Ville Vehniä

Implementing Azure Active Directory Integration with an Existing Cloud Service

School of Technology and Innovations
Master's thesis in Automation and Information Technology

Vaasa 2020

UNIVERSITY OF VAASA		
School of Technology and Innovations		
Author:	Ville Vehniä	
Title of the Thesis:	Implementing Azure Active Directory Integration with an Existing Cloud Service	
Degree:	Master of Science in Technology	
Programme:	Automation and Information Technology	
Instructor:	MSc Heikki Mattila	
Supervisor:	Professor Timo Mantere	
Year:	2020	**Pages:** 93

ABSTRACT:

Training Simulator (TraSim) is an online, web-based platform for holding crisis management exercises. It simulates epidemics and other exceptional situations to test the functionality of an organization's operating instructions in the hour of need. The main objective of this thesis is to further develop the service by delegating its existing authentication and user provisioning mechanisms to a centralized, cloud-based Identity and Access Management (IAM) service. Making use of a centralized access control service is widely known as a Single Sign-On (SSO) implementation which comes with multiple benefits such as increased security, reduced administrative overhead and improved user experience.

The objective originates from a customer organization's request to enable SSO for TraSim. The research mainly focuses on implementing SSO by integrating TraSim with Azure Active Directory (AD) from a wide range of IAM services since it is considered as an industry standard and already utilized by the customer. Anyhow, the complexity of the integration is kept as reduced as possible to retain compatibility with other services besides Azure AD. While the integration is a unique operation with an endless amount of software stacks that a service can build on and multiple IAM services to choose from, this thesis aims to provide a general guideline of how to approach a resembling assignment.

Conducting the study required extensive search and evaluation of the available literature about terms such as IAM, client-server communication, SSO, cloud services and AD. The literature review is combined with an introduction to the basic technologies that TraSim is built with to justify the choice of OpenID Connect as the authentication protocol and why it was implemented using the *mozilla-django-oidc* library. The literature consists of multiple online articles, publications and the official documentation of the utilized technologies. The research uses a constructive approach as it focuses into developing and testing a new feature that is merged into the source code of an already existing piece of software.

KEYWORDS: Azure Active Directory, Client-server communication, Cloud computing, Identity and Access Management, Single Sign-On

Contents

Figures

Tables

Abbreviations

3DES	Triple Data Encryption Algorithm
ACID	Atomicity, Consistency, Isolation and Durability
AD	Active Directory
AES	Advanced Encryption Standard
API	Application Programming Interface
AWS	Amazon Web Services
CAA	Crypto Approval Authority
CD	Continuous Delivery
CI	Continuous Integration
CMS	Content-Management-System

CPU	Central Processing Unit
CSS	Cascading Style Sheets
DOM	Document Object Model
DTL	Django Template Language
EC2	Elastic Compute Cloud
FIM	Federated Identity Management
FOSS	Free and Open-Source Software
FTP	File Transfer Protocol
GUI	Graphical User Interface
HMAC	Hash-based Message Authentication Code
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Security
IAM	Identity and Access Management
IDaaS	Identity as a Service
IdP	Identity Provider
IP	Internet Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
LDAP	Lightweight Directory Access Protocol
M2M	Machine to Machine
MFA	Multi-Factor Authentication
MITM	Man in the Middle
MTV	Model Template View
MVC	Model-View-Controller
MVCC	Multiversion Concurrency Control
NIST	National Institute of Standards and Technology
OAuth	Open Authorization
OIDC	OpenID Connect
OS	Operating System

OTP	One-Time Password
OWASP	Open Web Application Security Project
PSL	Python Standard Library
PyPI	Python Package Index
REST	REpresentational State Transfer
SaaS	Software as a Service
SAML	Security Assertion Markup Language
SHA2	Secure Hash Algorithm 2
SP	Service Provider
SPA	Single-Page Application
SQL	Structured Query Language
SSO	Single Sign-On
Sudo	Superuser do
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TraSim	Training Simulator
UI	User Interface
URL	Uniform Resource Locator
UX	User Experience
VCS	Version Control System
VM	Virtual Machine
WSGI	Web Server Gateway Interface
XML	Extensible Markup Language
XSS	Cross-Site Scripting

Acknowledgement

I would like to present a thank you to my workmates at Insta Digital for providing me the opportunity to write my thesis. Major respect to all members of the Trust domain and a special mention to Heikki Mattila for instructing me throughout the whole research process. It has been a pleasure to dive into the world of web development with guidance from true professionals such as Valtteri Luoma and Juanjo Díaz.

From University of Vaasa I express my gratitude to Professor Timo Mantere for supervising both my Master and Bachelor’s theses. Additional mention to Petri Välisuo and Tobias Glocker for holding numerous interesting courses and always giving lectures with encouraging attitude.

Most importantly, I would like to thank my family and friends for all the support I have received during these years.

Ville Vehniä
Tampere, April 30, 2020.

1 Introduction

Do you know how to manage a crisis? They happen all the time and bring big change. As we have seen with the ongoing coronavirus outbreak, the operating environment of whole countries let alone companies can change in a flash. When the change is inevitable, one can protect themselves or their organization from disruptive and unexpected events by preparing how to manage the situation in a way that lessens or completely prevents the damage inflicted (Tobak 2008). While crisis management is not an unambiguous process, the senior executives of organizations may not rise to the occasion and their reactions can make matters worse.

1.1 Case company and product

This thesis is written for Insta Digital, which is a software company founded as a startup of three persons during 2012 in the aftermath of Nokia's meltdown (Latvala 2018). By 2018, the company had profitably grown from the three founding members to 75 software specialists from 14 different nationalities. About the same time, Insta Digital (formerly known as Intopalo Digital) was acquired by Insta Group, a Finnish family owned business focused on industrial automation, digitalization, cybersecurity and defense technologies (Insta Group 2020). Insta Digital then became one of the five segment companies that together compose the Group. At the time of writing, Insta Digital alone employs over one hundred professionals in Tampere, Helsinki and Munich, while the Group has in total over 1100 employees.

The core service of Insta Digital is to solve a wide range of problems related to following domains: leading digital transformation, design, web and cloud, advanced user interfaces, cybersecurity, data science and embedded systems. Despite its focus being on providing professional services, Insta Digital offers a crisis management product called Training Simulator (TraSim). The product is originally developed by a security company called Countsec which was acquired by Insta Group in a similar fashion to Insta Digital.

TraSim and the related crisis management consultancy business was handed over to Insta Digital a while after its acquisition had taken place as a more suitable stakeholder for the entity.

Essentially TraSim provides a safe platform to test the functionality of an organization's operating instructions in the hour of need. It offers an online platform for holding visual crisis management exercises that can be bundled with consultancy from crisis management experts to raise the operational readiness of an organization's key personnel to the required level. TraSim prepares organizations for a true incident that is bound to happen by simulating disruptions, emergencies, epidemics and other exceptional situations. The product is served in a Software as a Service (SaaS) manner and it has been continuously enhanced during its lifetime including one major rewrite to become compliant with the protection level IV set by Crypto Approval Authority (CAA).

1.2 Problem and objectives

The main research question of this thesis is how to delegate the built-in authentication and user provisioning mechanisms of an existing cloud service (TraSim) to a centralized, cloud-based identity and access management service. The question originates from a request of a customer organization to allow their employees to sign-on to TraSim using the credentials stored to the organization's Azure Active Directory (Azure AD) tenant. This kind of implementation is called Single Sign-On (SSO) and it requires an integration between a Service Provider (SP) and Identity Provider (IdP), which are TraSim and Azure AD in this case.

Implementing SSO to a cloud service is not an uncommon request as the average enterprise was expected to use 17 SaaS applications in 2017 which is a 50 % increase from 2015 (Solomon 2016). One may speculate the increasing demand for SSO consists of multiple factors such as the rise of cloud applications, password fatigue, new developer methodologies, enterprise mobility, web and cloud native applications and the improved security that SSO brings (Drinkwater 2018).

While there is an endless amount of software stacks that a SP can build on and a multitude of different IdPs, the basic aspects of an integration remain similar in every implementation. The case specific underlying technologies only place certain limitations or freedoms on which authentication protocols can be utilized and how they are to be implemented. While every implementation is unique, this thesis aims to provide a general guideline of how to complete a comprehensive integration process.

The research begins with a literature review about the themes behind the main objective such as IAM, client-server communication, SSO, cloud services and Azure AD. The review is then followed with an introduction to the basic technologies that TraSim is built with to justify the choices made in the subsequent chapter, which deals with the actual integration process. The literature consists of multiple online articles, publications and the official documentation of the utilized technologies. The research uses a constructive approach as it focuses into developing and testing a new feature that is merged into the source code of an already existing piece of software. Finally, the study is concluded by recapping the results and answering following questions: how the combination of the technical decisions played out in the end, what could have been done better and are there any promising areas of future studies.

2 Theoretical background

Before starting to plan the actual integration, one must acquire the basic knowledge of related concepts. This chapter begins with introduction to Identity and Access Management (IAM) and client-server communication. Then the focus shifts to SSO and authentication protocols, cloud computing and the service models and then the chapter is ended with an orientation to AD.

2.1 Identity and Access Management

The description of IAM is written below according to an online glossary published by Gartner (2020). For in depth explanation, the individual words shall be disassembled and expanded in more technical point of view during following chapters.

> *"Identity and access management is the discipline that enables the right individuals to access the right resources at the right times for the right reasons. IAM addresses the mission-critical need to ensure appropriate access to resources across increasingly heterogeneous technology environments, and to meet increasingly rigorous compliance requirements. IAM is a crucial undertaking for any enterprise. It is increasingly business-aligned, and it requires business skills, not just technical expertise. Enterprises that develop mature IAM capabilities can reduce their identity management costs and, more importantly, become significantly more agile in supporting new business initiatives."*

2.1.1 Identity

In the digital world, where one can not physically prove their identity with the flash of a government registered identification document, there are other means to prove the digital identity. Basically, digital identity is the compilation of information about a user, that exists in digital form. This information can be grouped in two categories: user attributes and user activities. These pieces of information, combined or solely, are the construction blocks of the digital identity. Examples of common user attributes and activities are listed below (Table 1).

Table 1. Common user attributes and activities.

User attributes	User activities
userId	Social media activity
username	Geotagging
email	Purchace history
commonName	Search Queries
surname	Mouse movements
telephone	Time of activity
address (IP)	Session history
address (mailing)	Topics of interest

The attributes *userId, username* and *email* are bolded to symbolize the fact they are oftentimes used as the unique identifier. This means a special kind of attribute, which is used to separate different users from each other in a specific namespace. Again, a namespace stands for a group where the unique identifier of an item is truly unique. For example, in the citizen registry of Finland, there may exist only a single person with the personal identification number *012345-6789*. The process of choosing the unique identifier must be thoroughly considered. Things such as coverage, cardinality, revocation and re-assignment of the unique identifier can cause trouble during the lifecycle of a system. The unique identifier might need to be changed or re-assigned to another user. Also, it must be made sure that the range of possible unique identifiers is wide enough to cover all users. (Linden 2017).

2.1.2 Access

Access is something gained via a digital identity. It is the credibility that allows a user to access a resource, use a service or interact with people in a trustworthy manner online. Before gaining access, the identity of the user must be verified, or in other words, the user must be authenticated. There are multiple authentication methodologies and protocols, designed for building a strong assurance of the identity of one entity to another. (Gunter, Liebovitz & Malin 2011).

First, to start the authentication process, one must have an existing identity in the target entity. Then the entity verifies that the user who is attempting to sign-in, truly owns the identity registered to the entity. A successful authentication begins a session which lasts for a specified time period or until the user signs out. The definition of a session from an online session management guide published by The Open Web Application Security Project (OWASP) (2020) is following:

> "During a session the session ID or token is temporarily the equivalent to the strongest authentication method used by the application, such as username and password, passphrases, one-time passwords (OTP), client-based digital certificates, smartcards, or biometrics (such as fingerprint or eye retina)."

This definition introduces us to the authentication methodologies with varying integrities, which are often divided into three groups. The first group consists of credentials and PIN-codes, things that the user can remember. The second group consists of physical items such as a smart card, credit card, mobile phone or a one-time pad. The final group includes the biometric authentication methodologies: a fingerprint, iris patterns or some other unique feature. The authentication is considered strong when at least two methodologies from different groups are present concurrently in terms of Multi-Factor Authentication (MFA). (Linden 2017).

Machine to Machine (M2M) authentication is often executed using a cryptographic protocol based on either symmetric or asymmetric key pairs. Because of the complex nature of the encryption algorithms to maintain cryptographic security of the ciphertext, the decryption keys may become extremely long. Therefore, the cryptographic protocol cannot be adopted for human authentication. Anyhow, cryptography can be utilized in the second group of authentication methodologies. For example, a smart card can first authenticate its holder with a PIN code, then a server authenticates the smart card using a cryptographic protocol. Such a process also fulfills the MFA requirements. (Linden 2017).

2.1.3 Cryptography

Cryptography is the use of encryption and decryption techniques, which are processes for making data, a message for example, unreadable by scrambling and turning it back into intelligible form by the person or persons who are meant to receive it (Electronic Frontier Foundation 2018). Traditional, symmetric key, encryption systems use the same secret, or key, to encrypt and decrypt data. Therefore, it is useful for sharing information between a set of people who are all authorized to access it. Advanced encryption standard (AES) and triple data encryption algorithm (3DES) are popular symmetric key encryption algorithms, ciphers.

Public key encryption uses two keys: one for encryption and another for decryption. The encryption key is public and can be openly distributed, but the decryption key is only known to the owner and it must be kept private to maintain security. For example, in a messaging system that utilizes public key cryptography, any person can encrypt a message using the receiver's public key, and that encrypted message can only be decrypted with the receiver's private key. For example, RSA is a widely used asymmetric key cryptosystem named after the inventors Rivest, Shamir, and Adelman. Symmetric and asymmetric encryption systems are presented in the figure below (Figure 1). (Linden 2017).

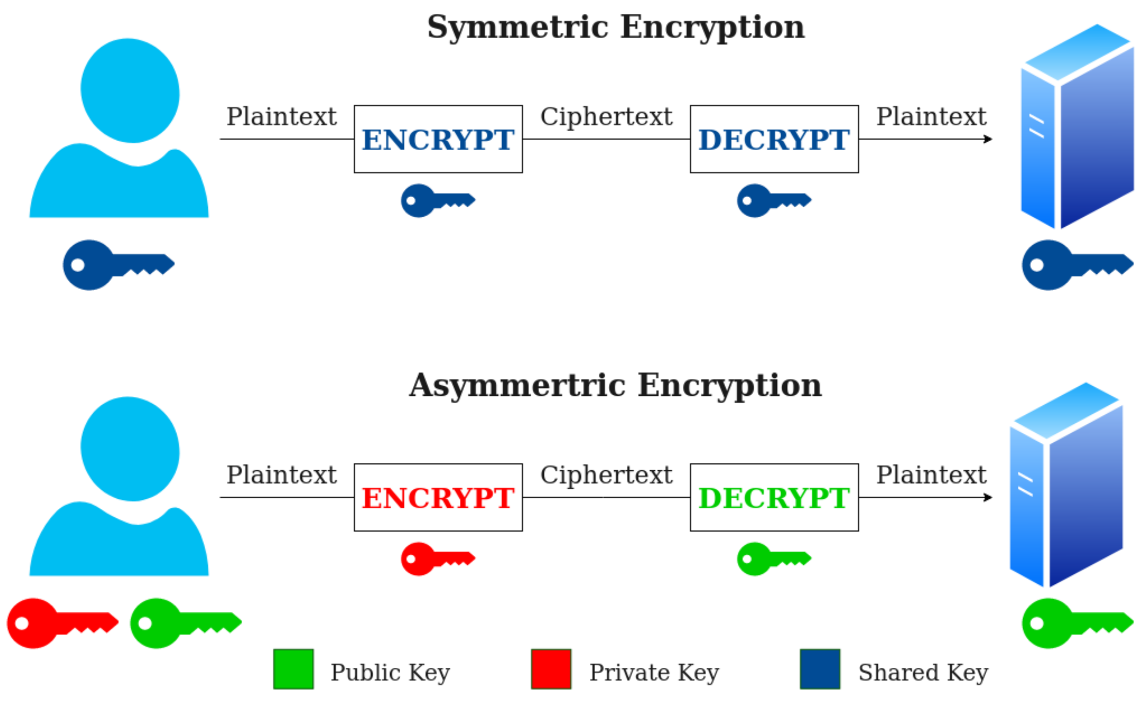


Figure 1. Description of symmetric and asymmetric encryption systems.

The characteristics of public key cryptography allows the creation of a digital signature, which gives the receiver of a message a reason to believe the message was sent by the claimed sender and has not been modified during transport. The digital signature is created by regularly encrypting a message with the receiver’s public key, and then encrypting the hash with the sender's private key and appending it to the original message. The receiver can then use the sender’s public key to decrypt the digital signature. If the hash of the original message completely matches with the decrypted digital signature, the authenticity of the message is verified. The whole process is described in the figure below (Figure 2). (Sectigo 2020).

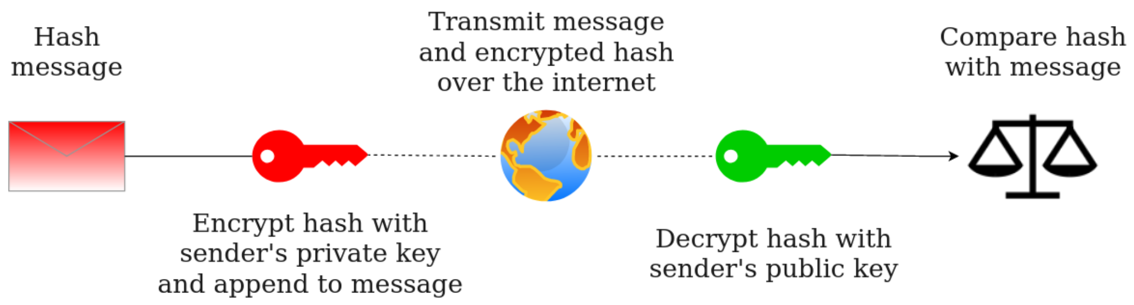

Hash message
Transmit message and encrypted hash over the internet
Compare hash with message
Encrypt hash with sender's private key and append to message
Decrypt hash with sender's public key

Figure 2. Process to validate the authenticity and integrity of a message.

These distinctive features made public key encryption become a fundamental part of modern applications and protocols to provide security. It is used in the Transport Layer Security (TLS) standard, which keeps internet connections secure and safeguards any sensitive data being transported between systems over the internet. Therefore, any confidential information should only be sent to a server over a TLS protected connection, Hypertext Transfer Protocol Security (HTTPS) for example. It is also essential to encrypt any user credentials before storing them to a database in the server side to prevent the plaintext credentials from leaking in case of a security breach.

2.2 Client-server communication

Clients and servers communicate over networks by sending small packages which are combined into individual messages. Hypertext Transfer Protocol (HTTP), or its secure version HTTPS, is the foundation of any communication on the internet. It is a client-server protocol, where the messages sent by the client are called requests and the messages sent by the server are called responses. HTTP is an application layer protocol where the communication, requests and responses, is established and maintained through Transmission Control Protocol (TCP). It works closely with the Internet Protocol (IP), which defines how the packages are sent between the client and the server. HTTPS, the secure version of the protocol, is sent over a TLS encrypted TCP connection. Due to its extensible nature, HTTP is used to fetch Hypertext Markup Language (HTML) documents, images and videos or to post content such as information entered to a form. (MDN Web Docs: HTTP Headers 2019).

2.2.1 Components

HTTP-based systems have three components: client, web server and proxies. The client is a tool, usually the web browser, that initiates the request. Take a web page for example, the browser first fetches the HTML document, parses it and makes additional requests to fetch Cascading Style Sheets (CSS) information and sub-resources of the page. Then the browser combines the resources and presents a complete web page to the user. The execution scripts of the page can later make additional requests based on user actions. (MDN Web Docs: HTTP Overview 2019).

A Web server can refer to hardware or software, or a combination of both. The hardware side means a computer, like a virtual machine, that stores the website's files. It is a device connected to the internet and supports physical data interchange with other devices. On the software side, it is a piece of software that allows clients to access the hosted files. At bare minimum, a server needs to understand Uniform Resource Locators (URLs) and HTTP requests to be accessible and able to deliver requested content to the client. On the opposite, a web server can be a complex entity, consisting of a huge collection of servers, which appears to be only a single machine because of a reverse proxy. (MDN Web Docs: HTTP Overview 2019).

Proxies are machines that lay between the client and the server to relay HTTP messages. They often operate at a lower level becoming transparent at the HTTP layer. Proxies can either modify the messages or forward them to the original destination without any altering. Proxies may perform functions such as caching, filtering, load balancing, authentication and logging. Load balancing is managed by a reverse proxy sitting in front of web servers, that distributes client requests in a manner that maximizes capacity utilization and ensures no one server is overloaded. Proxies provide security and anonymity in contrast: a reverse proxy hides the identity of a web server and a forward proxy hides the identity of a client. This functionality is illustrated in the figure below (Figure 3). (Nginx: Reverse Proxy 2020).

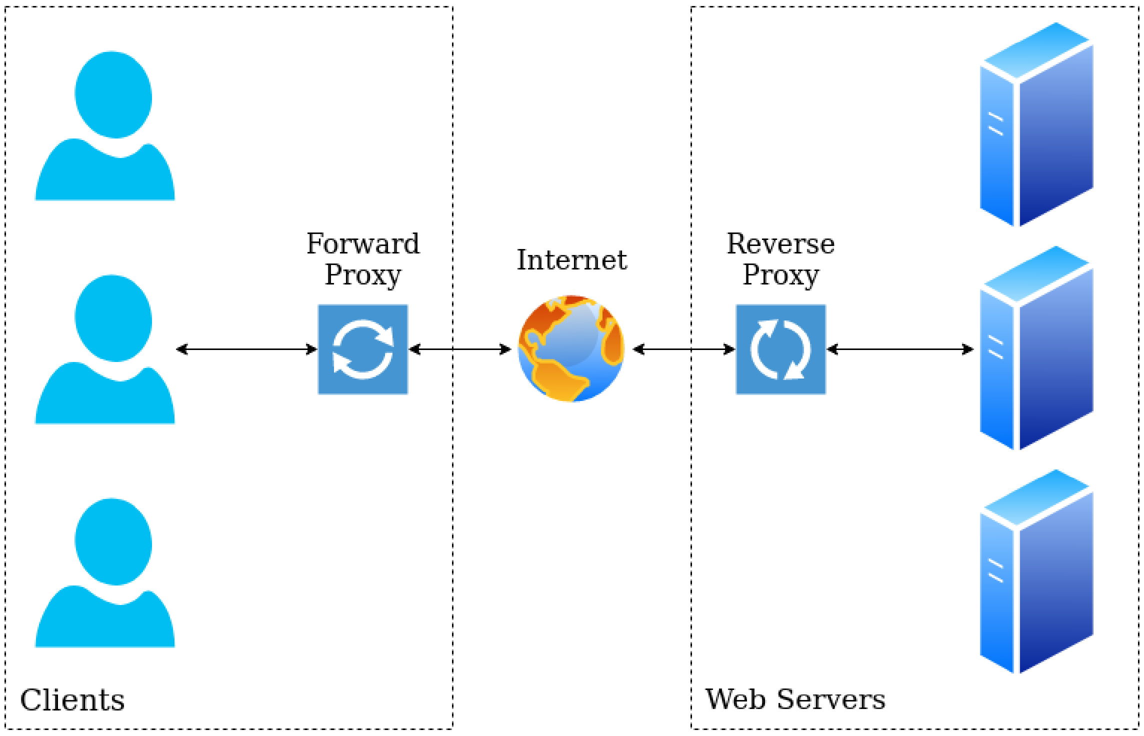


Figure 3. Illustration of a forward proxy and a reverse proxy.

2.2.2 Basic aspects

HTTP is simple and human readable by design for easier debugging and decreased complexity. Because of its stateless nature, there is no link between two HTTP requests being successfully carried out on the same connection. Anyhow, HTTP cookies allow the use of stateful sessions. Cookies are added to the header of a request, allowing creation of a session where all the requests share the same state. Basically, the header is a slot in the HTTP message where the client and the server can pass additional information.

The headers make HTTP extensible as they provide more control and functionality. Common controllable features include caching, relaxing the origin constraint, authentication, proxy and tunneling and sessions. Caching means the headers can be used to inform a special type of cache proxies to ignore certain stored documents. Relaxing the origin constraint is a broader subject which points to removal of the same origin constraint, meaning that web pages can source information from different domains. Controlling authentication means proving user identity via headers like WWW-Authenticate or the use of cookies. Proxies and tunneling can be controlled by identifying the protocol (HTTP or HTTPS), original host IP and original client IP and adding the information to the header. (MDN Web Docs: HTTP Headers 2020).

2.2.3 Messages

The HTTP flow is simple, and it performs in the following steps: First a reliable TCP connection is opened between the sender and the receiver to ensure no messages are lost. Then the actual HTTP request is sent with required information. Finally, a response is received from the server and the TCP connection is closed or reused for another request. (MDN Web Docs: HTTP Overview 2019).

First element of a request is an HTTP method. GET, POST, PUT and DELETE are typical methods with self-explanatory names. Typically, the methods are used for fetching or modifying web resources. Second element is the path of the resource to modify. Path means the URL of the resource, which can be stripped from the protocol, domain and port. To fetch a resource from the root of a domain, one would simply set *"/"* as the path. Then comes the HTTP version, optional headers and possibly a body, which contains the resources to be sent with the request. (MDN Web Docs: HTTP Overview 2019).

The structure of a response is comprehensive to a request. It has the version protocol, headers and an optional body. The differentiating factor is a status code, which indicates if the request was successful, or not, and why it failed. The status code can range from

100 to 599 and they are grouped in five classes: information responses (100-199), successful responses (200-299), redirects (300-399), client errors (400-499) and server errors (500-599). There is also the status message, which is a short description of the status code. (MDN Web Docs: HTTP Overview 2019)

2.3 Single Sign-On

SSO is a property of IAM that enables the use of one set of login credentials to access multiple systems within a single organization. This requires the existence of a single point of authentication, the IdP. Basically, it is a server that manages user authentication. Whenever a user tries to log into an application, instead of providing credentials to the SP which is the server hosting the application, the IdP passes a token and user attributes to the SP after a successful authentication. The token and user attributes, claims, are provided using standard identity protocols such as Security Assertion Markup Language 2.0 (SAML 2.0), OpenID Connect (OIDC) or Open Authorization 2.0 (OAuth 2.0). These protocols are further introduced in the following chapters. The figure below (Figure 4) describes how the IdP and SP function when a login process is initiated. First, the client authenticates with the IdP via username and password or some other mean. Then the IdP passes the token and user attributes to the SP, using its protocol of choice. Generally, the SP will save the values of the attributes to a corresponding user object and update it on every login. (Sharma & Dave 2015).

Security personnel are often concerned that SSO creates a security risk by having the same credentials across multiple services. However, there are many aspects of SSO that counteract the concern. After careful consideration the less secure systems can be excluded from the SSO environment. A great benefit is the ability to enforce more stringent password restrictions across the environment from a central IdP. These restrictions may include a minimum length, password expiry time and invalid dictionary lists. Even specific Operating System (OS) and application restrictions can be brought in line with the IdP configuration. With only one password to remember there is less password fatigue and it is less likely written on a piece of paper. Some IdPs allow end users to reset their password after validating their identity which reduces help desk costs and risk of social engineering. The reduced likelihood of a failed login process is another aspect leading to lower help desk workloads. Capability to quickly deactivate an employee's access to multiple systems and the possibility to enhance security by enabling multi factor authentication are other major benefits of SSO. Even with all the security enhancing features it is important to remember that even the strongest security databases have exploitable weaknesses and organizations must have tools for monitoring user sign-on actions to detect possible intrusions. (Kelly 2020).

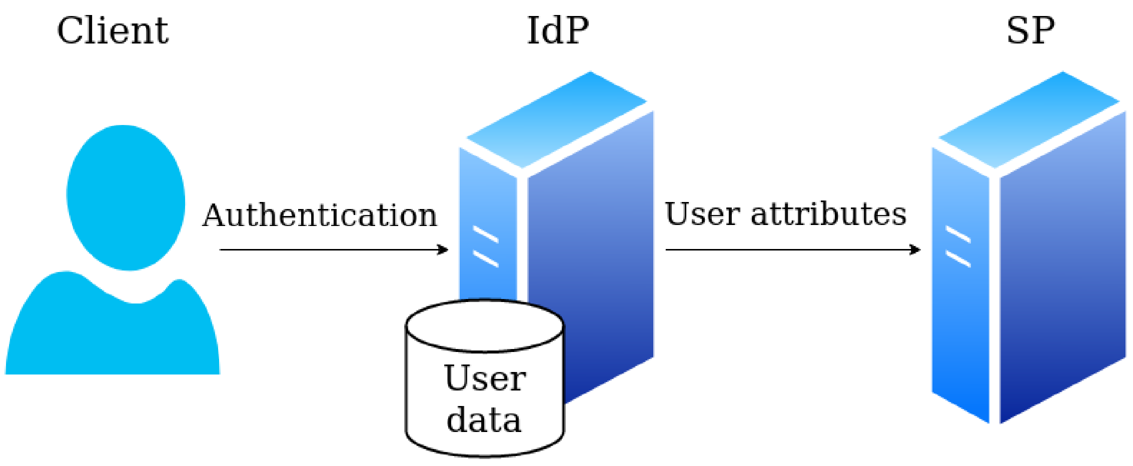


Figure 4. Basic components of SSO and a schema of the login process.

2.3.1 SAML 2.0

The SAML 2.0 protocol is based on Extensible Markup Language (XML), which is a format for encoding documents in both human and machine-readable form. SAML 2.0 was approved as Organization for the Advancement of Structured Information Standards (OASIS) standard in 2005 has been widely used by enterprises ever since. In align with the other protocols, SAML 2.0 allows the management of user identities and authorizations across multiple applications.

Essentially, SAML 2.0 has three components: assertion, protocol and bindings. Assertions are requests made by SP to the IdP and the protocol specification defines three different kinds of assertion statements: authentication assertion, attributes assertion and authorization decision assertion. Authentication assertion declares identity of user, attributes assertion contains details about the user, and authorization decision assertion, a request to allow the assertion subject to access the specified resource has been granted or denied (IBM 2020). Protocol component defines how to respond to the SAML 2.0 requests. Bindings component states whether the response should be loaded into a HTTP POST or HTTP Redirect request. (Sharma et al. 2015).

A simple use case is applied for every protocol, where Bob, on his browser, tries to access a protected resource on TraSim, a list of ongoing situations. In the figure below (Figure 5), SP is the server hosting TraSim and IdP is the server where an identity manager is installed, Azure Active Directory for example. In SAML 2.0 context, Bob is the principal, the user SP is trying to authenticate and learn about. After trying to access the protected resource, the SP redirects Bob to the IdP with an authentication request on the query parameter. The IdP uses the authentication request to identify the service provider, and unless a valid session already exists, Bob is prompted to enter his credentials. After the IdP successfully authenticates Bob, it creates a new session and sends a response back to the SP. The response states that authentication was successful, and it includes assertions about the principal for creating a new user to the service or updating an existing one. (Cheung 2016).

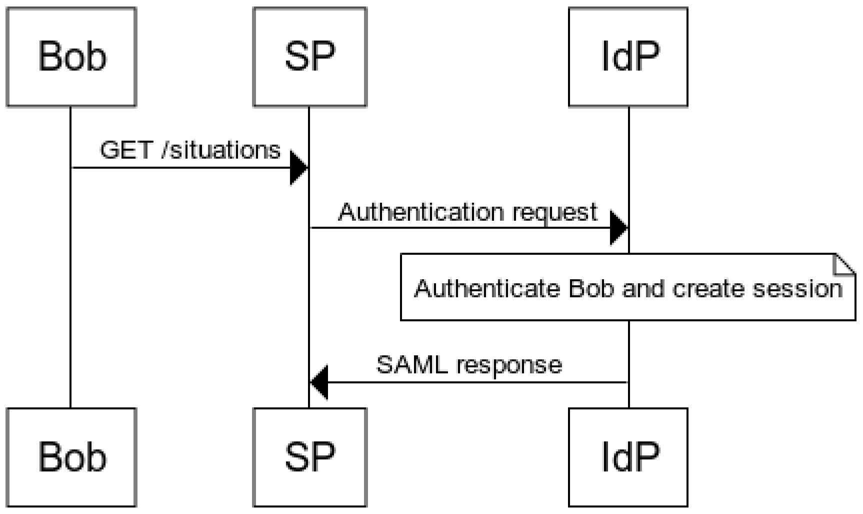


Figure 5. Sequence diagram of SAML 2.0 single sign-on flow.

2.3.2 OAuth 2.0

The OAuth 2.0 protocol can use four different process flows known as authorization grant types: authorization code grant type, implicit grant type, resource owner password credentials grant type and client credentials grant type. First of these types, the authorization code grant type will be under the loop, as it is a popular one. The OAuth 2.0 authorization framework defines a web Application Programming Interface (API) called authorization endpoint. This endpoint handles authorization requests and responses.

An authentication response has an access token as its parameter. The token is in JSON web token (JWT) format which contains three parts: header, payload and a signature. The header has information about the type of the token and the algorithms used to secure its contents. The payload contains a set of claims (statements about the permissions that should be allowed), and other information such as the expiration time of the token. The signature is used to validate whether the token is trustworthy by means that are described in the digital signature chapter. Following paragraph describes the basic OAuth 2.0 authorization code flow presented in the figure below (Figure 6). (OAuth 2020).

When Bob tries to access the protected resource, SP sends an authorization grant request, and Bob is redirected to the IdP. The SP receives the authorization grant after Bob allows the permission. The grant represents Bob's authorization, and the SP sends it as a parameter of the authentication request to receive an access token. Provided the token is valid, it can be used to fetch Bob's resources from the IdP.

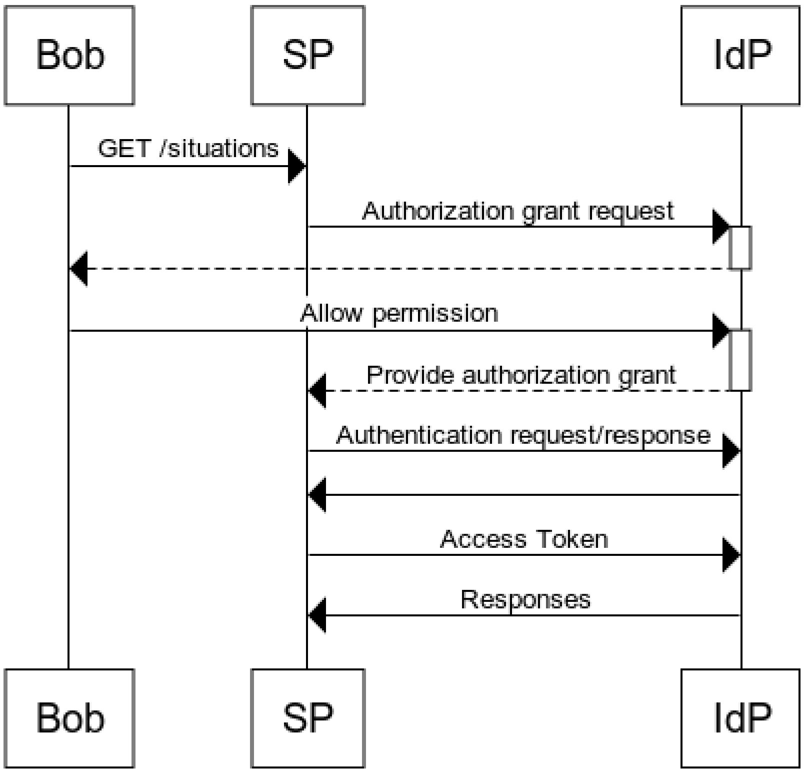

Bob
SP
IdP
GET /situations
Authorization grant request
Allow permission
Provide authorization grant
Authentication request/response
Access Token
Responses
Bob
SP
IdP

Figure 6. Sequence diagram of basic OAuth 2.0 authorization code flow.

2.3.3 OpenID Connect

OIDC is a specification as to how to issue ID tokens. It is a simple identity layer on top of the OAuth 2.0 protocol allowing the verification of user identity and obtaining their attributes in REpresentational State Transfer-like (REST) manner. Compared to OAuth 2.0, OIDC performs many of the same tasks, but in a way that is API-friendly, and usable by native and mobile applications. It defines optional mechanisms for robust signing and encryption, roughly meaning that a ciphertext cannot be decrypted under a different key. (OpenID 2018).

In contrast to SAML 2.0, OIDC satisfies the same use cases with a simpler, JavaScript Object Notation (JSON) and REST based protocol. Also, SAML does not have mobile or native support, it is only designed for web applications. While all the protocols tend to use different naming conventions for the service and identity provider, this thesis will stick to the SP and IdP naming convention. For example, the OIDC equivalent for SP is Relying Party and the IdP equivalent is OpenID Provider.

The authorization endpoint, described in the OAuth 2.0 chapter, requires *response_type* as a mandatory request parameter, for which OIDC has defined flows to issue ID tokens by extending the specification of the *response_type* request parameter. This extension means that OIDC has added *id_token* and none as possible values for the *response_type* parameter in addition to code and token. OIDC has also allowed the parameter to be any combination of code, token and *id_token*. In practice, this means there are eight different combinations of OIDC authorization and authentication flows. The SSO flow of a case, where the value of *response_type* parameter equals to code, is presented in the figure below (Figure 7) and described in the following paragraph. This case is chosen because it is the most popular one. (Kawasaki 2017).

When Bob tries to access the protected resource, he is redirected to the IdP and a client ID, specific to SP, is passed via the same request. Bob gets redirected back to the SP with an authorization code after successful authentication. Then the SP sends the received authentication code, client ID and client secret to an authorization endpoint, which responds with an access token. Finally, the access token can be used to fetch user attributes from a *userInfo* endpoint. Both endpoints are usually tied to the IdP.

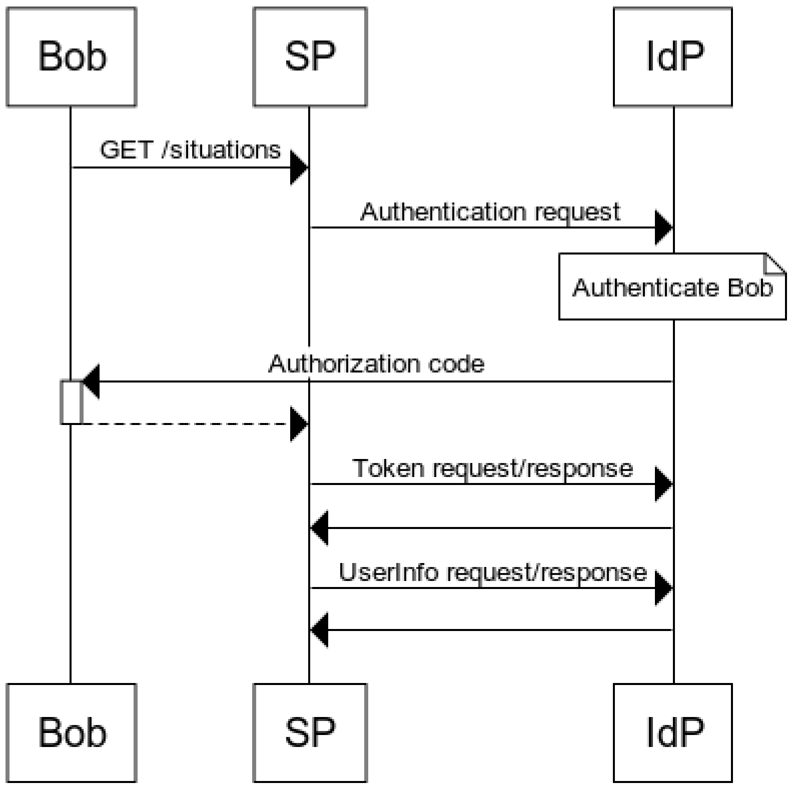


Figure 7. Sequence diagram of OIDC single sign-on flow.

2.4 Cloud computing

"Simply put, cloud computing is the delivery of computing services—including servers, storage, databases, networking, software, analytics, and intelligence—over the Internet ("the cloud") to offer faster innovation, flexible resources, and economies of scale. You typically pay only for cloud services you use, helping lower your operating costs, run your infrastructure more efficiently and scale as your business needs change."

That is how Microsoft (2020) defines cloud computing in an article that acts as an introduction to their own cloud computing platform called Azure. Basically, the term describes data centers where one may purchase computing services without the burden of managing their own computing facilities. In its modern form, cloud computing was introduced in August 2006, when Amazon created Amazon Web Services (AWS) and introduced Elastic Compute Cloud (EC2) interface (Amazon 2006). That is the single point in history which, according to an online article *Shift to Service Based Model* by Gartner (2008), started the shift from company-owned hardware and software assets to service-based models and will cause a dramatic change in how IT products are consumed. AWS was soon followed by Google App Engine, OpenNebula, Microsoft Azure, OpenStack, IBM SmartCloud, Oracle Cloud and Google Compute Engine.

2.4.1 Traditional service models

National Institute of Standards and Technology (NIST) has declared three standard service models for cloud computing providers: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and the previously mentioned SaaS. The models are often presented as a pyramid model as in Figure 8, but one may also exist without the other. SaaS can be implemented directly on a physical machine without the need of PaaS or IaaS layers, or on the contrary, a program can be directly accessed on IaaS without SaaS bundling.

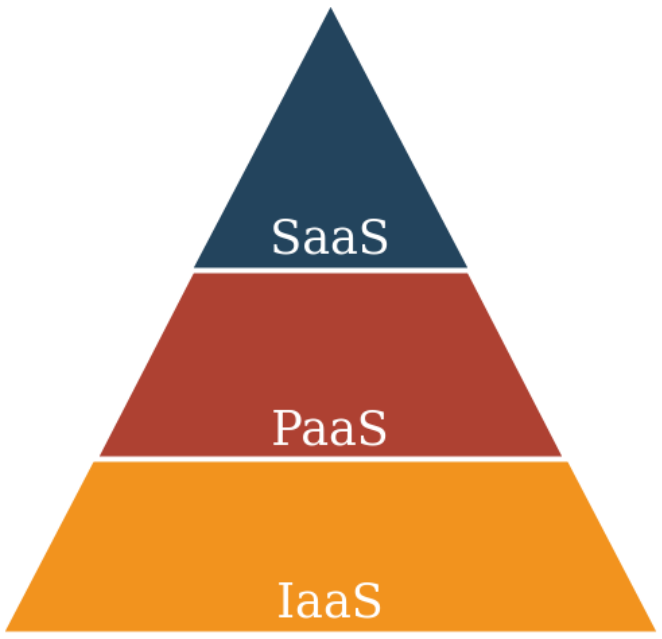


Figure 8. Pyramid design of traditional cloud computing service models.

IaaS, at the bottom of the pyramid, is the computing infrastructure of a data center, or a cluster of data centers, managed over the internet. IaaS can be quickly scaled up and down based on demand without worrying about the physical servers and networking infrastructure. The cloud computing platforms, Azure for instance, can be called IaaS providers as they manage the physical infrastructure. Then the customer, or their IT administrator, must only purchase, configure and manage their own software (operating systems, middleware and applications), and of course, pay the bills. (Microsoft: IaaS Overview 2020).

PaaS, at the middle of the pyramid, is a platform where consumers may deploy consumer-created applications onto a cloud infrastructure. The twist is that the applications are created using tools languages, libraries, services and tools supported by the PaaS provider. Unlike with IaaS, the consumer has no control over the cloud infrastructure apart from possible configuration settings for the hosting environment. The benefits of PaaS are automatically scaling storage, computing and licensing resources. Famous example of a PaaS provider is Heroku by Salesforce or Elastic Beanstalk by AWS. (Mell & Grance 2011).

SaaS, at the top of the pyramid, is a service available to a consumer on a cloud infrastructure. The consumer has no control over any of the underlying infrastructure or application specific technical solutions. Therefore, there is zero need of management for the platform where the application runs. Usually the user specific settings are the only configurable options which leaves the IT departments at ease. Oftentimes IAM of the service, one of the key topics of this thesis, is the important task that remains for them to grasp. The scalability trend continues with SaaS, as the service is often subscription or license based, and easily adjustable as the number of users varies. (Liu et al. 2018).

Anyhow, SaaS also has its drawbacks. Unless there is an SSO or a Federated Identity Management (FIM) implementation, data of the users must be stored on the SaaS providers database on their server. This is problematic especially because of the user credentials.

According to cybersecurity institute Sans, incorrect use, storage and transmission of such credentials could lead to compromise of very sensitive assets and be a springboard to wider compromise within an organization. The SaaS provider is solely responsible for the credential management chain.

2.4.2 Identity as a Service

An article published by PingIdentity states that while companies are embracing cloud and mobile technologies, they are moving beyond traditional network boundaries and the capabilities of their legacy IAM solutions. Identity as a Service (IDaaS) is a cloud based IAM offering built and operated by a third-party provider. It allows companies to use SSO, authentication and access controls to enable secure access to their SaaS applications.

The basic idea behind IDaaS is to support the SaaS applications. This is effective for small and medium sized companies that already are strongly attached to the cloud. On the enterprise level, with complex IT environments including a mix of on-premises and SaaS applications, IDaaS is typically used as an extension to existing IAM infrastructure. Therefore, the IDaaS provider must be able to build a bridge to existing user directories for authentication, integrate with multiple applications hosted on varying locations and provide access management for web, mobile and API environments. Some of the leading IDaaS vendors include OneLogin, Centrify, Microsoft and Okta. (PingIdentity 2020)

A press release by Gartner (2019) predicts that by 2022, 80 percent of IAM purchases by global midsize and larger organizations will use the IDaaS delivery model. They believe this is due to the ease of implementation and rapid time to value of IDaaS offerings. Especially companies that favor SaaS adoption and do not consider the operational management of IAM functionality as their core business highly value IDaaS. Besides the steady movement of application to cloud, the combination of configurable, rather than customizable, functional offerings and modern application architectures is causing a substantial portion of the market adaptation towards IDaaS.

2.5 Directory services

Following sections examine the directory services developed by Microsoft, Active Directory and Azure Active Directory.

2.5.1 Active Directory

Active directory is a central repository for information of all resources that exist in an organization’s network including users, groups, devices, programs and documents. It is used across most of today’s major organizations as a primary tool for managing the organization’s information. For example, a network administrator can create a group of users and give them access to specific directories or services. Traditionally AD comes with a Windows Server OS and it is designed to work with the Windows ecosystem.

The structure of AD consists of domains, trees and forests. Objects, such as users and devices, that use the same database can be grouped into a single domain. Multiple domains can be combined into a single group called a tree. Multiple trees can then be combined into a collection called forest. AD is a massive entity and it provides several services, the Active Directory Domain Services (AD DS), including domain services, certificate services, lightweight directory services, directory federation services and rights management. (Microsoft: Identity 2017).

AD DS is a centralized data store that manages login authentication, search functionality and other communication between users and domains. The certificate services create, distribute and manage secure certificates. Lightweight directory services support directory-enabled applications using Lightweight Directory Access Protocol (LDAP), which is a protocol for making directory information available over the internet. Directory federation services provide the SSO functionality to authenticate users in web applications in one session. Finally, the rights management services protect copyrighted digital content.

Active Directory Federation Services (ADFS) was born out of the increasing need to find a way to authenticate and authorize users to web applications. It works in conjunction with AD to establish a trust relationship between a Windows domain controller and a service. ADFS uses a claims-based access-control authorization model to authenticate users via cookies and SAML. All through ADFS is a free feature on Windows Server, it requires the actual Windows Server license which can be expensive and a server for hosting it. ADFS can also be a burden to integrate with the cloud or mobile application and it requires IT resources to install, configure and maintain. (Lujan 2019).

LDAP is another traditional way to authenticate users on AD networks and implement SSO. It is a lightweight subset of X.500 directory access protocol and has been around since the early 1990s. As a lightweight protocol LDAP runs efficiently on systems and it gives great control over authentication and authorization. Whereas ADFS is focused on Windows environments, LDAP can accommodate other environments including Linux and OS X. However, comprehensively to ADFS, implementing LDAP is a formidable technical process requiring a significant amount of work and deep technical skills. (DeMeyer 2019).

Transition to the cloud has made a huge influence on the way modern organizations authenticate users. Any reliance on on-premises functionality has become a hindrance, rather than a help. A lot of organizations are looking to free themselves from legacy methods, such as the ADFS and LDAP. There is no denying that these methods still have their time and place, but the huge advantage of the cloud is the flexibility to choose just the right authentication method for a given situation. Microsoft launched Azure AD in 2015, which is an online service for providing identity management. The following chapter will introduce Azure AD and a brief explanation on how to use it for authenticating web applications. (Guest 2019).

2.5.2 Azure Active Directory

Azure is not limited to compute and storage features, as it also provides a variety of services for security, virtual networking, communication mechanism and caching strategies. Primarily Azure AD implements a cloud service for web application authentication, SSO and user management. It is often used as a standalone cloud directory service for implementing SSO between a corporation and SaaS applications and synchronizing directories of the SaaS applications. Basically, Azure AD is a REST-based extension of AD, which is regularly used alongside on-premises AD. (Tejada, Bustamante & Ellis 2015).

Even though Azure AD has similarities to AD DS, there are many distinguishing features between the two. It is essential to notice that using Azure AD is not equivalent to on-premises AD or to deploying an AD domain controller on a virtual machine on Azure and adding it to on-premises domain. Above all, Azure AD is an identity solution, designed for accepting HTTP and HTTPS communications. These communication restrictions deny the use of Kerberos and prevent Azure AD from being queried with LDAP. Instead, the previously introduced protocols SAML 2.0, OAuth 2.0 and OIDC can be utilized. Despite its name, Azure AD does not only provide identity management for Microsoft services, or services registered and listed in their official application gallery. Third party service providers can benefit from leveraging Azure AD if their applications use supported communication protocols. (Altili 2017).

To go more in depth with the differences of AD and Azure AD following use cases can be considered: An application has two versions, on-premises and cloud. The on-premises version uses AD for identity provision, whereas the cloud version uses Azure AD in the same way. As previously mentioned, Azure AD has users and groups just like on-premises AD, and user permissions can be assigned on a group level. Azure AD also adds features like self-service password reset, self-service group membership and multi-factor authentication. There is no subscription requirement for Azure AD tenants, so it is free to use with certain limitations. In some cases, where a company can solely rely on Azure AD, it removes the obligation of maintaining an on-premises AD reducing costs. (Jensen 2018).

Azure AD users can be sourced from multiple locations. First option is Azure AD sourced users which means creating the users by hand in the directory. Second option is synchronizing user accounts from on-premises AD, or Windows Server AD by using a tool called Azure AD Connect. This way all users will have the same information as in the on-premises AD. The final source of users is guest users. If there is a demand to allow an external user to access an application registered to a corporation, a guest user can be invited and assigned the permission to access the application. The figure below (Figure 9) contains example architecture of an on-premises AD which has been migrated to cloud (Azure AD) and uses multiple SSO solutions with different identity protocols. Notice that ADFS only supports SAML 2.0 and the applications can also be hosted on-premises. (Jensen 2018).

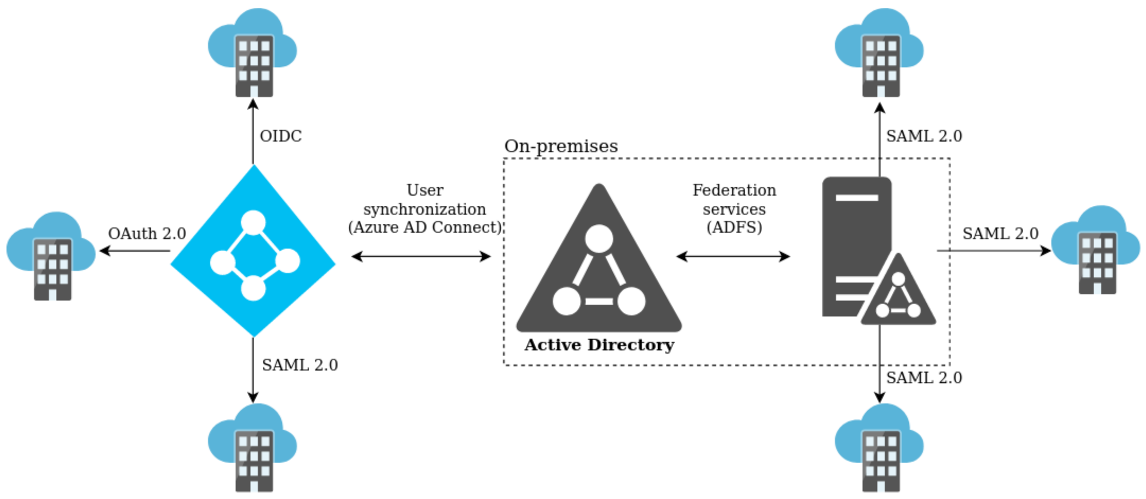


Figure 9. System architecture with ADFS and Azure AD SSO solutions.

3 Application specific information

This chapter introduces the technology stack TraSim is built with and how it is hosted and served to the clients. All this information acts as a base for setting the requirements for a successful integration. The chapter begins with describing the basics of Linux OS and Docker platform that together form the hosting environment. Then focus shifts to Nginx and Gunicorn which are used for serving the application to clients. To speed up the serving process a caching system, Redis, is utilized. Dynamic content is stored in a PostgreSQL database. Finally, the main building blocks of TraSim, programming languages Python and JavaScript are introduced. These languages are extended to frameworks and libraries such as Django and React to simplify the development process and maintenance of the application.

3.1 Environment

TraSim is a multi-container Docker application that is hosted on a virtual machine powered by a Linux OS. To understand this architecture, one must have insight about the technologies. While they are both broad subjects, the basic features are explained below.

3.1.1 Linux

Linux has been around since the mid-1990s and has since reached a huge user-base. It is used everywhere: phones, thermostats, cars, refrigerators and it runs most of the internet, stock exchanges and all the world's top supercomputers. As an OS just like Windows or OS X, Linux manages the communication between software and hardware, without which the software could not function. The perks of being less vulnerable to cyberattacks, zero cost of entry, simplicity and reliability have enabled Linux to become as popular as it is today. (Linux 2020).

Linux has several different versions to suit a variety of users and use cases. These versions are called distributions such as Ubuntu, Debian and Fedora targeted for desktop use and Ubuntu Server, CentOS and Red Hat Enterprise Linux for server use. Most of the server distributions are free, but Red Hat Enterprise Linux comes with an associated price which in the other hand does include support. When choosing the server-only distribution one must decide whether a desktop interface is needed or is a command line enough. Having no Graphical User Interface (GUI) means the server will not be stalled by loading graphics, but it requires a solid understanding of the Linux command line. While CentOS offers everything that is required of a server out of the box, it is possible to start up with a desktop distribution like Ubuntu and add pieces as required to support server functions. (Linux 2020).

Most of the Linux distributions include an app store: a centralized location for installing software. While the app store has a different name on the distributions, it has the same purpose on them all. For GUI-less servers, the applications are installed via command-line tools. Whereas Debian-based distributions install applications with a tool called apt-get, Fedora-based distribution takes use of yum. Just like the app stores, these tools work similarly to each other despite the different names. To install an application via these command line tools, one must add a special sudo keyword to the top of the command. Sudo stands for “superuser do” and it prompts for a password in prior to executing a command to issue that the current user has super user privileges which are required in order to install software. By default, all commands are run as a non-privileged user due to security concerns. (Linux 2020).

3.1.2 Docker

The *Overview* chapter of Docker’s formal documentation (2020) describes itself accordingly:

> "Docker is an open platform for developing, shipping, and running applications. Docker enables you to separate your applications from your infrastructure so you can deliver software quickly. With Docker, you can manage your infrastructure in the same ways you manage your applications. By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production"

Most essentially, Docker allows one to package and run an application in a loosely isolated environment called a container, which could be imagined as a lightweight Virtual Machine (VM). But unlike VMs, containers run on the host machine's kernel and the extra load of a hypervisor, the process that separates OS and applications from underlying physical hardware, is not required. According to Docker, this means *"...you can run more containers on a given hardware combination than if you were using virtual machines".* Docker provides tooling and a platform to manage containers throughout their whole life cycles. Containers can be used when developing an application and its supporting components, distributing and testing the application and deploying it to the production environment. Therefore, containers are great for Continuous Integration (CI) and Continuous Delivery (CD) workflows. (Docker 2020).

The containers are created from images, which are read-only templates containing instructions on how to create a certain container. An image is often based on another image, which is customized to fill the requirements of a service. For example, one may build an image based on the Ubuntu image, install the dependencies via the apt-get tool and copy source code and configuration files of an application to make it run. An image is built by creating a Dockerfile that uses YAML syntax to define the steps needed to create an image and run it. Each instruction in the file creates a layer in the image. If the file is later modified, only the layer that has changed needs to be rebuilt. Another type of YAML file, docker-compose, exists to define and run multi-container Docker applications. It allows multiple services to be run with a single command. (Docker 2020).

Docker uses a client-server architecture, where a client talks to a daemon, which builds, runs and distributes containers. The client and daemon communicate using a REST API over UNIX sockets or a network interface. The daemon listens for Docker API requests and manages Docker objects such as images, containers, networks and volumes accordingly. The client is the principal way how users interact with Docker. Then there are Docker registries that store Docker images. For example, Docker Hub is a public registry available for anyone to store and retrieve images from. The registries can also be private, and they can be utilized in a similar manner to general software development platforms such as GitHub that use a distributed Version Control System (VCS) Git to efficiently handle projects. (Docker 2020).

3.2 Web server

Software side of the web server relies on Nginx as a HTTP server, load balancer and reverse proxy which is then coupled with a Python Web Server Gateway Interface (WSGI) server called *"Green Unicorn",* Gunicorn.

3.2.1 Nginx

According to the official documentation, Nginx is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server, originally written by Igor Sysoev. An internet monitoring company Netcraft stated in February 2020, that Nginx served or proxied 25.68 % of the busiest sites and had the biggest market share of all sites compared to other web server developers. The indicated load balancing abilities run side by side with the reverse proxy component of Nginx. Both act as intermediaries in the communication between clients and server, performing functions that improve efficiency (Nginx: About 2020).

Load balancing is often enabled on busy sites that need multiple servers to handle the high volume of requests efficiently. Having multiple servers also removes a single point of failure, therefore increasing reliability. All the servers commonly host the same service, which simplifies the load balancer's job to distribute the workload in a way that makes the best use of each server's capacity and results in the fastest possible user experience. Besides the faster response times, the load balancer decreases the number of error responses a client sees. The errors simply do not happen as often when the load balancer diverts requests from unhealthy servers to healthy ones. To quote Nginx's (2020) glossary about *Load Balancing*: *"the state of a server is checked via a sophisticated method in which the load balancer sends separate health-check requests and requires a specified type of response to consider the server healthy".*

Nginx uses an asynchronous event-driven approach to handle requests. This modular architecture provides a predictable performance even under a significant load. Nginx's HTTP proxy and web server features can handle over 10 000 simultaneous connections with efficient memory use, when 10 000 inactive HTTP keep-alive connections require 2.5MB of memory. The goal and a driving factor for Nginx development was to outperform the popular Apache web server. The goal was met as Nginx can handle approximately four times more requests per second than Apache, while using significantly less memory. Though there are some drawbacks as Nginx is less stable on Windows systems and harder to install and configure than Apache. Other domains Nginx is suitable for include serving static files, handling slow clients, terminating SSL and forwarding dynamic requests to Gunicorn. (Nginx: About 2020).

3.2.2 Gunicorn

A web server is an interface between the server and outside world, that allows clients to access the hosted files. Anyhow, it cannot directly talk to Django applications, or Python applications in general. Another interface is required between the web server and the application to run and serve it to the clients. To solve this problem, the Python standard called WSGI was developed. Having the standard promotes scaling abilities of web traffic by segregating the responsibilities between the servers and web frameworks. It also gives developers flexibility to swap out web stack components when necessary. From a variety of valid options, TraSim utilizes the WSGI server Gunicorn where Nginx passes requests according to its configuration. (Gunicorn 2020).

Gunicorn is designed for UNIX systems and it is compatible with various Python web frameworks. Gunicorn, or a WSGI server in general, runs on the web server as a separate container, that further on runs the actual web application. It translates the HTTP requests arriving from Nginx to WSGI compatible form, that can invoke a callable object of a module on a Python application. Then it translates the WSGI responses of the application back into HTTP format and directs them to Nginx. While Gunicorn can be used as the front-facing web server, it is not recommended as Gunicorn has security vulnerabilities, cannot do SSL termination and it is simply less advanced than an explicit web server. (Makai 2020).

3.3 Database and caching

Web services often require a database or multiple databases when dynamic content, data processing functionality or a user registration system is incorporated. While there are a multitude of options, TraSim uses a relational database PostgreSQL. Generally, a relational database is a set of formally described tables which allows the data to be identified and accessed in relation to another fragment of data in the database by using Structured Query Language (SQL) standard. Then, to achieve higher performance, a Redis cache is enabled for the PostgreSQL database.

3.3.1 PostgreSQL

PostgreSQL is a powerful relational database that extends the SQL standard combined with additional features. It is an open source project that dates back to 1986. PostgreSQL has a proven architecture that stores, processes and retrieves data accurately and consistently throughout its entire life cycle. The community behind PostgreSQL has made it compatible with all major operating systems and highly extensible via a robust feature set. For example, one may define custom data types and functions or write code from a variety of different programming languages without recompiling the database. PostgreSQL can be scaled to manage petabytes of data in production environments. (PostgreSQL 2020).

To make sure that concurrent operations generate correct results, PostgreSQL uses a method called Multiversion Concurrency Control (MVCC). This method gives each transaction a snapshot of the database, which allows changes to be made without affecting other transactions. This guarantees the validity of Atomicity, Consistency, Isolation and Durability (ACID) properties of the database even in the event of an error or a failure. Because of its stability and open source nature, PostgreSQL requires a minimum amount of maintenance and is free to use. Therefore, its total cost of ownership is low in comparison with other database management systems. For that reason, many companies such as Apple, Fujitsu, Red Hat and Cisco have built their products and solutions using PostgreSQL. (PostgreSQL 2020).

3.3.2 Redis

As an in-memory key-value store with high data access speed, Redis is a good option for enabling client-side caching. Basically, it is a technique that can increase scalability, storing speed and data availability of an existing PostgreSQL instance. It does this by exploiting the available memory in the application servers in order to store some subset of the database directly in the application side. Normally when data is requested from a server, it will query the database for information. But with client-side caching, the application will store the reply inside the application memory and there is no need for a database query. (Redis 2020).

Accessing data from the store is magnitudes faster than a standard database query and it prevents the database from being overloaded with exceedingly large amounts of queries from different clients. Caching is especially useful if the database has a lot of inefficient data models and queries. However, instead of introducing caching to bypass the problematic designs, they should be fixed to increase the overall efficiency of the application. It should also be investigated if there are specialized tools that would lead to the same result as caching but in a more sophisticated manner. (Redis 2020).

The problem with caching lies in deciding what to cache and when to refresh it. Basically, there are three ways to use Redis with a database. First option is to off-set some reads by caching commonly accessed data in Redis. Second option is to write data to Redis first and push it to the database later. Third option is to cut out the database entirely, as some data can live only in Redis. Caching should be taken into consideration if a query is complex, commonly used and it blocks the rendering of an UI. The query should also return data that can be slightly stale, and one can stand losing. To ensure the data on Redis is as up to date as possible, it must be refreshed based on cache invalidation policies. Redis can be configured to automatically expire an item after a set time period, meaning the item will be queried from the database. Data with high read rates and low write rates can be configured to be refreshed each time something is written. There is even a possibility to cache everything and gradually delete the data from the cache that is not getting read. (Redis 2020).

3.4 Programming languages

The programming languages Python and JavaScript, basic building blocks of TraSim are introduced in the following sections and further utilized in the *Frameworks and libraries* chapter.

3.4.1 Python

The description of the programming language from a topic called *The Python Tutorial* of Python's official documentation (2020) is quoted below. As the description includes a lot of terms, they are further explained in the following paragraphs to gain better insight about the language.

> "Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms."

Python can be considered simple and easy to learn because of its pseudo-code nature and simple syntax. Python is classified as Free and Open-Source Software (FOSS) and it has been created and is constantly improved by a community. Being a high-level language, one does not need to manage low-level details like memory management when writing a program. Due to its FOSS nature, python has been ported to work on most available platforms, for example Linux, Windows, OS X, FreeBSD and many more. Having an interpreted nature means that Python first internally converts the source code to a transitional form called bytecodes and then translates the bytecodes to native language of the machine. Following features were not mentioned in the description but Python is both extensible and embeddable, meaning a part of a Python program can be written in C or C++ and vice versa. (Swaroop 2003).

One of the greatest advantages of Python are the extensible libraries, both Python Standard Library (PSL) and Python Package Index (PyPI). The standard library is always available with a python installation whereas third party PyPI packages must be downloaded and installed by using the package installer pip. Web development is one of the many application domains that python is used in. Following web development choices are offered via PyPI: frameworks (Django and Pyramid), micro-frameworks (Flask and Bottle) and advanced content management systems such as Plone and Django CMS. These offerings are often dependent on Python's standard libraries that support many internet protocols such as HTML, XML, JSON, Email processing and File Transfer Protocol (FTP). (Python 2020).

3.4.2 JavaScript

JavaScript is an interpreted, asynchronous programming language commonly used in web development. It is a client-side scripting language, originally developed by Netscape, that is processed by the client's web browser rather than on the web server. The programs written with JavaScript are called scripts and they can be embedded to the HTML document of a website. While JavaScript is strongly influenced by Java, it is very different in this aspect. As JavaScript evolved, it became an independent language based on a specification called ECMAScript.

All the major browsers have their own JavaScript engines: Chakra in Internet Explorer and Edge, SpiderMonkey in Firefox and V8 in Chrome and Opera. As a side note, the V8 engine has also been expanded to Node.js, a JavaScript runtime that executes JavaScript code outside of the browser. Node.js unifies web application development as it enables the use of a single programming language for both server and client-side scripts. The engines work in a complex manner, but at the essence they first parse through the JavaScript code, convert it to machine code and execute. There are various small steps in the process which include a lot of optimization, analyzing and more optimization of the code. (Kantor 2020).

JavaScript has certain limitations and features, and its capabilities immensely depend on the environment it is running in. For example, Node.js allows the code to read and write arbitrary files and perform network requests. In-browser JavaScript’s abilities are limited due to security concerns. It cannot execute programs, read or write arbitrary files or interact with devices such as a microphone without permission. Moreover, in-browser JavaScript is designed as a safe language, that does not provide low-level access to memory or Central Processing Unit (CPU). It can perform actions related to manipulating a webpage and interacting with a user and a web server. These actions incorporate adding and modifying HTML content, reacting to user actions, sending requests and setting headers and storing data to client-side local storage. (Kantor 2020).

In recent years, defining a web page's structure in JavaScript instead of in HTML using frameworks such as React, Angular and Vue.js has become increasingly popular. Web pages structured in this manner, that dynamically rewrite their content instead of loading entire new pages, are called Single-Page Applications (SPAs). Reasoning behind the transition lies in simplified development and maintenance of the user interaction, front-end, code that otherwise easily becomes unnecessarily complex.

3.5 Frameworks and libraries

Web pages can be stripped down into two fundamental components, front-end and back-end, referring to the client-server model. Anyhow, the difference between these two is not always clear. Front-end, the client side, is ordinarily considered as the HTML, CSS and JavaScript controlling the page. Whereas back-end, the server side, is where an underlying database and any related logic lives. A Python web framework Django and a JavaScript library React are presented in the following paragraphs.

3.5.1 Django

Official home of the Django project describes it in the following manner:

> "Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel. It's free and open source."

At the time when Django was released in 2005, frameworks such as React did not exist and the concept of front-end was more imprecise than it is today. Hence, Django uses a basic templating language known as Django Template Language (DTL) for rendering content, which has been kept immutable despite the incoming pressure from rapidly evolving front-end technologies. An article written by Will Vincent argues that instead of single-handedly being a back-end framework, Django is a full-stack framework, because it can be solely used to build powerful websites. Even so when full stack is often considered as an internal API that interacts with a separate front-end JavaScript framework.

Even when Django has remained true to its origins as a Model-View-Controller (MVC) designed to operate with relational databases, many third-party packages have been created to keep Django up to date. These packages support technologies such as non-relational databases (NoSQL), real-time internet communication and modern JavaScript practices. The MVC, or more precisely its variation called Model Template View (MTV), means that the development process begins from writing a model, then a view which connects to a path, and then a template for presenting data (Galvis 2018). Django models are the tools used to work with data and databases. Then the view can be considered as a way of presenting the model in a certain format. It retrieves data from a database via the model, formats it and bundles it up in an HTTP response. Finally, the data is presented in the template. The figure below (Figure 10) illustrates a simplified version of the Django MTV model. (George 2020).

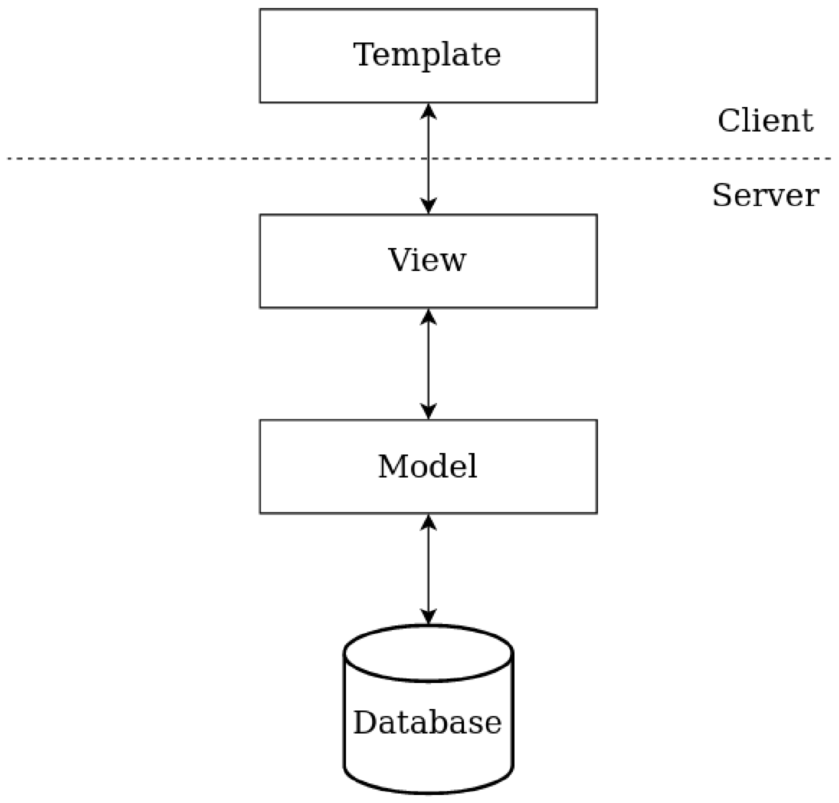


Figure 10. Simple interpretation of the Django MTV model.

While Django alone can be used to build a REST API, an extension called Django REST Framework (DRF) has been created to make the APIs more powerful and flexible. In general, REST API is a popular architectural style of providing interoperability on the internet. Basically, the REST API can be considered as the interface that allows the browser, or the frontend created with React in this case, to exchange information with the backend. Hence, DRF can be chosen over the traditional Django approach in case one only wants to create APIs instead of a complete web application. (Django REST Framework 2020)

Django's automatic admin site might lead to an assumption that Django is a Content-Management-System (CMS). Anyhow, this is false as Django is not a turnkey product, meaning a fully complete and ready to operate framework like Drupal. Instead, it is a programming tool for building web pages and the admin site is just a single module of the framework. Although Django has accommodation towards building CMS applications, that does not mean it is any less suitable for building applications without any resemblance to CMS aspects.

3.5.2 React

React is a library instead of a framework. Why is that? The key difference between these two is the inversion of control. While a library is a collection of class definitions, from which the required asset, a class or a function, can be accessed when necessary. A framework is regularly more complex: there are predefined actions to be taken, a schema according to which the code needs to be written.

React is designed for building User Interfaces (UIs). It couples the rendering logic with other UI logic such as event handling, state management and preparing data for display. Instead of putting markup language and logic in separate files, React has units called components that contain both. This is achieved with the use of JSX, a special syntax extension to JavaScript that produces React elements. Syntax wise JSX is close to HTML, but it has all the JavaScript features. A JavaScript expression can be embedded in JSX by wrapping it in single curly braces inside the JSX. After compilation, JSX expressions are evaluated to regular JavaScript objects. The compilation is done by Babel which is a FOSS JavaScript transcompiler. React Document Object Model (DOM) escapes any values embedded in JSX and converts them to a string before rendering. Therefore, injection attacks such as Cross-Site Scripting (XSS) are prevented and it is safe to embed user input in JSX. (React: Introducing JSX 2020).

Elements are the smallest building blocks of React applications and they describe what is seen on the screen. Unlike browser DOM elements, React elements are plain objects and inexpensive to create. As previously mentioned, the React DOM takes care of updating the browser DOM and matching the React elements. This is done by creating a root HTML element where the React DOM inserts elements with the help of some DOM-specific methods. The elements are immutable, which means they cannot be changed. The only way to update the UI is to create a new element. The React DOM does a lot of optimizations in the process and only updates necessary elements. (React: Rendering Elements 2020).

The components, containing markup language and logic, are conceptually like JavaScript functions. Instead of parameters, components accept arbitrary inputs called props and return elements. Besides functions, JavaScript classes introduced in ECMAScript 6 can be used to define a component with some additional features. A component is rendered by using an element to represent it, and a single component may refer to multiple other components in its output. SPAs typically have a single component at the top of the hierarchy. Splitting components into smaller components is considered as a good practice, to a certain extent of course. This extraction, combined with descriptive naming convention, increases the reusability of components in larger apps. (React: Components and Props 2020).

4 Executing the Integration

Complexity of the integration is kept as reduced as possible and only the necessary user attributes email, first name and last name are claimed from the IdP. This means that the user roles and groups on the IdP side, which could be utilized to build complex logic to the user management of the service, are completely ignored. The restriction is due to a limited development budget and the customer’s lack of suitable groups in their Azure AD tenant. Also, the integration is more likely to remain compatible with a variety of IdPs when it is kept simple. Therefore, TraSim becomes further productized as the integration can be completed also for other customers with a feasible amount of work. The following sections examine the whole integration process.

4.1 Establishing integration requirements

The requirements listed in the integration plan are separated into general and application specific categories and presented in the table below (Table 2). The firstly mentioned includes general level requirements including technical and security related topics such as using a protocol compatible with Azure AD, implementing the protocol in a secure manner, using SHA-2 algorithm with a minimum digest size of 256 bits for digital signatures, making the integration widely configurable and supporting Linux, Python 3.6+ and Django 2.2+. The algorithm and digest size limitations are there to fill the minimum requirements for protection level IV (Viestintävirasto 2018). The application specific requirements include connecting OIDC user identities with Django user identities, creating new user accounts, claiming user attributes from the IdP, populating or updating the Django user instance on authentication, allowing the use of multiple authentication sources and setting the passwords of SSO users as unusable and unchangeable.

Table 2. The list of integration requirements.

Type	Requirement
General	Integrate using Azure AD compatible protocol
	Implement the protocol securely
	Use at minimum SHA-256 for digital signatures
	Make the integration widely configurable
	Support Linux, Python 3.6+ and Django 2.2+
Specific	Connect OIDC user identities with Django user identities
	If a user signs in and has no user account, create a new one
	Claim email, first name and last name of a user from IdP
	Populate or update the user instance with claims during authentication
	Allow the existence of multiple authentication sources in parallel
	Make the passwords of OIDC users unusable and unchangeable

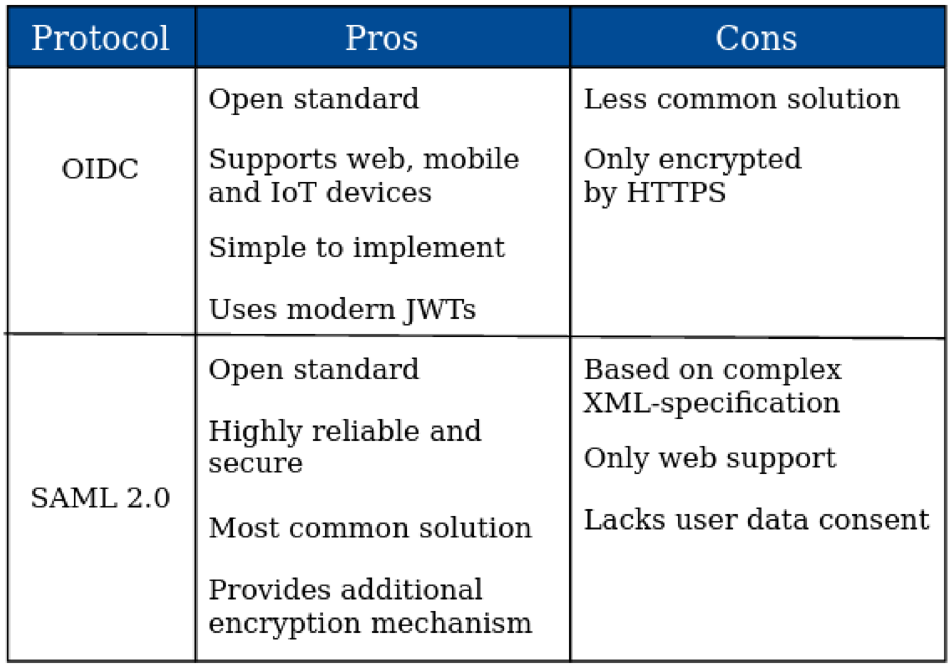
4.2 Drawing up a plan

The requirements bring in a multitude of factors to consider while creating an implementation plan. Hence, a top-down approach is taken to first fulfill the major, more general requirements and to reduce the amount of options. All requirements are evidently met by gradually following the implementation plan.

4.2.1 Choosing the protocol

Azure AD supports multiple protocols for authentication and authorization, such as SAML 2.0, OIDC and OAuth 2.0. While the latter is more of an authorization framework, SAML 2.0 and OIDC, the industry standards for federated authentication, are left to choose from. Both options have their pros and cons, which are more demonstratively presented in a table format (Table 3). Briefly put, SAML 2.0 is most common out of available solutions and highly reliable when implemented correctly. Anyhow, it is arguably complex and extremely verbose specification to implement, which can lead to long development cycles and compromised security. OIDC relies on HTTPS for encryption and trust and uses the JWT standard to store and verify identity data instead of the complex XML-based schema. (Vaughn 2019).

Table 3. Pros and cons of SAML 2.0 and OIDC protocols. (Vaughn 2019).



Protocol	Pros	Cons
OIDC	Open standard Supports web, mobile and IoT devices Simple to implement Uses modern JWTs	Less common solution Only encrypted by HTTPS
SAML 2.0	Open standard Highly reliable and secure Most common solution Provides additional encryption mechanism	Based on complex XML-specification Only web support Lacks user data consent

From these two protocols, OIDC is chosen mainly due to its simple but secure nature, use of modern JWTs and Microsoft’s instructions to apply it for modern applications, especially when building a completely new one (Microsoft: Manage Applications 2020). SAML 2.0 got multiple pros and it is firmly established as a standard within many companies. Anyhow, the increased technical complexity and the lack of flexibility for addressing modern topologies made SAML 2.0 the less desirable choice. Also, its own richness translates into expensive requirements in terms of cryptography and bandwidth that are not proportionate to the actual needs of modern applications (Bertocci 2016).

4.2.2 Selecting a library

After selecting the protocol, it must be implemented in the application. As it is not feasible nor secure to implement OIDC support from scratch, a community developed PyPI library needs to be utilized. There are certain technical requirements from the library, such as it must support Linux OS, Python version 3.6 and Django version 2.2. In other words, the library needs to be designed for a Django project. Other requirements include clear documentation, wide configurability, security, mature development status, active community, open source licensing agreement and support for multiple authentication backends to work in parallel.

Two suitable candidates exist, and they are briefly compared in the table below (Table 4). The libraries are very similar: both support the OIDC authorization code flow in a lightweight manner and provide all the endpoints, data and logic needed to add OIDC capabilities to a Django project. *Mozilla-django-oidc* is backed by Mozilla, the organization behind the Firefox browser and multiple other projects. The other candidate, *django-oidc-provider*, is originally designed by a private individual Juan Fiorentino. All the information in the table is collected from official GitHub repositories where the projects are maintained.

The decision between the libraries is not simple. In the end, it heavily depends on personal preference. Fiorentino's library is more popular and widely used in terms of GitHub stars and forks. It was also established in 2015, two years prior to the library by Mozilla. Anyhow, both libraries are in mature production development status and fulfil the technical requirements. The facts encouraging the use of Mozilla's library are great documentation, configurability and active community pushing additional features and bug fixes. The library has also been end-to-end tested and audited by the Mozilla InfoSec team. Based on this information, *mozilla-django-oidc* is the library of choice.

Table 4. Comparison of Django compatible libraries implementing OIDC. (GitHub 2020).

Library	django-oidc-provider	mozilla-django-oidc
GitHub	https://github.com/juanifioren/django-oidc-provider	https://github.com/mozilla/mozilla-django-oidc
Stars	277	130
Forks	166	77
License	MIT	MPL-2.0
Development Status	Production/Stable	Production/Stable
Last Updated	October 17, 2018	January 10, 2020
Python 3.6+	Yes	Yes
Django 2.2+	Yes	Yes
Security Audit Conducted	Unknown	Yes
Documentation	Good	Great
Configurability	Good	Great
Supports Multiple Authentication Backends	Yes	Yes

4.2.3 Building test scenarios

Testing is required to validate whether the actual OIDC SSO establishment functions as expected, is defect free and matches the original requirements. This is done by first creating an Azure AD tenant and integrating it with a testing environment. Then the functionality of the integration is verified with certain test cases.

The test scenarios are presented in the following table (Table 5). They are designed to test the application, service provider, side. Azure AD is trusted to handle the user authorization, password restrictions, MFA and other similar concerns as promised on Microsoft’s documentation. The scenarios include basic functionality such as redirecting a user to the IdPs authentication page after a push of the SSO button. After successful authentication, the user is expected to be directed to the default page of the service. Then there are scenarios for the initial login process, consecutive logins, signing out, requesting a password reset link, using a regular Django user and turning the SSO implementation off via environment variables. Once all scenarios pass, the functionality of the integration can be verified to function at an appropriate level.

Table 5. Test scenarios to verify successful integration.

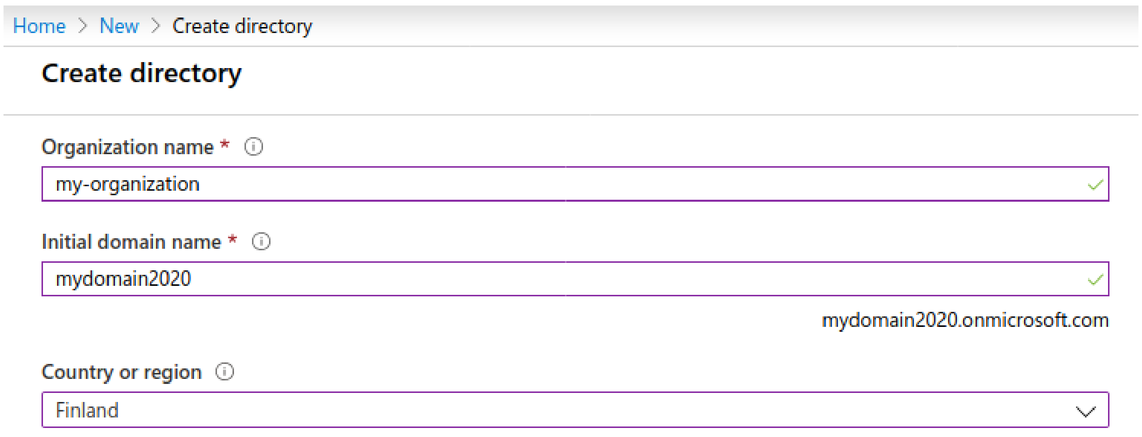
Test Scenario	Expected Result
Press SSO button	Redirect to IdP authentication page
Authenticate successfully	Redirect to default page
Check user status and attributes after initial login	New user is created with correct attributes (OIDC claims, "is_sso_user", unusable password)
View edit user page and modify information	Set password fields are removed and information can be modified without additional authentication
Recheck user status and attributes on following logins	OIDC user identity is connected to the initially created user and attributes are updated on every login
Remove user access during session	Silent session refresh that occurs every 15 minutes terminates the session
Sign out	Terminate session and redirect to login page
Request password reset link to a user with unsuable password	The request is denied by Django
Authenticate as a regular Django user	Django's core authentication system and other features work regularly
Turn the SSO implementation OFF via environment variables	The service works regularly but users can not authenticate via external IdP

4.3 Setting up Azure AD

Before the integration can take place and the SP can be configured, the IdP needs to be set up. Following chapters go through the process of creating an Azure AD tenant, managing users, registering an application and finding integration endpoints.

4.3.1 Creating tenant

New Azure AD tenant can be created by navigating to *https://portal.azure.com* and clicking *Create a resource* on the command bar. Then a marketplace page is loaded, and Azure AD can be selected by search or from the *Identity* tab. Then the dialog box presented in the figure below (Figure 11) opens and it is supplied with organization name, initial domain name and region.



Create directory

Organization name * : my-organization

Initial domain name * : mydomain2020

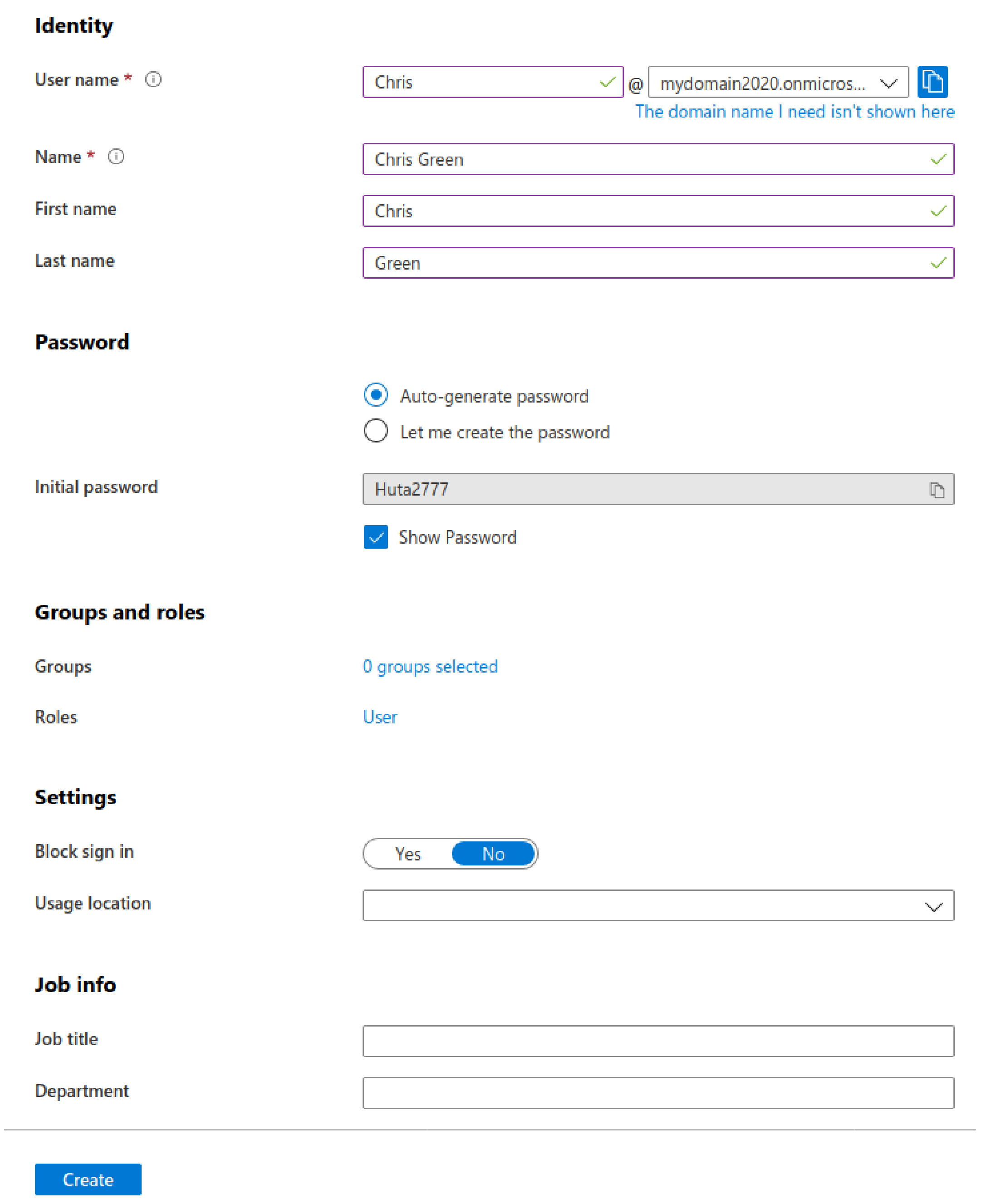
mydomain2020.onmicrosoft.com

Country or region : Finland

Figure 11. The dialog box for creating a new directory

4.3.2 Creating user

The next step is to populate the directory by provisioning users. This can be done using Azure AD management portal, Microsoft Graph API or Azure AD Connect. In this case, the management portal is utilized, and it can be accessed by clicking *Azure AD* on the command bar, selecting *Users* tab and then clicking *New user*. Now a dialog box opens for creating a new user or inviting a guest user. After filling up all required information the user can be created, and it is added to *All users* list in the *Users* tab. The *Block sign in* -field that can be seen in the figure below (Figure 12) is left disabled.



Identity

User name *	Chris @ mydomain2020.onmicros...
	The domain name I need isn't shown here
Name *	Chris Green
First name	Chris
Last name	Green

Password

[✓ Auto-generate password] Let me create the password

Initial password	Huta2777

✓ Show Password

Groups and roles

Groups	0 groups selected
Roles	User

Settings

Block sign in	Yes [✓ No]
Usage location	

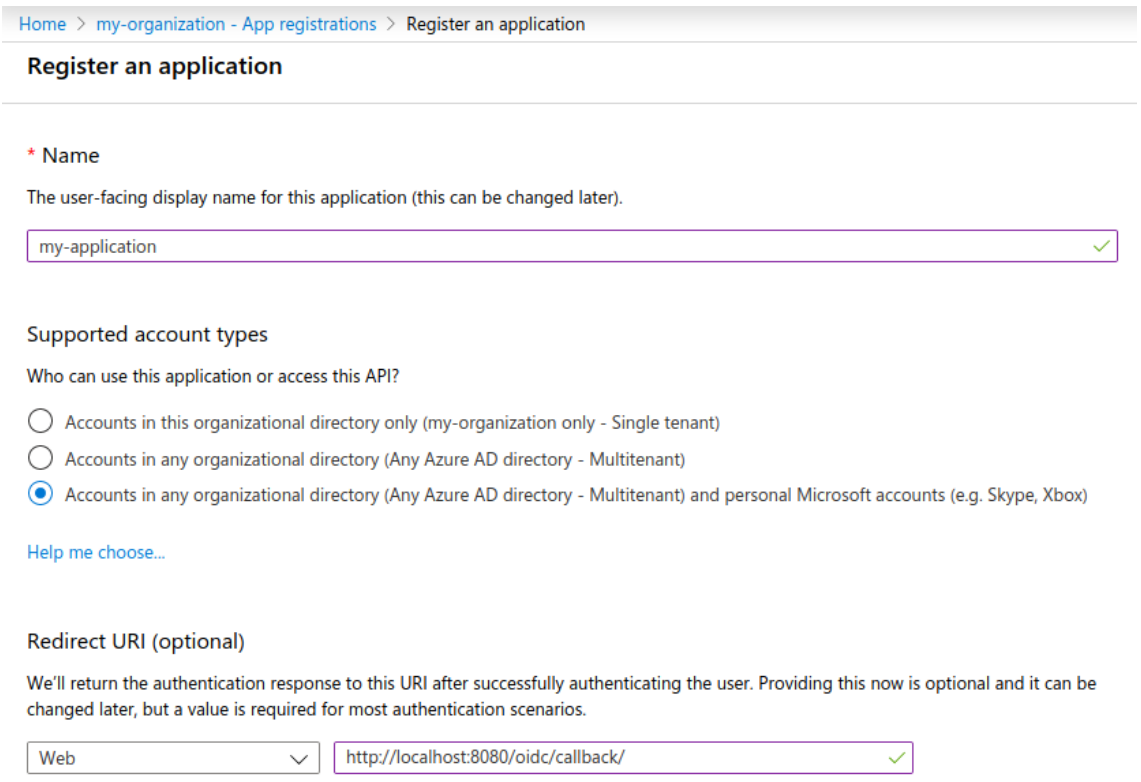
Job info

Job title	
Department	

Figure 12. The dialog box for creating a new user.

4.3.3 Registering application

The application is registered by navigating to the newly created directory, selecting *App Registrations* tab and clicking *New registration*. Then a dialog box opens for registering the application and it is provided with a name and the redirect URI. There is also a field to select what kind of user accounts can access the endpoints of the application. What should be selected depends on the requirements of the particular use case. The third option, “*Accounts in any organizational directory (Any Azure AD directory - Multi Tenant) and personal Microsoft accounts (e.g. Skype, Xbox*)”, is selected in the figure below (Figure 13) for testing purposes. The application is enabled for users to sign-in by default and user assignment is not required, which means all tenant users are granted access to the application.


Home > my-organization - App registrations > Register an application

Register an application

* Name

The user-facing display name for this application (this can be changed later).

my-application

Supported account types

Who can use this application or access this API?

- ○ Accounts in this organizational directory only (my-organization only - Single tenant)
- ○ Accounts in any organizational directory (Any Azure AD directory - Multitenant)
- [✓ Accounts in any organizational directory (Any Azure AD directory - Multitenant) and personal Microsoft accounts (e.g. Skype, Xbox)]

Help me choose...

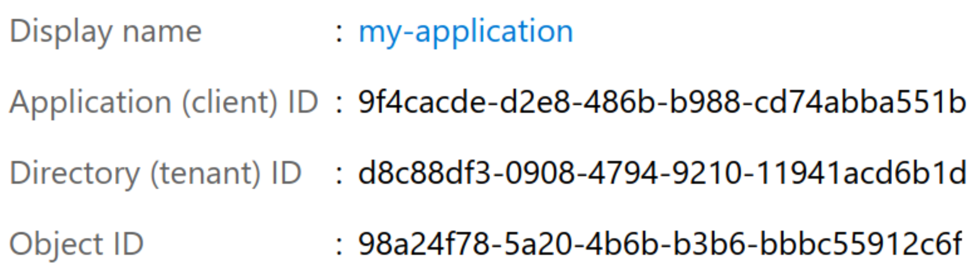
Redirect URI (optional)

We'll return the authentication response to this URI after successfully authenticating the user. Providing this now is optional and it can be changed later, but a value is required for most authentication scenarios.

Web | http://localhost:8080/oidc/callback/

Figure 13. The dialog box for registering an application.

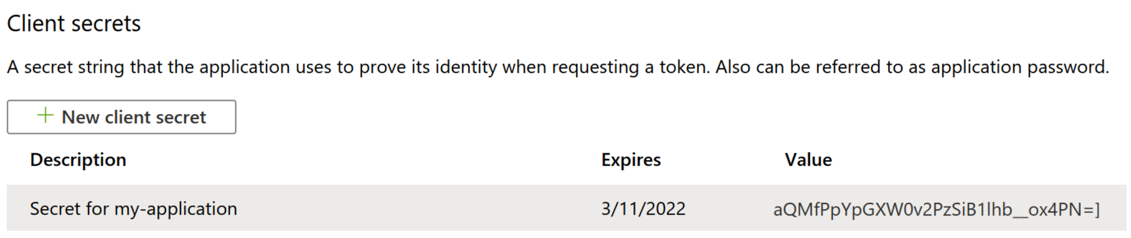
After the registration is complete, client ID and tenant ID are found from the Overview tab (Figure 14). While the tenant ID is used for identifying the directory and protocol endpoints, the client ID is a public identifier for applications. Basically, every application registered to the Azure AD tenant has its own unique identifier, so that the tenant knows which application a user is trying to access.



Display name	:	my-application
Application (client) ID	:	9f4cacde-d2e8-486b-b988-cd74abba551b
Directory (tenant) ID	:	d8c88df3-0908-4794-9210-11941acd6b1d
Object ID	:	98a24f78-5a20-4b6b-b3b6-bbbc55912c6f

Figure 14. Application information from the *Overview* tab.

The secret that is used for decrypting the encrypted hash of a signed JWT is added in the *Certificates & secrets* tab of the application page in Azure by clicking *New client secret*. It can be set to expire in a year, two years or never. The secret must be kept secure under any circumstances to maintain the confidentiality of the application. The tenant and the application, for which the secret presented in the figure below (Figure 15) is created, are only created for screening purposes and are never used even in a testing environment.



Client secrets

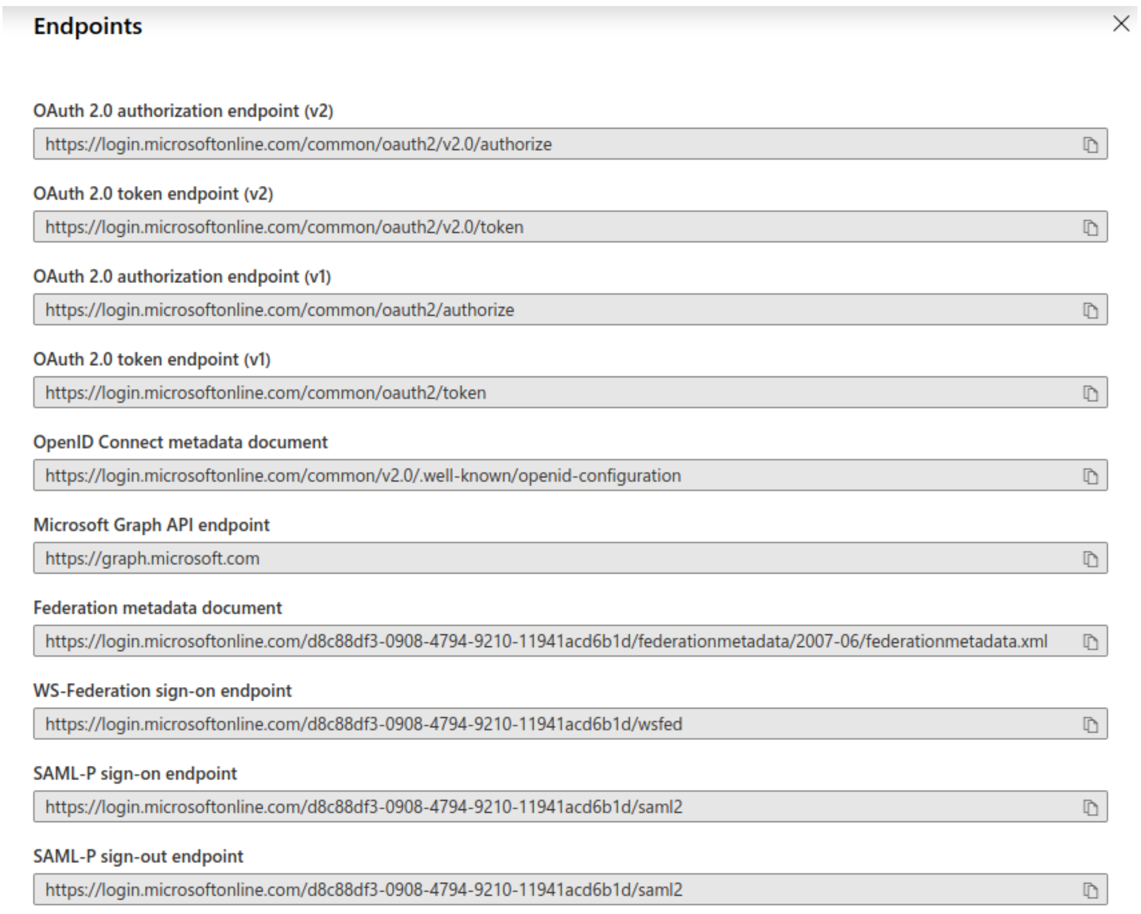
A secret string that the application uses to prove its identity when requesting a token. Also can be referred to as application password.

Description	Expires	Value
Secret for my-application	3/11/2022	aQMfPpYpGXW0v2PzSiB1lhb__ox4PN=]

Figure 15. Client secret created at the *Certificates & secrets* tab.

4.3.4 Finding integration endpoints

Azure AD exposes endpoints for WS-Federation metadata and sign-on, SAML 2.0 sign-on and sign-out, OAuth 2.0 token and authorization, OIDC and Azure AD Graph API (Tejada et al. 2015: 318). All the endpoints are presented in the figure below (Figure 16) and they can be viewed on Azure by navigating to the directory, selecting *App registrations* tab and clicking *Endpoints*. When integrating using OIDC, all required endpoints are found in the OIDC metadata document which is available at *https://login.microsoftonline.com/fee7bd5a-1343-4aa8-88ea-5414e64986d8/.well-known/openid-configuration*.


Endpoints

OAuth 2.0 authorization endpoint (v2)
https://login.microsoftonline.com/common/oauth2/v2.0/authorize

OAuth 2.0 token endpoint (v2)
https://login.microsoftonline.com/common/oauth2/v2.0/token

OAuth 2.0 authorization endpoint (v1)
https://login.microsoftonline.com/common/oauth2/authorize

OAuth 2.0 token endpoint (v1)
https://login.microsoftonline.com/common/oauth2/token

OpenID Connect metadata document
https://login.microsoftonline.com/common/v2.0/.well-known/openid-configuration

Microsoft Graph API endpoint
https://graph.microsoft.com

Federation metadata document
https://login.microsoftonline.com/d8c88df3-0908-4794-9210-11941acd6b1d/federationmetadata/2007-06/federationmetadata.xml

WS-Federation sign-on endpoint
https://login.microsoftonline.com/d8c88df3-0908-4794-9210-11941acd6b1d/wsfed

SAML-P sign-on endpoint
https://login.microsoftonline.com/d8c88df3-0908-4794-9210-11941acd6b1d/saml2

SAML-P sign-out endpoint
https://login.microsoftonline.com/d8c88df3-0908-4794-9210-11941acd6b1d/saml2

Figure 16. The list of all protocol endpoints provided by Azure AD.

The OIDC library requires JWKs, token, authorization and user endpoints. When authenticating a user, the request containing client ID is first sent to the authorization endpoint of the tenant. Then the user is presented a login page, and after successful authentication, a response containing JWT is returned to the redirect URI set in the registration process. The authenticity of the response is verified using the client secret to make sure it is valid and signed by a trusted issuer. In production environments the traffic to domains such as *login.microsoftonline.com* is often blocked by a firewall, so the SSO implementation will not work until settings are modified to enable the traffic.

The IdP side is now set up and accessible for all users registered to the tenant. Next, TraSim needs code to satisfy the OIDC authentication scenario including the client ID, client secret and the endpoints listed below:

```
“jwks_uri”:
"https://login.microsoftonline.com/d8c88df3-0908-4794-9210-
11941acd6b1d/discovery/keys",

“token_endpoint”:
"https://login.microsoftonline.com/d8c88df3-0908-4794-9210-
11941acd6b1d/oauth2/token",

“authorization_endpoint”:
"https://login.microsoftonline.com/d8c88df3-0908-4794-9210-
11941acd6b1d/oauth2/v2.0/authorize",

“user_endpoint”:
”https://login.microsoftonline.com/d8c88df3-0908-4794-9210-
11941acd6b1d/openid/userinfo”
```

4.4 Implementing the plan

This chapter walks through how the library that implements the OIDC protocol is installed and configured.

4.4.1 Library installation

The *mozilla-django-oidc* library is first added to special type of a file called *requirements.tx*t. This file allows the installation of multiple PyPI libraries with a single command, that is completed in the project’s Dockerfile. Basically, for *requirements.tx*t to work, it only requires the name and version of all the libraries one wants to install. This is done in following manner:

```
Example-library==1.0.0
…
mozilla-django-oidc==1.2.3
…
Example-library-N==1.0.0
```

Anyhow, before the library can be installed, a package called *libffi-dev* needs to be installed to the system running the application. This package provides a portable, high level programming interface to provide a bridge from the interpreter program to compiled code (Sourceware 2020). While the system in use is a minimal Docker image based on Alpine Linux with Python installed, *libffi-dev* and multiple other packages that remain undefined due to security concerns are installed using Alpine Linux package management tool apk. These packages are installed in the same Dockerfile as the PyPI libraries listed in the *requirements.txt* file.

```
FROM python:3.6.10-alpine

RUN apk add --no-cache \
     Example-package \
     …
     libffi-dev \
     …
     Example-package-N

RUN python3 -m pip install -r requirements.txt --no-cache-
dir
…
```

4.4.2 Configuring settings

The OIDC library expects that all OIDC related settings are configured in a single file, *settings.py*, which is a common approach for Django projects. As the settings file has a tendency of becoming unnecessarily large and complex, the OIDC specific settings and additional configurations are defined in separate files, *auth.py* and *oidc.py*, which are then imported to the settings file. Anyhow, adding environment specific information to the code causes trouble when reconfiguring the project for different environments or protecting sensitive data, such as secret keys, from being stored to a VCS. The struggle is further increased in containerized Docker environments. Environment variables are introduced to battle these issues. They are defined in a specific *.env* file and accessed in the code with the help of a useful library called *django-environ*. They enable the configuration of OIDC settings, and Django settings in general, without hardcoding them. The whole design is presented in the figure below (Figure 17).

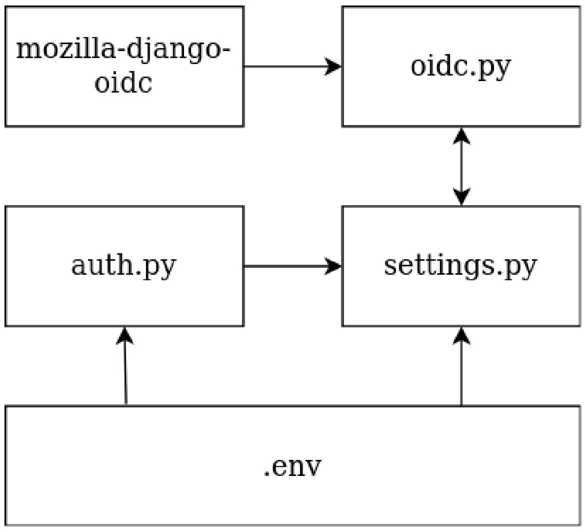


Figure 17. Architecture of OIDC settings files.

The *.env* file includes OIDC configuration information presented in the code block below. Variable *ENABLE_OIDC* tells Django that the current environment has enabled OIDC authentication, or SSO, and certain actions should be performed. *OIDC_USERNAME_CLAIM* defines which claim should be mapped as the username of the user about to authenticate. These two are custom variables that the library does not require or support by default. The rest are standard variables and include information such as the client ID, client secret, signing algorithm, JWKs endpoint, authorization endpoint, token endpoint and user endpoint.

For the JWT signing algorithm there are two possibilities, HS256 and RS256. The latter points to asymmetric key algorithm that uses a combination of Hash-based Message Authentication Code (HMAC) and SHA-256 to generate and validate a signature. HS256 in turn points to symmetric key algorithm with only one secret key shared between the parties. Therefore, the same key is used both to generate a signature and to validate it. RS256 was eventually chosen because Azure AD provides an endpoint for fetching the public key of the service by default and the consumer does not need to know the secret key. The rest of the standard values are provided after creating an Azure AD tenant and registering an application. This process is thoroughly described in the testing chapter.

```
ENABLE_OIDC=True
OIDC_USERNAME_CLAIM=email
OIDC_RP_CLIENT_ID=9f4cacde-d2e8-486b-b988-cd74abba551b
OIDC_RP_CLIENT_SECRET=aQMfPpYpGXW0v2PzSiB1lhb__ox4PN=]
OIDC_RP_SIGN_ALGO=RS256
OIDC_OP_JWKS_ENDPOINT=https://login.microsof-
tonline.com/d8c88df3-0908-4794-9210-11941acd6b1d/discov-
ery/v2.0/keys
OIDC_OP_AUTHORIZATION_ENDPOINT=https://login.microsof-
tonline.com/d8c88df3-0908-4794-9210-
11941acd6b1d/oauth2/v2.0/authorize
OIDC_OP_TOKEN_ENDPOINT=https://login.microsof-
tonline.com/d8c88df3-0908-4794-9210-11941acd6b1d/oauth2/to-
ken
OIDC_OP_USER_ENDPOINT=https://login.microsof-
tonline.com/d8c88df3-0908-4794-9210-
11941acd6b1d/openid/userinfo
```

The environment variables are then accessed in the *auth.py* file. They are casted to a specific variable type and given a default value, which is applied in case the corresponding environment variable is not found. *The ENABLE_OIDC* variable plays a major role while it is used as a condition in *settings.py* file to enable or disable the SSO implementation.

```
import environ

env = environ.Env()

# SSO setup (OpenID Connect)
ENABLE_OIDC=env('ENABLE_OIDC', cast=bool, default=False)
OIDC_USERNAME_CLAIM=env('OIDC_USERNAME_CLAIM', cast=str, de-
fault='email')
OIDC_RP_SIGN_ALGO=env('OIDC_RP_SIGN_ALGO',   cast=str,   de-
fault=None)
OIDC_RP_CLIENT_ID=env('OIDC_RP_CLIENT_ID',   cast=str,   de-
fault=None)
```

```
OIDC_RP_CLIENT_SECRET=env('OIDC_RP_CLIENT_SECRET', cast=str,
default=None)
OIDC_OP_JWKS_ENDPOINT=env('OIDC_OP_JWKS_ENDPOINT', cast=str,
default=None)
OIDC_OP_AUTHORIZATION_ENDPOINT=env('OIDC_OP_AUTHORIZA-
TION_ENDPOINT', cast=str, default=None)
OIDC_OP_TOKEN_ENDPOINT=env('OIDC_OP_TOKEN_ENDPOINT',
cast=str, default=None)
OIDC_OP_USER_ENDPOINT=env('OIDC_OP_USER_ENDPOINT', cast=str,
default=None)
```

After defining the variables in *auth.py* they are imported to *settings.py*, where the OIDC library accesses them by default. Besides importing the variables, changes presented in the following code block need to be made. The changes include conditionally appending a silent session refresh to middleware, *mozilla-django-oidc* to installed applications and inserting an overridden version of the OIDC library's *OIDCAuthenticationBackend* class to authentication backends and allowing the access of URLs starting with *"/oidc"* without authentication.

Middleware is a framework that hooks components into Django's request and response processing. The installed applications setting is used to wire the combination of models, views, templates and middleware of an application, or a library, to a project. Authentication backends setting provides an extensible system when a user needs to be authenticated against a different service than Django's default (Django: Authentication 2020). The if condition is added as a security measure: some installations do not use SSO and therefore these settings are unnecessary and should not be appended.

```
from .auth import *

…

if ENABLE_OIDC:
     MIDDLEWARE += [
          'mozilla_django_oidc.middleware.SessionRefresh',
     ]
     INSTALLED_APPS += [
          'mozilla_django_oidc',
     ]
     AUTHENTICATION_BACKENDS.insert(1,'core.oidc.CustomOI-
     DCAuthenticationBackend')
```

```
          LOGIN_EXEMPT_URLS += [
               r'^oidc'
          ]

     …
```

Finally routing options of the OIDC library are added to *urls.py* in a similar manner:

```
     …

     if settings.ENABLE_OIDC:
          urlpatterns += [
              url(r'^oidc/', include('mozilla_django_oidc.urls'))
          ]

     …
```

4.4.3 Additional configuration

The additional configurations modify the way Django users are created or updated and how OIDC user identities are connected to Django users. While users log into the application by authenticating with the IdP, the default behavior of the OIDC library is to look up a Django user matching the email field to the email address in the user attributes claimed from the IdP (GitHub: mozilla-django-oidc 2020). This method of connecting the users is made configurable by importing the *OIDC_USERNAME_CLAIM* attribute from the *settings.py* file and a class called *OIDCAuthenticationBackend* from the OIDC library. Then the *filter_users_by_claims()* method is overridden by subclassing the previously imported class.

Methods *create_user()* and *update_user()* are overridden accordingly. The logic is to map the claimed user attributes, which are first name, last name name and the configurable attribute to corresponding Django user fields. The configurable attribute, most commonly email, is mapped to the username of a Django user. In addition to mapping the claims, the password of the Django user is as unusable and an additional user profile field called *is_sso_user*, which is of type Boolean and false by default, is set to true. This

Boolean attribute is used for controlling the fields of update profile forms. More specifically, the ability to set or reset the password of an *"SSO user"* is disabled. The methods for creating and updating a user are very similar, firstly mentioned creates a new user if a matching username is not found, and the latter updates the user attributes on every login.

```
from  mozilla_django_oidc.auth  import  OIDCAuthentication-
Backend
from django.conf import settings

class   CustomOIDCAuthenticationBackend(OIDCAuthentication-
Backend):
    def filter_users_by_claims(self, claims):
        username = claims.get(settings.OIDC_USERNAME_CLAIM)
        if not username:
            return self.UserModel.objects.none()

        try:
            return              self.UserModel.objects.fil-
ter(username__iexact=username)

        except:
           return self.UserModel.objects.none()

    def create_user(self, claims):
        user    =    super(CustomOIDCAuthenticationBackend,
self).create_user(claims)
        user.username           =           claims.get(set-
tings.OIDC_USERNAME_CLAIM, '')
        user.email = claims.get('email', '')
        user.first_name = claims.get('given_name', '')
        user.last_name = claims.get('family_name', '')
        user.set_unusable_password()
        user.save()

        user.profile.is_sso_user = True
        user.profile.save()

        return user

    def update_user(self, user, claims):
        user.username           =           claims.get(set-
tings.OIDC_USERNAME_CLAIM, '')
        user.email = claims.get('email', '')
        user.first_name = claims.get('given_name', '')
        user.last_name = claims.get('family_name', '')
        user.set_unusable_password()
        user.save()
        user.profile.is_sso_user = True
        user.profile.save()

        return user
```

4.4.4 Login template

Login functionality is enabled by adding a link that initializes the OIDC authentication process to the login template. The link is assigned classes to achieve button style and the previously introduced logic is followed as the button is only displayed when integration is enabled. The code is presented below:

```
{% block login_form %}
  <div class=”row”>

    …

    {% get_setting_by_key "ENABLE_OIDC" as enable_oidc %}
    {% if enable_oidc %}
      <br />
      <div class="text-center">
        <a  href="{%  url  'oidc_authentication_init'  %}"
class="btn btn-primary btn-block">
          {% trans 'Single sign-on' %}
        </a>
      </div>
    {% endif %}

   …

  </div>
{% endblock login_form %}
```

The code results in added Single Sign-On button in the login form which is presented in the following figure (Figure 18).

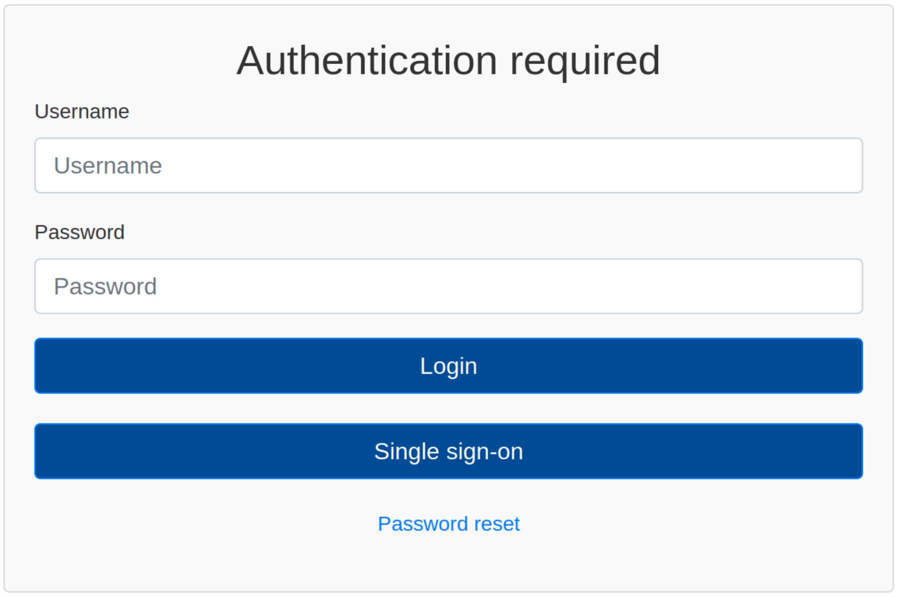


Figure 18. Login screen after modification.

After clicking the Single sign-on button, the user is redirected to authenticate with Azure AD (Figure 19). Then the user is redirected to the service.

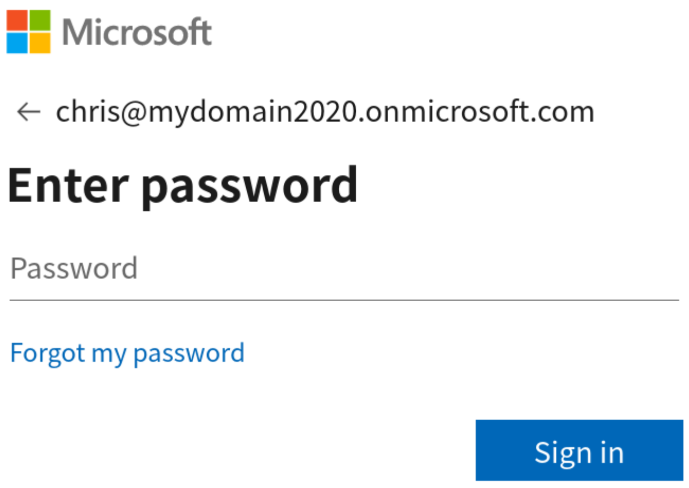


Figure 19. Microsoft authentication dialog box.

4.4.5 User Management

Some of the logic considering user management needs to be modified when applying the SSO implementation. Adding an additional Boolean field to the Django user object is an effective way to accomplish this. Django provides built-in support for database migrations, which makes it easy to rework a model, as migrations can be quickly generated to create necessary tables in the database. In practice this is achieved by first adding the name, type and default value of the new field:

```
class PersonProfile(models.Model):
   …
   is_sso_user      =      models.BooleanField(default=False,
blank=True)
   …
```

Then the migration is created and applied to make changes in the database with following commands:

```
python manage.py makemigrations
Python manage.py migrate
```

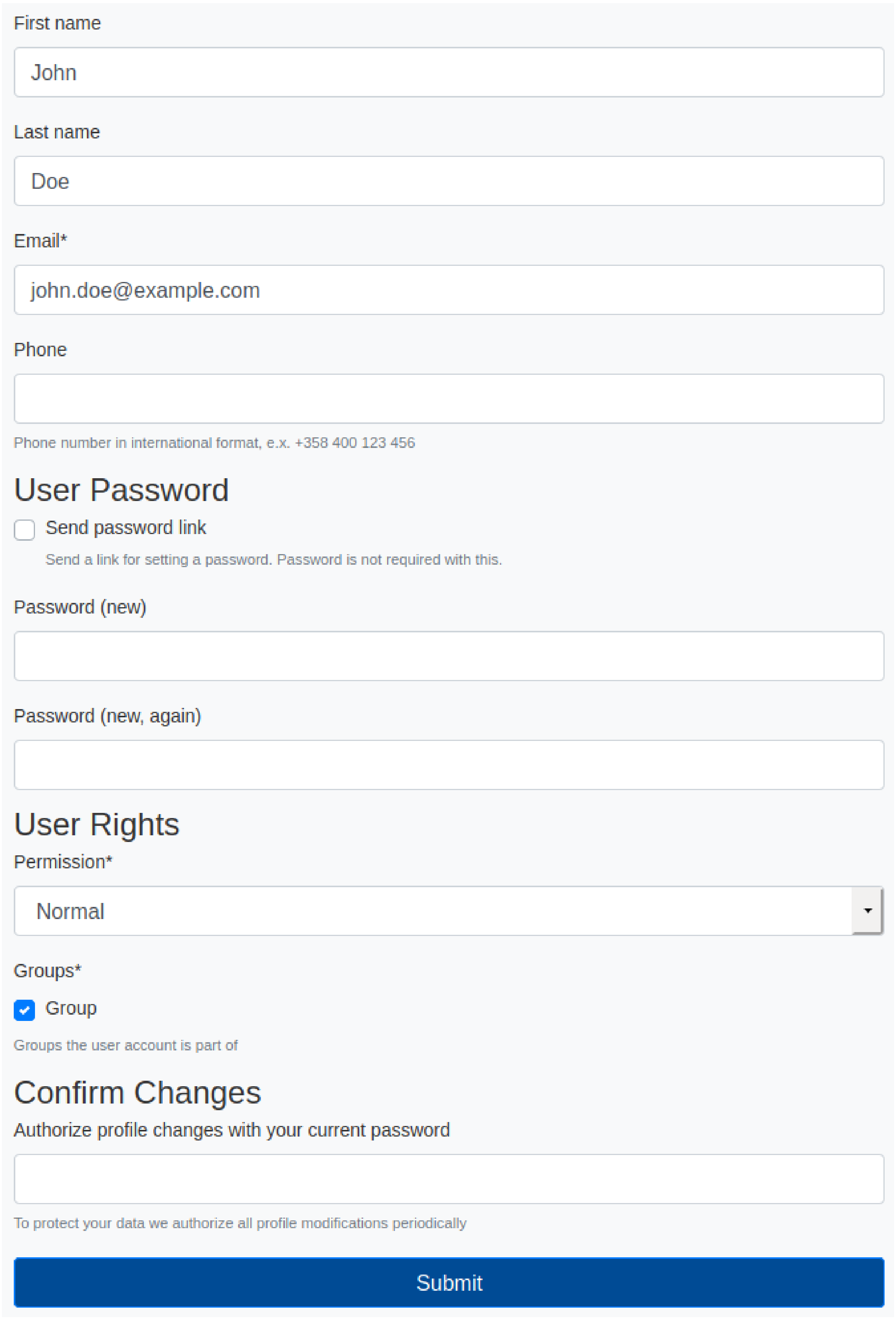
TraSim's built-in user management requires some fine tuning after implementing the integration. As the passwords of *SSO users* are set as unusable, it must be made sure that they remain in the state and cannot be modified. Therefore, *password* and *confirm_password* fields of class *BaseProfileForm*, which extends as a base form for all profile related actions, are removed from all *SSO users*.

```
# Base form for all person forms
class BaseProfileForm(ModelForm):

…

def __init__(self, *args, **kwargs):
       super(BaseProfileForm, self).__init__(*args, **kwargs)
       try:
           user = self.instance.user
           self.fields['email'].initial = user.email
           self.fields['first_name'].initial            =
user.first_name
```

```
                self.fields['last_name'].initial = user.last_name
                if user.profile.is_sso_user:
                    self.fields.pop('password')
                    self.fields.pop('confirm_password')
            except ObjectDoesNotExist:
                pass
```

There remains a field called *authenticate* requiring the password of the user who is modifying a profile to authenticate the changes. Hence *SSO users* do not have a password, they normally cannot modify their own profile nor someone else's. This is corrected by adding a logical OR-operator to the if-statements of the user profile editing logic, that checks the authenticity of the entered password. Now the authorization field is still displayed for everyone, but *SSO users* may leave the field empty whereas regular users are required to enter their password. The regular and updated edit user forms are presented in the figures below (Figure 20 & Figure 21).

```
def edit(request, pk):
     …
     if request.method == 'POST':
          …
          if form.is_valid():
               if check_authorization(request, request.user,
form) or request.user.profile.is_sso_user:
               # Updating the user and user profile (email,
               first name, last name, phone)
          else:
               # Raise form validation error
     else:
          …
          if  not  request.user.profile.is_password_author-
ized() and not request.user.profile.is_sso_user:
               form.fields['authorization'].required = True
```



First name

John

Last name

Doe

Email*

john.doe@example.com

Phone

Phone number in international format, e.x. +358 400 123 456

User Password

☐ Send password link

Send a link for setting a password. Password is not required with this.

Password (new)

Password (new, again)

User Rights

Permission*

Normal

Groups*

☑ Group

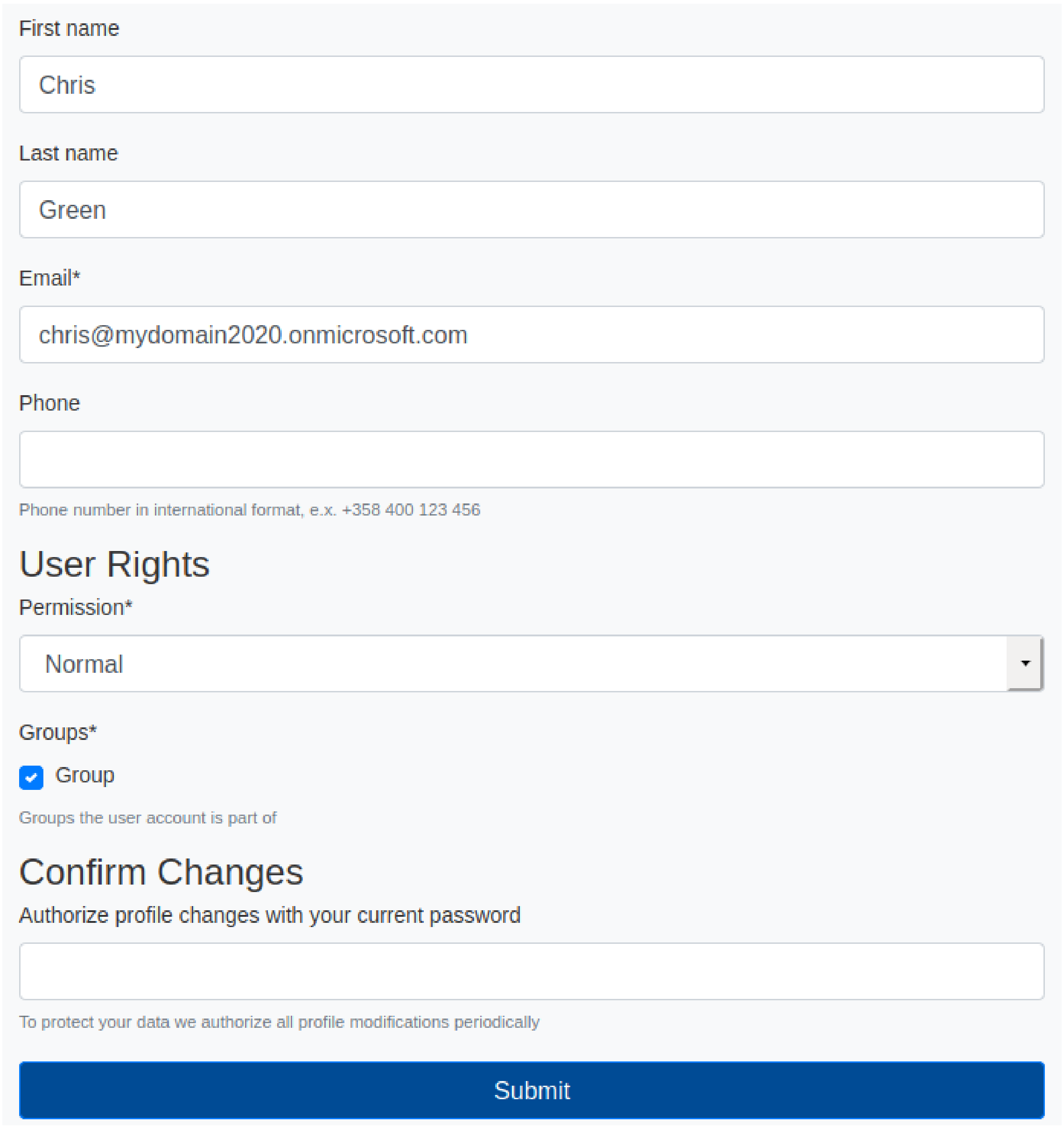
Groups the user account is part of

Confirm Changes

Authorize profile changes with your current password

To protect your data we authorize all profile modifications periodically

Figure 20. Edit user form of a regular user.



First name

Chris

Last name

Green

Email*

chris@mydomain2020.onmicrosoft.com

Phone

Phone number in international format, e.x. +358 400 123 456

User Rights

Permission*

Normal

Groups*

☑ Group

Groups the user account is part of

Confirm Changes

Authorize profile changes with your current password

To protect your data we authorize all profile modifications periodically

Figure 21. Edit user form of an SSO user with reduced number of fields.

4.5 Executing test scenarios

The test scenarios with actual results and pass or fail statuses are presented in the table below (Table 6). The only test scenario with failed status is following: *Remove user access during session*. The main functionality works, as the user session is terminated after the silent refresh if the user has been suddenly unauthorized. Anyhow, the functionality has an unwanted side effect.

After establishing a session and leaving it open for a while, the silent re-auths of the session refresh middleware exceeded the maximum amount of entries Azure AD allows. This security feature caused all incoming messages from the corresponding IP address to become automatically blocked for a few hours. A possible workaround is to increase the length of the interval between re-auths, but it is not guaranteed to solve the problem with full certainty. Therefore, some built-in session expiry mechanisms of Django are trusted to prevent long sessions from happening.

Table 6. Actual results and pass or fail statuses of the test scenarios.

Test Scenario	Actual Result	Pass/Fail
Press SSO button	Redirection was successful	✓
Authenticate successfully	Redirection was successful	✓
Check user status and attributes after initial login	New user was created with correct user information	✓
View edit user page and modify information	The user was able to modify personal information and password fields were removed	✓
Recheck user status and attributes on following logins	The identities were connected and user information was updated	✓
Remove user access during session	Maximum amount of allowed connection retries of the Azure AD tenant was exceeded and the connection became blocked	✗
Sign out	Redirection and session termination were successful	✓
Request password reset link to a user with unsuable password	Request was denied as expected	✓
Authenticate as a regular Django user	Core authentication system and other features worked regularly	✓
Turn the SSO implementation OFF via environment variables	Django users worked regularly and OIDC users became unusable as expected	✓

5 Results

The main research question of this thesis is how to implement Azure AD integration with an existing cloud service to enable SSO and user provisioning. Being able to answer this question requires a basic knowledge about many concepts such as IAM, web technologies, SSO, cloud computing and AD. Then, to be able to proceed with the integration after embracing the basic knowledge, one needs to distinguish the technologies of the service they are developing that, together with the properties of Azure AD, adds boundaries on how the integration can be completed. Another important factor is the complexity of the SSO solution, the range of information that is claimed from the IdP. This widely affects the user management processes of the service.

After obtaining all the information, a set of requirements was created to actualize the boundaries of an operative solution and to assist in the decision-making process. The requirements are presented in Table 2 and include general requirements considering the technologies and security of the service and more exact application specific requirements. Then the requirements were used to choose a protocol for enabling the Azure AD integration. OIDC was the protocol of choice due to its simple and modern, but secure nature. Next step was to plan the implementation of the protocol. While it was considered infeasible and unsustainable to securely implement the protocol from ground up exclusively for this purpose, a community developed PyPI library, *mozilla-django-oidc*, was utilized. This library was selected, because it was one of the few that supported the requirements of the service, and it was widely used, configurable, actively developed and provenly secure. Finally, the planning phase is ended with building the test scenarios, which are presented in Table 5. The scenarios are combinations of actions, for example checking the values of certain user attributes after the authentication process initialized by a click of a button. All the scenarios are initially based on the requirements and intent to verify whether the integration behaves as expected.

Before the library could be installed and configured to work with the SP, the IdP had to be set up. The processes of creating an Azure AD tenant, managing users, registering an application and finding the integration endpoints for OIDC were thoroughly explained using figures. Then, after obtaining the necessary information, the implementation phase could take place. It started with installing *the mozilla-django-oidc* library and defining the required packages in the Dockerfile of the image used by the SP. Environment variables were introduced to separate the settings that are used to configure the library from VCS and to make them easier to change or completely disable. The environment variables are accessed and managed in a slightly complex manner between the Python modules which is presented in Figure 17. The values of the variables were obtained while setting up the IdP and they include information such as the client ID, client secret and JWKs, authorization, token and user endpoints.

Setting up the library was followed by additional configuration of how the OIDC user identities are connected to the Django user identities and how users are created or updated after authentication. Basically, this phase included making the decision which one of the claimed user attributes should be mapped to the unique username field of the Django user. The email field was chosen, because it is quite standard approach and the field is unique also on the Azure AD side. Also, the password of the Django users that are connected to OIDC user identities, were configured to be set as unusable to prevent them from using Django's core authentication system. Lastly, before executing the test scenarios, an SSO-button was added to the template of the login page to initialize the authentication process and an additional Boolean field was added to the Django user object. The field was added to fine tune the user management of the service by denying the SSO-users from setting a password and disabling the requirement which obligates one to verify any changes made to their personal information using their password. The original and modified versions of the user profile edition forms are presented in figures 20 & 21.

Finally, after creating the IdP and registering the application, installing and setting up the OIDC library, configuring the SP and building the Docker image, the service could be started in a local development environment and the test scenarios were executed. All but one of the scenarios passed: the failed scenario had to do with removing user access to the service during a session. The expected result was that the user session would be eventually terminated when the recurring silent re-auth happens. It did terminate the session as expected, but there were some compatibility issues with Azure AD, which caused the connection to become blocked due to the excessive amount of authentication requests. The silent re-auth middleware was eventually removed and built-in session expiry mechanisms of Django were trusted to prevent long sessions from happening.

6 Conclusions and discussion

This chapter focuses on reviewing the status of the Azure AD integration: how did the combination of technologies, protocols and technical decisions play out in the end. The review is followed by examining the areas of future studies which mainly focus on what could have been done better.

6.1 Meeting integration requirements

One could argue that the integration was not successful as one test scenario failed, that might compromise the security of the service. Anyhow, this is a niche case, which only occurs if the user is unauthorized from the service during a session. Even if it does happen, the session expiry mechanisms of Django are configured to automatically expire the cookie after its age exceeds four hours. Therefore, the user would have on average 2 hours to access the resources instead of 7.5 minutes with the session custom silent re-auth middleware. The difference is significant, but it still outweighs the multitude of issues caused by a blocked authorization endpoint. Root cause of the problem has not been closely examined but it is held unlikely fixable without contributing to the open source project.

6.2 Experience with the library

All in all, the *mozilla-django-oidc* was clear and easy to follow. It has nearly 30 different Django settings that can be used to customize the configuration, of which every single one is precisely described. The part of the documentation that addressed connecting OIDC user identities to Django users did not work as indicated. Anyhow, a quick look at the source code of the library, which is clean and has transparent error messages, solved the problem. Even when the settings were split in multiple modules to better isolate the whole OIDC implementation and corresponsive configurations, the library continued to work as intended.

6.3 Operating with existing installation

When integrating IAM solution with a service that has an already existing user registry, it must be ensured that there is no unintended overlapping of identifying user information fields between the user registries of the existing service and the newly integrated IAM solution. The OIDC authentication library used in this thesis connects the user identities registered to TraSim and Azure AD by claiming the identifying field of a user registered to Azure AD and mapping it to a user with equivalent identifying field on user registry of the existing service, TraSim. This mapping, or authentication process can only take place if the password of the user registered to TraSim is set as unusable, which prevents the user from authenticating via Django's core authentication system.

6.4 Azure subscription requirement

Creating the Azure AD tenant is straightforward and free of cost. Anyone with a Microsoft account can create a tenant of their own. Anyhow, the application registration, following the creation of the tenant, requires a premium Azure AD subscription. With an annual commitment, Premium P1 subscription costs $6 per user per month and P2 subscription drags the price up to $9 (Microsoft: Azure AD Pricing 2020). The P2 subscription offers the same features as P1 but it is enchanted with advanced identity protection and identity management capabilities. Fortunately, the premium features can often be trialed even with a free tier user account without a purchased subscription.

6.5 Future studies

This chapter examines the ideas for future studies that appeared during the writing process: adding support for multiple directories to the *mozilla-django-oidc* library, a review and comparison of IAM solutions and service providers and the possible utilization of Keycloak for standardizing the way application handle user management.

6.5.1 Contributing to open source

Some use cases may require a single environment of the service to be integrated with multiple Azure AD tenants. By default, this is not supported by *mozilla-django-oidc* library. Adding the support would be possible by either contributing to the project or subclassing additional classes of the library and inserting new custom logic. Option number one would be a better choice in terms of credibility but it can be a slow process and there is a possibility that the pull request is simply denied by the authors. Anyhow, it is something worth the investigation and a great chance for learning Python.

User synchronization is an additional feature that would be increasingly useful especially when operating with a large user base. It would automatically create matching users from the Azure AD tenant to the built-in user management system of TraSim. In a scenario where the tenant already contains suitable groups for the executive team, occupational health and safety personnel and more, the members can be assigned to corresponding groups on the application. The group membership status would then be kept up to date by the automatic directory synchronization when any change on the tenant side is reflected to the application. All in all, this feature would decrease the need to use TraSim’s user management and move more responsibility to the tenant and its admins.

6.5.2 Acquiring user phone number

One might have noticed that the phone number field of the profile edit forms is not set by default nor claimed from the IdP. This is due to the fact that Microsoft identity platform implementation of OIDC does not support phone scope, a collection of phone numbers registered to a user that can be represented as a claim value in the JWT or ID token (Microsoft: Permission and Consent 2020). As address is another unsupported scope, one may draw the conclusion that these restrictions are made to protect user privacy. It is possible that the phone number information could be claimed from some other API provided by Microsoft. Anyhow, this functionality can easily become complex and make the implementation unnecessarily delicate.

6.5.3 Alternative identity providers

Managing identities has become of the most challenging aspects information technology professionals face in their operating environment with an ever-widening array of software services and other network boundaries (Ferrill 2019). Therefore, a more extensive study that provides information about the IAM solutions to more effectively realize the benefits of SaaS applications and reduce their operating costs. Oftentimes these solutions provide directory services for sourcing identities and directory extension services for connecting identities in the cloud or simply to synchronize information from another system. There might exist considerably large differences in terms of usability and reliability of the IAM solutions. A company specialized in the IAM field could provide a significantly advanced solution compared to a general solution built by a universal company. Lately an open source solution, Keycloak, has been gaining a lot of interest.

Keycloak is an Apache licensed solution to add authentication and security services to applications. It was launched in 2013 by Red Hat and remains actively developed with a new release every six weeks. It supports SSO using all the standard protocols, MFA, One-time password (OTP), social login while providing centralized user management and directory services (Keycloak 2020). The project is robust with broad documentation and many examples. It is hosted on GitHub with about 280 contributors, 5600 stars and 2700 forks. According to an article by Sairam Krish, the handling of user management is a repeated process across projects, but Keycloak supports a wide variety of the use cases and saves developers from the repetitive task of writing authentication code. Hence, Keycloak integration allows developers to build an established way of setting up the user management features required by their applications. Anyhow, Keycloak brings more to the table when working technologies do not come with built-in authentication and user management systems. Hence, the use of Keycloak could be studied when working on a project with more suitable technologies.

References

Altili, E (2017). *Difference Between Azure AD vs Active Directory (AD) and AWS Directory Service* [Online]. [26.02.2020]. Available: https://medium.com/@ealtili/difference-between-azure-ad-vs-active-directory-ad-and-aws-directory-508fca4d6d0a

Amazon (2006). *Announcing Amazon Elastic Compute Cloud EC2 Beta* [Online]. [27.01.2020]. Available: https://aws.amazon.com/about-aws/whats-new/2006/08/24/announcing-amazon-elastic-compute-cloud-amazon-ec2---beta

Bertocci, V (2016). *Modern Authentication with Azure Active Directory for Web Applications* [Online]. [19.03.2020]. Available: https://ptgmedia.pearsoncmg.com/images/9780735696945/samplepages/9780735696945.pdf

Cheung, J (2016). *SAML for Web Developers* [Online]. [14.02.2020]. Available: https://github.com/jch/saml

DeMeyer, Z (2019). *The Difference Between LDAP and SAML SSO* [Online]. [17.02.2020]. Available: https://securityboulevard.com/2019/04/the-difference-between-ldap-and-saml-sso

Django (2020). *Overview* [Online]. [03.03.2020]. Available: https://www.djangoproject.com/start/overview

Django (2020). *Customizing Authentication in Django* [Online]. [19.03.2020]. Available: https://docs.djangoproject.com/en/3.0/topics/auth/customizing

Django REST Framework (2020). *About* [Online]. [24.03.2020]. Available: https://www.django-rest-framework.org/#

Docker (2020). *Overview* [Online]. [01.03.2020]. Available: https://docs.docker.com/get-started/overview/

Drinkwater, D (2018). *What is Single Sign-On? How Single Sign-On Improves Security and the User Experience* [Online]. [24.01.2020]. Available: https://www.csoonline.com/article/2115776/what-is-single-sign-on-how-sso-improves-security-and-the-user-experience.html

Electronic Frontier Foundation (2018). *Deep Dive End to End Encryption: How Do Public Key Encryption Systems Work?* [Online]. [05.02.2020]. Available: https://ssd.eff.org/en/module/deep-dive-end-end-encryption-how-do-public-key-encryption-systems-work

Ferrill, T (2019). *The Best Identity Management Solutions for 2020* [Online]. [20.03.2020]. Available: https://uk.pcmag.com/cloud-services/71363/the-best-identity-management-solutions-for-2020

Galvis, J (2018). *From Django to React: How to Write Frontend if You Are a Backend Developer* [Online]. [07.03.2020]. Available: https://iamondemand.com/blog/ django-react-write-frontend-backend-developer

Gartner (2008). *Shift to Service Based Model* [Online]. [27.01.2020]. Available: https://www.gartner.com/it/page.jsp?id=742913

Gartner (2019). *Gartner Predicts Increased Adoption of Mobile-Centric Biometric Authentication and SaaS-Delivered IAM* [Online]. [11.02.2020]. Available: https://www.gartner.com/en/newsroom/press-releases/2019-02-05-gartner-predicts-increased-adoption-of-mobile-centric

Gartner (2020*). Identity and Access Management (IAM)* [Online]. [27.01.2020]. Available: https://www.gartner.com/en/information-technology/glossary/identity-and-access-management-iam

Gartner (2020). *Gartner Predicts Increased Adoption of Mobile-Centric Biometric Authentication and SaaS-Delivered IAM* [Online]. [10.02.2020]. Available: https://www.gartner.com/en/newsroom/press-releases/2019-02-05-gartner-predicts-increased-adoption-of-mobile-centric

George, N (2020). *Django's Structure – A Heretic's Eye View* [Online]. [07.03.2020]. Available: https://djangobook.com/mdj2-django-structure

GitHub (2020). *Django-oidc-provider* [Online]. [19.03.2020]. Available: https://github.com/juanifioren/django-oidc-provider

GitHub (2020). *Mozilla-django-oidc* [Online]. [19.03.2020]. Available: https://github.com/mozilla/mozilla-django-oidc

Guest, D (2019). *ADFS Azure vs. AD: How Microsoft Changed the Authentication Game* [Online]. [24.02.2020]. Available: https://thirdspace.net/blog/adfs-azure-ad-microsoft-changed-authentication

Gunicorn (2020). *Gunicorn – WSGI Server* [Online]. [13.03.2020]. Available: https://docs.gunicorn.org/en/stable

Gunter, C., Liebovitz D. & B. Malin (2011). *Experience-Based Access Management: A Life-Cycle Framework for Identity and Access Management Systems* [Online]. [03.02.2020]. Available: https://ieeexplore-ieee-org.proxy.uwasa.fi/document/5887313

IBM (2020). *SAML Concepts* [Online]. [10.02.2020]. Available: https://www.ibm.com/support/knowledgecenter/SSGMCP_5.3.0/com.ibm.cics.ts.securityextensions.doc/topics/saml_concepts.html

Insta Group (2020). *About Us* [Online]. [24.03.2020]. Available: https://www.insta.fi/en/about-us

Jensen, J (2018). *What is Azure Active Directory?* [Online]. [26.02.2020]. Available: https://cloudpuzzles.net/2018/01/what-is-azure-active-directory

Kantor, I (2020). *An Introduction to JavaScript* [Online]. [07.03.2020]. Available: https://javascript.info/intro

Kawasaki, T (2017). *Diagrams of All the OpenID Connect Flows* [Online]. [14.02.2020]. Available: https://medium.com/@darutk/diagrams-of-all-the-openid-connect-flows-6968e3990660

Kelly, M (2020). *Is Single Sign-On a Security Risk?* [Online]. [28.02.2020]. Available: https://www.giac.org/paper/gsec/811/single-sign-security-risk/101711

Keycloak (2020). *About* [Online]. [20.03.2020]. Available: https://www.keycloak.org/about.html

Krish, S (2018). *Keycloak Integration: Overview* [Online]. [20.03.2020]. Available: https://medium.com/@sairamkrish/keycloak-integration-part-1-overview-812010d6c7cf

Latvala, J (2018). *Joining Forces with Insta* [Online]. [24.03.2020]. Available: https://www.intopalo.com/blog/2018-11-13-joining-forces-with-insta

Liu, S., Yue, K., Yang, H., Liu, L., Duan, X & T. Guo (2018). *The Research on SaaS Model Based on Cloud Computing* [Online]. [16.02.2020]. Available: https://ieeexplore-ieee-org.proxy.uwasa.fi/document/8469462

Linden, M (2017). *Identiteetin- ja Pääsynhallinta* [Online]. [02.02.2020]. Available: https://trepo.tuni.fi/bitstream/handle/10024/116591/linden_identiteetin_ja_paasynhallinta.pdf?sequence=1&isAllowed=y

Linux (2020). *What is Linux?* [Online]. [28.02.2020]. Available: https://www.linux.com/what-is-linux

Lujan, V (2019). *SSO vs. LDAP* [Online]. [17.02.2020]. Available: https://jumpcloud.com/blog/sso-vs-ldap

Makai, M (2020). *WSGI Servers* [Online]. [06.03.2020]. Available: https://www.fullstackpython.com/wsgi-servers.html

MDM Web Docs (2020). *HTTP Headers* [Online]. [07.03.2020]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers

MDN Web Docs (2020). *Django Web Framework (Python)* [Online]. [05.03.2020]. Available: https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django

MDN Web Docs (2019). *HTTP Overview* [Online]. [05.03.2020]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

Mell, P & T. Grance (2011). *The NIST Definition of Cloud Computing: Recommendations of the National Institute of Standards and Technology* [Online]. [16.02.2020]. Available: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

Microsoft (2017). *Active Directory Domain Services* [Online]. [17.02.2020]. Available: https://docs.microsoft.com/en-us/windows-server/identity/ad-ds/active-directory-domain-services

Microsoft (2020). *Azure Active Directory Pricing* [Online]. [23.03.2020]. Available: https://azure.microsoft.com/en-us/pricing/details/active-directory

Microsoft (2020). Cloud Computing Overview [Online]. [18.02.2020]. Available: https://azure.microsoft.com/en-us/overview/what-is-cloud-computing

Microsoft (2020). *Single Sign-On to Applications in Azure Active Directory* [Online]. [18.03.2020]. Available: https://docs.microsoft.com/en-us/azure/active-directory/manage-apps/what-is-single-sign-on

Microsoft (2020). *Permissions and Consent in the Microsoft Identity Platform Endpoint* [Online]. [23.03.2020]. Available: https://docs.microsoft.com/en-us/azure/active-directory/develop/v2-permissions-and-consent

Microsoft (2020). *What is IaaS?* [Online]. [02.02.2020]. Available: https://azure.microsoft.com/en-us/overview/what-is-iaas

Netcraft (2020). *February 2020 Web Server Survey* [Online]. [13.03.2020]. Available: https://news.netcraft.com/archives/2020/02/20/february-2020-web-server-survey.html

Nginx (2020). *About* [Online]. [06.03.2020]. Available: https://nginx.org/en

Nginx (2020). *What Is a Reverse Proxy Server?* [Online]. [06.03.2020]. Available: https://www.nginx.com/resources/glossary/reverse-proxy-server

OAuth (2020). *OAuth 2.0 Authorization Framework* [Online]. [15.02.2020]. Available: https://auth0.com/docs/protocols/oauth2

OWASP (2020). *Session Management Cheat Sheet* [Online]. [05.02.2020]. Available: https://owasp.org/www-project-cheat-sheets/cheatsheets/Session_Management_Cheat_Sheet

PingIdentity (2020). *What is Identity as a Service (IDaaS)?* [Online]. [10.02.2020]. Available: https://www.pingidentity.com/en/resources/client-library/articles/identity-as-a-service-idaas.html

PostgreSQL (2020). *About* [Online]. [15.03.2020]. Available: https://www.postgresql.org/about

Python (2020). *Applications for Python* [Online]. [02.03.2020]. Available: https://www.python.org/about/apps

Python (2020). *The Python Tutorial* [Online]. [02.03.2020]. Available: https://docs.python.org/2/tutorial

React (2020). *Components and Props* [Online]. [11.03.2020]. Available: https://reactjs.org/docs/components-and-props.html

React (2020). *Introducing JSX* [Online]. [11.03.2020] Available: https://reactjs.org/docs/introducing-jsx.html

React (2020). *Rendering Elements* [Online]. [11.03.2020]. Available: https://reactjs.org/docs/rendering-elements.html

Redis (2020). *Redis Server-Assisted Client-Side Caching* [Online]. [16.03.2020]. Available: https://redis.io/topics/client-side-caching

Sans (2014). *Database Credentials Coding Policy* [Online]. [02.02.2020]. Available: https://www.sans.org/security-resources/policies/server-security/pdf/database-credentials-policy

Sectigo (2020). *What Is a Digital Signature?* [Online]. [05.02.2020]. Available: https://www.instantssl.com/digital-signature

Sharma, A., Sharma, S & M. Dave (2015). *Identity and access management- a comprehensive study* [Online]. [03.02.2020]. Available: https://ieeexplore.ieee.org/document/7380701

Solomon, S (2016). *Growing Pains: Latest Research Shows IT Struggling to Meet SaaS Application Demand* [Online]. [24.01.2020]. Available: https://www.bettercloud.com/monitor/cloud-application-use-growth-impact

Sourceware (2020). *Libffi* [Online]. [02.03.2020]. Available: https://sourceware.org/libffi

Swaroop, C.H (2020). *About Python* [Online]. [02.03.2020]. Available: https://python.swaroopch.com/about_python.html

Tejada, Z., Bustamante, M.L & I. Ellis (2015). *Developing Microsoft Azure Solutions: Exam Ref 70-532*. Microsoft Press, 2015. 432p.

Tobak, S (2008). *How to Manage a Crisis, Any Crisis* [Online]. [24.03.2020]. Available: https://www.cnet.com/news/how-to-manage-a-crisis-any-crisis

Vaughn, A (2019). *Which Single Sign-On Technology Should I Choose?* [Online]. [18.03.2020]. Available: https://mindtouch.com/resources/which-sso-technology-should-i-choose

Viestintävirasto (2018). *Kryptografiset Vahvuusvaatimukset Luottamuksellisuuden Suojaamiseen - Kansalliset Suojaustasot* [Online]. [26.03.2020]. Available: https://www.kyberturvallisuuskeskus.fi/sites/default/files/media/regulation/ohje-kryptografiset-vahvuusvaatimukset-kansalliset-suojaustasot.pdf

Vincent, W (2018). *Is Django a Full Stack Framework?* [Online]. [07.03.2020]. Available: https://wsvincent.com/is-django-a-full-stack-framework