CSCI5409 - Advanced Topics in Cloud Computing

Summer 2023

TERM ASSGINMENT REPORT

GitLab Repository Link: TermAssignment

A Cloud Native Hybrid Mobile Application - BStore

Submitted by

EMAYAN VADIVEL - B00934556

Under the Guidance of

PROF. ROBERT HAWKEY
Faculty of Computer Science

August 1, 2023

Contents

Part I.

Project Introduction

1. Project Introduction

1.1. Summary

This project is a hybrid cloud-native mobile application named **BStore**. I developed this application using **React Native Expo** [8] [9]. The application serves as an online platform for a hypothetical beauty store that sells a wide range of beauty products. The primary goal of this application is to provide a seamless and user-friendly shopping experience for customers, allowing them to browse, select, and purchase beauty products from the comfort of their homes.

The application is designed to cater to a broad user base, from beauty enthusiasts to casual shoppers, who are looking for a convenient and efficient way to shop for beauty products. The application's users are primarily individuals who are comfortable with technology and prefer online shopping over traditional stores. This application is intended to perform a variety of functions. Users can browse through various beauty products, add them to their cart or wishlist, and proceed to checkout when they are ready to make a purchase. The application also provides detailed information about each product, including its price, category, description, image and quantity, to help customers make informed purchasing decisions.

In the development of this BStore application, I had particular attention to ensure a smooth and responsive user experience. To achieve this, I employed Redux, a predictable state container designed to help JavaScript applications behave consistently across different environments [20]. Redux played a crucial role in managing the state of the application, particularly in handling activities related to the wishlist and shopping cart. With Redux, actions such as adding, removing, or displaying items in the wishlist or cart were handled efficiently. This ensured that the application's state always reflected the most recent user interactions, providing real-time updates to the user interface [20]. This state management solution was instrumental in maintaining the performance targets of the application, ensuring that user interactions were handled swiftly and accurately, regardless of the number of concurrent users or the complexity of the state changes.

In terms of performance targets, the application is designed to handle a large number of users concurrently, ensuring that the user experience is not compromised even during peak usage times. So, my application will load quickly, provide real-time updates on product availability, and process users' activity securely and efficiently [1]. The use of AWS services, such as AWS Cognito for user authentication and AWS Lambda for serverless computing, ensures that the application is scalable, secure, and reliable. I will discuss further in the following sections how I have used all the AWS services for devloping my mobile application as a cloud-native product.

The choices I made during the development of this project, from the selection of the tech stack to the design of the application's architecture, were all driven by the goal of creating an application that is robust, user-friendly, and capable of meeting the demands of a busy online beauty store. The following sections of this report will delve deeper into these choices, providing a comprehensive overview of the project's development process.

1.2. Implemented Functionalities

So far, I have implemented the following list of features using various AWS Services in my mobile application:

Table 1.1.: Feature Descriptions

Features	**Description**
User Authentication	Users can Register and Sign In into the application and use the Forgot Password option, if they want to reset their password.
Home Screen	Allows users to view all the available beauty products in BStore application. Besides, users can also add their preferred products into the wishlist.
Product Screen	Allows users to view the specific details about a particular beauty product in BStore application and also users can add them into the cart in their preferred quantities.
Wishlist	Allows users to see their favourite beauty products present in their wishlist.
Cart	Allows users to see their favourite beauty products into the cart along with the product quantities and total price.
Notifications	Sends notifications to users when they add or modify items in their wishlist or cart.

1.3. Project's Tech Stack

```
# Tech Stack
- Programming Language: React Native
- Framework: Expo
- Cloud Services: AWS (Lambda, S3, DynamoDB, Cognito, SNS, API Gateway, Step Functions)
- Payment Gateway Integration: Stripe payment gateway API (Needs to be implemented)
- CI/CD: Docker and Kubernetes (Needs to be implemented)
```

1.3.1. Tech Stack Description

The following list of items will describe my project tech stack briefly:

- **Programming Language:** The primary programming language used for this application is React Native. React Native is a popular framework for building mobile applications using JavaScript and React [8]. It allows developers to build hybrid mobile applications for Android and iOS using a single codebase.
- **Framework:** The application is built using the Expo framework. Expo is a set of tools built around React Native that help developers to build and deploy cross-platform mobile applications more easily [9]. It provides a range of APIs and services that make it easier to use native device functionality.
- **Cloud Services:** The application heavily relies on several AWS services. AWS Lambda [5] is used for running the application's serverless functions. AWS S3 is used for storing high-resolution product images, while DynamoDB is used for storing product details. AWS Cognito is used for user authentication, and AWS SNS is used for sending notifications to users. AWS API Gateway is used to manage the application's APIs [4], and AWS Step Functions are used to coordinate multiple AWS services into serverless workflows [6].
- **Payment Gateway Integration:** The Stripe payment gateway API is planned to be integrated into the application in the future to handle transactions securely [10].
- **CI/CD:** Docker and Kubernetes are planned to be used for the application's CI/CD pipeline. Docker will be used to containerize the application, allowing it to run in any environment. Kubernetes will be used to manage and scale these containers [11, 12].

Part II.

Requirements Analysis

2. AWS Services used and it's alternatives

2.1. Satisfying the requirements mentioned in the assignment description

As per the requirements mentioned in the term assignment description, I have used the following list of AWS services in different categories for various features of my mobile application:

Table 2.1.: AWS Services Usage

AWS Service Category	**AWS Service Used**	**Count**
Compute	AWS Lambda and AWS Step functions	2
Storage	AWS S3 and AWS DynamoDB	2
Network	AWS API Gateway	1
General	AWS Cognito and AWS SNS	2

In the assignment description, it is mentioned that to use any 1 storage related AWS Service. But according to my BStore project requirements, I have used 2 services in the storage section namely S3 and DynamoDB. Because, due to the space constraints for storing images as items in DynamoDB (*DynamoDB has a maximum item size of 400KB per item*) [3], I have used S3 buckets for storing all my high-resolution static product images as objects. Then I used all the object URLs as items in my DynamoDB tables along with other products data. I will explain more about my data storage in the following sections.

2.1.1. AWS Service mappings for Implemented functionalities

Table 2.2.: Feature Implementation based on the used AWS Services

Features	**Description**
User Authentication	AWS Cognito was used for user authentication. It provides features such as user registration, sign in, and password reset.
Wishlist and Cart	AWS API Gateway, Step Function, and Lambda were used to implement CRUD operations on wishlist and cart items.
Product Images and Details	AWS S3 was used to store high-resolution product images, and DynamoDB was used to store product details, including the URLs of the images stored in the S3 bucket.
Notifications	AWS SNS was used to send notifications to users when they add or modify items in their wishlist or cart.

So, Table 2.1 describes the requirements satisfaction of all the required AWS Services in my BStore project and Table 2.2 describes the features in which those AWS Services are used.

2.1.2. Overview

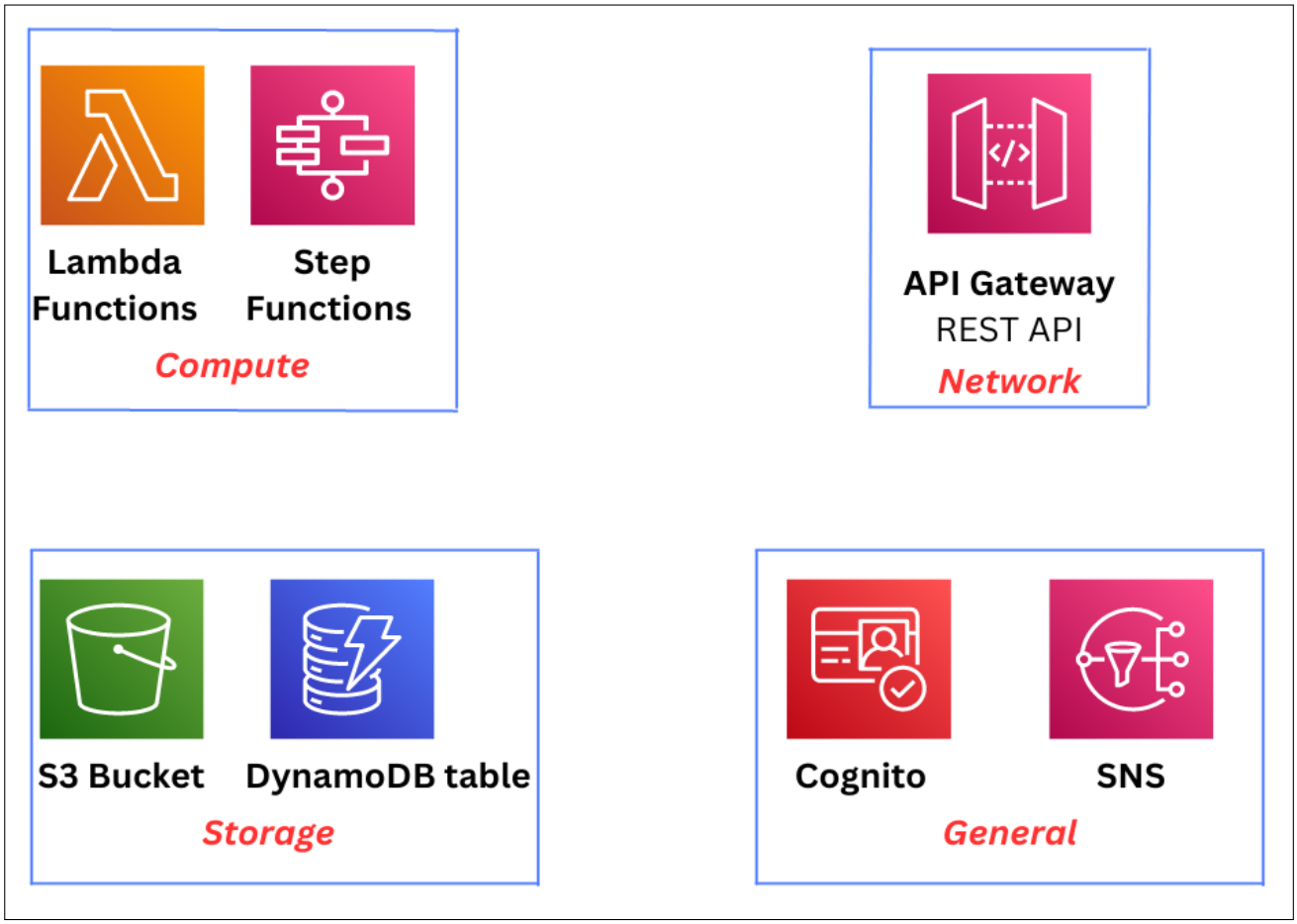

Figure 2.1.: **List of AWS services used in my project**

2.2. Comparison with Alternatives

Table 2.3.: Analysis with Alternative services

Services	Alternatives	Description
AWS Cognito	AWS IAM	AWS IAM (Identity and Access Management) is an alternative to Cognito for controlling access to AWS services. However, it doesn't support features like user registration or password reset [13].
AWS API Gateway	AWS App Runner	AWS App Runner is a service that makes it easy for developers to build, deploy, and run containerized applications quickly. It can serve as an alternative to API Gateway for certain use cases [14].
AWS Step Functions	AWS SWF	AWS SWF (Simple Workflow Service) is an alternative to Step Functions. It helps developers build, run, and scale background jobs that have parallel or sequential steps [15].
AWS Lambda	AWS EC2	AWS EC2 (Elastic Compute Cloud) provides resizable compute capacity in the cloud and can be used as an alternative to Lambda for more complex and longer-running applications [16].
AWS S3	AWS EFS	AWS EFS (Elastic File System) provides a simple, scalable file storage for use with Amazon EC2 instances. It could be used as an alternative to S3 for certain use cases [17].
AWS DynamoDB	AWS RDS	AWS RDS (Relational Database Service) provides an alternative to DynamoDB for applications that require a relational database structure [18].
AWS SNS	AWS SQS	AWS SQS (Simple Queue Service) offers a secure, durable, and available hosted queue that lets you integrate and decouple distributed software systems and components. It can serve as an alternative to SNS for certain use cases [19].

2.2.1. Reason for using mentioned services over alternatives

- **AWS Cognito over AWS IAM:** I chose Cognito for user authentication due to its out-of-the-box support for user registration, sign-in, and password reset, which are not available with IAM [1]. While IAM provides robust control over access to AWS services, I think that it lacks the user-facing features that Cognito provides, making Cognito a more suitable choice for user authentication in applications [13].

- **AWS API Gateway over AWS App Runner:** I opted API Gateway because its ability to manage and control traffic to back-end web applications [4]. It provides robust features for routing requests, transforming data, and handling responses, which are not as robust in App Runner. App Runner is more suited for quickly deploying containerized applications, not for managing APIs [14]. Moreover, API Gateway allows for easy deployment, monitoring, and security enforcement of APIs, which are crucial for the implementation of the wishlist and cart functionalities in my BStore application.

- **AWS Step Functions over AWS SWF:** I chose Step Functions due to their seamless integration with AWS Lambda and their ability to handle complex workflows. Additionally, Step Functions provides a graphical console to visualize the components of my application, which aids me in monitoring and debugging [6]. While SWF can also handle workflows, it requires more manual setup and does not integrate as smoothly with Lambda functions [15].

- **AWS Lambda over AWS EC2:** I opted for Lambda functions mainly for its serverless architecture, meaning there is no need to manage the underlying server infrastructure [5]. This makes my mobile application more cost-effective and scalable than EC2, especially for applications with fluctuating demand. This contrasts with EC2, which requires server management and may not be as cost-effective for applications with unpredictable demand and adds additional complexity and cost [16].

- **AWS S3 over AWS EFS:** I used S3 buckets for storing my product images due to its durability, scalability, and cost-effectiveness [2]. EFS, while useful for certain use cases, may be overkill for simply storing images and could incur unnecessary costs. Moreover, EFS also provides scalable file storage, it is designed to be used with EC2 instances and may not provide the same level of accessibility and cost-effectiveness as S3 when used to serve static content over the internet [17].

- **AWS DynamoDB over AWS RDS:** I used DynamoDB for storing product details due to its seamless scalability and fast, consistent performance. While RDS provides a relational database structure, it may not perform as well as DynamoDB under heavy load and may require complex queries to retrieve data [3]. Moreover, RDS is a powerful relational database service, it may not provide the same level of performance for read-intensive workloads, and it may require more management overhead compared to DynamoDB [18].

- **AWS SNS over AWS SQS:** I used SNS mainly for its ability to send notifications to a large number of subscribers, including distributed systems, end-user emails, SMS, and HTTP endpoints [7]. While SQS is a robust queuing service for decoupling and scaling microservices, distributed systems, and serverless applications, it doesn't natively support message fan-out to multiple subscribers, making SNS a more suitable choice for my project's notification use case [19].

These reasons are based on my personal experiences and knowledge with the above-mentioned services and it may vary with several other use cases. But for my BStore project, I've used these services for the above reasons.

Part III.

Deployment and Delivery Models

3. Cloud Deployment and Delivery models

3.1. Choosing the right deployment model

To begin with, As per the professor's notice in Teams Channel for the people who developed mobile applications for this term assignment about the deployment, I want to be clear that I haven't deployed my BStore mobile application on AWS. Instead, I demonstrated my application through my local simulator in my video submission, which professor said it's acceptable.

However, In a hypothetical scenario, I would like to explain my deployment model in AWS Cloud by explaining what type of deployemnt model I will choose and the reason for that deployment model preference in this section.

For the deployment of the BStore mobile application, I think the most suitable cloud deployment model is the **Public Cloud**. In this deployment model, the application's resources and services are hosted and managed by a third-party cloud service provider, which in this case would be AWS (Amazon Web Services) [21]. Here's my detailed explanation of why the Public Cloud deployment model is an ideal choice for the deployment of my BStore mobile application:

- **Scalability and Elasticity:** The public cloud provides immense scalability and elasticity, which is crucial for a shopping application like BStore. The app might experience fluctuating traffic, with peak times during holidays or sales events. With AWS, I can automatically scale my resources up and down as needed, which isn't as straightforward with a private, community, or hybrid cloud deployment.
- **Global Reach:** The Public Cloud providers, such as AWS, have data centers distributed across multiple regions globally. This global reach ensures low-latency access to the application for users from various geographical locations. As my BStore application aims to target a broad audience, a global presence is crucial for delivering an excellent user experience [21].
- **Cost Efficiency:** A significant advantage of the public cloud is its pay-as-you-go pricing model. I only pay for the resources we use, which can significantly cut costs, especially for a start-up application like BStore. A private or community cloud, on the other hand, requires a considerable upfront investment in infrastructure.
- **Speed to Market:** Deploying on a public cloud like AWS will allow me to move more quickly [21]. I can get my infrastructure set up faster than I could with a private cloud, which involves setting up physical servers and networking equipment. This speed will enable me to get BStore in the hands of users quicker.

- **Focus on Core Business:** With a public cloud, I offload the responsibility of maintaining the infrastructure to the cloud provider, allowing me to focus on developing and improving the BStore application itself [21]. This offloading would not be possible with a private or community cloud.
- **Rich Suite of Services:** AWS provides a vast array of services that can enhance our application. For example, AWS Cognito for authentication, AWS S3 for storage, and AWS DynamoDB for a NoSQL database are all services that I can utilize without additional setup. This availability contrasts with a private cloud where I'd have to manually set up these services.
- **Managed Services:** Public Cloud providers offer a wide range of managed services, including AWS Cognito, S3, DynamoDB, API Gateway, Lambda, Step Functions, and SNS, which align perfectly with the requirements of my BStore application. These managed services simplify development, deployment, and maintenance, allowing developers to focus on building the application's core functionalities further in the future.
- **Security and Compliance:** Public Cloud providers adhere to stringent security practices and compliance standards, ensuring the safety of user data and transactions [21]. AWS, for example, provides a secure infrastructure and various security features, such as Identity and Access Management (IAM), encryption, and DDoS protection, to safeguard the application and user information.
- **High Availability and Reliability:** Public Cloud services are designed to offer high availability and reliability. With multiple data centers and redundancy mechanisms, the application can continue to function even in the event of hardware failures or disruptions in one region.
- **Global Developer Community:** AWS, being one of the largest Public Cloud providers, has a vast and active developer community. This community support provides access to resources, best practices, and solutions to common challenges, issues and enhancing the development process and promoting innovation.

While a hybrid cloud model could provide some benefits in terms of data security and control, the increased complexity of managing resources across both a public and private environment wouldn't bring sufficient advantages for my BStore application. In the case of the multi-cloud model, it could offer increased reliability and less vendor lock-in, but it also comes with increased complexity in terms of different APIs, different consoles, and cost management across multiple vendors.

3.1.1. Finalizing the right deployment model

I will choose the Public Cloud deployment model for the BStore mobile application because of the numerous benefits it provides in terms of scalability, cost-efficiency, global reach, security, and ease of development. As a startup or small business, I think my BStore application's primary focus should be on building a feature-rich, user-friendly application rather than managing complex infrastructure.

By opting for the Public Cloud, I can leverage the expertise and resources of AWS to create a robust and highly available platform that caters to a global user base without incurring significant upfront

costs [21]. Moreover, the Public Cloud's pay-as-you-go model will allow me to manage expenses efficiently and scale resources as the application gains traction. The managed services offered by AWS simplify the development process, reducing the time and effort required to deploy new features and updates. Additionally, the security and compliance measures provided by the Public Cloud instill confidence in both myself and the users, ensuring that sensitive data is protected from potential threats.

In summary, the Public Cloud deployment model aligns perfectly with BStore's goals of delivering an efficient, reliable, and secure e-commerce application to users worldwide while optimizing costs and development efforts.

3.2. Choosing the right delivery model

For the delivery model of the BStore mobile application, I think the most suitable option is **SaaS (Software as a Service).** SaaS is a cloud computing model where the application is hosted and provided as a complete service to end-users over the internet [22]. In my case, BStore's mobile application will be delivered as a fully functional software solution to customers without the need for them to manage any underlying infrastructure or software updates. This is my in-depth explanation of why this SaaS model suits the requirements of BStore mobile application:

- **Ease of Access and Convenience:** As a SaaS application, BStore can be accessed by users from any device with an internet connection and a web browser. This level of accessibility and convenience aligns perfectly with the mobile application's nature, enabling users to shop on the go and from any location [22].
- **Rapid Deployment and Time-to-Market:** With SaaS, the application is already deployed on the cloud infrastructure, and users can start using it immediately after signing up [22]. This significantly reduces the time-to-market for BStore, allowing me to quickly launch the application and start serving customers without delays.
- **Automatic Updates and Maintenance:** In a SaaS model, the cloud service provider is responsible for managing updates, patches, and maintenance of the application and underlying infrastructure [22]. This ensures that users always have access to the latest features and enhancements without any effort on their part or mine. In other words, I'd say that SaaS applications can be updated centrally in the cloud without requiring user intervention. This feature enables me to quickly respond to user feedback, fix bugs, and add new features, ensuring that all users have access to the latest and greatest version of BStore.
- **Scalability and Elasticity:** As the BStore application gains popularity and attracts more users, the SaaS model can easily scale to accommodate the increasing demand. The cloud infrastructure can automatically allocate additional resources to ensure optimal performance and responsiveness. Also, the SaaS model is inherently scalable. As BStore's user base grows, I can readily scale the application to meet the increased demand without worrying about acquiring additional infrastructure or managing platform configurations.

- **Cost-Effectiveness:** The SaaS delivery model is cost-effective for both BStore and its users. Users don't need to invest in hardware, software licenses, or ongoing maintenance, as they pay a subscription fee based on their usage [22]. For BStore, the pay-as-you-go pricing model offered by the cloud service provider ensures that I only pay for the resources used by the application, optimizing costs and eliminating the need for large upfront investments. With a SaaS model, I can efficiently manage my resources. The application's back-end operations are centralized in the cloud, meaning I do not have to allocate resources to each user's device [22]. This centralization contrasts with an IaaS model, where I would have to manage and allocate infrastructure for each user or instance of the application.

- **Focus on Core Business:** By choosing the SaaS model, I can focus on developing and enhancing the core features of the BStore application, as the management of infrastructure, security, and updates is handled by the cloud service provider. This allows me to concentrate on delivering the best possible shopping experience to users.

- **Global Reach and Scalability:** The SaaS model allows BStore to cater to a global audience seamlessly. With data centers distributed across various regions, the application can deliver low-latency performance to users worldwide, contributing to a positive user experience and customer satisfaction

- **No Client-side Installation:** A crucial feature of SaaS is that the application runs on the cloud, meaning users do not need to install or maintain any software on their devices [22]. This approach eliminates the need for users to worry about system requirements, software updates, and troubleshooting, all of which would be essential considerations in an Infrastructure as a Service (IaaS) or Platform as a Service (PaaS) model.

3.2.1. Finalizing the right delivery model

I will choose the SaaS delivery model for the BStore mobile application because it aligns perfectly with my goals of providing a user-friendly, feature-rich, and cost-effective e-commerce solution. As a developer, my primary focus is on delivering a top-notch shopping experience to customers and users without getting burdened by managing infrastructure and maintenance.

By selecting the SaaS model, I can leverage the expertise and reliability of the cloud service provider to ensure that the BStore application remains up-to-date, secure, and highly available at all times [22]. The automatic scaling capabilities of the SaaS model provide the flexibility to accommodate user demand as the application grows, ensuring that BStore can handle peak shopping seasons and traffic spikes without any performance issues. Moreover, the cost-effectiveness of the SaaS model allows me to optimize expenses and allocate resources efficiently. Instead of investing in hardware and software licenses, I can channel my financial resources into improving the application's features and marketing efforts.

In summary, the SaaS delivery model is the perfect fit for the BStore mobile application, as it provides ease of access, rapid deployment, automatic updates, scalability, cost-effectiveness, and allows me to focus on delivering the best possible shopping experience to users worldwide. With SaaS, BStore

can truly stand out in the competitive e-commerce market and achieve its performance targets while staying agile and customer-centric.

3.3. Scope of Migration towards FaaS delivery model

While the BStore mobile application is currently designed as a Software as a Service (SaaS) model, there is potential scope for migrating towards a Function as a Service (FaaS) model, specifically for certain parts of the application infrastructure [23]. FaaS is a subset of serverless computing that allows developers to execute modular pieces of application code in response to events. The following list of points will describe how I envision this migration towards the FaaS delivery model for my BStore mobile application:

- **Stateless Functions:** FaaS is ideal for stateless functions or services. So, for example, I can identify specific functionalities in the application, such as sending notification emails, generating recommendations, or processing payments, and convert these into individual functions that can be executed on a FaaS platform like AWS Lambda.
- **Improved Scalability and Cost Optimization:** One of the primary motivations for migrating towards FaaS is the ability to optimize costs and achieve better scalability. With the FaaS model, the application can automatically scale based on the number of incoming requests, and I will only pay for the actual execution time of each function [23]. This allows me to efficiently allocate resources and ensure that I'm not overprovisioning or underutilizing server capacity.
- **Focus on Business Logic:** By adopting FaaS, I can shift my focus away from managing and maintaining servers to concentrating on developing and refining the core business logic of the BStore application. The cloud provider takes care of the underlying infrastructure, ensuring that I can invest more time and effort in enhancing the user experience, adding new features, and optimizing performance.
- **Event-Driven Architecture:** FaaS is well-suited for event-driven architectures, where functions respond to specific events or triggers [23]. Since FaaS is inherently event-driven, I can redesign parts of my application to follow an event-driven architecture. For instance, when a user adds an item to their cart, it can trigger a function to update the cart and perhaps another function to make product recommendations.
- **DevOps Streamlining:** FaaS can help streamline DevOps practices, reducing the time and effort needed for server management and enabling me to focus more on the core functionalities of BStore [23]. It would also facilitate continuous delivery and integration, as I can update and deploy individual functions independently without affecting the entire application.
- **Reduced Development Time:** Since I will be focusing on developing individual functions rather than managing server resources, the development cycle can be significantly reduced. This will enable me to iterate and release new features more quickly, enhancing the application's agility and responsiveness to user feedback.

- **Improved Fault Tolerance and Resilience:** With FaaS, the cloud provider manages fault tolerance and recovery. In case of a function failure, the provider automatically retries the execution or redirects it to healthy instances. This enhances the overall reliability and resilience of the BStore application, ensuring uninterrupted service to users.

3.3.1. Reasons for Migration towards FaaS

- **Cost-Efficiency:** FaaS offers a pay-as-you-go pricing model, which means I will only be charged for the actual compute time used by each function. This cost-efficiency is particularly beneficial for a growing application like BStore, as it allows me to manage expenses effectively and avoid unnecessary overhead. This is the place where I'm charged based on the number of function executions and their duration. This model can be more cost-effective than maintaining servers that are running continuously, as in the SaaS model.

- **Reduced Operational Overhead:** By adopting FaaS, I can offload much of the server management and operational responsibilities to the cloud provider. This shift allows me to focus on improving the application's functionalities and user experience.

- **Scalability:** As the BStore application gains more users and experiences fluctuations in traffic, FaaS can automatically scale up or down based on demand. This ensures that the application remains responsive even during peak times, providing a seamless shopping experience for users. With FaaS, each function can scale independently based on demand. If a particular feature of BStore becomes extremely popular, the corresponding function can automatically scale to handle the increased load without affecting the performance of other functions.

- **Serverless approach:** By adopting FaaS, I embrace a serverless approach, which eliminates the need for provisioning and managing servers. This greatly simplifies the architecture and reduces administrative tasks, allowing me to focus on delivering value to users.

- **Event-Driven Flexibility:** FaaS allows me to respond to specific events, such as user interactions, database changes, or external API calls. This event-driven flexibility enables me to design a more reactive and efficient system, streamlining workflows and optimizing performance.

- **Vendor-Agnostic:** FaaS is generally offered by major cloud providers, such as AWS Lambda, Azure Functions, and Google Cloud Functions. This vendor-agnostic nature gives me the freedom to choose the best fit for my application and allows for potential migration between providers if needed in the future.

3.4. Conclusion

In conclusion, the scope of migration towards the FaaS delivery model for the BStore mobile application is promising. By adopting FaaS, I can optimize costs, focus on business logic, implement an event-driven architecture, reduce development time, and enhance fault tolerance and resilience. The reasons for considering this migration include cost-efficiency, scalability, serverless approach, event-driven flexibility, and the ability to choose from reputable cloud providers. Embracing FaaS aligns with

my goal to deliver a performant, cost-effective, and agile shopping platform for users while enabling me to concentrate on refining the application's features and providing a seamless shopping experience.

While migrating to a FaaS model has its benefits, it's crucial to note that it wouldn't necessarily replace the entire SaaS model for BStore. The SaaS model is still ideal for delivering the complete software package to the end-users. However, by utilizing FaaS for specific components of the back-end infrastructure, I can leverage its benefits while maintaining the comprehensive user-facing features of a SaaS model.

Part IV.

Final Architecture

4. AWS Cloud Architecture

4.1. Architecture diagram

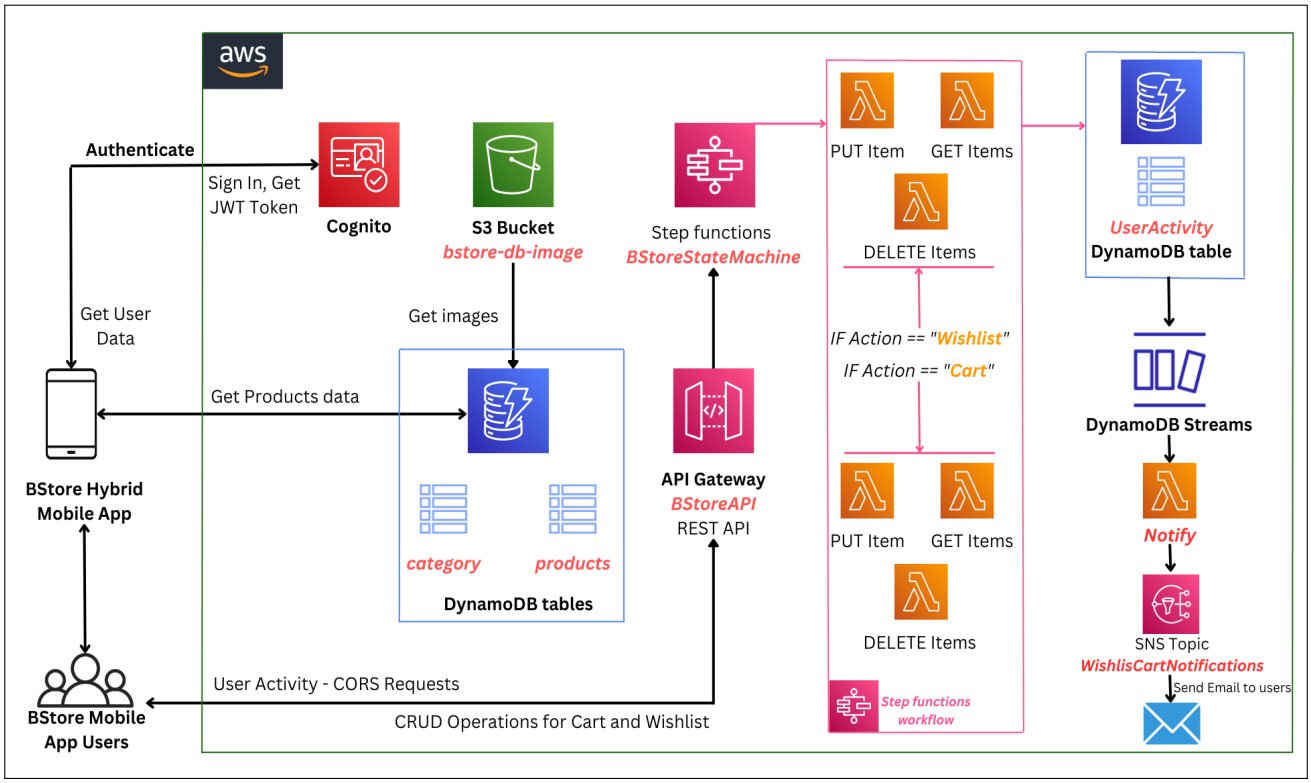

Figure 4.1.: **Cloud Architecture Diagram for BStore mobile application**

4.1.1. Description of my cloud architecture diagram

Cloud Mechanisms and their Integration:

- **AWS Cognito:** I have chosen AWS Cognito for user authentication and managing user state, credentials and identities. Cognito securely stores user credentials and provides features like user sign-up, sign-in, password reset and multi-factor authentication. It seamlessly integrates with other AWS services, allowing users to securely access their cart and wishlist data [1]. Once users

are authenticated, they interact with the application interface, developed using React Native Expo, to browse products, manage their cart, and checkout.

- **Amazon S3:** For storing static product and category images and other assets of the application, I have utilized Amazon S3, a simple and cost-effective storage service. S3 ensures reliable and scalable storage for images used in the application, improving performance and reducing latency for users [2]. I stored all my static images in a bucket called **bstore-db-image** and limiting it's access only to my dynamoDB tables which makes it to improve my data security.
- **Amazon DynamoDB:** As a NoSQL database, DynamoDB stores product information, user data, cart details, and wishlist items [3]. Also it handles CRUD operations for user cart items and wishlist. Its flexible schema and high scalability make it a suitable choice for managing dynamic data in the BStore application. So, I have 3 tables namely: **category, products and UserActivity** to store the category, products and CRUD operations of user on wishlist and cart respectively.
- **AWS Lambda:** I rely on AWS Lambda, a serverless computing service, to execute individual functions in response to specific events [5]. Lambdas handle various functionalities, including processing user requests for adding/removing items to/from the cart or wishlist, fetching product details, and handling notifications. Totally I implemented 7 lambda functions in which 6 of them are for processing operations related to user activity on wishlist and cart, and the remaining 1 is for triggering the SNS topic for sending email notifications for the respective users.
- **Amazon API Gateway:** API Gateway acts as a front-end for AWS Lambda functions, providing a secure and scalable API endpoint for the BStore application. In fact, the APIs are connected to a suite of AWS Lambda functions. It enables users to interact with the backend services, triggering the execution of Lambdas in response to HTTP requests [4].
- **Amazon Step Functions:** For orchestrating the workflow of my Lambdas and coordinating the CRUD operations on DynamoDB tables, I have implemented Step Functions, which allow me to create serverless workflows using state machines [6]. Step Functions help streamline complex business processes and ensure consistency in the application's behavior. This service provides an intuitive visual interface to orchestrate multiple Lambda functions and manage stateful, long-running processes.
- **Amazon SNS:** To send notifications to users for cart or wishlist updates, I utilize Amazon SNS, a messaging service. SNS enables push notifications to be delivered across multiple platforms, ensuring that users stay informed about changes in their shopping preferences and their recent activity on their wishlists and cart items.

4.1.2. Data Storage

My BStore application data is stored in 3 main locations:

- I used **AWS Cognito User Pool** for storing user credentials, and other authentication related information. This cognito user pool contains all the authentication data of all the users of my BStore application.

- I stored all the high resolution static images needed for my application like onboarding, product and category images in a **S3 bucket**.
- Then, all the other respective data related to those products and categories are stored in **DynamoDb** tables called category and products. These tables also contain the object URLs of images stored in S3. By storing all these data in dynamoDB tables, I can ensure quick access and real-time data updates for users.
- Then, if a user performs any activity on his cart or wishlist in the BStore application, then that data will be stored in another dynamoDB table called UserActivity. The data from this table will be further processed by the Notify lambda function to trigger the SNS topic for sending email notifications to the user.

4.1.3. Choosing the right programming Language and Code Implementation

To build the BStore mobile application, I primarily used JavaScript and React Native. JavaScript is a versatile and widely used language in web and mobile development, and React Native allows me to create a cross-platform application with a native look and feel. Specifically, I used Expo framework for my own convenience. With React Native, I can reuse code across iOS and Android platforms, reducing development time and effort.

React Native, a JavaScript framework, was chosen for front-end development due to its cross-platform capabilities and a large community of developers. It allows for sharing of codebase between different platforms (iOS and Android), thereby accelerating the development process.

On the back-end, Python 3.8, is used to write the Lambda functions. This choice contrasts with the use of JavaScript on the front-end, providing a diversified language experience across the application development.

The application's deployment to the cloud is facilitated using AWS CloudFormation. I have used CloudFormation scripts to define the necessary AWS resources, their configurations, and interconnections. Once the template is prepared, it's deployed to AWS, automatically setting up the entire infrastructure.

The parts of the application that required code implementation include:

- **Frontend components:** The user interface of the application, including screens, navigation, and user interactions, is implemented using React Native components.
- **API Integration:** JavaScript code is used to interact with AWS services, such as Cognito, DynamoDB, and Lambda, via API calls. These interactions allow the application to perform user authentication and access, as well as CRUD operations on user data and product details.
- **Business Logic:** JavaScript functions are written to handle business logic, such as cart and wishlist management, user authentication, and event handling for notifications.

4.1.4. System Deployment to the cloud

To begin with, please see the deployment model section 3.1 about my deployment state in this project. However, In a hypothetical scenario, I will deploy this BStore mobile application to the cloud using the services provided by Amazon Web Services (AWS). The deployment process may include the following steps:

- Code Packaging: The React Native application is bundled into a deployable package containing JavaScript code and assets.
- Serverless Deployment: The Lambda functions, along with their dependencies, are packaged into deployable units.
- AWS Infrastructure Setup: AWS CloudFormation is used to define the application's infrastructure as code. This includes setting up Cognito for user authentication, creating DynamoDB tables, and configuring API Gateway, Step Functions, Lambda functions and SNS triggers.
- Deployment Automation: Continuous Integration/Continuous Deployment (CI/CD) pipelines are set up to automate the deployment process. Code changes trigger the CI/CD pipeline, which packages and deploys the application to AWS services. I will include this automation mechanism in the future versions of the project.
- Monitoring and Scaling: After deployment, the system is monitored using AWS CloudWatch to track performance, detect errors, and manage resource utilization. Auto-scaling mechanisms are configured to handle varying workloads.

4.1.5. Architecture Comparison and Rationale

My system aligns with modern cloud-native architecture principles, leveraging serverless technologies and microservices patterns. The architecture emphasizes scalability, cost-efficiency, and rapid development. By adopting serverless computing and managed services, I can focus on building application features and delivering value to users without worrying about managing infrastructure.

The use of AWS Lambda and API Gateway promotes event-driven development and decoupling of components, ensuring flexibility and resilience in the application's workflow. Step Functions help manage complex business logic, orchestrating Lambda functions in a coordinated manner. This architecture enables better fault tolerance and maintainability.

Overall, the chosen architecture for my BStore mobile application embraces cloud best practices and aligns with the specific requirements of a dynamic and scalable e-commerce platform. It empowers me to deliver a performant, secure, and user-friendly shopping experience to customers while benefiting from the inherent advantages of cloud-native solutions.

4.1.6. Basic workflow of my BStore application

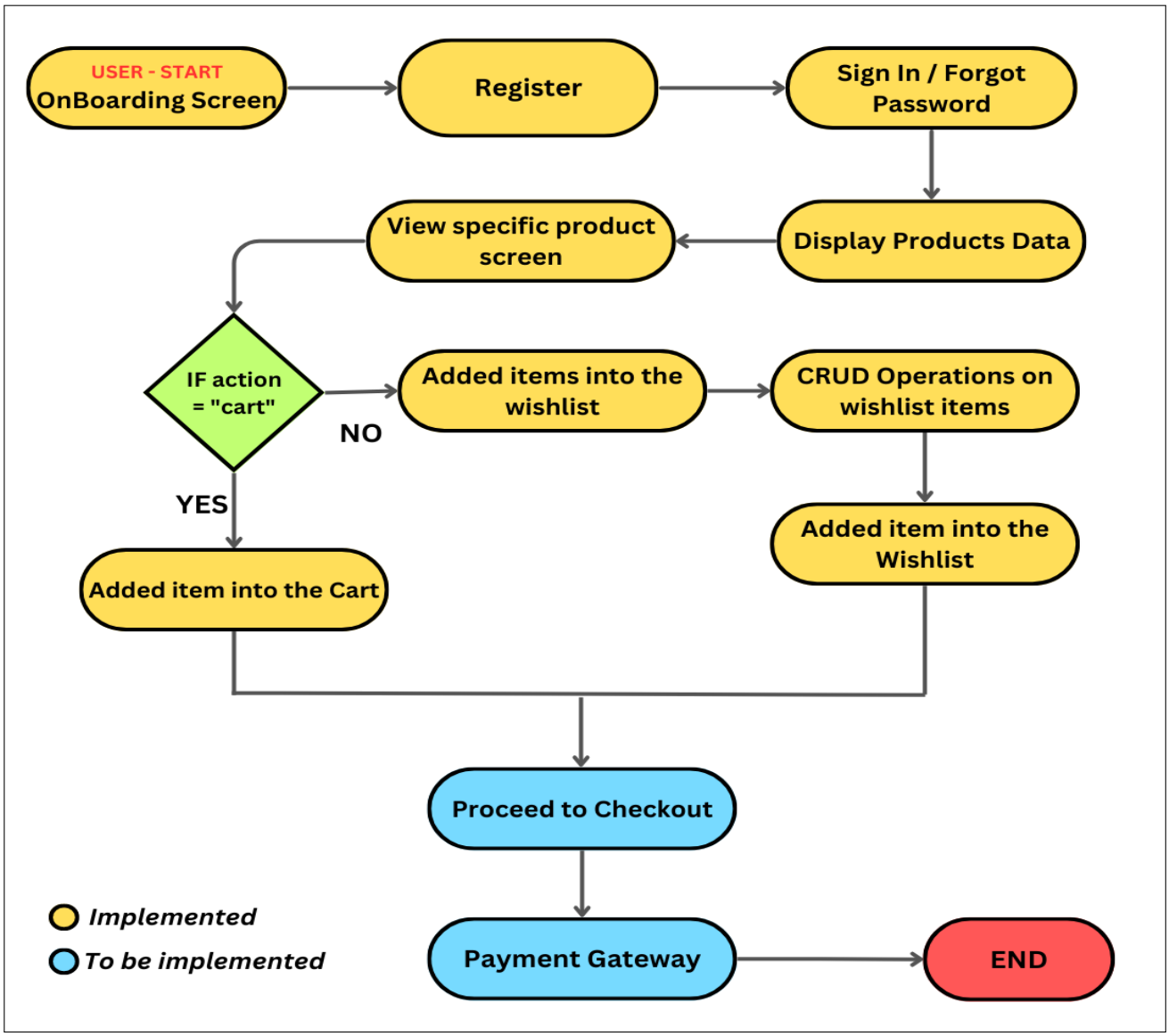

Figure 4.2.: **Basic high-level workflow of my BStore mobile application**

Part V.

Project Demonstration Screenshots

5. Project Implementation

5.1. Demonstration on iOS

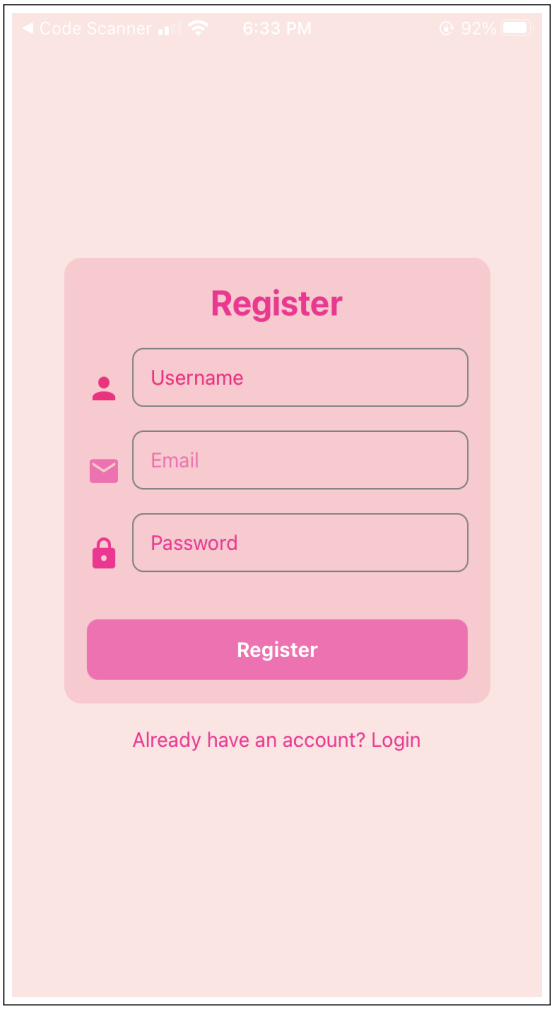

Figure 5.1.: **User Registration Screen**

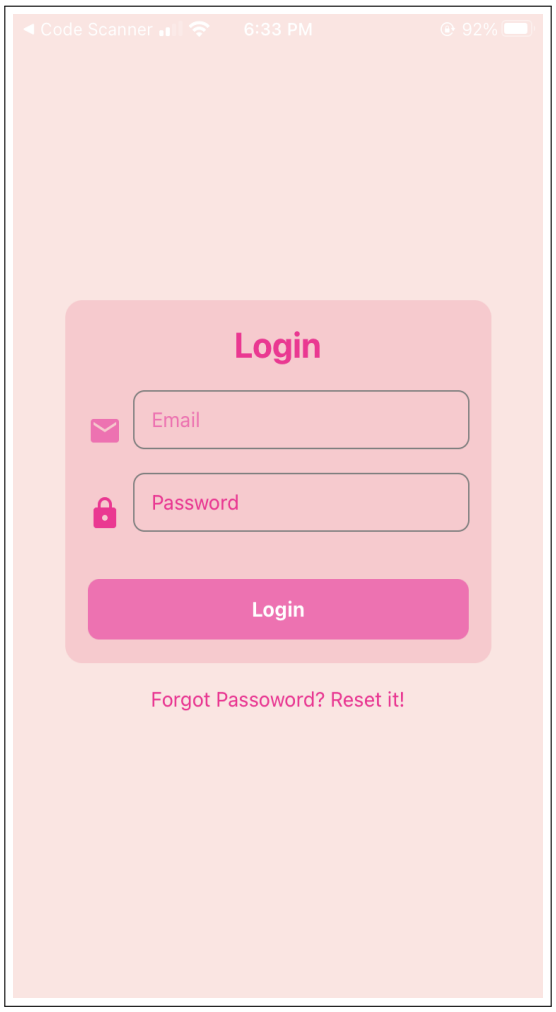

Figure 5.2.: **User Login Screen**

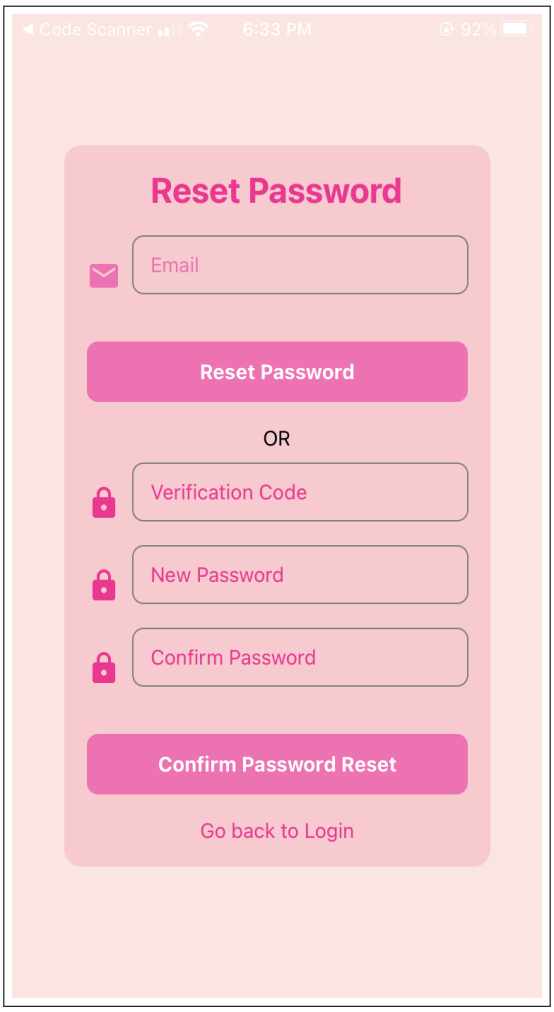

Figure 5.3.: **Password reset screen**

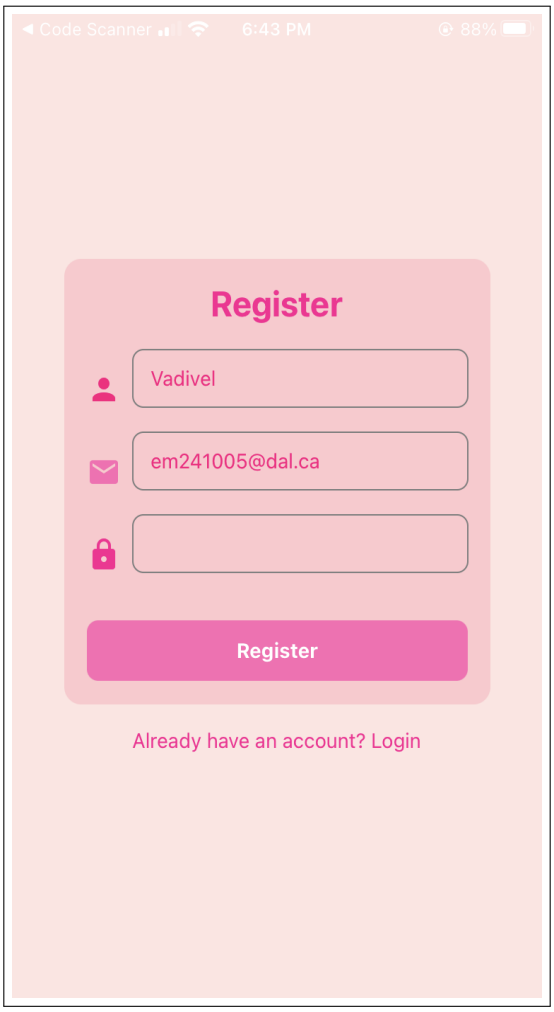

Figure 5.4.: **User registration 1 - Password abstracted in screenshot**

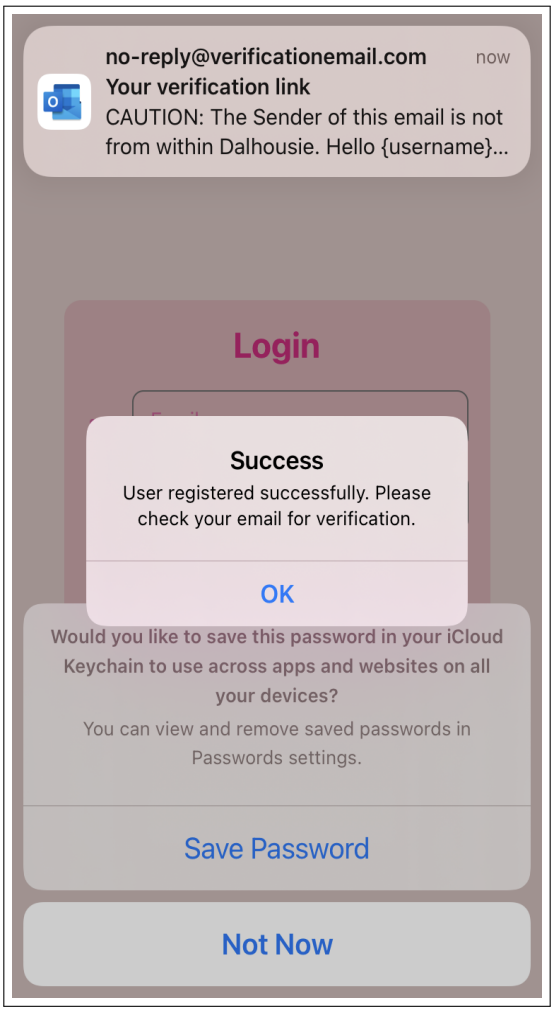

Figure 5.5.: **Registration success and recieved email for verification**

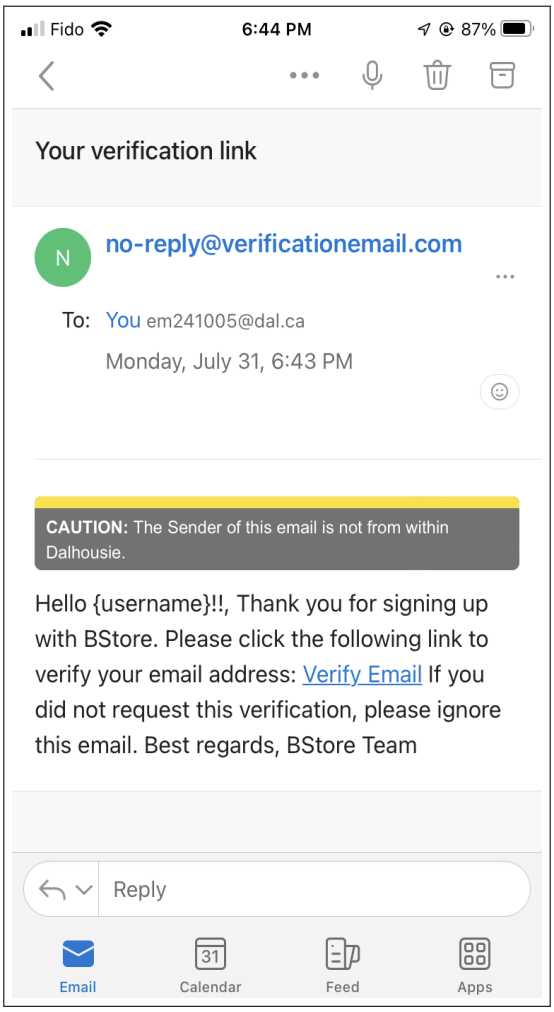

Figure 5.6.: **Verification email for user registration**

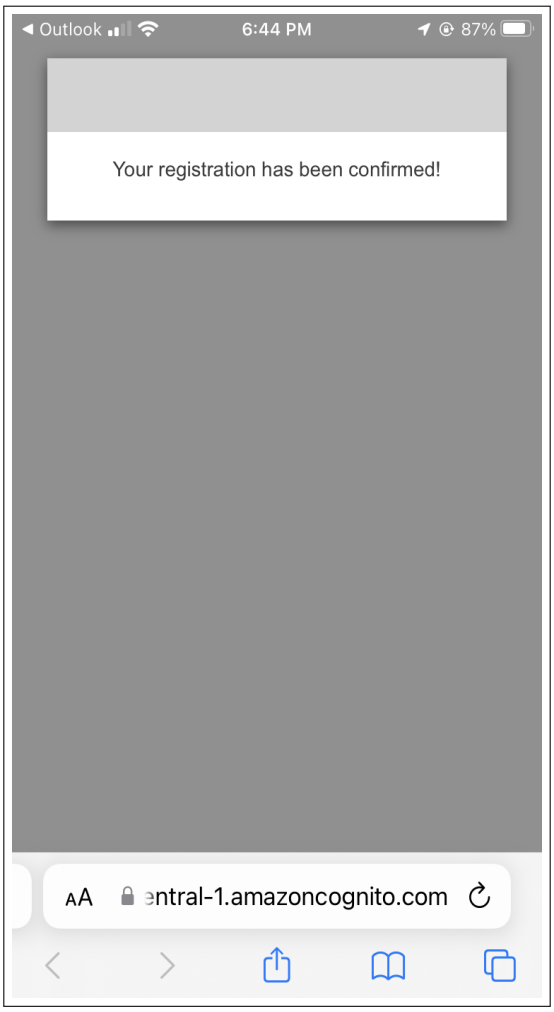

Figure 5.7.: **Confirmed registration - verified email**

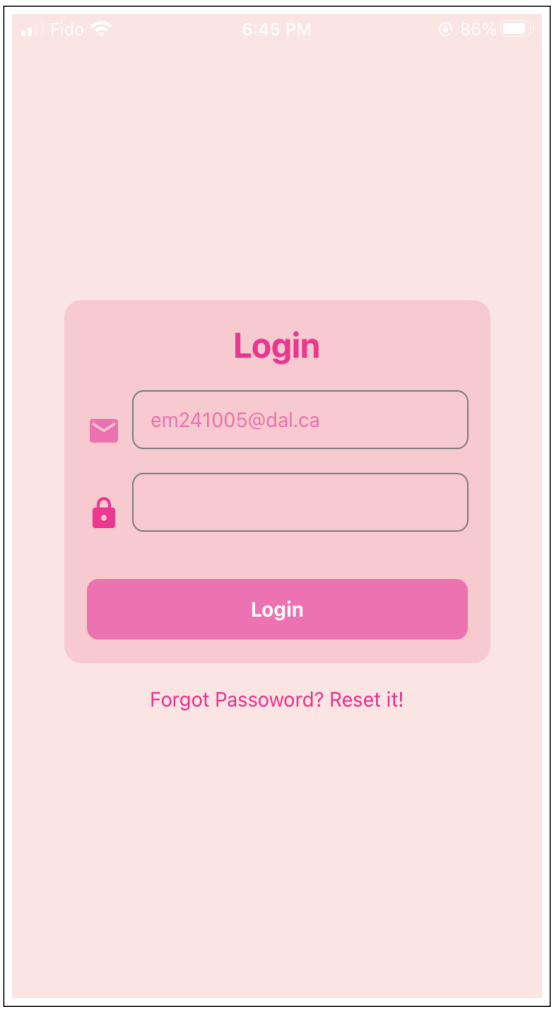

Figure 5.8.: **User Login 1 - Password abstracted in screenshot**

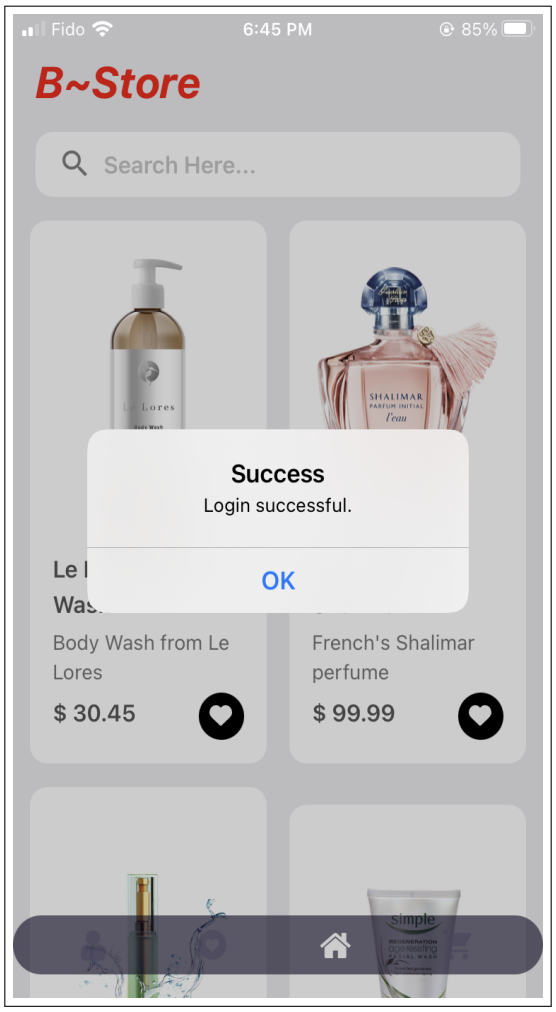

Figure 5.9.: **User Login 1 - Successful**

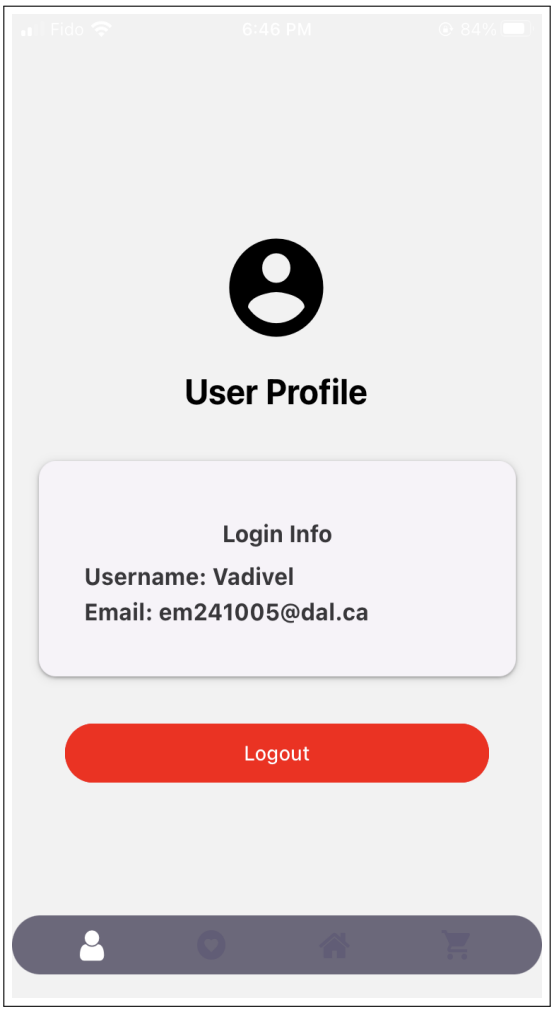

Figure 5.10.: **User profile component**

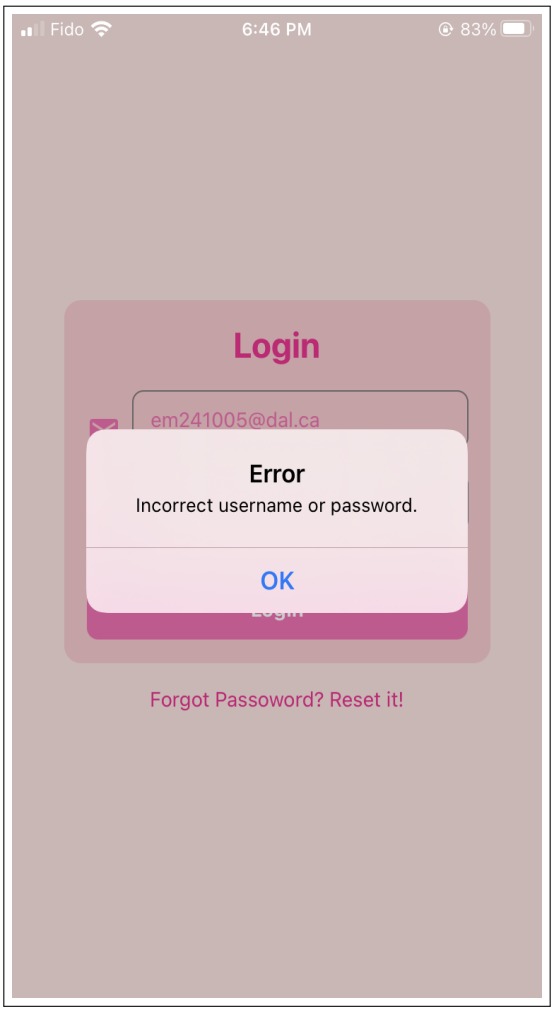

Figure 5.11.: **Logged out and Log in again with invalid credentials**

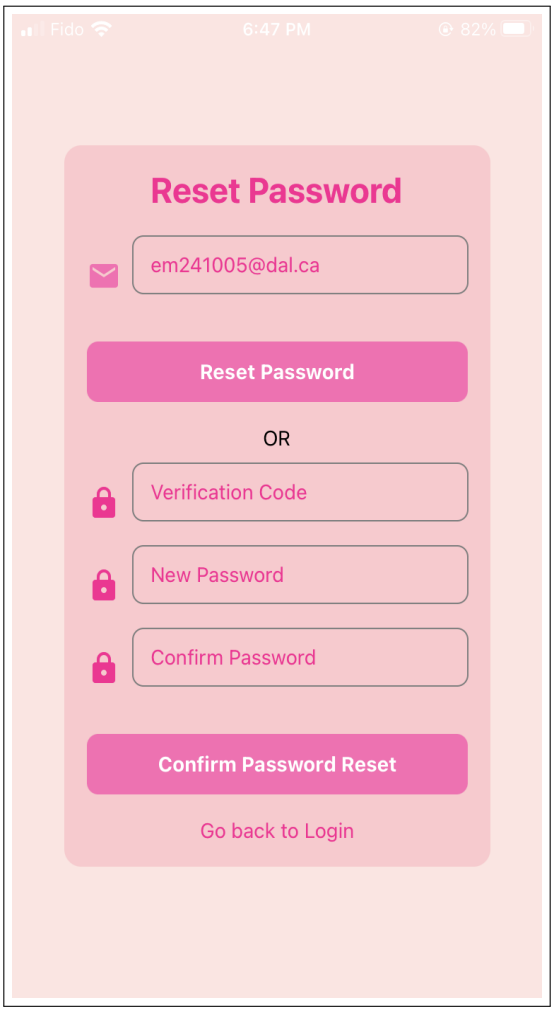

Figure 5.12.: **Password reset component**

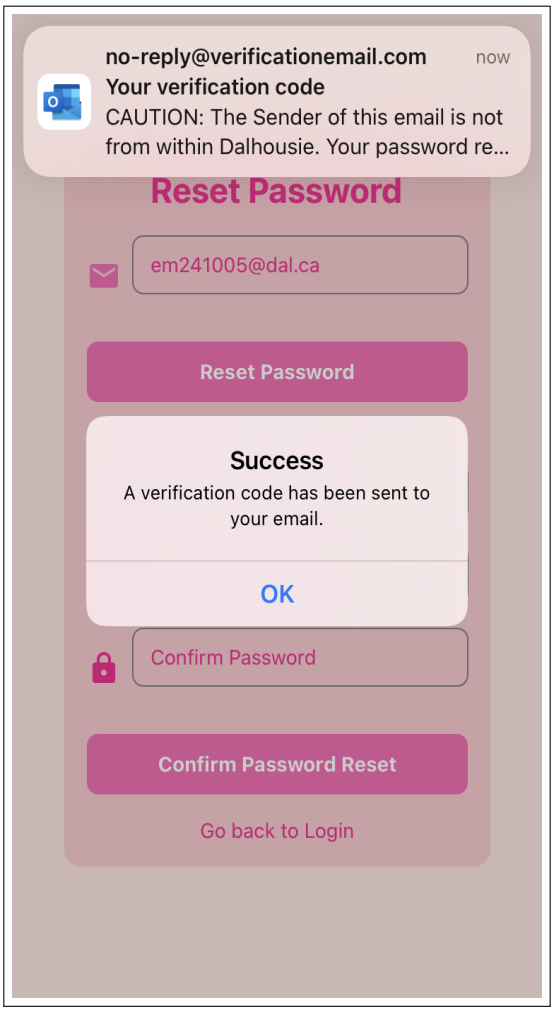

Figure 5.13.: **Password reset verification mail sent successfully**

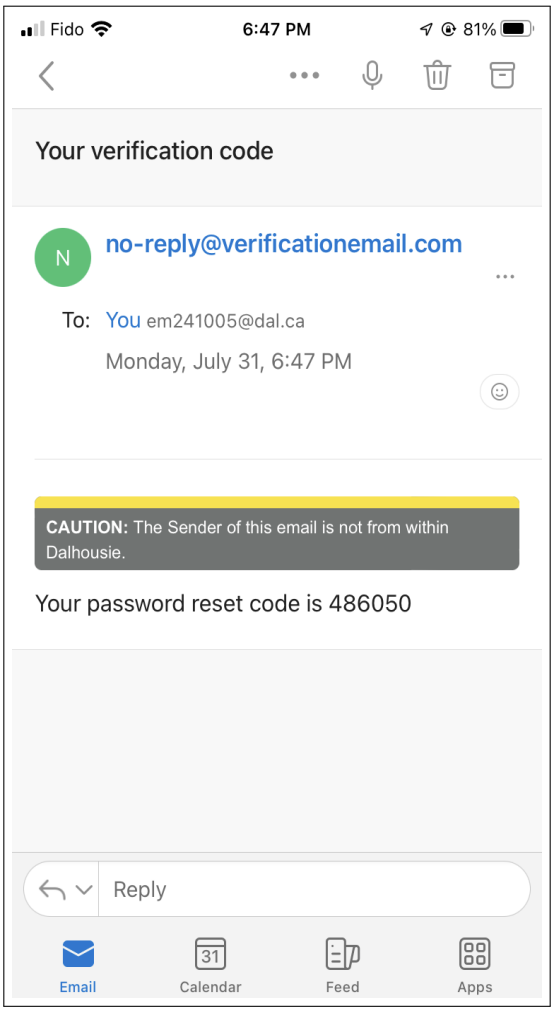

Figure 5.14.: **Password reset verification mail received successfully**

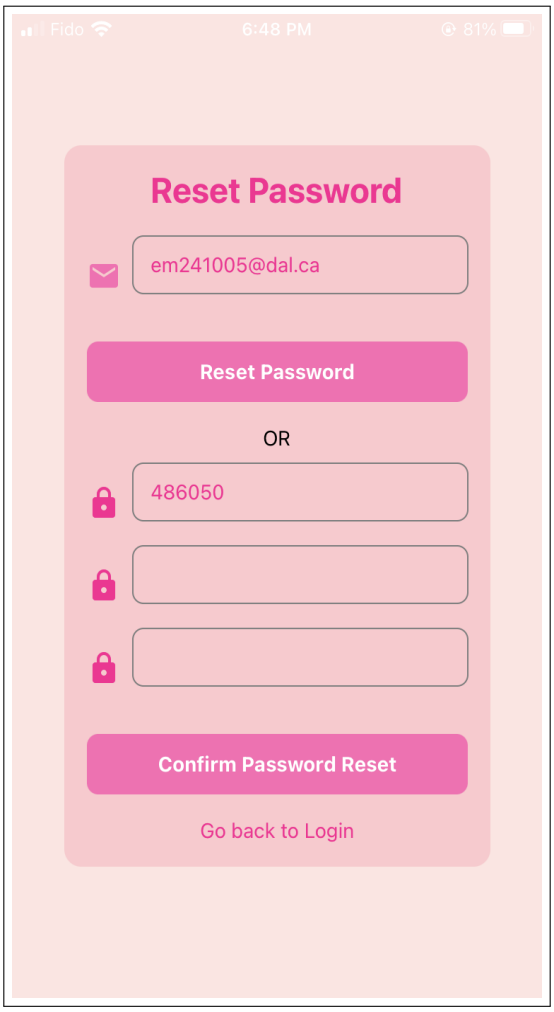

Figure 5.15.: **Entering the code to reset password - Passwords abstracted in screenshot**

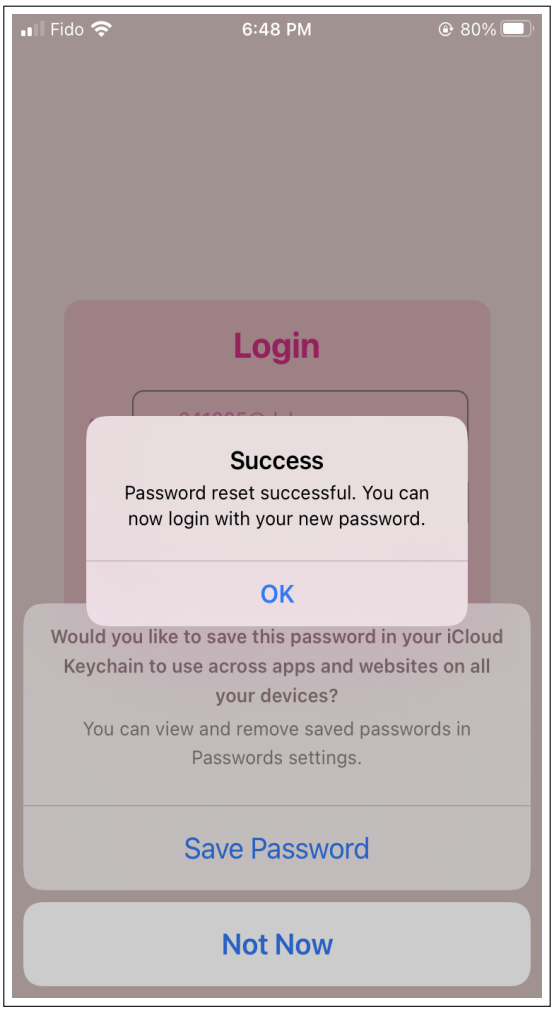

Figure 5.16.: **Password reset successful**

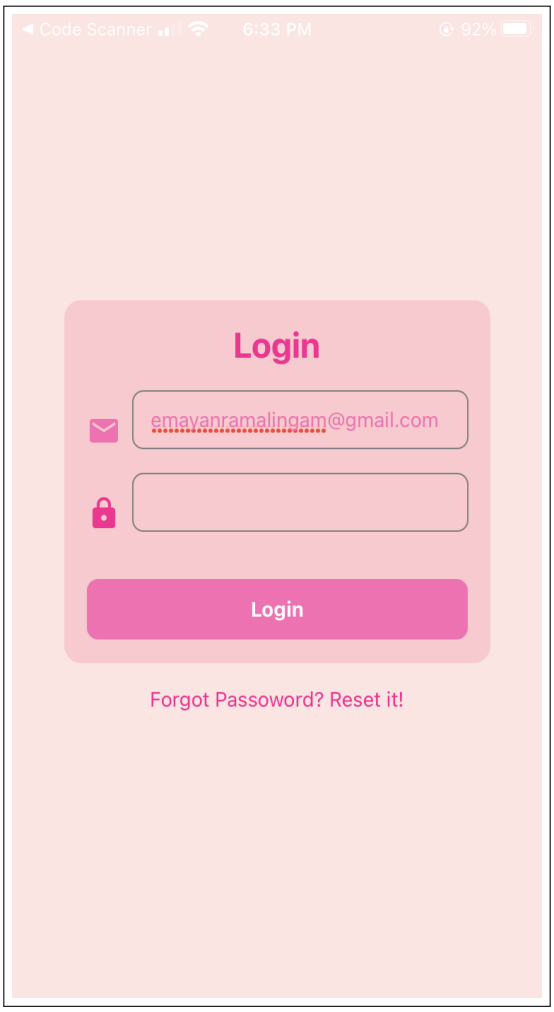

Figure 5.17.: **User Login 2**

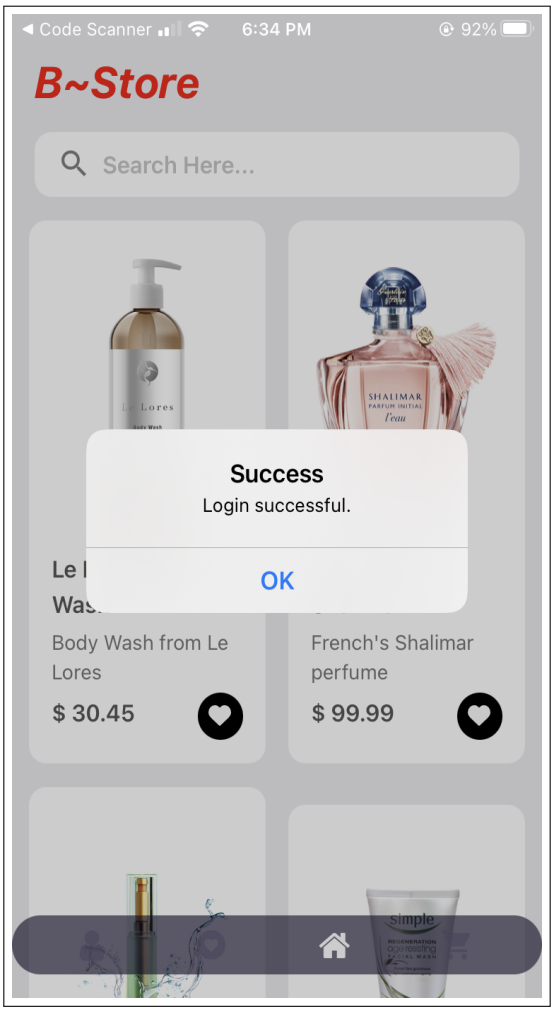

Figure 5.18.: **User Login 2 - successful**

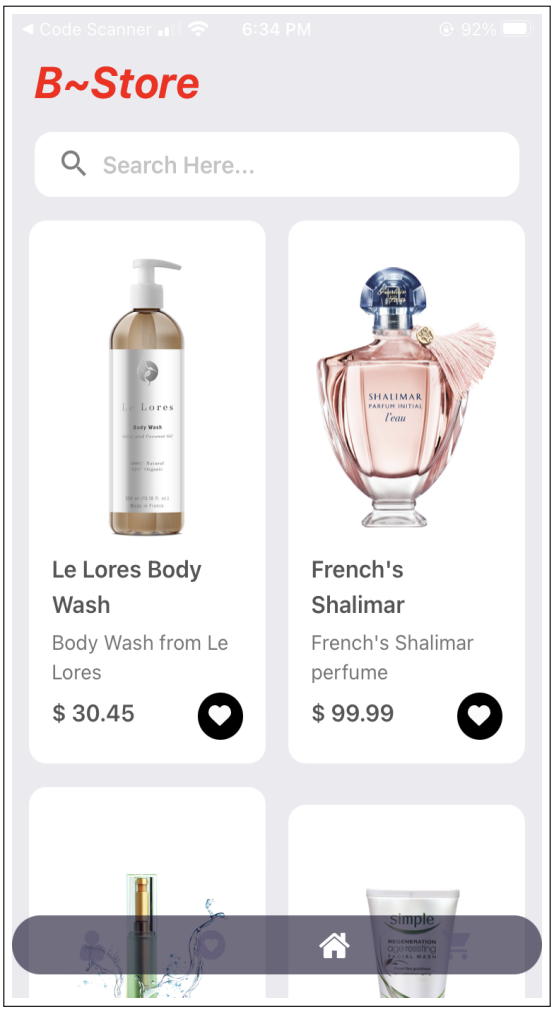

Figure 5.19.: **Home Screen with list of all products - 1**

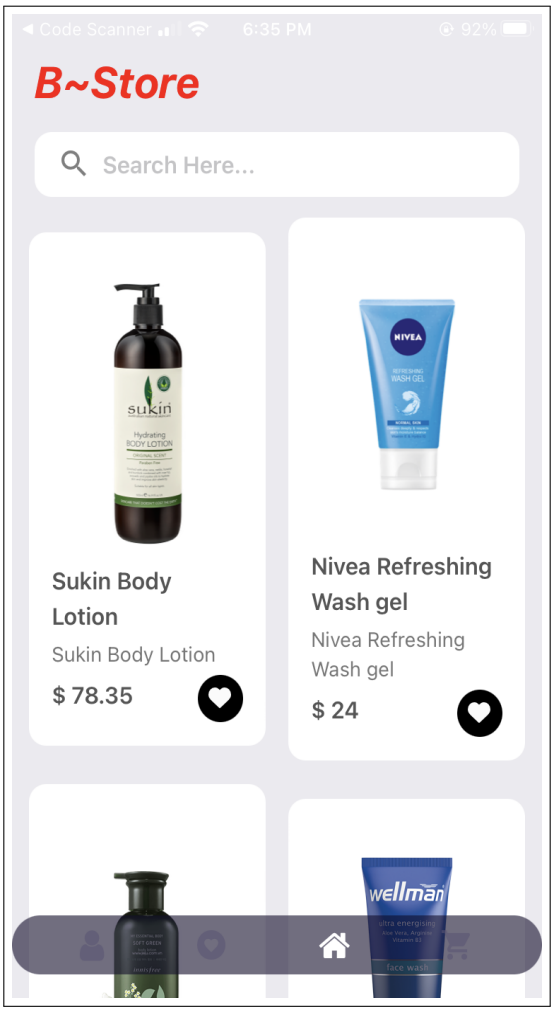

Figure 5.20.: **Home Screen with list of all products - 2**

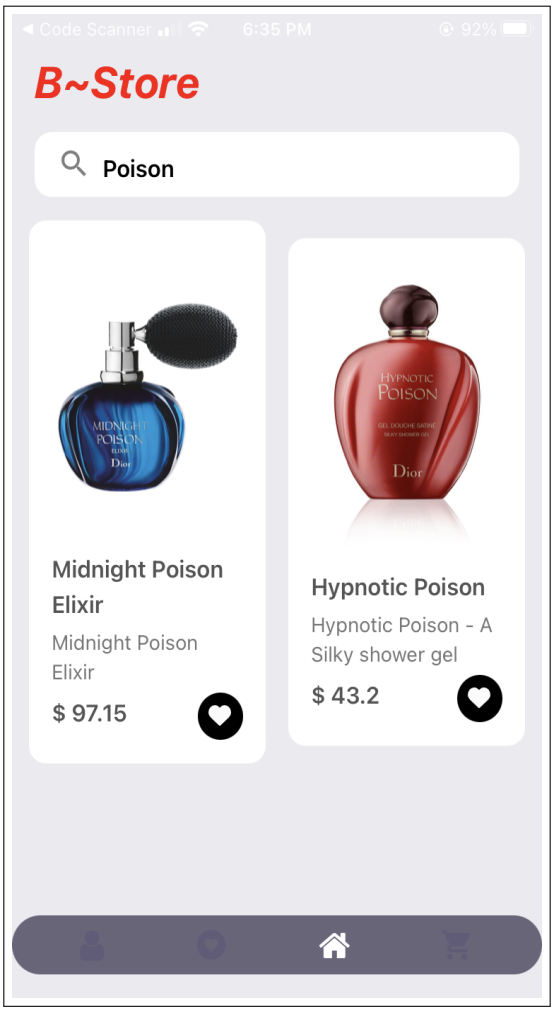

Figure 5.21.: **Filtering items through search bar by product's name - 1**

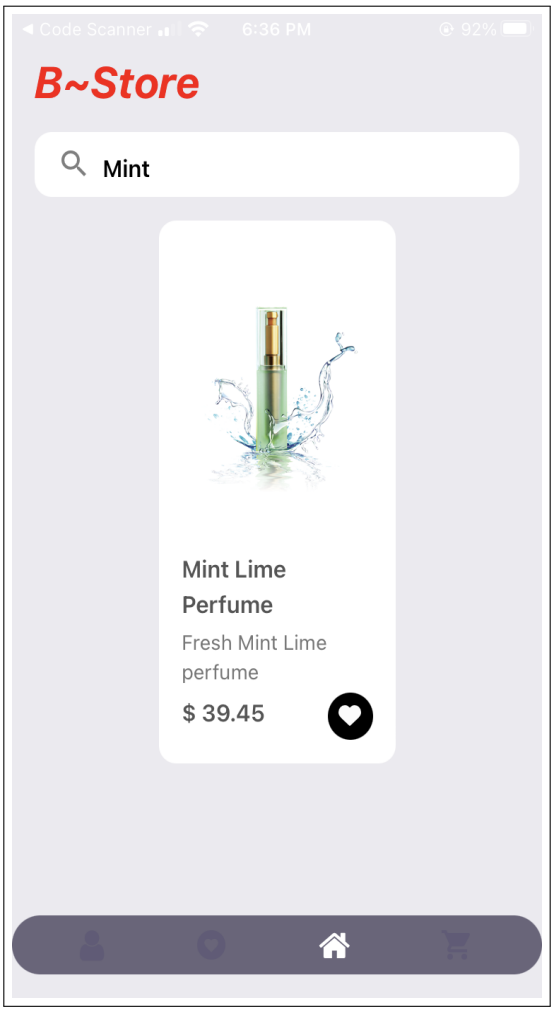

Figure 5.22.: **Filtering items through search bar by product's name - 2**

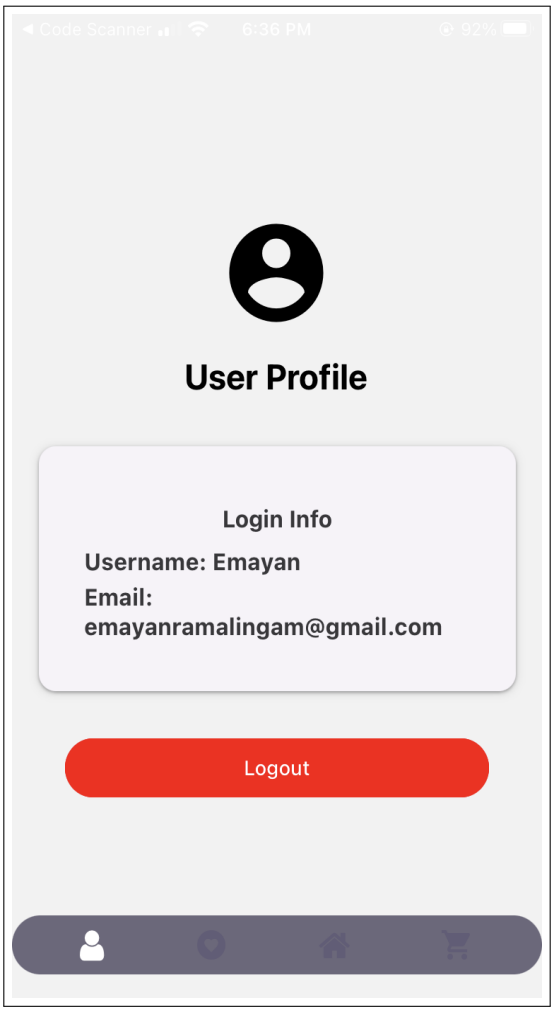

Figure 5.23.: **User profile screen with a Logout button**

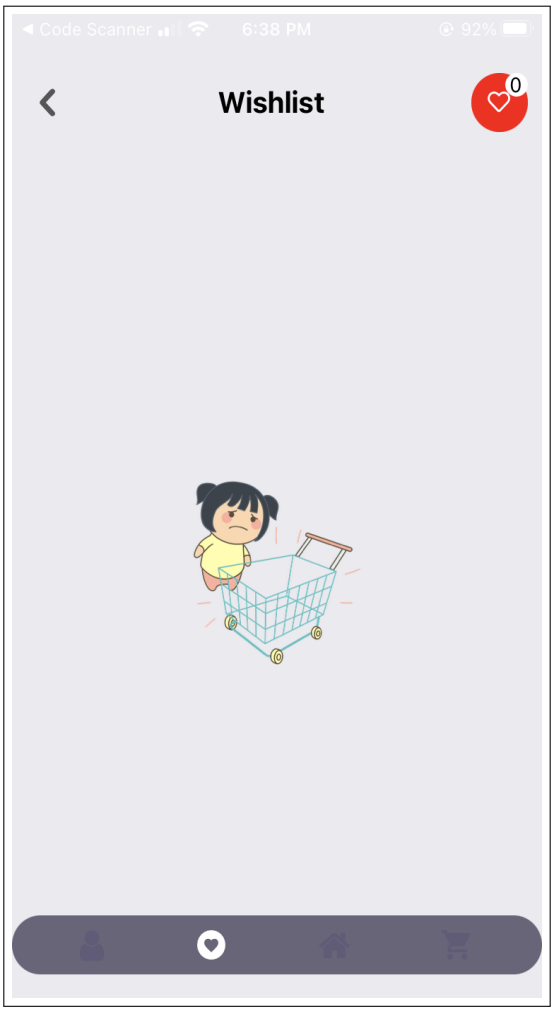

Figure 5.24.: **Empty wishlist**

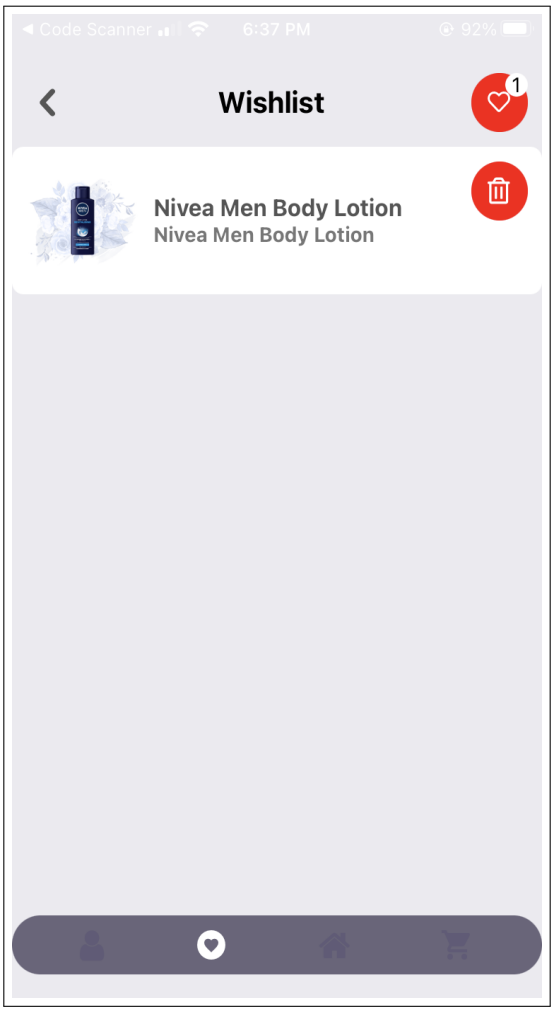

Figure 5.25.: **Added one item into the wishlist**

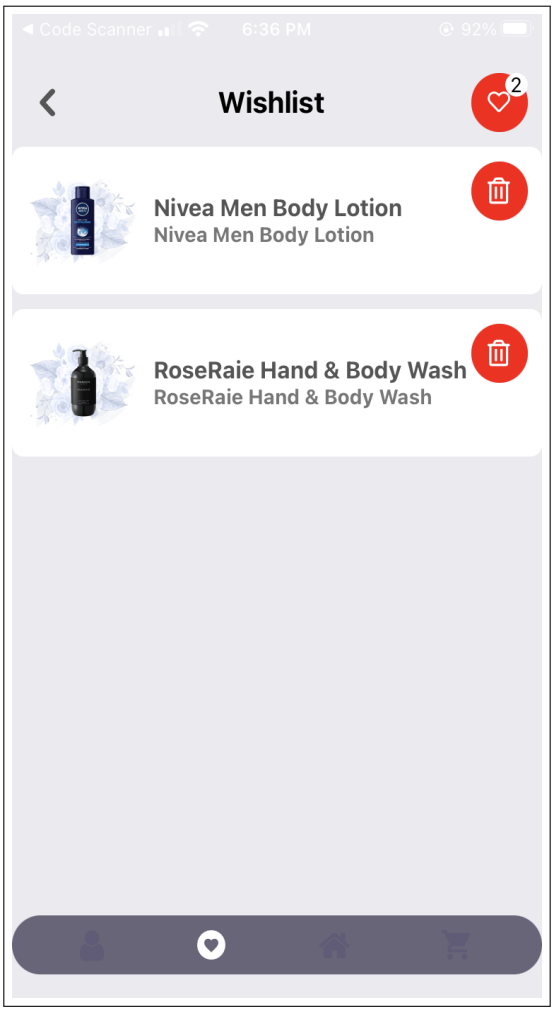

Figure 5.26.: **Added two items into the wishlist**

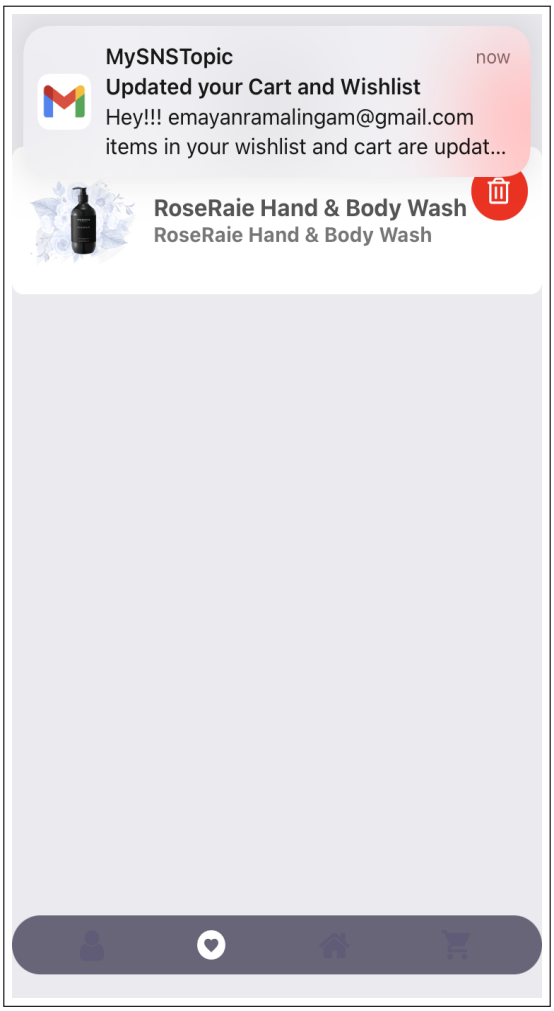

Figure 5.27.: **Got email notification for adding an item into the wishlist**

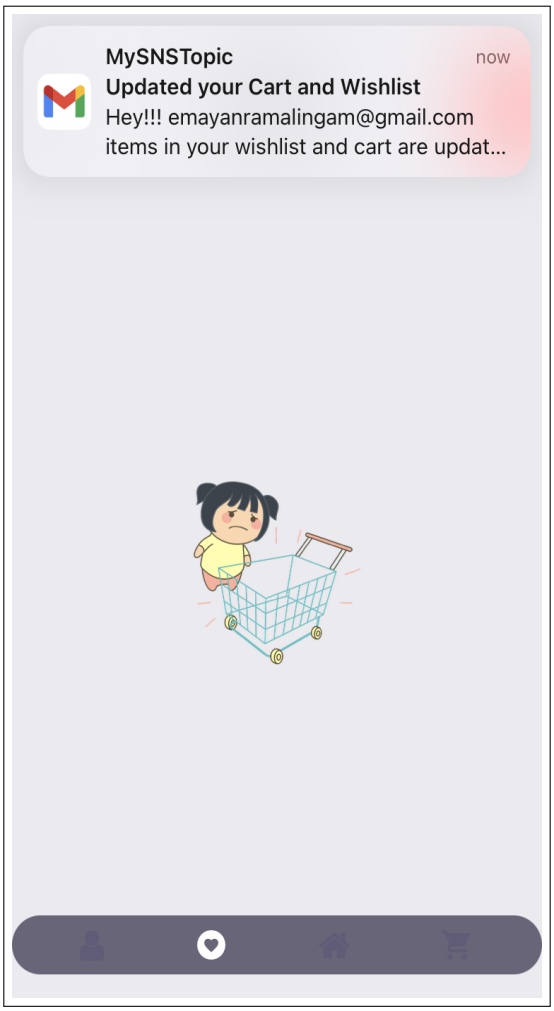

Figure 5.28.: **Got email notification for deleting an item into the wishlist**

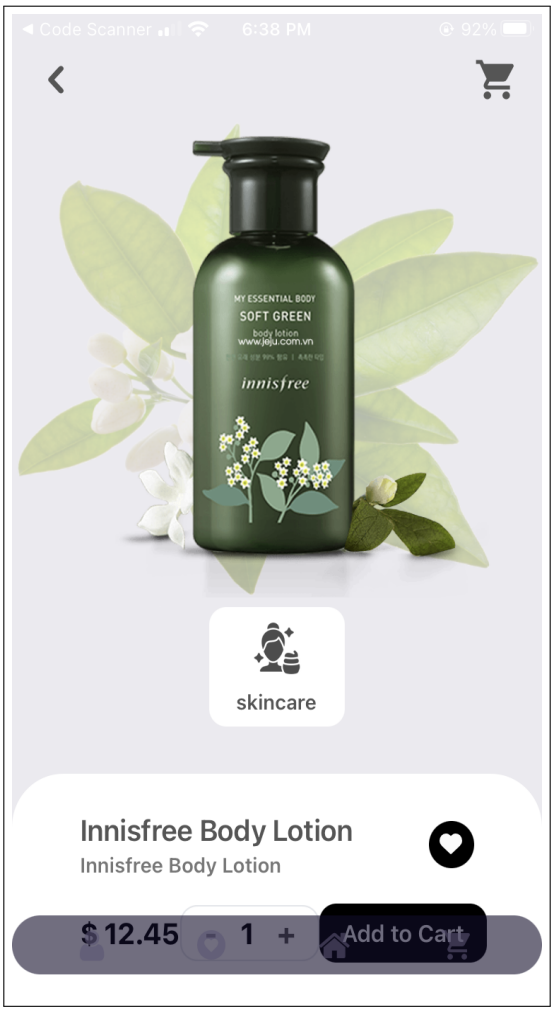

Figure 5.29.: **Product screen rendered with product and background images**

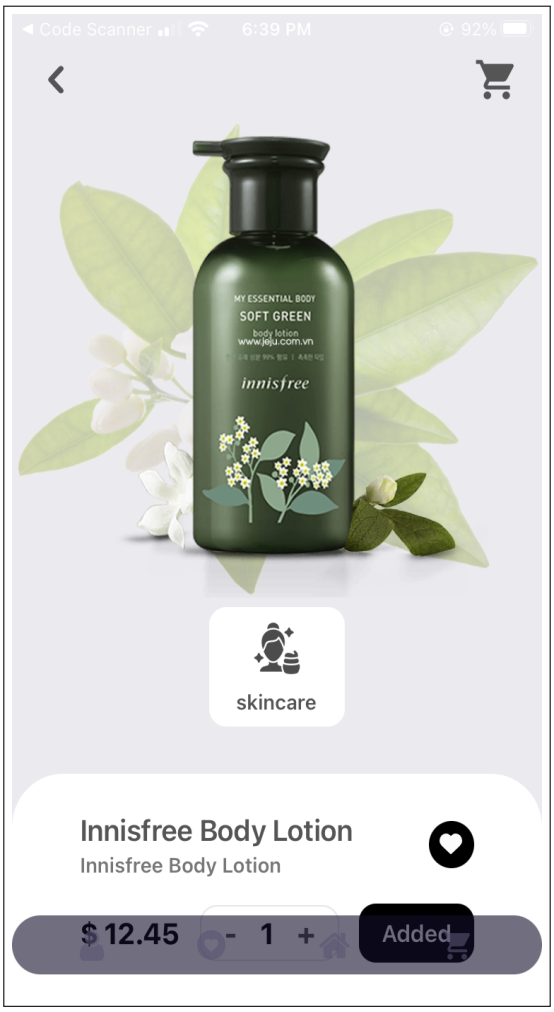

Figure 5.30.: **Added this product into the cart - Add To Cart button**

Figure 5.31.: **Another product screen**

Figure 5.32.: **Added this product into the cart**

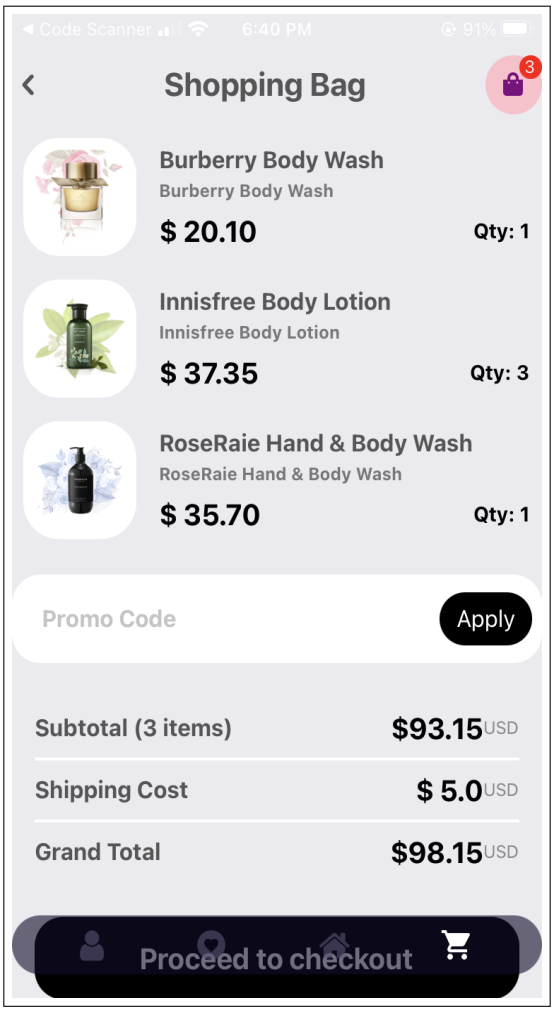

Figure 5.33.: **Cart screen with products and total price**

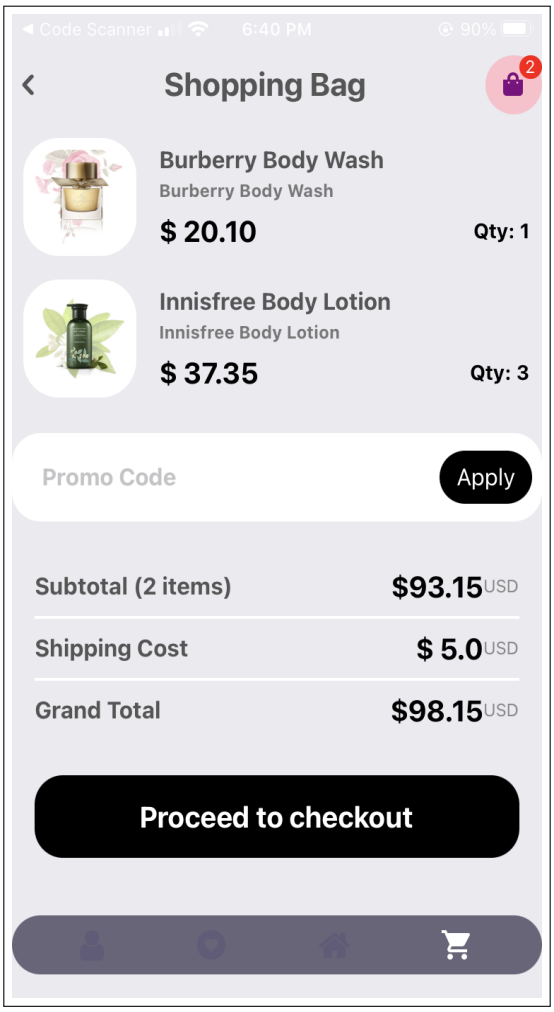

Figure 5.34.: **Deleting cart items - swipe left**

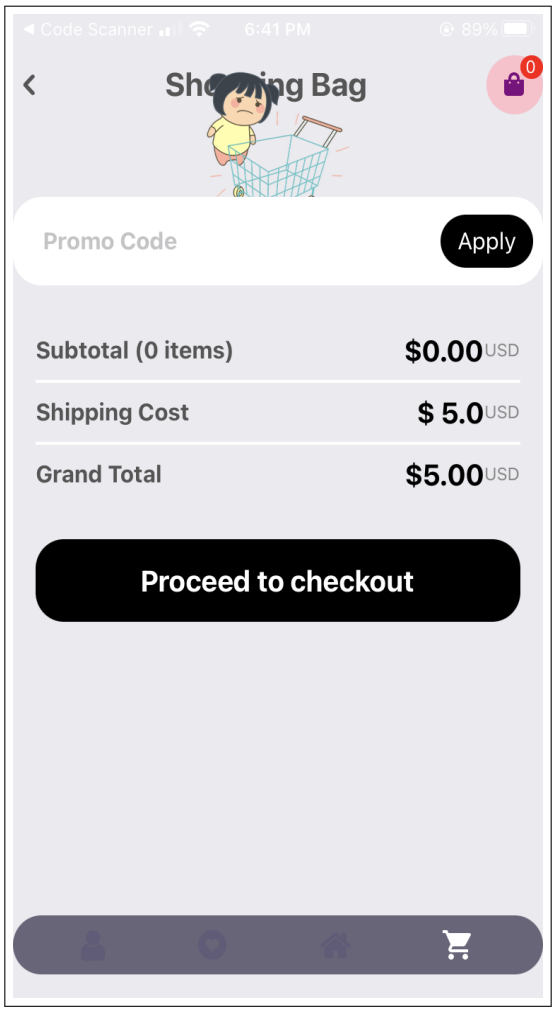

Figure 5.35.: **Empty cart**

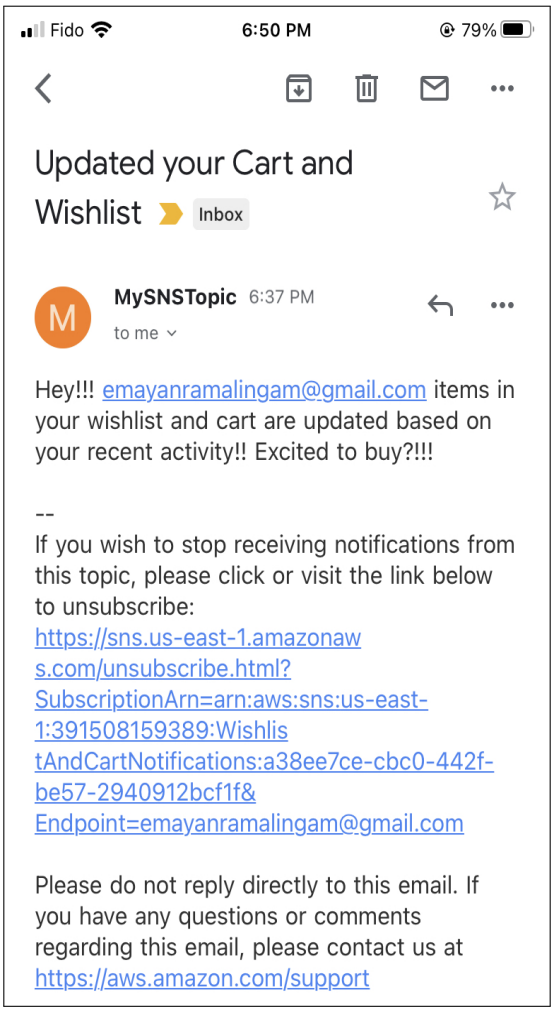

Figure 5.36.: **Got email for user activity - CRUD operations on cart**

Part VI.

Security

6. Data Security

6.1. Security as a top priority

In my BStore mobile application architecture, I consider data security is a top priority, and I have implemented several mechanisms to ensure that data is secure at all layers. Here's an elaborate explanation of how I have addressed data security:

- **Authentication and Authorization:** To protect user data, I have integrated AWS Cognito for authentication and authorization. Cognito provides secure user sign-up and sign-in processes, allowing only authenticated users to access the application's features. Additionally, it supports multi-factor authentication, I will add this as an extra layer of security to user accounts in the future versions of my application [1].
- **Secure Communication:** All communication between the mobile app and the backend services is encrypted using HTTPS. This ensures that data transmitted over the internet is secure and protected from eavesdropping and tampering.
- **Data Encryption:** Sensitive data, such as user passwords and personally identifiable information (PII), is encrypted at rest and in transit. Amazon S3 and Amazon DynamoDB offer encryption options, and I have chosen to enable encryption for these services to safeguard data from unauthorized access [3].
- **Least Privilege Principle:** In AWS IAM (Identity and Access Management), I have implemented the principle of least privilege, granting users and Lambda functions only the permissions they need to perform their specific tasks by creating those resources and access-based policies for only the LabRole. This minimizes the potential impact of a security breach by restricting access to sensitive resources [24].
- **Secure Lambda Execution Environment:** AWS Lambda provides a secure execution environment for serverless functions. Each Lambda function runs in an isolated container, ensuring that code from one function cannot access data from another function. Moreover, the execution environment is automatically patched and updated by AWS, reducing the risk of known vulnerabilities.
- **Input Validation and Sanitization:** I have implemented robust input validation and data sanitization techniques in my Lambda functions to prevent injection attacks and other security vulnerabilities. This ensures that user input is validated before processing, reducing the risk of malicious code execution.

- **Monitoring and Logging:** AWS CloudWatch is utilized to monitor the application and collect logs, providing real-time visibility into potential security threats or anomalies. By monitoring CloudWatch logs, I can quickly detect and respond to security incidents.

6.2. Addressing existing Vulnerabilities

One potential vulnerability could be the security of the client-side code and the storage of sensitive data in the device's local storage. React Native Expo, the framework used for building the BStore application, compiles into native code, which can be reverse-engineered to potentially expose secrets embedded in the code. To minimize this risk, I avoided storing sensitive information such as user passwords or API keys in the application code or device storage. Instead, these are securely managed on the server side or using secure cloud services.

Another vulnerability can be a DDoS (Distributed Denial of Service) attack. AWS offers services like AWS Shield and AWS WAF (Web Application Firewall) to mitigate such attacks. I plan to leverage these services in the future to add an extra layer of security to the application.

6.3. Security mechanisms used

- AWS Cognito for user authentication and secure password storage.
- HTTPS (TLS/SSL) for secure data transmission.
- IAM roles and policies for fine-grained access control (Usage of LabRole).
- Data Encryption at Rest and in Transit: Encryption is enabled for Amazon S3 and Amazon DynamoDB.
- AWS Lambda Execution Environment: Lambda's isolated execution environment ensures that each function runs independently and securely. This prevents any unauthorized access to resources or data leakage between functions.
- AWS KMS for encryption and decryption of data stored in DynamoDB.

Each of these mechanisms was chosen for its ability to deliver robust security features while allowing me to maintain focus on the core application functionality. By incorporating these security mechanisms, my application architecture is designed to protect sensitive data at all layers and reduce the risk of data breaches or unauthorized access. However, security is an ongoing concern, and continuous monitoring and updates are essential to address emerging threats and maintain a secure environment. Regular security audits and penetration testing can further identify and remediate potential vulnerabilities, ensuring the safety of user data and the application as a whole.

Part VII.

Estimated Cost

7. Estimated Cost

7.1. Estimated cost for my architecture on a Private cloud

To reproduce my BStore mobile application architecture in a private cloud, I would need to carefully consider the required hardware and software components to achieve a similar level of availability as my cloud implementation. While the exact costs can vary based on various factors, I will provide a rough estimate and outline the key elements I would need to purchase:

7.1.1. Hardware

- **Servers and Virtualization Software:** To host my application on-premise, I would need to invest in a cluster of high-performance servers with sufficient computing power, memory, and storage capacity to handle the application's workload. Additionally, I would need virtualization software like VMware or Microsoft Hyper-V to create and manage virtual machines on these servers [19]. To host databases, APIs, authentication services, and serve static files, multiple high-performance servers would be required. The cost can vary greatly depending on the specifications, but let's estimate about $5,000 - $10,000 per server. Assuming we need at least 5 servers for redundancy and load balancing, the cost would be around $25,000 - $50,000.

- **Networking Equipment:** A robust and redundant networking infrastructure is crucial for ensuring seamless communication between different components of the application. This includes network switches, routers, firewalls, load balancers, and other networking equipment. Routers, switches, load balancers, firewalls, and other networking gear would be needed to manage network traffic and security [11]. This could add another $10,000 - $20,000.

- **Storage:** Similar to my cloud architecture, I would need on-premise storage solutions, such as Network-Attached Storage (NAS) or Storage Area Network (SAN), to store static product images, user data, and other application-related files. For data storage and backup, we'd need high-capacity, reliable storage systems. Enterprise-level storage systems can range from $5,000 to over $100,000. For our rough estimate, let's allocate $25,000 for storage.

7.1.2. Software

- **Database Server and Software:** I would require a dedicated database server to host the data stored in my application. Depending on the database technology used, I might need to purchase licenses for commercial database software or consider open-source alternatives. I'd need a NoSQL database system to replace DynamoDB. There are open-source options like Apache Cassandra, but they come with significant setup, maintenance, and scaling challenges.

- **Server Software:** To replace Lambda and API Gateway, we'd need a server and API management software. Options like Node.js and Express.js are available for free but require significant setup and maintenance work.
- **Authentication:** To replace Cognito, an identity provider like Keycloak could be used. Again, while it's free, it comes with setup and maintenance costs.
- **Notification System:** Replacing SNS would require a messaging system like RabbitMQ or an email service. These can be free or have small costs but need maintenance.
- **Security Tools and Solutions:** To maintain data security and protect against potential threats, I would need to invest in security tools like antivirus software, intrusion detection systems (IDS), and encryption technologies.
- **High Availability and Load Balancing:** Achieving a similar level of availability to the cloud implementation would require implementing high availability and load balancing mechanisms on-premise. This might involve purchasing redundant hardware, failover systems, and load balancers to distribute incoming traffic.
- **Monitoring and Management Software:** To monitor the performance and health of the on-premise infrastructure, I would need monitoring and management software that provides real-time insights into resource utilization, application performance, and potential issues.

Backup and Disaster Recovery Solutions: As part of ensuring data resilience and disaster recovery capabilities, I would need to invest in backup solutions and off-site data replication to protect against data loss.

Skilled IT Staff - Manpower: Building and maintaining an on-premise infrastructure of this scale would require a team of skilled IT professionals, including system administrators, network engineers, database administrators, and security experts. Setting up and maintaining this infrastructure would require a team of skilled IT professionals, including system administrators, network engineers, and database administrators. Considering an average salary of $70,000 per year per person and a minimum team of 5 people, that's an annual cost of $350,000.

7.1.3. Total estimated cost

It is challenging to provide an exact cost estimate, as it depends on factors like the size of the infrastructure, hardware specifications, software licenses, and the region's costs. Adding up the hardware, software, and manpower costs, the total comes to around $410,000 - $445,000 for the initial setup, and $350,000 per year for maintenance, not including electricity, cooling, and other operational costs.

Please note these are very rough estimates and the actual cost could be significantly higher or lower depending on various factors. This calculation illustrates the significant costs and complexities involved in setting up a private cloud infrastructure that provides similar functionality and availability as the existing AWS-based solution.

My decision to use AWS services for the BStore application was largely driven by the ability to access a wide range of services with high availability and scalability at a fraction of the cost of setting up and maintaining a similar infrastructure in a private cloud.

7.1.4. Considerations

While reproducing the architecture on-premise is possible, it is essential to consider the potential challenges and drawbacks. Private cloud infrastructure requires substantial upfront investment and ongoing maintenance costs. It may also lack the scalability and flexibility of the public cloud, making it difficult to handle sudden spikes in traffic or scale down during periods of low demand. Additionally, managing an on-premise infrastructure involves dealing with physical hardware, which introduces additional complexities and responsibilities for ensuring hardware reliability and replacement.

In conclusion, while the option of a private cloud deployment is viable, I would carefully weigh the benefits and drawbacks before deciding on an on-premise solution. For my BStore mobile application, the public cloud with its robustness, scalability, and managed services has proven to be a more suitable and cost-effective choice.

Part VIII.

Future RoadMap

8. Application Development: Next Steps

8.1. New set of features

If I were to continue the development of my BStore mobile application, my primary focus would be on improving the overall user experience and functionality of the application. To achieve this, I would add several new features and optimize existing ones. The following list of points will describe my next steps and future roadmap for the implementation of the new set of features in my application:

- **Stripe Payment Gateway Integration:** Implementing the Stripe Payment Gateway would enable users to securely add their debit/credit card information and make payments during the checkout process [10]. For secure checkout and to provide a seamless user experience, I would integrate Stripe as a payment gateway. Stripe has a well-documented API and provides high levels of security, making it an ideal choice. Stripe's developer-friendly documentation and robust security features make it a suitable choice for seamless payment transactions [10]. To achieve this, I would use AWS Lambda to create serverless functions for handling payment processing. I would store transactions and order data in DynamoDB, allowing for fast and efficient retrieval.
- **Improved User Profile Management:** This is crucial for enhancing the personalized shopping experience. I'd introduce a more extensive user profile section where users can manage their personal details, order history, payment methods, and shipping addresses. AWS Cognito would be used to securely manage user profiles, while DynamoDB would store user-related data [1].
- **Order Tracking:** Implementing real-time order tracking would significantly enhance the user experience. Users would be able to track the status of their orders from the warehouse to their doorstep. AWS Step Functions could be used here to manage the state of each order, and updates can be pushed to users via AWS SNS.
- **Promo Code Implementation:** To drive sales and improve customer retention, I'd introduce a system for promotional codes that users can apply for discounts. This would require updates to the current pricing logic and can be handled via AWS Lambda functions. DynamoDB would be used to store and validate promo codes. When a user applies a promo code, a Lambda function would check its validity in DynamoDB and calculate the discounted price before finalizing the order [5].
- **Personalized Recommendations:** I would introduce a feature to provide personalized product recommendations based on the user's browsing history and purchase behavior. To implement this, I could use Amazon Personalize, a real-time personalization and recommendation service powered by machine learning.

- **Product Reviews and Ratings:** Allowing users to review and rate products can not only aid other users in making purchase decisions but can also help in improving the quality of products on offer. I would use DynamoDB to store reviews and ratings and AWS Lambda to handle the serverless logic of posting and retrieving them.
- **Enhanced Search Feature:** A robust search function is crucial for any online marketplace. I would enhance the existing search feature by providing more filters and sort options. I could use Amazon Elasticsearch for this purpose, which would enable a scalable and flexible search functionality.
- **In-app Chat Support:** To enhance user engagement and customer support, I would consider adding an in-app chat support feature. Users can interact with customer support representatives in real-time to resolve queries, seek assistance, or provide feedback. For this feature, I could leverage AWS Simple Notification Service (SNS) and Amazon API Gateway to send and receive messages, and AWS Lambda to handle chat-related logic [4, 7].

In conclusion, By continuing the development in the above-mentioned direction, I hope my BStore mobile application will transform into a more comprehensive, user-friendly, and feature-rich shopping application. As I continue developing BStore, I will prioritize feature implementations like Stripe Payment Gateway integration, enhanced user profile management, order tracking, and promo code functionality to elevate the application's capabilities and user experience. Moreover, introducing in-app chat support as an additional feature would further solidify the application's position as a user-centric and customer-oriented platform. Through these advancements, I aim to establish BStore as a leading mobile application in the market, while also demonstrating my expertise in cloud-based application development and user-focused feature design.

Part IX.

REFERENCES

References

[1] Amazon Web Services, Inc., "AWS Cognito," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/cognito/. [Accessed: July 31, 2023].

[2] Amazon Web Services, Inc., "AWS S3," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/s3/. [Accessed: July 31, 2023].

[3] Amazon Web Services, Inc., "AWS DynamoDB," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/dynamodb/. [Accessed: July 31, 2023].

[4] Amazon Web Services, Inc., "AWS API Gateway," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/api-gateway/. [Accessed: July 31, 2023].

[5] Amazon Web Services, Inc., "AWS Lambda," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/lambda/. [Accessed: July 31, 2023].

[6] Amazon Web Services, Inc., "AWS Step Functions," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/step-functions/. [Accessed: July 31, 2023].

[7] Amazon Web Services, Inc., "AWS SNS," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/sns/. [Accessed: July 31, 2023].

[8] Facebook, "React Native," Facebook. [Online]. Available: https://reactnative.dev/. [Accessed: July 31, 2023].

[9] Expo, "Expo," Expo. [Online]. Available: https://expo.dev/. [Accessed: July 31, 2023].

[10] Stripe, "Stripe: Payment Gateway API," Stripe. [Online]. Available: https://stripe.com/docs/api. [Accessed: July 31, 2023].

[11] Docker, "Docker: Empowering App Development for Developers," Docker. [Online]. Available: https://www.docker.com/what-docker. [Accessed: July 31, 2023].

[12] Kubernetes, "Kubernetes (K8s): Production-Grade Container Orchestration," Kubernetes. [Online]. Available: https://kubernetes.io/. [Accessed: July 31, 2023].

[13] Amazon Web Services, Inc., "AWS IAM," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/iam/. [Accessed: July 31, 2023].

[14] Amazon Web Services, Inc., "AWS App Runner," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/apprunner/. [Accessed: July 31, 2023].

[15] Amazon Web Services, Inc., "AWS SWF," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/swf/. [Accessed: July 31, 2023].

[16] Amazon Web Services, Inc., "AWS EC2," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/ec2/. [Accessed: July 31, 2023].

[17] Amazon Web Services, Inc., "AWS EFS," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/efs/. [Accessed: July 31, 2023].

[18] Amazon Web Services, Inc., "AWS RDS," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/rds/. [Accessed: July 31, 2023].

[19] Amazon Web Services, Inc., "AWS SQS," Amazon Web Services, Inc. [Online]. Available: https://aws.amazon.com/sqs/. [Accessed: July 31, 2023].

[20] Redux, "Redux: A Predictable State Container for JS Apps," Redux. [Online]. Available: https://redux.js.org/. [Accessed: July 31, 2023].

[21] Microsoft Azure, "What is a public cloud?" Microsoft Azure. [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-a-public-cloud/. [Accessed: July 31, 2023].

[22] Salesforce, "What is SaaS?" Salesforce. [Online]. Available: https://www.salesforce.com/saas/. [Accessed: July 31, 2023].

[23] IBM Cloud, "What is Function as a Service (FaaS)?" IBM Cloud. [Online]. Available: https://www.ibm.com/cloud/learn/faas. [Accessed: July 31, 2023].

[24] Amazon Web Services, "AWS Well-Architected Framework," Amazon Web Services, Inc. [Online]. Available: https://docs.aws.amazon.com/?nc2=h˙ql˙doc˙do. [Accessed: July 31, 2023].