University of Southampton
Faculty of Software Engineering
School of Electronics and Computer Science

Modern Serverless Web Application Architecture

by

Sharang Deepak Gupta

September, 2021

Supervisor: Dr Reza Rezazadeh
Second Examiner: Dr Stuart Boden

Dissertation for the degree of of MSc. Computer Science

Abstract

With multiple tech giants offering various cloud based services at competitive prices, lots of companies are shifting their infrastructure to the cloud. This has resulted in huge gains for companies, as they get to focus on their business rather than infrastructure management. In recent times, a new architectural design pattern has emerged, born out of cloud native. This new architecture offers a pay as per use model, allowing companies to spend only as per their requirements. This architecture also allows for infinite scaling, both scale-up and scale-down. Modern workloads on applications vary drastically, and have become increasingly difficult to predict. This leads to the conundrum of resource management, allocating excessive resources in order to prepare for a surge in visitors causes wastage in terms of unused compute resources, whereas allocating a smaller amount of resources can be even more dangerous, leading to loss of potential customers. Serverless architecture allows for a granular level of control over resources, with auto-scaling, thus ensuring that we pay for only what we use, along with scale-up and scale-down depending on workloads. This project aims to compare the various serverless offerings, their pros and cons, and develop a standardised application using the best of services and practices to ensure that the application is fully scalable, and economical such that it serves as a benchmark and proof of concept for all such applications.

For the purpose of this dissertation, we would be building a recipe sharing website, with multiple features like signing up, OAuth login, recipe creation, update, liking, commenting as well as reporting a recipe if it violates our terms. An admin panel allowing for managing complaints, warning users, banning users and archiving recipes would also be supported. A robust search and filtering mechanism would allow any user, even without authentication to browse recipes, and liking or commenting on any would require signing in. Apart from these, our application would automatically be able to estimate the calories of a recipe based on the ingredient list, for which we use a third party API. Overall, this project aims to provide a pleasant user experience for any consumer wishing to browse recipes or share their creations with the world. The future scope of this project would involve a donations page, allowing for users to donate to various charities via the website and help in the maintenance of the application.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to express my heartfelt gratitude to **Dr. Abdolbaghi Rezazadeh**, my project supervisor, under whose tutelage I have gained a great insight into the project. Our weekly meetings with all the students under professor have been immensely helpful, as it lent to me the motivation and drive to match up with my peers. I also want to thank him for being receptive to any issue I face during the implementation, and for his wholehearted support and advice towards its mitigation.

I would like to thank my sister, **Diya Gupta**, an avid vegan activist, for introducing me to veganism, and for all the recipes she provided for uploading to the website. It is with her help that I am able to create enough content for this project.

Finally, I would like to thank my father **Deepak Gupta** and my aunt **Shashikala Gupta**, who have supported me throughout my life, both morally and financially. It is solely because of them that I got an opportunity to pursue my undergraduate and postgraduate degree.

Chapter 1

Introduction

Most of the time, a software is a combination of different feature sets. In Monolithic Architecture, all the features of a software reside in a single a single file. If any code updates are required, then those updates cannot be accommodated independently. The developer must use the same code base, make the required code changes, and then re-deploy the updated code. So even a single change requires the whole code base to touched and re-deployed.

The above traditional architecture comes with a major caveat, that is, beyond a point, scaling the application becomes disproportionately difficult, with respect to time, human resources, computer resources and storage. This is because of the atomic nature of resources, for example storage. A minor increase in the number of users and consequent requests would require a new server to be provisioned, complete with storage, compute and memory. This new resource would not be utilized to the fullest, until the number of users increases, thus increasing the request load to an optimum amount. The biggest challenge though, is when the load is not balanced or well distributed, and there may be a peak times requiring all the deployed resources, while during the remainder, those provisioned resources lie idle, which is a huge waste of resources.

Another commonly use architecture, called Microservice Architecture, has gained popularity over the recent years, due to its ability to overcome some of the disadvantages of Monolithic Architecture. It involves splitting the application into multiple "microservices", or smaller, fully independent components, which can operate, scale and be maintained, independently of the other components. It can be said to be a mini app, complete with its own database as well. This comes with its own challenges of complexity, and with synchronous update of state throughout the assorted services, although it provides more control over the resources, and mitigates some of the scaling challenges. For example, if a single microservice is running out of storage, there is no need to scale the infrastructure across all the services, we can

get away with scaling just the one we need. This raises an important conundrum though, of how much should the application be divided, as smaller, "microservices" offer a more fine-grained control, but proportionally increases the complexity of the overall application.

This leads us to the most recent, ever evolving and improving Serverless Architecture. A serverless architecture abstracts the underlying infrastructure and offers a "pay as you use" model, wherein you always only pay for the resources you need. If the load increases, it scales automatically, and you pay per request, and thus when the load decreases, you pay much less.

The main aim of this project will be to build a fully scalable architecture, using as many serverless features as possible, such that there is no wasting of resources, and we are billed for exactly what we use. If we see a sudden growth of users towards our platform, we should be able to serve them all, while if there are no visitors, our costs should be minimal.

Key Words: Serverless, pay as you use, scalable architecture, dynamic load, idle resource utilization

1.0.1 Problem Statement

With so many providers and options for these services, it could get daunting for a new startup to decide on their technology stack and architecture, as the difference between success and failure for their venture could depend largely on how well it performs, and how economically. Thus, establishing an efficient, well defined standard for development, right from the selection of service providers to the appropriate tools to be used for the job is the need of the hour.

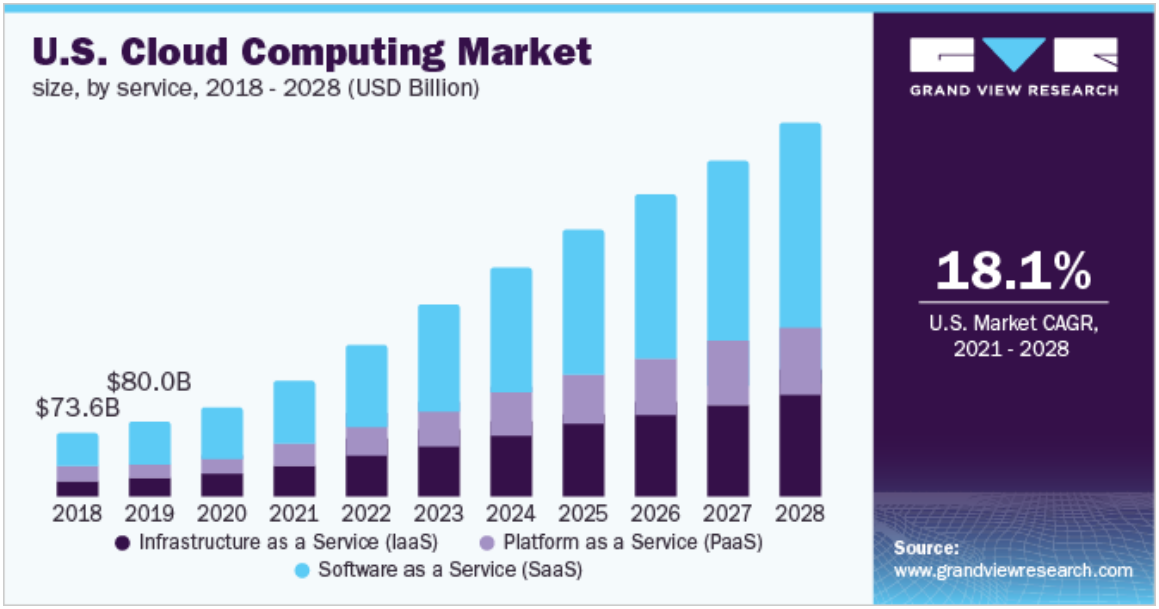
1.0.2 Background

In recent times, the technology stack used for software development has evolved from a simple, standard architecture and limited set of tools to a myriad of options, with each aspect of development and infrastructure offering choices with certain advantages and tradeoffs. These options, while good to have, have raised a conundrum, that is the abundance syndrome. Be it front-end frameworks, UI frameworks, backend frameworks, database engines or software architectures, the appropriate choice of technologies to choose is no longer an easy task.

With the rise of cloud computing, large companies like Amazon, Google and Microsoft have jumped into the fray, initially offering infrastructure as a service, passing onto their customers a share into the economy of scales they are able to leverage, thus reducing the costs of operation. To outbid their competitors, these cloud service providers started offering more managed services, with complexities abstracted behind easy to use APIs, using their experience developing software at scale to provide improved management of common

requirements like, for instance, databases. Eventually, in this competitive environment, these managed services evolved into the serverless architecture we know today.

Figure 1.1: Cloud Computing market size



As we can see from the above market research trends, the cloud computing market has been growing rapidly, and is expected to continue the trajectory in the long term.

Thus, leveraging this boom and evaluating the best provider and services is of utmost importance for any company wishing to stay with the trend in the future, which looks to be heading towards managed cloud services.

1.1 Aims and Objectives

With a view to evaluate cloud computing offerings from various cloud service providers, this project aims to develop a cloud native vegan recipe sharing website infinitely scalable, highly efficient and economical, using the pay for what you use model.

Furthermore, the objectives of the project are explicitly listed below:

1. Building a full stack application, mimicking the most widespread use case for most web applications **(O1)**.
2. Allowing users to signup and login, add recipes, delete their recipes, comment on recipes, report recipes and also browse for recipes, with extensive filtering **(O2)**.
3. Using the most modern front-end technologies, based on React/Next and following best practices **(O3)**.

4. Building a complete serverless backend bit by bit, using serverless offerings from popular cloud services providers like AWS (Amazon Web Services) and GCP (Google Cloud Platform), thus demonstrating a fully scalable app **(O4)**.
5. Exploring modern data querying languages, GraphQl, in line with our main objective of reducing waste, that is querying for exactly what we need **(O5)**.
6. Exploring tools like Amplify to create a proof-of-concept template to make provisioning all the services easy and rapid **(O6)**.

1.2 Project Scope

The project scope encompasses the analysis, design, implementation and evaluation of a cloud-native, web-based system (as stated in the objectives) that is able to mimic the most widespread web applications, like that of a blog **(O1)**. It would allow users to register, login, add recipes, search and filter for recipes, edit their own recipes, as well as comment or like other recipes **(O2)**. They would also be able to report recipes to the admin if they feel that a recipe is not vegan, or is in violation of any rules. This project, however, does not allow for any orders for any of the posted recipes. It is not a food delivering or ecommerce application, we do not facilitate payments to any of the recipe bloggers. This project will use AWS Amplify to provision the resources needed for the backend, a GraphQL API and DynamoDB as the scalable serverless database **(O4, O5, O6)**.

1.3 Dissertation Structure

For the sake of building a coherent dissertation report, this dissertation will follow a logical advancement approach, very similar to the development of the actual application. It begins with an abstract providing the gist of what we are trying to accomplish. The first chapter introduces the project to us, with details such as the aims and objectives of the project. This would immediately be followed by a literature review, where we justify our tech stack and choices with respect to cloud service providers. In the next chapter, we discuss about project management, that is the timeline of the project, schedule and software development methodology.

Chapter 2

Literature Review

2.1 Cloud Computing

Google cloud services and amazon web services have dominated the cloud computing space right from when it started gaining popularity. In August 2020, a report from Gartner[1] named both Google and Amazon in a group of 5 public cloud infrastructure providers that make up 80

2.1.1 Function as a Service

At the heart of serverless, lies FaaS, or Function as a Service. In essence, FaaS allows us to borrow compute time with a millisecond granularity, thus adhering to the pay for only what you use principle of serverless. Being such an essential feature of serverless, all the popular cloud service providers offer this functionality in their own way. The most popular ones used are AWS Lambda functions offered by Amazon and Google Cloud Functions offered by Google. Depending on the use case, one or the other may be more suitable to our requirements. We can compare these two on a number of parameters, with pricing being at the forefront.

A. AWS v/s GCP Pricing comparison

Table 2.1: AWS v/s GCP Pricing comparison

Metric	AWS	GCP
Free Monthly Duration (GB-seconds)	400,000	400,000
Free Monthly Requests	1 Million	2 Million

Cost of Each Additional 1 Million Requests	$0.20	$0.40
Cost of Each Additional 1 GB-second	$0.000016	$0.0000125
Duration granularity	1ms	100ms

From the above pricing comparison, we can infer that while GCP is very generous with its free tier monthly requests, the cost of additional 1 million requests is double of that of AWS, whereas that of an extra 1GB-second is lower. For large scales, where we expect to exceed the free tier over vehemently, the additional cost for each million requests is half of that of GCP for AWS. Depending on the GB-second requirement, the right choice at scale varies on the memory and time requirements. For memory intensive tasks, which could also be time consuming, GCP is the more economical choice, whereas for lighter, numerous tasks, AWS shines. For our use case, we use cloud functions to fetch calorie data from an API depending on the recipe ingredients, and hence, being a light task, AWS could be a more viable choice at scale. However, for a smaller number of users and recipes, within the free tier, GCP would be a better option.

B. AWS v/s GCP Scalability

All serverless functions are built with scalability in mind, and as such, there is no difference in this aspect for GCP and AWS. Both of them use containers in the background to support FaaS, and hence, can easily spawn more or less containers depending on the workload automatically based on a script.

C. AWS v/s GCP Concurrency

Concurrency for serverless functions dictates how a second request is processed by a FaaS instance while an earlier request is already being processed. In the background, it works by spawning another container the moment a concurrent request is sent. AWS lambda offers a soft limit of 1000 simultaneous requests in a region, with an option to reserve or provision more beforehand. GCP cloud functions have no advertised concurrency limit. Thus, as we can see, AWS provides us with customised concurrency management options, whereas GCP is vague on how concurrent requests are handled.

2.1.2 Database as a Service

DataBase as a Service(DBaaS) also known as managed database service, is a cloud computing service that lets users access and use a cloud database system without purchasing and setting up their own hardware, installing their own database software, or managing the database themselves. The cloud provider takes care of everything from periodic upgrades to backups to ensuring that the database system remains available and secure 24/7. All the popular cloud service providers offer their own managed service for the same, with popular ones being DynamoDB from Amazon and Firebase from Google. We can compare the two on a number of parameters in order to contrast their use cases.

1. DynamoDB v/s Firebase Pricing comparison

Table 2.2: DynamoDB v/s Firebase Pricing comparison

Metric	AWS DynamoDB	GCP Firestore
Free Tier storage	25 GB per month	1 GB per day
Free Tier Reads	25 Million read requests	50,000 read requests per day
Free Tier Writes	-	20,000 writes per day
Free Tier Deletes	-	20,000 deletes per day
Write Requests	$1.25 per million write request units	$1.08 per million write requests
Read Requests	$0.25 per million read request units	$0.36 per million read request units
Storage	$0.25 per GB-month	$0.108 per GB-month

From the above pricing comparison, we can draw 2 major conclusions:

1. AWS DynamoDB is cheaper for read requests, but more expensive for writes
2. GCP Firestore is cheaper for data storage

2. DynamoDB v/s Firebase Performance comparison

While comparing insertion performance for DynamoDB and Firestore is difficult due to its differences, we can compare the retrieval performance. Firestore, while nifty for querying, falls short with respect to querying as compared to DynamoDb coupled with a GraphQl API, which allows for very specific optimised queries.

3. DynamoDB v/s FireBase Fault Tolerance comparison

   All the data is stored on solid-state disks (SSDs) and is **automatically replicated across multiple Availability Zones** in an AWS Region, providing built-in high availability and data **durability**[6]. While firestore does not explicitly mention

   any fault tolerance mechanism, they have stated certain limitations, within which we must operate[7].

2.1.3 Storage as a Service

Storage as a Service or STaaS is cloud storage that you rent from a Cloud Service Provider (CSP) and that provides basic ways to access that storage. Enterprises, small and medium businesses, home offices, and individuals can use the cloud for multimedia storage, data repositories, data backup and recovery, and disaster recovery. There are also higher-tier managed services that build on top of STaaS, such as Database as a Service, in which you can write data into tables that are hosted through CSP resources.

The key benefit to STaaS is that you are offloading the cost and effort to manage data storage infrastructure and technology to a third-party CSP. This makes it much more effective to scale up storage resources without investing in new hardware or taking on configuration costs. You can also respond to changing market conditions faster. With just a few clicks you can rent terabytes or more of storage, and you don't have to spin up new storage appliances on your own.

Table 2.3: Amazon Simple Storage Service v/s Google Cloud Storage

Metric	AWS S3	GCP Cloud Storage
Cost of storage	$0.021 per GB per month	$0.020 per GB per month
Cost of Download	$0.05 per GB	$0.12 per GB
Free Tier allowance	-	15 GB

Comparing the two offerings from Google and Amazon, we can see that while the cost of storage for the two is comparable, with S3 being minutely more expensive, the cost of download is significantly cheaper. It is important to note that S3 has tiers of fault

tolerance available, and the one mentioned is the cheapest offering.

2.2 React v/s Angular v/s Vue

Performance is one of the most important aspects to be considered for a front-end application. And when it comes to evaluating the performance of Angular, React and Vue, keep in mind that DOM is considered as the UI of any application. Both React and Angular take different approaches to update HTML files, but Vue has the best of both React and Angular frameworks. Let's get deep into Angular vs React vs Vue comparison:

React

Pros:

- React is a front-end library that uses the **Virtual** DOM and enhances the performance of any size of application which needs regular content updates. For example, Instagram.
- React is based on **single-direction data flow**. This will provide better control over the entire project.
- **Up to date factor**. Facebook team supports the library. Advice or code samples can be given by Facebook community.Using React+ES6/7, application gets high-tech and is suitable for high load systems.

Cons:

- Learning curve. Being not full-featured framework it is requered in-depth knowledge for integration user interface free library into MVC framework.
- View-oriented is one of the cons of ReactJS. It should be found 'Model' and 'Controller' to resolve 'View' problem.Not using isomorphic approach to exploit application leads to search engines indexing problems.
- Lots of developers dislike JSX React's documentation, manuals are difficult for newcomers' understanding. React's large size library.

Angular

Pros:

- MVC architecture allows Angular to **split tasks into logical chunks, reducing the initial load time** of a web pages.
- The MVC also allows separation of concerns, with the view part being present on the client side, **drastically reducing queries in the background**.

Cons:

- Due to the many features of this framework, sometimes they can create a burden for your projects, all translating into a heavier application and slower performance compared to React or Vue.
- New, significant changes are introduced often. This can cause problems for developers when it comes to adapting to them.

Vue

Pros:

- Vue makes development absolutely easy as the production-ready project weighs 20KB after min+gzip. That results in faster runtime and also stimulates development and **allows developers to separate template-to-virtual DOM from the compiler. More than that, when you have a minimum project size**, you don't need to put an extra effort over-optimization.
- One of the most important advantages of using Vue.js is its size as you can **get production-ready build project weighs just 20KB after min+gzip**. Size is unbeatable with all other frameworks such as Angular, ReactJS, and jQuery.

Cons:

- Common plugins are useful as they work with various other tools to make development easy. Vue.js does not have most of the common plugins, and that is the drawbacks of Vue.js.
- Being a new member of a family, Vue has the smallest community support as compared to React and Angular.

Conclusion

React is scalable - React plays a key role in the largest social media platform in the world—Facebook. It's proven at massive scale and has a dedicated team of developers at Facebook with hundreds of other contributors outside the company.

React is fast - React uses intermediate representation (which they call "Virtual DOM") so that they can diff changes between different states, and make changes to a minimal amount of browser DOM nodes.

2.3 GraphQL v/s REST

With **REST**, there's a lot of back and forth and manual work. Calls can result in either over-fetching or under-fetching based on the API contract.

Whereas **GraphQL** gets exactly the data you want on an API call. You have control over the query on a granular level, which is not something you can not easily do with REST since it's not made for that specific purpose. Having this granular control will allow you to have fewer network calls and will require fewer developmental changes on client applications, since responsibility has been shifted to backends.

While REST offered many advantages to developers and became the de facto standard for businesses that deploy APIs, it also has a few disadvantages. These disadvantages arise from the fact that the server creates the representation of the resource, and the response to the client uses that.

RESTful APIs often returned more data than what the client needed, alternatively, the client had to make multiple API calls to get all the data it needed. Developers also had to design the API endpoints keeping the front-end views in mind, and changes to the front-end views required changes to the API endpoint.

Conclusion

GraphQL has faster product iterations on the frontend - Developers can write queries specifying their data requirements, and the iterations for developing of the frontend can continue without having to change the backend.

GraphQL enables better analytics on the backend - This enables the application owner to gain insights about which data elements are in demand, moreover, they will know which data elements aren't being used by clients anymore.

2.4 MaterialUI v/s Bootstrap v/s Ant design

MaterialUI is React Components that Implement Google's Material Design.

Bootstrap is simple and flexible HTML, CSS, and JS for popular UI components and interactions. Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web

Ant Design is a set of high-quality React components. An enterprise-class UI design language and React-based implementation. High-quality React components out of the box.

Conclusion

Ant Design has powerful theme customization in every detail. It's written in TypeScript.

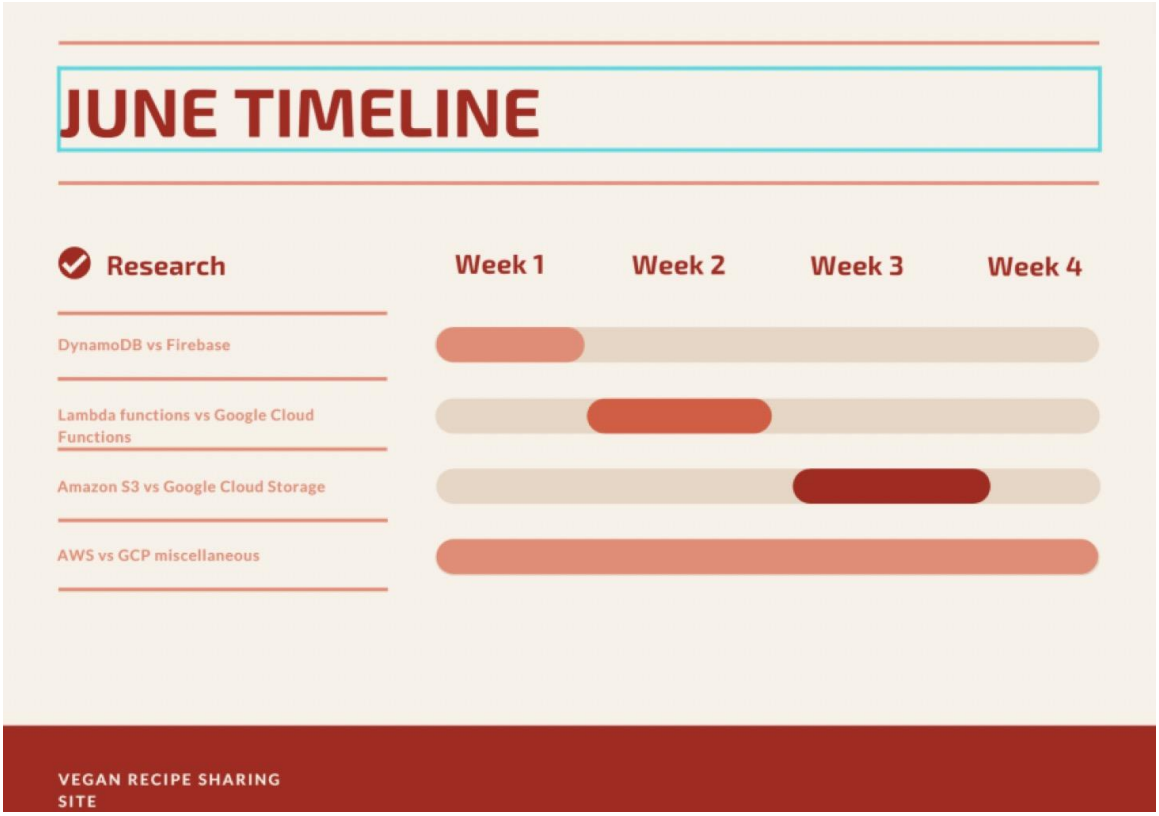
Chapter 3

Project Management

3.1 Software Development Schedule

During the first month my focus will be on research, exploring various cloud-based technologies, comparing them, testing them and finalising the cloud service provider.

Figure 3.1: June Plan

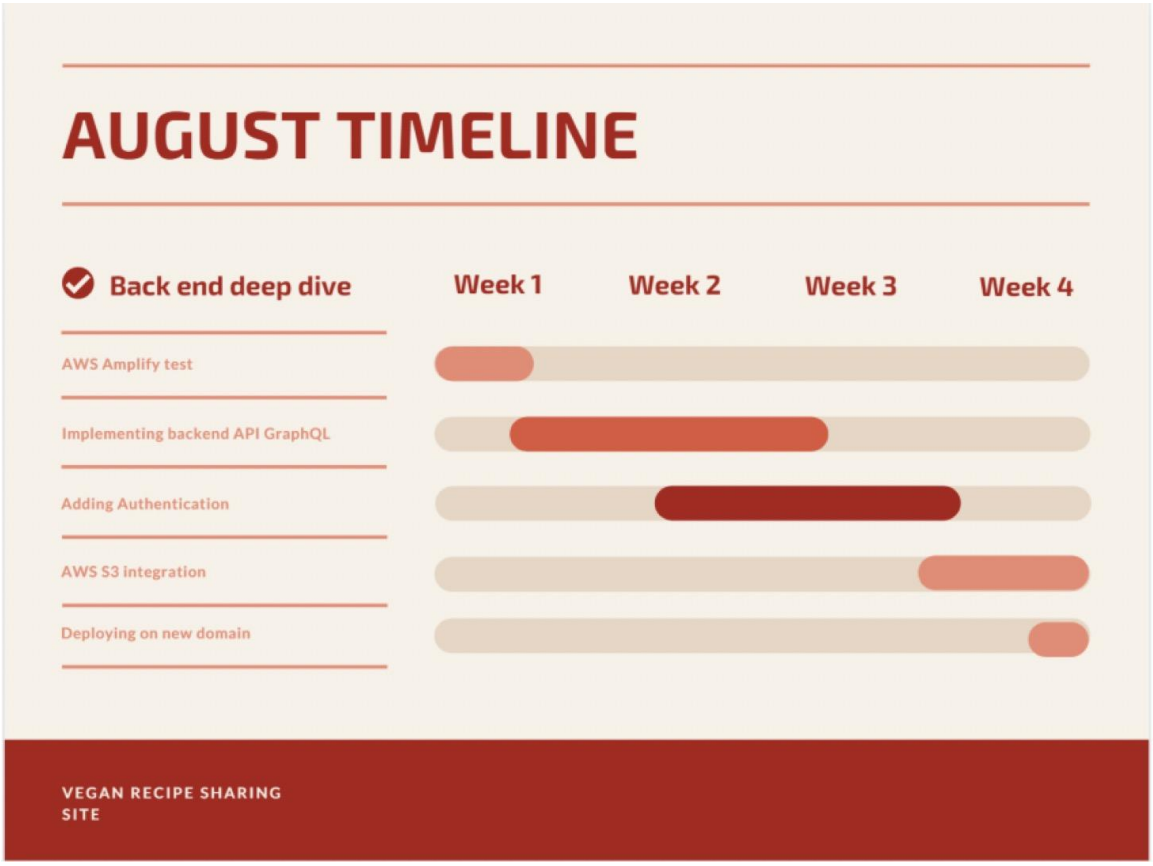


During the second month, I would be focusing on the front end, various frameworks, the best fit for my application and starting to build a User Interface.

Figure 3.2: July Plan

During the last month, I plan to work on the serverless backend, and its integration with the front end, to be ready with the final product.

Figure 3.3: August Plan



Some extra features I plan to work on are an admin dashboard, route53 as a DNS server and lambda functions for getting calories information via an API.

3.2 Software Development Methodology

DevOps deployment methodology : DevOps is not just a development methodology but also a set of practices that supports an organizational culture. DevOps deployment centers on organizational change that enhances collaboration between the departments responsible for different segments of the development life cycle, such as development, quality assurance, and operations.

DevOps is focused on improving time to market, lowering the failure rate of new releases, shortening the lead time between fixes, and minimizing disruption while maximizing reliability. To achieve this, DevOps organizations aim to automate continuous deployment to ensure everything happens smoothly and reliably.

I aim to incorporate this methodology for the project, as it is one of the most popular models used in recent times, and is growing in popularity with effective CI/CD pipelines set up for deployment. As we use amplify for our project, our code is synced from github, our choice of version control, and changes we make are pulled into production.

3.3 Risk Analysis

In building a project of this scale, we are likely to have to account for certain risks and plan to mitigate them:

- External API dependency to nutritionix API, if they change the API format, our application will need updates. If their API stops working, an important calorie feature of our app will be missing.
- Updates in frontend libraries which we are using could make certain aspects obsolete, and would require a code review and refactor.
- Changes in the UI library which we are using to build certain components.
- Changes in the AWS features, like AppSync, lambda functions or even DynamoDB, while changes may not be breaking, could lead to depreciation and would cause maintenance issues.
- Domain name availability for our application depends on when we purchase and reserve the domain name from Route53.

While some of the risks mentioned are unavoidable, we could plan to mitigate a few. We can monitor the API and also shortlist some other backup services providing a similar functionality in case it goes down. Any updates on the frontend libraries, for example nextjs or ant design can be easily accommodated as we are following a

DevOps deployment methodology, which fits our project well. All we need to do is make updates we need and push into github, and the changes will be easily integrated with our CI/CD pipeline. As for the domain name, we can purchase “theveganmanna” beforehand from route53 so that we do not have issues with availability.

Chapter 4

System Analysis

4.1 Stakeholders of the system

The following stakeholders can be identified:

(a) Admin managing reports of users and recipes

(b) Vegan Food Bloggers creating their profile and adding recipes

(c) Logged in registered users allowed to like and comment on recipes

(d) Visitors browsing recipes casually without registering

4.2 User requirements

(a) Register with email id and password

(b) Register and login with OAuth like facebook, instagram, google

(c) Update personal profile to add a bio, description, and social links

(d) Add a new recipe:

   a Admin managing reports of users and recipes

   b Vegan Food Bloggers creating their profile and adding recipes

   c Logged in registered users allowed to like and comment on recipes

   d Visitors browsing recipes casually without registering

(e) Update an existing recipe

(f) Delete an existing recipe

(g) Like recipes with a button

(h) Browse for recipes, search and filter them

4.3 System requirements

(a) Cognito or Auth service to manage users. No resources provisioned priorly, Amplify will manage auth by adding or removing users on demand.

(b) Backend scalable database, NoSQL based to allow for dynamic structure like recipes, which can have any number of ingredients and steps. DynamoDB is a good choice. No priorly provisioned hardware/ram/compute, fully scalable.

(c) Lambda functions based on python or javascript(nodejs) to hit the external nutritionix API in order to get calories based on recipe ingredients. Lowest tier lambda function should be suitable for our use case.

(d) S3 storage in order to save static images or videos, for cheap and quick access. No priorly provisioned hardware/ram/compute, fully scalable. S3 infrequently accessed tier would suffice our needs. Can shift to a more performant tier upon scaling up and acquiring enough customers to generate revenue.

4.4 User Roles

A user story is the smallest unit of work in an agile framework. It's an end goal, not a feature, expressed from the software user's perspective.The purpose of a user story is to articulate how a piece of work will deliver a particular value back to the customer.

With respect to our application, we can identify a number of users and their corresponding end goals:

Table 4.1: User roles

Story name	As a/an	I want to be able to	So that	Priority	Acceptance criteria
Create recipe	Chef	Add a new vegan recipe with recipe name, category, cuisine, preparation time, servings, description, ingredient list and steps, along with photos and videos for the same	A recipe is registered against my account, saved for everyone to view and engage with	1	A clean, intuitive form should be provided that caters to all the necessary fields needed for the recipe, with dynamic ingredients and steps accounted for
View created recipes	Chef	View my created recipes.	Validate which recipes have been added by me	1	An interface to display the recipes added by a user should be provided
Browse recipes	User	Browse recipes, filter recipes, open recipe page	Engage with the corresponding recipes	1	An interface to display recipes, along with a robust filtering and pagination technique, so that only requested recipes are shown
Edit profile	User	Edit my user profile bio, photo, and social links	My latest information is available to all visitors in order to contact me	2	An interface to upload user photo, edit bio, add social links like instagram, facebook and twitter
Popular recipes	User	View the most popular recipes of a chef	No time wasted in browsing all the recipes	3	A list of popular recipes shown on user profile page

Like recipes	User	Like recipes	My favourite recipes across all chefs are saved and easily accessible	1	A button to like a recipe should be available and upon liking a recipe, it should be added to a user's liked recipe list
Calorie count	User	See calories for a recipe	Make an informed decision and search for recipes fitting my diet plan	2	Calories for a recipe should be shown
Report recipes	User	Report a recipe for not being vegan or for spam content	There is no clutter and inappropriate content	3	A report option for a recipe should be available
Comment on recipes	User	Comment on recipes	Share my views about a recipe	2	An interface on the recipe details page allowing for commenting and viewing comments
Delete comments	User	Delete prior comments	Remove comments I made by mistake	2	An option beside a comment for the user to delete their own comment
Recent recipes	User	Jump to recent recipes from the home page	I can save time and browse only recent recipes	3	Recent recipes available as links on home page footer

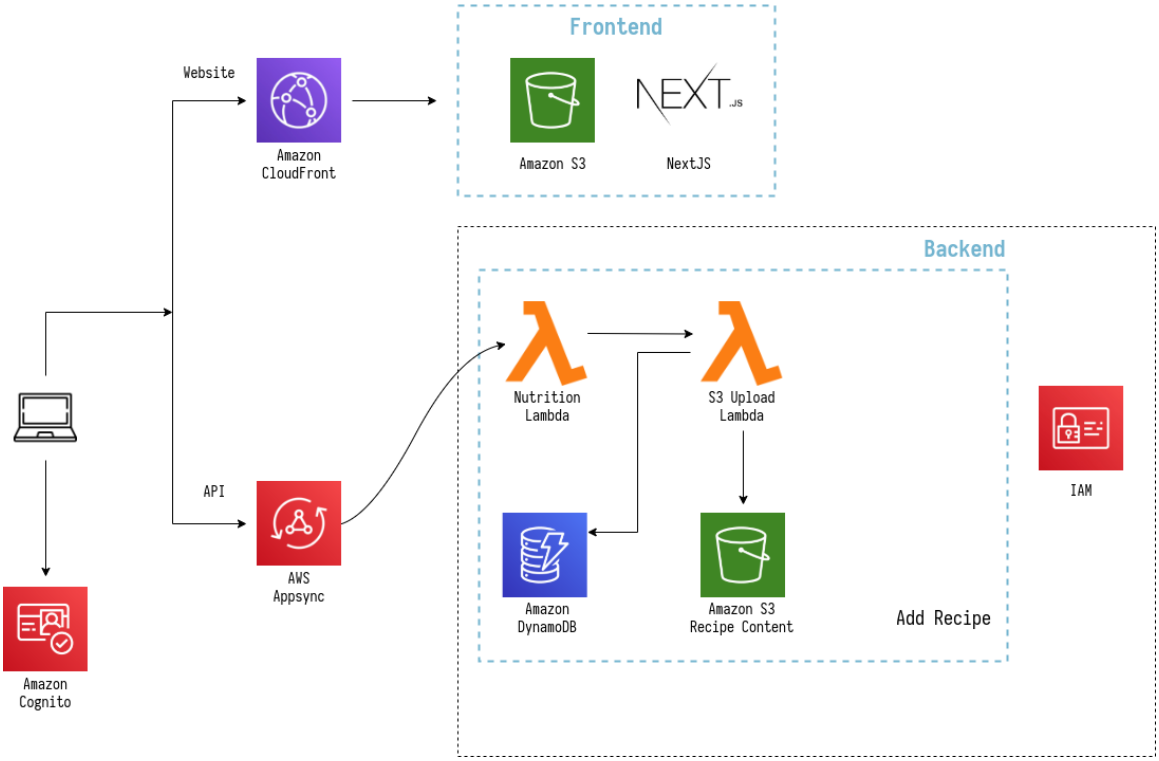
Popular categories	User	Jump to popular categories from the home page	I can save time and browse only categories that are popular	3	Popular categories available as links on home page footer
Filter users	Admin	Filter users who have been reported	Easily act on such users, either banning them or warning them	2	An admin interface to view reported users
Remove user	Admin	Remove user	Remove users who have repeatedly broken rules	2	An admin interface allowing for removal of users
Warn user	Admin	Send a warning message	So that users breaking rules are intimated about consequences of their posts	2	An admin interface allowing for sending messages to users
Delete users	Admin	Delete a recipe	So that recipes which violate our terms can be removed from the site	1	An admin interface allowing for deleting reported recipes upon review

Chapter 5

System Architecture

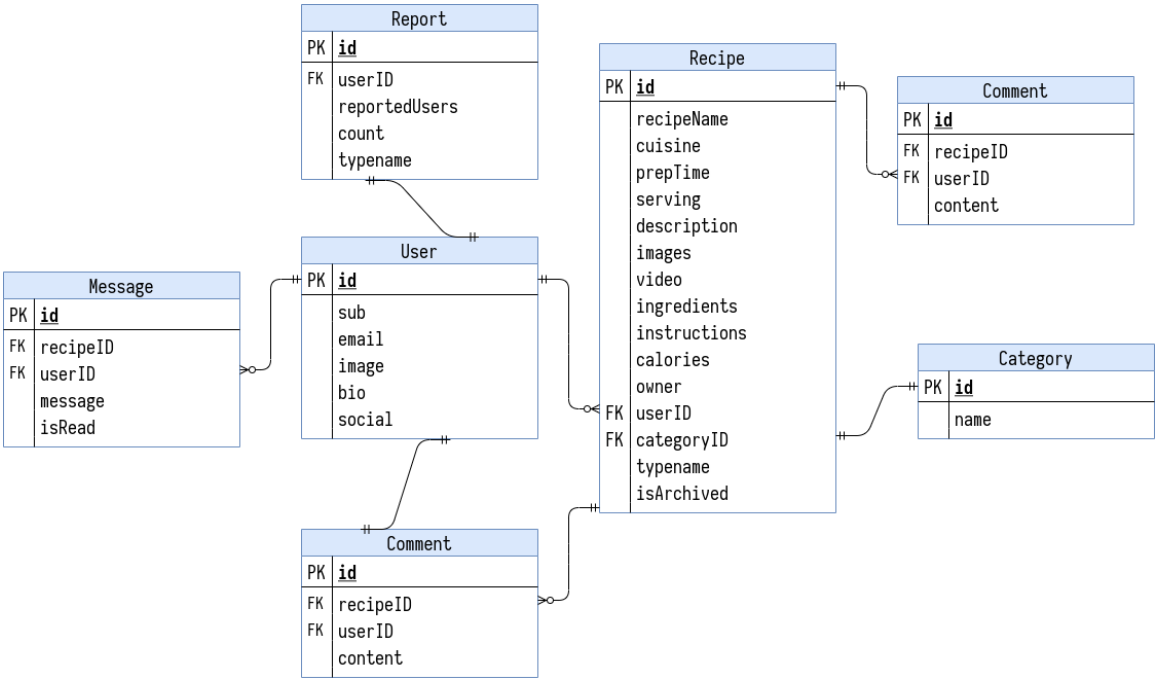
5.1 Application Architecture

Figure 5.1: Application Architecture



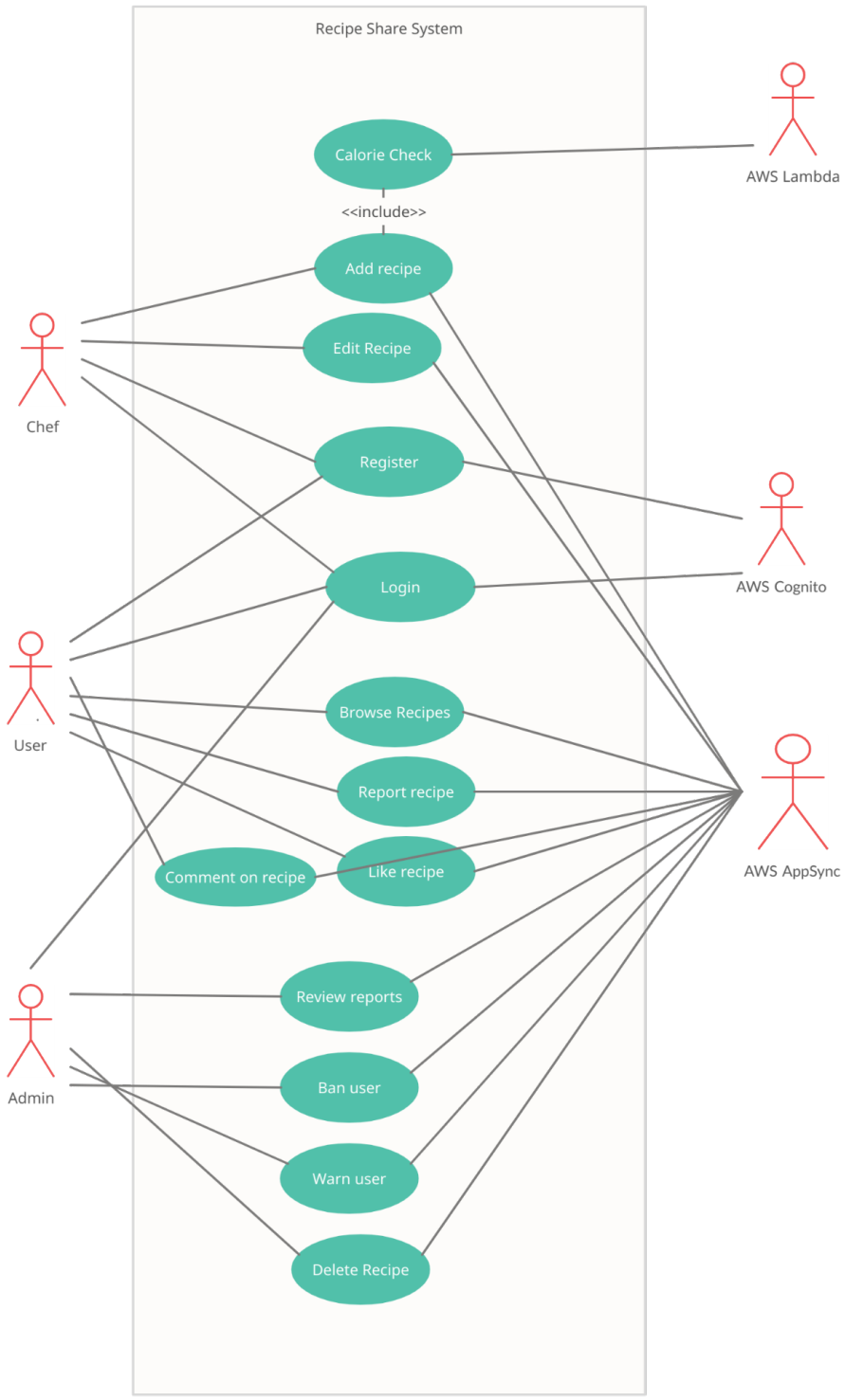
5.2 ER Diagram

Figure 5.2: ER Diagram



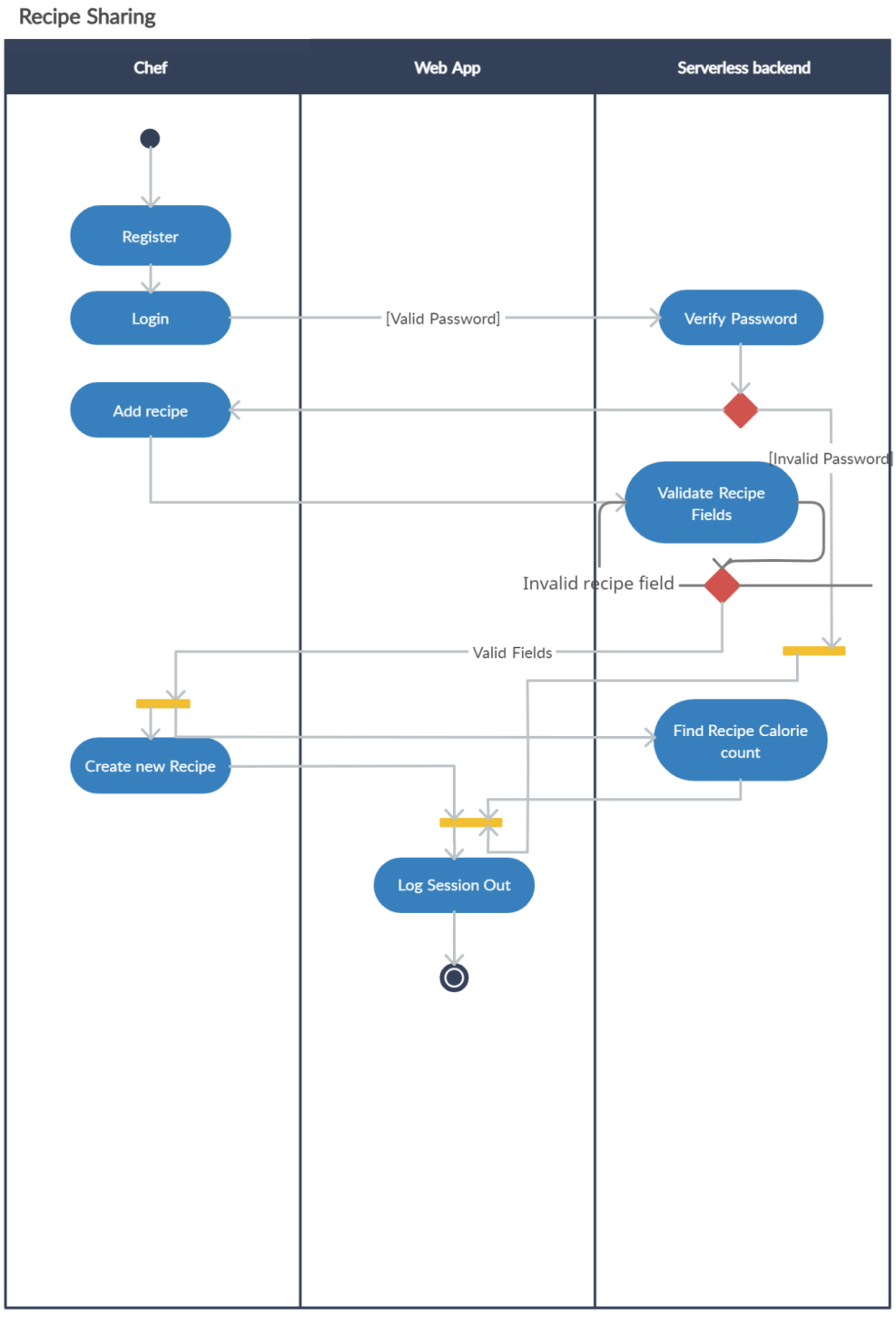
5.3 Use Case Diagram

Figure 5.3: Use Case Diagram



5.4 Activity Diagram

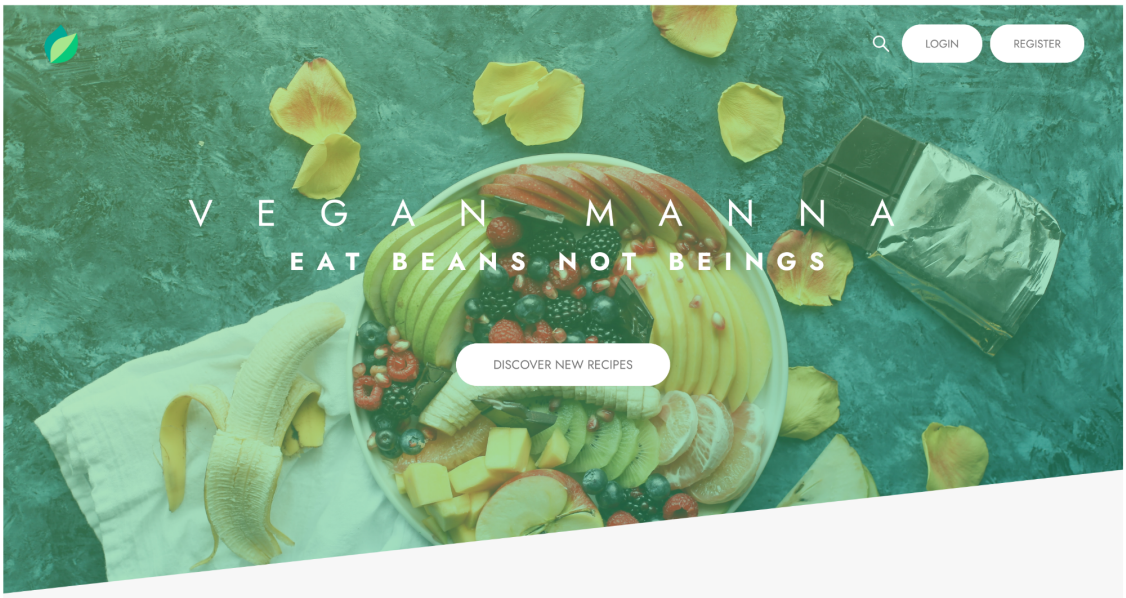
Figure 5.4: Activity Diagram



5.5 User Interface Design

Most of our design needs custom CSS to achieve the desired look. Depending solely on a UI library would rid the website of its uniqueness, making it lacklustre.

Figure 5.5: Header Section



The above effect of the header section is achieved using the clip-path of CSS, into a polygon.

Figure 5.6: About Section

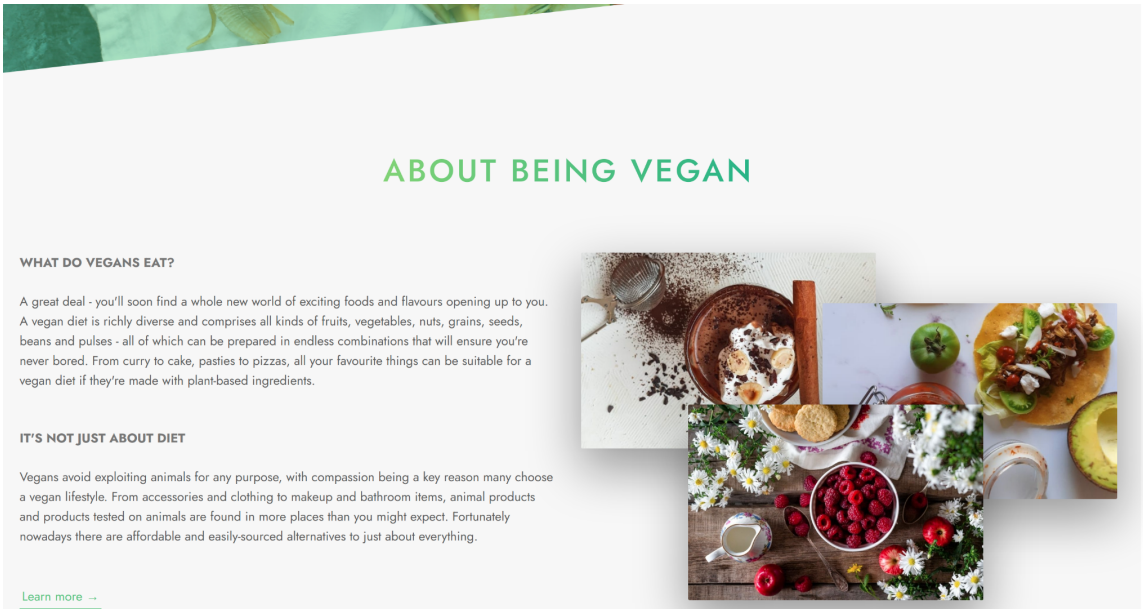
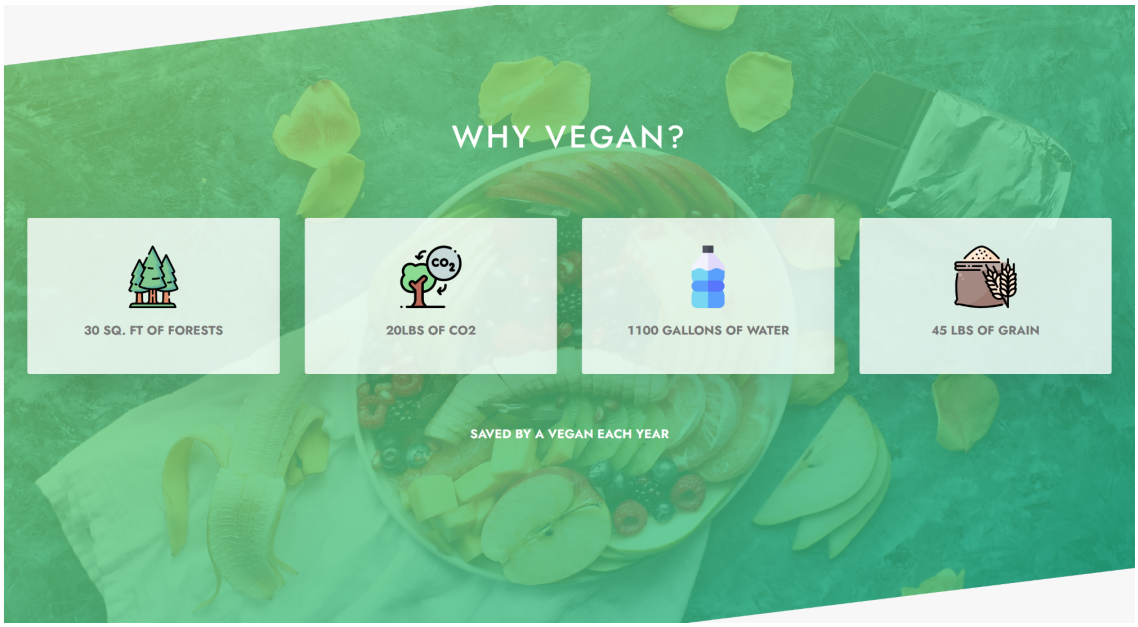
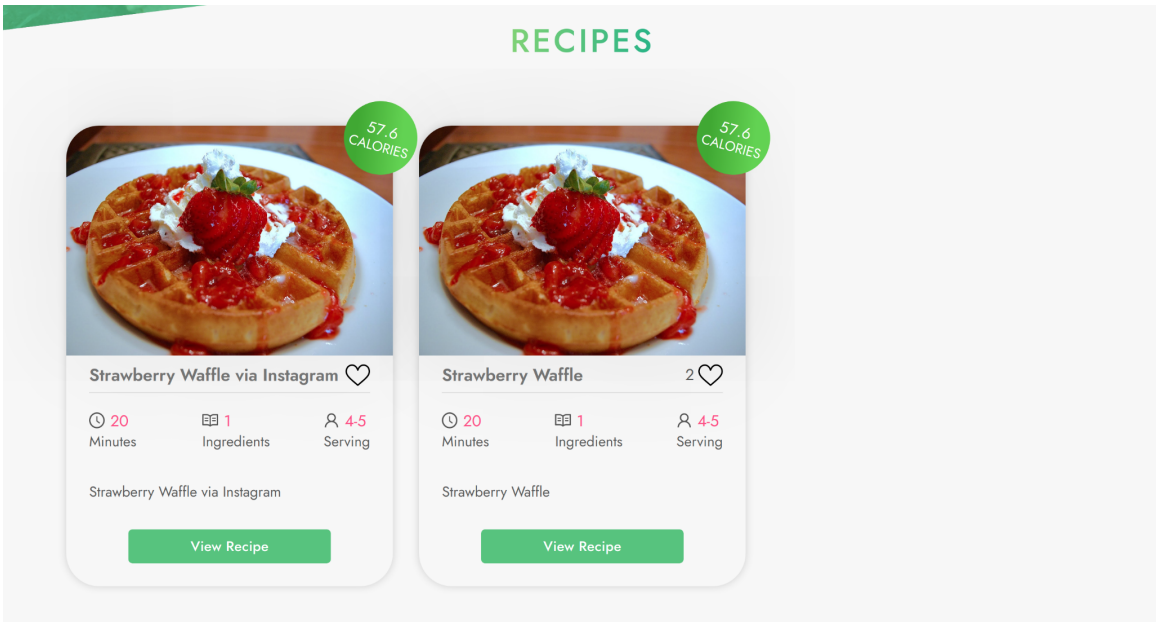


Figure 5.7: Image Collage



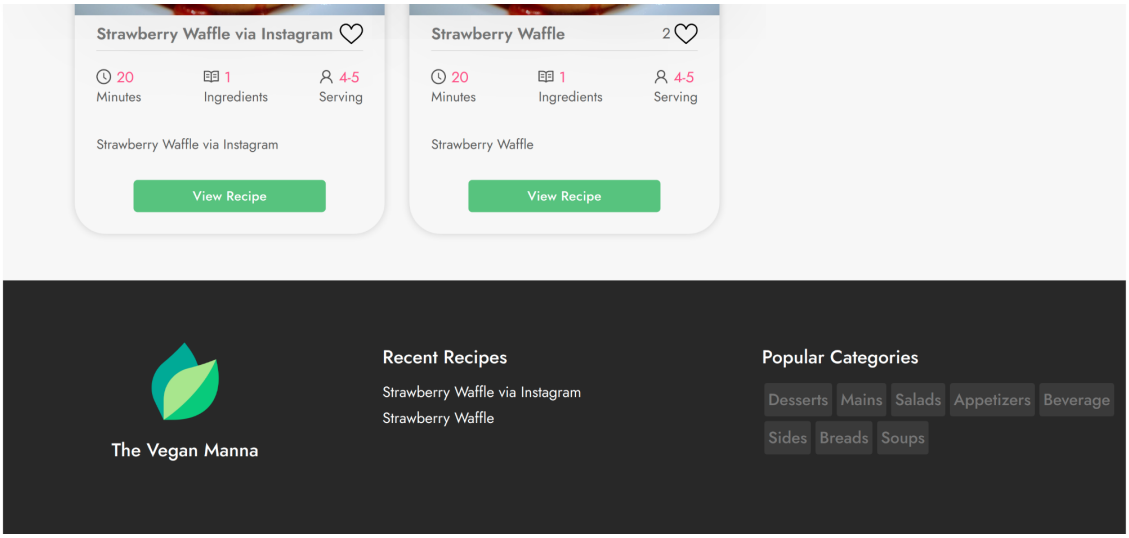
The animations for the image collage as well as the call to action cards are implemented using only css transitions, on the hover state.

Figure 5.8: Recipe Card



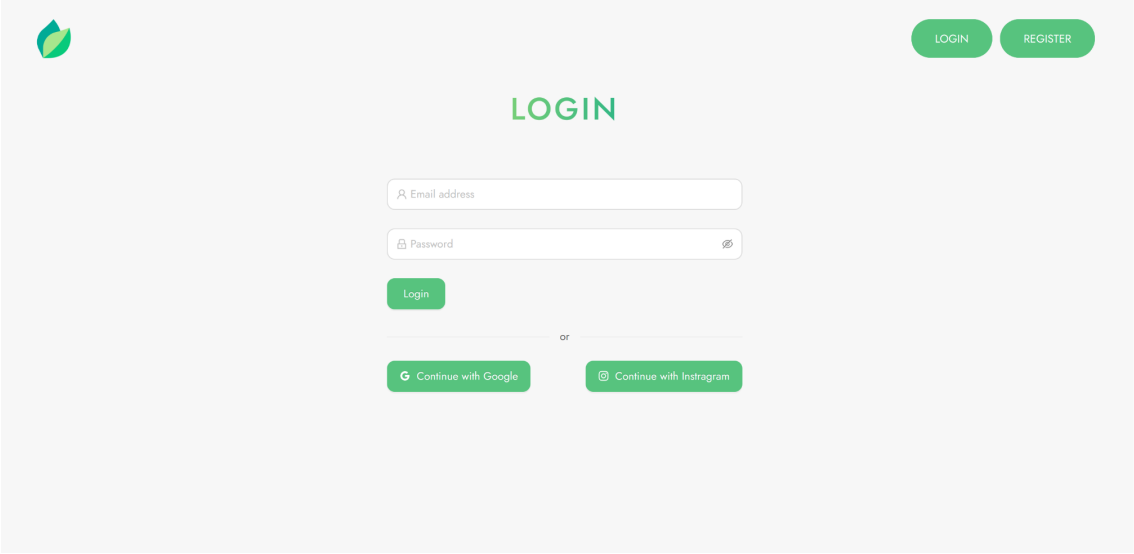
The cards are built using only CSS, using flexbox.

Figure 5.9: Footer Section



We use Flexbox for laying out the internal structure of UI components like the footer.

Figure 5.10: Login UI



We use Ant design for the login and register fields.

Figure 5.11: Register UI

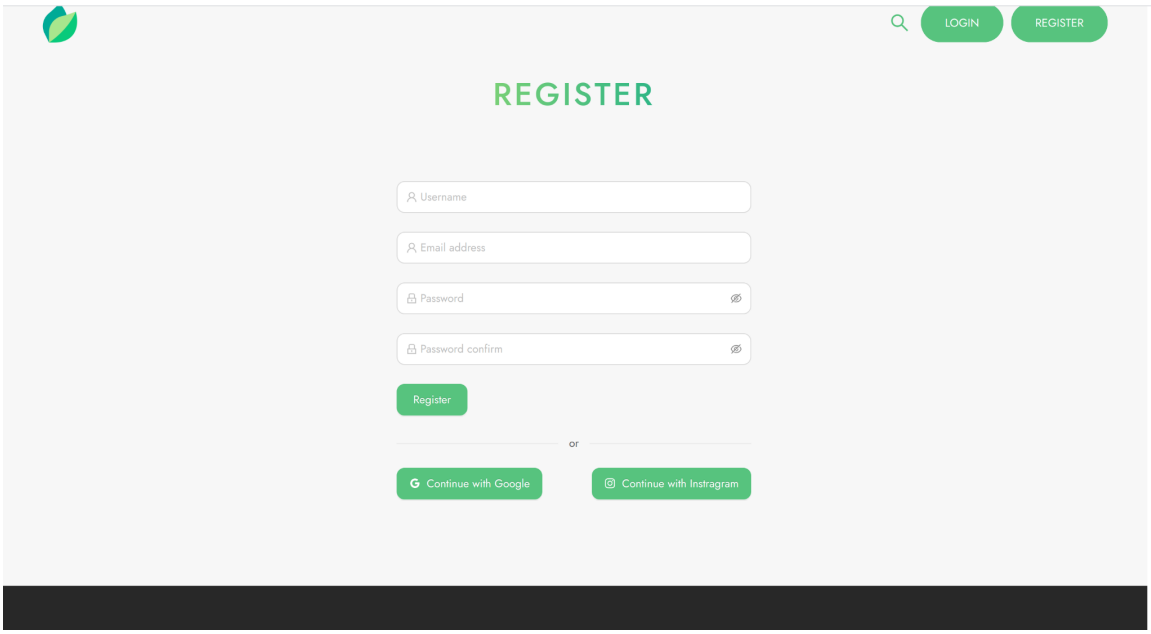
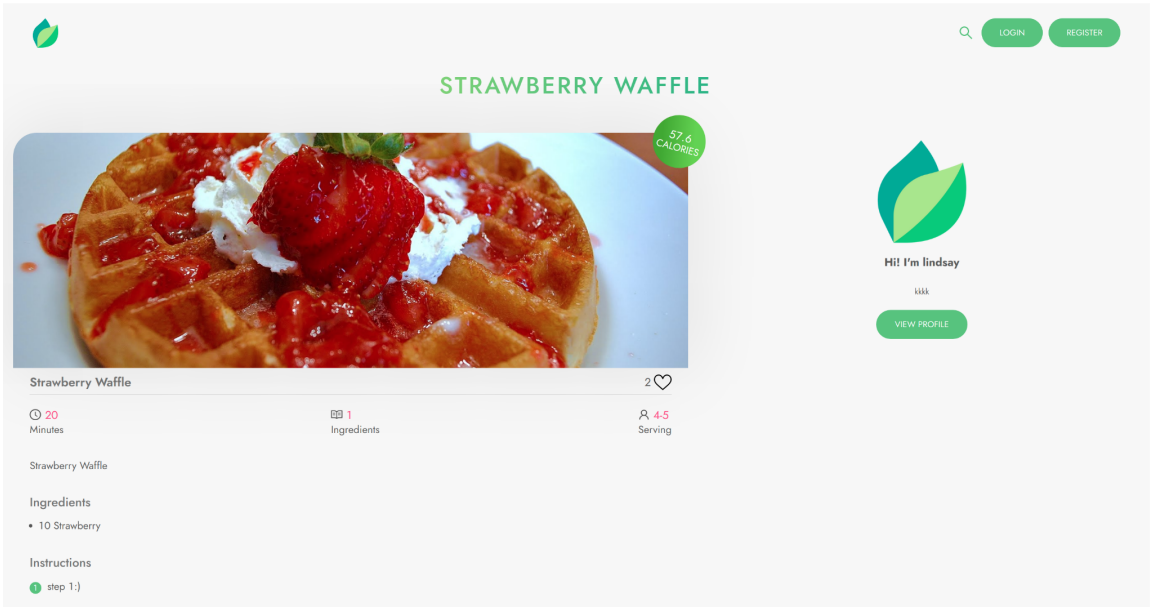


Figure 5.12: Search bar

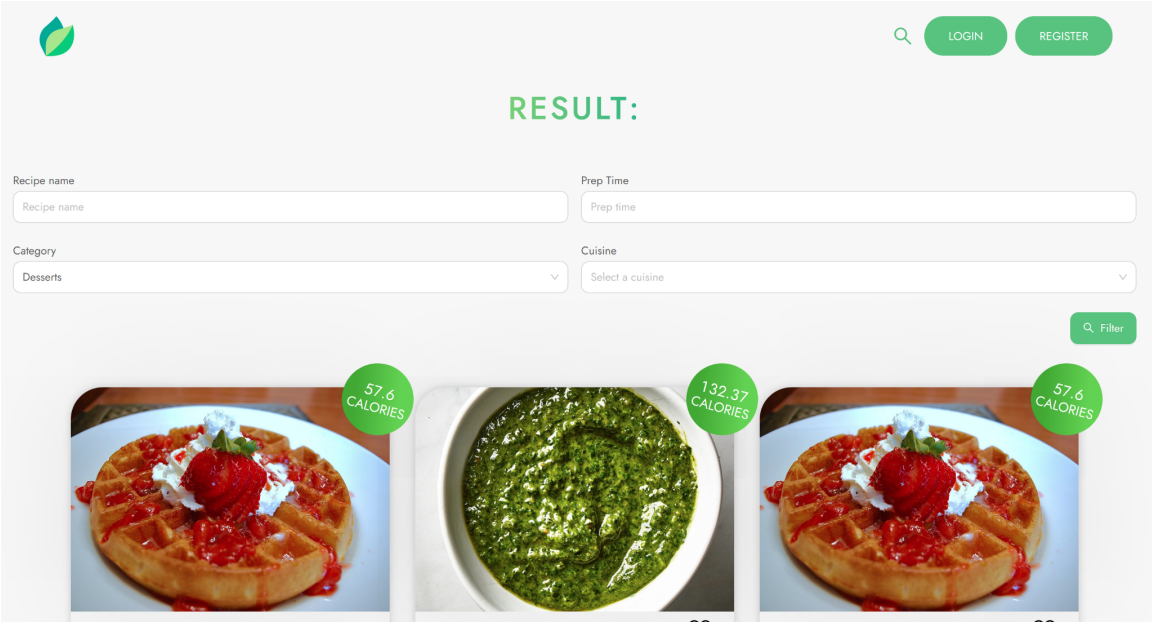
We also have a search bar implemented, which shows up only upon clicking

Figure 5.13: Recipe Detail



The recipe details page is similar to our card design, with a link to the respective user profile added.

Figure 5.14: Search UI



We also have a page to filter out search results based on recipe name keywords, categories, preparation-time and cuisine.

Chapter 6

Implementation

6.1 Technologies

- NextJS
- Ant Design
- AWS DynamoDB
- AWS S3
- AWS Lambda functions
- AWS Cognito
- AWS AppSync
- GraphQL

6.2 Tools

- Visual Studio
- Amazon console
- Postman
- AWS Amplify

6.3 Code Snippets

Figure 6.1: Delete Comments after deleting Recipe

```
const comments = recipe.comments?.items;

if (comments?.length === 0) return;

const commentMutations: any = comments?.map(
  (comment: Comment, i: number) => {
    return `mutation${i}: deleteComment(input: {id: "${comment.id}"}) { id }`;
  }
);

await API.graphql(
  graphqlOperation(`
  mutation deleteRecipeComments {
    ${commentMutations}
  }
`)
);
```

Figure 6.2: Recipe Pagination

```
type getRecipesProps = {
  id?: string;
  nextToken: string | null;
};

export const getRecipes = async ({ nextToken }: getRecipesProps) => {
  try {
    const res: any = await API.graphql(
      graphqlOperation(recipesByDate, {
        filter: { isArchived: { eq: 0 } },
        typename: "Recipe",
        limit: 6,
        sortDirection: "DESC",
        nextToken,
      })
    );
    const recipes = res?.data?.recipesByDate;
    return recipes;
  } catch (err) {
    catchError(err);
    return false;
  }
};
```

6.4 NextJS File Structure

Table 6.1: NextJS File Structure

Directory/File	Description
|– theveganmanna	Root folder
|– amplify	Amplify resource files
|– node_modules	Dependent modules
|– public	Public assets
|– src	Main application folder
|– api	API files
|– component	Component files
|– contexts	React Contexts
|– graphql	GraphQL files
|– mutations.js	
|– queries.js	
|– subscriptions.js	
|– hooks	React Hooks
|– interfaces	TypeScript Interfaces
|– layouts	Layout components
|– pages	NextJS pages
|– styles	Style files
|– aws-exports.js	Amplify config
|– tests	Jest test files
|– .babelrc	Babel config
|– graphqlconfig.yml	GraphQL config
|– jest.config.js	Jest config
|– next.config.js	Custom webpack config
|– package.json	Requirements of application
|– tsconfig.json	TypeScript config
|– yarn.lock	Lock file

Chapter 7

Testing and Evalutation

7.1 Unit Testing

7.1.1 Nutrition Lambda function

Testing Lambda Express function locally using Amplify Mock function

> $ amplify mock api lambdaFunction

Figure 7.1: Nutrition Lambda code

```
> const post = (ingredients) => {···
  };

  app.get("/", async function (req, res) {
    const event = req.apiGateway.event;
    if (event.ingredients) {
      const data = await post(event.ingredients);
      const calories = JSON.parse(data).foods.map((food) => food.nf_calories);
      const totalCalories = calories.reduce((accumulator, currentValue) => {
        return accumulator + currentValue;
      }, 0);

      res.json({ totalCalories });
    } else {
      res.json({ message: "Ingredients parameter missing" });
    }
  });
```

Figure 7.2: Nutrition Lambda output

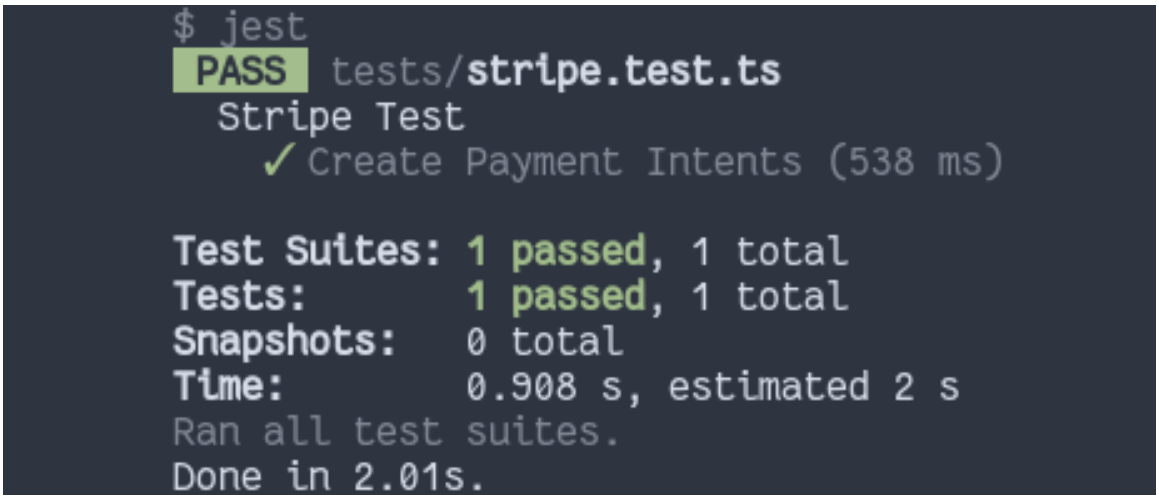
```
Ensuring latest function changes are built...
Starting execution...
EVENT: {"ingredients":"10 strawberry"}
App started
Result:
{
  "statusCode": 200,
  "body": "{\"totalCalories\":57.6}",
  "headers": {
    "x-powered-by": "Express",
    "access-control-allow-origin": "*",
    "access-control-allow-headers": "*",
    "content-type": "application/json; charset=utf-8",
    "content-length": "22",
    "etag": "W/\"16-20kOSqyBGc1uV4CcKu1qBOfSCZo\"",
    "date": "Mon, 02 Aug 2021 06:46:17 GMT",
    "connection": "close"
  },
  "isBase64Encoded": false
}
Finished execution.
```

7.2 Integration Testing

7.2.1 Stripe testing

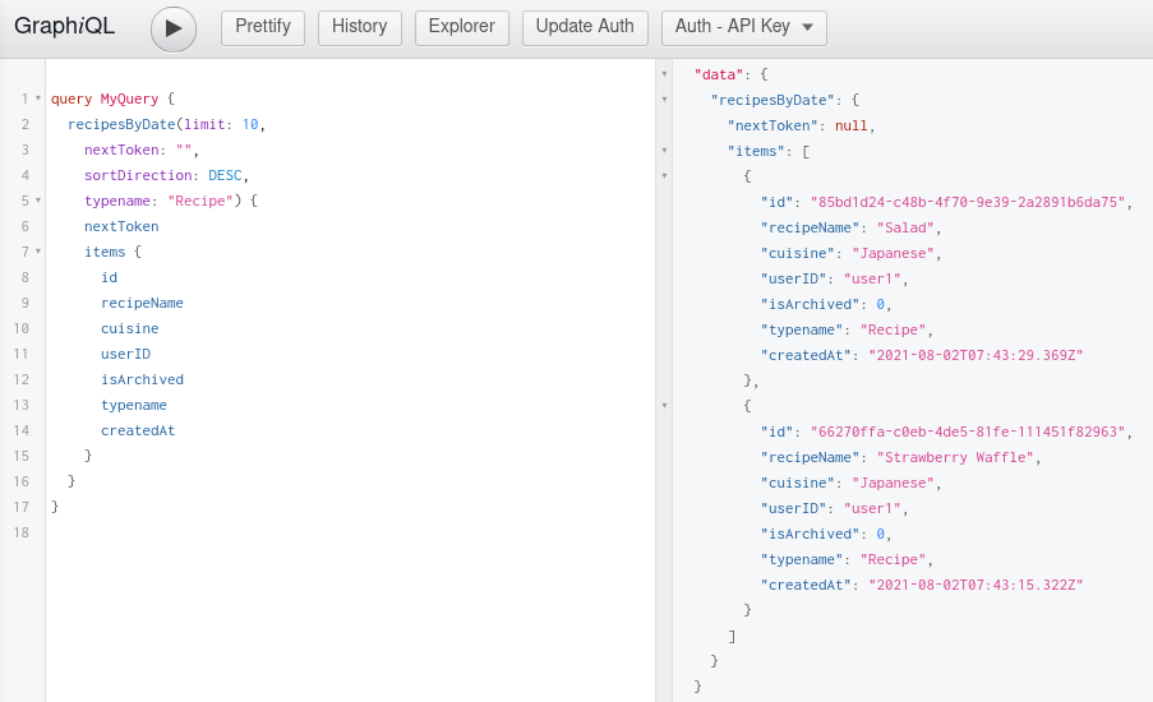
Figure 7.3: Stripe Payment Intent Test Code

Figure 7.4: Stripe Payment Intent Test Output



7.2.2 Fetch all recipes in descending order as unauthenticated user

Figure 7.5: Fetch all recipes



Chapter 8

Reflection

Chapter 9

Conclusion

Chapter 10

References

1. https://www.gartner.com/en/newsroom/press-releases/2019-07-29-gartner-says-worldwide-iaas-public-cloud-services-market-grew-31point3-percent-in-2018
2. https://cloud.google.com/functions/pricing
3. https://aws.amazon.com/lambda/pricing/
4. https://cloud.google.com/firestore/pricing
5. https://aws.amazon.com/dynamodb/pricing/on-demand/
6. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
7. https://firebase.google.com/docs/database/usage/limits
8. https://www.grandviewresearch.com/industry-analysis/cloud-computing-industry