Indy 1 - Health/Wellness App

CS 4850-03 - Spring 2025
3/27/25
Cassidy Sieverson, Olivia Moats, Tejaswini Sankar, Pamela Quintero
Professor Sharon Perry
Website: https://vive-senior-project.github.io/Vive/
GitHub: https://github.com/Vive-Senior-Project/Vive.git

Stats and Status	
LOC	1,357
Project Components	React Native, Node.js, AWS, OpenAI
Hours Estimate	154
Hours Actual	174

Table of Contents

1. Introduction

Vive is an AI-powered health and wellness application designed specifically for women, offering a personalized and adaptive approach to fitness and overall well-being. The project leverages advanced artificial intelligence to create dynamic workout plans that align with individual user profiles, considering factors such as experience level, goals, and progress. This ensures a tailored fitness experience that evolves alongside the user, fostering sustainable, long-term results.

The primary objective of Vive is to deliver a holistic wellness solution that goes beyond traditional fitness tracking. By integrating features such as mindfulness exercises, breathing techniques, and meditation, the app supports mental and emotional well-being alongside physical fitness. Additionally, Vive incorporates considerations for unique factors, such as the user's menstrual cycle, to provide a truly customized health plan.

The scope of the project includes the development of a multi-tiered fitness system to support beginners, intermediate users, and advanced athletes, all while ensuring the app dynamically adapts to new goals and milestones. Through this approach, Vive aims to redefine the user's relationship with health and fitness by promoting a balanced and adaptable lifestyle tailored to their specific needs.

2. Requirements

2.1. Project Goals

For our project, we plan on dividing our goals into two major phases. Phase one will solely focus on the mobile app alone and offer limited workout plans to avoid scope creep. By the end of phase one, our goal is to have a working minimum viable product that will be ready for presentation. As for phase two, this will focus on adding additional functionalities through the use of a watchOS app that can be paired with the mobile app. This is to allow for better personalization as the app will have access to health data from the watch. We will also include a small database of meal plans that the app can pull from to make user choices more diverse.

2.2. Definitions and Acronyms

AWS: Amazon Web Services – A cloud computing platform
EC2: Elastic Cloud Compute – A web service offered by AWS that provides virtual servers
S3: Simple Storage Service - An object storage service offered by AWS
Lambda: A serverless computing service that allows users to run code without managing servers offered by AWS
RDS: Relational Database Service – A relational database service offered by AWS.
OpenAI: A set of tools that allows developers to integrate AI models, such as GPT-4, into their applications
MVP: Minimum Viable Product
NUQ: New User Quiz

2.3. Design Constraints

Device compatibility: (Phase 2 only): The app must work across multiple platforms such as smartphones and wearable devices such as an Apple Watch. Devices must be able to sync up to provide additional health data to the mobile app.

AI driven recommendations: The app must have a set minimum amount of user data needed to provide accurate wellness plans and recommendations. Additionally, the frontend and backend must be able to support dynamic updates.

Manual recommendations: The app must receive some initial setup data through user data to better the accuracy of the AI recommendations. This information will help tailor the app's features to better suit individual preferences and needs. During the initial setup, users will be promoted to provide important data points such as:

- **Age range** to improve the AI's recommendations and give appropriate responses based on the user's age range.
- **Fitness level** to improve the AI's recommendations and give appropriate responses based on the user's level of experience
- **Workouts of interest** to further personalize the AI to the user's interests
- **Goal description** to include any other relevant information the user may want to describe. This can be anything such as dietary restrictions or mobility limitations.

2.4. Functional Requirements

Below are the functional requirements of Vive:
Hardware: The app must be compatible with iOS and Android smartphones. It should support Bluetooth connectivity for possible future expansion with syncing to fitness trackers like Apple Watches. The app should work with Apple Watches as

well (phase 2 only) to record biometric data such as steps taken, calories burned, running zone, etc.
Software: The app must integrate with OpenAI API for generating personalized meal plans and workout routines based on user input. These recommendations must be formatted properly before being sent to the front end. The backend must include error handling for API failures, such as timeouts or invalid responses. It must be integrated with AWS services like S3, EC2, and Lambda for scalability, dynamic processing, and handling the storage of user data. The back end will be hosted on AWS EC2 to handle API requests. AWS Lambda must process real-time AI recommendations and respond to user inputs instantly. It also must handle scheduled jobs, such as daily plan updates. The app must also connect with Apple HealthKit for iOS for syncing health data.
Additionally, each of the following screens and components should be available for navigation and functioning:

1.0. Welcome Page
    1.1. New User
        1.1.2. New User Quiz
    1.2. Returning User
2.0. Home Page
    2.1. Today's Plan
3.0. Wellness
    3.1. Journal
    3.1.1. New Journal Entry
    3.1.2. Previous Journal Entry
    3.2. Breathing Exercises
    3.3. Yoga & Meditation
4.0. Workouts
    4.1. Warmup
    4.2. Weight Training
    4.3. Cardio
5.0. Meal plans
    5.1 Breakfast
    5.2 Lunch
    5.3 Dinner
6.0 Achievements

2.5. Nonfunctional Requirements

Security: All user data must be encrypted, as the app handles sensitive user data such as health metrics. Passwords should be hashed, and biometric authentication should be integrated to enhance login security. AI-generated meal plans, workouts, and journal entries should be stored securely on S3. For testing purposes, EC2

should only allow incoming traffic from known IP addresses to avoid security breaches.

Performance: The app must be responsive and smooth with little downtime or delay. AI-generated responses must be generated in under 5 seconds to ensure a seamless user experience. Additionally, the app should implement error handling for issues such as API failures, lack of internet connection, or failure to retrieve data.

User Interface: The app must have an intuitive and visually appealing user interface. The home screen should provide easy navigation to the AI-generated plan for the day, meal plans, workouts, wellness, and achievements sections. Users should be able to customize their profile and personal goals by entering their preferences in the NUQ. The AI-generated recommended plans should be displayed in a visually engaging format that leads you to the appropriate tab of the aforementioned plan. The wellness section should allow users to access their journal and past entries, as well as create new journal entries. This section should also allow users to access different breathing, yoga, and meditation exercises. The workout section should allow users to access different exercises based on their interests. The meal plan section should allow users the option to view different meals. The achievements section should allow for users to see all the achievements they have received since using the app.

3. Design Analysis

The following section provides a high-level overview of our project, including assumptions and dependencies, architectural strategies, a breakdown of the system's architecture, and a detailed system design to classify and define each constraint, as well as identify their specific constraints.

3.1. Assumptions and Dependencies

3.1.1. Assumptions

- All AWS services are configured properly and remain available
- The OpenAI API remains available for generating workout and wellness plans
- The OpenAI is able to handle periods of high request volume
- Users provide reasonable input data in the NUQ

3.1.2. Dependencies

- React Native for frontend implementation
- Node.js for backend implementation

- PostgreSQL hosted via AWS RDS for data storage
- AWS EC2 for hosting the Node.js backend
- AWS S3 for storing static user content
- AWS Lambda for executing Python script containing the OpenAI API
- OpenAI API for generating personalized content

3.2. Architectural Strategies

3.2.1. Phase One

AWS: The EC2, RDS, S3, and Lambda services were chosen because everything we would need to develop a mobile app is in one place. EC2 can host our app, S3 and RDS provide reliable storage, and Lambda can trigger events when needed. Overall, it's a centralized architecture that keeps everything secure and organized. Additionally, if we chose to continue building upon this app in the future, AWS is easily scalable.
OpenAI: The OpenAI API was chosen to give users a personalized experience on Vive. Depending on how the system message is defined, we can make the API feel like a real conversation. Additionally, it saves a lot of development time, which is important for such a short window of time to develop an MVP.
React Native: For frontend development, React Native was chosen for its ability to test on real devices (using Expo Go). Additionally, it provides cross-platform development, allowing us to deploy our app on both iOS and Android devices.

3.2.2. Phase Two

Swift: Because React Native does not directly support watchOS applications, we will be using Swift to build a compatible Apple Watch app, ensuring stability and seamless integration with the Apple ecosystem. Additionally, Swift's native support for Apple frameworks such as the WatchKit will allow us to easily make an interactive app for our users.
WatchOS: watchOS is the operating system designed for the Apple Watch. It allows users to run apps, receive notifications, and most importantly, track fitness and health metrics which can then be relayed back to the mobile app to allow for a more personalized health and fitness plan.

3.3. System Architecture

Vive is built with a cloud-based, modular architecture that connects a mobile frontend to backend services on Amazon Web Services (AWS). The system

emphasizes scalability, security, and real-time performance, using serverless computing where appropriate.

The mobile app, developed with Expo and React Native, serves as the main user interface. Users can register, complete wellness quizzes, submit journal entries, and receive personalized AI-generated health plans. These interactions are sent to a backend server hosted on an AWS EC2 instance.

The EC2 server handles business logic, API routing, and connects to Amazon RDS (PostgreSQL) for managing structured data such as user profiles and journal content. For storing AI-generated plans and user-uploaded files, the backend uses Amazon S3.

To generate personalized content, the backend triggers AWS Lambda functions that call the OpenAI GPT-4 API. The results are stored in S3 and returned to the app. This architecture offloads heavy processing from EC2, reduces latency, and supports future scalability.

Figure 3.3.1 (below) demonstrates the overall interaction between the frontend and AWS services used in the Vive ecosystem.

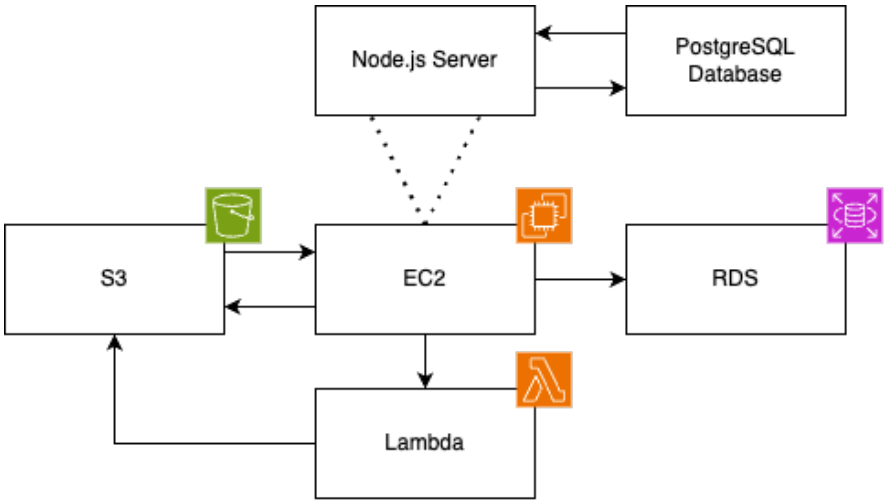


Figure 3.3.1: Vive system architecture showing interaction between the mobile app and AWS services including EC2 (backend), RDS (database), S3 (file storage), Lambda (serverless compute), and OpenAI (AI plan generation).

3.4. Detailed System Design

The following is a detailed overview of each component described in sections three and four. Each component will be classified and defined in addition to listing their constraints.

3.4.1. Classification

EC2: Cloud computing service
RDS: Relational database service
S3: Static storage service
Lambda: Event-driven processing service
OpenAI API: AI model integration
React Native: Mobile application UI

3.4.2. Definitions

EC2: Used to host the backend Node.js server that processes all app requests, handles API routing, and manages interactions with other AWS servers. The central hub for logic and user data handling.
RDS: A managed database service. In relationship to Vive, it hosts a PostgreSQL database that stores structured data such as user accounts, quiz responses, journaling entries, and goal tracking.
S3: A static storage service that stores AI-generated workout plans.
Lambda: Handles on-demand AI processing and real time event responses.
OpenAI API: Generates dynamic and personalized fitness plans for users based on their input.
React Native: Provides the user interface for the app, enabling users to input fitness goals, view their peronslaized workout plans, and track progress.

3.4.3. Constraints

EC2: Must be consistently available and configured correctly
RDS: Endpoint must be consistently available, and all configurations must be accurate
S3: Must be encrypted for security and must provide fast responses when fetching AI generated content
Lambda: Must trigger events as soon as the trigger is initialized with little delay
OpenAI API: Must provide appropriate responses based on the NUQ. Responses must also be formatted correctly
React Native: Must efficiently handle API requests and support dynamic updates based on the user's interaction with the app

4. Development

The following section discusses the development process of our project. This includes concepts, database connection, and any setup or configuration done for each component of Vive.

4.1. Concepts Outlines

NUQ: When a user initially signs up for the app, they will be prompted with a few questions to give the OpenAI model some basic information to generate a workout/wellness plan. This information will be stored in RDS where our python script will be able to fetch the user's information to generate the plan. After the plan is generated, it will be stored in S3, where it can then be relayed to the user.
Adaptive Health/Wellness Plan: As a user logs progress or changes they would like to make, Vive will adapt to the user's new wants and needs by updating information in RDS. Once daily, the user is given a 'Daily Plan' upon opening the app, which will discuss what the user should focus on for that day. Because this information is constantly updated in RDS, the plan will also constantly update.
Achievements & Goal Tracking: Users will be allowed to customize personal fitness and well-being goals (e.g., weight loss, muscle gains, or mindfulness). Vive will monitor progress from data input from the user, tracking completed workouts, meals adherence, and overall consistency. When the user achieves specific milestones, they will be rewarded with achievements, which will be maintained in the achievements table on RDS. These achievements can be displayed on the achievements page, which will be used to encourage the user.

4.2. Database Connection

To connect our database, we first created an RDS instance with the following configurations:

- DB engine version: PostgreSQL 17.2
- DB instance identifier: vivedatabase
- DB instance class: db.t4g.micro
- Storage type: General Purpose SSD (gp2)

These configurations were done to comply with the AWS Free Tier, to avoid any unexpected costs. After creating the database, we connected it to our EC2 instance with the following code:

```
const pool = new Pool({

 host: 'vivedatabase.**********.rds.amazonaws.com',

 port: 5432,
```

```
 user: '********',
 password: '***************',
 database: 'vivedatabase',
 ssl: {
  rejectUnauthorized: false
 }
});
```

This code connects our app to the RDS Postgres instance by using the endpoint that it generated and the default Postgres port number. It's important to note that all of these values are kept in our dotenv file, rather than being hardcoded. Hardcoding these values was for demonstration only. Additionally, on the EC2 instance, we create our 'users' table which uses 'id' as a primary key, and holds the user's name, email, and password, which is hashed for improved security. From here on, we have code to handle app operations such as user registration and user login. Below is our user table after some sample data has been entered:

```
 id |  name   |       email       |                           password
----+---------+-------------------+--------------------------------------------------------------
  1 | Cassidy | 123@gmail.com     | $2b$10$CxYreS4ixqnWMqalHZ2/A.F/ubVynUxtzrVwBfR8Q.AaRkjuqkWPK
  2 | Pamela  | xxx@gmail.com     | $2b$10$olt6LkrbBnQLB3S9p5ZaJOCVw8xfLRbHMcaCant2mlqvxGpSuvw2G
  3 | Olivia  | OliviaM@gmail.com | $2b$10$cIy/tV6wTbuQDmiNYmqUWe4sL23WtfRK..TVrQFOiDtCBXAxml5XK
  4 | Teja    | TSenkar@gmail.com | $2b$10$b8/NBhrrtoIpZleOuNl4Heq2zfBssO.ficU7T.gCxabf8wti4ywfm
```

4.3. Setup Process

4.3.1. AWS Setup

Security group configuration:

- Basic details
  - Security group name: vive-security-group
  - VPC: default
  - Inbound rules:

Type	Protocol	Port range	Source
Custom TCP	TCP	3000	My IP
Postggre SQL	TCP	5432	localhost
SSH	TCP	22	My IP

- Outbound rules:

Type	Protocol	Port range	Source
All traffic	All	All	localhost

EC2 instance configuration:

- Name and tags
  - Name: Vive
- Application and OS Images (Amazon Machine Image)
  - OS: Ubuntu
  - AMI: Ubuntu Server 24.04 LTS (HVM), SSD Volume Type
  - Architecture: 64-bit
- Instance type
  - Instance type: t2.micro
- Key pair (login)
  - Key pair name: vive-senior-project
- Network settings
  - Select existing security group
  - Common security groups: vive-security-group
- Configure storage
  - 1x **8** GiB **gp3**

S3 bucket configuration:

- General configuration
  - Bucket type: General purpose
  - Bucket name: *varies*
- Object Ownership
  - ACLs disabled
- Block Public Access settings for this bucket
  - Block *all* public access
- Bucket Versioning
  - Bucket Versioning: Disable
- Default encryption
  - Encryption type: Server-side encryption with Amazon S3 managed keys (SSE-S3)
  - Bucket Key: Enable

4.3.2. OpenAI Setup

To setup the OpenAI API, a key must be generated from the OpenAI website. After setting the key up and storing it in the dotenv file, we then imported the OpenAI API into our python script and initialized the system message and user message:

sys_mess = ('You are a personal health and wellness coach texting your client what you think you should do today. '
'Based on your client\'s response, you will give extremely high quality and specific recommendations. '
'You will tell them what to do, and how to do it, all while encouraging your client. '
'Always start out by greeting the client. Keep it semi-casual'
'Avoid first person pronouns and overly dramatic verbs, and put your response in paragraph format. ')

user_mess = (f"I am a {skill_level}, and I am interested in the following: "
f"Cardio: {cardio}, Weight Training: {weight_interest}, "
f"Yoga: {yoga}. I also have these additional comments I'd like to add: {add_comm}")

For our app, we are using the 4o-mini model because it handles specific tasks, such as generating a health and wellness plan. After receiving the information from the NUQ via RDS, the API will be triggered by Lambda and print out a response, which is then sent to S3.

5. Test Plan and Report

5.1. Test Plan

Our testing approach includes unit testing, integration testing and user acceptance testing to verify all features work as expected. For unit testing, we plan on using the following as input to achieve as much path coverage as possible:

- Mock A
    - Age range: Under 18
    - Fitness level: Beginner
    - Interests: Yoga, Cardio
    - Describe goals: “Get into fitness, increase stamina, increase flexibility”
- Mock B

    - Age range: 18-24
    - Fitness Level: Beginner
    - Interests: Strength Training, Cardio
    - Describe goals: “”
- Mock C
    - Age range: 25-34
    - Fitness level: Advanced
    - Interests: Strength training
    - Describe goals: “To stay fit despite a busy lifestyle”
- Mock D
    - Age range: 35+
    - Fitness level: Intermediate
    - Interests: Yoga
    - Describe goals: “To continue to stay mobile as I get older”

For integration testing, we will mostly be focusing on seeing how the database interacts with other components of the app, such as S3 and Lambda. Finally, for user acceptance testing, we will gather a small, non-technical group of people to go through the app to observe areas that work well and areas that may need improvement.

5.2. Test Report

Requirements	**Pass/Fail**
Login	Pass
New Account Creation	Pass
New User Quiz successfully submits	Pass
Today’s plan successfully triggered in Lambda	Fail
Today’s plan is uploaded to S3	Fail
Today’s plan displays successfully	Fail
Today’s plan updates on day-to-day basis	Fail
Navigation to journal is correct	Pass
New journal entry is created successfully	Pass
Navigation to yoga & meditation workouts is correct	Pass
Warmup workout is displayed	Fail
Weight training workout is displayed	Fail
Cardio workout is displayed	Fail
Breakfast meal plans are displayed and viewable	Fail

Lunch meal plans are displayed and viewable	Fail
Dinner meal plans are displayed and viewable	Fail
Navigation to achievements is correct	Pass
Workout plan stored successfully in database	Fail
EC2 instance successfully connects to server	Pass
Workout plan updates based on user interactions	Fail
UI is easy to navigate	Pass

6. Version Control

Our team used GitHub for version control throughout the project. Our repository is available on our GitHub website as well as on our cover page. All major features were implemented with the use of branches and merged into the main branch. Commits were marked to show different stages of work such as initial setup, frontend views, backend integration, and AWS deployment. This assisted in tracking bugs, rollbacks, and task splitting.

7. Summary

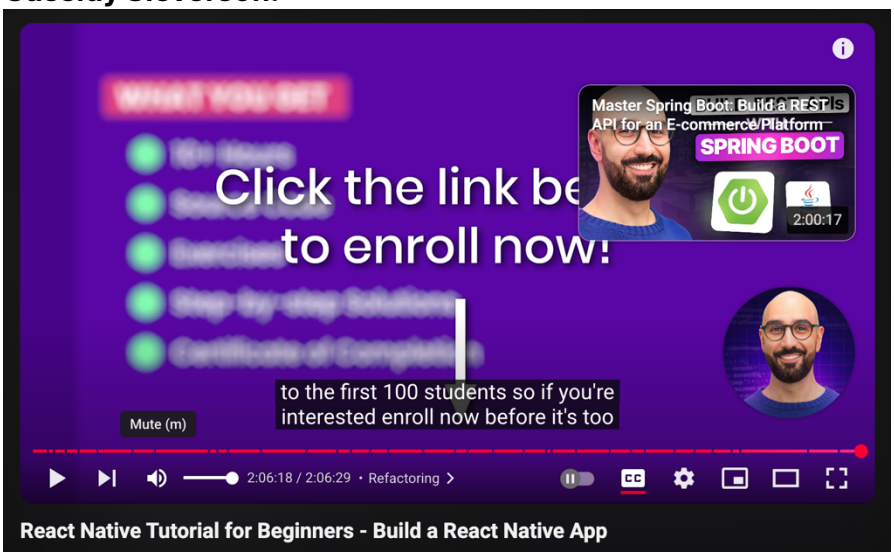
Our mobile app, Vive, is about 70% complete. The frontend using React Native currently supports user registrations, login, the NUQ, and basic navigation around the app. The backend, hosted on EC2, successfully hosts our Node.js server and allows our app to connect to our PostgreSQL database on RDS. User data and NUQ data is also stored and retrieved successfully upon request. Although we have not yet been able to implement the results of our OpenAI generated responses, the API works as intended and generates an appropriate response. AWS Lambda and S3 are a part of our intended system architecture, however, those have not yet been implemented. Lambda's intended use is to trigger the Python script containing the OpenAI API once the NUQ is submitted. Once the plan is generated, it is supposed to be stored in S3. While these components are still a work in progress, we were still able to implement critical functionalities for Vive, utilizing several Amazon Web Services, the OpenAI API, and React Native.

8. Appendix

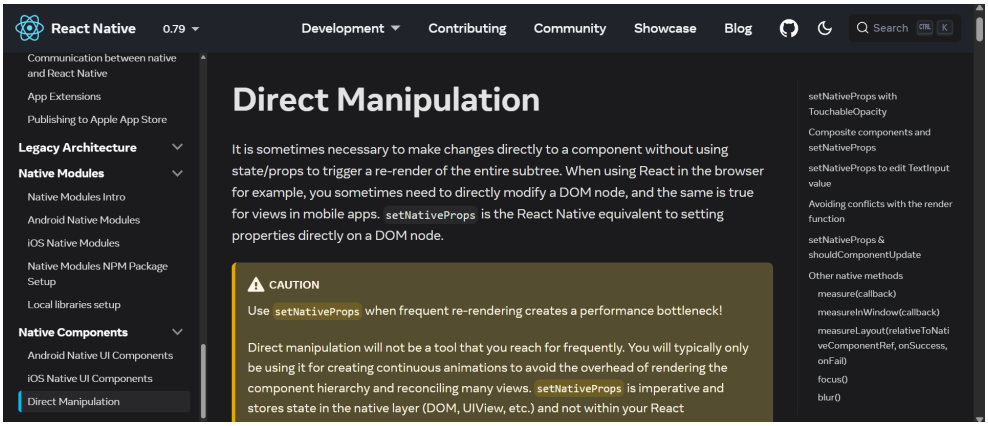
8.1. React Native Training Verification

The following screenshots show proof that every member of Indy 1 – Health/Wellness App have completed a React Native tutorial:

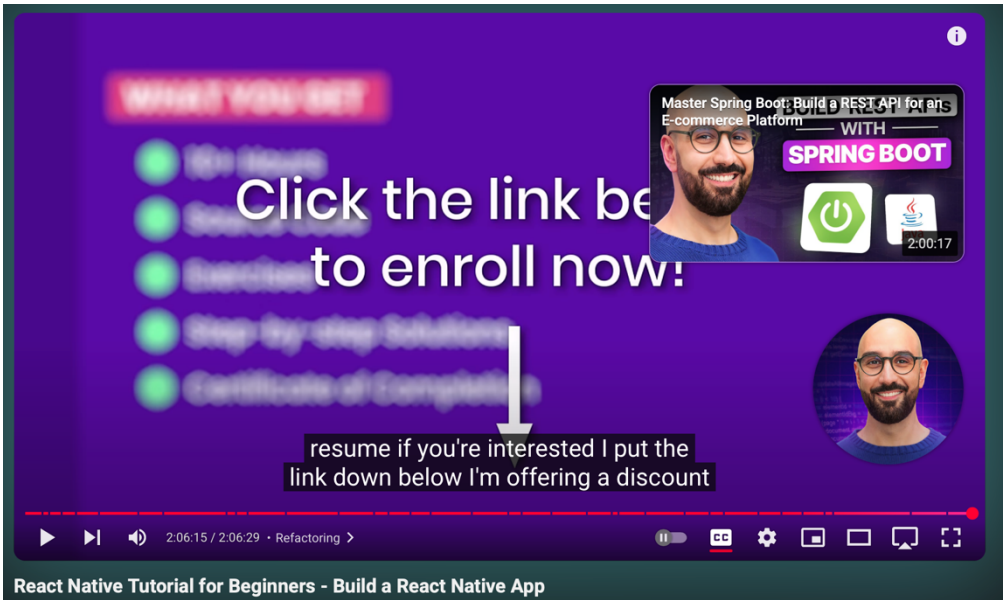
Cassidy Sieverson:



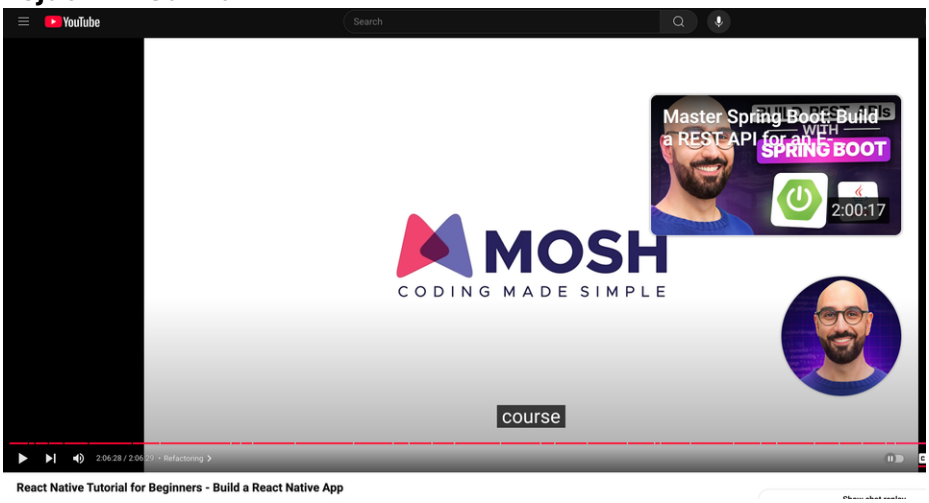
Olivia Moats:



Pamela Quintero:



Tejaswini Sankar:



8.2. Figma Screen Mockups

Below is a link to our Figma mockups:
https://www.figma.com/design/VJaAtznxgG8uaJ09z41NDc/Vive-Mockup?node-id=603-2&p=f&t=xGptB3O3o0rKVicv-0