Bachelor's thesis

Business Information Technology

2021

Antti Komulainen

DEVELOPING A WEB API WITH .NET CORE ON AN AWS LAMBDA PLATFORM

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Business Information Technology

2021 | 40 pages

Antti Komulainen

DEVELOPING A WEB API WITH .NET CORE ON AN AWS LAMBDA PLATFORM

The goal of this thesis was to develop a proof-of-concept solution of a document archive service with a web API. The API would be used for managing the document uploads and downloads. The service would be built according to a specification given by the customer. The service had to be built on Amazon Web Service (AWS) ecosystem using serverless technologies and .NET Core framework. The HTTP requests had to be authorized using OAuth 2.0 technology. The system had to be able to maintain the archived data periodically and autonomously.

The API was developed on Amazon Web Service Lambda platform using .NET Core 3.1 framework. The AWS services were implemented as follows: The AWS Lambda function handles the document management. The Lambda function is available through an AWS API Gateway using REST API. The archived documents are stored on Amazon Simple Storage Service and the metadata of the documents are stored on AWS Relational Database Service using PostgreSQL database. The authorization of the requests was implemented using OAuth 2.0 and OpenID Connect standards.

The thesis project succeeded on building the proof of concept using the already mentioned methods to meet the specifications. Some of the required components were replaced with similar alternative components for the sake of simplicity but the same protocols and techniques as required were used. The project is viable for further development.

KEYWORDS:

cloud services, cloud storage, serverless computing, AWS Lambda, Amazon Simple Storage Service, .NET Core

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietojenkäsittely

2021 | 40 sivua

Antti Komulainen

VERKKORAJAPINNAN KEHITTÄMINEN .NET CORELLA AWS LAMBDA -ALUSTALLE

Opinnäytetyön tavoitteena oli kehittää toteutettavuusdemonstraatio dokumenttiarkistopalvelusta verkkorajapinnalla. Verkkorajapintaa käytettiin dokumenttien latauksien hallintaan. Palvelu toteutettiin asiakkaan antaman määrittelyn mukaisesti. Palvelu tuli rakentaa Amazon Web Service -ekosysteemiin käyttäen palvelimetonta tietojenkäsittelyä sekä .NET Core -ohjelmistokehystä. HTTP-pyynnöt oli autorisoitava OAuth 2.0 -teknologialla. Arkistodataa oli pystyttävä ylläpitämään autonomisesti ja säännöllisesti.

Rajapinta kehitettiin Amazon Web Servicen Lambda -alustalle käyttäen .NET Core 3.1 -ohjelmistokehystä. AWS Lambda käsitteli dokumentteja koskevat pyynnöt. Lambda -funktio tarjottiin AWS API Gatewayn välityksellä REST-rajapintana. Arkistoidut dokumentit tallennettiin Amazon Simple Storage Serviceen ja niiden oheistiedot tallennettiin AWS Relational Database Servicen PostgreSQL -tietokantaan. HTTP-pyyntöjen autorisointi implementoitiin OAuth 2.0- ja OpenID Connect -standardien mukaisesti.

Opinnäyteprojekti onnistui, ja tulokseksi saatiin määritysten mukaisesti toimiva konseptiversio. Joitakin määriteltyjä komponentteja korvattiin samankaltaisilla vastaavilla komponenteilla yksinkertaisemman rakenteen saavuttamiseksi. Komponentit käyttivät kuitenkin samoja protokollia ja tekniikoita kuin määrittelyissä vaaditutkin. Palvelua voidaan jatkokehittää tämän projektin tulosten pohjalta.

ASIASANAT:

pilvipalvelut, pilvitallennus, palvelimeton tietojenkäsittely, AWS Lambda, Simple Storage Service, .NET Core

CONTENTS

FIGURES

PICTURES

LIST OF ABBREVIATIONS

API	Application Programming Interface (Fielding, 2000)
ARN	Amazon Resource Name (AWS, 2021i)
AWS	Amazon Web Services (AWS, 2021a)
DNS	Domain Name System (IETF, 1987)
GUID	Globally Unique Identifer (IETF, 2005b)
HTTP	Hypertext Transfer Protocol (IETF, 1996)
IANA	Internet Assigned Number Authority (IANA, 2021a)
IETF	Internet Engineering Task Force (IETF, 2012a)
JSON	JavaScript Object Notation (Ecma International, 2017)
JWT	JSON Web Token (IETF, 2015a)
OIDC	OpenID Connect (OpenID Foundation, 2021a)
OIDF	OpenID Foundation (OpenID Foundation, 2021b)
PKCE	Proof Key for Code Exhange (IETF, 2015b)
RDS	Relational Database Service (AWS, 2021e)
REST	Representational State Transfer (Fielding, 2000)
S3	Simple Storage Service (AWS, 2021f)
SAM	Serverless Application Model (AWS, 2021m)
SDK	Software Development Kit (Gartner, 2021)
UML	Unified Modeling Language (ISO, 2005)
URI	Uniform Resource Identifier (IETF, 2005a)
VPC	Virtual Private Cloud (AWS, 2021g)

1 INTRODUCTION

The goal of this thesis is to develop a document archive service with a Web API following the specification given by the customer. The specification is described in the following section in detail. The API's purpose is to control a document archive and it is meant to provide a more flexible alternative with a more modern solution for an existing system. The service consists of a user interface, web API, document storage, metadata database and access control components.

The main focus is on the service's fundamental functionalities. The service is a proof-of-concept and the service is not fully production ready after the development process. There are, for example, environment-related issues that affect to the security, authentication, and authorization properties of the service. Those are out of the scope of this thesis.

The thesis is structured as follows: Chapter 2 introduces the feature and technology requirements given by the customer. Chapter 3 briefly describes the different .NET frameworks available and the reasons behind the selection of the framework used in this thesis project. Chapter 4 introduces the Amazon Web Services and the individual services used in the project. Chapter 5 goes into the details of OAuth 2.0 standard and the OpenID Connect extension built on the standard. Chapter 6 explains the implementations of the features and technologies. And finally, chapter 7 shares the results of the project and points out some further development ideas and notes gathered during the process.

2 CUSTOMER SPECIFICATION

The customer of the project has given a specification for the service. The specification defines some of the key features of the service that it must or should have. The main platform and some of the technologies used in the system are defined in the specification to ensure the conformity with the existing systems of the customer.

The key features of the user interface are search, download and upload functionalities. The API must have endpoints for the functionalities. The user must be able to search documents with metadata and download the found documents. The user must also be able to upload documents and the related metadata into the document archive.

The customer requires that the system must be built on Amazon Web Service (AWS) platform. The system must be a so called serverless solution and the system must use AWS S3 cloud service as a document storage solution. The document upload process must be designed to be fail safe. The metadata related to the documents must be stored on a separate database within the same fail-safe storage process. The metadata database should use PostgreSQL technology if applicable.

The access to the service must be secure. All the requests made to the service by the user must be authorized with OAuth 2.0 technology using a Keycloak server. The authorization must be controlled through user roles and the access to the documents must be restricted depending on the user roles and the document metadata.

The system must autonomously be able to remove documents and metadata related to documents that are older than a specified time threshold defined by the system administrator. The system must maintain the document data periodically according to intervals defined by the system administrator.

3 .NET CORE

There are couple of different software frameworks that use the .NET name or prefix. Let's first clarify which is what in chronological order. The first framework is Microsoft's .NET Framework. It is a predecessor to the current .NET Core frameworks. The .NET Framework is targeted only to the Microsoft Windows ecosystem and provides tools for development of applications and web services for Windows based systems. The last version is .NET Framework 4.8 which was launched in 2019. It is still supported and actively used, but it is not recommended by Microsoft to start any new projects built on it. (Microsoft, 2020b)

The next framework of the .NET family is the .NET Framework's successor .NET Core. It is a cross-platform open source software development framework. It supports several operating systems including the most commonly used Windows, macOS and Linux. It is free to use and it is released under the MIT and Apache 2 licenses. The last version under the .NET Core name is 3.1 and it was released in 2019. (Microsoft, 2020a)

The latest release of the .NET framework family is .NET 5.0 framework. It is based upon the .NET Core 3.1 and it is similarly open source and free to use. The naming of the framework has skipped 4.0 to avoid the possible confusion with the original .NET Framework. It has also dropped the Core extension of the name because it is the sole main branch of the .NET framework that is going to be further developed in the future. The version was released in 2020 and there are already projected schedules for versions 6, 7 and 8. (Microsoft, 2020a)

The thesis project implements .NET Core 3.1 framework. It is selected because, as the Amazon's documentation (2021k) confirms, the AWS Lambda functions do not directly support the use of the .NET 5.0 framework at the moment. The documentation notes that is possible to run the framework in Lambda function, but it would require using Docker containers to host the function. This would unnecessarily complicate the application. In addition, there is no significant benefit on running the latest version as all the features required for the function are already available on .NET Core 3.1.

.NET Core framework supports C#, F# and Visual Basic languages. The framework specifies .NET Standard APIs which form the base for the framework. The framework can be extended with NuGet packages containing libraries. (Microsoft, 2021b)

4 AMAZON WEB SERVICES

Amazon Web Services (AWS) is a part of the Amazon.com Incorporation. AWS was founded in 2006. According to their own words “AWS provides a highly reliable, scalable, low-cost infrastructure platform in the cloud”. Their services include a vast variety of cloud computing services to businesses and individuals around the globe. AWS announces low cost, agility, flexibility, and security as the main benefits on running services on their platform. AWS use pay-as-you-go pricing on their services. This means that the customer pays only for the services they use without any up-front payments. (AWS, 2021a)

AWS has a global infrastructure of regions which are physical locations around the world containing logical availability zones. Each region contains multiple availability zones which are physically separate clusters of data centers grouped as a one logical data center. In addition, there are also available for an example local zones which improve performance of some services on targeted local areas. The infrastructure provides availability, reliability and performance through redundancy and geographical coverage. The infrastructure enables to orchestrate services, so the most suitable combination is used depending on the customer’s needs. (AWS, 2021q)

According to AWS (2021h) the different services can be developed and controlled with a multitude of different tools. The first one and the only one when starting the use of AWS is the management console which offers a graphical user interface for the services. In addition to the web console AWS introduces command line tools which allow to perform activities on command line interface and to write scripts for automation purposes. AWS also offers plugins and SDKs for different integrated development environments and languages. On some services there is also an HTTPS API available for programmatical use. All the tools enable the user to perform configuration and management activities within the limits of the explicit permissions granted to the currently active user. AWS introduces on their website (2021t) a plethora of tools for developers to use sorted by different use cases.

The following sections describe the services and concepts applied in this project. The descriptions focus on the features of the services which are specifically applied to this project and are not exhaustive. There are some configurations and properties which do not affect this project but should be applied to some other use cases. There are also

some features which are out of the scope of this thesis and therefore are not discussed. It is always a good practice to reference documentation and best practices provided by AWS to ensure availability, performance, and the security of the services.

4.1 Security in AWS

Security of the services is one of the most important things in today's internet. AWS uses a shared responsibility model in their services (AWS, 2021s). This means that the security of the service is divided between AWS and the customer. According to the model AWS is responsible for protecting the infrastructure that the service is running on. This includes all the hardware, networking and software used by the platform. The model defines also that the customer is responsible of the security of using the services and the scope of responsibility depends on the used services. AWS refers to this model as the "Security of the Cloud vs. Security in the Cloud" (AWS, 2021s).

Responsibilities of the Security of the Cloud consists of software and hardware sections (AWS, 2021s). Software section includes all the software related to computing, storages, databases and networking of the underlying systems. These provide the environment of the service that the customer is using. The hardware section includes regions, availability zones and edge locations which provide the physical hardware and networking infrastructure the software is running on.

Security in the Cloud describes the responsibilities of the customer concerning the security of the AWS services (AWS, 2021s). AWS notes that the responsibilities depend on the services that the customer is using. Some infrastructure services require the customer to configure all the security measures of the system and some more abstract services require less configuration. In general, according to the shared responsibility model, the customer is responsible of securing the platform and applications, the data in the system, the access management, and the configurations of the system.

System security is fundamentally very difficult to accomplish as Saltzer and Kaashoek (2009) point out. They promote using the principle of least privilege and implementing explicit permissions to perform actions. Principle of least privileges tries to prepare for the inevitable and limit the damages when unauthorized activities happen. The principle says that a user should have only the privileges that are necessary, and they should be used only when needed. Using the explicit permissions enforces the principle by denying

the user activities by default unless it is explicitly allowed. These two fundamentals are enforced if the recommendations and instructions of the AWS documentation (2021j) are followed.

AWS controls users' and services' authentication and authorization with Identity and Access Management (IAM) service. The first user to be created for the account is a root user which has privileges to perform any activities available in the service. It is recommended by AWS to use root privileges only to perform a restricted set of strictly necessary activities and to use other users to perform day to day activities. As a very simplified version of the authorization model the authorizations are controlled by defining users, groups, and roles. A user can be added to a group that is granted with similar privileges. The users and the groups or services are referred as principals within the AWS system. The principals can be granted a permission to assume roles which have certain policies attached to them. A policy defines a permission to allow or deny different actions or operations on different resources on certain conditions. The permissions are explicit and if the action is denied in any policy or it is not specifically allowed in any policy the action is always denied. In practice the permission system is a lot more complicated and the detailed instructions and recommended best practices are available in the IAM documentation. (AWS, 2021j)

AWS provides a Virtual Private Cloud (VPC) service which can be used on networking different services together securely. The VPCs are logically isolated and they make it easier to manage the access and control the internal networking of different cloud services. VPC allows also for an example to manage routing for external services such as authentication server or allow incoming connections for database management. (AWS, 2021g)

AWS provides extensive documentation (AWS, 2021l), best practices (AWS, 2021p), tools (AWS, 2021o), technical guides and reference materials (AWS, 2021r) which provide assistance on security related issues.

4.2 AWS API Gateway

AWS API Gateway is a service which enables control over REST, HTTP and WebSocket APIs. The HTTP and WebSocket APIs are not covered here as they are out of the scope of this thesis. The service allows to create, control, and secure the APIs. It enables the

customer to create a controlled access for a client to an application, a resource or to another AWS service through the gateway. (AWS, 2021b)

According to the service's documentation (AWS, 2021b) the HTTP and REST APIs created by the service follow the constraints of REST architecture. The REST architecture (Fielding, 2000) defines constraints which specify high level guidelines to the APIs. The service's documentation state that the APIs are HTTP-based, enable stateless client-server communication, and use standard HTTP methods.

While choosing which API type to use one must consider the needs of the use case in question. The documentation (AWS, 2021b) lists some core features which guide the selection of the API type. In this use case the REST API was selected for the more advanced control over the requests and responses compared to HTTP APIs. REST API also allows the use of API keys to improve security of the API.

The API Gateway service enables the control of multiple API stages and deployments. It is possible to have multiple versions of the same API at the same time as different stages. This allows a lifecycle control of the APIs. For an example there can be a development version and a production version of the API with different access privileges. This is very convenient while developing, testing, and releasing new features as the modifications can be made without affecting the production use. (AWS, 2021b)

The deployed API has one or more resources and methods which can be configured to react to the request accordingly. The methods can be configured individually, and it is for an example possible to validate payloads at the gateway level if necessary or process the request otherwise. It is also possible to authorize requests on the gateway level. After the request is processed the data required by the target or the whole request is passed forward to the target endpoint. (AWS, 2021b)

This thesis project uses the Lambda proxy integration which relays the entire incoming request to a Lambda function for further processing. According to the AWS documentation (2021b) it is possible to relay all HTTP methods using general ANY method. In this instance the methods are restricted to a limited selection by the gateway. This helps to ensure that the receiving endpoint does not receive requests that it is not able to handle.

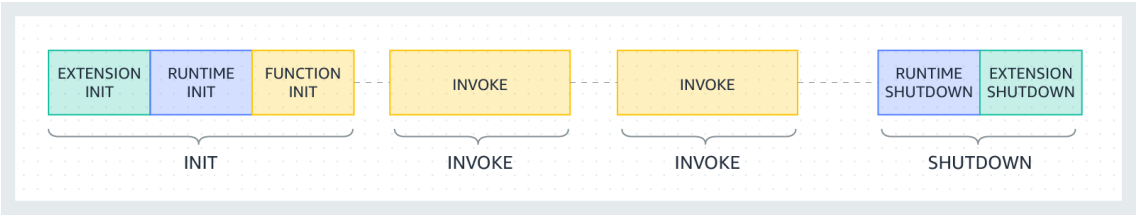
According to the AWS website (2021c) the pricing model of AWS API Gateway is very simple. On HTTP and REST APIs the pricing is based on the number of incoming

requests and the amount of data transferred. Other optional features might increase the cost. The documentation of the API Gateway (AWS, 2021b) notes that any failed or invalid requests are not charged.

4.3 AWS Lambda

AWS Lambda is a service which enables running code in a serverless environment as a function. Serverless environment means that AWS takes care of the underlying infrastructure which the function requires to be able to run the code. The customer configures the function, uploads the code to be executed and determines the triggers which invoke the function. The function executes when invoked and after the execution has finished it shuts down. Lambda functions support several languages using specific runtimes. Supported runtimes include for an example Node.js, Python, .NET Core, Java, Ruby and Go. It is also possible to run container images on Lambda functions by modifying base images provided by AWS. (AWS, 2021k)

The invoked function runs inside of an isolated environment. The environment controls the underlying infrastructure and provides lifecycle support for the runtime which the function is running on. The lifecycle consists of three phases as depicted in the Picture 1. The first one is the init phase which creates or revives the execution environment. The environment is set up according to the configuration and it prepares to run the actual function. The second phase is the invoke phase which invokes the function handler containing the business logic. The invocation phase can be run repeatedly within the same existing environment without running the init phase every time unless the environment is shut down. The last phase of the lifecycle is shut down phase which tries to shut down the runtime in controlled fashion and releases the reserved resources. (AWS, 2021k)



Picture 1. Lambda execution environment lifecycle. (AWS, 2021k)

The time frame of the lifecycle depends on the configuration of the execution environment and the processing of the handler function. The init phase has a fixed timeout value of 10 seconds and if it is not completed on time it is retried in the beginning of the first function invocation. The Lambda configuration has a timeout value to limit how long the invoke phase can run. The default value is 30 seconds, but it can be set up to 900 seconds. If the handler function is not completed successfully on time the runtime is shut down and execution environment is restarted. The shutdown phase has a fixed limit of two seconds, and the process is terminated forcefully if all the resources are not able to perform controlled shut down on time. (AWS, 2021k)

The environment instance is not deleted immediately after the completion. The instance remains frozen for some time and is ready to be reused if a new invocation request arrives. This has some implications which must be taken into account while designing the function. For an example database connection object might exist if it was created outside of the handler function but this cannot be assumed to be true every time the handler function is invoked. (AWS, 2021k)

AWS Lambda requires all the supported runtimes to implement a programming model that defines interfaces which enable the customer's code and the execution environment to co-operate. The model defines a handler interface which is the entry point of the customer code. The handler method of the function is defined in the environment configuration. The handler method receives as parameter an event object or a custom input depending on the source of the invocation. The object contains information which the invoking source has sent with the request. The method can optionally receive also a second object which is the context object. The object holds information and methods related to the execution environment. The handler method returns a response. If the method is asynchronous the return type should be void. If the method is synchronous the return type can be any supported data type that is available. It must be taken into consideration that the functions are stateless and any persistent information the method might produce must be recorded somewhere outside of the function if not returned with the response. (AWS, 2021k)

Lambda functions can be integrated with different services to be triggered by events created by the services. The triggering event can be an external request, a scheduled internal event, a direct request from an AWS service, or a lifecycle event on some of the AWS services. Lifecycle events do not trigger the Lambda function directly and they must be read through an event source mapping. The mapping reads items from the source

service and handles them accordingly. Some of the AWS services invoke Lambda functions asynchronously. In such a case the function must take care of the possible error handling and further actions required by the erroneous processing. Synchronous invocations receive responses as usual. (AWS, 2021k)

Lambda functions scale up and down in a very flexible manner. When a request arrives and all the currently existing environment instances are still processing the previous requests Lambda fires up a new instance for the new request. After processing a request, the execution environment stand by for some time waiting for incoming requests and if none arrives they shut down. Basically, there is always an environment available or created for the request unless the regional quota reserved for the function is reached. In that case the function throttles which means that the function invocation is prevented. (AWS, 2021k)

According to the AWS documentation (2021k) there are two ways to control the concurrency of Lambda function instances. The first one is the reserved concurrency and the other one is the provisioned concurrency. The reserved concurrency makes sure that there is a certain number of instances available for the specific function to use. These instances are not available for any other function and if they are not used, they are shut down but ready to be fired up if needed. The provisioned concurrency keeps a certain number of instances ready to receive requests and they are not shut down. This lowers the latency of request handling, but the instances are also using resources and generating costs while just standing by. The documentation also notes that these configurations can be managed with Application Auto Scaling feature which allows to manage the values automatically by defining a schedule or a set of rules based on the utilization levels. If quota or concurrency limits are reached the function throttles.

AWS Lambda documentation (2021k) states that a Lambda function requires a permission to execute the functions and also an explicit permission to use other AWS services integrated to the function. The documentation recommends that the permissions are managed and distributed through roles using AWS Identity and Access Management service. Every function can be configured with an appropriate execution role which allow the correct combination of permissions.

The latest currently available documentation (AWS, 2021k) claims that there are no costs for creating the Lambda functions. The documentation emphasizes that the costs of running functions consists of the actual processing time of the functions and the data

transfer between integrated services. The documentation also notes that some of the optional features for an example concurrency related features generate additional costs.

4.4 AWS Simple Storage Service

AWS Simple Storage Service (S3) is a data storage. It is designed to be easily accessible and to be able to store any amount of data with highly scalable and reliable infrastructure. The service is designed to be very simple and robust. The data is stored into buckets as objects and the service does not take any stand on the structure of the data content itself. (AWS, 2021f)

The data is stored as objects and the object consists of the file data and metadata attached to it. The data can be basically any file containing bits. The maximum size of an object is five gigabytes using a single upload request and can be extended up to five terabytes using multipart upload requests. The metadata is a set of name-value pairs which can contain any information about or related to the object. There are some default system-defined metadata values such as creation date automatically attached to the object. The object metadata also contains some values defined in the HTTP standard such as Content-Length which are automatically set by the system. Other values are user-defined metadata set and controlled by the user. The user can add additional custom metadata values as necessary. The objects can have versioning enabled if multiple versions of a single object is required. (AWS, 2021f)

Every object has an identifier key which is unique within a bucket. The key is specified when the object is created, and it is a sequence of Unicode characters. The length of the UTF-8 encoded key cannot be longer than 1024 bytes. The key can contain any UTF-8 character, but it is recommended by AWS to use a set of safe characters to ensure conformity with some applications and protocols. The key can be used to form logical hierarchies within a bucket by using prefixes and delimiters in the keys of the objects. (AWS, 2021f)

Buckets are containers for the objects. Every object belongs to a bucket. The buckets do not have any substructures. They can be used to organize access control and namespaces within the account. The bucket must have a globally unique and dns-compliant name. The bucket is created into a certain region. The documentation assures that the objects in a bucket do not leave the region at any time if not explicitly transferred

elsewhere. The bucket can be configured to manage object lifecycles. The configurable rules can be set for an example to archive or remove an object after a certain period of time. (AWS, 2021f)

S3 is equipped with a REST API which it is used through. The API uses standard HTTP requests, and it is possible to use with any application or tool that supports the standard. There are SDKs available to simplify the programmatical usage, but it is also possible to use regular HTTP requests to manage the storage. Unless using anonymous requests all the requests must be authenticated. The SDKs are recommended especially with authenticated requests as they make the process of creating valid signatures for the requests considerably more straightforward and less error prone. (AWS, 2021f)

The pricing model of S3 is similar to other AWS services. User pays according to the usage of the service. S3 pricing is based on six cost components: storage, requests, data transfer, replication, management and analytics features, and S3 Object Lambda. The storage is priced at gigabytes per month basis and there are different storage types for data with different access requirements. Request cost is based on per thousands of requests and what data is targeted with the requests. Data transfer cost is simple per gigabytes of data transferred. Management and analytics costs as also the S3 Object Lambda costs depend on the features enabled and are optional. (AWS, 2021f)

4.5 AWS Relational Database Service

Amazon provides several different relational databases as a service. There are MySQL, MariaDB, PostgreSQL, Oracle, Microsoft SQL Server, and Amazon Aurora databases available. The service handles the underlying infrastructure, and the user only has to manage the configuration of the database and the service itself. The service is scalable, and the user can configure the service to use the appropriate amount of resources to meet the performance requirements. The service can be configured to automatically take care of backing up the data and redundancy is also available for enhanced availability. (AWS, 2021e)

Usage of a database running on Relational Database Service (RDS) is very similar to a regular database running on a dedicated server. After the service is fired up and configured accordingly the database is available for connecting. The database can be utilized with any standard sql client for an example with the database providers own client

application. The database instance is recommended to be run inside of a virtual private cloud (VPC). The access to the database requires granting an explicit permission for the client to access the VPC by granting an access through a security group. If the client is outside of the AWS ecosystem there must be also a network gateway available for the client to be able to connect to the service. The access can be restricted with client's VPC or IP address. (AWS, 2021e)

Each database instance has dedicated resources assigned to it. The resources dedicated for the database affect the performance and the amount of storage available. The amount of resources reserved for the instance affect also to the cost of running the instance. There are several database instance classes which determine the amount of reserved computation and memory capacity. There are some classes which have special properties designed for a certain usage such as memory-intensive or high throughput applications. There are also more flexible instance classes available which are able to momentarily provide extra performance if required. The instance class can be modified during the use to optimize the balance of performance and costs. The storage type can be selected according to the usage and required responsiveness of the database. The selected database provider and the storage type affect to the minimum and maximum size of the storage available. (AWS, 2021e)

Database services can utilize the availability zones to improve availability. The database can be configured to utilize multiple zones by Multi-AZ deployment. Each zone is isolated from each other and the service can recover from a availability zone failure using the other zones. One of the database instances is assigned as the primary instance and the secondary instances on other availability zones are replicated synchronously. On failover the primary instance is replaced with a secondary instance until the primary instance has recovered. (AWS, 2021e)

As an example of the pricing model the RDS with PostgreSQL database engine is based on five components: database instance type, storage type, backup storage, snapshot export and data transfer. The database instance cost depends on the selected instance class and the use of single or multiple availability zone deployment. There are two pricing options depending on how the instance cost is billed. The reserved instance is billed with a cheaper but continuous rate for a longer period of time while the on-demand instance is billed with a higher rate but only according to the processing time used. Storage type cost depends on the selected storage format with a varying billing terms. Backup storage cost is included if the backup is stored in the same region and the used storage space is

no more than the total amount of storage reserved for the service. The excess storage and storage used on other regions are billed accordingly. Snapshots and the data transfer are billed per gigabyte basis. Other database engines may be priced differently, and some engines might include additional licensing fees. (AWS, 2021d)

5 AUTHORIZATION

Authenticity, integrity, and authorization are defined by Saltzer and Kaashoek (2009) as the three corner stones of protecting computer systems. According to their definition authenticity defines if the agent or user in this case are who they claim to be. Integrity means in their definition the integrity of the authenticity claim. Authorization is defined as having a permission to perform some action as the authenticated user.

Authentication and authorization are easily mixed up with each other. In this thesis the focus is on authorization as the developed service does not really take a stand on who is using the service and does not require the authentication information. There is a third-party service used to authenticate and deliver the authorization information about the end user. The authorization of the end user is requested using OAuth 2.0 framework and OpenID Connect which is built on OAuth 2.0.

OAuth authorization enables authorizing a user to a resource without authenticating the user directly to the resource server. This is especially convenient when the user does not trust the target service completely or there is no reason to hand over the identity information to the target service. OAuth reduces the amount of identification information shared with the resource and the users have more control over the use of their credentials. (IETF, 2012a)

OpenID Connect (OIDC) is an identity layer built on the OAuth 2.0 framework and it allows authenticating a user without delivering the user's actual credentials directly to the resource server (OpenID Foundation, 2021a). OIDC focuses more on who the user is unlike OAuth 2.0 which should only be used to authorize users.

The following sections describe the OAuth and OIDC from the point of view of this thesis. There are aspects on these technologies which are not discussed because they are out of the scope of this thesis. It is always recommended to consult the official up to date documentation while implementing them.

5.1 OAuth 2.0

The OAuth 2.0 Authorization Framework is based on a proposed standard by the Internet Engineering Task Force (IETF, 2012a). The proposed standard defines the purpose of the framework as follows:

> *"The OAuth 2.0 authorization framework enables a third-party application to obtain limited access to an HTTP service, either on behalf of a resource owner by orchestrating an approval interaction between the resource owner and the HTTP service, or by allowing the third-party application to obtain access on its own behalf."* (IETF, 2012a)

The standard (IETF, 2012a) defines a communication protocol and a flow of activities for the authorization process. The flow includes redirecting the client into an authorization server for authentication. The authorization server delivers the client an access token which can be used to authorize access to the resource server. The abstraction of the authorization process is presented in Figure 1. The standard specifies that the authorization server and the resource server can be but are not required to be the same server. The standard's definition for a client is very broad and includes basically any application trying to access the resource.

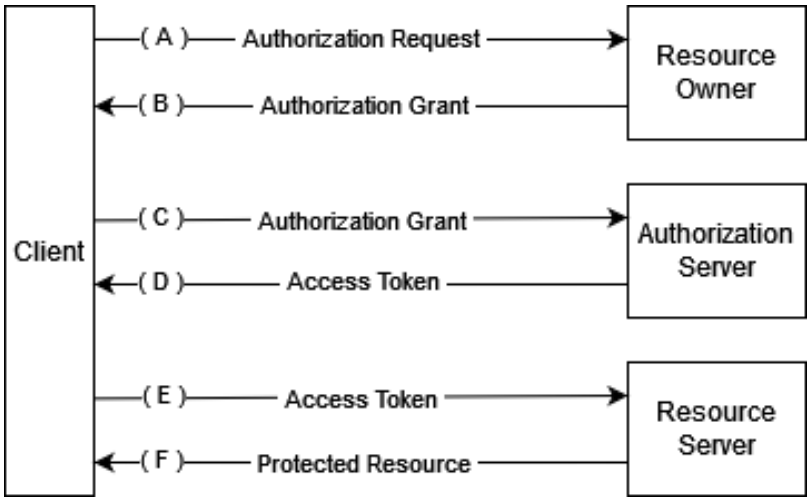


Figure 1. Abstract protocol flow (IETF, 2012a).

Authorization cannot be successfully completed by anyone just capable of accessing the authorization server. The user making the request must have an authorization grant requested from the resource owner. The grant contains a proof of ownership for an

example credentials of the resource owner, the information identifying the target resource and URI of the endpoint where the authentication must be done. If the information attached to the grant is not valid the request is denied or dropped. (IETF, 2012a)

With the grant the user is allowed to perform the authentication on the authorization server. If the authentication is successful, the user is redirected back to the client with the authorization code as a query parameter. The client is able to request an access token directly from the authorization server with a valid authorization code. The client can use the access token to authorize the access to the resource server. The flow of requesting the authorization code and the access token is depicted in detail in the Figure 2. (IETF, 2012a)

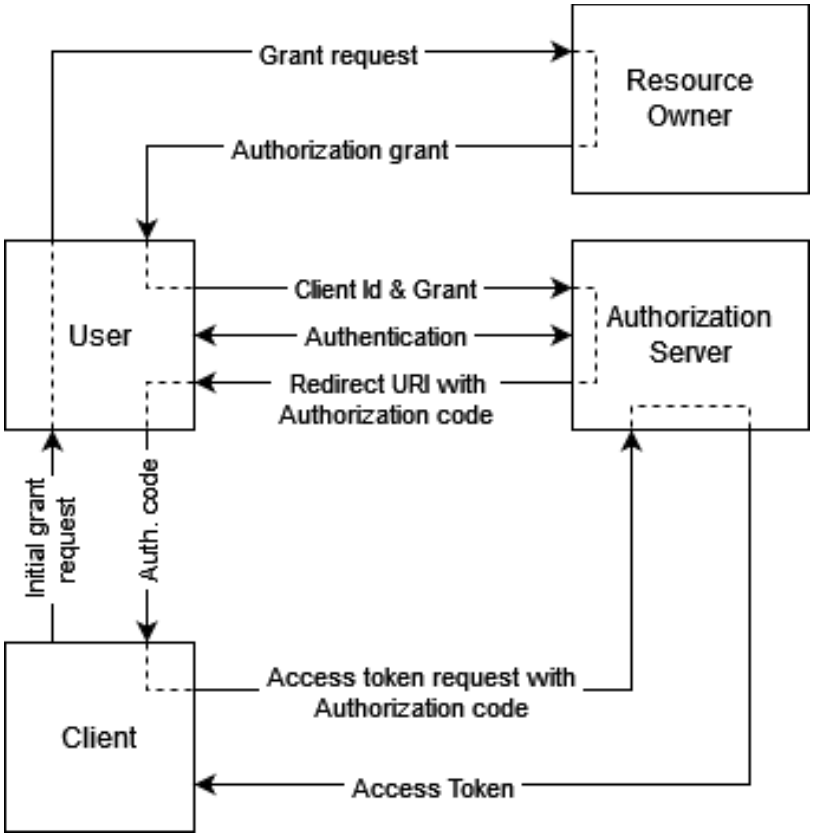


Figure 2. Authorization code flow (IETF, 2012a).

The client must register to the authorization server in a separate process before the authorization can take place. This ensures that the authorization server can trust the client when requesting tokens and reduces the chance of access tokens being delivered to fake clients. In the registration the client provides the redirection URI which the

authorization server redirects the user to after the authentication. In this use case the client receives a client identifier and a client secret from the authorization server which are used on validating the requests. (IETF, 2012a)

The OAuth 2.0 standard defines an optional scope parameter for the authorization request. The value can be used to further specify the target resource of the request. The value can contain a single scope, multiple scopes or it can be omitted. The authorization server processes the scopes of the request and grants or denies the authorization accordingly. If the scope is omitted the pre-defined default value on the authorization server can be used if available or the request is denied with invalid token response. If the request contains scopes not available to the resource the server is allowed to ignore them. In such a case the server must include the granted scope in the response. (IETF, 2012a)

The OAuth 2.0 standard (IETF, 2012a) introduces various safety considerations to be taken into account while implementing the standard. In addition to other measures the standard requires that the authorization code must have a short expiration time to minimize the timespan when the authorization code can be misused.

There are also additional standards available for reinforcing the authorization process. IETF introduces the Proof Key for Code Exchange (PKCE) standard (IETF, 2015b) which defines an additional key to be used to mitigate authorization code interception attacks. The PKCE standard defines a cryptographically random code verifier that is created for every request. The standard defines that a code challenge is derived from the verifier and the challenge is added to the authorization request. The standard notes that the authorization process continues as usual on other phases but when the authorization code is used to fetch the access token the code verifier is sent with the request. The authorization server can validate the request by calculating the code challenge from the code verifier and comparing it to the authorization request's code challenge. The standard assures that if the authorization code has been compromised it cannot be used to fetch the access token without the possession of the code verifier.

The OAuth standard is accompanied with another standard defining the bearer token usage (IETF, 2012b). The bearer token usage standard defines the details of the bearer token which can be used as an access token. The standard enforces the use of encrypted network traffic while using bearer tokens. The bearer token is a proof of authorization to do something by the possession of the token. The standard defines three

methods which can be used to send the token. The recommended ways are to send the token as request header field or as a form-encoded body parameter. If these both are impossible the standard defines a third method by sending the token as a query parameter, but this is very insecure and should be avoided if possible. The form of bearer token can be an obfuscated string of characters containing the token or it can be in the form of a JSON Web Token (JWT) which is defined in a separate standard (IETF, 2015a).

5.2 OpenID Connect

OpenID Connect (OIDC) can be used to identify and authorize a user for a service and to deliver some information about the user at the same time. OIDC uses the OAuth 2.0 framework and the bearer token as a basis for the token exchange. The authorization process is similar to the OAuth 2.0 but the authorization request must have openid scope attached to it. The response is an ID token instead of an access token. The token is in the form of a JSON Web Token (JWT) containing the information about the user. (OpenID Foundation, 2021a)

The ID token contains claims which are defined in the JWT standard (IETF, 2015a). The standard defines a group of registered claim names which form a basic set of claims which can be used. Use of the registered claims is defined optional in the standard but they cover some default items such as issuer, subject and expiration time claims. In addition to the default claims the standard allows to use public claims registered in the IANA JWT claims registry (IANA, 2021b). The JWT standard allows the authorizer and the client also to agree upon using a custom set of private claims if necessary, but this is to be used with caution due to the risk of collision with other claims.

The claim of an ID token is some information about the subject which the authorizer assures to be valid. The claim constructs from a key-value pair which contains the name of the claim and the value of the claim. The name is a string, and it has to be unique within the claims set. The value of the claim can be any valid JSON value. (IETF, 2015a)

6 DEVELOPMENT PROCESS

This project's goal was to build a proof of concept of a document archive using customer specified technologies. Addition to that an unofficial personal goal was to loosely follow the fail fast philosophy on the development process. The philosophy revolves around the notion of proving quickly if something works or not and iterating that until finding the solution as Adam Savage summarizes effectively in a short interview (Alexa Developers, 2019). He reminds that iteration that fails to find a working solution is not a failure but instead it successfully eliminates a wrong solution. In this project the idea was to build a fairly crude pipeline that proves that the components work, and the system is possible to build with the intended components.

The development process started with a technology overview based on the customer specification. The overview's function was to get an overall conception of the possibilities and restrictions of the technologies required in the specification. During the overview the use of the components in the system were further detailed and sketched into a system architecture using unified modeling language (UML). The system architecture was presented to the customer at a very early stage and through a couple of iterations the current version was fixed in place. The final iteration of the system architecture is depicted in Figure 3.

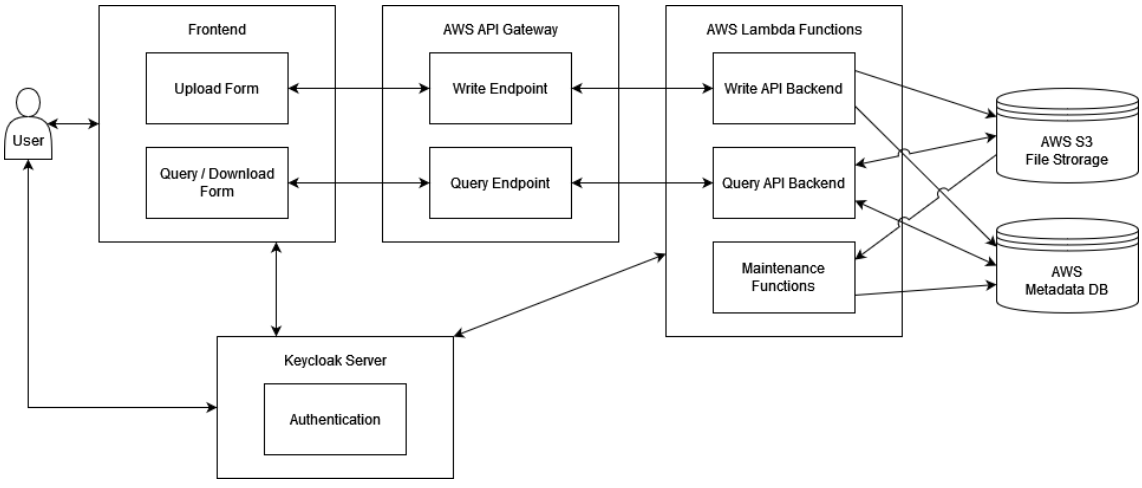


Figure 3. System architecture.

With the system architecture available it was possible to design the details of the process which the user's actions required. There were a couple of obvious use cases: the upload

of a document and the search of a document. The use cases were modeled with sequence diagrams using UML to clarify the required process. As an example of the diagrams the upload sequence diagram is attached as Figure 4.

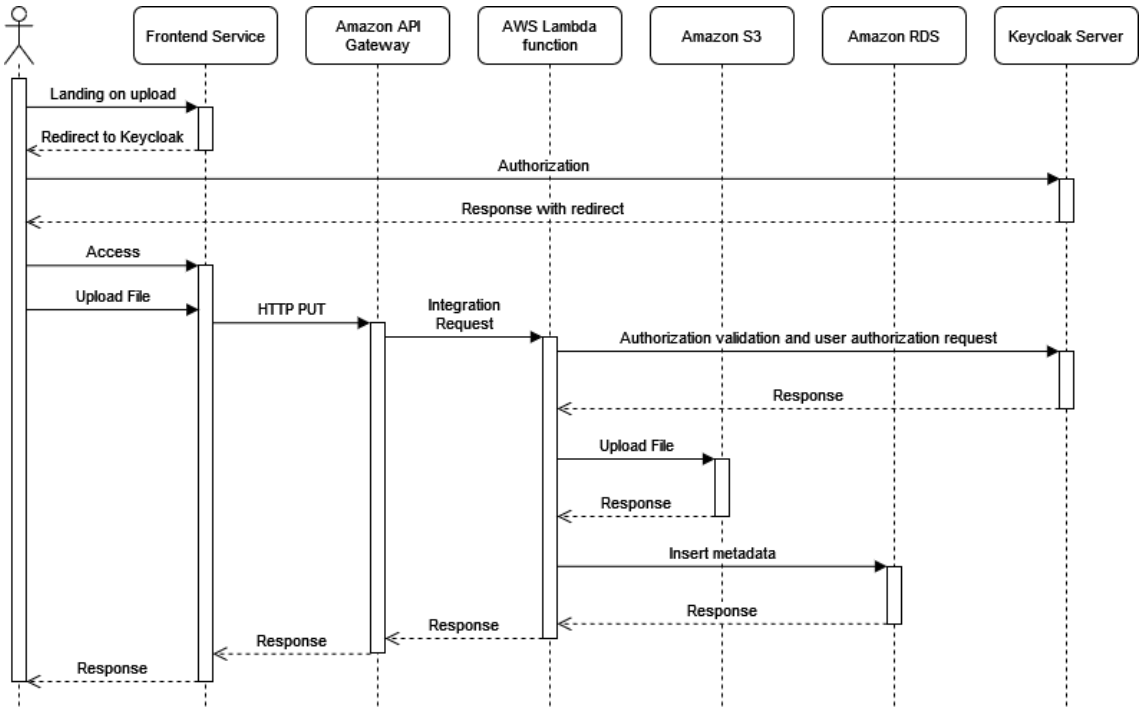


Figure 4. Upload sequence diagram.

The tools to be used in the development were selected based on the .NET Core framework and C# language used on the project. As mentioned in the section 4 AWS delivers a wide range of tools for the development. In this project the main tools were the AWS Management Console and Microsoft Visual Studio extended with AWS Toolkit for Visual Studio extension. Git was used for version control.

6.1 Frontend

The frontend service is very simple and it is meant to emulate the possible future usage of the system. It is just barely in the scope of the thesis, and it can be replaced by integrating the user interface into another web service. The main responsibilities of the frontend are redirecting the user for authentication to the authorization service and delivering the access token with the request to the archiving system. It is modified out of a default ASP.NET Core MVC template available in Visual Studio.

The most significant modification to the template was implementing the authorization of incoming requests. It was configured to use cookies and OpenID Connect using methods provided by ASP.NET Core's Authentication NuGet package and applying the documentations (Microsoft, 2021a) instructions. The instructions did not apply out of the box and had to be supplemented with the authorization service's documentation (Okta, 2021). The authorization service's details are further discussed in the section 6.6.

The authorization is applied to the requests as middleware. The middleware checks the user's authorization and if not valid, it takes the necessary steps to authorize the user. The claims received from the authorization server are added to the user object residing in the request context object. The access token delivered by the authentication service is added to the cookie, so the user does not have to authenticate every time accessing the service. It is good to point out that the expiration time of the access token must be limited to an acceptable timespan to minimize the chance of misuse.

The frontend has only two functions available for the user. The user is served with a user interface for uploading the document and for searching and downloading the documents. The user interface is very rugged and does not contain any excess functionalities. The usability or layout of the user interface was not taken into consideration.

The controllers of the frontend authorize the user for the action, validate the users' inputs, and build the requests to the AWS service. The controllers fetch the user claims from the request context object and validate the permission to perform the action. The controllers also attach the OAuth 2.0 access token to the outbound requests. The requests must be made using encrypted connection so the access token cannot be extracted if the request is intercepted in transit.

6.2 Amazon API Gateway

Amazon API Gateway functions as a gatekeeper for other services. In this case the gateway is configured to validate the requests and to redirect the valid ones to the Lambda function. The gateway is set to allow three HTTP methods: GET, POST and PUT.

All the methods require an API key which must be included on the request header. The API key does not take any stand on the validity of the actual request. It is the first line of

defense and allows the gateway to drop requests which are obviously not intended for the service.

Valid requests are passed to the Lambda function as Lambda Proxy integration requests which adds all the request details to the event object of the Lambda function. It would be possible to authorize the requests and to validate the payloads on the gateway level (AWS, 2021b). This has not been implemented for simplicity on the proof-of-concept project but would be worth investigating further if the gateway is used in production.

The API is configured to pass certain HTTP content types forward as binaries to minimize encoding issues. These content types are not transformed into text by the gateway and are available as binaries to the receiving service. The use of binary media types requires the response to the integration request to be encoded into base64 encoded string (AWS, 2021b).

6.3 AWS Lambda function

The Lambda function code was developed with Microsoft Visual Studio extended with AWS Toolkit for Visual Studio extension. The toolkit enables control over the AWS services into some extent. In development phase it makes deployment easier and faster than manually uploading the code. It would be also possible to use command line tools to achieve the same result, but this felt like more simple solution at this point.

Amazon’s toolkit introduces templates which can be used as a base for Lambda functions. The template contains the required minimum structure and code for the function. The templates are available among the other project templates in the Visual Studio. (AWS, 2021n)

The developed function is responsible of authorizing the request, validating the content, and performing the requested action. The request authorization process is discussed in detail on section 6.6. If the authorization is granted, the request content is put through validation. The validation can be done more thoroughly if required, but in this case the function just checks the contents existence to avoid issues on further processing. The requested actions depend on the request's HTTP verb. A simple switch statement delivers the request to the correct method for further processing.

AWS provides a handful of NuGet packages specifically for Lambda development on .NET Core as the AWS documentation (2021k) confirms. The packages provide basic tools for development. The basic tools include for an example Lambda logger and JSON serialization methods. There are also plenty of service specific NuGet packages available. For an example S3 is much more fluent to use with the tools of the dedicated AWS SDK package than using the HTTP API of the service.

The Lambda function is set to trigger from the API Gateway. The actual configuration can be done automatically while defining the gateway methods. The function has a separate trigger for all the gateway's HTTP methods. It is easy to have an overview on all the function's triggering events from the function configurations.

The function needs permissions to be able to use the required AWS services. The permissions are granted to a role defined in Identity and Access Management (IAM) service. The role is configured to be the execution role of the function. This way the function is able to assume necessary roles within the limits of the execution role and for an example call other services. (AWS, 2021k)

Different variable values can be stored for the function in the environment variables. In this project the values contain for an example access credentials, connection strings and service URIs. Environment variables can contain any information as a string which is required for the function and is not recommended to be hard coded into the source code. Environment variables enable also to use different versions of the function without changing the source code. The variables can be reviewed and edited on function's environment variable configuration settings. According to the AWS Lambda documentation the environment variables are encrypted in rest with AWS managed key. The values are still accessible in plain text through the management console. (AWS, 2021k)

For the function to be able to reach other services it is required to have an access through a virtual private cloud (VPC). The VPC can be configured through security groups to allow access to required AWS services and external resources. In this case the function requires outbound connection to the external authorization server. All the rules affecting the connectivity can be reviewed from the function's VPC configuration settings. (AWS, 2021k)

The functions invocations and debugging can be logged through the CloudWatch service. The service records logs for an example the function execution flow and loggings

made with the Lambda logger in the function's code. The use of the service requires the function to be implicitly allowed to record logs into the service. Without the logs it is nearly impossible to get any information about the state of the internal processing or how the function is running. (AWS, 2021k)

6.4 Simple Storage Service

The archived documents are stored in Simple Storage Service (S3) buckets as objects. The object is given a name which is also the objects key. S3 documentation (AWS, 2021f) states that key must be unique within the bucket. Lambda function gives for the new uploaded object a new GUID value as the name. The GUID standard (IETF, 2005b) guarantees the uniqueness of the value.

The object can be stored in the S3 bucket with metadata attached to it. In this case only the document type is added but the metadata could contain also other file specific information. The object contains automatically some information set by the system such as the content length. The metadata is not that significant in this case because the file specific metadata is stored into a separate database for searching and indexing purposes.

The bucket is accessed through an access point. The access point allows to set access controls through policies. In this case the bucket is blocked for public access and the only way to be able to connect to it is to get an access permission through bucket policy. The policy restricts the access to a certain VPC and thus enhances security. It also allows only certain actions and for an example prevents removal of objects.

Lifecycle management was not implemented in the thesis project, but the mechanisms required for the functionality exist. The S3 enables lifecycle management which can be set to remove documents after a certain time. There can be multiple rules affecting different kind of documents if necessary. The lifecycle management also allows to set different storage classes for documents with different storage and availability requirements. For an example some documents might need to be available for a longer time, but they can be stored on a cheaper storage type than some other documents. (AWS, 2021f)

S3 allows encryption of the stored objects. In this case the encryption was not implemented but it is recommended by the documentation. The objects can be encrypted

also before being sent to the bucket. This helps to mitigate the risk of data being leaked due to misconfiguration. (AWS, 2021f)

6.5 Relational Database Service

The metadata of the document is stored into PostgreSQL database running on AWS Relational Database Service (RDS). The data is used for searching the documents. The requirements for the database are not demanding. The most significant feature is to be able to search the data efficiently. In this case the selection was made based on the customer's existing systems and experience on the database.

The database content was administered with pgAdmin which is an open source administration platform for PostgreSQL (pgAdmin Development Team, 2021). To be able to connect to the database from outside of the AWS ecosystem the connection has to be specifically allowed by the VPC security group (AWS, 2021e). The security group assigned to the database instance must have a rule to allow the connection from the client.

The database structure is very simple and has not been optimized in any way in this project. The database has only one table with one row per document. The table has one column per each piece of metadata. The document's object key is stored as the primary key of the table.

The database can handle a fairly large amount of document objects without any optimization. With a larger amount of documents, the database design and usage should be reconsidered to ensure the performance. The optimization might be achieved for an example through indexing and more efficient search statements (The PostgreSQL Global Development Group, 2021).

The removal of expired document's metadata was not implemented to the project but can be accomplished with AWS tools. The maintenance functions can be executed for an example by invoking a Lambda function with S3 object removal event or by running a scheduled function.

6.6 Authorization Server

The authorization server was intended to be a Keycloak server to ensure the compatibility to the customer's existing system. The keycloak server would have required setting up a separate virtual server to the AWS or to an external server and configuring the service from scratch. For this reason, the server was replaced with Okta web service which offers authentication as a service (Okta, 2021). The Okta service could emulate the Keycloak server close enough to be used as a replacement. The Okta service forced the project to follow the OpenID Connect standard on authorization but otherwise there were no major changes.

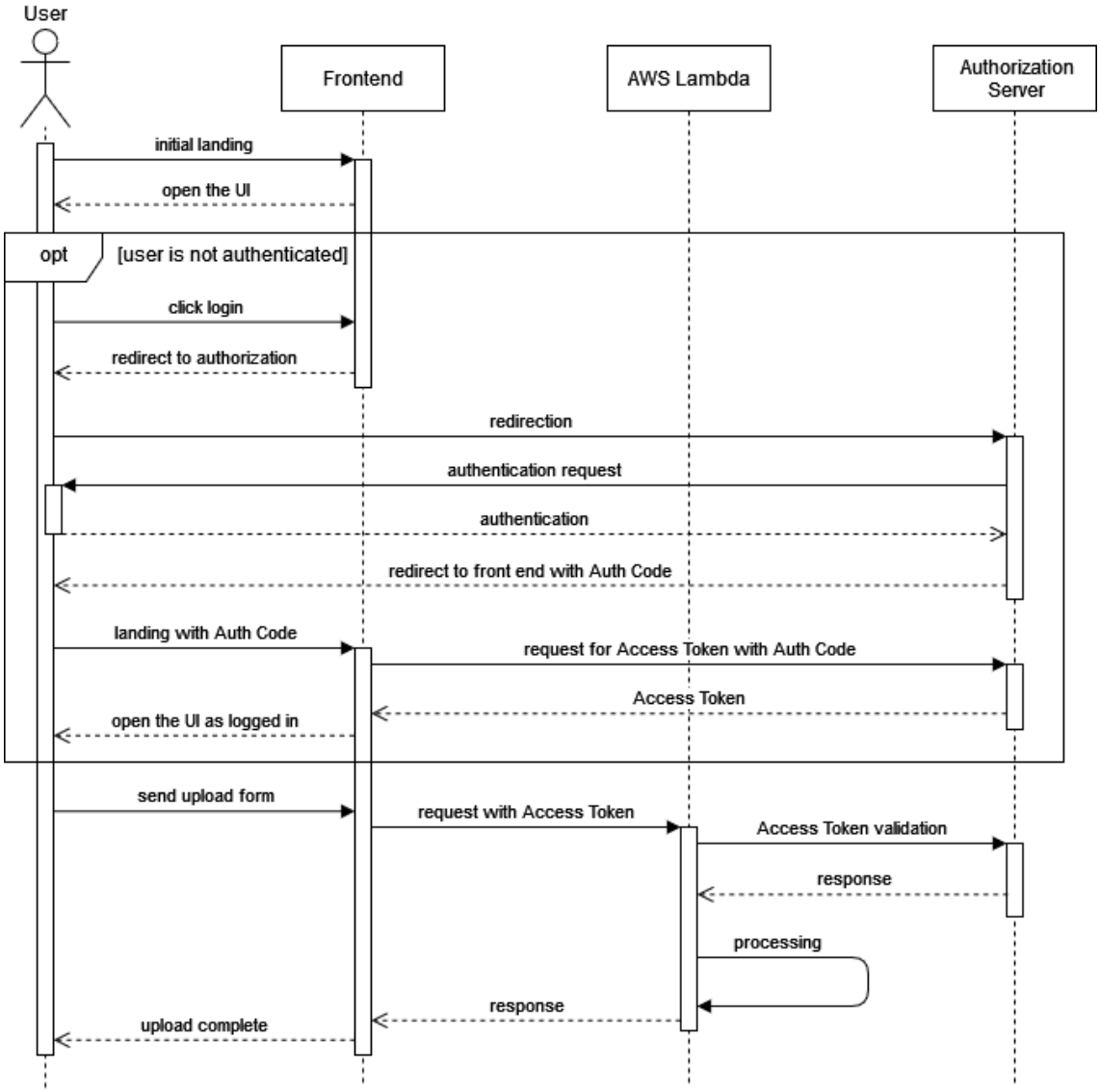


Figure 5. Authorization sequence on document upload.

The document archive does not care who the user is. The only concern the archive has is that the user is authorized to access. The authorization is implemented using methods described in section 5. The user must be registered to the Okta service and granted with an access for different archive operations. The permissions are granted by adding the user to different access groups.

The authorization process is initiated on the frontend when the unauthenticated user lands on the web page. The user is redirected to the authorization server for authentication. The user returns with authentication code and is able to perform requested actions if the frontend service is able to fetch and verify the authorization of the user. The authorization sequence is depicted in Figure 5.

In this project the authorization information of the user is stored in the cookie of the user for development reasons. The cookie is a relatively unsafe method of storing sensitive information and it should be implemented in other ways in production version. One viable option is to implement the zero trust principle and to verify every request from the authentication server (Rose, et al., 2020). This removes the need of storing the authorization information but increases the number of requests to the authorization server.

7 CONCLUSION

The thesis project fulfilled the requirements set to the project and successfully created a proof-of-concept solution of the document archive. The product was functional and contained the necessary features. Some of the properties were not implemented in practice due to time limits but the implementations were planned in theory, and they should be executable if the process continues.

There were no major issues during the development. The greatest problem during the process was an encoding issue with the PDF document during the Lambda function processing. The problem was found and fixed with changing how the document binaries are processed in the function. Another time-consuming issue was handling the network traffic in the AWS VPC during the database development phase. Simultaneous traffic from external network to configure the database and from the internal VPC to access the data was complicated to configure with the minimal resource usage. The issue was solved by alternating manually the security policies depending on the current usage profile.

If the project is continued further, here are some development suggestions and ideas. The Lambda function might be beneficial to be separated into dedicated functions for the upload and download operations. Currently they both run on the same function and the function is more complicated than necessary. Especially if the usage is relatively constant, the separate functions could deliver some performance gain and the resource usage could be more accurately controlled.

The deployment process should be made automatic by using the Serverless Application Model (SAM). The SAM provides a framework for modelling and building applications using YAML syntax. The SAM defines the application, and it can be used during deployment to create the services automatically according to the model. (AWS, 2021m)

An alternative implementation idea came up during the demonstration of the solution. The service could be implemented also by uploading the document directly to the S3 bucket while the metadata would be processed with Lambda function triggered by the S3 upload. In this solution, the file archiving process is more straight forward but the data validation and user authorization responsibility are completely on the frontend. The user cannot get immediate response with success or fail message of storing the document.

REFERENCES

Alexa Developers, 2019. *Adam Savage: Forget About "Fail Fast"—the Future is "Iterate Fast".* s.l.:YouTube.

AWS, 2021a. *About AWS.* [Online]
Available at: https://aws.amazon.com/about-aws/
[Accessed 10 September 2021].

AWS, 2021b. *Amazon API Gateway Developer Guide.* [Online]
Available at:
https://docs.aws.amazon.com/apigateway/latest/developerguide/apigateway-dg.pdf
[Accessed 7 September 2021].

AWS, 2021c. *Amazon API Gateway pricing.* [Online]
Available at: https://aws.amazon.com/api-gateway/pricing/
[Accessed 15 September 2021].

AWS, 2021d. *Amazon RDS for PostgreSQL Pricing.* [Online]
Available at: https://aws.amazon.com/rds/postgresql/pricing/
[Accessed 15 September 2021].

AWS, 2021e. *Amazon Relational Database Service User Guide.* [Online]
Available at: https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/rds-ug.pdf
[Accessed 1 July 2021].

AWS, 2021f. *Amazon Simple Storage Service User Guide.* [Online]
Available at: https://docs.aws.amazon.com/AmazonS3/latest/userguide/s3-userguide.pdf
[Accessed 1 July 2021].

AWS, 2021g. *Amazon Virtual Private Cloud User Guide.* [Online]
Available at: https://docs.aws.amazon.com/vpc/latest/userguide/vpc-ug.pdf
[Accessed 13 September 2021].

AWS, 2021h. *AWS Fundamentals Overview.* [Online]
Available at: https://aws.amazon.com/getting-started/fundamentals-overview/
[Accessed 15 September 2021].

AWS, 2021i. *AWS General Reference.* [Online]
Available at: https://docs.aws.amazon.com/general/latest/gr/aws-general.pdf
[Accessed 17 September 2021].

AWS, 2021j. *AWS Identity and Access Management Documentation.* [Online]
Available at: https://docs.aws.amazon.com/IAM/latest/UserGuide/iam-ug.pdf
[Accessed 15 September 2021].

AWS, 2021k. *AWS Lambda Developer Guide.* [Online]
Available at: https://docs.aws.amazon.com/lambda/latest/dg/lambda-dg.pdf
[Accessed 25 5 2021].

AWS, 2021l. *AWS Security Documentation.* [Online]
Available at: https://docs.aws.amazon.com/security/
[Accessed 7 September 2021].

AWS, 2021m. *AWS Serverless Application Model.* [Online]
Available at: https://aws.amazon.com/serverless/sam/
[Accessed 26 September 2021].

AWS, 2021n. *AWS Toolkit for Visual Studio.* [Online]
Available at: https://docs.aws.amazon.com/toolkit-for-visual-studio/latest/user-guide/aws-tkv-ug.pdf
[Accessed 26 September 2021].

AWS, 2021o. *AWS Well-Architected.* [Online]
Available at: https://aws.amazon.com/architecture/well-architected
[Accessed 7 September 2021].

AWS, 2021p. *Best Practices for Security, Identity, & Compliance.* [Online]
Available at: https://aws.amazon.com/architecture/security-identity-compliance
[Accessed 7 September 2021].

AWS, 2021q. *Global Infrastructure.* [Online]
Available at: https://aws.amazon.com/about-aws/global-infrastructure
[Accessed 13 September 2021].

AWS, 2021r. *Security Learning.* [Online]
Available at: https://aws.amazon.com/security/security-learning
[Accessed 7 September 2021].

AWS, 2021s. *Shared Responsibility Model.* [Online]
Available at: https://aws.amazon.com/compliance/shared-responsibility-model/
[Accessed 7 September 2021].

AWS, 2021t. *Tools to Build on AWS.* [Online]
Available at: https://aws.amazon.com/tools/?e=gs2020&p=fundoverview&p=gsrc&c=fo
[Accessed 15 September 2021].

Ecma International, 2017. *ECMA-404 The JSON data interchange syntax.* [Online]
Available at: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf
[Accessed 22 September 2021].

Fielding, R. T., 2000. *Architectural Styles and the Design of Network-based Software Architectures.* Irvine(California): University of California.

Gartner, 2021. *Definition of SDK (Software Development Kit).* [Online]
Available at: https://www.gartner.com/en/information-technology/glossary/sdk-software-development-kit
[Accessed 13 September 2021].

IANA, 2021a. *About us.* [Online]
Available at: https://www.iana.org/about
[Accessed 22 September 2021].

IANA, 2021b. *JSON Web Token (JWT).* [Online]
Available at: https://www.iana.org/assignments/jwt/jwt.xhtml
[Accessed 22 September 2021].

IETF, 1987. *Request for Comments: 1034.* [Online]
Available at: https://datatracker.ietf.org/doc/rfc1034/
[Accessed 17 September 2021].

IETF, 1996. *Request for Comments: 1945.* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc1945
[Accessed 9 September 2021].

IETF, 2005a. *Request for Comments: 3986.* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc3986
[Accessed 17 September 2021].

IETF, 2005b. *Request for Comments: 4122.* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc4122
[Accessed 17 September 2021].

IETF, 2012a. *Request for Comments: 6749.* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc6749
[Accessed 15 August 2021].

IETF, 2012b. *Request for Comments: 6750.* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc6750
[Accessed 15 August 2021].

IETF, 2015a. *Request for Comments: 7519.* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc7519
[Accessed 20 September 2021].

IETF, 2015b. *Request for Comments: 7636.* [Online]
Available at: https://datatracker.ietf.org/doc/html/rfc7636
[Accessed 17 September 2021].

ISO, 2005. *ISO/IEC 19501:2005.* [Online]
Available at: https://www.iso.org/standard/32620.html
[Accessed 22 September 2021].

Microsoft, 2020a. *.NET documentation.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/
[Accessed 15 August 2021].

Microsoft, 2020b. *.NET Framework documentation.* [Online]
Available at: https://docs.microsoft.com/en-us/dotnet/framework/
[Accessed 15 August 2021].

Microsoft, 2021a. *Overview of ASP.NET Core authentication.* [Online]
Available at: https://docs.microsoft.com/en-us/aspnet/core/security/authentication/
[Accessed 23 September 2021].

Microsoft, 2021b. *What is .NET?.* [Online]
Available at: https://dotnet.microsoft.com/learn/dotnet/what-is-dotnet
[Accessed 25 September 2021].

Okta, 2021. *Okta Developer Portal.* [Online]
Available at: https://developer.okta.com/
[Accessed 23 September 2021].

OpenID Foundation, 2021a. *OpenID Connect Core 1.0.* [Online]
Available at: https://openid.net/specs/openid-connect-core-1_0.html
[Accessed 15 August 2021].

OpenID Foundation, 2021b. *OpenID Foundation.* [Online]
Available at: https://openid.net/foundation/
[Accessed 22 September 2021].

pgAdmin Development Team, 2021. *pgAdmin 4 5.7 documentation.* [Online]
Available at: https://ftp.postgresql.org/pub/pgadmin/pgadmin4/v5.7/docs/pgadmin4-5.7.pdf
[Accessed 26 September 2021].

Rose, S., Borchert, O., Mitchell, S. & Connelly, S., 2020. *Zero Trust Architecture,* s.l.: National Institute of Standards and Technology (NIST).

Saltzer, J. H. & Kaashoek, F. M., 2009. *Principles of Computer System Design: An Introduction Part II.* Version 5.0 ed. Online: Massachusetts Institute of Technology.

The PostgreSQL Global Development Group, 2021. *PostgreSQL: Documentation.* [Online]
Available at: https://www.postgresql.org/files/documentation/pdf/13/postgresql-13-A4.pdf
[Accessed 26 September 2021].