VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

SMART TAGGING SYSTEM FOR DIVING EQUIPMENT

Information Technology
2016

ACKNOWLEDGMENTS

I would like to begin by expressing my gratitude to my thesis supervisor, Timo Kankaanpää, for his guidance throughout this thesis. I would also like to thank Tommi Rintala and Antti Backman, the client supervisors, for their support during the project.

In addition, I would like to take this opportunity to thank all the teachers and staff at Vaasa University of Applied Sciences(VAMK) for their help during my studies.

The completion of this thesis and my degree program in VAMK would not have been possible without the continuous support of my family. I am very grateful and lucky to have them.

VAASA UNIVERSITY OF APPLIED SCIENCES
Degree Programme of Information Technology

ABSTRACT

Author	Rohullah Ayoub
Title	Smart Tagging System for Diving Equipment
Year	2016
Language	English
Pages	70
Name of Supervisor	Timo Kankaanpää

The use of Near Field Communication (NFC) has revolutionized many industries through digitalization. This process of digital immersion has been further accelerated through the mainstream availability of NFC-enabled devices and the substantial decline in the cost of NFC smart tags.

The purpose of this thesis was to design and implement an end-to-end, smart tagging solution for diving equipment. The project involved an Android application, an AngularJS web application and the back-end was developed using Amazon Web Services (AWS). A server-less architecture using AWS micro services was employed in the project.

The Android application is used to register NFC tags by writing and reading data from NFC tags and communicating with the backend through a RESTful API. The AngularJS application provides access to the corresponding data. In addition, user authentication is achieved by using Google as an Identity Provider (IdP).

This document provides an overview of the steps necessary to implement and integrate applications running on different platforms with AWS services, in a cost-effective and scalable manner. Even though this document addresses topics relevant to a specific project, most of the implementation and design instructions can be used to serve other use-cases, particularly by startups.

Since the project involves applications developed on different platforms, only the most important aspects of the process are presented throughout this document.

Keywords NFC, Android, AWS, Identity Provider, AngularJS

VAASAN AMMATTIKORKEAKOULU
Tietoteknikaan koulutusohjelma

TIIVISTELMÄ

Tekijä	Rohullah Ayoub
Opinnäytetyön nimi	Sukellusvarusteiden Älykäs Merkintäjärjestelmä
Vuosi	2016
Kieli	Englanti
Sivumäärä	70
Ohjaaja	Timo Kankaanpää

Lyhyen kantaman tiedonsiirron (NFC:n) käyttö on mullistanut monia teollisuuden aloja digitalisoinnin kautta. Näiden digitaalisen upotuksien prosessi on kiihtynyt entisestään, NFC yhteensopivien laitteiden ja saatavuuden noustessa. Prosessi myös supistaa toimintotunnisteiden kustannuksia merkittävästi.

Tämän opinnäytetyön tarkoituksena on suunnitella ja toteuttaa päästä päähän toimintotunnisteratkaisu ajovarusteisiin. Projektiin sisältyy Android sovellus, AngularJS web sovellus ja back end on kehitetty käyttäen Amazon Web Serviceä (AWS). AWS micro palveluja käytetään projektissa palvelimettoman arkkitehtuurin avulla.

Android sovellusta käytetään NFC-tunnisteien rekisteröimiseen dataa kirjoittamalla ja lukemalla niitä NFC-tunnisteesta sekä kommunikoimalla back endiin RESTfulAPI:n kautta. AngularJS sovellus tarjoaa pääsyn vastaavaan tietoon. Lisäksi käyttäjän todennus saavutetaan käyttämällä Googlen Identity Provideria (idP).

Tässä dokumentissa on yleiskatsaus tarvittavista toimenpiteistä, joilla toteutus ja integrointi pystytään tekemään, eri alustoilla käynnissä olevilla prosesseilla AWS palveluissa.kustannustehokkaasti ja mitattavissa olevilla tasoilla. Vaikka tässä asiakirjassa käsitellään tiettyyn projektiin liittyviä aiheita, useimpia toteutus- ja suunnitteluohjeita voidaan myös soveltaa muihin käyttötarkoituksiin, erityisesti startup ideoille.

Koska projekti sisältää sovelluksia, jotka on kehitetty eri alustoille, ainoastaan tärkeimmät prosessin näkökohdat on esitetty dokumentissa.

Avainsanat NFC, Android, AWS, Identity Provideria, AngularJS

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF ABBREVIATIONS

API	Application Programming Interface
AWS	Amazon Web Services
CORS	Cross-Origin Resource Sharing
CRUD	Create, Read, Update, Delete
CSS	Cascading Stylesheet
DOM	Document Object Model
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IAM	Identity Access Management
NDEF	NFC Data Exchange Format
NFC	Near Field Communication
REST	Representational State Transfer
RFID	Radio Frequency Identification
SDK	Software Development Kit
SOA	Service Oriented Architecture
S3	Simple Storage Service
IdP	Identity Provider
URL	Uniform Resource Locator

1 INTRODUCTION

This document provides an overview of developing a smart tagging system using NFC enabled devices and NFC tags. The solution was built for Delektre Ltd and consists of an Android application, an AngularJS web application and Amazon Web Services.

1.1 Client Organization

Delektre Ltd is a Finnish product development company located in Vaasa. Founded in 2010, Delektre offers R&D and consultancy services to both institutions and business clients.

In addition to aforementioned services, Delektre is involved in the development of several of their own products such as REBEL Ring, an activity tracker worn as a ring.

1.2 Current State

Presently, scuba diving equipment is mostly marked using felt pens. Using felt pens to mark equipment poses several problems such as the markings wearing down through contact with water for an extended period of time, difficulty changing the markings and only allowing a small amount of data to be written on the equipment.

In some cases, tapes are used to mark the equipment. This makes it easier to change the markings but it does not address the other issues present.

Since scuba diving equipment needs maintenance and inspections throughout the year, the lack of an intuitive centralized system makes it difficult for the equipment shops to keep track of all the equipment.

1.3 Project Objectives

Delektre would like to offer a complete solution, addressing the issues briefly mentioned in the previous section and providing additional functionality that streamlines the experience for both the equipment shops and the equipment owners.

The solution in its initial stage will make use of:

- Near Field Communication (NFC) tags: These are used to mark the equipment with the basic information necessary for identification, inspection and maintenance
- Mobile application: This will be used to register new tags to the system and read existing tags
- Web application: This will allow the management of tags, provide reporting and additional details
- Amazon Web Services (AWS): Various services offered by AWS will be used to implement the back-end for the solution

The scope of the thesis is to develop the initial version of this system of tag registration. The thesis will cover the mobile application, web application and the REST services needed to support both the mobile and web applications.

1.4 Author’s role

The author of this document is responsible for researching possible solutions to achieve the aforementioned objectives. The author is also responsible for designing the architecture used in the project, implementing the corresponding applications on all platforms, testing and documenting the process.

2 TECHNOLOGIES

This chapter briefly describes the technologies used throughout the development of this project.

The mobile application is built using the Android application framework. Since Near Field Communication is a necessary feature, the corresponding NFC libraries provided by the Android framework are also used.

The backend of the solution is implemented using Amazon Web Services. This involves, among other steps, deploying the API using the API Gateway, running the server-side code in Lambda and using DynamoDB as a database.

The web application is developed using AngularJS. In addition, AngularUI libraries alongside Bootstrap are used to provide a rich User Interface.

2.1 Near Field Communication

As the name implies, Near Field Communication (NFC) enables a device to interact wirelessly with another compatible device over a short range. This range is usually a radius of approximately 10 cm. /8/

A full NFC device can operate in the following modes:

- Reader/writer: allows the device to read/write information to NFC tags
- Card emulation: the device acts as smartcard, offering features such as contactless payments
- Peer-to-Peer: enables the device to communicate and exchange data with another compatible NFC device

There is a wide variety of NFC-enabled devices available today. However, they can generally be divided into two groups: active and passive.

2.1.1 Active NFC devices

These devices can send and receive data from any NFC device. An NFC enabled smartphone, for example, is able to not only communicate with other compatible devices, but to read information stored in NFC tags or even modify that information, if allowed.

2.1.2 Passive NFC devices

These NFC devices do not possess their own power-source and they receive power during communication with an Active NFC device. Therefore, these devices are able to communicate only with Active NFC devices and cannot process any information themselves. Passive NFC devices include small transmitters like NFC tags. Figure 1 illustrates the communication between an active and passive device.

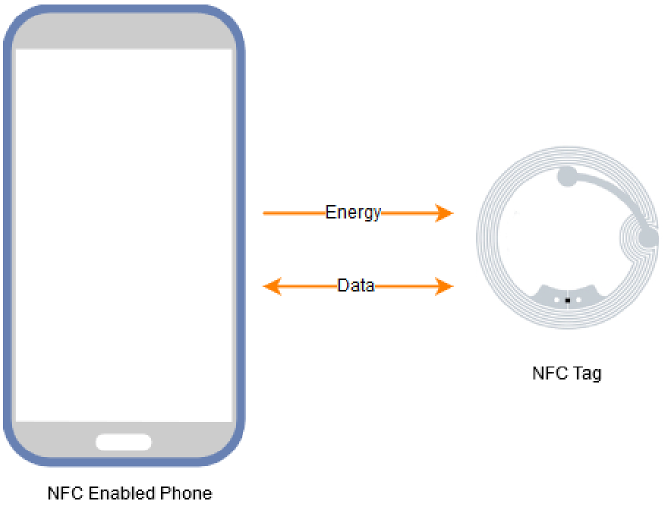


Figure 1: NFC tag communication with a phone

The effective range of a passive NFC device is usually 10-15cm and depends on the type, design and quality of the device, as well as the material used for the antenna. This material is typically silver or copper.

2.1.3 RFID vs NFC

Even though the means of communication for both Radio Frequency Identification (RFID) and NFC are radio signals, RFID is more ubiquitous today when compared to NFC. /10/

An RFID communication involves an RFID tag that stores information, and an RFID reader that reads that information. This may seem very similar to the communication between an NFC tag and an NFC device. However, there is a major difference here since the communication in RFID is one way whilst the Peer-to-Peer mode in NFC allows devices to exchange data in both directions. /10/

In addition, RFID communication may work well up to several meters compared to 10 cm of NFC.

2.1.4 NDEF

Created by the NFC forum, NFC Data Exchange Format (NDEF) is a standard format that defines encoding an NFC tag and communication between two NFC devices. /9/

Perhaps one of the most widely supported standard by the NFC industry, the NDEF format comprises of NDEF Messages and NDEF Records. Each NDEF Message can contain several NDEF Records.

2.1.4.1 NDEF Record Layout

Even though records in NDEF format may have variable length, they follow a specific format that provides insight into the type and content of the record. Figure 2 shows the common fields of NDEF Records followed by a brief explanation of each field.

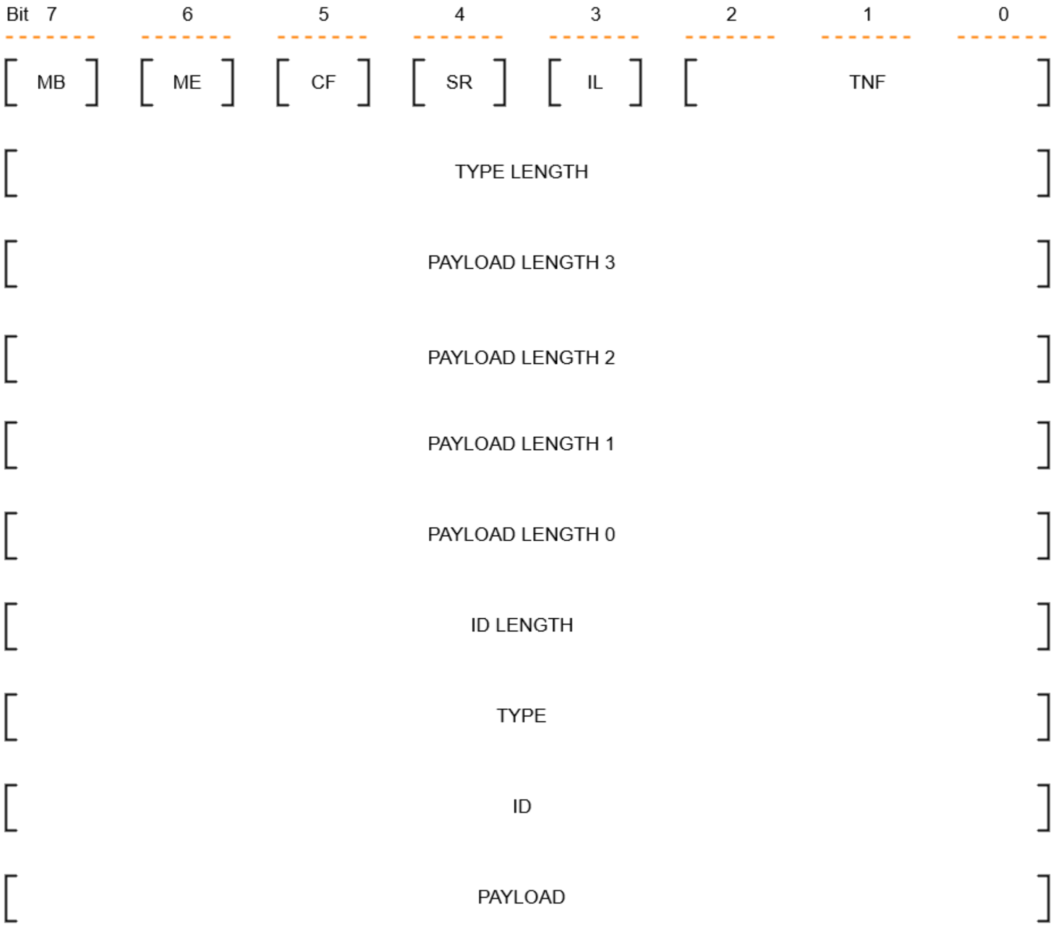


Figure 2: NDEF Record. Adapted from /9/

- Header
  - Type Name Format (TNF): This three-bit value defines the type of records, giving insight into the rest of content and structure.
  - ID Length (IL): Indicates whether the ID Length field is present. 0, if not present.
  - Short Record (SR): If set, indicates that the payload length field is 1 byte or less.
  - Chunk Flag (CF): Indicates whether this is the first or middle chunk of a chunked payload.
  - Message End (ME): If set, indicates that this is the last record in the message.
  - Message Begin (MB): Indicates whether this is the beginning of the message.

- Type Length: The length of the type field is specified here in bytes.
- Payload Length: The length of the payload in bytes is indicated here. The size of this field is dependent upon the Short Record (SR) bit above. If SR is 0, the payload length field will be four bytes long. Otherwise, it will be one byte long.
- ID Length: This field specifies the length of the ID and is present if the ID Length field in the record header is set to 1.
- Record Type: Corresponding with the value of the TNF field in the record header. This field indicates the exact type of the record
- Record ID: Present only if the Indicates the IL field in the record is set to 1, this field indicates the ID value.
- Payload: The actual payload intended to be used. The size of this record is exactly the amount specified in the Payload Length field, in bytes.

2.2 Java

Developed at Sun Microsystems in California and released in 1995, Java is a programming language based on C/C++. Initially, Java was aimed at making programs for consumer appliances such as microwave ovens. However, due to its use in writing applets, applications running in the browser, Java was soon widely adopted as a web programming language. /1/

Today, thanks to its write once, run anywhere (WORA), Java is used to develop standalone applications on a variety of platforms and has become one of the most popular programming languages in the world.

2.2.1 Android Application Framework

Composed of a set of Java libraries, the Android application framework provides the components and tools needed to develop mobile applications for Android handheld devices, defining how applications should work. /22/

Android Application Framework wraps the lower level libraries in the Android Operating System. This allows the developers to utilize the low-level libraries that are written in C/C++, by using Java code and without the need to know how each of those low-level libraries work.

It is possible to develop applications for Android OS without using the application framework, through direct communication with the low-level C/C++ libraries. However, that is beyond the scope of this thesis.

2.2.1.1 Android NFC

Even though Android supports other NFC standards, most of the functionality is aimed towards tags using the NDEF standard. When working with NDEF, Android supports:

- Reading data from an NFC tag
- "Beaming" data from one device onto another

When an NFC tag is detected, the Tag Dispatch System is responsible for analyzing, categorizing and determining the appropriate application to launch, from a list of applications that have registered to be expecting NFC communication. The Tag Dispatch System does this by analyzing the first data record in the tag and trying to map it to a known type. /7, 22/

After the dispatch system has finished analyzing the tag, one of the following three intents, in the order of their priority level, are created, and based on the result of the analysis:

- ACTION_NDEF_DISCOVERED: Intent for when the Tag is identified to be in the NDEF format. This has the highest priority and an application filtering for this intent will be launched.
- ACTION _TECH_DISCOVERED: Intent for when the Tag is identified to be in the NDEF format but no activity is found filtering for the previous intent. This intent is also created when the Tag is not using the NDEF format but the technology type is known.
- ACTION_TECH_DISCOVERED: Created when no activity is found that handles any of the above intents.

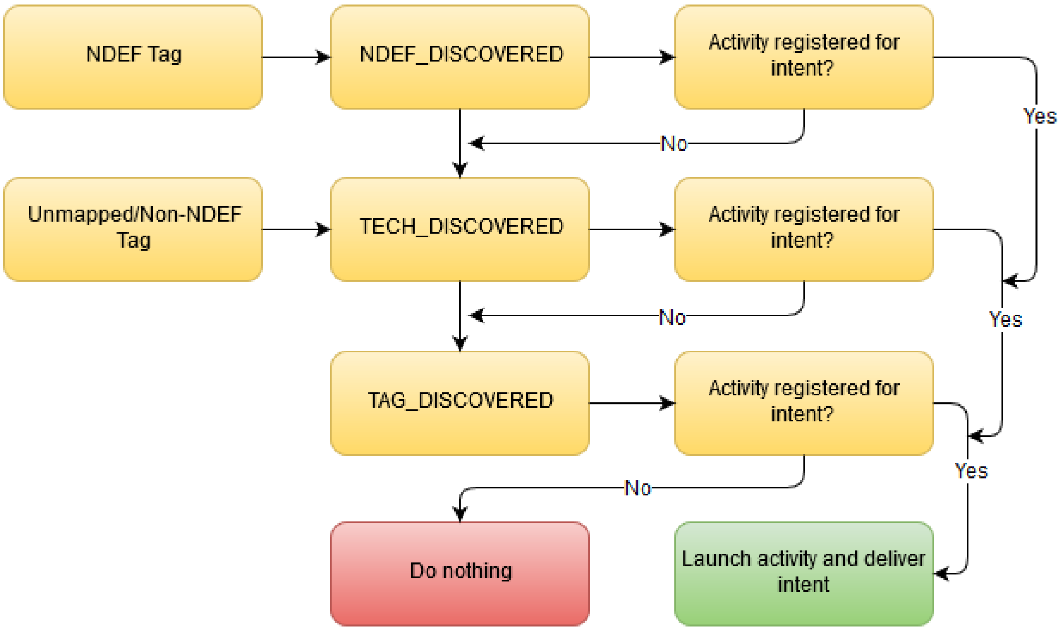

NDEF Tag
NDEF_DISCOVERED
Activity registered for intent?
Yes
No
Unmapped/Non-NDEF Tag
TECH_DISCOVERED
Activity registered for intent?
No
Yes
TAG_DISCOVERED
Activity registered for intent?
Yes
No
Do nothing
Launch activity and deliver intent

Figure 3: Android NFC intents. Adapted from /7/

In Android, an intent object is used to deliver information within components. Using an intent, the Tag Dispatch System delivers the tag information to the corresponding application.

Tag Dispatch System allows the appropriate applications to be launched without the need for the user to select applications through a dialog. This ensures a seamless experience by reducing the amount of user interaction.

2.2.1.2 Material design

Offered by Google and available in Android 5.0+, Material design for Android is a set of design guidelines and interactive components to be used when developing mobile applications for Android devices./26/. This includes, but is not limited to:

- Comprehensive design specification
- Material theme
- Widgets
- Animations

Using Material design accelerates the development of Android applications while retaining a good User Experience.

2.2.1.3 OpenNFC emulator

OpenNFC is an open-source implementation, consisting of APIs to provide NFC capabilities to different Operating Systems. As part of their product catalog, they offer an Android emulator which, together with their desktop application, provides basic NFC functionality including reading and writing to a variety of virtual NFC tags.

2.2.2 Gradle

Gradle is a build automation system for Java. Even though they are over-rideable, Gradle builds are written using a set of conventions that allow developers to understand each other’s builds.

In contrast to Maven and Ant, the other two popular build automation systems, Gradle uses a “declarative build language”, focusing on what needs to be built rather than how it needs to be built. In addition, the build language for Gradle is a Domain Specific Language (DSL) called “Groovy”. Therefore, unlike other build automation systems, configuring Gradle builds may be easier to achieve since it uses a language that is specific to the build.

Furthermore, Gradle supports dependencies and multi-project builds. Therefore, if an application requires many different projects to be built, Gradle allows that out of the box.

2.2.3 JUnit

Created by Kent Beck and Enrich Gamma, JUnit is a unit testing framework for Java. /5/

JUnit follows the xUnit architecture, which provides a model for the structure and functionality of unit testing frameworks, and allows to write and to run repeatable tests. /5/. The main features of JUnit are detailed in Table 1.

Table 1: Main features of JUnit. Adapted from /5/

Feature	Role	Explanation
Assertion	Tests expected results	Verifies whether the result of a function, with given parameters, matches against a predefined, expected value
Text fixture	Share data between tests	Ensures that there is a known, fixed environment for the repeatable tests. Comprises of a fixed state of a particular set of objects, to be used as a base for the test.
Test runners	Run the tests	Allows running tests and displaying the results. IDEs tend to have graphical test runners included.

2.3 JavaScript

Initially named “Mocha” and later renamed to JavaScript, this interpreted language was developed at NetScape Communications Corporation in 1995. It is standardized through EcmaScript specification.

JavaScript is a weakly typed language with support for object oriented programming. However, its approach to OOP is in contrast to the class-based programming style found in popular languages such as Java and C++. JavaScript does not possess a concept of classes, but rather prototypes. This is known as prototype-based programming. /6/

Even though JavaScript has, for the most part, been used for client-side programming in the web, it has also been employed in developing hybrid mobile applications and in server-side programming. In fact, the rise of JavaScript on the server has introduced an entirely new software stack comprising of MongoDB, Express, AngularJS and NodeJS. This is more commonly referred to as the MEAN stack and will be discussed in the upcoming sections.

2.3.1 NodeJS

First introduced in 2009 by its creator, Ryan Dahl, NodeJS is an open-source, server-side, runtime environment that uses JavaScript as its language. Bringing JavaScript to the server side, NodeJS is truly cross platform. Therefore, JavaScript code running inside NodeJS can be deployed to Microsoft Windows, Linux and OSX.

NodeJS is event-driven. Default design pattern of JavaScript in NodeJS is modular and it encourages writing asynchronous, non-blocking code. Consequently, it makes use of callback mechanism when performing most tasks. This is perhaps one of the biggest distinctions to be made between NodeJS and other more traditional web servers like IIS.

In addition to being asynchronous by default, NodeJS standard libraries are also low level. Therefore, it might not provide a lot of “helpers” to make the code easy and fast to write, out of the box. However, it offers good building blocks for higher level implementations that are provided by the community and available through the Node Package Manager (NPM). /12/

2.3.1.1 Express

Express is a lightweight and minimalist web development framework for NodeJS. Being un-opinionated, Express provides a flexible solution to handle tasks like routing and rendering.

Essentially a package in the Node Package Manager (NPM), one of the key features of Express is that it allows “middleware” when responding to HTTP requests. These functions have access to the request/response object and the next middleware function in the request/response cycle. Using middleware functions, the following tasks can be performed:

- Modify the request/response object
- Execute code
- End the cycle
- Call the next middleware

The minimalist approach of Express framework coupled with the middleware and routing it provides, offers a good stack for developing an Application Programming Interface (API). /13/

2.3.1.2 Gulp

Gulp is a development task runner. In other words, it allows developers to automate the repetitive tasks of the development process including but not limited to running unit tests, refreshing browser on file save and minifying source code.

A common Gulp workflow could be described in the following steps:

1. A new task is defined
2. Files needed for accomplishing that task are loaded into the Gulp stream
3. Any necessary modifications to the files are made
4. The modified files are saved to the stated destination.

Gulp is stream-based. This means that a write to the file system is only required when Gulp is finished with all modifications. This is in contrast to Grunt, another popular task runner, that requires writes to temporary folders between modifications. /14/

2.3.2 AngularJS

Largely maintained by Google, AngularJS is an open-source, MVC and opinionated JavaScript framework for creating web applications, specifically Single Page Applications (SPA). /2/

Angular allows the client to send and receive data from the backend, by handling the Ajax communication with the server. It also handles the presentation of the data on the page through partial templates or manipulation of the existing DOM. The view and model are then kept in sync using two-way binding.

Perhaps one of the most unique characteristics of AngularJS is that it extends HTML by providing its own elements and properties. These are referred to as “directives” and are used to interact with the HTML DOM. In addition, testability is an important aspect of AngularJS and it was a primary consideration behind the project. It supports both isolated unit tests and integrated end to end tests.

2.3.2.1 AngularUI

Cohesively packaged together, AngularUI consists of multiple UI libraries for AngularJS. Ranging from robust, stand-alone modules to small utility tools. /18/

AngularUI Bootstrap offers components written in pure AngularJS. This eliminates the need to add jQuery as a dependency, when using Bootstrap. In addition, the UI-Router module offers a more flexible and powerful alternative to the AngularJS ngRoute module.

2.4 Amazon Web Services

Offered by Amazon, Amazon Web Services includes a set of IT infrastructure services, offered to individuals and organizations. These web services, also known as Cloud services, provide the capability to *rent* compute resources on-demand. /16/

Typically considered Infrastructure-as-a-Service (IaaS), AWS offers Platform-as-a-Service (PaaS) functionalities as well. AWS has a global footprint. Available in 190 countries, AWS are offered through 12 Regions and 32 Availability Zones. /19/

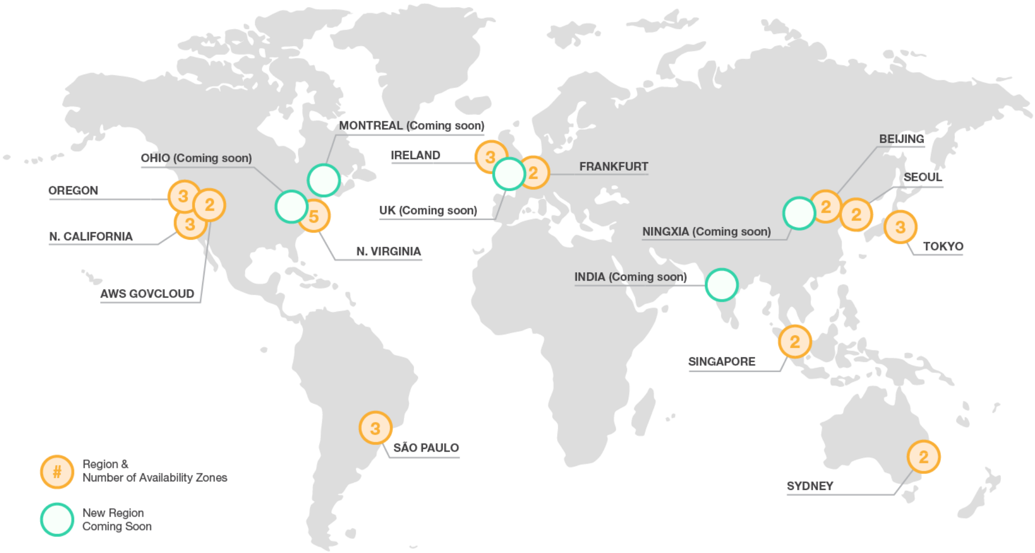


Figure 4: AWS Regions and Availability Zones. Adapted from /16/

AWS Regions are geographic locations where the AWS services can be procured and are completely independent from each other. This results in a high fault-tolerance. For example, if an application is hosted through multiple AWS Regions, it will be available even if an entire region goes down.

A combination of one or more data centers, AWS Availability Zones are segmented locations within a region and are physically isolated from other availability zones. There is, however, a low-latency connection between availability zones within a region. This allows, for example, the rapid provisioning of a server within a different availability zone, when an availability zone fails.

In the following sections, some of the AWS services relevant to this project will be discussed in more detail.

2.4.1 Identity Access Management

Also known as IAM, Identity Access Management is a service provided by AWS to manage access to AWS resources. /20/

IAM addresses two key issues:

- Authentication: Depending on the use-case, both long-standing and temporary credentials can be generated to determine the identity of the requesting source.
- Authorization: Based on the identity of the requesting source, access to specific actions on specific resources is granted.

2.4.1.1 Permissions

Permissions are used to define and configure access to AWS resources. AWS supports assigning permissions in the following two ways:

- Identity-Based: When permissions are assigned to an identity such as User, Group or Role. This involves *attaching* a policy to the identity.
- Resource-Based: When permissions are assigned to an AWS resource. This involves attaching a policy to the resource.

2.4.1.2 Policies

Policies are used to configure and assign permissions to an IAM User, Group or Role. An AWS Policy consists of the following basic components:

- Actions: The actions that are allowed in the current policy are listed here. All other actions possible on the resource and not explicitly mentioned, are denied. Wildcards can also be used to define Actions.
- Resources: The resources that the actions are allowed for, are specified here. Again, permission is granted for only the resources explicitly mentioned here and wildcards can be used to define Resources.
- Effect: This can either be “Allow” or “Deny”. If “Allow” is specified, the requesting user will be given permissions based on the Actions and Resources mentioned in the Policy. Otherwise, the request will be denied.

There are currently two methods of defining policies:

- Managed: Policies that can be attached to multiple Users, Groups or Roles. In order to ease the process of authorization, AWS offers a large collection of managed policies for a variety of use cases. However, the users can create policies themselves, for more control over the allowed resources and actions.

- Inline: Policies that are attached to a single User, Group or Role. Usually, this method is used for Resource-Based Permissions.

2.4.1.3 Identities

An IAM Identity provides access to AWS resources to a person or process. This can involve long-standing credentials to be used by a specific Identity or generating temporary credentials on-demand. IAM identities include:

- User: A person or service that interacts with AWS through AWS console or AWS CLI. An IAM User is usually created to share access within an AWS account. Every IAM User has its own AWS Credentials.
- Groups: A collection of users, IAM Groups allow specifying authorization to AWS resources for a collection of users. An IAM Group is usually used to form development teams, based on a project and the required AWS resources for that project.
- Roles: Similar to an IAM User. However, IAM Roles are not attached to a specific entity and make use of *Temporary Credentials*. IAM Roles are generally used to provide temporary access to AWS resources, based on the IAM Policy attached to it. A sample use case for IAM Roles is user login through Identity Federation, instead of IAM.

2.4.2 Lambda

AWS offers Lambda as a compute service that in response to the defined events, runs the provided code and automatically manages the compute resources. /21/

When using AWS Lambda, the two primary components are the *Lambda function* and the *Event Source.* Lambda function is the application code that is uploaded to AWS Lambda and the event source is the component that invokes the Lamdba function.

2.4.2.1 Invocation

The Lambda function can be invoked in the following ways:

- Automatic: When the Lamdba function is invoked automatically through an event published by an event source. The event sources in this invocation are other AWS services including but not limited to:
    - DynamoDB
    - Simple Notification Service
    - Cognito
    - S3
    - CloudWatch Events
- On-Demand: When the Lamdba function is invoked over HTTPS. This is usually accomplished through configuring a RESTful API using Amazon API Gateway. However, the On-Demand invocation can also be achieved by defining custom event sources.

2.4.3 API Gateway

Traditionally, deploying an API requires provisioning servers that host the API. Amazon offers its API Gateway as a managed service for creating and managing RESTful APIs. This allows creating API endpoints for backend applications without the need to manage backend servers. Since resource provisioning is handled by Amazon, the solution is highly scalable and flexible.

The API Gateway can be used to call Lambda functions available in the AWS account, publicly accessible HTTP endpoints and other AWS services. /17/

The group of resources and methods in the API gateway, also known as endpoints, can be deployed to different stages, creating a development lifecycle. The endpoints can also be versioned, allowing to activate a previous version of the API, in a seamless manner.

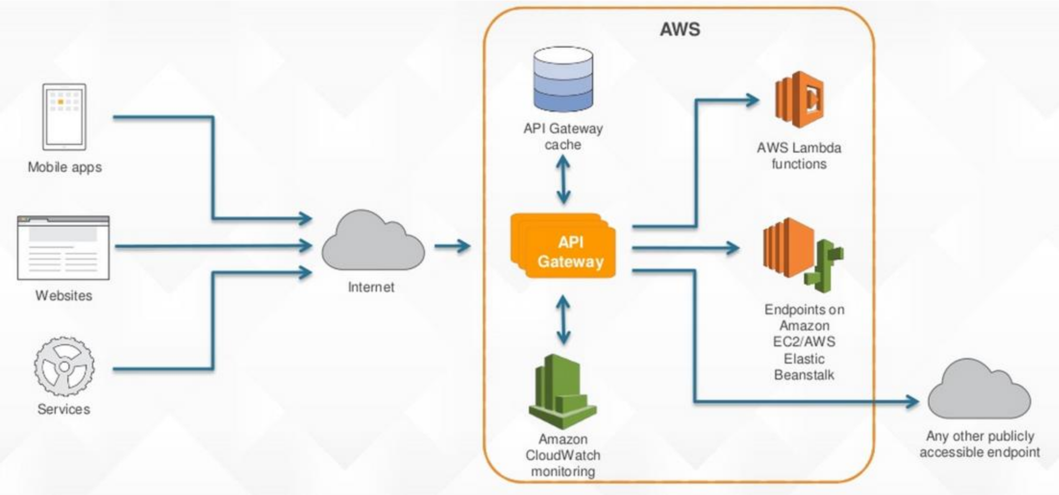

AWS
Mobile apps
Websites
Services
Internet
API Gateway cache
API Gateway
Amazon CloudWatch monitoring
AWS Lambda functions
Endpoints on Amazon EC2/AWS Elastic Beanstalk
Any other publicly accessible endpoint

Figure 5: Amazon API Gateway call flow. Adapted from /17/

2.4.3.1 Client SDK

Amazon API Gateway allows generating SDKs for any stage of an API. The SDKs can then be used to call the API, abstracting much of the development needed to communicate with an API. The currently supported platforms for SDK generation are:

- JavaScript
- Android
- iOS

2.4.3.2 Monitoring and metrics

AWS provides a range of tools that enables tracking the use and performance of the API. The API calls, errors and latency are all monitored. These can be accessed via the API dashboard or through logs in the AWS CloudWatch.

2.4.4 Simple Storage Service

Widely known as S3, Amazon provides this cloud storage service as a means to simplify the task of storing and retrieving objects. It is a highly scalable and on-demand service targeting development teams. /24/

S3 offers automatic object backups across multiple datacenters and has versioning in place, in case objects are deleted by applications/users accidently. In addition, since the service is on-demand, charges are only inferred for the storage used.

2.4.4.1 Static websites

One of the key features of AWS S3 is the ability to host static websites. Web applications that solely rely on client-side scripts for their functionality and do not require server side implementation, can be hosted directly on AWS S3.

By default, S3 allows the hosted website to be accessed from an automatically generated URL. However, a custom domain can also be used to serve the website. This is done in conjunction with Amazon Route 53, a Domain Name System (DNS) service.

2.4.5 DynamoDB

DynamoDB is a NoSQL database offered as a service by Amazon. Being fully managed by AWS, it offers a fast and highly scalable alternative to the conventional method of managing own database server.

2.4.5.1 Capacity units

While creating a table in DynamoDB, the corresponding provisioning settings need to be configured. Using these settings, AWS allocates the required resources to be used for this table and charges are inflicted based on this value. The term “Capacity Units” is used to refer to this setting and it is described as follows:

- 1 write capacity unit: 1 write per second
- 1 read capacity unit: 1 strongly consistent or 2 eventually consistent reads per second

If the frequency of read/write exceeds than the provisioned capacity units, the request fails.

2.4.6 Cognito

Amazon offers Cognito as a user identity management service. In addition, Cognito offers user data synchronization across multiple platforms and devices. /23/

Cognito supports login through external identity providers. These include public identity providers such as Google and Facebook, as well as developer authenticated identities.

2.4.6.1 Identitiy Pools

Cognito uses the concept of Identity Pools. Used to store user identities and the data that may be associated with that identity, Identity Pools also help in syncing user accounts on a variety of platforms.

An Identity Pool supports the following two forms of identities:

- Authenticated: these identities are authenticated using a public or developer login provider
- Unauthenticated: these identities can also be referred to as guests and are not authenticated.

Permissions to Identity Pools are assigned using IAM Roles.

2.5 Miscellaneous

2.5.1 MongoDB

A non-relational database, MongoDB stores information in the form of JavaScript Object Notation (JSON) documents collection rather than tables. /27/

MongoDB is schema-less. This means that two documents in a collection do not have to have the follow the same pattern of key-value pairs. Even though it might be a good practice to follow a specific pattern in a collection to avoid confusion, MongoDB offers the flexibility of having no schema.

Since MongoDB stores data in a JSON format, the data structure is similar to what developers work with inside their programs. This makes it easier to communicate with the MongoDB, specifically when using JavaScript. /27/

The idea behind MongoDB is to retain most, if not all, the scalability and performance of key-value data stores while providing a multitude of functionality. Some of the missing functionality of MongoDB, when compared to a Relational Database Management System (RDBMS), is the lack of Joins and Transactions. However, Joins, for example, is a feature of RDBMS that scales poorly and Transactions is a feature that is not needed most of the time in MongoDB, due to the way data is stored.

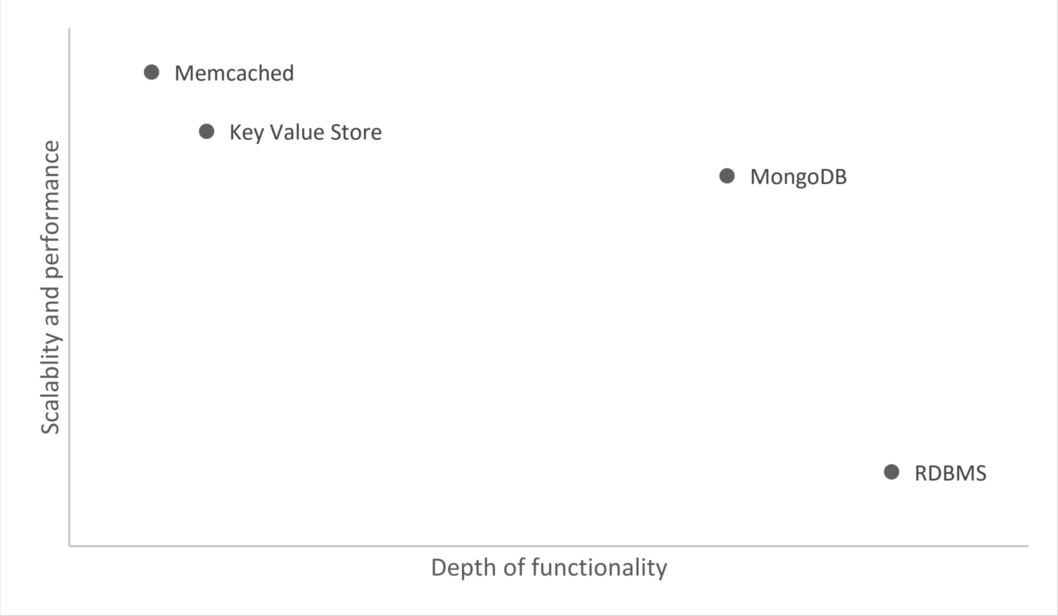


Figure 6: MongoDB comparison with other database systems

2.5.2 REST

Coined in the year 2000 by Roy Fielding in his dissertation, Representational State Transfer (REST) describes a series of constraints that should be laid out whenever two systems communicate with each other. REST provides a series of rules for the server so that all the clients using the service understand what it does and how it works. /4/

Communication with a RESTful interface takes place around resources or a series of resources. Therefore, the service URI will contain nouns and not verbs. However, the HTTP verb used in the request will dictate the type of activity taking place on the resource. HTTP verbs include but are not limited to GET, POST, PUT, DELETE and PATCH.

Another important characteristic of a RESTful interface is Hypermedia as the engine of application state, also known as HATEOS. HATEOS states that in the response of each request to the service, there should be set of hyperlinks that can be used to navigate the API.

2.5.3 CORS

For security reasons, a web browser's same-origin policy allows scripts in a web page to access information in another web page only if they both originate from a single origin. The origin is known based on host name, port number and URI scheme.

Cross-Origin Resource Sharing, simply known as CORS, is a specification from World Wide Web Consortium (W3C) that allows scripts in a browser to access resources on another domain. /15/

In order to enable CORS, the server needs to be configured so that the response header contains the appropriate fields. The fields necessary for a successful exchange vary depending on the HTTP methods used in the request and if any custom headers are used. However, a basic CORS exchange can be achieved by configuring the server to respond to GET requests with an "Allow-Control-Allow-Origin" to a specific origin or "*" for all origins. This will either allow GET requests initiated from a specific origin or all origins, depending on the setting.

2.5.3.1 Preflight requests

Preflight requests involve the browser making a HTTP OPTIONS request to the target resource, before making the actual request initiated by the client. It is used to determine whether the target resource expects the type of request that the client is trying to make and thus allowing or disallowing the request to be made.

The browser may “pre-flight” requests made to another domain if HTTP methods other than GET, POST or HEAD are used in the request. The requests are also preflighted if the request contains custom headers. /15/

2.5.4 OAuth2

An authentication protocol, OAuth2 allows applications to access user data on another application without the need to acquire user password. This is achieved by delegating user authentication to a service where the user already has an account.

2.5.4.1 Application Identity

Before a client can delegate authentication to an Identity Provider, it needs to be registered with that provider. This, depending on the particular service used, will generate keys that can be used later to prove the client’s identity.

2.5.4.2 Flow

Consider a user logging into a blog to leave a comment on a post, using their Google account. OAuth2 defines 4 roles:

1. Client: The blog
2. Resource Owner: The user trying to login to the blog
3. Authorization Server: Google’s service handling user consent and authentication
4. Resource Server: Google’s service for retrieving user profile information

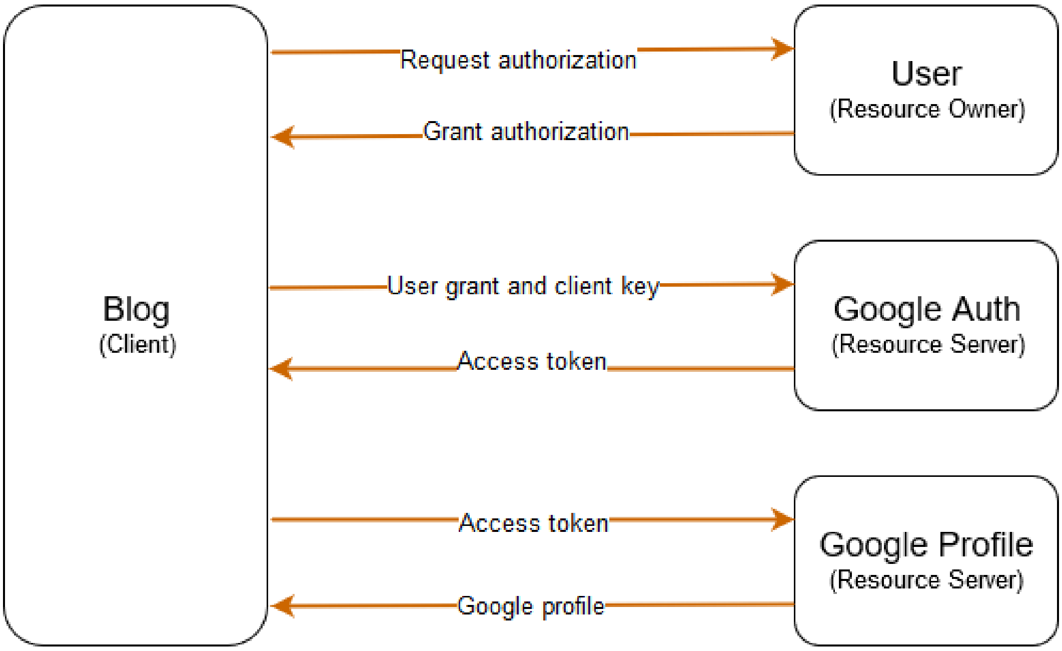


Figure 7: Common OAuth2 flow

3 SYSTEM DESCRIPTION AND DESIGN

3.1 Requirements specification

The requirements for the project can be divided into the following three categories, based on their priority for the success of the implementation:

- Must- haves (1): The basic characteristics the applications must implement.
- Should-haves (2): The necessary features for the application to be satisfactory.
- Nice-to-haves (3): Even though not necessary, these attributes are desirable for the application and could result in a better user experience.

Table 2: Project requirements

No.	Description	Priority
1	User is able to read/write NFC tags using the mobile application	1
2	The mobile application makes use of tabbed UI	3
3	User is able to view tag details through the web application	2
4	The mobile application communicates with the web API for storing/retrieving tag information	1
5	User is able to view history of tags through the mobile application	3
6	The web application allows modification of maintenance and other details, after identification	2

7	The mobile application displays information to be written and make the tag write-protected	1
8	The web application frontend makes use of AngularJS	3
9	The mobile application provides usage instructions	2
10	The mobile application shows history of writes, reads and pending registers to the web service	3

3.1.1 Non-functional requirements

Besides the requirements regarding the functionality of the application, there are also requirements related to the quality of the solution. These include:

- Scalability: the solution must be able to handle spikes in traffic while delivering a seamless user experience.
- Learnability: The use of NFC technology in the mobile application increases the need for usage guidance. Therefore, the application must provide relevant instructions and continuous feedback.
- Security: Since the solution will require user login, the architecture should be designed so that the user data is safe, while providing a secure method of communication with the backend.

3.2 Use Cases

Identifying use cases for an application provides an overview of the interaction between the application and its user/external system. This in turn offers a clear understanding of the functionality that the application is to deliver, based on an input from an external entity.

The use cases can be presented in a logical approach using a Use Case Diagram. The aforementioned diagram and an explanation of each use case are provided in the following sections.

3.2.1 Android Application

3.2.1.1 Use Case Diagram

Figure 8 shows the interaction between the user and DiveSafe mobile application through a use case diagram.

The actor in a use case diagram performs actions that trigger an event in the application flow. Since the application is primarily going to be used by diving equipment shops, the actor in this use case is the shop staff.

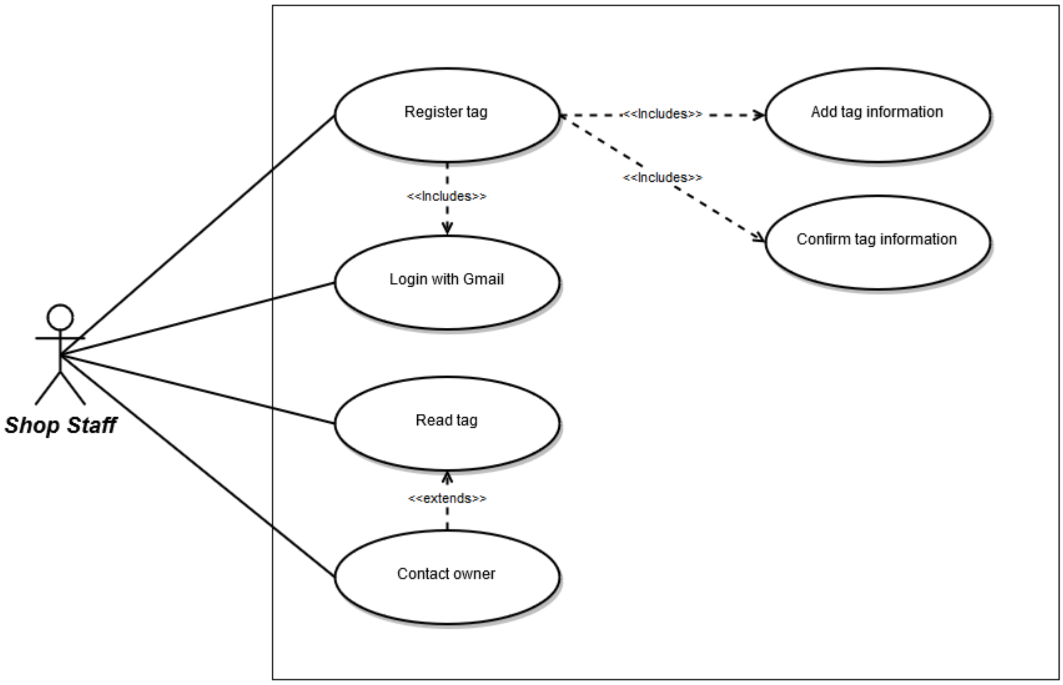


Figure 8: Use Case Diagram for the DiveSafe mobile application

3.2.1.2 Login with Gmail

Since this project uses Identity Federation, the users of the mobile application login using their Google account in order to access some features.

The characteristics of this use case can be seen in Table 2.

Table 3: Use case “Login with Gmail”

Description	The user logs in using their Google account

Preconditions	Valid Google account
Input	Google Identity Token
Triggered Actions	Using the Identity Token, the user is verified to AWS Cognito and temporary IAM credentials are issued
Result	User is informed of the login and their display name is shown
Exceptions	No network access

3.2.1.3 Register Tag

In order for a Tag to be part of the DiveSafe system, it must be registered first. Registering a new Tag involves writing the information the user provides to both the DiveSafe database and the tag itself.

For a successful registration to occur, the user must provide the appropriate details through the mobile application, review and confirm the information, and tap the tag with the mobile device.

It is only possible to register tags that are not already part of the system. Therefore, the registration will not proceed if the tag is not empty.

The characteristics for this use case can be seen in Table 3.

Table 4: Use case “Register tag”

Description	The user registers a new tag to the system
Preconditions	The tag is not already in the system
Input	All tag and owner related data

Triggered Actions	Write all data to the server and if successful, add basic data to the Tag.
Result	A success message is shown to the user. The tag is now in the system
Exceptions	The tag is already registered/not empty/not valid

3.2.1.4 Read tag details

Table 4 describes the characteristics of “Read basic tag details” use case.

Only a few of the basic details that are provided by the user when registering the tag are stored on the tag. Therefore, this information can be accessed even when no internet access is available.

Table 5: Use case “Read tag details”

Description	The user reads the information stored on the tag
Preconditions	-
Input	The tag is tapped with the device
Triggered Actions	-
Result	Tag details are shown to the user with the controls for SMS, call and email ability
Exceptions	The tag is empty

3.2.2 Web Application

3.2.2.1 Use case diagram

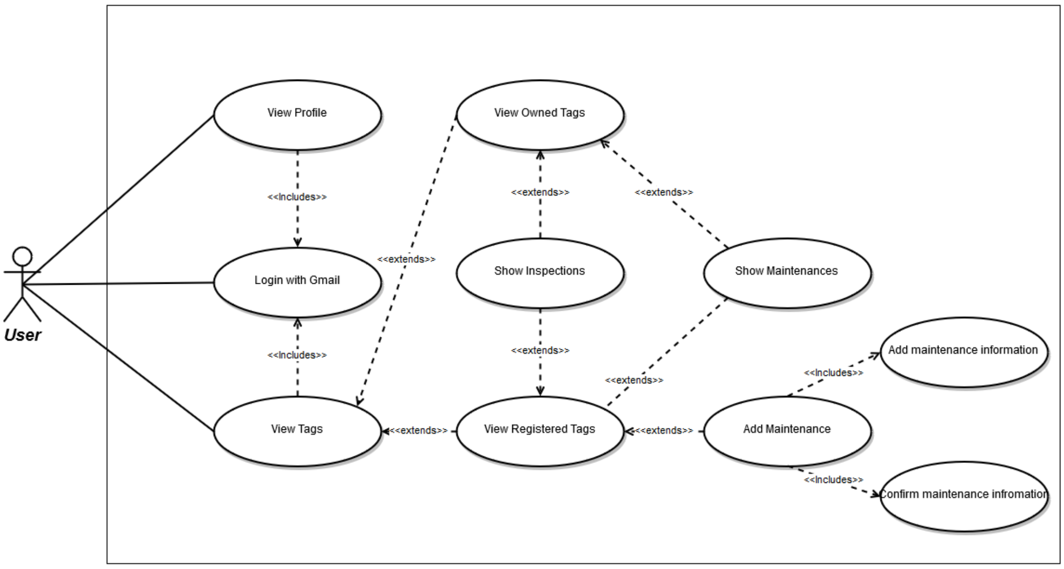


Figure 9: Use case diagram for DiveSafe web

3.2.2.2 Login with Gmail

Since this project uses Identity Federation, the users of the mobile application login using their Google account in order to access most of its features.

The characteristics of this use case can be seen in Table 5.

Table 6: Use case “Login with Gmail”

Description	The user logs in using their Google account
Preconditions	Valid Google account
Input	Google Identity and Access Tokens
Triggered Actions	Using the Identity Token, the user is verified to AWS Cognito and temporary IAM credentials are issued. Basic user profile is fetched using the Access Token

Result	User is informed of the login and redirected to the application dashboard
Exceptions	No internet access

3.2.2.3 View Profile

Users are able to view brief information about their profile through the Profile section. The information largely displays the data fetched from the Google APIs. The idea is to allow users observe what Google account they are using to perform various operations in the application and possibly switch the account, if needed.

Table 7: Use case “View Profile”

Description	Tag owner views their Profile information
Preconditions	User must be logged into the application
Input	-
Triggered Actions	User information fetched from the Google API during the login process is loaded from the local storage
Result	Profile information is shown to the user
Exceptions	Session expired

3.2.2.4 View registered Tags

As part of the Tag registration process through the Android application, the Google account logged into the mobile application is assigned the “registerer” role. By logging into the web application using the same account, all the Tags registered by the account can be accessed.

Table 7 shows the features of this use case.

Table 8: Use case “View registered Tags”

Description	The user accesses the Tags registered by their account
Preconditions	User must be logged into the application
Input	-
Triggered Actions	Using the temporarily issued AWS credentials, all tags in the system registered to this account are fetched through the API
Result	A list of Tags with the relevant information is shown on the screen
Exceptions	Session expired

3.2.2.5 Add maintenance

Through the DiveSafe web application, the logged in user can add maintenance information to the Tags they have registered using the Android application.

Table 9: Use case “Add maintenance”

Description	Maintenance information is added by the user
Preconditions	User must be logged into the application and possess the “registerer” permissions to the Tag
Input	All maintenance related information for instance date, name and comments
Triggered Actions	Using the temporarily issued AWS credentials, a new maintenance instance is recorded in the database through the API
Result	Success message is shown to the user

Exceptions	Invalid data, session expired

3.2.2.6 View owned Tags

Tag owners are able to view their Tags through the DiveSafe web application, by logging into the application using the Google account provided at tag registration. Tag owners are able to, along with basic Tag information, view inspections and maintenances. They are not, however, able to add instances of inspections or maintenances.

Table 10: Use case “View owned Tags”

Description	Tag owner views the information of their Tags
Preconditions	User must be logged into the application and possess the “owner” permissions to the tag
Input	-
Triggered Actions	All Tags having the logged in Google account as the owner are fetched from the database and through the API
Result	A list of Tags is shown to the user
Exceptions	Session expired

3.2.2.7 Show inspections/maintenances

Through the DiveSafe web application, users are able to view the inspection/maintenance information for both the Tags they own and the Tags that have been registered using their Google account.

Table 11: Use case “Show inspections/maintenances”

Description	User views the inspection/maintenance information of a tag
Preconditions	User must be logged into the application

Input	-
Triggered Actions	The list of maintenance/inspection information, using the Tag id, is fetched through the API
Result	A list of inspections/maintenances is shown on the screen
Exceptions	Session expired

3.3 Sequence Diagrams

Modeling the flow of events in a sequential manner, sequence diagrams are used to document and communicate the design of an application. /3/

This section provides the sequence diagrams related to activities in the DiveSafe system. The figures illustrate how the user interacts with the system and the activities that happen in the background, in sequence.

The sequence diagram for reading the basic tag details can be seen in Figure 10. As it is shown in the figure, after the user has tapped the tag with the device, the Android Operating System (OS) detects and dispatches the tag to the DiveSafe application, invoking the necessary methods that continue the process.

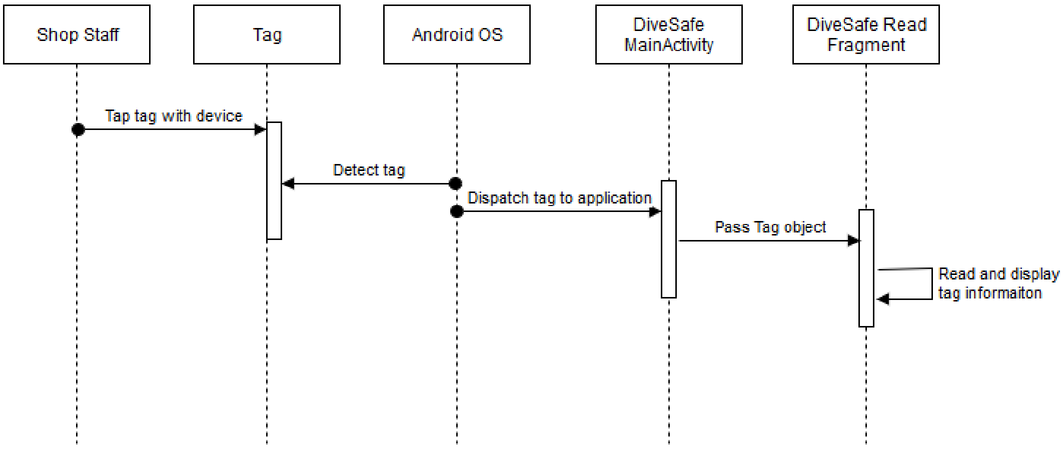


Figure 10: Sequence diagram "Read tag details"

The sequence diagram for logging into the application using Google is shown in Figure 11.

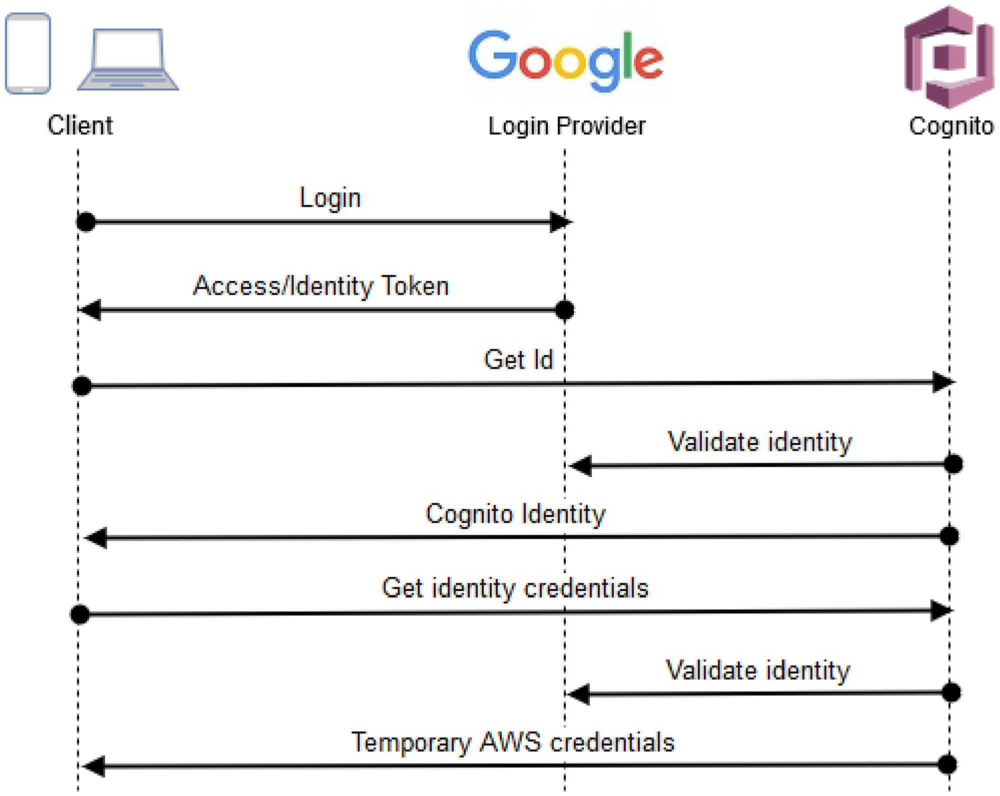


Figure 11: Sequence diagram "Login with Google"

The sequence diagram for registering a new Tag to the system can be seen in Figure 12.

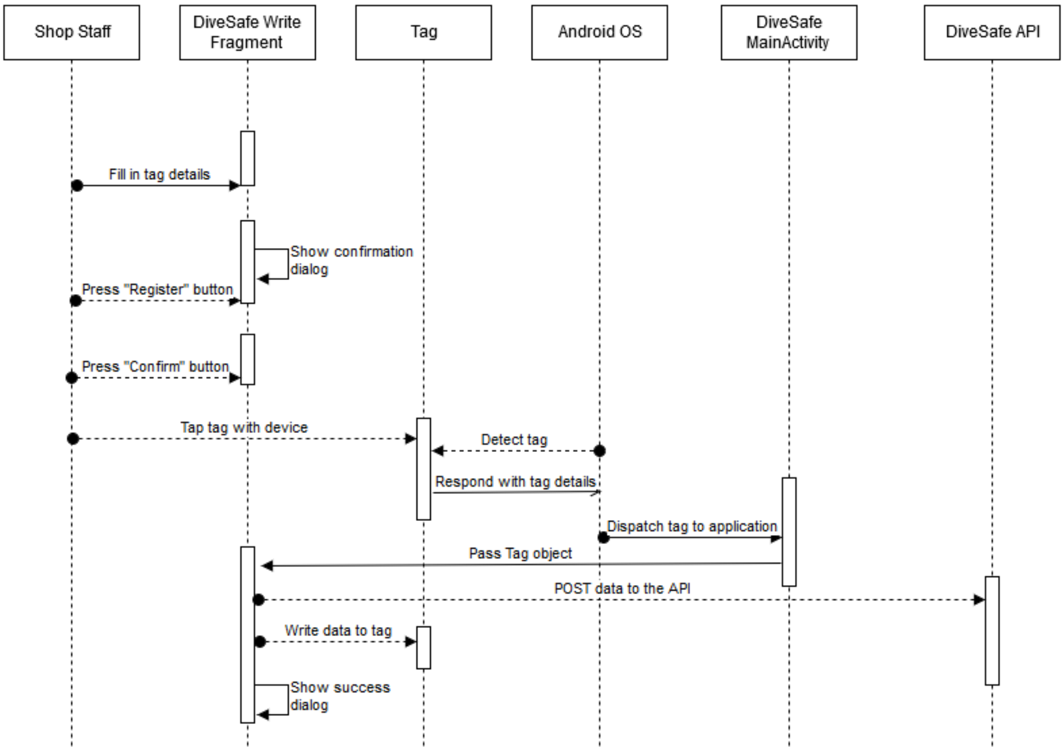


Figure 12: Sequence diagram “Register new Tag”

3.4 Application design

In this chapter, the system design and how different platforms are integrated, will be discussed on an abstract level. A more detailed view of how the design is implemented in practice, is discussed in the next chapter.

Since the DiveSafe project makes use of applications developed for and running on different platforms, it is appropriate to describe the design and architecture of each application separately, in addition to the system as a whole.

3.4.1 Three-tier Architecture

When dealing with user-facing applications, the three-tier architecture is a common configuration. In this pattern, the tiers that form the application are the presentation, logic and data tiers.

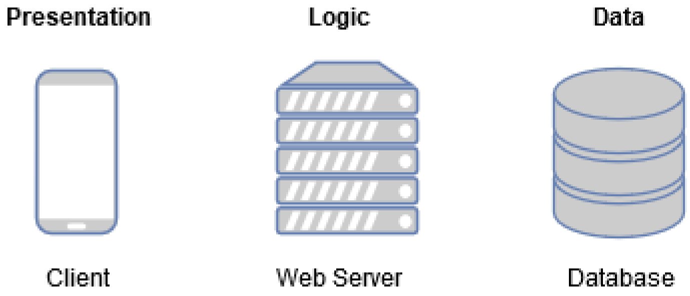


Figure 13: Three-tier Architecture

The Presentation tier comprises of the components that the user can directly interact with. These can include, for example, mobile app UIs and webpages. The Logic tier comprises of the code and functionality required to determine application behavior, in response to interactions at the Presentation tier. The Data tier involves the persistence of changes and information.

3.4.2 Service-Oriented Architecture

A service is a piece of software which provides functionality to other pieces of software within a system. The other pieces of software could be, for example, a website, a mobile application, a desktop application or another service. For instance, in the context of an e-commerce solution, the functionality of placing an order could be handled by a service that performs all the necessary operations for processing an order successfully. Therefore, this service provides functionality to the e-commerce website.

A system that employs a service or multiple services to provide functionality in this manner is considered to be using a Service-Oriented-Architecture (SOA). Using this architecture allows clients from different platforms to connect and use the same service, reusing functionality.

SOA eases the process of scaling an application, in response to the increase in demand. This is achieved through copying the service into new servers and using a Load Balancer to redirect traffic to servers, depending on how busy they are.

Furthermore, SOA provides reusability of functionality. For example, in our previous example of an e-commerce solution, the service could provide the functionality of placing an order to a website or a mobile application.

One of the key aspects of an SOA is the use of standardized contracts or interfaces. Therefore, the signature of the method that the client uses to interact with services does not change when the service is modified. Therefore, upgrading a service does not require upgrading the clients that rely upon that service, as long as the interface has not been modified.

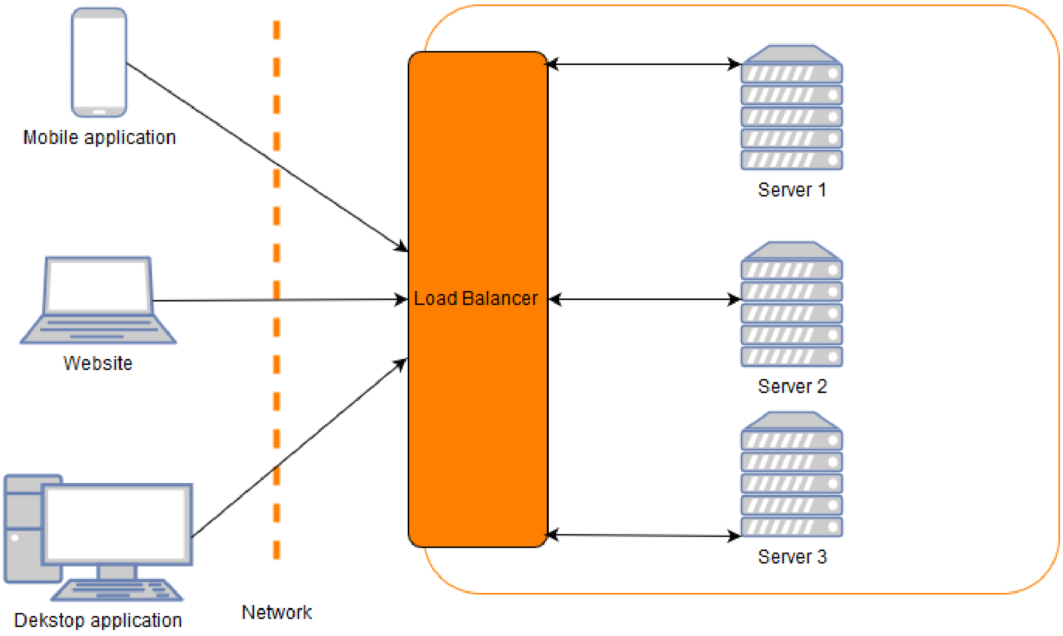


Figure 14: SOA and scaling

3.4.3 Microservices Architecture

Due to the lack of specific guidelines on sizing services in the traditional SOA, the services became large and monolithic, as the requirements grew. This resulted in services being difficult to change and less scalable.

Defined as an improved version of the Service-oriented Architecture, Microservices Architecture shares the main characteristics of SOA. These include the reusability, scalability, statelessness and the standardized interface for backwards compatibility.

The Microservice Architecture proposes guidelines for sizing services. This architecture, as the name suggests, uses smaller services for addressing specific needs. This in turn makes the services more efficient to scale and flexible.

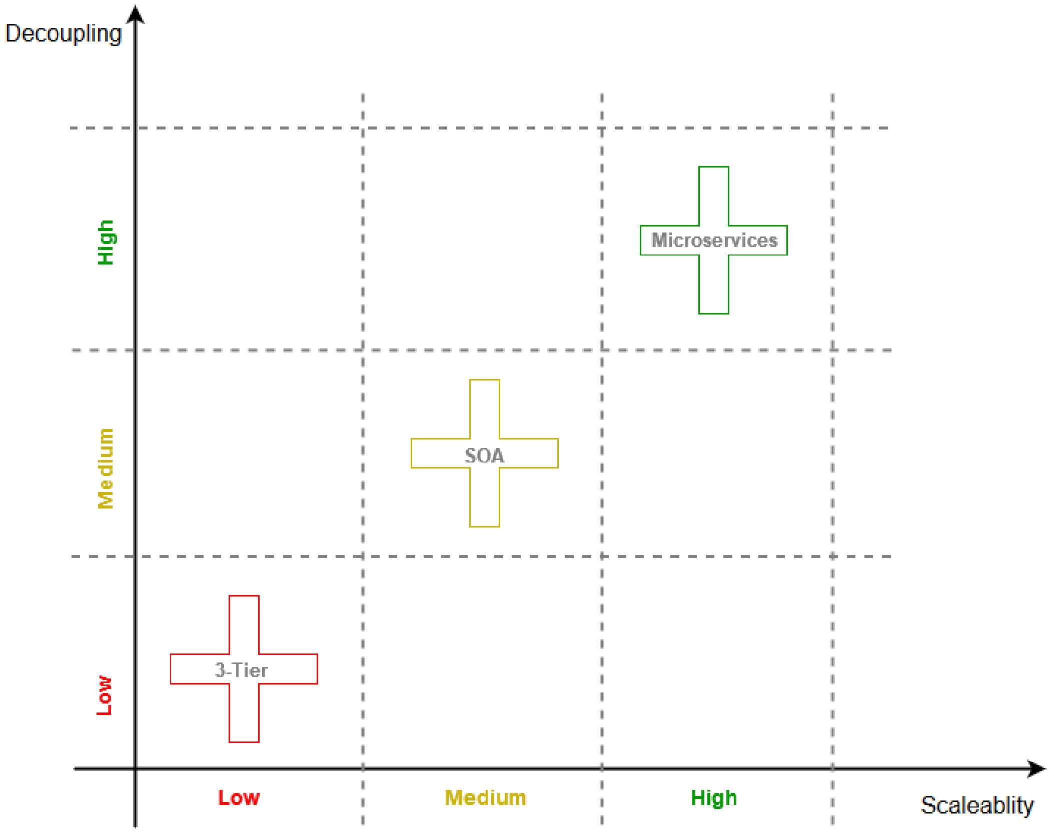


Figure 15: Architecture Comparison

Applications using this architecture make use of multiple micro services to provide functionality to specific part of the application. Since the micro services may communicate with client or other micro services, the communication mechanism is lightweight.

Services in a Microservices Architecture are independent of each other. This means that each service can be changed and deployed, without affecting other services in the system.

3.4.4 Server-less Architecture using AWS Microservices

In this project, an architecture commonly referred to as “Server-less Architecture” was employed using Amazon Web Services. When using this architecture, the provisioning of servers is handled by AWS. This eliminates the following risk factors:

- Under-provisioning: leading to performance decline
- Over-provisioning: leading to increased cost
- Development cost: without the need to manage and secure servers, the development cost is reduced

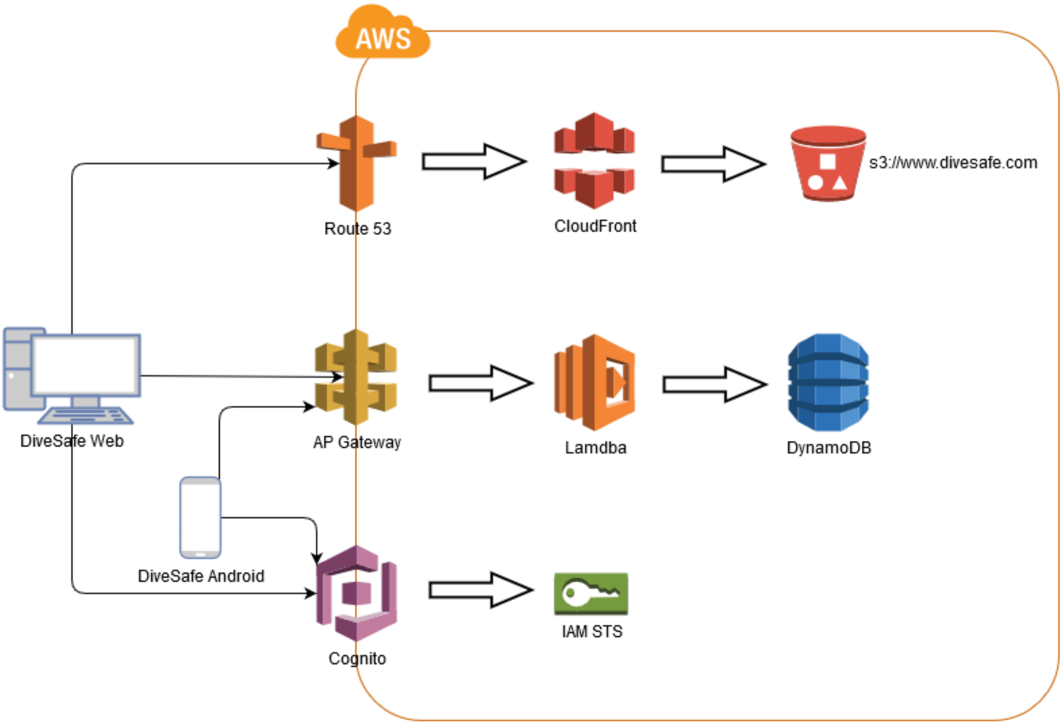


Figure 16: “Server-less” Architecture using AWS

As mentioned in the “Used Technologies” chapter, AWS S3 can be used to serve static files. This makes it possible to host websites on S3 that do not need server side implementation and can solely rely on communicating with API to fetch the necessary data. AngularJS and other client-side JavaScript frameworks allow this functionality.

3.4.4.1 Event-driven

Since resources are assigned in response to the traffic, the system can be described as event-driven. If there are no requests made to the micro-services, there will be no servers running in the system, thus minimizing the cost of the solution. For example, when a Lambda function is invoked, it automatically assigns resources to run the code and perform the tasks required.

3.4.5 Fragment-Oriented Architecture for Android

To produce code that is maintainable and testable, the view of an application is separated into different components. In Android, this is achieved using Fragments.

Using Fragment-Oriented Architecture, the Android application is divided into Fragments so that the final solution contains one primary Activity with multiple Fragments. The Fragments provide the User Interface(UI) and related logic for a particular functionality. The Activity is mainly used to add/remove Fragments and for inter-Fragment communication.

It is a recommended practice to use an Interface for communication between a Fragment and its containing Activity or other Fragments. This is to keep the Fragment modular and easier to test.

Figure 17 shows the architecture diagram of the Android application.

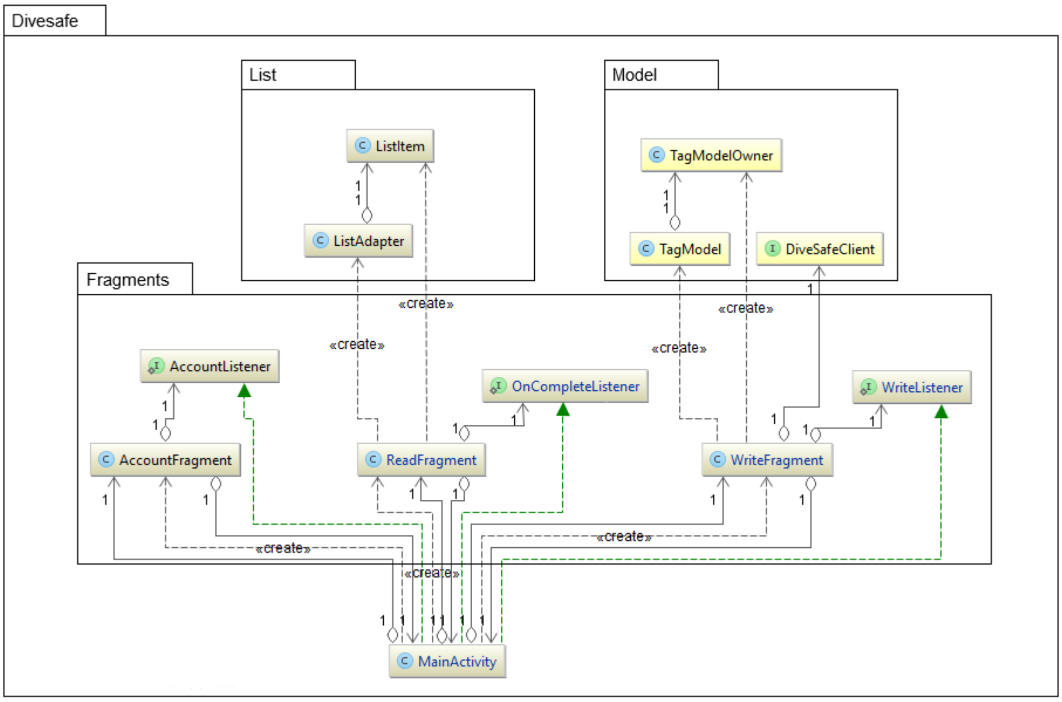


Figure 17: Fragment-Oriented Architecture

4 IMPLEMENTATION

This chapter provides an overview of the key implementation topics in the project. Most of the theory needed for this chapter is provided in the “Technologies” chapter. However, some project-specific information will also be provided throughout the upcoming sections.

Since the project involves multiple platforms, it is perhaps more appropriate to discuss the implementation aspect of each platform separately.

4.1 Android

The mobile application for this project was developed using the so called “Fragment-oriented Architecture”. In this design pattern, Android applications contain a single Activity and multiple Fragments. Each fragment is responsible for a specific task, handling both the UI and logic aspects of that task. The Activity is primarily used for inter-app/inter-fragment communication, application UI and session management.

Using the Fragment-oriented Architecture provides a good separation of concerns, easier inter-app communication and generally produces a more maintainable code.

4.1.1 Enabling NFC

In order to enable the application to access the device’s NFC capabilities, the AndroidManifest.xml file needs to include the following permission:

```
<uses-permission android:name=”android.permission.NFC” />
```

An additional step is to limit the installation of the application to NFC enabled devices. This, while optional, enhances user experience by notifying users whether they are actually able to use the application:

```
<uses-feature android:name="android.hardware.nfc" />
```

4.1.2 Reading NFC Tag

As discussed in the “Used Technologies” chapter, when an NFC tag is detected, Android begins a process aimed at finding the best candidate Activity to launch and pass the Tag object. This is done by analyzing the list of Activities expecting NFC Tags, whether or not `enableForegroundDispatch()` method has been called by an activity and the corresponding filters.

Even though an Android Activity is on the foreground and is visible to the user, it does not automatically possess any priority over other applications/activities registered for NFC tags. The method `enableForegroundDispatch()` provides the activity that priority.

As mentioned, the first priority is given to the Activity that has called `enableForegroundDispatch()`. If no activity has called the method, the type of the first record in the Tag is determined. Next, the Activity registered for that Tag type using the most specific filter matching the determined type is launched and the Tag is passed to it. Therefore, to make sure that this application is launched, filters were assigned according to the Tag type and stored data.

```
/*
* Invoke only when the Activity is in the foreground and not writing tag
*/
public void enableTagReadMode() {
    mWriteMode = false;
    // since Tags are expected to be in NDEF format, use the corresponding filter when
    // assigning found tags to this activity
    IntentFilter[] filters = new IntentFilter[1];
    filters[0] = new IntentFilter();
    filters[0].addAction(NfcAdapter.ACTION_NDEF_DISCOVERED);
    filters[0].addCategory(Intent.CATEGORY_DEFAULT);
    try {
        filters[0].addDataType(MIME_TEXT_PLAIN);
    } catch (IntentFilter.MalformedMimeTypeException e) {
        throw new RuntimeException("Check the mime type");
    }

    // give Tag handling priority to this activity, with the given filters
    mNfcAdapter.enableForegroundDispatch(this, mPendingIntent, filters, null);
}

/*
* Invoke in the onPause() method
*/
public void disableTagReadMode() {
    mNfcAdapter.disableForegroundDispatch(this);
}
```

When the Tag object is received as part of the delivered intent, the stored records can either be retrieved using `Ndef.getCachedNdefMessage().getRecords()`. The method for retrieving data from the `NdefRecord` depends on the type of the record. This can be found using the NFC Forum Specification. Following is an example of retrieving data from an `NdefRecord` with type equal to TEXT.

```
/*
 * http://www.nfc-forum.org/specs/
 *
 * bit_7 defines encoding
 * bit_6 reserved for future use, must be 0
 * bit_5..0 length of IANA language code
 */
byte[] payload = record.getPayload();

// Get the Text Encoding
String textEncoding = ((payload[0] & 128) == 0) ? "UTF-8" : "UTF-16";

// Get the Language Code
int languageCodeLength = payload[0] & 0063;

// Get the Text
return new String(payload, languageCodeLength + 1, payload.length - languageCodeLength
- 1, textEncoding);
```

4.1.3 Writing NFC Tag

In order to write to the Ndef Tag, an `NdefMessage` with at least one `NdefRecord` needs to be created. The required method for creating a record depends on the type of record. Following is an example of creating an `NdefRecord` with type TEXT.

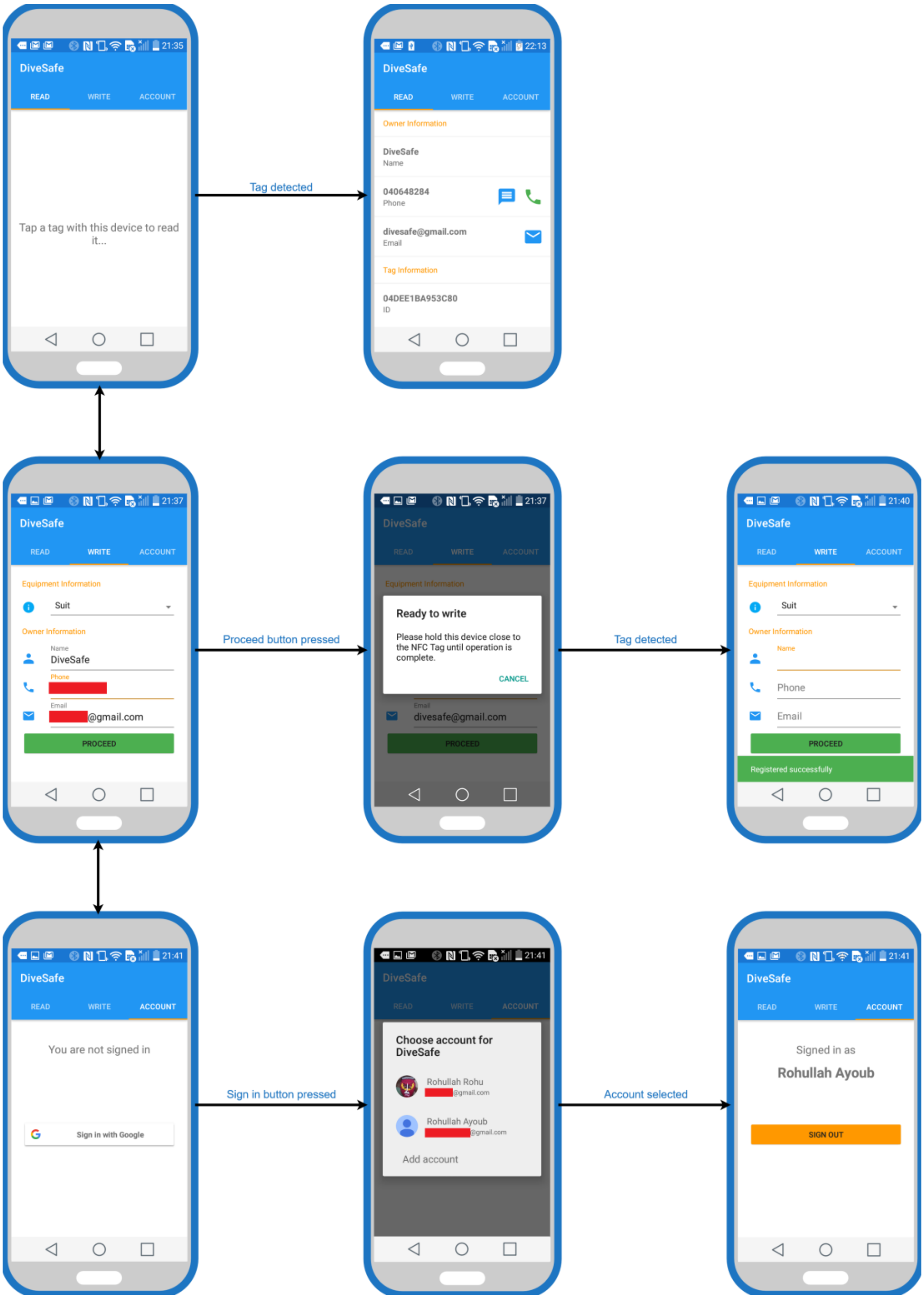
```
// Generate a language code in bytes
String lang = "en";
byte[] langBytes = lang.getBytes("US-ASCII");
int langLength = langBytes.length;

// Generate text data in bytes
byte[] textBytes = text.getBytes();
int textLength = textBytes.length;

// Actual payload to be saved to the record
byte[] payload = new byte[1 + langLength + textLength];

// set status byte (see NDEF spec for actual bits)
payload[0] = (byte) langLength;

// copy langbytes and textbytes into payload
System.arraycopy(langBytes, 0, payload, 1, langLength);
System.arraycopy(textBytes, 0, payload, 1 + langLength, textLength);

return new NdefRecord(NdefRecord.TNF_WELL_KNOWN, NdefRecord.RTD_TEXT, new byte[0],
payload);
```

When the NdefMessage has been created and the tag is in range, the information can be written to the tag by first calling `Ndef.connect()` followed by `Ndef.writeNdefMessage(NdefMessage)`. After the write operation is finished, `Ndef.close()` is called to gracefully close I/O operation.

4.1.4 Application views

Since the mobile application involves communication with an NFC tag, it is imperative that it provides a user interface that is easy to follow. Therefore, a tabbed user interface was implemented using Google's Material design. Following Material design specification ensures that the user is presented with a familiar array of controls.

Figure 18 shows the corresponding views of the three tabs used in the initial version of the mobile application. Please note that only screens for successful operations are given here.



Tap a tag with this device to read it...

Owner Information

DiveSafe
Name

040648284
Phone

divesafe@gmail.com
Email

Tag Information

04DEE1BA953C80
ID

Equipment Information

Suit

Owner Information

Name
DiveSafe

Phone
[redacted]

Email
[redacted]@gmail.com

Ready to write

Please hold this device close to the NFC Tag until operation is complete.

Email
divesafe@gmail.com

Equipment Information

Suit

Owner Information

Name

Phone

Email

Registered successfully

You are not signed in

Choose account for DiveSafe

Rohullah Rohu
[redacted]@gmail.com

Rohullah Ayoub
[redacted]@gmail.com

Add account

Signed in as
Rohullah Ayoub

Figure 18: DiveSafe Mobile views

4.2 AngularJS

In order to streamline the development process and produce code that is easier to migrate to Angular 2, Angular best practices authored by John Papa and backed by the Angular team/11/ were used in this project. This involves, among other things, separating HTML partials, controllers, services and directives into smaller files/folders, based on their role in the project.

4.2.1 Authentication

Since the web application of this project is hosted on AWS S3, the OAuth 2.0 Implicit Grant type authentication is used, which does not require a server side implementation. An abstract explanation of the steps required for authentication and authorization is given below.

1. Client: Build request to Google authorization server with configuration. Create a popup for authorization, using the built request.
2. Authorization Server: If authorized by user, respond with the access and identity token to the popup.
3. Client: Extract identity and access token from popup URL. Close the popup and save the tokens to local storage.
4. Client: Retrieve Google user profile using access token and save to local storage.
5. Client: Request AWS credentials from Cognito, using local identity token
6. Cognito: Validate the identity with Google authorization server and if token is valid, respond with temporary AWS credentials
7. Client: Use AWS credentials with all future requests.

4.2.2 Optimizing for production

The implication of John Papa's widely used design pattern is the necessity to include all application/vendor JavaScript and HTML files into the `index.html`. Each of these will require a separate network call. In addition, separate network calls will be made for any application/vendor stylesheets. This is not a desired behavior in production.

To produce an optimized build for deployment, Gulp with plugins is used to minify and combine all:

- Application JavaScript into `app.js`
- Vendor(excluding apigateway sdk) JavaScript into `lib.js`
- HTML partials into Angular `$templateCache` and inject into app.js
- ApiGateway SDK into `apigateway.js`
- Application CSS into `app.css`
- Vendor CSS into `lib.css`

The files are then injected into the index file. Additionally, file name revisions are used in all generated files. This is used for the purpose known as "Cache-busting". Cache-busting is used to prevent browsers from using outdated resources that are cached. Each modification results in a unique file name, forcing the browser to request the new resource.

4.3 AWS

The server-less implementation in this project involves using Lambda as the backend, DynamoDB as the data store and invoking Lambda through the API Gateway. Even though these services are all offered by AWS and logged in account may have write/read access to all of these services, IAM Roles with necessary access need to be used to grant the services access to each other.

For example, the Lambda functions accessing DynamoDB tables will need to have a policy attached to them that allows access to those tables.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "TagsDynamoDBRead",
      "Effect": "Allow",
      "Action": [
        "dynamodb:GetItem",
        "dynamodb:Scan"
      ],
      "Resource": [
        "arn:aws:dynamodb:eu-west-1:884453216036:table/Tags"
      ]
    }
  ]
}
```

The policy given above grants read access to the Tags table in DynamoDB and is attached to the `readTag` and `listTags` Lambda functions.

4.3.1 Deploying a RESTful API

AWS API Gateway provides the ability to deploy the API to custom deployment stages. The names and number of these deployment stages can vary depending on the product lifecycle methodology used within the organization. For the purpose of this project, the API resides in the “dev” stage.

The diagram of the /tags and its child /tags/{id} resource is given in the Figure 19:

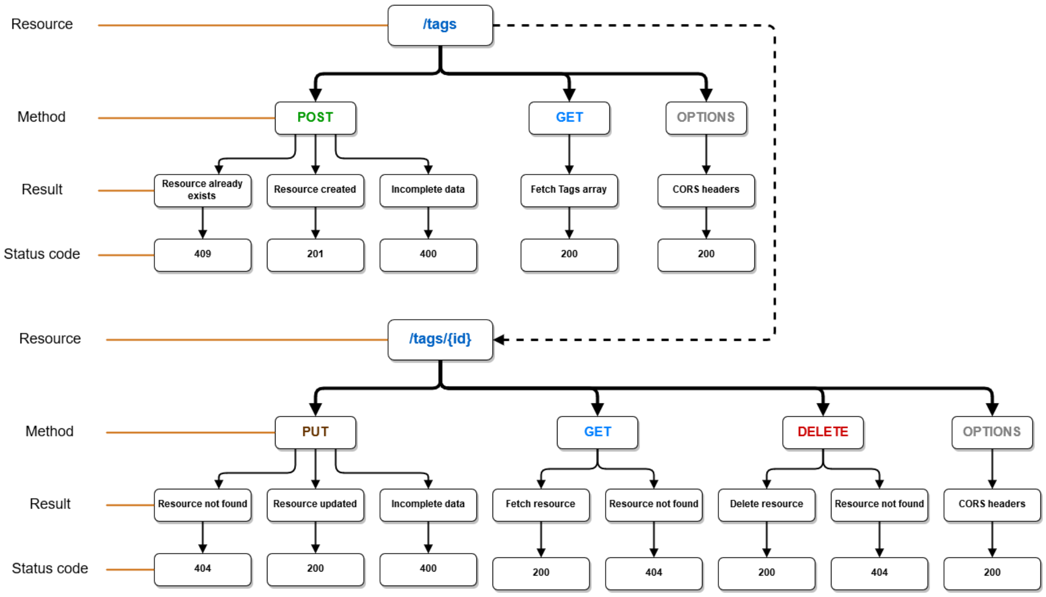


Figure 19: API diagram for /tag resource

Currently all the methods that provide CRUD functionality on resources require IAM authorization. This means that a custom “X-Amz-Security-Token” is required in the header of each request.

4.3.1.1 Models

Even though not obligatory, it is good practice to create Models for the resources in AWS API Gateway. The models are defined using JSON Schema and are mainly used to communicate the expected format of data. This provides several benefits including:

- automatically generating output templates
- class description for Models in the client SDK
- reducing the chances of a malformed request

Since this project primarily focuses on Tag and User resources, a model for each was created. The model for Tags is created as follows:

```
{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "DiveSafeTagModel",
  "type": "object",
  "properties": {"id": { "type": "string" },"type": {"type": "string"},
    "registrationDate": {"type": "string"},
    "maintenance": {
      "type": "object",
      "properties": {
        "date": {"type": "string"},"name": {"type": "string"},
        "comment": {"type": "string"}
      }
    },"inspection": {
      "type": "object",
      "properties": {
        "date": {"type": "string"},"name": {"type": "string"},
        "comment": {"type": "string"}
      }
    },"owner": {
      "type": "object",
      "properties": {
        "name": {"type": "string"},"phone": {"type": "string"},
        "email": {"type": "string"}
      }
    },"registerer": {"type": "string"}
  }
}
```

Using the Model above, a large part of effort to form proper requests is abstracted from the clients.

4.3.2 CRUD Operations in Lambda

While creating resources through the API Gateway, each HTTP method is assigned an existing AWS Lambda function that will be invoked on a request. Each Lambda function is a standalone module.

For the purpose of this project, Node.js runtime environment was used for the execution of server-side functions. Each function is called with two arguments:

- `event`: This object contains any passed in values like HTTP request body, query and url parameters
- context: This object provides runtime information including Lambda configuration. It also provides callback functions to indicate the result of the operation

For example, the following code fetches a Tag item from DynamoDB, using the provided id through the `event` object. It then invokes the callback functions in the `context` object, depending on the result of the operation.

```
var doc = require('dynamodb-doc');
var dynamodb = new doc.DynamoDB();
// dynamodb table with hash key = id
var tableName = "Tags";
exports.handler = function(event, context) {
    var payload = {
        "TableName": tableName,
        "Key" : {
            "id": event.id
        }
    };
    dynamodb.getItem(payload, function(err, data){
        if(err){
            context.fail();
        }
        // if the result is empty, provide the proper response
        if(JSON.stringify(data) === JSON.stringify({})){
            context.fail("Not Found: The requested Tag cannot be found");
        }
        context.succeed(data);
    });
};
```

4.3.3 Deploying to S3

After the web application code has been optimized, it can be deployed to S3 using the AWS console. This is done by creating a new S3 bucket and uploading all the files, with index.html residing at the root of the bucket. An automatic URL is generated in the following format:

`<bucket-name>.s3-website-<AWS-region>.amazonaws.com`

In order to make it easier to use a custom domain at later point, the bucket name is chosen so that it follows the same pattern as that of a future domain name. In addition, two buckets are created: bucket name with and without the “www” prefix. This is necessary due to the internals of how Route 53 maps the domains to S3 buckets and to be able to serve the website for both `domain.com` and `www.domain.com`.

4.3.4 Configuring Cognito

AWS Cognito offers a simple interface for configuring well known OpenID providers by providing dedicated fields, tailored to the corresponding OpenID provider. For example, only a Google application client Id is needed to configure a Google sign in.

However, when using Google sign in on more than one platform, Google has to be configured as an OpenID provider through IAM and each platform's Google client id needs to be added. If Google is configured both through the IAM and Cognito, none of the two configurations will work.

4.4 Tests and Analysis

This section provides an overview of the necessary setup for testing the major parts of the application. It also details the outcome of the project in terms of learning and achievements.

4.4.1 Testing fundamentals

Testing the NFC capabilities of an Android application, which is one of the primary test cases in this project, generally requires physical NFC tags. This is due to the complexities involved in Activity/Fragment lifecycle and the lack of comprehensive resources on the subject. In the beginning of the project and before the physical NFC tags were delivered, the emulator provided by the OpenNFC project was used to test the application.

4.4.1.1 Using Reflection for mocking

To automate the testing process, a mock of the Ndef Tag object needs to be created. However, the constructor for Tag object is only for internal use and is not public. /25/ In addition, Android framework does not provide accessible methods for instantiating a new Tag object. Therefore, Reflection needs to be used to mock the Tag.

The term “Reflection” is used in Java and other programming languages for the concept of examining and possibly modifying the behavior of classes at runtime. It gives programs access that would be otherwise illegal, like accessing private methods. Reflection, in this case, is used to access a private method of the Tag class, in order to create a mock Tag.

The Android Tag object includes a method `createMockTag` that can, through reflection, be invoked with the necessary arguments to return a Tag instance corresponding to the arguments. The Tag instance is then passed on for testing.

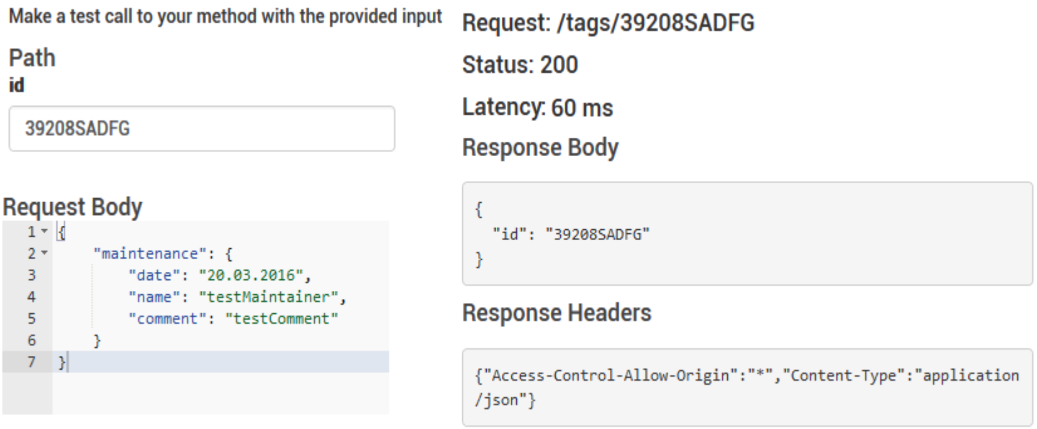
```
// Using Reflection, gain access to the otherwise inaccessible createMockTag method
Class tagclass = Tag.class;
Method createMockTagMethod = tagclass.getMethod("createMockTag", byte[].class,
        int[].class, Bundle[].class);

// NdefMessage with a simple text record
NdefMessage myNdefMessage = new NdefMessage(Utils.createTextRecord(textToWrite));

// values obtained from the tag's specification
Bundle ndefBundle = new Bundle();
ndefBundle.putInt(EXTRA_NDEF_MAXLENGTH, 137); // maximum message length
ndefBundle.putInt(EXTRA_NDEF_CARDSTATE, 2); // read-write ability
ndefBundle.putInt(EXTRA_NDEF_TYPE, 2); // Type 2 tag
ndefBundle.putParcelable(EXTRA_NDEF_MSG, myNdefMessage);  // add the NDEF message

// Unique 7-byte identifier for type 2 tag
byte[] tagId = new byte[] { (byte)0x3F, (byte)0x12, (byte)0x34, (byte)0x56,
(byte)0x78,
        (byte)0x90, (byte)0xAB };


Tag mockTag = (Tag)createMockTagMethod.invoke(
        null, // null since we dont have an object
        tagId, // tag UID/anti-collision identifier
        new int[] {TECH_NDEF }, // tech-list
        // tech-extra bundles mapping to to entries in tech-list
        new Bundle[] { ndefBundle });
```

The mock Tag is then used, for example, to perform unit tests on the Fragment responsible for reading Tags.

4.4.1.2 API Gateway and Lambda Integration

AWS console provides an interface for testing the integration of an API Gateway method against the corresponding Lambda function. This is important since in some cases, the request parameters received in the API Gateway need to be modified into a format that can be consumed by the Lambda function.

Figure 20 shows an integration test of `/tags/{id}` resource and the `updateTag` Lambda function.



Make a test call to your method with the provided input

Path

id

39208SADFG

Request Body

```
{
    "maintenance": {
        "date": "20.03.2016",
        "name": "testMaintainer",
        "comment": "testComment"
    }
}
```

Request: /tags/39208SADFG

Status: 200

Latency: 60 ms

Response Body

```
{
  "id": "39208SADFG"
}
```

Response Headers

```
{"Access-Control-Allow-Origin":"*","Content-Type":"application
/json"}
```

Figure 20: API-Lambda Integration test

4.4.1.3 Test Harness using Lambda

In addition to testing Lambda functions through the AWS console, a separate Lambda function was created for unit testing Lambda functions. The test function runs the specified Lambda function using the given `event` object. It then saves the result of the test in a DynamoDB table for later analysis.

For example, the following is a result recorded in the DynamoDB table, after running the test against the `readUser` Lambda function:

```
{
  "iteration": 0,
  "passed": true,
  "result": {
    "Item":{"name":"DiveSafe","id":"divesafe@gmail.com","phone":"0498830653"}
  },
  "testId": "readUser"
}
```

The benefit of using this approach is that as the test harness is just another Lambda function and therefore it requires no resource provisioning. In addition, any information necessary to the function under test can be recorded in the DynamoDB table.

4.4.2 Analysis

Implementing the server-less architecture using AWS micro services proved to be a challenging task, especially in the initial phase. This was partially due to the unavailability of many resources on the relevant topics, at the time of writing this document.

Prior to selecting the server less architecture to be used in the project, a more conventional solution that involved an API to be hosted on an Amazon EC2 server was developed. The basic API was built using NodeJS, Express framework and MongoDB. However, the architecture was later changed in favor of the server less architecture, which takes full advantage of the services offered by AWS.

4.4.2.1 Learning outcome

The developer of this project had previous experience with developing web and mobile applications. The developer had also been involved, to some degree, in architecting applications in the past. However, the developer had no considerable prior experience with either cloud computing or Android NFC. This posed a challenge during the development, especially in the initial stages.

Perhaps one of the more challenging aspects of this project, from the developer's point of view, was designing the architecture in a meaningful way that took advantage of the powerful services provided by AWS.

The project provided the developer with a solid knowledge base in cloud computing as well as developing for NFC-enabled devices.

4.4.2.2 Future development

While the current implementation offers the basic functionality needed for the DiveSafe solution and it integrates different platforms in a scalable fashion, there are still additions that may prove helpful in the future. These include:

- Offline Tag registration: The mobile application should allow shops to register tags even when internet access is not available. This could be achieved by saving Tags to local storage and committing them when internet is available.

- Flexible fields in the tag: Currently, the mobile application allows specific fields to be written to the tag. This could be changed so that the user is able to dynamically decide what information to store on the tag.
- iOS application: As of writing this document, Apple does not provide APIs to develop NFC based applications. Developing the DiveSafe mobile application for iOS devices, when possible, will indeed increase the user base.

On top of the technologies used in this project, Amazon Web Services (AWS) continues to offer other services that can be added in order to further enhance the project's usability. This, for example, includes Simple Notification Service(SNS).

5 CONCLUSION

Even though this project achieves the objectives set forth in the beginning, there are still additions needed to make the DiveSafe project into an industry ready solution. However, these largely include modifications dependent on business related decisions and are not necessarily significant from a software development point of view.

The server-less architecture using micro services is a relatively new concept and not a lot of resources are readily available. Therefore, this document promises to be a good reference for future projects being implemented using these technologies. The project offers a good overview of the steps necessary to integrate different platforms with Amazon Web Services (AWS) and typical challenges that newcomers may face.

Although the implementation and design instructions documented in this thesis target the specific objectives of this project, they can easily be tailored for various other purposes. Specifically, the server-less architecture is a low cost and a highly scalable approach that can be employed for a variety of needs. This, coupled with the fact that AWS offers a free trial of 12 months as of writing this document, provides a good platform for startups and small companies.

REFERENCES

/1/ Thomas Wu. 2001. An introduction to object-oriented programming with Java. 2nd edition. McGraw-Hill

/2/ AngularJS website. Last Accessed 21.12.2015

https://angularjs.org/

/3/ Sequence Diagrams: And Agile Introduction. Last Accessed 15.12.2015

http://www.agilemodeling.com/artifacts/sequenceDiagram.htm

/4/ Roy Fielding's Dissertation. Last Accessed 20.12.2015 https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm

/5/ JUnit website. Last accessed 22.12.2015

http://junit.org

/6/ Kyle Simpson. 2015. You don't know JS. O'Reilly Media Inc.

/7/ NFC Basics. Last Accessed 12.12.2015

http://developer.android.com/guide/topics/connectivity/nfc/nfc.html

/8/ How NFC works: Last Accessed 23.12.2015

http://www.nearfieldcommunication.org/how-it-works.html

/9/ NFC Forum. 2006. NDEF Technical Specification. NFC Forum Inc.

http://members.nfc-forum.org/specs/spec_list/

/10/ What's the difference between RFID and NFC? Last Accessed 23.12.2015

http://electronics.howstuffworks.com/difference-between-rfid-and-nfc.htm

/11/ John Papa. Angular 1 Style Guide. Last accessed 10.03.2016 https://github.com/johnpapa/angular-styleguide/tree/master/a1

/12/ NodeJS. Last accessed 16.12.2015

https://nodejs.org/en/

/13/ ExpressJS. Last accessed 16.12.2015

http://expressjs.com

/14/ Gulp. Last accessed 20.12.2015

http://gulpjs.com/

/15/ HTTP Access Control. Last accessed 21.03.2016

https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS

/16/ What is AWS? Last accessed 22.01.2016

https://aws.amazon.com/what-is-aws/

/17/ AWS API Gateway Developer guide. Last accessed 27.01.2016

http://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html

/18/ AngularUI. Last accessed 26.01.2016

https://angular-ui.github.io/

/19/ AWS Cloud Products. Last accessed 28.01.2016

https://aws.amazon.com/products/

/20/ What is IAM? Last accessed 27.01.2016

http://docs.aws.amazon.com/IAM/latest/UserGuide/introduction.html

/21/ AWS Lambda. Last accessed 15.01.2016 http://docs.aws.amazon.com/lambda/latest/dg/welcome.html

/22/ Reto Meier. 2012. Professional Android™ 4 Application Development. John Wiley & Sons, Inc.

/23/ AWS Cognito. Last accessed 28.01.2016

http://docs.aws.amazon.com/cognito/devguide/

/24/ AWS S3. Last accessed 23.03.2016

https://aws.amazon.com/s3/

/25/ Android Tag source. Last accessed 08.03.2016

https://android.googlesource.com/platform/frameworks/base.git/+/4049f9d00a86f848d42d2429068496b31a6795ad/core/java/android/nfc/Tag.java

/26/ Material design specification. Last accessed 10.02.2016

https://www.google.com/design/spec/material-design/introduction.html

/27/ MongoDB for NodeJS. Last accessed 29.03.2016

https://docs.mongodb.org/getting-started/node/