n
      description: Add
      operationId: add
      parameters:
        - name: "body"
          in: body
          schema:
```

Figure 6. Converting OpenAPI specification to JSON in Swagger Editor

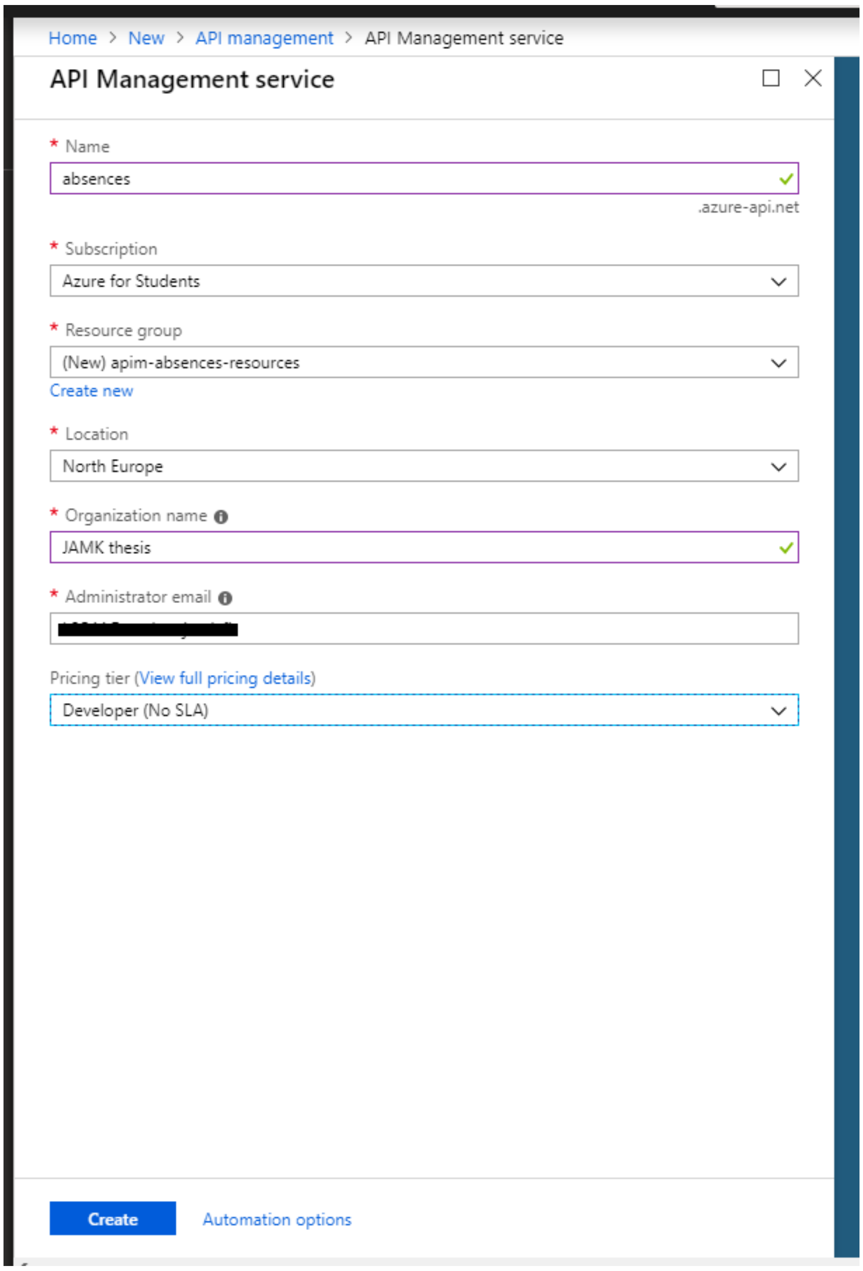
6.3 Implementing the API in Azure

This chapter goes through the implementation of the API in Azure and discusses what kind of actions are to be taken in Azure to benefit API-first development.

Creating a new Azure API Management service instance

An API Management (APIM in short) service is needed to start serving APIs in Azure Portal. Absences API will be added under the API Management service that is going to host it.

A new APIM service is created by going to “Create a resource” in Microsoft Azure Portal and finding API management service in Azure Marketplace. Creating a new APIM service requires adding general information about the new service as described in Figure 7.



Home > New > API management > API Management service

API Management service

* Name
absences
.azure-api.net

* Subscription
Azure for Students

* Resource group
(New) apim-absences-resources
Create new

* Location
North Europe

* Organization name
JAMK thesis

* Administrator email
[redacted]

Pricing tier (View full pricing details)
Developer (No SLA)

Create   Automation options

Figure 7. Creating a new API Management service

Import OpenAPI specs to Azure

When the API management service is made and active, it is possible to add new APIs there. In this case, the OpenAPI specification made previously will be imported by adding a new API from OpenAPI specification option as shown in Figure 8.

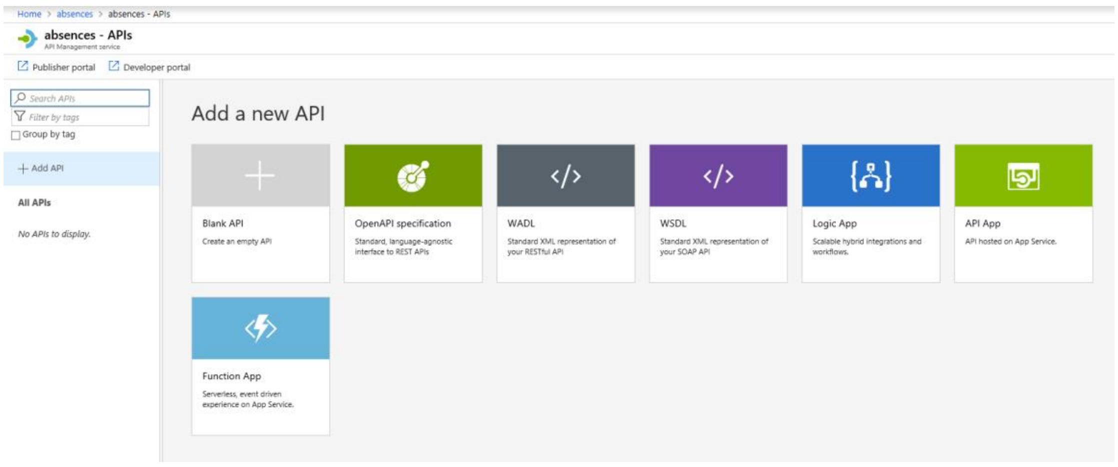
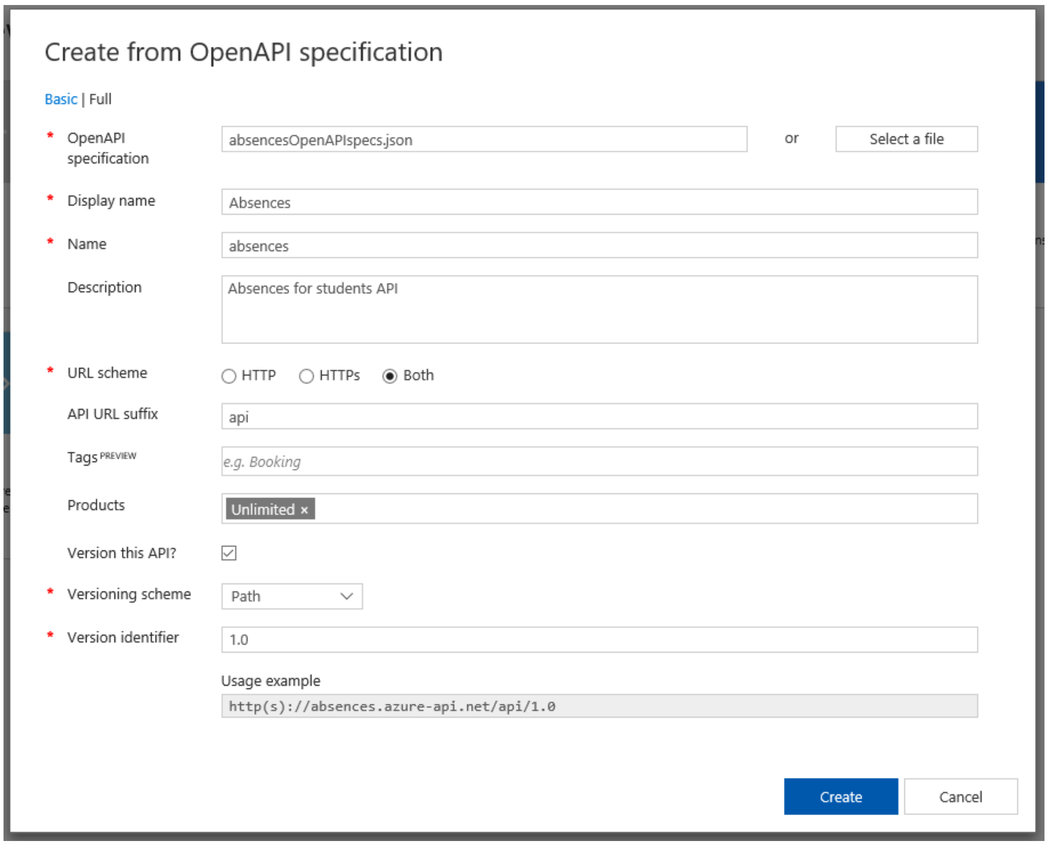

Home > absences > absences - APIs
absences - APIs
API Management service
Publisher portal
Developer portal
Search APIs
Filter by tags
Group by tag
+ Add API
All APIs
No APIs to display.
Add a new API
Blank API
Create an empty API
OpenAPI specification
Standard, language-agnostic interface to REST APIs
WADL
Standard XML representation of your RESTful API
WSDL
Standard XML representation of your SOAP API
Logic App
Scalable hybrid integrations and workflows.
API App
API hosted on App Service.
Function App
Serverless, event driven experience on App Service.

Figure 8. Adding new API by importing OpenAPI specification

New API needs to be given information such as the name of the API as shown in Figure 9. All the other information that is needed to set up an API comes from the OpenAPI specification. All of the API calls, paths and other necessary information is added to the API management service according to the specification automatically.



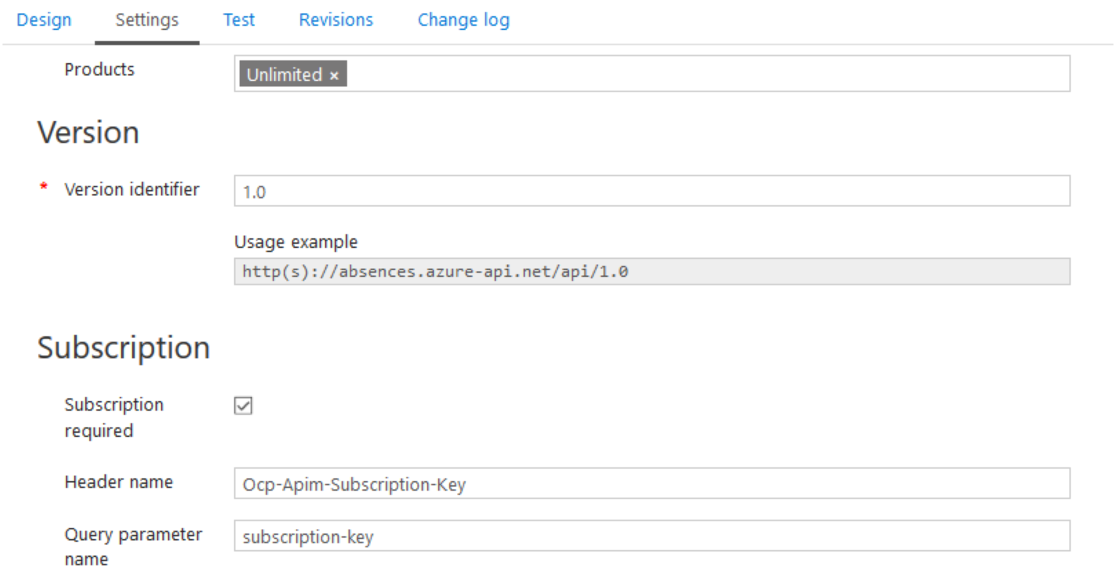
Create from OpenAPI specification

Basic | Full

* OpenAPI specification	absencesOpenAPIspecs.json or Select a file
* Display name	Absences
* Name	absences
Description	Absences for students API
* URL scheme	HTTP [✓ Both] HTTPs
API URL suffix	api
Tags PREVIEW	e.g. Booking
Products	Unlimited ×
Version this API?	✓
* Versioning scheme	Path
* Version identifier	1.0
Usage example	http(s)://absences.azure-api.net/api/1.0

Figure 9. Creating new API from OpenAPI specification

After the API has been created, *Subscription required* option can be checked to use a subscription key to secure access to APIs in Azure API management as demonstrated in Figure 10. A valid subscription key is required in any HTTP requests made to APIs with this setting checked.



Products	Unlimited ×

Version

* Version identifier	1.0
	Usage example: `http(s)://absences.azure-api.net/api/1.0`

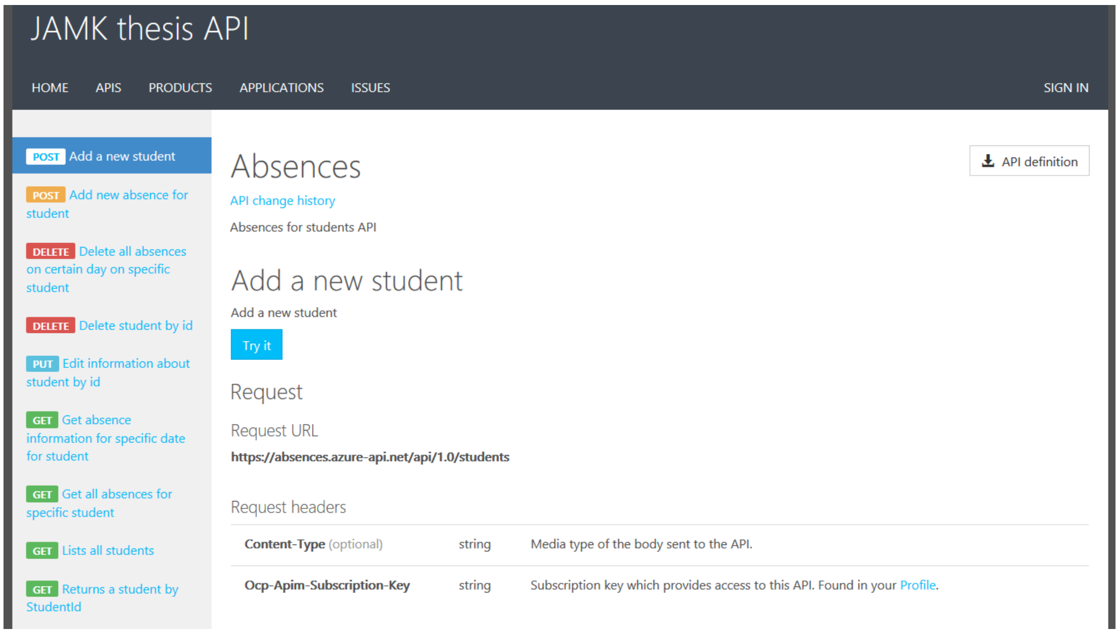
Subscription

Subscription required	☑
Header name	Ocp-Apim-Subscription-Key
Query parameter name	subscription-key

Figure 10. Settings to requiring a subscription key to access the API

Developer Portal

Now the API can be inspected in Developer portal that has automatically generated the documentation for it. Each operation can be inspected separately for information about them in a similar manner to Swagger Editor as seen in Figure 11.


JAMK thesis API
HOME
APIS
PRODUCTS
APPLICATIONS
ISSUES
SIGN IN
POST Add a new student
POST Add new absence for student
DELETE Delete all absences on certain day on specific student
DELETE Delete student by id
PUT Edit information about student by id
GET Get absence information for specific date for student
GET Get all absences for specific student
GET Lists all students
GET Returns a student by StudentId
API definition
API change history
Try it

Absences

Absences for students API

Add a new student

Add a new student

Request

Request URL

https://absences.azure-api.net/api/1.0/students

Request headers

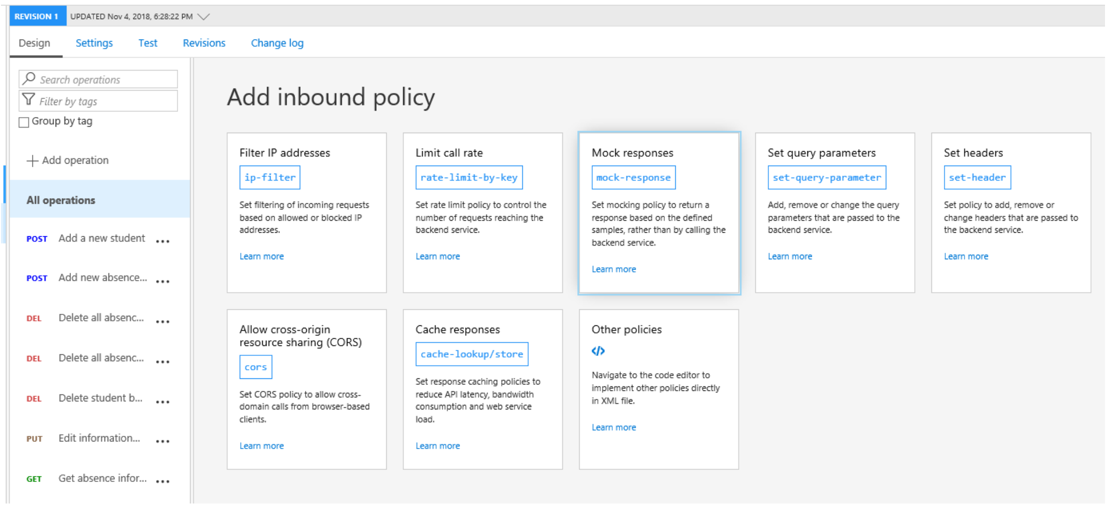
Content-Type (optional)	string	Media type of the body sent to the API.
Ocp-Apim-Subscription-Key	string	Subscription key which provides access to this API. Found in your Profile.

Figure 11. Documentation of Absences API in Developer portal

Mocking the requests for new API

At this point there are no back-end services or databases for the API to connect to, so testing the new API always returns “404 Not Found” response code. Not having a functioning back-end right after the API is implemented in API management service is intended in the API-first process. To start the development against the API, the next step is to turn on API management’s mock-responses that allow one to get responses based on the example-attributes specified in the OpenAPI specification.

Mocking can be turned on for all or just the chosen operations by adding a new in-bound policy as shown in Figure 12. It is possible to choose between successful re-quests which one will be returned to mocked calls. “200 OK” response was chosen for this case, so testing the operations always returns *200* response code with the ex-ample data from the API definition.



Set filtering of incoming requests based on allowed or blocked IP addresses.

Set rate limit policy to control the number of requests reaching the backend service.

Set mocking policy to return a response based on the defined samples, rather than by calling the backend service.

Add, remove or change the query parameters that are passed to the backend service.

Set policy to add, remove or change headers that are passed to the backend service.

Set CORS policy to allow cross-domain calls from browser-based clients.

Set response caching policies to reduce API latency, bandwidth consumption and web service load.

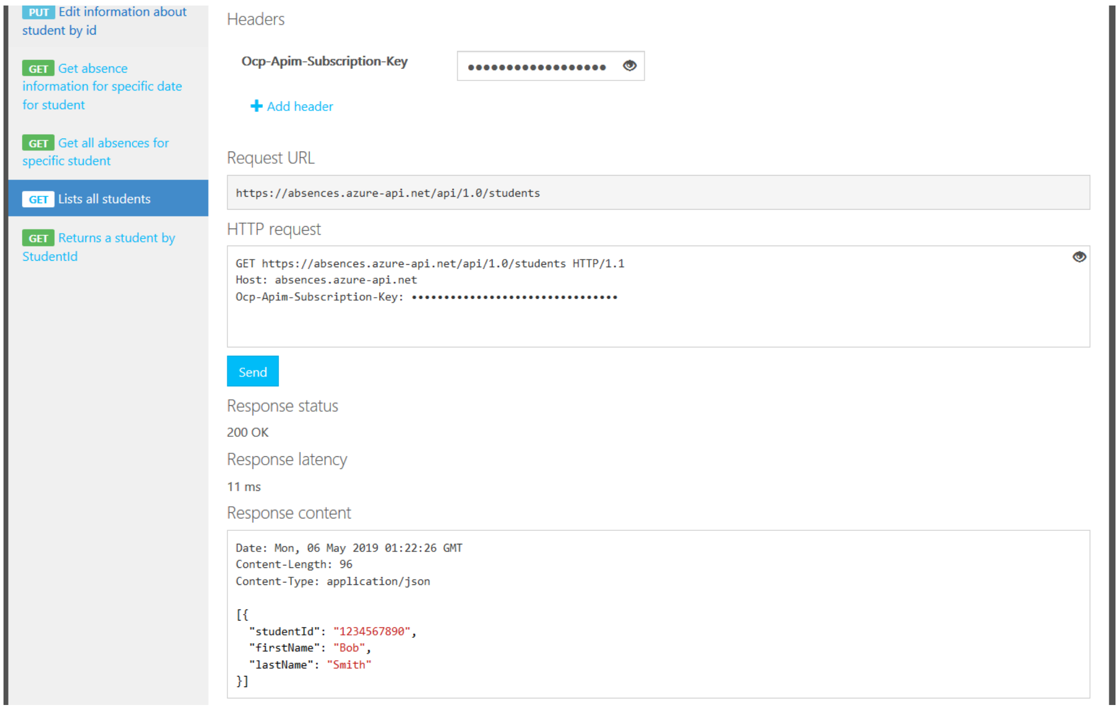
Navigate to the code editor to implement other policies directly in XML file.

Figure 12. Adding mock response policy

Testing mock API calls in Developer Portal

Calls against the mocked API can be tested in the Developer portal seen in Figure 13.

Subscription is needed to access the mocked resource. It is possible to either send a subscription key in the request header manually or become through Developer por-tal a subscriber who has access to make API calls. Becoming a subscriber requires an approval from an administrator.


PUT Edit information about student by id
GET Get absence information for specific date for student
GET Get all absences for specific student
GET Lists all students
GET Returns a student by StudentId
Headers
Ocp-Apim-Subscription-Key
Add header
Request URL
HTTP request
Send
Response status
Response latency
Response content

```
https://absences.azure-api.net/api/1.0/students
```

```
GET https://absences.azure-api.net/api/1.0/students HTTP/1.1
Host: absences.azure-api.net
Ocp-Apim-Subscription-Key: ••••••••••••••••••••••••••••••••
```

200 OK

11 ms

```
Date: Mon, 06 May 2019 01:22:26 GMT
Content-Length: 96
Content-Type: application/json

[{
  "studentId": "1234567890",
  "firstName": "Bob",
  "lastName": "Smith"
}]
```

Figure 13. Testing API call to GET a list of students in Developer portal

6.4 Using Swagger's code generator to create Client

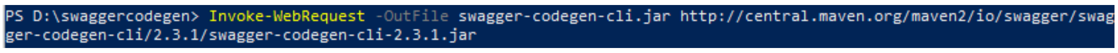
This chapter goes through generating an API client from OpenAPI definition with Swagger Code Generator. This demo was created in Windows 10 environment using Visual Studio 2017, and it is done in C# programming language.

Generating a client SDK with Swagger's Code Generator

Swagger Codegen is installed locally by downloading a .jar file which contains it. The file is the only interface for Swagger Codegen and its operations are accessed with command line. Using Swagger Codegen requires an installation of the latest version of Java.

Step 1: Download Swagger Codegen

Swagger Codegen's .jar package file can be downloaded through Windows PowerShell using the command shown in Figure 14. Here it is executed in a folder called *swaggercodegen* but the file can be located anywhere.

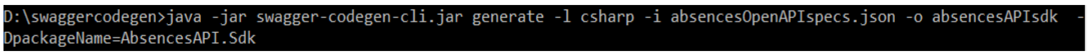


```
PS D:\swaggercodegen> Invoke-WebRequest -OutFile swagger-codegen-cli.jar http://central.maven.org/maven2/io/swagger/swag
ger-codegen-cli/2.3.1/swagger-codegen-cli-2.3.1.jar
```

Figure 14. Command for downloading Swagger Codegen package

Step 2: Generate new Client SDK

Creating a new Client SDK package that meets the needs for this demo can be done by a command shown in Figure 15. The command refers to previously downloaded Swagger Codegen .jar package and OpenAPI specification of the Absences API in .json format that was downloaded from Swagger Editor (Chapter 6.2). They are both located in the same folder.



```
D:\swaggercodegen>java -jar swagger-codegen-cli.jar generate -l csharp -i absencesOpenAPIspecs.json -o absencesAPIsdk  -
DpackageName=AbsencesAPI.Sdk
```

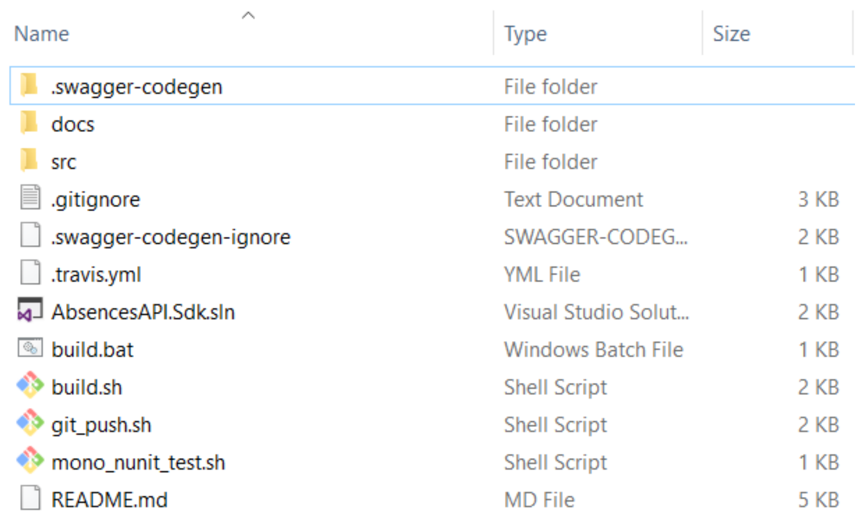
Figure 15. Generating client from OpenAPI

The base command to run Swagger Codegen and generating a new client is *java -jar swagger-codegen-cli.jar generate*. After it the command needs options that specify the programming language and OpenAPI specification used and the path to the folder to which it is generated. The command also uses an optional line to name the package other than its default name. These options are:

1. *-l csharp:* Using C# language
2. *-i absencesOpenAPIspecs.json*: Path to the OpenAPI specification
3. *-o:* Define path to the folder
4. *-DpackageName=AbsencesAPI.Sdk:* Naming the package Absences API.Sdk

By default, the command for creating a new Client SDK creates everything that is available for Swagger Codegen to generate from API specifications. It includes all APIs, models and tests.

After running the command, there should be three folders, *.swagger-codegen*, *docs* and *src*, and nine files such as the Visual Studio solution file as seen in Figure 16.



Name	Type	Size
.swagger-codegen	File folder	
docs	File folder	
src	File folder	
.gitignore	Text Document	3 KB
.swagger-codegen-ignore	SWAGGER-CODEG...	2 KB
.travis.yml	YML File	1 KB
AbsencesAPI.Sdk.sln	Visual Studio Solut...	2 KB
build.bat	Windows Batch File	1 KB
build.sh	Shell Script	2 KB
git_push.sh	Shell Script	2 KB
mono_nunit_test.sh	Shell Script	1 KB
README.md	MD File	5 KB

Figure 16. Contents generated by Swagger Codegen

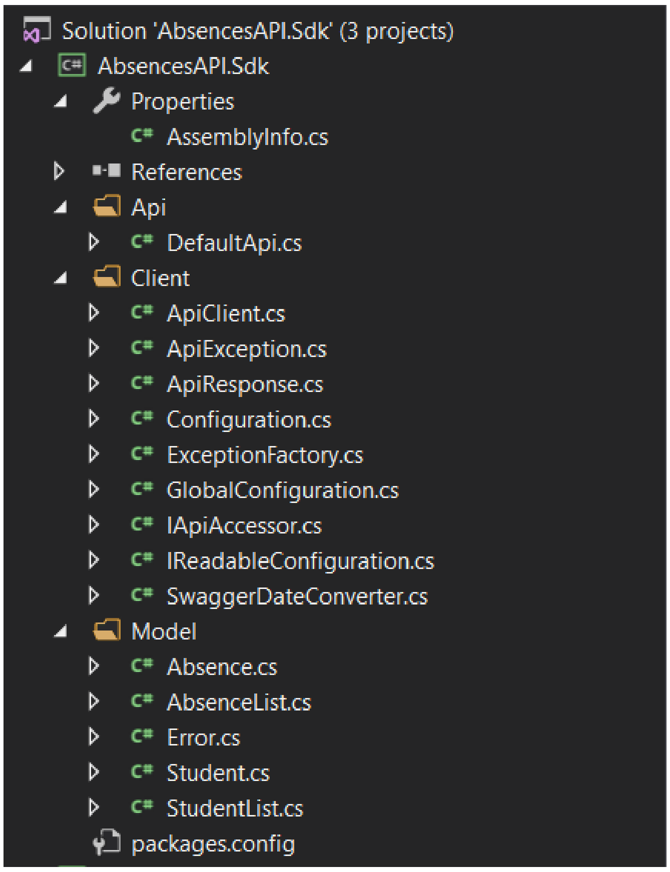
Step 3: Open the generated SDK in Visual Studio

The Visual Studio solution file of the new Client SDK can be opened in Visual Studio.

1. After opening the solution, it is recommended to click on the solution and press “Restore NuGet Packages” to avoid any problems caused by missing NuGet packages.
2. Clients generated by Swagger Codegen use RestSharp library. For RestSharp, Copy Local property has to be set true from References under AbsencesAPI.Sdk, otherwise the build will fail.
3. Build solution.

Details about what Swagger Codegen created

Swagger Codegen generates a solution that has a folder structure such as presented in Figure 17.

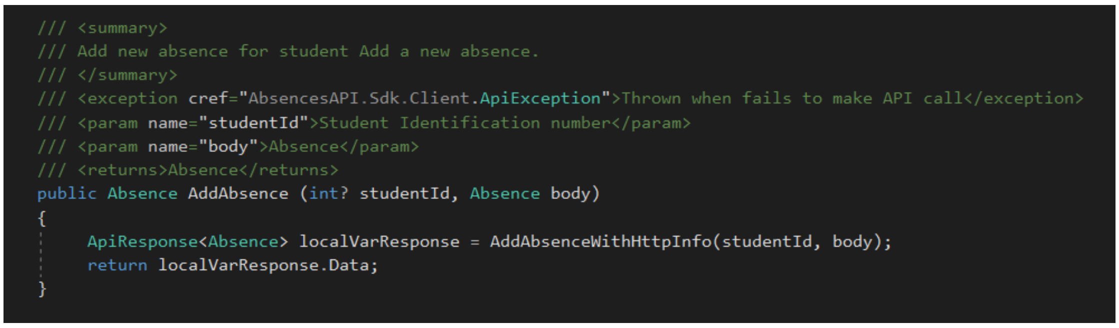


```
Solution 'AbsencesAPI.Sdk' (3 projects)
◢ AbsencesAPI.Sdk
  ◢ Properties
      AssemblyInfo.cs
  ▷ References
  ◢ Api
    ▷ DefaultApi.cs
  ◢ Client
    ▷ ApiClient.cs
    ▷ ApiException.cs
    ▷ ApiResponse.cs
    ▷ Configuration.cs
    ▷ ExceptionFactory.cs
    ▷ GlobalConfiguration.cs
    ▷ IApiAccessor.cs
    ▷ IReadableConfiguration.cs
    ▷ SwaggerDateConverter.cs
  ◢ Model
    ▷ Absence.cs
    ▷ AbsenceList.cs
    ▷ Error.cs
    ▷ Student.cs
    ▷ StudentList.cs
    packages.config
```

Figure 17. Folder structure of the generated solution

The solution has configuration that knows the API's base path which was defined in the OpenAPI specification and there are different options for handling the security of the API, such as sending access tokens or API keys.

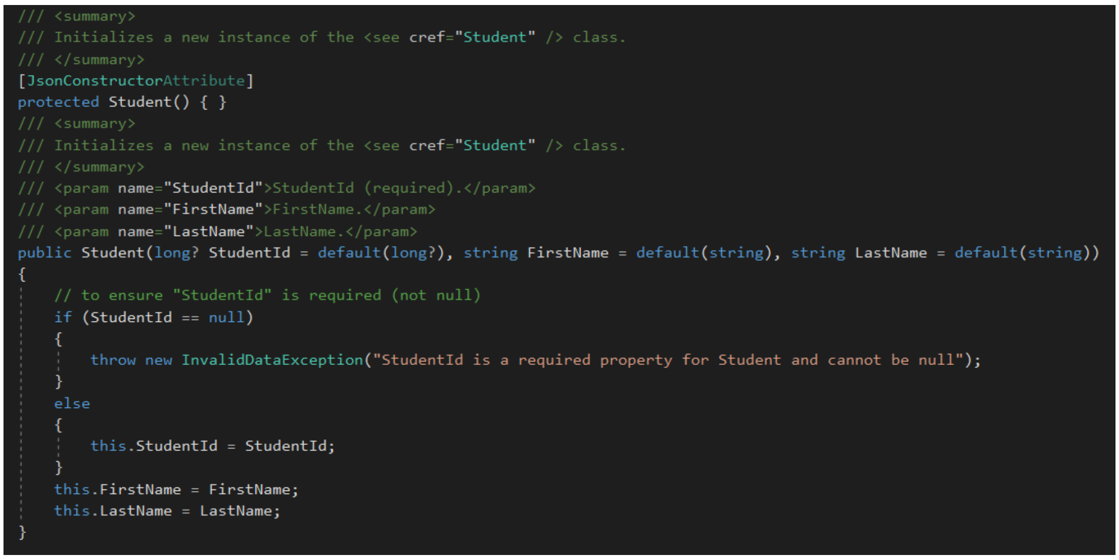
DefaultAPI.cs contains functions that make calls to the API endpoints. These functions have names, parameters and descriptions based on the OpenAPI definition as demonstrated in Figure 18.



```
/// <summary>
/// Add new absence for student Add a new absence.
/// </summary>
/// <exception cref="AbsencesAPI.Sdk.Client.ApiException">Thrown when fails to make API call</exception>
/// <param name="studentId">Student Identification number</param>
/// <param name="body">Absence</param>
/// <returns>Absence</returns>
public Absence AddAbsence (int? studentId, Absence body)
{
    ApiResponse<Absence> localVarResponse = AddAbsenceWithHttpInfo(studentId, body);
    return localVarResponse.Data;
}
```

Figure 18. Example of a function generated by Swagger Codegen

Swagger Codegen also generates classes which are based on the models from the API definition. They have all the same attributes as shown in Figure 19.

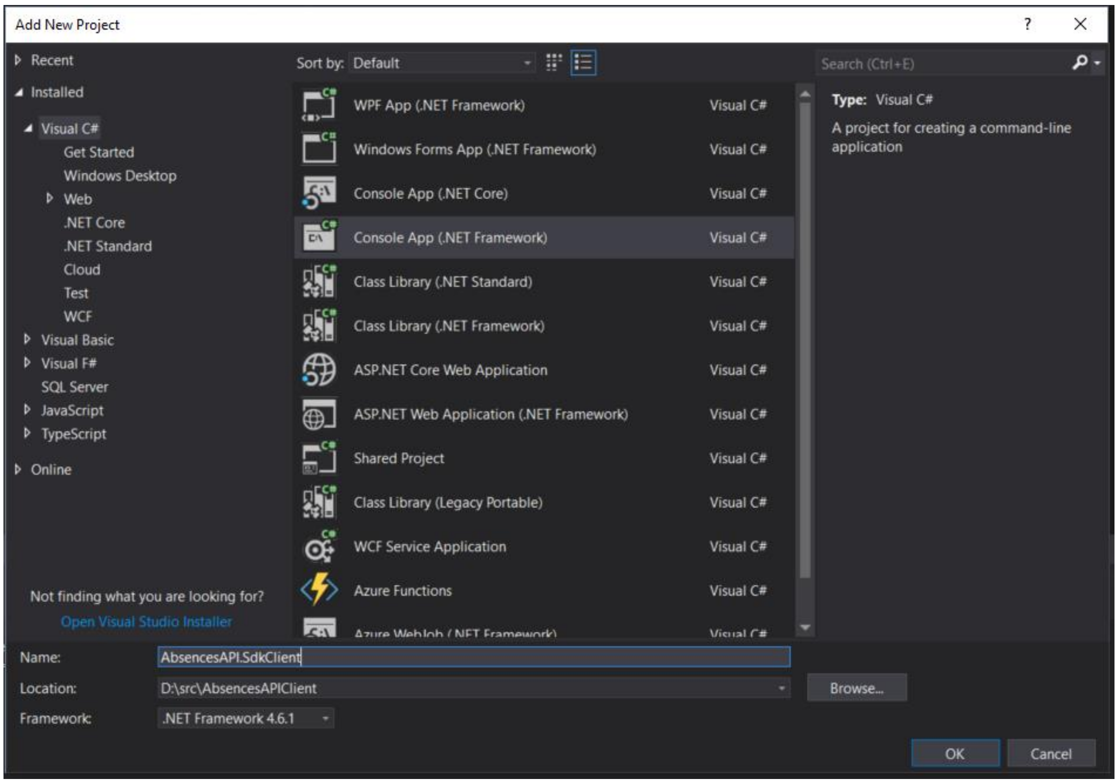


```
/// <summary>
/// Initializes a new instance of the <see cref="Student" /> class.
/// </summary>
[JsonConstructorAttribute]
protected Student() { }
/// <summary>
/// Initializes a new instance of the <see cref="Student" /> class.
/// </summary>
/// <param name="StudentId">StudentId (required).</param>
/// <param name="FirstName">FirstName.</param>
/// <param name="LastName">LastName.</param>
public Student(long? StudentId = default(long?), string FirstName = default(string), string LastName = default(string))
{
    // to ensure "StudentId" is required (not null)
    if (StudentId == null)
    {
        throw new InvalidDataException("StudentId is a required property for Student and cannot be null");
    }
    else
    {
        this.StudentId = StudentId;
    }
    this.FirstName = FirstName;
    this.LastName = LastName;
}
```

Figure 19 A class according to the Student model from OpenAPI specification

Step 4: Creating a Console Application to make API calls

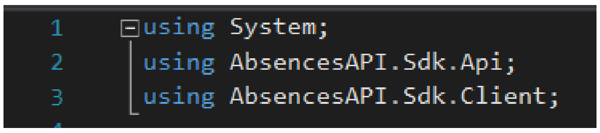
1. Add a new Console Application project to AbsencesAPI.Sdk solution as shown in Figure 20.
2. Set the new Console App project as the StartUp project of the solution.
3. Add a reference to the AbsencesAPI.Sdk project.
4. Add the lines *using AbsencesAPI.Sdk.Api;* and *using AbsencesAPI.Sdk.Client;* to the beginning of Program.cs as shown in Figure 21.


Add New Project
Recent
Installed
Visual C#
Get Started
Windows Desktop
Web
.NET Core
.NET Standard
Cloud
Test
WCF
Visual Basic
Visual F#
SQL Server
JavaScript
TypeScript
Online
Not finding what you are looking for?
Open Visual Studio Installer
Sort by: Default
WPF App (.NET Framework) Visual C#
Windows Forms App (.NET Framework) Visual C#
Console App (.NET Core) Visual C#
Console App (.NET Framework) Visual C#
Class Library (.NET Standard) Visual C#
Class Library (.NET Framework) Visual C#
ASP.NET Core Web Application Visual C#
ASP.NET Web Application (.NET Framework) Visual C#
Shared Project Visual C#
Class Library (Legacy Portable) Visual C#
WCF Service Application Visual C#
Azure Functions Visual C#
Search (Ctrl+E)
Browse...
OK
Cancel

Type: Visual C#
A project for creating a command-line application

Name: AbsencesAPI.SdkClient
Location: D:\src\AbsencesAPIClient
Framework: .NET Framework 4.6.1

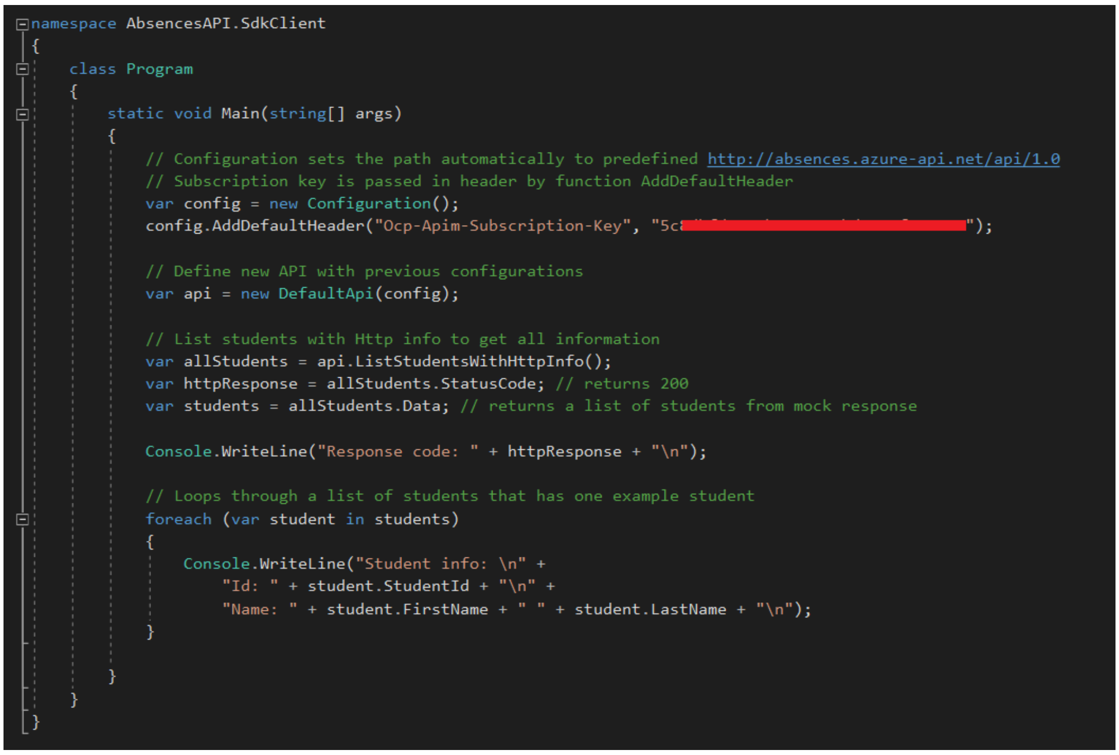
Figure 20. Creating a new Console App



```
using System;
using AbsencesAPI.Sdk.Api;
using AbsencesAPI.Sdk.Client;
```

Figure 21. Adding the using directives

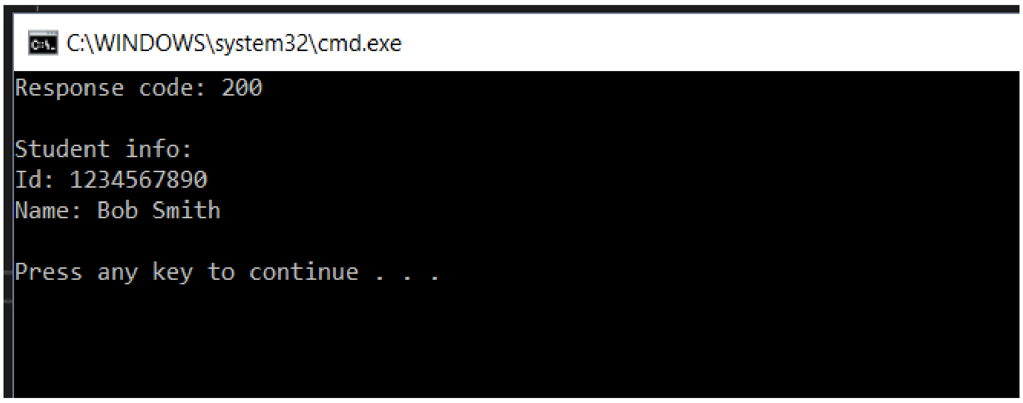
Now that all the references are in place, the calls to the API can be added to the console application’s Main method. The full program is shown in Figure 22 with comments explaining it.



```
namespace AbsencesAPI.SdkClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // Configuration sets the path automatically to predefined http://absences.azure-api.net/api/1.0
            // Subscription key is passed in header by function AddDefaultHeader
            var config = new Configuration();
            config.AddDefaultHeader("Ocp-Apim-Subscription-Key", "5c[redacted]");

            // Define new API with previous configurations
            var api = new DefaultApi(config);

            // List students with Http info to get all information
            var allStudents = api.ListStudentsWithHttpInfo();
            var httpResponse = allStudents.StatusCode; // returns 200
            var students = allStudents.Data; // returns a list of students from mock response

            Console.WriteLine("Response code: " + httpResponse + "\n");

            // Loops through a list of students that has one example student
            foreach (var student in students)
            {
                Console.WriteLine("Student info: \n" +
                    "Id: " + student.StudentId + "\n" +
                    "Name: " + student.FirstName + " " + student.LastName + "\n");
            }

        }
    }
}
```

Figure 22. Console application that makes API calls to Azure

As a result the call gets a mocked response from Azure APIM that was defined in OpenAPI specification as illustrated in Figure 23.



C:\WINDOWS\system32\cmd.exe

```
Response code: 200

Student info:
Id: 1234567890
Name: Bob Smith

Press any key to continue . . .
```

Figure 23. Response from mocked API to client application

7 Conclusions

The result of the thesis is a functioning API-first process in Azure environment using two tools, Swagger Editor and Codegen, and the OpenAPI standard in the development. API-first design and the tools were tested and evaluated in iterations by developing an API and the client application based on the previous iterations. The API-first process documented in Chapter 6 is the final result that fulfills the requirements of suitable tools and efficiency.

This chapter aims to answer the research questions and opens up each question based on the research.

What is API-first design?

API-first design is an API development method, where the development starts from the design phase. Designing an API includes deciding what API endpoints it needs in order to meet its requirements and writing an API definition which describes the API in a human and machine readable format. Writing an API definition is an important part of API-first design because it can be used as the API's documentation and the API is implemented based on the definition. Using API-first method compels to think about an API's structure and purpose carefully, which results in APIs of higher quality.

How do tools and standards support API-first design?

Tools and standards that were expected to support API-first design were evaluated and tested for their suitablity with the method and Azure. The tools presented in this thesis were valuable enough to the process and were suitable for precisely API-first approach.

Using tools when developing APIs is efficient and time-saving and makes the whole process relatively easy to go through for the first time. Since tools make the whole process faster, they can make up for the time spent designing the API and thereby lower the bar to start applying API-first design. Even though careful API designing has many benefits and therefore is seen worthy of the time it takes, having to take all that extra time in the beginning can seem less than an ideal solution. Using proper tools help to compensate this issue.

Making OpenAPI specifications is a useful skill to learn for everyone working with REST APIs, since there is so much work that can be mitigated through tools that support OpenAPI. The specification was the backbone of the whole process because it helped testing the API design and was used to deploy the API in Azure and generate the client SDK with Swagger Codegen. Learning to make a proper OpenAPI definition was the most time consuming technical issue during this research and also an important matter to get right because other steps relied on it.

Swagger Editor supported writing OpenAPI specification well. The most important feature in regard to learning to write the definition was getting instant feedback that informed about mistakes and showed the definition visually as it was growing. When testing Swagger Editor, one issue came up. A syntax error was not caught by Swagger Editor and came up while creating a new API in Azure from the API definition. Azure APIM would not accept the same OpenAPI definition in JSON format in which Swagger Editor did not find any errors with.

Using Swagger Codegen to create a client SDK can save plenty of time while offering an encompassing project with all basics for making API calls. The generated code in itself also has some benefits; it will be consistent so it is easy and predictable to work with. Combining generated code with mocked API calls is a practical solution. It is possible to immediately start developing a front-end client application against the simulated API.

There were overall four tools tested in this research. All of the four tools tested in iterations helped to get a better understanding of using API-first method and finding out how OpenAPI specification can be applied beneficially; yet, two tools did not fit an effective API-first design process when combined with Azure API management. These tools were used in the first iteration and were cut from the final API-first process documentation. Their functionalities did not add enough value compared to the time it took to set them up and learn to use them because they overlapped with Azure API management's functionalities. Additionally, they would not be free to use in professional usage.

How can API-first design benefit API development in Azure environment efficiently?

API-first process presented in this thesis shows an effective way to apply API-first design with Azure. It includes using tools that support the process, and steps that benefit from Azure's own features.

API-first design has many benefits that were confirmed during this research. The benefits of using API-first design in Azure environment are:

- Good and clear API documentation is created according to the API definition automatically. The documentation can be seen in Azure Portal.

- Deploying the API to Azure is easy to commit when there is an API definition.

- The API is precice to the business needs because close attention was paid to its design.

- It decreases the possibility of mistakes by having tools that can catch them and making parts of the process automatic.

- Designing and writing an API definition affects the API's developer experience by creating a good documentation and a consistent API.

The final API-first process with the used tools and Azure API management did not fully follow the API-first steps from Chapter 4.3. This was a result from iterations, which concluded that the steps needed to be altered to fit Azure API management better. A process following strictly the API-first steps could have been done with a different set of tools, while it was not worthwile with Azure. The final result differs from the API-first steps in regard to testing at an early stage, where testing API calls against a tool before implementation was recommended. With Azure API management, implementing the API based on its OpenAPI specification is fast and effortless, which makes changing it afterwards a possibility. Early testing of the API design can be committed by comparing the REST structure to user stories. Azure APIM also offers good features for testing in Azure Portal.

8 Discussion

API-first design as a subject was relatively new and at the time other researches about it could not be found. However, there was enough material from multiple sources for getting reliable information and ending up with a good understanding of the subject and a base for the research. Most of these sources were online articles and blog posts from company representatives specializing in API development. The information about API-first design from different sources was consistent, so the theoretical framework of this research can be considered reliable.

The research method used in this thesis, design research, fits the purpose of this research well. The research aimed to solve a problem with a practical solution and the iterative nature of design research was needed in order to get a beneficial and refined result. Each iteration gave more insights about the API-first process itself, how to benefit more from the set of tools and to find out if there are tools or practices that are not valuable enough to the process.

This research covered one case in a specific development environment using Windows and Visual Studio due to thesis assigner's needs, which means it is not fully applicable to other environments as such; however, the results should be reproducable in any API development done in the same environment. Going through multiple iterations helped to assure that the technical parts of the research work reliably.

Since the API-first process in question was implemented as a demo rather than a real business case, it lacks the aspect of testing it in a real production environment and hearing developers' opinions about its usefulness. While this was the intent all along to keep the scope of the thesis small enough, the results could have varied and been more reliable in such environment. Additionally, only short-term benefits of API-first design could be tested in the context of this research. This was expected as well; however, follow-up research about the long-term benefits would be a welcome addition.

Since this research was done from an introductory point of view to API-first design, there are many subjects to research that can be derived from this thesis. The most important aspect not included in this thesis in practice is API-first design's effect on teamwork. It can be concluded from the results of this reseach that it is possible to work simultaniously on front-end and back-end after mocked API calls are turned on, so further research about its impact could be made. Another possible research subject could be implementing API-first design with products that already exist. Is it worthwile or even possible?

Because API economy is a growing trend, any research about efficient API development methods would be topical.

References

About | Swagger. N.d. Swagger. Accessed 28.9.2018. Retrieved from https://swagger.io/about/.

API Design – API-University. N.d. API-University. Accessed 10.10.2018. Retrieved from https://api-university.com/api-lifecycle/api-design/.

API Management: Establish API Gateways. N.d. Microsoft Azure. Accessed 5.11.2018. Retrieved from https://azure.microsoft.com/en-us/services/api-management/.

Baggrett, D. 2018. Practicing API-First Design. Medium. Accessed 6.9.2018. Retrieved from https://medium.com/rocket-fuel/practicing-api-first-design-18b3d55ea4ab.

Basic Structure | Swagger. N.d. Swagger. Accessed 4.11.2018. Retrieved from https://swagger.io/docs/specification/2-0/basic-structure/.

Cater, M. 2013. A Brief History of API-Based Web Applications. Smartbear. Accessed 13.10.2018. Retrieved from https://smartbear.com/blog/test-and-monitor/a-brief-history-of-api-based-web-applications/.

De, B. 2017. API Management: An Architect's Guide to Developing and Managing APIs for Your Organization. Apress. Accessed 6.9.2018. Retrieved from https://janet.finna.fi/, Books24x7.

Doglio, F. 2015. Pro REST API Development with Node.js. Apress. Accessed 6.9.2018. Retrieved from https://janet.finna.fi/, Books24x7.

Download Swagger Codegen. N.d. Swagger. Accessed 3.5.2019. Retrieved from https://swagger.io/tools/swagger-codegen/download/.

Glickenhouse, A. 2017. API Monetization – What Does It Really Mean? IBM Developer. Accessed 3.12.2018. Retrieved from https://developer.ibm.com/apiconnect/2017/10/17/api-monetization-really-mean/.

Glickenhouse, A. 2018. What is an API? and What is the API Economy? IBM Developer. Accessed 3.12.2018. Retrieved from https://developer.ibm.com/apiconnect/2018/01/04/api-api-economy/.

Hoffman, K. 2016. An API-first approach for cloud-native app development. O'Reily. Accessed 13.9.2018. Retrieved from https://www.oreilly.com/ideas/an-api-first-approach-for-cloud-native-app-development.

Jarman, S. 2017. The Best Practices for a Great Developer Experience (DX). Hackernoon. Accessed 11.12.2018. Retrieved from https://hackernoon.com/the-best-practices-for-a-great-developer-experience-dx-9036834382b0.

Jarrel, J. 2018. Writing Great User Stories For Developing APIs. Jeremy Jarrel. Accessed 4.11.2018. Retrieved from http://www.jeremyjarrell.com/user-stories-apis/.

Jaswal, A. 2017. Why the phrase 'API-first' should be at the heart of every digital experience. Digital Pulse. Accessed 17.10.2018. Retrieved from https://www.digitalpulse.pwc.com.au/phrase-api-first-heart-every-digital-experience/.

Jauker, S. 2014. 10 Best Practices for Better RESTful API. M-Way Solutions. Accessed 24.4.2019. Retrieved from https://blog.mwaysolutions.com/2014/06/05/10-best-practices-for-better-restful-api/.

Kananen, J. 2012. Kehittämistutkimus opinnäytetyönä : kehittämistutkimuksen kirjoittamisen käytännön opas. Jyväskylä: Jyväskylän Ammattikorkeakoulu.

Levin, G. 2016. An API-First Development Approach. Dzone. Accessed 6.9.2018. Retrieved from https://dzone.com/articles/an-api-first-development-approach-1.

Liew, Z. 2018. Understanding And Using REST APIs. Smashing Magazine. Accessed 5.11.2018. Retrieved from https://www.smashingmagazine.com/2018/01/understanding-using-rest-api/.

Mackory, M. 2018. Getting Started with the OpenAPI Specification. Runscope Blog. Accessed 5.10.2018. Retrieved from https://blog.runscope.com/posts/getting-started-with-the-openapi-specification.

Microsoft Azure Portal. N.d. Microsoft Azure. Accessed 5.11.2018. Retrieved from https://azure.microsoft.com/en-us/features/azure-portal/.

Moilanen, J., Niinioja, M., Seppänen, M. & Honkanen, M. 2018. API-talous 101. Liettua: BALTO print.

OpenAPI Specification. N.d. Swagger. Accessed 5.10.2018. Retrieved from https://swagger.io/specification/.

OpenAPI v3 support in Azure API Management. 2018. Microsoft Azure. Accessed 5.10.2018. Retrieved from https://azure.microsoft.com/en-us/updates/openapi-v3-support-in-azure-api-management/.

Patni, S. 2017. Pro RESTful APIs: Design, Build and Integrate with REST, JSON, XML and JAX-RS. Apress. Accessed 23.9.2018. Retrieved from https://janet.finna.fi/, Books24x7.

Pinkham, R. 2017. API Developer Experience: Why it Matters, and How Documenting Your API with Swagger Can Help. Accessed 5.11.2018. Retrieved from https://swagger.io/blog/api-documentation/api-documentation-and-developer-experience/.

Raboy, N. 2016. TPDP Episode #4: What is All This Mobile First, Offline First, and API First Jargon? The Polyglot Developer. Accessed 13.9.2018. Retrieved from https://www.thepolyglotdeveloper.com/2016/04/tpdp-episode-4-what-is-all-this-mobile-first-offline-first-and-api-first-jargon/.

Riggins, J. 2015. How To Design Great APIs With API-First Design. Programmableweb. Accessed 6.9.2018. Retrieved from https://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10.

Robles, P. 2014. The Emerge of API-first Development. ProgrammableWeb. Accessed 6.9.2018. Retrieved from https://www.programmableweb.com/news/emergence-api-first-development/2014/01/09.

Rosenstock, L. 2018. OpenAPI and Design-First Principles – Stoplight API Corner. Stoplight. Accessed 2.10.2018. Retrieved from https://blog.stoplight.io/openapi-and-design-first-principles-96e7c4b2aec1.

Santos, W. 2016. Introduction to API-First Design. Programmableweb. Accessed 6.9.2018. Retrieved from https://www.programmableweb.com/news/introduction-to-api-first-design/analysis/2016/10/31.

Swagger Codegen. N.d. Swagger. Accessed 5.11.2018. Retrieved from https://swagger.io/tools/swagger-codegen/.

Swagger Editor. N.d. Swagger. Accessed 5.11.2018. Retrieved from https://swagger.io/tools/swagger-editor/.

Taman, M. 2019. Effective Design of RESTful APIs. Accessed 24.4.2019. Retrieved from https://prezi.com/vprunu_rrffc/effective-design-of-restful-apis/.

Trieloff, L. 2017. Three Principles of API First Design. Medium. Accessed 21.9.2018. Retrieved from https://medium.com/adobetech/three-principles-of-api-first-design-fa6666d9f694.

Urpilainen, A. 2017. Pikakurssi API-talouteen. HiQ Finland. Accessed 17.10.2018. Retrieved from http://blogi.hiqfinland.fi/pikakurssi-api-talouteen.

Vasudevan, K. 2016a. Best Practices in API Design. SwaggerBlog. Accessed 10.10.2018. Retrieved from https://swagger.io/blog/api-design/api-design-best-practices/.

Vasudevan, K. 2016b. What is API Design (And Why Does It Matter?). SwaggerBlog. Accessed 10.10.2018. Retrieved from https://swagger.io/blog/api-design/what-is-api-design/.

Vasudevan, K. 2017a. Design First or Code First: What’s the Best Approach to API Development? | Swagger. SwaggerBlog. Accessed 3.10.2018. Retrieved from https://swagger.io/blog/api-design/design-first-or-code-first-api-development/.

Vasudevan, K. 2017b. What Organizations Need to Know When Deprecating APIs. SwaggerBlog. Accessed 5.11.2018. Retrieved from https://swagger.io/blog/api-strategy/best-practices-for-deprecating-apis/.

Vinogradsky, V. 2017. Azure API Management overview and key concepts. Microsoft Azure. Accessed 5.11.2018. Retrieved from https://docs.microsoft.com/en-us/azure/api-management/api-management-key-concepts.

Vinogradsky, V., Budzynski, M., Kornich, J., Johnson, W. 2018. Import and publish your first API. Microsoft Docs. Accessed 6.9.2018. Retrieved from https://docs.microsoft.com/en-us/azure/api-management/import-and-publish.

Viswanathan, K. 2017. The Basics of API-Driven Development. Dzone. Accessed 23.9.2018. Retrieved from https://dzone.com/articles/abcs-of-api-driven-development.

Wagner, J. N.d. Understanding the API-First Approach to Building Products. Swagger. Accessed 6.5.2019. Retrieved from https://swagger.io/resources/articles/adopting-an-api-first-approach/.

What is Azure – Microsoft Cloud Services. N.d. Microsoft Azure. Accessed 5.11.2018. Retrieved from https://azure.microsoft.com/en-us/overview/what-is-azure/.

Appendices

Appendix 1. OpenAPI Specification for Absences API in YAML format

```
swagger: '2.0'
info:
  title: Absences
  version: '1.0'
  description: Absences for students API
host: absences.azure-api.net
basePath: /api/1.0
schemes:
  - http
  - https
paths:
  /students:
    post:
      summary: Add a new student
      description: Add a new student
      operationId: addStudent
      parameters:
        - name: "body"
          in: body
          schema:
            $ref: '#/definitions/Student'
          required: true
          description: Student that will be added
      consumes:
        - application/json
      responses:
        '200':
          description: OK - student added
          schema:
            $ref: "#/definitions/Student"
        '500':
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
      produces:
        - application/json
    get:
      summary: Lists all students
      description: Lists all students
      operationId: listStudents
      responses:
        '200':
          description: OK - Got list of all students.
          schema:
            $ref: '#/definitions/StudentList'
        '500':
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
      produces:
        - application/json
  '/students/{studentId}':
    get:
      summary: Returns a student by StudentId
      description: Get information on specific student
      operationId: getStudentById
      parameters:
        - name: studentId
          in: path
          description: Student Identification number.
          required: true
          type: integer
      responses:
        '200':
          description: OK - Got student
          schema:
            $ref: '#/definitions/Student'
        '404':
          description: Student not found
          schema:
            $ref: '#/definitions/Error'
        '500':
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
      produces:
        - application/json
    put:
      summary: Edit information about student by id
      description: Change information on specific student.
      operationId: editStudentById
      parameters:
        - name: studentId
          in: path
          description: Student Identification number.
          required: true
          type: integer
        - name: student
          in: body
          schema:
            $ref: '#/definitions/Student'
      consumes:
        - application/json
      responses:
        '200':
          description: Change successful
          schema:
            $ref: '#/definitions/Student'
        '404':
          description: Student not found
          schema:
            $ref: '#/definitions/Error'
        '500':
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
      produces:
        - application/json
    delete:
      summary: Delete student by id
      description: Delete information on specific student
      operationId: deleteStudentById
      parameters:
        - name: studentId
          in: path
          description: Student Identification number.
          required: true
          type: integer
      responses:
        '200':
          description: Delete successful
        '404':
          description: Student with this ID not found
          schema:
            $ref: '#/definitions/Error'
        '500':
          description: Unexpected error
          schema:
            $ref: '#/definitions/Error'
      produces:
        - application/json
  '/students/{studentId}/absences':
    post:
        summary: Add new absence for student
        description: Add a new absence.
        operationId: addAbsence
        parameters:
        - name: studentId
          in: path
          description: Student Identification number
          required: true
          type: integer
        - in: body
          name: body
          description: "Absence"
          required: true
          schema:
            $ref: "#/definitions/Absence"
        consumes:
          - application/json
        responses:
          '200':
            description: OK - Absence added to a student
            schema:
              $ref: '#/definitions/Absence'
          '500':
            description: Unexpected error
            schema:
              $ref: '#/definitions/Error'
        produces:
          - application/json
    get:
      summary: Get all absences for specific student
      description: Get all absence information on specific student
      operationId: getAbsencesByStudentId
      parameters:
        - name: studentId
          in: path
          description: Student Identification number.
          required: true
          type: integer
      responses:
        '200':
          description: Successful
          schema:
            $ref: '#/definitions/AbsenceList'
        '500':
          description: Unexpected error
          schema:
              $ref: '#/definitions/Error'
      produces:
        - application/json
  '/students/{studentId}/absences/{date}':
    get:
      summary: Get absence information for specific date for student
      description: Get absence information for specific student on
specific date.
      operationId: getAbsencesForStudentbyDate
      parameters:
        - name: studentId
          in: path
          description: Student Identification number
          required: true
          type: integer
        - name: date
          in: path
          description: 'Format: date. Date for absence to be posted.
Format example: 2018-07-31'
          required: true
          type: string
      responses:
        '200':
          description: OK - Got absences for specific date
          schema:
            $ref: '#/definitions/Absence'
        '500':
          description: Unexpected error
          schema:
              $ref: '#/definitions/Error'
      produces:
        - application/json
    delete:
      summary: Delete the absence for certain day on specific stu-
dent
```

```
      description: Delete the absence for certain day on specific
student
      operationId: deleteAbsencesForStudentByDate
      parameters:
        - name: studentId
          in: path
          description: Student Identification number
          required: true
          type: integer
        - name: date
          in: path
          description: 'Format: date. Date for absence to be posted.
Example: 2018-07-31'
          required: true
          type: string
      responses:
        '200':
          description: OK - Delete successful
        '500':
          description: Unexpected error
          schema:
              $ref: '#/definitions/Error'
      produces:
        - application/json
definitions:
  StudentList:
    type: array
    items:
      $ref: '#/definitions/Student'
  Student:
    type: object
    properties:
      studentId:
        type: integer
        format: int64
        example: '1234567890'
      firstName:
        type: string
        example: Bob
      lastName:
        type: string
        example: Smith
    required:
      - studentId
  AbsenceList:
    type: array
    items:
      $ref: '#/definitions/Absence'
  Absence:
    type: object
    properties:
      date:
        type: string
        format: date
        example: '2018-09-21'
        description: Date for absence (date format).
      reason:
        type: string
        example: 'Absence due to sickness'
        description: Reason why student was absent
    required:
      - date
  Error:
    type: object
    properties:
      code:
        type: string
      message:
        type: string
    required:
      - code
      - message
tags: []
```