Supplement tracker: Development of a web application utilizing self-reported user data

Hans Wiksten

Degree Thesis
Information Analytics
2019
Sammandrag:


Nyckelord: backend, frontend, datalagring, moln, lokalt-körd, amazon

Sidantal: 41
Språk: Engelska
Datum för godkännande:
Abstract:

In the recent years a lot of tools and frameworks have been introduced to solve different aspects of application development, both on the frontend with libraries and frameworks and on the backend with many cloud providers, data storage solutions and many more. The purpose for this study was to illustrate the process of deciding what technologies to choose. The focus of this study will be on the backend. An example application was created. The application chosen was described and the requirements for it were defined. Since the application was data centric, cloud vs. on-premise was investigated and cloud providers and services were investigated. Data storage solutions were investigated and evaluated to identify which one was the best fit for the application. The frontend aspect was briefly gone through to give a better image of the whole application design. A planning chapter was included to provide an image of how to apply the technologies chosen in the previous chapters to construct a plan for the following chapter. An implementation chapter that utilized the planning to illustrate what was achieved in a more practical way from the whole process. The study will conclude with a discussion about what was achieved and what future studies should be considered.
## CONTENTS

1 Introduction .................................................................................................................. 6
  1.1 Purpose ..................................................................................................................... 6
  1.2 Structure ................................................................................................................... 7
  1.3 Limitations ............................................................................................................... 7

2 Application ..................................................................................................................... 8
  2.1 Requirements .......................................................................................................... 9
  2.2 Technical requirements ......................................................................................... 11

3 Design choices .............................................................................................................. 12
  3.1 Cloud vs. On-premise ........................................................................................... 12
  3.2 Backend .................................................................................................................. 17
    3.2.1 Data Storage .................................................................................................... 17
    3.2.2 Database ......................................................................................................... 17
    3.2.3 Relational database management system ...................................................... 17
    3.2.4 NoSQL ............................................................................................................. 18
    3.2.5 Data Warehouse .............................................................................................. 18
    3.2.6 Data Lake ........................................................................................................ 19
    3.2.7 Chosen solution .............................................................................................. 20
  3.3 Frontend ................................................................................................................... 21
    3.3.1 Why TypeScript? ............................................................................................ 21

4 Plan ............................................................................................................................... 23
  4.1 Visual mock-up ........................................................................................................ 26

5 Implementation ........................................................................................................... 28
  5.1 DynamoDB .............................................................................................................. 28
  5.2 Lambda .................................................................................................................... 29
  5.3 API .......................................................................................................................... 30
    5.3.1 Event Scheduler ............................................................................................. 31

6 Conclusion ................................................................................................................... 32
  6.1 Future studies ......................................................................................................... 33

7 Svensk sammanfattning ............................................................................................... 34

8 References .................................................................................................................... 37
Figures

Figure 1, Average Time Spent per day with Mobile Internet among US Mobile Users, eMarketers 2018 ................................................................. 10
Figure 2, The Magic Quadrant for Cloud IaaS, Gartner 2018................................. 14
Figure 3, Enterprise Public Cloud Adoption, RightScale 2018............................... 15
Figure 4, Data Warehouse vs Database, Amazon 2019 .......................................... 19
Figure 5, Cloud Architecture ............................................................................. 23
Figure 6, DynamoDB capacity resource window ............................................... 24
Figure 7, Amazon CloudWatch, Amazon 2019 ................................................ 25
Figure 8, Amazon API Gateway, Amazon 2019 ................................................ 25
Figure 9 Login page mock-up .......................................................................... 26
Figure 10, Questionnaire page mock-up ............................................................. 27
Figure 11, Report page mock-up ....................................................................... 27
Figure 12, Example row in a DynamoDB Table, Questionnaire table ...................... 28
Figure 13, Example of a lambda function, Questionnaire submit function ............ 30
Figure 14, API Endpoint resource window ......................................................... 31
Figure 15, CloudWatch event resource .............................................................. 31
1 INTRODUCTION

In the recent years, a lot of frameworks and tools have been introduced to solve a multitude of tasks related to developing web applications.

On the frontend side, new languages such as TypeScript, premade libraries and frameworks like React, Vue, Angular and many more offer both functionality and visualization. On the backend side, many cloud providers offer complete infrastructures filled with services for different kinds of purposes, on-premise solutions, different kinds of data storage systems addressing a variety of issues and development challenges and many more.

This presents a challenge for developers starting new projects as to which solutions one should select.

1.1 Purpose

The aim of this thesis is to illustrate the selection process and comparison of which tools and frameworks to choose based on the web application you are trying to make. The application that will be created will be a supplement tracker application that utilizes user data which will be described in the next chapter. As the application will be focused around data, an in-depth review of data storages will be conducted, since this is critical for the application.

The main purpose of this study is to try to give insight into how one would compare and choose the backend solution stack for the application, in this case a small scale progressive\(^1\) web application. The study will also investigate if one should use a cloud provider or not, and which one to choose based on research material available. Front-end technologies will be partially researched, to discover what sort of requirements an application like this has and what tools would be optimal to use for efficiency and usability.

\(^1\) A progressive web application is a web application that uses modern web capabilities to deliver an app-like experience, changing from web pages in the browser tab to top-level applications. (Alex Russel 2015)
1.2 Structure

The first chapter will go through the introduction, what purpose and limitations the thesis will have. Chapter two goes into more detail of the overall idea of the application and requirements that should be met. The third chapter, goes into more detail of what was considered, what tools were selected and which services were used for the application. The thesis focuses on the backend and data related aspects. The fourth, presents a brief planning phase, where the general architecture was decided and what will be implemented in the following chapters. The fifth chapter, illustrates how the plan was implemented, as well as illustrating details about results of the implementations with examples and descriptions of each major component. In the sixth chapter, I will present what was done in the study, what conclusions were reached and what future studies could be conducted.

1.3 Limitations

This study only focuses on the tools relevant to the application and as such will not include every possible framework and tool out there. The study will not implement a solution of each tool and framework mentioned, and as such, will draw conclusions from reviewing documentation, case studies and literature of the tools and frameworks. The time limitation will also not allow any thorough testing to be done as this would require feedback from users, possibly also some analysis based on it. The study won’t address any non-technical part of the application that isn’t directly relevant to the design process of the application, as the thesis is intended to be focused on software development. The study will also mainly focus on the backend of the application and as such, the implementation chapter will not describe the frontend.
2 APPLICATION

In this chapter the reasons and requirements of the app will be discussed.

The application that will be implemented for this thesis should be able to illustrate the design process with a tangible result. The app is a self-reported web application tracking the users supplement usage. This application was made in collaboration with a fellow student, as he was writing about Continuous Integration and Continuous Delivery and wanted a use case. There are a few reasons behind the choice of a supplement tracking.

1) The increasing number of supplementation usage, according to (Council for Responsible Nutrition, 2017) their annual survey reported that already around 76% of US adults consumed dietary supplements. However it is not clear if there is a benefit to the consumption.

2) Eliseo and Guallar showcases a few articles where no benefit was found from taking supplements (Eliseo, Guallar et al, 2013). This indicates that some people are taking supplements for no real reason and could have a benefit from gaining insights if the consumption actually helps them in anyway or not.

3) There exists a lot of tracking applications already such as fitnesspal and alike, but their focus isn’t on giving feedback to the user on the usage itself of the supplement but rather as a form of data collection for the user and remainder to take their supplements. There is a clear missed potential here as the users might not be fully aware if there is a benefit to what they take, rather they just write daily to see if they took it or not.

The purpose of this application is to give meaningful feedback to the user based on the data received by the user, this would significantly increase the usefulness of supplement tracking applications for the users, as they would not only serve as a journal to record when one took the supplements, but also to gain insight if they benefit from them. For instance, if a user starts to take magnesium supplements to combat an issue that is generally found to be relieved by magnesium, the application would first for a set period ask the user to answer questions daily. This would function as a baseline period, which is compared with a period after, where the user doesn’t take any supplements and answers the questions. After the second period ends the results will be compared to the baseline, a report will be made out of this comparison. This report will then be accessible to the user to see what changes have happened.
2.1 Requirements

Traditionally supplement tracking applications rely on the user downloading a mobile application; this limits the accessibility for users as some might not always have access to a mobile device or have restrictions for downloading and installing third party applications. What was decided for this application to allow more accessibility was a native progressive web application, this would allow the user to answer the questions through any device.

As illustrated by Kleiner Perkins Internet Trends (2018, page 11), users spend more and more of their day on their phone, but also a sizable amount on their computer. This would make a web application ideal since any of the devices can utilize it, but also would reduce development time and cost since there wouldn’t be a need to create an application for the iOS operative system and the android operative system.

A mobile application should still be considered in the future as even though there is a larger variety of devices capable of using the app, one should also take into consideration how users generally tend to use it from the device. As illustrated in figure 1, users spend a significant amount of their time in-app vs. on the web. If future user analysis would indicate that the majority of the userbase were mobile users, it would be smart to implement a mobile application also of it as mobile users tend to favour in-app vs. on the web.

Also the native capability should be taken into consideration, as this would allow for an ease of writing a mobile application and at the same time share a considerable amount of the code base with the web application, recent frameworks such as React Native or NativeScript allow for such a design model.
Average Time Spent per Day with Mobile Internet Among US Mobile Users*, In-App vs. Mobile Web, 2012-2018

Note: ages 18+; time spent with each device includes all time spent with that device, regardless of multitasking; for example, 1 hour of multitasking on apps while on a mobile website is counted as 1 hour for in-app and 1 hour for mobile web; *smartphone and tablet users
Source: eMarketer, April 2016

Figure 1, Average Time Spent per day with Mobile Internet among US Mobile Users, eMarketers 2018
2.2 Technical requirements

As the application relies on storing and using user data, data storage systems are required. An authentication data storage would be required to be able to keep track of users by assigning user id’s on the reports, and questionnaires answered by the user. Also to provide a secure way for them to send and access their data. The primary function for the storage is login and registration.

Next would be the answer data itself, the reports and questionnaires could be mixed in the same storage system, but this might not be ideal as usage frequencies for these differ. Questionnaires would require once a day per user access, and as such have a daily frequency, reports would be generated much more infrequently, and users getting them from the database would be rather infrequent also. As such, different pricing model and implementations should be considered for them as it could save monthly costs or overall improve the efficiency of the backline.

A medium between the client and data storages would be required, for instance an API could be good to utilize, this will be discussed further in the next chapter. Also some form of periodical system should be decided on to generate reports.

The visualization aspect of the application should be designed with the consideration of simplicity and efficiency. The task of answering questions daily will get quickly dull and if it’s not made efficient the user might get discouraged of entering regularly in the data. (Garret, Renee et al 2017) illustrates this with their literature review on website design and user engagement, key points were described as to what often was seen as good features, simplicity was one of them and an “Uncluttered layout” was mentioned several times. Efficiency was also mentioned in different groups, for instance in graphical representation, a positive factor was “Minimizing loading time for visual elements”. As such, it should be a fast and efficient design rather than information cluttered.
3 DESIGN CHOICES

In this chapter the study will compare different tools and available resources that could be used for implementing the application, and also describe which one was chosen and why. The measurements that will be used to gauge on what is the best fit for the application are the following:

- Cost. Different providers and services have different cost models and as therefore these should be compared to see which is the most suitable for the application in mind.
- Development time and quality. The correct tools should be chosen to fit the application, but it should also be taken into consideration that development time is a cost factor, and as such, instead of choosing to program everything in for instance assembly which could give a slight speed increase in specific things, more appropriate languages and frameworks should be chosen.
- Fitting the requirement criteria of the application. Even if a service would be very cost effective, if it lacks a crucial part of the requirements, then it is better to just choose a less cost effective one that fulfils the requirements.

3.1 Cloud vs. On-premise

A cloud-based approach was chosen over on-premise for a few reasons.
First of all cost, (Greg Deckler 2016) wrote an analysis of on-premise vs. cloud, after reviewing what he found to be realistic factors he came to the conclusion that the cloud would be approximately 30% cheaper. (Sherweb 2016) also conducted a study on this and found much higher results of 79% cheaper, although they factored in the cost of a system administrator to manage it with a 60 000$ annual salary, this might have skewed their results, either way, they bring up an important factor that employment salaries of people managing the hardware will also drain a lot of resources. If you have on-premise, you are in charge of all the hardware, and as such, hardware maintenance and upgrading could become costly. Nick Kartman wrote about this and expressed similar concerns (Nick Kartman, 2018). Also other events such as hardware breaking or malfunctioning would become a concern. Secondly there is the matter of security, and as seen in the study
by (Schneider Electric 2017), already 78% of IT professionals feel confident that the cloud is secure. As the cloud provider would offer the service and be in charge of it, this might be more reliable and less costly than to have your own security experts. Keeping everything up to date and properly functioning requires people with expertise in implementing the solutions according to their standards.

Amazon was chosen as the cloud provider for this application, others were considered such as Google and Microsoft, as can be seen in Figure 2, Amazon (AWS), Microsoft (Azure) and Google (Google Cloud) scored highly on Gartner’s magic quadrant.

Ability to execute was particularly important as this describes their ability to perform based on a few key metrics. How many products and services they provide, pricing, overall viability which is essentially how successful they were based on their revenue trends, and investment strategies, track record, customer experience, marketing execution, and operations which is essentially how well they were able to meet their goals and commitments.

Completeness of vision had a similar structure of certain key metrics with a different goal in mind, this was more about the financial and marketing aspect of how they’ve performed. Marketing and sales strategy, how well they understood the market, business models, geographical strategy, vertical/industry strategy, product strategy and innovation.

The niche players were not considered due to the following reasons. As they did not score well on the Ability to execute, this would be a concern as this could indicate poor track records and similar. Another reason was because of the use cases described in the (Gartner 2018) report. Strengths and weaknesses for all the cloud providers were described, the niche players were highly use case specific which didn’t align with the requirements of the application, for instance Alibaba Cloud had a recommended use case of “Digital business workloads for customers that are based in China, or need to locate cloud infrastructure in China.” As this application isn’t based in China nor has yet any plans to make a translated version for Chinese users, this would not be a good pick.
Amazon was chosen due to the following reasons. As can be seen in figure 3, Amazon and Microsoft were the most used and had the most running applications on, this could indicate that a lot of companies find them useful and trustworthy as they have the largest portion of the share and because of how well they performed in the magic quadrant.

Why Amazon over Microsoft? Both of them have a large portfolio of offerings in their cloud services. As can be seen in their documentations, for instance, Microsoft offers API Apps, Azure SQL Database that are billed by an hourly usage rate with tiered rates. Microsoft also has serverless capability in the forms of Azure Functions that could allow on-cloud processing for convenient development of custom API functionality and database handling (Microsoft Azure 2019a, b, c). Amazon offers similar products, they have their own easy to deploy API Gateway service, SQL databases in the forms of multiple different ones, Postgres, MariaDB and more and NoSQL in the form of DynamoDB. Serverless
capability is also provided through Lambda functions, which will be explained in the next chapter.

Amazon's pricing differs in that the services are not on an hourly rate but on a usage rate, there is also a free tier usage rate which is rather large. For instance amazons API allows the use of one million requests per month without costing anything, this would allow for applications such as this to exist with very low if any costs at the start. It would also allow for rather low costs as one starts to have more requests since it’s billed on the amount of requests (Amazon 2019a,b,c,e). One could also design the applications with this in mind to send payloads of requests in the future to mitigate costs rather than having a constant hourly runtime cost such as Microsoft has.

Amazons pricing model would also be optimal for seasonal applications\(^2\) as the costs would be non-existing in the off seasons since requests are low and moderate in the seasons when a large number of requests start to come in, this would save a lot of money as

\(^2\) Certain supplements like Vitamin-D are often recommended in seasonal frequencies. (National Institute of Health 2018) describes that sun exposure isn’t enough during seasons like winter, this could increase the usage of the application as more people take supplements during that time and could have more interest in an application such as this.
the average costs would drastically drop as opposed to the model of Azure where you pay constantly the same rate.

Also, if we look at Amazon’s recommended use cases, (Gartner 2018) describes it as “All use cases that run well in a virtualized environment.” As this application does not have any requirements that disallow virtualized environments, it would fit well for it. Microsoft’s recommended use case as described in the report is “All use cases that run well in a virtualized environment, particularly for Microsoft-centric organizations.” This could be a good or concerning fit depending on how significant the Microsoft-centric component of that statement is. (Tushar Agarwal 2019) argues that you should choose Microsoft mainly if you are already using their products, he also points out that the large part of their userbase is those with existing Microsoft relationships. (Amit Ashwini 2018) also raises concerns with that unless you are planning to run everything on windows servers, it might not be the best choice.

This would indicate that if one would decide to go with Microsoft, one would be inclined to use Microsoft centric products, this could be a concern in the future as this application does not require Microsoft centric products and could be a limiting factor if for instance some of the offerings aren’t compatible with new offerings another provider would provide that fits to the application.

(Tushar Agarwal 2019) also describes that that Amazon would be the choice if one would require AI services, as this application is about data and analysing the data, this could have great potential in the future for more sophisticated analysis of users supplement behaviour by utilizing different AI approaches.

(Gartner 2018) also describes in the report that one of Amazon’s strengths is that it is not only considered the “safe choice” in the market but also that it is the most mature with a strong track record.
3.2 Backend

This chapter will go through the design choices of the backend perspective, the technical parts that the users won’t see that handles the data and dataflows.

3.2.1 Data Storage

As this application is very data centric, relying on being able to store, manage and utilize data from users, appropriate data storage mediums should be chosen. Currently the main method to do this would be to utilize databases management systems in some form, as Amazon was decided to be the cloud provider, its offerings will be considered.

Different formats of database management systems are available, relational database management system (RDBS), data warehouses, databases, data lakes and NoSQL. This chapter will briefly go through them, what they are, and which was chosen for this application.

3.2.2 Database

As described in (Techopedia 2019a), Databases are in the general sense electronic systems that manage over a collection of organized data. Allowing ease of access, manipulation and updating of the data.

3.2.3 Relational database management system

A Relational database management system is one of the most common database management systems used today (Software testing help 2019). As defined in (Techopedia 2019b), it is essentially a database design that relies on a relational model. To illustrate what this means, you could have two database tables, products and orders. Each product row would have a “product id” and each order row would have a “order id” and by utilizing these you could form a relation between them to be able to extract for instance more detailed information about products that were in an order. As (Zhan Wei Wu 2018) describes, the use case for this type of database management system is mainly when you have structured
data, require data security following the ACID model, and when one requires complex data querying.

Amazon provides a few options for RDMS, MySQL, PostgreSQL, MariaDB, Oracle, Microsoft SQL Server and their own Amazon Aurora (Amazon 2019b).

### 3.2.4 NoSQL

Recently a new type of database management system has been adopted by many known as NoSQL, as described in (Techopedia 2019c), it is essentially a form of database management system that does not rely on traditional SQL, but rather through varying data models utilizing key values, document, columnar and many more. Amit Ashwini (2017) describes the benefits and use cases for NoSQL, the main reason to use it would be when you have a lot of unstructured data, this allows for very flexible data insertion as NoSQL doesn’t rely on specific data structures. It’s also mentioned that this type of data management is more optimal for scaling and Big Data applications, and as such, should be the choice for solutions revolving around that.

Amazon provides NoSQL in the form of DynamoDB (Amazon 2019c)

### 3.2.5 Data Warehouse

As described in (Informatica 2019), data warehouses are in essence the idea of having one centralized repository of information that is utilized in various large-scale analyses. Informatica also describes how this would differentiate from a regular database management system, in essence data warehouses specialize in data aggregation while RDMS specialize in maintaining momentary accuracy in data and being able to have a large volume of transactions running. As can be seen in Amazons comparison in Figure 4, data warehouses serve a much different purpose than a transactional database such as RDMS, and should be evaluated based on those criteria if one should utilize it or not.
3.2.6 Data Lake

As defined by (Amazon 2019f), a data lake is essentially a centralized repository for information, data lakes differ from data warehouses in that you can send all your data in, no matter what form, unstructured or structured. (Limor Wainstein 2019) discusses when to utilize a data lake approach, it is described that three major points should be considered when evaluating between a data warehouse and a data lake as they are similar in their purpose.

Data Structure, a data lake would be better to utilize if one would require to initially store all relevant data without caring about their source and structure as opposed to data warehouses requiring it. Adaptability, a data lake would allow for quick and efficient organizational changes as it doesn’t care what data comes in, meanwhile the data warehouse would require complex changes. Performance, when deciding on data lakes, performance might be a concern as they are not optimized for processing as opposed to data warehouses.

Amazon provides a quick start into deploying data lakes through their data lake solution documentation (Amazon 2019d)
3.2.7 Chosen solution

What was chosen for this application was Amazon’s implementation of NoSQL, DynamoDB.

For the authentication service as specified in the requirements, the pricing model of DynamoDB would serve as an excellent choice as one can store 25 GB of data storage, have 2.5 million stream read requests and 1GB of data transfers out per month for free (Amazon 2019c). Register and login requests would be very infrequent from the user. A login request would only be required a few times a day or less and a register request would only be once per user, meaning that there will be a lot of downtime when nothing is happening. Instead of paying monthly for only using a small portion of the time, you would pay for the requests. Data lake or data warehouse model should not be considered as authentication data wouldn’t need any form of analytical processing.

For the transactable questionnaire data, DynamoDB was also chosen. Users will only need to send in one request per day from the questionnaire and the pricing model is highly favourable for this kind of frequency. For the report data the frequency will be around the same if not less, generation reports would have a very infrequent amount while the requesting of made reports would be moderate. DynamoDB also functions very well since it doesn’t rely on relational models, and as such the data fed in can be very varying. The flexibility NoSQL provides would reduce development time and costs on the backend since the database schema wouldn’t have to be updated as frequently and would be more lenient to accept data in different forms. As this application deals with supplement usage data, new supplements emerge on the market frequently, many which contain different combinations of compounds that might require different kinds of questions. Instead of having to devote a lot of development time on managing the data structures for each specific case, one could just utilize the flexibility of NoSQL.

A few other of amazons services were also chosen to be part of the solution. An API solution called API Gateway which amazon provides that allows users to send in requests. Lambda functions, these are essentially a part of a serverless architecture, they are functions usually made with a scripting language that perform specific tasks in the backend, for instance a user could send in one days questionnaire data through the API, the API
forwards this to the Lambda function, the lambda function runs a script that checks if the data is valid, for instance no missing values like who sent it and so on, and then submits it to the appropriate DynamoDB table. Cloudwatch which is a system that allows you to submit timed events, for instance the report generation could be decided to be made every week at a certain day, Cloudwatch would handle submitting the request to execute the Lambda function that performs this task at a weekly interval.

In future considerations, if the application would grow large and popular, the transactable database should be converted or an extra layer should be added to form a data lake like structure. For now the DynamoDB will be sufficient as it would maintain low costs, but also allow for easy scaling and development in the future.

### 3.3 Frontend

This chapter will go briefly through the design choices of the frontend perspective, the visual parts that the user will be presented with.

The language and frameworks that were selected for this application were TypeScript, NativeScript, and Angular.

#### 3.3.1 Why TypeScript?

Mainly TypeScript (Which is essentially a superset of JavaScript) and JavaScript were considered, as these both fill the requirements of the application, especially since they have the capacity of building Native web applications, JavaScript having tools such as React Native and TypeScript having Angular with NativeScript to accomplish this.

The reason for choosing TypeScript was because of a few reasons. As written by (S. Somasegar 2012), TypeScript has a few benefits over JavaScript, for instance class declarations and modularity allowing much easier object oriented programming, Somasegar argues that this is important as object oriented programming techniques have been shown as effective means of building large systems. Another reasoning is static typing similar to how C# or Java has. (Ayo Alfonso 2017) describes that statically typed languages are
often faster and would encourage better programming habits as developers are more accountable. (Guru99 2019) Also argues for that TypeScript would be more maintainable and offer better productivity for developers.

Another reason is also efficiency, as can be seen in (Jacek Schae, 2019), when comparing TypeScript with Angular vs. JavaScript with React and Redux, TypeScript performs slightly better both in application size and performance.
4 PLAN

This chapter will go through the planning phase of the application. A planning will be done utilizing mock-up tools to create flowcharts and utilizing figures of different components of what Amazon provides. This will allow the development to have a clear goal to strive for.

The planning was started by creating a mock-up of the cloud architecture as seen in Figure 6.

*Figure 5, Cloud Architecture*

The client would first connect through a device and send a request to get a connection to the website. The website would be hosted in an S3 Bucket that would contain all the frontend programming. The website will then have different connectors to the API that will utilize an approach of starting Lambda functions. They will get and insert data into appropriate DynamoDB’s. There will also be a periodic function starting Lambda functions to generate reports through the usage of Amazon’s CloudWatch Events.
The reason for splitting up questionnaire data and report data was because of how DynamoDB tables function, they have a primary key which remains always unique, but otherwise they have very flexible structures that are harder to keep unique, it would be better to simply separate the two to utilize the inherent uniqueness in the ID fields. This also allows for better control in the pricing strategy as you could give different read/write capacities for them since they have different frequencies. As can be seen in Figure 6, you can assign different kinds of capacity units, these basically dictate how many requests per second you are able to handle.

![Figure 6, DynamoDB capacity resource window](image)

---

24
As can be seen in Figure 7, CloudWatch is a service that provides different services such as monitoring, event scheduling, collection and analysis. The part that will be used for the application is the event scheduling aspect.

The API would be made through the utilization of Amazon’s service Amazon API Gateway. As can be seen in Figure 8, it would serve as a medium to connect the client to Lambda and DynamoDB.
4.1 Visual mock-up

A sketch mock-up was also conducted to give a clearer image of how the application would look like, the login page, questionnaire page and report pages were made by utilizing a mock-up tool called Moqups.

Figure 9 Login page mock-up
Figure 10. Questionnaire page mock-up

Figure 11. Report page mock-up
5 IMPLEMENTATION

This chapter will go through the implementation of the backend portion of the application, it will not go into detail of how every single line of code was made, but it will provide a general idea of what was done.

5.1 DynamoDB

Three tables were made, one for login and registration purposes, one for questionnaires and one for reports. An example of how a row in DynamoDB looks like can be found in figure 12.

![Example row in a DynamoDB Table, Questionnaire table.](image)

The login and registration table was made, a primary key “email” was created to confirm that emails were unique, “password”, “userID” that is randomly generated with a hashing method and “creationDate”.

The questionnaire table was made, a primary key “qID” was made to have a unique identifier for the records, “userID” to know which user sent the data, “baseline” to check if the data was supposed to be used as the initial week where no supplements were used or not, “submissionDate” to be able to query effectively through the utilization of selecting only rows from a certain date range, for instance if a report is generated on a weekly basis, you could simply take the current date, get the date seven days ago, and only take results from this time period to process, and “questionData” which contains the answers to the questions.

The reports table was made with a similar structure, a “rID” primary key was assigned to keep unique IDs, “userID” to keep track of which user the report is about, “baseline”
which serves the same purpose as in the previous table, “creationDate” to be able to display and analyse better different periods of time and then summary fields of the report for the questionnaire data.

5.2 Lambda

Five Lambda functions were created. Node.js v8.0 was used as the runtime provider from the available options. An example of how a function looks like can be seen in Figure 13.

1. Login authentication function that checks if the user exists and if the user’s username and password matches to the values stored in the authentication database.
2. Register function that inserts the submitted data in-case there are no missing values.
3. Questionnaire submission function that inserts the data after checking that there are no missing values such as userID
4. Report get function that returns all the reports associated with the userID logged in currently.
5. Report generation function that takes N days back in time questionnaires, compiles a report out of it, and inserts it into the reports DynamoDB.
As defined in the plan, Amazon API Gateway was utilized to construct API connections. First an API endpoint was created that functions as a container for all the API resources and as a root destination to connect to from the client. In the endpoint was then created four paths with POST and GET functionality that were linked to each of their respective Lambda functions that then interact with the databases and either get data to the client or insert data from the client. In Figure 14 you can see how the implementation looks like in Amazon’s API Gateway resource window.

5.3 API
5.3.1 Event Scheduler

In CloudWatch the periodic event Lambda starter was created, it was defined as 7 days which is easily editable depending on future requirements and was linked to the Lambda function that handles the generation of reports. The event resource can be seen in Figure 15.
6 CONCLUSION

Deciding on what tools and services to use can be very challenging at first if one does not know what to look for or how to go at it. This study gave an illustration of the process utilizing a case study of an application that was small scale and data centric. It went through the process of first defining what the application was and what it required, this gave a clearer goal towards what to investigate further in. First of all one would require to have a place for servers and services to be hosted on, on-premise vs. cloud was compared and as cloud was generally the cheaper solution, it was investigated further. From the current cloud providers, the most suitable was selected after reviewing use cases and available services by them, also pricing was taken into consideration as different pricing models were utilized by different providers. As for this case, Amazon was the most favourable and as such was chosen.

Since the application was data centric, data storage was an important aspect to further investigate, the most commonly used were defined and looked into. Amazon had multiple offerings of them and a choice was made on what was the best fit for this use case, DynamoDB and NoSQL was the favorable option and as such was chosen. Frontend was also investigated based on the requirements defined and was investigated as to which languages and tools would meet the criteria. TypeScript was eventually found to be the best fit as it would meet all the goals and performed well in tasks that were sought after. Then a chapter going through the planning was shown to illustrate how the found information was utilized and finally a chapter of the backend was implemented and illustrated of what was accomplished. In conclusion a cheap and functional backend was achieved that meets all the requirements defined for the application. The frontend was not gone into detail as the focus was on the backend for this thesis. Also because of the time constraint not allowing for any testing or feedback on it meaning that if it was a successful implementation or not is hard to measure.
6.1 Future studies

As this study mainly focused on the backend perspective and was on a time limit constraint that prevented testing, further studies on the backend would be implementing a solution on alternative cloud providers and measure differences in how in the real world they would perform rather than only on paper. On the frontend to investigate more closely on user experience and how one would design and implement it on a more in-depth level would be interesting to conduct as this was outside of the scope of this thesis. Also more comparative testing would be interesting to conduct to determine if the stacks chosen truly were the best fit for the application in practice also.

port data. Varför man skiljer dem är på basen av säkerhet och användnings frekvens. Se-
dan behövs också ett sätt för användaren att ta kontakt till databasen, t.ex. en API och ett
sätt att göra rapporter vid vissa intervall. Visuella synpunkten förövare också igenom en del
var beskrevs att applikationen borde vara effektiv och enkel att använda för att det blir
snabbt repetitivt att svara dagligen på frågor. Efter det kommer ett kapitel där det först
undersöks om man ska använda moln baserade lösningar eller lokala. Kostnad, utveckl-
ingstid och hur bra det passade med kraven evaluerades. Vad som koms underfund var
att molnet borde vara en del billigare, mindre bekymmer för att du behöver inte uppehålla
hårdvara och var också säker. Sedan undersökes vilken molnleverantör som skulle bli
vald. Först valdes en mängd molnleverantörer baserat på deras position på marknaden,
hur bra de hade i historien klarat av att leverera bra tjänster och produkter och andra
relevanta faktorer. Efter det valdes de bästa ut ur dem och jämfördes, i den här situationen
blev Amazon och Microsoft vald och deras tjänster, priser och hur bra det passar till appli-
kations kraven jämfördes. Amazon blev vald för att deras pris strategi på t.ex. databaser
var bättre för den här applikationen, och för att applikationen inte hade någon krav på att
vara Microsoft produkt centrerad. Sedan forskades backend synpunkten var först gicks
igenom olika datalagrings system, vad de gör och i vilken situation man borde använda
dem. Vad som gicks igenom var RDMS, NoSQL, Data Warehouse och Data Lake. Ama-
zons NoSQL tjänst DynamoDB blev vald för att den passade bäst med pris och flexibili-
et, i framtiden också om applikationen växer tillräckligt kan det hända att en Data Lake
struktur skulle kunna vara bra att implementera. Amazons Lambda blev vald för att tillåta
en serverless struktur som underlätter kommunikationen mellan klienten och databasen,
det är i princip Scripts som man kan starta med olika tjänster som t.ex. API:n för att göra
databas läsande eller skrivande och annat. Efter det undersökes vilket frontend språk och
ramverk som skulle passa bäst till applikationen. TypeScript blev vald för att den har
nativ kapacitet och hade bättre egenskaper än React JS som också fyllde kravet med nativ
kapacitet. Efter det kom ett planerings kapitel som först med hjälp av mock-up illustrerade
arkitekturen som blev vald för applikationen, vad som användes var en S3 Bucket för att
hålla frontend delen när klienten tar kontakt till sidan. DynamoDB för login, frågeformu-
lär och rapport data. Amazon's API Gateway blev använd för API funktionalitet, så att
klienten kan kommunicera med databaserna. Lambda för att kunna behändla data:n, och
Amazon's Cloud Watch för att kunna periodiskt göra rapporter. Efter det gjordes en visuell
mock-up av hur applikationen skulle se ut, login sidan, frågeformulär sidan och rapport sidan gjordes för att ge en bättre bild på hur applikationen skulle se ut.

Sedan gjordes ett implementations kapitel som beskrev vad som hade gjorts. DynamoDB gicks igenom, tre DynamoDB tables var gjorda.
   1) Autentisering, för login systemet så att man kan hålla koll på vem som skickar data och vem som ska få vilka rapporter
   2) Frågeformulären, för att förvara frågeformulär data användarna skickar in
   3) Rapporter, för att förvara de gjorda rapporterna

Efter det gicks igenom hur Lambda hade implementerats, fem funktioner hade gjorts.  
   1) Login autentisering funktion som kollar om användaren finns och om användarnamnet och lösenordet är rätt
   2) Registrerings funktion som sätter in data om det inte finns något som saknas
   3) Frågeformulär funktion som sätter in data om inga värden saknas
   4) Rapport get funktion som returnerar alla rapporter för användaren som var loggad in
   5) Rapport generations funktion som tar under en viss period bakåt i tiden all frågeformulär data och gör en rapport av det.

Sedan gicks API:n igenom var det illustrerades de fyra slutpunkterna som hade gjorts med hjälp av Amazons API Gateway. Till sist också beskrevs Cloud Watch Event Scheduler implementationen som startar varje sjunde dagen rapport generationen.

Studien slutade med en slutsatskapitel som gick igenom vad som hade gjorts och åstadkommts av studien, och vad för framtida studier som skulle kunna göras av det, t.ex. tester som jämför hur det på riktigt sedan fungerar om man skulle implementera samma sak på olika molnleverantörer, det kan finnas gömda kostnader och dylikt som skulle kunna ändra vad som på riktigt är bättre.
8 REFERENCES

Available at:

Amazon (2019a). Amazon API Gateway. Available at:

Amazon (2019c). Amazon DynamoDB. Available at:

Amazon (2019b). Amazon Relational Database Service (RDS). Available at:

Aberdeen Group (2017) Angling for insight in today’s data lake. Available at:

Jacek Schae (2019). A Real-World Comparison of Front-End Frameworks with Benchmarks Available at:

Microsoft Azure (2019a). Azure API Apps. Available at:

Microsoft Azure (2019c). Azure Functions Available at:

Amazon (2019e). AWS Lambda. Available at:


Limor Wainstein (2019). *Data Warehouse or Data Lake? When to Use Each* Available at: https://datafloq.com/read/data-warehouse-or-data-lake-when-to-use-each/5189 [Accessed 10 May 2019]

Council for Responsible Nutrition (2019). *Dietary Supplement Usage Increases, Says New Survey.* Available at:

Eliseo, Guallar et al (2013), Enough Is Enough: Stop Wasting Money on Vitamin and Mineral Supplements. Available at:

Kleiner Perkins (2018) Internet Trends. Available at:

Gartner (2018) Magic Quadrant for Cloud Infrastructure as a Service, Worldwide. Available at:

Techopedia(2019c) NoSQL. Available at:

Alex Russel (2015). Progressive Web Apps: Escaping Tabs Without Losing Our Soul Available at:

Techopedia(2019b) Relational Database Management System (RDBMS). Available at:

Zhan Wei Wu (2018) Relational VS Non-Relational Databases Available at:

Amit Ashwini (2017). *Should You Use NoSQL Or SQL Db Or Both?* Available at: https://medium.com/swlh/should-you-use-nosql-or-sql-db-or-both-349cb26c9add [Accessed 11 May 2019]


